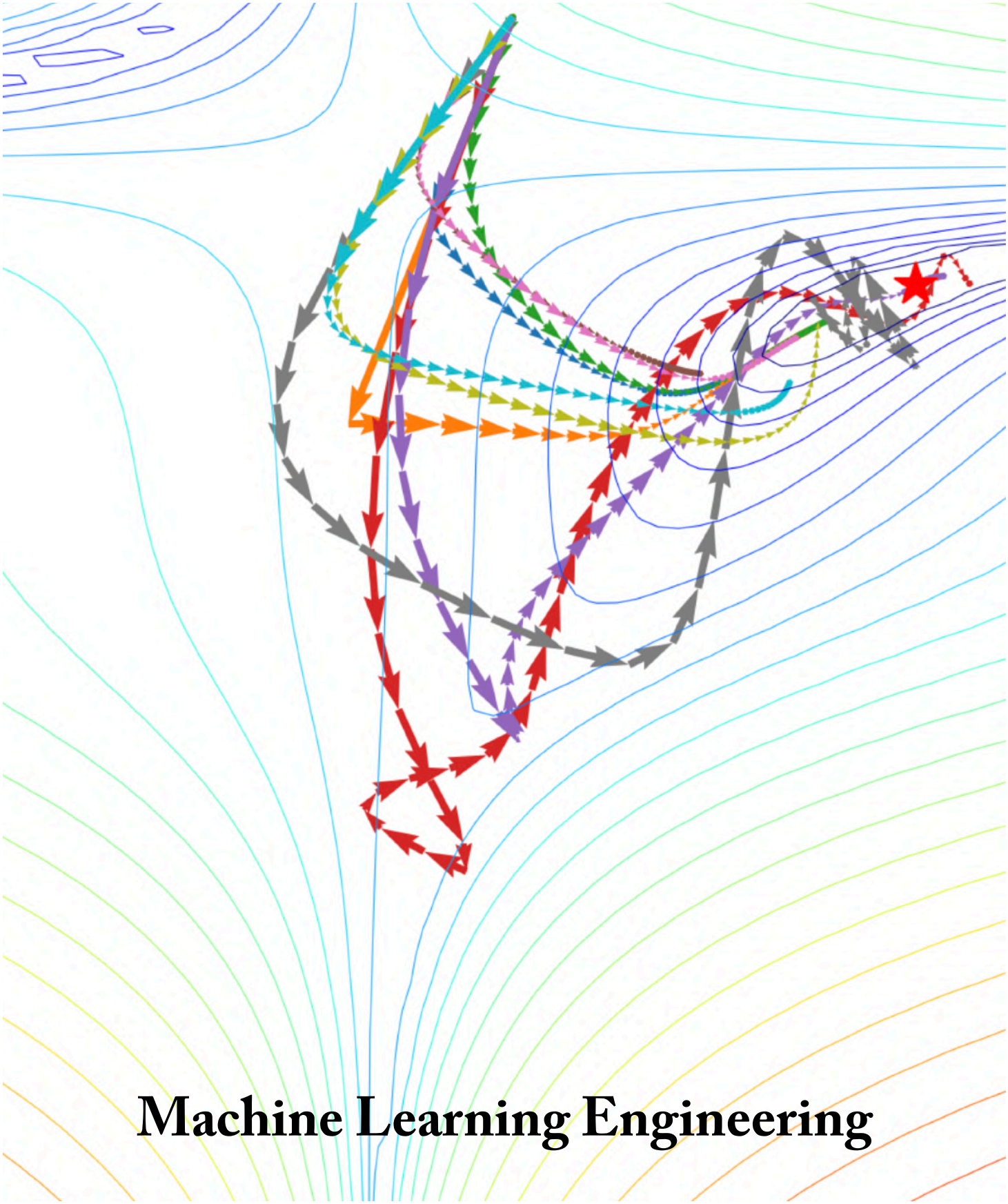


Machine Learning Engineering

Lecture 1: Introduction

A few useful things to know about machine learning

Joaquin Vanschoren

Why Machine Learning?

- 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,...

What is Machine Learning?

- Learn to perform a task, based on experience (examples) X, minimizing error $\mathcal{E}$
  - E.g. recognizing a person in an image as accurately as possible
- Often, we want to learn a function (model) f with some model parameters θ that produces the right output y

$$f_\theta(X) = y$$
$$\underset{\theta}{\operatorname{argmin}}\, \mathcal{E}(f_\theta(X))$$

- Usually part of a *much* larger system that provides the data X in the right form
  - Data needs to be collected, cleaned, normalized, checked for data biases,...

Inductive bias

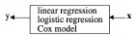
- In practice, we have to put assumptions into the model: *inductive bias* b
  - What should the model look like?
    - Mimick human brain: Neural Networks
    - Logical combination of inputs: Decision trees, Linear models
    - Remember similar examples: Nearest Neighbors, SVMs
    - Probability distribution: Bayesian models
  - User-defined settings (hyperparameters)
    - E.g. depth of tree, network architecture
  - Assuptions about the data distribution, e.g. $X \sim N(\mu, \sigma)$
- We can *transfer* knowledge from previous tasks: $f_1, f_2, f_3, \ldots \Longrightarrow f_{new}$
  - Choose the right model, hyperparameters
  - Reuse previously learned values for model parameters θ
- In short:

$$\underset{\theta, b}{\operatorname{argmin}}\, \mathcal{E}(f_{\theta,b}(X))$$

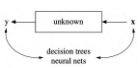
Machine learning vs Statistics

- See Breiman (2001): Statistical modelling: The two cultures
- Both aim to make predictions of natural phenomena:

- Statistics:
  - Help humans understand the world
  - Assume data is generated according to an understandable model



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



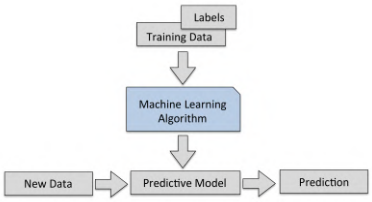
Types of machine learning

- **Supervised Learning**: learn a *model* f from *labeled data* (X, y) (ground truth)
  - Given a new input *X*, predict the right output *y*
  - Given examples of stars and galaxies, identify new objects in the sky
- **Unsupervised Learning**: explore the structure of the data (X) 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

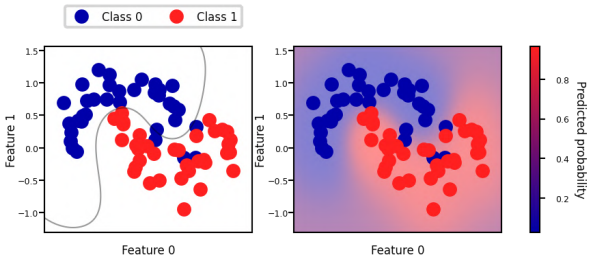


Example: Flower classification

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

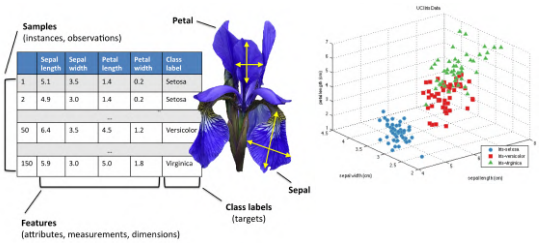
Classification

- 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



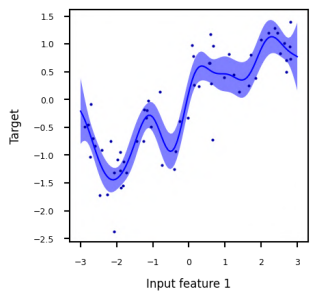
Representation: input features and labels

- 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



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.

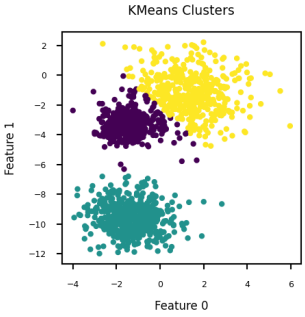


Unsupervised Machine Learning

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

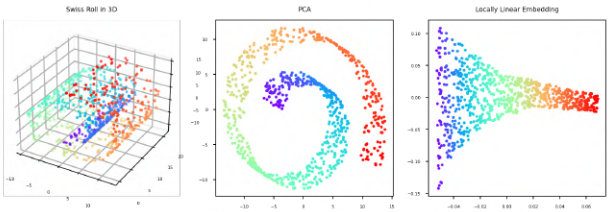
Clustering

- 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



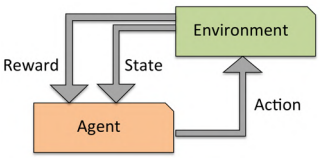
Dimensionality reduction

- 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:

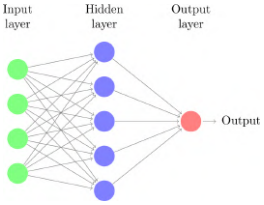
- **Representation**: A model f_θ must be represented in a formal language that the computer can handle
  - Defines the 'concepts' it can learn, the *hypothesis space*
  - E.g. a decision tree, neural network, set of annotated data points
- **Evaluation**: An *internal* way to choose one hypothesis over the other
  - Objective function, scoring function, loss function $\mathcal{L}(f_\theta)$
  - 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
  - Start with initial set of model parameters, gradually refine them
  - Many methods, differing in speed of learning, number of optima,...

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

Neural networks: representation

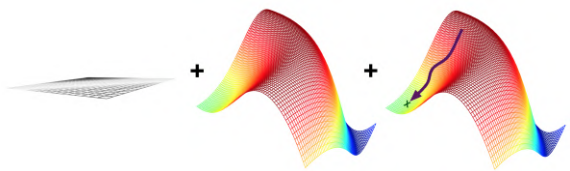
Let's take neural networks as an example

- Representation: (layered) neural network
  - Each connection has a *weight* θ_i (a.k.a. model parameters)
  - Each node receives weighted inputs, emits new value
  - Model f returns the output of the last layer
- The architecture, number/type of neurons, etc. are fixed
  - We call these *hyperparameters* (set by user, fixed during training)



Neural networks: evaluation and optimization

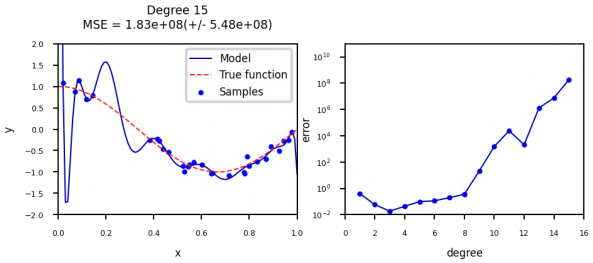
- Representation: Given the structure, the model is represented by its parameters
  - Imagine a mini-net with two weights (θ_0, θ_1): a 2-dimensional search space
- Evaluation: A *loss function* $\mathcal{L}(f_\theta)$ computes how good the predictions are
  - *Estimated* on a set of training data with 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
  - E.g. Gradient descent: $\theta_i^{new} = \theta_i + \frac{\mathcal{L}(f_\theta)}{\partial \theta_i}$



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* you 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

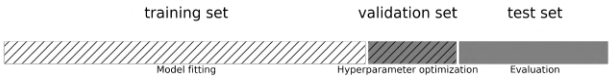


Model selection

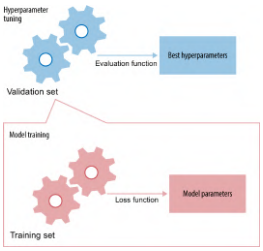
- Next to the (internal) loss function, we need an (external) evaluation function
  - Feedback signal: are we actually learning the right thing?
    - Are we under/overfitting?
  - Carefully choose to fit the application.
  - Needed to select between models (and hyperparameter settings)

© XKCD

- Data needs to be split into *training* and *test* sets
  - Optimize model parameters on the training set, evaluate on independent test set
- Avoid *data leakage*:
  - Never optimize hyperparameter settings on the test data
  - Never choose preprocessing techniques based on the test data
- To optimize hyperparameters and preprocessing as well, set aside part of training set as a *validation* set
  - Keep test set hidden during *all* training



- For a given hyperparameter setting, learn the model parameters on training set
  - Minize the loss
- Evaluate the trained model on the validation set
  - Tune the hyperparameters to maximize a certain metric (e.g. accuracy)

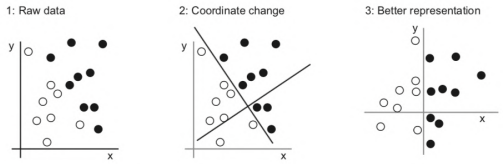


Only generalization counts!

- Never evaluate your final models on the training data, except for:
  - Tracking whether the optimizer converges (learning curves)
  - Diagnosing under/overfitting:
    - Low training and test score: underfitting
    - High training score, low test score: overfitting
- Always keep a completely independent test set
- On small datasets, use multiple train-test splits to avoid sampling bias
  - You could sample an 'easy' test set by accident
  - E.g. Use cross-validation (see later)

Better data representations, better models

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



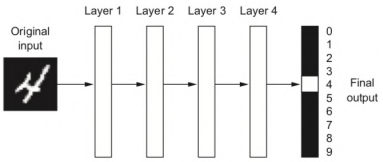
Feature engineering

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

Build prototypes early-on

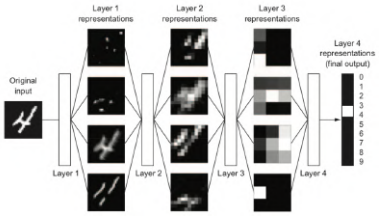
Learning data transformations end-to-end

- 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



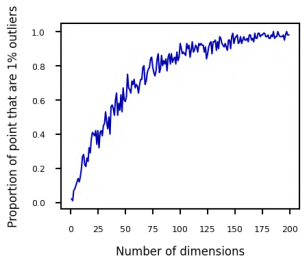
Example: digit classification

- 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

- Just adding lots of features and letting the model figure it out doesn't work
- Our assumptions (inductive biases) often fail in high dimensions:
    - Randomly sample points in an n-dimensional space (e.g. a unit hypercube)
    - Almost all points become outliers at the edge of the space
    - Distances between any two points will become almost identical



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)

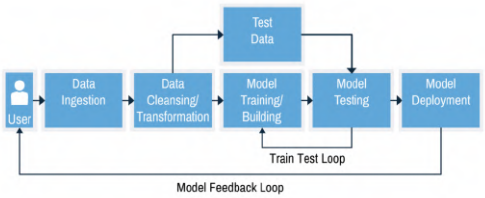
- More data reduces the chance of overfitting
- Less sparse data reduces the curse of dimensionality
- *Non-parametric* models: number of model parameters grows with amount of data
  - Tree-based techniques, k-Nearest neighbors, SVM,...
  - They can learn any model given sufficient data (but can get 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, which we will cover in upcoming lectures:

- Preprocessing: Raw data is rarely ideal for learning
  - 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
  - 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
- 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



Summary

- Learning algorithms contain 3 components:
  - Representation: a model f that maps input data X to desired output y
    - Contains model parameters θ that can be made to fit the data X
  - Loss function $\mathcal{L}(f_\theta(X))$: measures how well the model fits the data
  - Optimization technique to find the optimal θ: $\underset{\theta}{\operatorname{argmin}}\ \mathcal{L}(f_\theta(X))$
- Select the right model, then fit it to the data to minimize a task-specific error $\mathcal{E}$
  - Inductive bias b: assumptions about model and hyperparameters

    $\underset{\theta,b}{\operatorname{argmin}}\ \mathcal{E}(f_{\theta,b}(X))$
- Overfitting: model fits the training data well but not new (test) data
  - Split the data into (multiple) train-validation-test splits
  - Regularization: tune hyperparameters (on validation set) to simplify model
  - Gather more data, or build ensembles of models
- Machine learning *pipelines*: preprocessing + learning + deployment

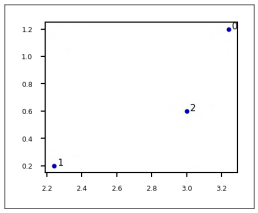
Lecture 2: Linear models

Basics of modeling, optimization, and regularization
Joaquin Vanschoren

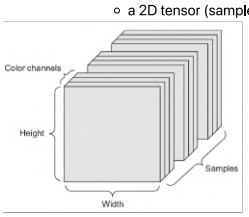
Notation and Definitions

- 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:

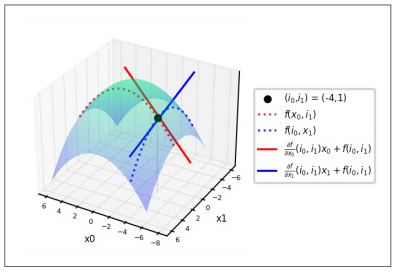
$$\sum_{i=0}^{p} = x_0 + x_1 + \ldots + x_p \quad \prod_{i=0}^{p} = x_0 \cdot x_1 \cdot \ldots \cdot x_p$$

- 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

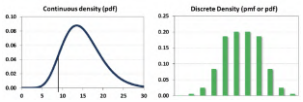
- A *derivative* f' of a function f describes how fast f grows or decreases
- The process of finding a derivative is called differentiation
  - Derivatives for basic functions are known
  - For non-basic functions we use the chain rule: $F(x) = f(g(x)) \rightarrow F'(x) = f'(g(x))g'(x)$
- A function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

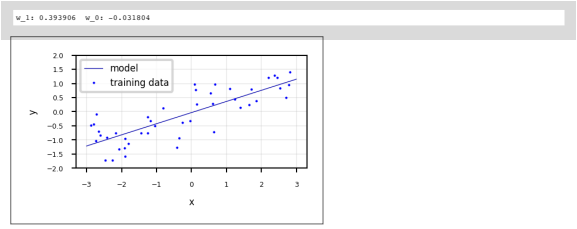
$$\underset{\mathbf{w}}{\operatorname{argmin}}\, \mathcal{L}(f_{\mathbf{w}}(X))$$

- 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 $N(0, \sigma)$

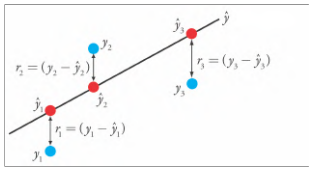
$$\hat{y} = \mathbf{w}\mathbf{x} + w_0 = \sum_{i=1}^{p} w_i \cdot x_i + w_0 = w_1 \cdot x_1 + w_2 \cdot x_2 + \ldots + w_p \cdot x_p + w_0$$



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:

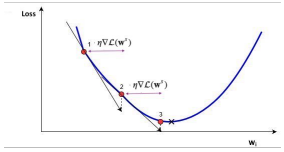
$$w^* = (X^T X)^{-1} X^T Y$$

  - 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

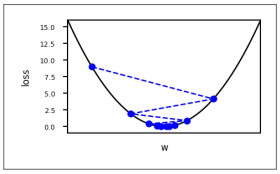
Gradient Descent

- 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):

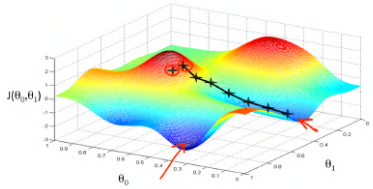
$$\mathbf{w}^{s+1} = \mathbf{w}^s - \eta\nabla\mathcal{L}(\mathbf{w}^s)$$



- 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:



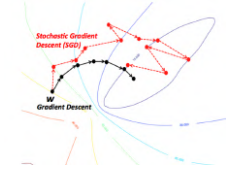
- You can get stuck in local minima (if the loss is not fully convex)
  - If you have many model parameters, this is less likely
  - You always find a way down in some direction
  - Models with many parameters typically find good local minima

- 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): 30.934563673640078
```

```
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:

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

- 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

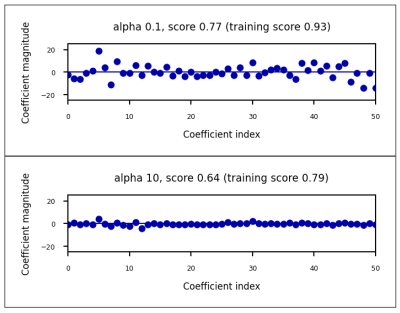
In practice

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

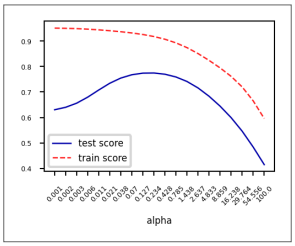
```
Weights (coefficients): [-1.414 -1.557 -1.465 -0.127 -0.079  8.332  0.255 -4.941  3.899 -1.059
 -1.584  1.051 -4.012  0.334  0.004 -0.849  0.745 -1.431 -1.63  -1.405
 -0.045 -1.746 -1.467 -1.332 -1.692 -0.506  2.622 -2.092  0.195 -0.275
  5.113 -1.671 -0.098  0.634 -0.61   0.04  -1.277 -2.913  3.395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

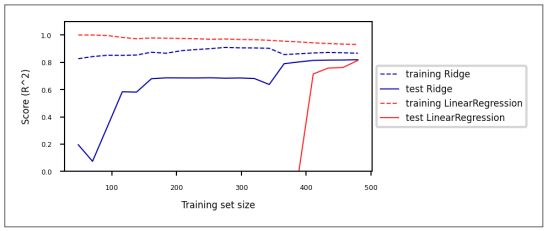


- When we plot the train and test scores for every α value, we see a sweet spot around $\alpha = 0.2$
  - Models with smaller α are overfitting
  - Models with larger α are underfitting



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)

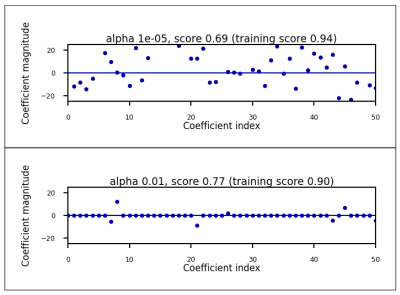
- Adds a different penalty term to the least squares sum:

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

- 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*

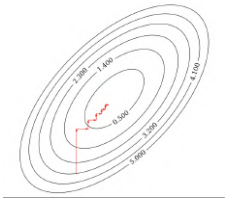
Analyze what happens to the weights:

- 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

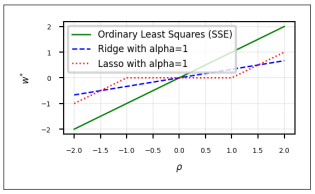
$$\begin{aligned} \partial_{w_i}\mathcal{L}_{Lasso}(w_i) &= \partial_{w_i}\mathcal{L}_{SSE}(w_i) + \partial_{w_i}\alpha|w_i| \\ 0 &= (w_i - \rho_i) + \alpha \cdot \partial_{w_i}|w_i| \\ w_i &= \rho_i - \alpha \cdot \partial_{w_i}|w_i| \end{aligned}$$

  - 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

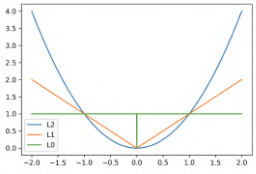
$$w_i^* = S(\rho_i, \alpha) = \begin{cases} \rho_i + \alpha, & \rho_i < -\alpha \\ 0, & -\alpha < \rho_i < \alpha \\ \rho_i - \alpha, & \rho_i > \alpha \end{cases}$$

  - 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



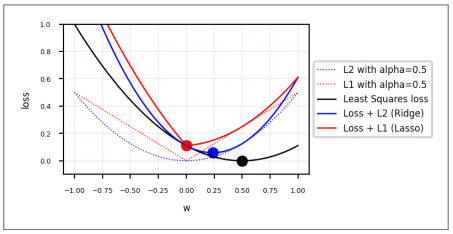
$$\ell_2(w) = \sqrt{\sum_i w_i^2}$$

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

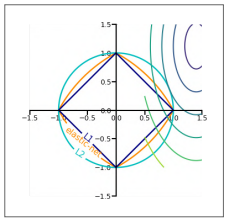
$$\ell_0(w) = \sum_i 1_{w_i \neq 0}$$

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

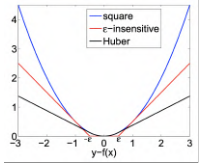
- Adds both L1 and L2 regularization:

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

- 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)

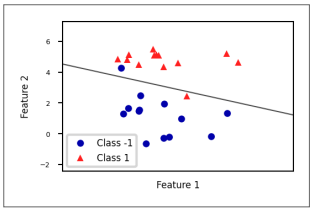
Other loss functions for regression

- 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



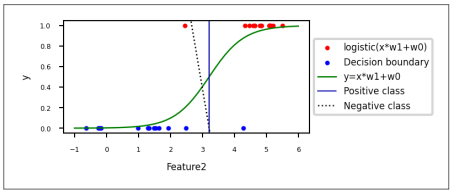
- There are many algorithms for linear classification, differing in loss function, regularization techniques, and optimization method
- Most common techniques:
  - Convert target classes {neg,pos} to {0,1} and treat as a regression task
    - Logistic regression (Log loss)
    - Ridge Classification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

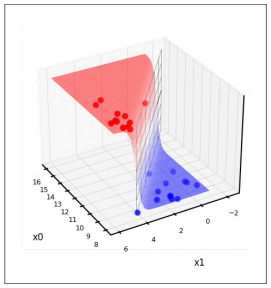
- Aims to predict the *probability* that a point belongs to the positive class
- Converts target values {negative (blue), positive (red)} to {0,1}
- Fits a *logistic* (or *sigmoid* or *S* curve) function through these points
  - Maps (-Inf,Inf) to a probability [0,1]

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

- 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)

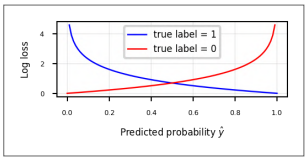


Loss function: Cross-entropy

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



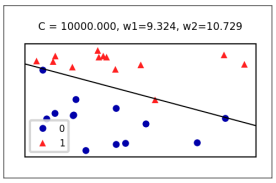
Optimization methods (solvers) for cross-entropy loss

- 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 $H = [\frac{\partial^2 \mathcal{L}}{\partial x_i \partial x_j}]$: $\mathbf{w}^{s+1} = \mathbf{w}^s - \eta H^{-1}(\mathbf{w}^s) \nabla \mathcal{L}(\mathbf{w}^s)$
  - 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

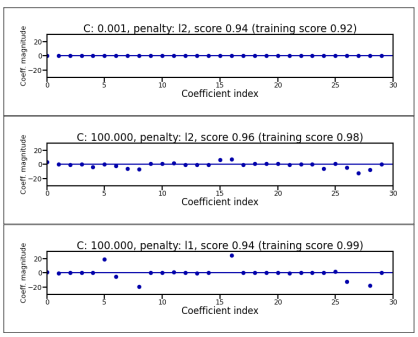
- Logistic regression can also be found in `sklearn.linear_model`.
  - `C` hyperparameter is the *inverse* regularization strength: $C = \alpha^{-1}$
  - `penalty`: type of regularization: L1, L2 (default), Elastic-Net, or None
  - `solver`: newton-cg, lbfgs (default), liblinear, sag, saga
- Increasing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

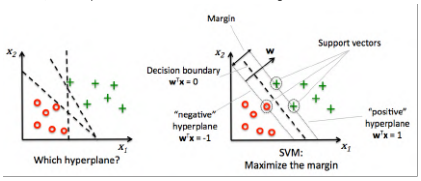
- Instead of log loss, we can also use ridge loss:

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

- 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

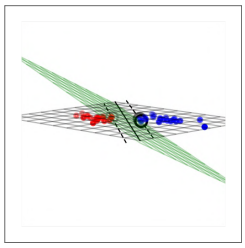
Support vector machines

- 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*



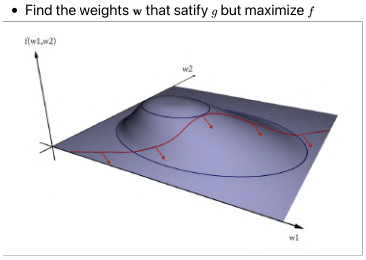
Solving SVMs with Lagrange Multipliers

- Imagine a hyperplane (green) $y = \sum_i^p \mathbf{w}_i * \mathbf{x}_i + w_0$ 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):

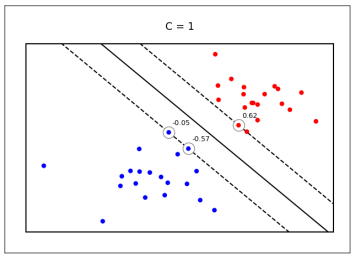
$$\mathcal{L}_{Dual} = \sum_{i=1}^{l} a_i - \frac{1}{2}\sum_{i,j=1}^{l} a_i a_j y_i y_j (\mathbf{x_i}\mathbf{x_j})$$

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 $\mathbf{x_i}\mathbf{x_j}$ 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

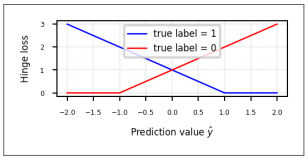
- If the data is not linearly separable, (hard) margin maximization becomes meaningless
- Relax the contraint by allowing an error ξ_i: $y_i(\mathbf{wx_i} + w_0) \geq 1 - \xi_i$
- Or (since $\xi_i \geq 0$):

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

- 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

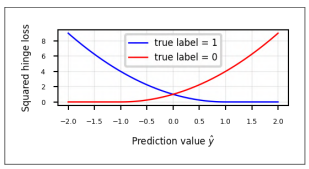


Least Squares SVMs

- 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

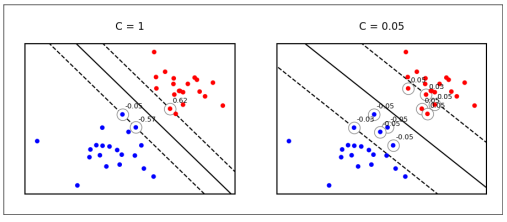
$$\mathcal{L}(\mathbf{w}) = ||\mathbf{w}||^2 + C \sum_i^n \xi_i^2$$

- 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

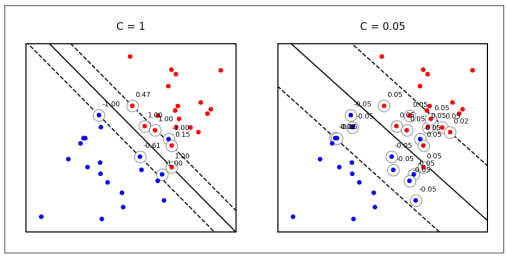


Effect of regularization on margin and support vectors

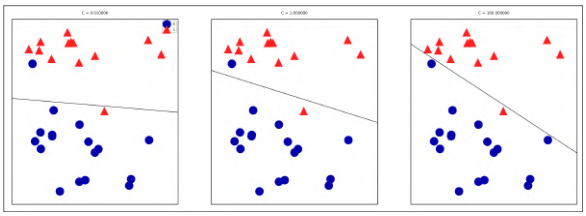
- SVM's Hinge loss acts like L1 regularization, yields sparse models
- C is the *inverse* regularization strength (inverse of α in Lasso)
  - Larger C: fewer support vectors, smaller margin, more overfitting
  - Smaller C: more support vectors, wider margin, less overfitting
- Needs to be tuned carefully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

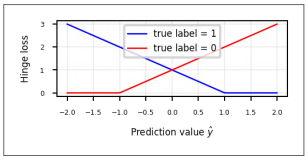
```
Support vectors:
[[-1.021  0.241]
 [-0.467 -0.531]
 [ 0.951  0.58 ]]
Coefficients:
[[-0.048 -0.569  0.617]]
```

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:

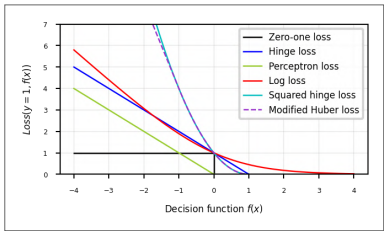
$$\mathcal{L}_{Hinge}(\mathbf{w}) = max(0, 1 - y_i(\mathbf{wx_i} + w_0))$$
$$\frac{\partial \mathcal{L}_{Hinge}}{\partial w_i} = \begin{cases} -y_i x_i & y_i(\mathbf{wx_i} + w_0) < 1 \\ 0 & \text{otherwise} \end{cases}$$

- Can be solved with (stochastic) gradient descent



Generalized SVMs

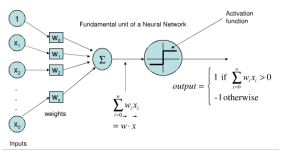
- Because the derivative of hinge loss is undefined at y=1, smoothed versions are often used:
  - Squared hinge loss: yields *least squares SVM*
    - Equivalent to Ridge classification (with different solver)
  - Modified Huber loss: squared hinge, but linear after -1. Robust against outliers
- Log 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

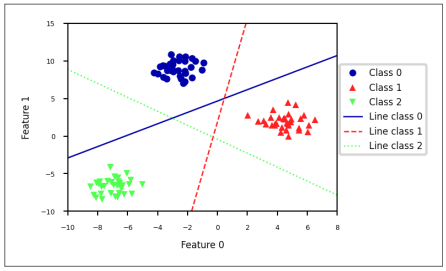
$$\mathcal{L}_{Perceptron} = max(0, -y_i(\mathbf{wx_i} + w_0))$$
$$\frac{\partial \mathcal{L}_{Perceptron}}{\partial w_i} = \begin{cases} -y_i x_i & y_i(\mathbf{wx_i} + w_0) < 0 \\ 0 & \text{otherwise} \end{cases}$$



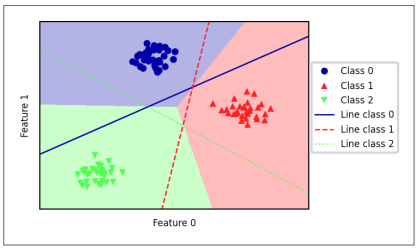
Linear Models for multiclass classification

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

Lecture 3: Kernelization

Making linear models non-linear

Joaquin Vanschoren

Feature Maps

- Linear models: $\hat{y} = \mathbf{w}\mathbf{x} + w_0 = \sum_{i=1}^{p} w_i x_i + w_0 = w_0 + w_1 x_1 + \ldots + w_p x_p$
- When we cannot fit the data well, we can add non-linear transformations of the features
- Feature map (or *basis expansion*) $\phi : X \rightarrow \mathbb{R}^d$

$$y = \mathbf{w}^T\mathbf{x} \rightarrow y = \mathbf{w}^T\phi(\mathbf{x})$$

- E.g. Polynomial feature map: all polynomials up to degree d and all products

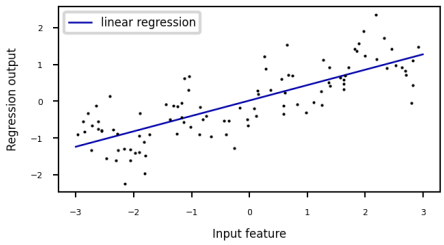
$$[1, x_1, \ldots, x_p] \overset{\phi}{\rightarrow} [1, x_1, \ldots, x_p, x_1^2, \ldots, x_p^2, \ldots, x_p^d, x_1x_2, \ldots, x_{p-1}x_p]$$

- Example with $p = 1, d = 3$:

$$y = w_0 + w_1 x_1 \overset{\phi}{\rightarrow} y = w_0 + w_1 x_1 + w_2 x_1^2 + w_3 x_1^3$$

Ridge regression example

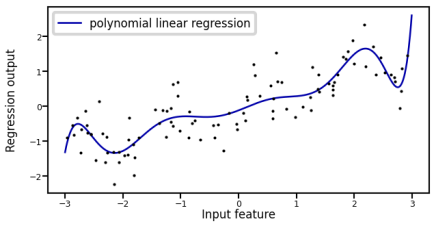
```
Weights: [0.418]
```



- Add all polynomials x^d up to degree 10 and fit again:
  - e.g. use sklearn `PolynomialFeatures`

	x0	x0^2	x0^3	x0^4	x0^5	x0^6	x0^7	x0^8	
0	-0.752759	0.566647	-0.426548	0.321088	-0.241702	0.181944	-0.136960	0.103098	
1	2.704286	7.313162	19.776880	53.482337	144.631526	391.124988	1057.713767	2860.360362	77
2	1.391964	1.937563	2.697017	3.754150	5.225640	7.273901	10.125005	14.093639	
3	0.591951	0.350406	0.207423	0.122784	0.072682	0.043024	0.025468	0.015076	
4	-2.063888	4.259634	-8.791409	18.144485	-37.448187	77.288869	-159.515582	329.222321	-6

```
Weights: [ 0.643  0.297 -0.69  -0.264  0.41   0.096 -0.076 -0.014  0.004  0.001]
```



How expensive is this?

- You may need MANY dimensions to fit the data
  - Memory and computational cost
  - More weights to learn, more likely overfitting
- Ridge has a closed-form solution which we can compute with linear algebra:

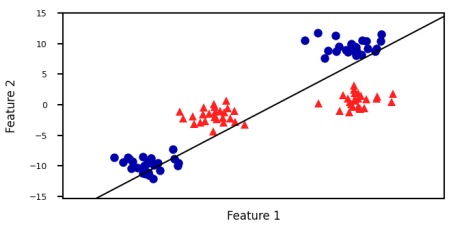
$$w^* = (X^TX + \alpha I)^{-1}X^TY$$

- Since X has n rows (examples), and d columns (features), X^TX has dimensionality dxd
- Hence Ridge is quadratic in the number of features, $\mathcal{O}(d^2n)$
- After the feature map Φ, we get

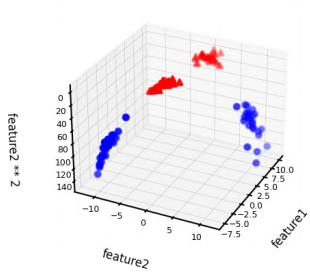
$$w^* = (\Phi(X)^T\Phi(X) + \alpha I)^{-1}\Phi(X)^TY$$

- Since Φ increases d a lot, $\Phi(X)^T\Phi(X)$ becomes *huge*

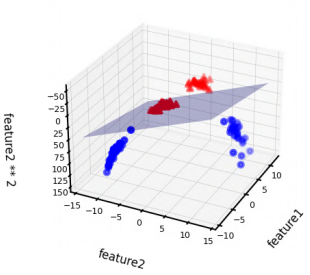
Linear SVM example (classification)



We can add a new feature by taking the squares of feature1 values

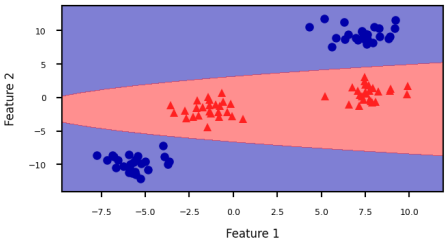


Now we can fit a linear model



As a function of the original features, the decision boundary is now a polynomial as well

$$y = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_2^2 > 0$$



The kernel trick

- Computations in explicit, high-dimensional feature maps are *expensive*
- For *some* feature maps, we can, however, compute *distances* between *points* cheaply
  - Without explicitly constructing the high-dimensional space at all
- Example: *quadratic* feature map for $\mathbf{x} = (x_1, \ldots, x_p)$:

$$\Phi(\mathbf{x}) = (x_1, \ldots, x_p, x_1^2, \ldots, x_p^2, \sqrt{2}x_1x_2, \ldots, \sqrt{2}x_{p-1}x_p)$$

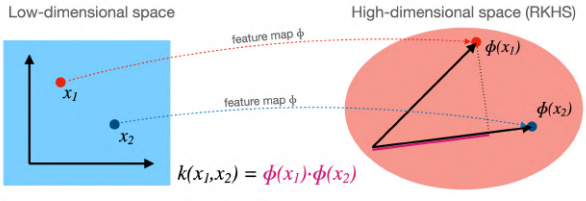
- A *kernel function* exists for this feature map to compute dot products

$$k_{quad}(\mathbf{x_i}, \mathbf{x_j}) = \Phi(\mathbf{x_i}) \cdot \Phi(\mathbf{x_j}) = \mathbf{x_i} \cdot \mathbf{x_j} + (\mathbf{x_i} \cdot \mathbf{x_j})^2$$

- Skip computation of $\Phi(x_i)$ and $\Phi(x_j)$ and compute $k(x_i, x_j)$ directly

Kernelization

- Kernel k corresponding to a feature map Φ: $k(\mathbf{x_i}, \mathbf{x_j}) = \Phi(\mathbf{x_i}) \cdot \Phi(\mathbf{x_j})$
- Computes dot product between x_i, x_j in a high-dimensional space $\mathcal{H}$
  - Kernels are sometimes called *generalized dot products*
  - $\mathcal{H}$ is called the *reproducing kernel Hilbert space* (RKHS)
- The dot product is a measure of the *similarity* between x_i, x_j
  - Hence, a kernel can be seen as a similarity measure for high-dimensional spaces
- If we have a loss function based on dot products $\mathbf{x_i} \cdot \mathbf{x_j}$ it can be *kernelized*
  - Simply replace the dot products with $k(\mathbf{x_i}, \mathbf{x_j})$



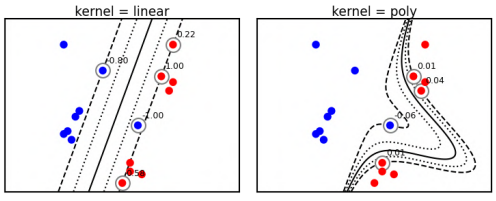
Example: SVMs

- Linear SVMs (dual form, for l support vectors with dual coefficients a_i and classes y_i):

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

- Kernelized SVM, using any existing kernel k we want:

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



Which kernels exist?

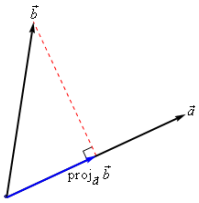
- A (Mercer) kernel is any function $k : X \times X \to \mathbb{R}$ with these properties:
  - Symmetry: $k(\mathbf{x_1}, \mathbf{x_2}) = k(\mathbf{x_2}, \mathbf{x_1}) \ \ \forall \mathbf{x_1}, \mathbf{x_2} \in X$
  - Positive definite: the kernel matrix K is positive semi-definite
    - Intuitively, $k(\mathbf{x_1}, \mathbf{x_2}) \geq 0$
- The kernel matrix (or Gram matrix) for n points of $x_1, \ldots, x_n \in X$ is defined as:

$$K = XX^T = \begin{bmatrix} k(\mathbf{x_1}, \mathbf{x_1}) & \ldots & k(\mathbf{x_1}, \mathbf{x_n}) \\ \vdots & \ddots & \vdots \\ k(\mathbf{x_n}, \mathbf{x_1}) & \ldots & k(\mathbf{x_n}, \mathbf{x_n}) \end{bmatrix}$$

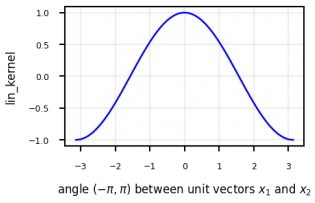
- Once computed ($\mathcal{O}(n^2)$), simply lookup $k(\mathbf{x_1}, \mathbf{x_2})$ for any two points
- In practice, you can either supply a kernel function or precompute the kernel matrix

Linear kernel

- Input space is same as output space: $X = \mathcal{H} = \mathbb{R}^d$
- Feature map $\Phi(\mathbf{x}) = \mathbf{x}$
- Kernel: $k_{linear}(\mathbf{x_i}, \mathbf{x_j}) = \mathbf{x_i} \cdot \mathbf{x_j}$
- Geometrically, the dot product is the *projection* of $\mathbf{x_j}$ on hyperplane defined by $\mathbf{x_i}$
  - Becomes larger if $\mathbf{x_i}$ and $\mathbf{x_j}$ are in the same 'direction'



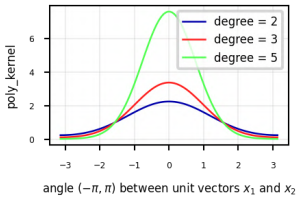
- Linear kernel between point (0,1) and another unit vector an angle a (in radians)
  - Points with similar angles are deemed similar



Polynomial kernel

- If k_1, k_2 are kernels, then $\lambda . k_1$ ($\lambda \geq 0$), $k_1 + k_2$, and $k_1 . k_2$ are also kernels
- The **polynomial kernel** (for degree $d \in \mathbb{N}$) reproduces the polynomial feature map
  - γ is a scaling hyperparameter (default $\frac{1}{p}$)
  - c_0 is a hyperparameter (default 1) to trade off influence of higher-order terms

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



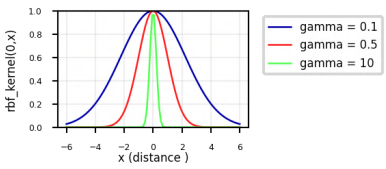
RBF (Gaussian) kernel

- The *Radial Basis Function* (RBF) feature map builds the Taylor series expansion of e^x

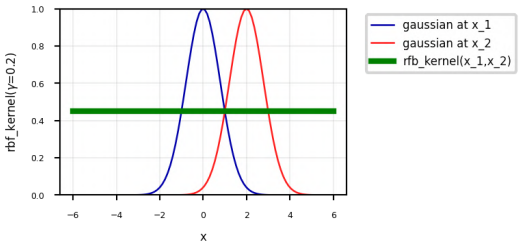
$$\Phi(x) = e^{-x^2/2\gamma^2}\left[1, \sqrt{\frac{1}{1!\gamma^2}}x, \sqrt{\frac{1}{2!\gamma^4}}x^2, \sqrt{\frac{1}{3!\gamma^6}}x^3, \ldots\right]^T$$

- RBF (or *Gaussian*) kernel with *kernel width* $\gamma \geq 0$:

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

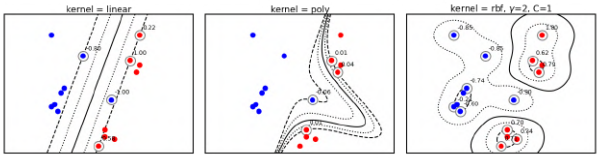


- The RBF kernel $k_{RBF}(\mathbf{x_1}, \mathbf{x_2}) = exp(-\gamma||\mathbf{x_1} - \mathbf{x_2}||^2)$ does not use a dot product
  - It only considers the distance between $\mathbf{x_1}$ and $\mathbf{x_2}$
  - It's a *local kernel* : every data point only influences data points nearby
    - linear and polynomial kernels are *global* : every point affects the whole space
  - Similarity depends on closeness of points and kernel width
    - value goes up for closer points and wider kernels (larger overlap)



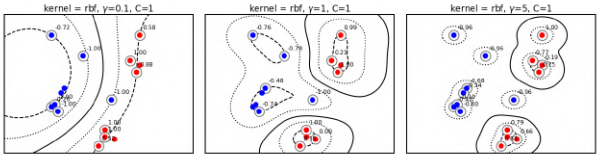
Kernelized SVMs in practice

- You can use SVMs with any kernel to learn non-linear decision boundaries



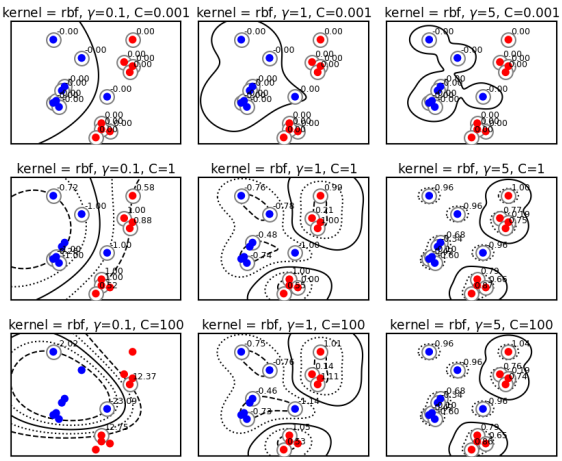
SVM with RBF kernel

- Every support vector *locally* influences predictions, according to kernel width (γ)
- The prediction for test point $\mathbf{u}$: sum of the remaining influence of each support vector
  - $f(x) = \sum_{i=1}^{l} a_i y_i k(\mathbf{x_i}, \mathbf{u})$

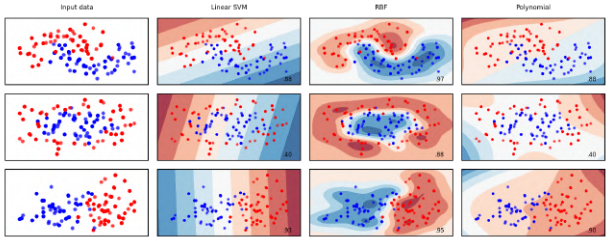


Tuning RBF SVMs

- gamma (kernel width)
  - high values cause narrow Gaussians, more support vectors, overfitting
  - low values cause wide Gaussians, underfitting
- C (cost of margin violations)
  - high values punish margin violations, cause narrow margins, overfitting
  - low values cause wider margins, more support vectors, underfitting



Kernel overview



SVMs in practice

- C and gamma *always* need to be tuned
  - Interacting regularizers. Find a good C, then finetune gamma
- SVMs expect all features to be approximately on the same scale
  - Data needs to be scaled beforehand
- Allow to learn complex decision boundaries, even with few features
  - Work well on both low- and high dimensional data
  - Especially good at small, high-dimensional data
- Hard to inspect, although support vectors can be inspected
- In sklearn, you can use `SVC` for classification with a range of kernels
  - `SVR` for regression

Other kernels

- There are many more possible kernels
- If no kernel function exists, we can still *precompute* the kernel matrix
  - All you need is some similarity measure, and you can use SVMs
- Text kernels:
  - Word kernels: build a bag-of-words representation of the text (e.g. TFIDF)
    - Kernel is the inner product between these vectors
  - Subsequence kernels: sequences are similar if they share many sub-sequences
    - Build a kernel matrix based on pairwise similarities
- Graph kernels: Same idea (e.g. find common subgraphs to measure similarity)
- These days, deep learning embeddings are more frequently used

The Representer Theorem

- We can kernelize many other loss functions as well
- The *Representer Theorem* states that if we have a loss function $\mathcal{L}'$ with
  - $\mathcal{L}$ an *arbitrary* loss function using some function f of the inputs $\mathbf{x}$
  - $\mathcal{R}$ a (non-decreasing) regularization score (e.g. L1 or L2) and constant λ

$$\mathcal{L}'(\mathbf{w}) = \mathcal{L}(y, f(\mathbf{x})) + \lambda\mathcal{R}(||\mathbf{w}||)$$

- Then the weights $\mathbf{w}$ can be described as a linear combination of the training samples:

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

- Note that this is exactly what we found for SVMs: $\mathbf{w} = \sum_{i=1}^{l} a_i y_i \mathbf{x_i}$
- Hence, we can also kernelize Ridge regression, Logistic regression, Perceptrons, Support Vector Regression, ...

Kernelized Ridge regression

- The linear Ridge regression loss (with $\mathbf{x_0} = 1$):

$$\mathcal{L}_{Ridge}(\mathbf{w}) = \sum_{i=0}^{n}(y_i - \mathbf{w}\mathbf{x_i})^2 + \lambda\|w\|^2$$

- Filling in $\mathbf{w} = \sum_{i=1}^{n} \alpha_i y_i \mathbf{x_i}$ yields the dual formulation:

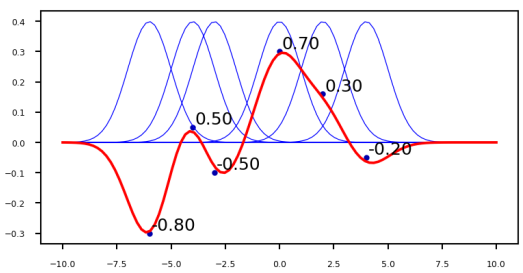
$$\mathcal{L}_{Ridge}(\mathbf{w}) = \sum_{i=1}^{n}(y_i - \sum_{j=1}^{n} \alpha_j y_j \mathbf{x_i}\mathbf{x_j})^2 + \lambda \sum_{i=1}^{n}\sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \mathbf{x_i}\mathbf{x_j}$$

- Generalize $\mathbf{x_i} \cdot \mathbf{x_j}$ to $k(\mathbf{x_i}, \mathbf{x_j})$

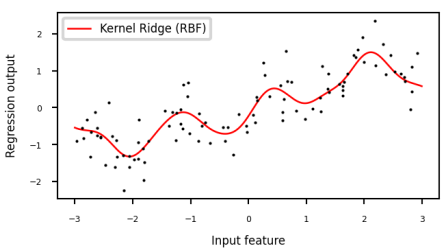
$$\mathcal{L}_{KernelRidge}(\alpha, k) = \sum_{i=1}^{n}(y_i - \sum_{j=1}^{n} \alpha_j y_j k(\mathbf{x_i}, \mathbf{x_n}))^2 + \lambda \sum_{i=1}^{n}\sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j k(\mathbf{x_i}, \mathbf{x_j})$$

Example of kernelized Ridge

- Prediction (red) is now a linear combination of kernels (blue): $y = \sum_{j=1}^{n} \alpha_j y_j k(\mathbf{x}, \mathbf{x_j})$
- We learn a dual coefficient for each point



- Fitting our regression data with `KernelRidge`

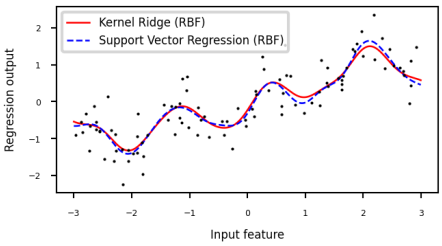


Other kernelized methods

- Same procedure can be done for logistic regression
- For perceptrons, $\alpha \rightarrow \alpha + 1$ after every misclassification

$$\mathcal{L}_{DualPerceptron}(x_i, k) = max(0, y_i \sum_{j=1}^{n} \alpha_j y_j k(\mathbf{x_j}, \mathbf{x_i}))$$

- Support Vector Regression behaves similarly to Kernel Ridge



Summary

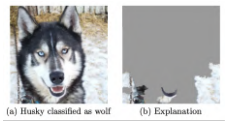
- Feature maps $\Phi(x)$ transform features to create a higher-dimensional space
  - Allows learning non-linear functions or boundaries, but very expensive/slow
- For some $\Phi(x)$, we can compute dot products without constructing this space
  - Kernel trick: $k(\mathbf{x_i}, \mathbf{x_j}) = \Phi(\mathbf{x_i}) \cdot \Phi(\mathbf{x_j})$
  - Kernel k (generalized dot product) is a measure of similarity between $\mathbf{x_i}$ and $\mathbf{x_j}$
- There are many such kernels
  - Polynomial kernel: $k_{poly}(\mathbf{x_1}, \mathbf{x_2}) = (\gamma(\mathbf{x_1} \cdot \mathbf{x_2}) + c_0)^d$
  - RBF (Gaussian) kernel: $k_{RBF}(\mathbf{x_1}, \mathbf{x_2}) = exp(-\gamma||\mathbf{x_1} - \mathbf{x_2}||^2)$
  - A kernel matrix can be precomputed using any similarity measure (e.g. for text, graphs,...)
- Any loss function where inputs appear only as dot products can be kernelized
  - E.g. Linear SVMs: simply replace the dot product with a kernel of choice
- The Representer theorem states which *other* loss functions can also be kernelized and how
  - Ridge regression, Logistic regression, Perceptrons,...

Lecture 4: Model Selection

Can I trust you?
Joaquin Vanschoren

Evaluation

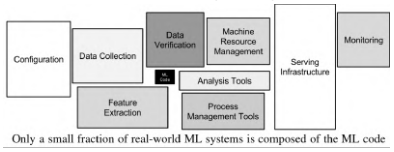
- To know whether we can *trust* our method or system, we need to evaluate it.
- Model selection: choose between different models in a data-driven way.
  - If you cannot measure it, you cannot improve it.
- Convince others that your work is meaningful
  - Peers, leadership, clients, yourself(!)
- When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



(a) Husky classified as wolf (b) Explanation

Designing Machine Learning systems

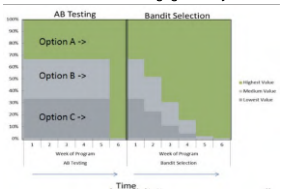
- Just running your favourite algorithm is usually not a great way to start
- Consider the problem: How to measure success? Are there costs involved?
  - Do you want to understand phenomena or do black box modelling?
- Analyze your model's mistakes. Don't just finetune endlessly.
  - Build early prototypes.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



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

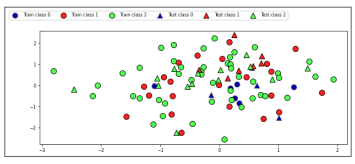
Real world evaluations

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



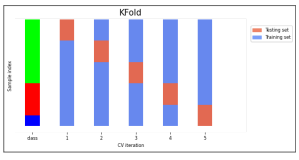
Performance estimation techniques

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



K-fold Cross-validation

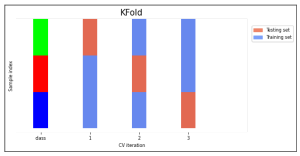
- Each random split can yield very different models (and scores)
  - e.g. all easy (of hard) examples could end up in the test set
- Split data into *k* equal-sized parts, called *folds*
  - Create *k* splits, each time using a different fold as the test set
- Compute *k* evaluation scores, aggregate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



Can you explain this result?

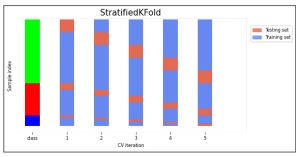
```
kfold = KFold(n_splits=3)
cross_val_score(logistic_regression, iris.data, iris.target, cv=kfold)
```

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



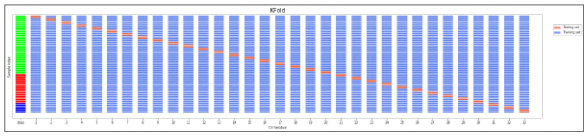
Stratified K-Fold cross-validation

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



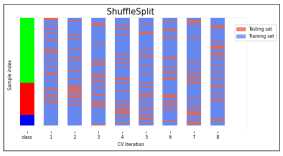
Leave-One-Out cross-validation

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



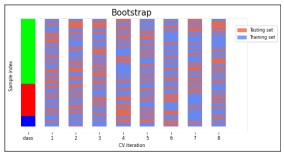
Shuffle-Split cross-validation

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



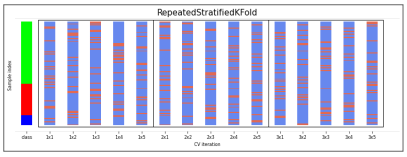
The Bootstrap

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



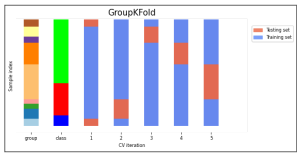
Repeated cross-validation

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



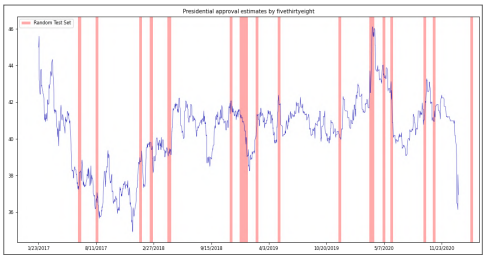
Cross-validation with groups

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



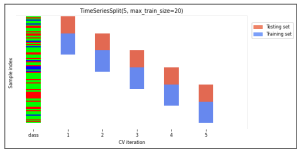
Time series

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



Test-then-train (prequential evaluation)

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



Choosing a performance estimation procedure

No strict rules, only guidelines:

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

Evaluation Metrics for Classification

Evaluation vs Optimization

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

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

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

Binary classification

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

Confusion matrices

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

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

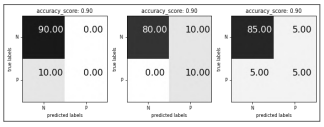
```
confusion_matrix(y_test, y_pred):
 [[48  5]
 [ 5 85]]
```

Predictive accuracy

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

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

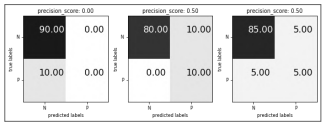
- 3 models: very different predictions, same accuracy:



Precision

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

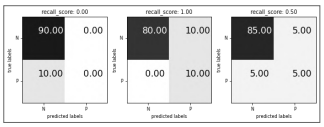
$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{2}$$

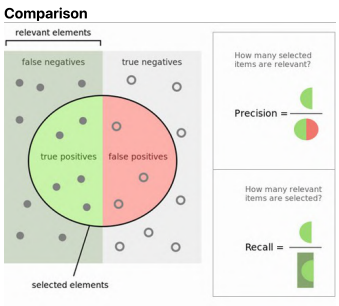


Recall

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

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

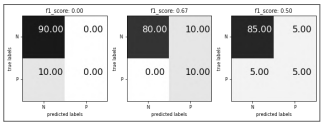




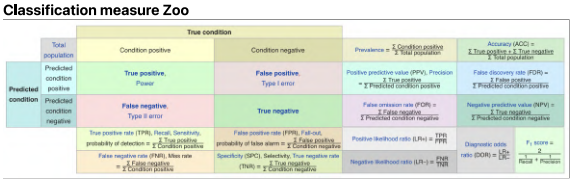
F1-score

- Trades off precision and recall:

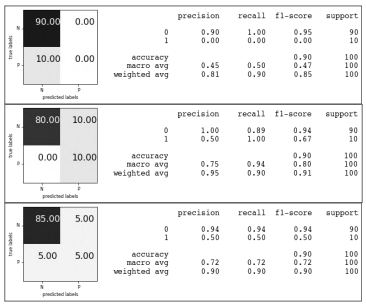
$$\text{F1} = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \tag{4}$$



Classification measure Zoo



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



Multi-class classification

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

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

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

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

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

Other useful classification metrics

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

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

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

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

Probabilistic evaluation

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

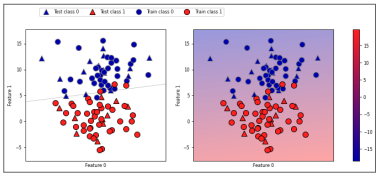
$$\hat{y} = w_0 * x_0 + w_1 * x_1 + \ldots + w_p * x_p + b$$

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

The decision function

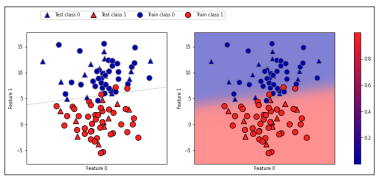
In the binary classification case, the return value of the decision function encodes how strongly the model believes a data point belongs to the "positive" class.

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



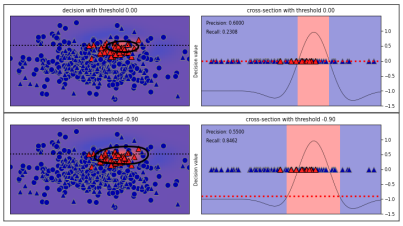
Predicting probabilities

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



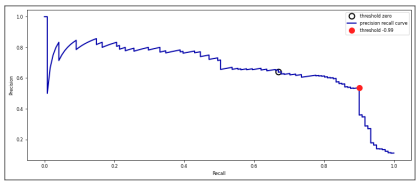
Threshold calibration

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



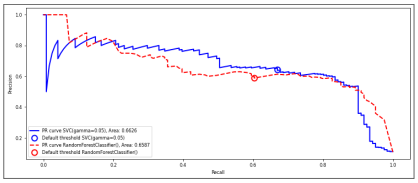
Precision-Recall curve

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



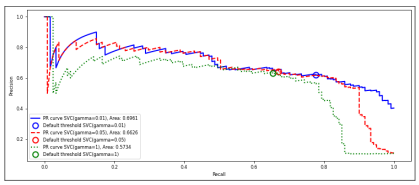
Model selection

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



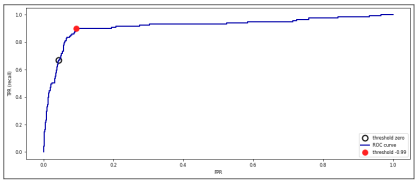
Hyperparameter effects

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



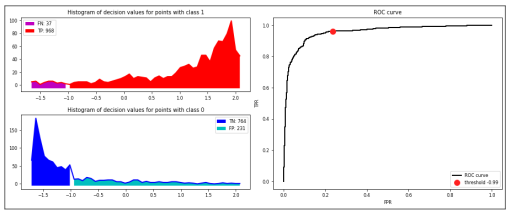
Receiver Operating Characteristics (ROC)

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



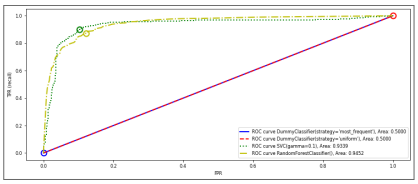
Visualization

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



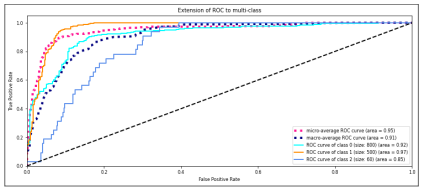
Model selection

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



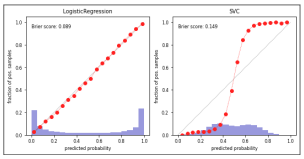
Multi-class AUROC (or AUPRC)

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



Model calibration

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



Brier score

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

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

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

Model calibration techniques

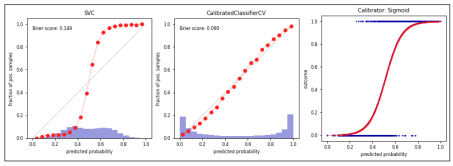
- We can post-process trained models to make them more calibrated.
- Fit a regression model (a calibrator) to map the model's outcomes $f(x)$ to a calibrated probability in [0,1]
  - $f(x)$ returns the decision values or probability estimates
  - f_{calib} is fitted on the training data to map these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

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

Platt Scaling

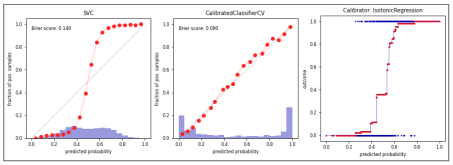
- Calibrator is a logistic (sigmoid) function:
  - Learn the weight w_1 and bias w_0 from data

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



Isotonic regression

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



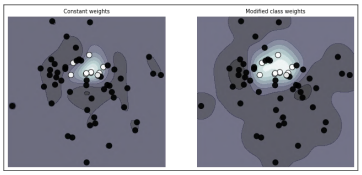
Cost-sensitive classification (dealing with imbalance)

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

Class weighting

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

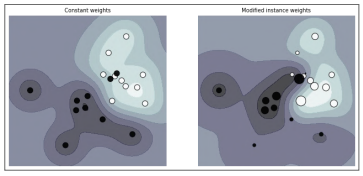
$$\mathcal{L}_{log}(\mathbf{w}) = -\sum_{c=1}^{C} w_c \sum_{n=1}^{N} p_{n,c} log(q_{n,c})$$



Instance weighting

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

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

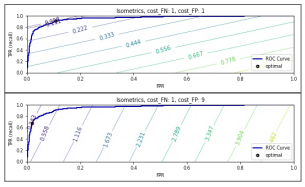


Cost-sensitive algorithms

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

Tuning the decision threshold

- If every FP or FN has a certain cost, we can compute the total cost for a given model:
  $$\text{total cost} = \text{FPR} * cost_{FP} * ratio_{pos} + (1 - \text{TPR}) * cost_{FN} * (1 - ratio_{pos})$$
- This yields different *isometrics* (lines of equal cost) in ROC space
- Optimal threshold is the point on the ROC curve where cost is minimal (line search)



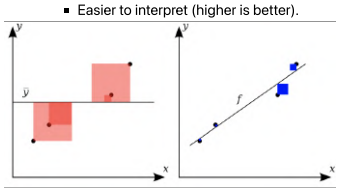
Regression metrics

Most commonly used are

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

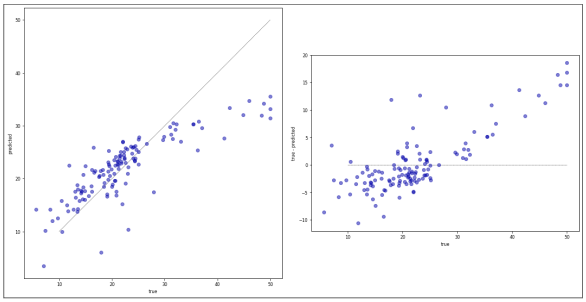
R squared

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



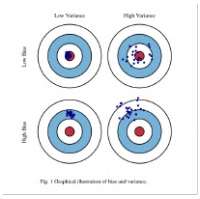
Visualizing regression errors

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



Bias-Variance decomposition

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

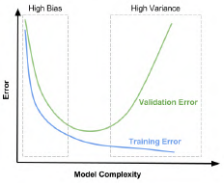


Computing bias and variance error

- Take 100 or more bootstraps (or shuffle-splits)
- Regression: for each data point x:
  - $bias(x)^2 = (x_{true} - mean(x_{predicted}))^2$
  - $variance(x) = var(x_{predicted})$
- Classification: for each data point x:
  - $bias(x)$ = misclassification ratio
  - $variance(x)$
    $= (1 - (P(class_1)^2 + P(class_2)^2))/2$
    - $P(class_i)$ is ratio of class i predictions
- Total bias: $\sum_x bias(x)^2 * w_x$ w_x: the percentage of times x occurs in the test sets
- Total variance: $\sum_x variance(x) * w_x$

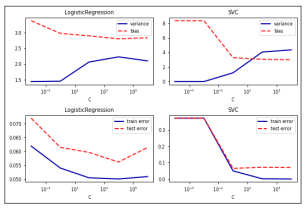
Bias and variance, underfitting and overfitting

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

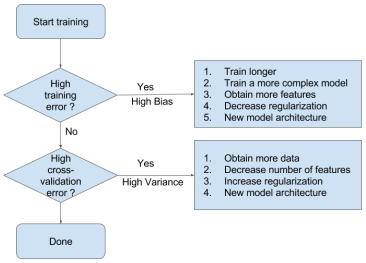


Understanding under- and overfitting

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

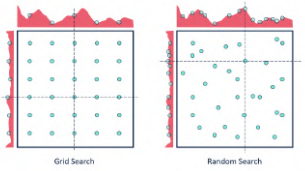


Summary Flowchart (by Andrew Ng)

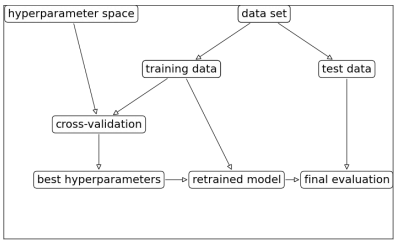


Hyperparameter tuning

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



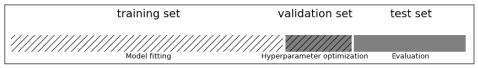
- First, split the data in training and test sets (outer split)
- Split up the training data again (inner cross-validation)
  - Generate hyperparameter configurations (e.g. random/grid search)
  - Evaluate all configurations on all inner splits, select the best one (on average)
- Retrain best configurations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



Nested cross-validation

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

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



Summary

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

Lecture 6. Data preprocessing

Real-world machine learning pipelines

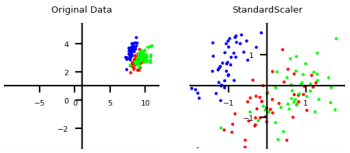
Joaquin Vanschoren

Data transformations

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

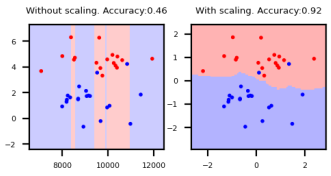
Scaling

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



Why do we need scaling?

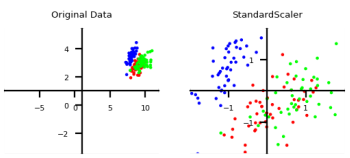
- KNN: Distances depend mainly on feature with larger values
- SVMs: (kernelized) dot products are also based on distances
- Linear model: Feature scale affects regularization
  - Weights have similar scales, more interpretable



Standard scaling (standardization)

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

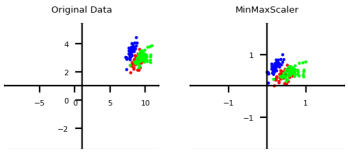
$$\mathbf{x}_{new} = \frac{\mathbf{x} - \mu}{\sigma}$$



Min-max scaling

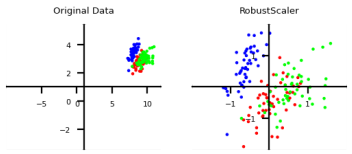
- Scales all features between a given min and max value (e.g. 0 and 1)
- Makes sense if min/max values have meaning in your data
- Sensitive to outliers

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



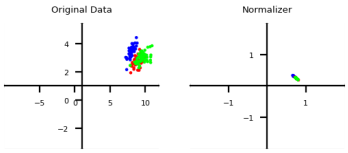
Robust scaling

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



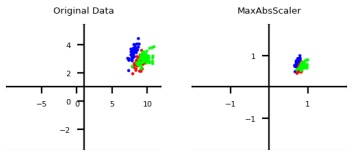
Normalization

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



Maximum Absolute scaler

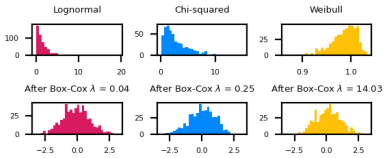
- For sparse data (many features, but few are non-zero)
  - Maintain sparseness (efficient storage)
- Scales all values so that maximum absolute value is 1
- Similar to Min-Max scaling without changing 0 values



Power transformations

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

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



Categorical feature encoding

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

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

Ordinal encoding

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

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

One-hot encoding (dummy encoding)

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

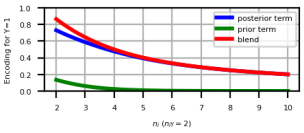
	boro	boro_Bronx	boro_Brooklyn	boro_Manhattan	boro_Queens	salary
0	Manhattan	0	0	1	0	103
1	Queens	0	0	0	1	89
2	Manhattan	0	0	1	0	142
3	Brooklyn	0	1	0	0	54
4	Brooklyn	0	1	0	0	63
5	Bronx	1	0	0	0	219

Target encoding

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

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

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



Example

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

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

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

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

In practice (scikit-learn)

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

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

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

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

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

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

Applying data transformations

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

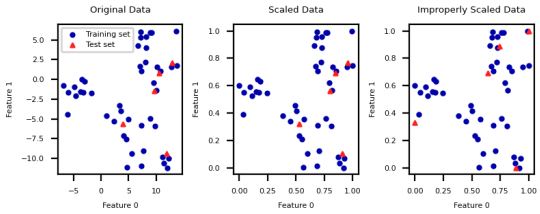
In practice (scikit-learn)

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

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

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

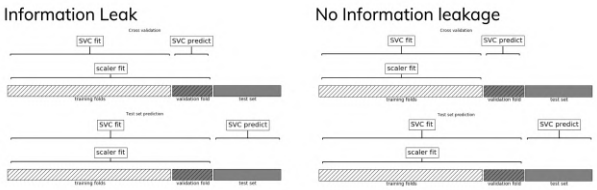
Test set distortion

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



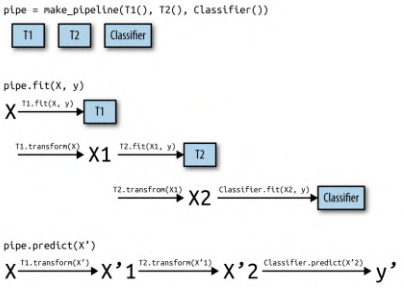
Data leakage

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



Pipelines

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



In practice (scikit-learn)

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

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

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

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

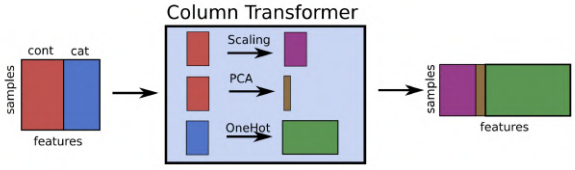
- If you want to apply different preprocessors to different columns, use `ColumnTransformer`
- If you want to merge pipelines, you can use `FeatureUnion` to concatenate columns

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

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

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

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

- `ColumnTransformer` concatenates features in order

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

(OneHotEncoder(),categorical_features))
```



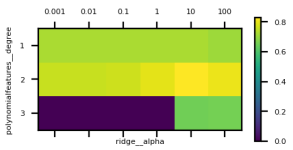
Pipeline selection

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

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

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

Example: Tune multiple steps at once

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



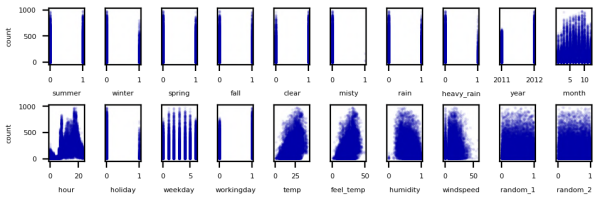
Automatic Feature Selection

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

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

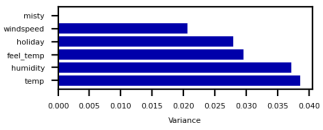
Example: bike sharing

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



Unsupervised feature selection

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

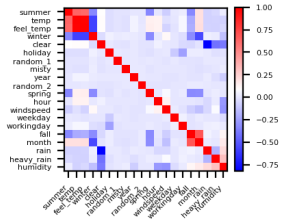


Covariance based feature selection

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

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

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

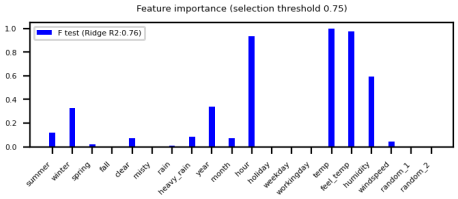


Supervised feature selection: overview

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

Univariate statistics (F-test)

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



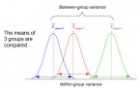
F-statistic

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

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

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

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

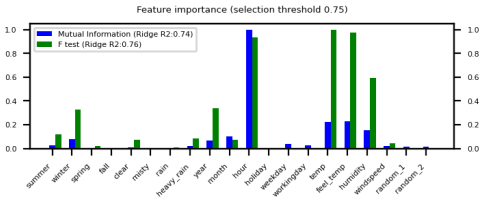


Mutual information

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

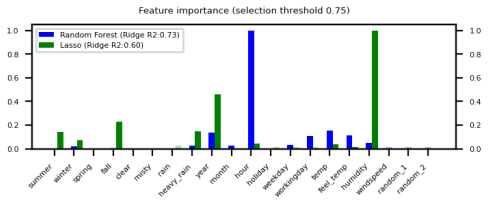
$$MI(X, Y) = H(X) + H(Y) - H(X, Y)$$

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



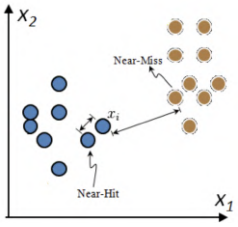
Model-based Feature Selection

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



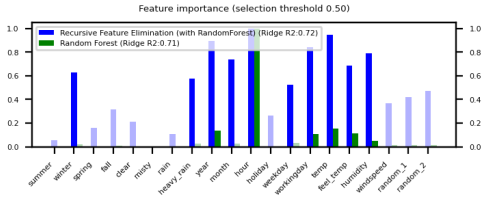
Relief: Model-based selection with kNN

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



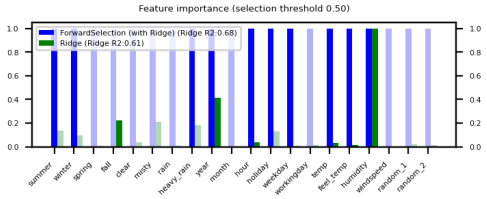
Iterative Model-based Feature Selection

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



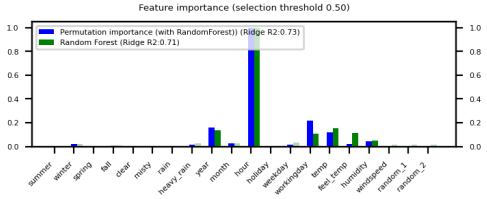
Sequential feature selection (Wrapping)

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



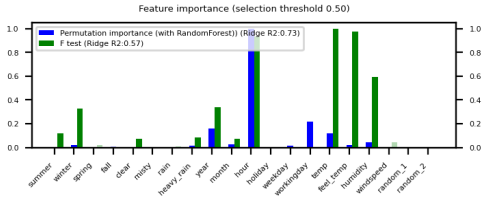
Permutation feature importance

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



Comparison

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



In practice (scikit-learn)

- Unsupervised: `VarianceTreshold`

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

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

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

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

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

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

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

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

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

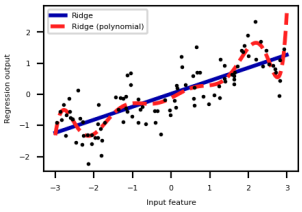
Feature Engineering

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

Polynomials

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

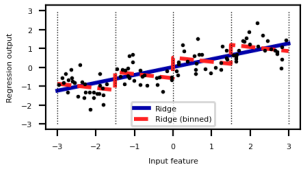
$$[1, x_1, \ldots, x_p] \rightarrow [1, x_1, \ldots, x_p, x_1^2, \ldots, x_p^2, \ldots, x_p^d, x_1 x_2, \ldots, x_{p-1} x_p]$$



Binning

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

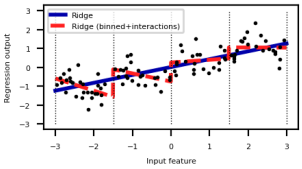
	orig	[-3.0,-1.5]	[-1.5,0.0]	[0.0,1.5]	[1.5,3.0]
0	-0.752759	0.000000	1.000000	0.000000	0.000000
1	2.704286	0.000000	0.000000	0.000000	1.000000
2	1.391964	0.000000	0.000000	1.000000	0.000000



Binning + interaction features

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

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



Categorical feature interactions

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

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

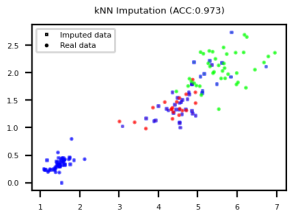
	age_M	pageviews_M	time_M	gender_M_M	age_F	pageviews_F	time_F	gender_F_F
0	14	70	269	1	0	0	0	0
1	0	0	0	0	16	12	1522	1
2	12	42	235	1	0	0	0	0
3	0	0	0	0	25	64	63	1
4	0	0	0	0	22	93	21	1

Missing value imputation

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

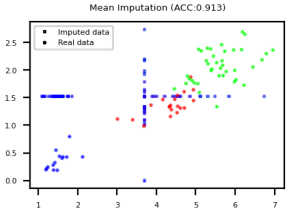
Overview

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



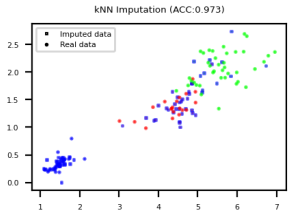
Mean imputation

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



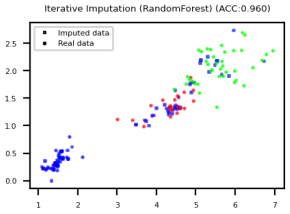
kNN imputation

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



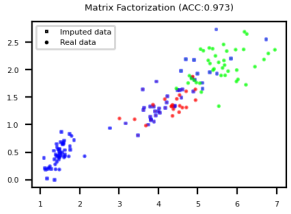
Iterative (model-based) Imputation

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



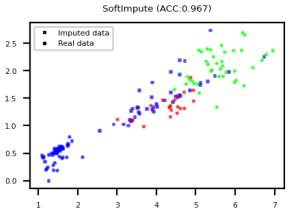
Matrix Factorization

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



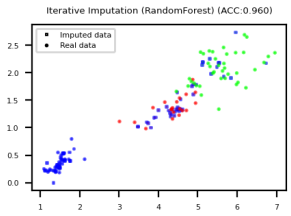
Soft-thresholded Singular Value Decomposition (SVD)

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



Comparison

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



In practice (scikit-learn)

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

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

- kNN Imputation: `KNNImputer`

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

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

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

In practice (fancyimpute)

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

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

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

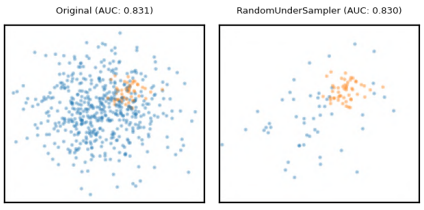
```
imp = MatrixFactorization(rank=10, learning_rate=0.001,
epochs=10000)
X_complete = imp.fit_transform(X)
```

Handling imbalanced data

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

Random Undersampling

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

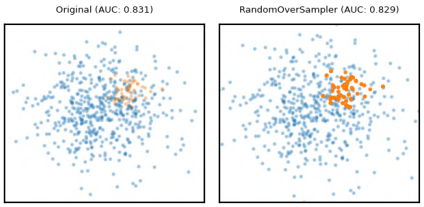


Model-based Undersampling

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

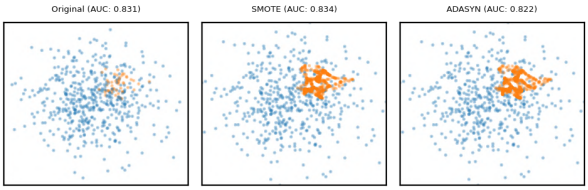
Random Oversampling

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



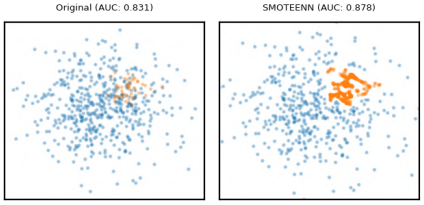
Synthetic Minority Oversampling Technique (SMOTE)

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



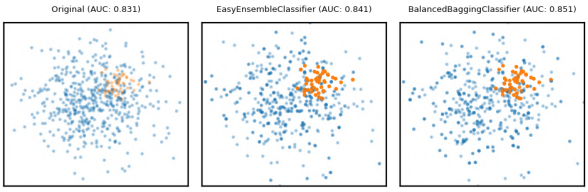
Combined techniques

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



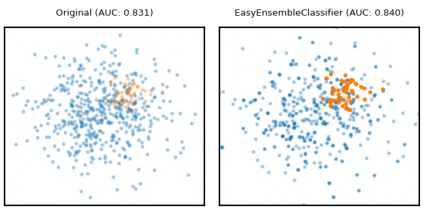
Ensemble Resampling

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



Comparison

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



In practice (imblearn)

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

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

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

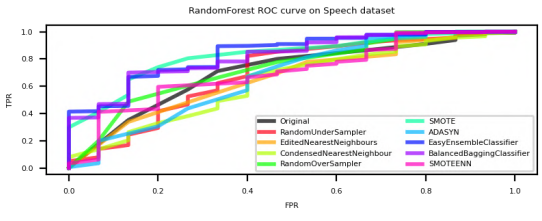
```
smote_pipe = make_pipeline(SMOTE(), LogisticRegression())
scores = cross_validate(smote_pipe, X_train, y_train)
param_grid = {"k_neighbors": [3,5,7]}
grid = GridSearchCV(smote_pipe, param_grid=param_grid, X, y)
```

- The ensembling techniques should be used as wrappers

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

Real-world data

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



Summary

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

Lecture 8. Neural Networks

How to train your neurons

Joaquin Vanschoren

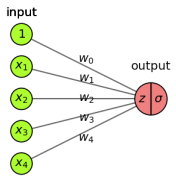
Overview

- Neural architectures
- Training neural nets
  - Forward pass: Tensor operations
  - Backward pass: Backpropagation
- Neural network design:
  - Activation functions
  - Weight initialization
  - Optimizers
- Neural networks in practice
- Model selection
  - Early stopping
  - Memorization capacity and information bottleneck
  - L1/L2 regularization
  - Dropout
  - Batch normalization

Linear models as a building block

- Logistic regression, drawn in a different, neuro-inspired, way
  - Linear model: inner product (z) of input vector $\mathbf{x}$ and weight vector $\mathbf{w}$, plus bias w_0
  - Logistic (or sigmoid) function maps the output to a probability in [0,1]
  - Uses log loss (cross-entropy) and gradient descent to learn the weights

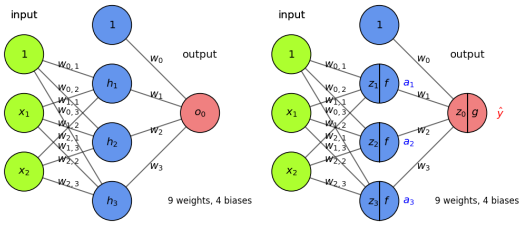
$$\hat{y}(\mathbf{x}) = \text{sigmoid}(z) = \text{sigmoid}(w_0 + \mathbf{w}\mathbf{x}) = \text{sigmoid}(w_0 + w_1 * x_1 + w_2 * x_2 + \ldots + w_p * x_p)$$



Basic Architecture

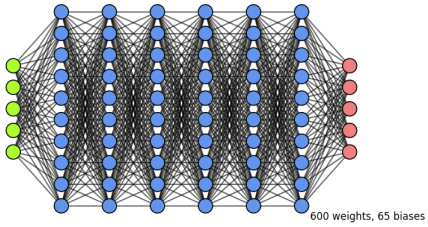
- Add one (or more) *hidden* layers h with k nodes (or units, cells, neurons)
  - Every 'neuron' is a tiny function, the network is an arbitrarily complex function
  - Weights $w_{i,j}$ between node i and node j form a weight matrix $\mathbf{W}^{(l)}$ per layer l
- Every neuron weights the inputs $\mathbf{x}$ and passes it through a non-linear activation function
  - Activation functions (f, g) can be different per layer, output $\mathbf{a}$ is called activation

$$h(\mathbf{x}) = \mathbf{a} = f(\mathbf{z}) = f(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{w}_0^{(1)}) \qquad o(\mathbf{x}) = g(\mathbf{W}^{(2)}\mathbf{a} + \mathbf{w}_0^{(2)})$$



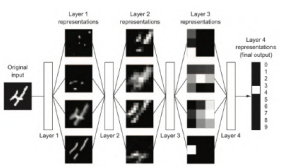
More layers

- Add more layers, and more nodes per layer, to make the model more complex
  - For simplicity, we don't draw the biases (but remember that they are there)
- In *dense* (fully-connected) layers, every previous layer node is connected to all nodes
- The output layer can also have multiple nodes (e.g. 1 per class in multi-class classification)



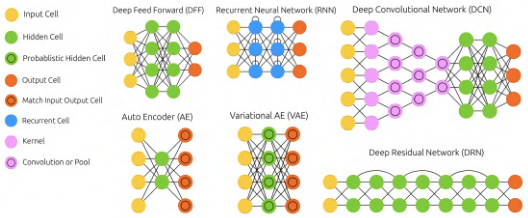
Why layers?

- Each layer acts as a *filter* and learns a new *representation* of the data
  - Subsequent layers can learn iterative refinements
  - Easier that learning a complex relationship in one go
- Example: for image input, each layer yields new (filtered) images
  - Can learn multiple mappings at once: weight *tensor* W yields activation tensor A
  - From low-level patterns (edges, end-points, ...) to combinations thereof
  - Each neuron 'lights up' if certain patterns occur in the input



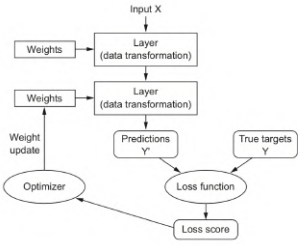
Other architectures

- There exist MANY types of networks for many different tasks
- Convolutional nets for image data, Recurrent nets for sequential data,...
- Also used to learn representations (embeddings), generate new images, text,...



Training Neural Nets

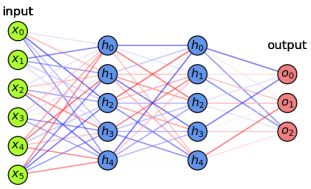
- Design the architecture, choose activation functions (e.g. sigmoids)
- Choose a way to initialize the weights (e.g. random initialization)
- Choose a *loss function* (e.g. log loss) to measure how well the model fits training data
- Choose an *optimizer* (typically an SGD variant) to update the weights



Mini-batch Stochastic Gradient Descent (recap)

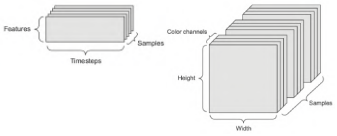
1. Draw a batch of *batch_size* training data $\mathbf{X}$ and $\mathbf{y}$
2. *Forward pass* : pass $\mathbf{X}$ though the network to yield predictions $\hat{\mathbf{y}}$
3. Compute the loss $\mathcal{L}$ (mismatch between $\hat{\mathbf{y}}$ and $\mathbf{y}$)
4. *Backward pass* : Compute the gradient of the loss with regard to every weight
    - *Backpropagate* the gradients through all the layers
5. Update W: $W_{(i+1)} = W_{(i)} - \frac{\partial L(x, W_{(i)})}{\partial W} * \eta$

Repeat until n passes (epochs) are made through the entire training set



Forward pass

- We can naturally represent the data as *tensors*
    - Numerical n-dimensional array (with n axes)
    - 2D tensor: matrix (samples, features)
    - 3D tensor: time series (samples, timesteps, features)
    - 4D tensor: color images (samples, height, width, channels)
    - 5D tensor: video (samples, frames, height, width, channels)

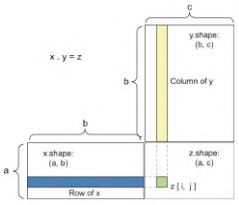


Tensor operations

- The operations that the network performs on the data can be reduced to a *series of tensor operations*
    - These are also much easier to run on GPUs
- A dense layer with sigmoid activation, input tensor $\mathbf{X}$, weight tensor $\mathbf{W}$, bias $\mathbf{b}$:

```
y = sigmoid(np.dot(X, W) + b)
```

- Tensor dot product for 2D inputs (*a* samples, *b* features, *c* hidden nodes)



Element-wise operations

- Activation functions and addition are element-wise operations:

```
def sigmoid(x):
  return 1/(1 + np.exp(-x))

def add(x, y):
  return x + y
```

- Note: if y has a lower dimension than x, it will be *broadcasted*: axes are added to match the dimensionality, and y is repeated along the new axes

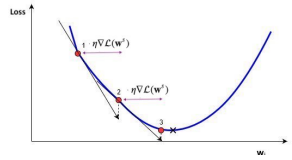
```
>>> np.array([[1,2],[3,4]]) + np.array([10,20])
array([[11, 22],
       [13, 24]])
```

Backward pass (backpropagation)

- For last layer, compute gradient of the loss function $\mathcal{L}$ w.r.t all weights of layer l

$$\nabla\mathcal{L} = \frac{\partial\mathcal{L}}{\partial W^{(l)}} = \begin{bmatrix} \frac{\partial\mathcal{L}}{\partial w_{0,0}} & \cdots & \frac{\partial\mathcal{L}}{\partial w_{0,j}} \\ \vdots & \ddots & \vdots \\ \frac{\partial\mathcal{L}}{\partial w_{k,0}} & \cdots & \frac{\partial\mathcal{L}}{\partial w_{k,j}} \end{bmatrix}$$

- Sum up the gradients for all $\mathbf{x}_j$ in minibatch: $\sum_j \frac{\partial\mathcal{L}(\mathbf{x}_j, y_j)}{\partial W^{(l)}}$
- Update all weights in a layer at once (with learning rate η): $W^{(l)}_{(i+1)} = W^{(l)}_{(i)} - \eta \sum_j \frac{\partial\mathcal{L}(\mathbf{x}_j, y_j)}{\partial W^{(l)}_{(i)}}$
- Repeat for next layer, iterating backwards (most efficient, avoids redundant calculations)

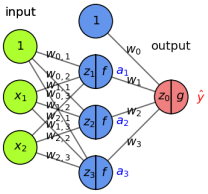


Backpropagation (example)

- Imagine feeding a single data point, output is $\hat{y} = g(z) = g(w_0 + w_1 * a_1 + w_2 * a_2 + \ldots + w_p * a_p)$
- Decrease loss by updating weights:
  - Update the weights of last layer to maximize improvement: $w_{i,(new)} = w_i - \frac{\partial\mathcal{L}}{\partial w_i} * \eta$
  - To compute gradient $\frac{\partial\mathcal{L}}{\partial w_i}$ we need the chain rule: $f(g(x)) = f'(g(x)) * g'(x)$

$$\frac{\partial\mathcal{L}}{\partial w_i} = \frac{\partial\mathcal{L}}{\partial g}\frac{\partial g}{\partial z_0}\frac{\partial z_o}{\partial w_i}$$

- E.g., with $\mathcal{L} = \frac{1}{2}(y - \hat{y})^2$ and sigmoid σ: $\frac{\partial\mathcal{L}}{\partial w_i} = (y - \hat{y}) * \sigma'(z_0) * a_i$

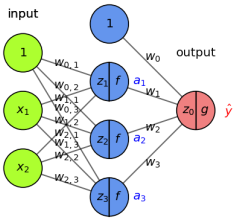


Backpropagation (2)

- Another way to decrease the loss $\mathcal{L}$ is to update the activations a_i
  - To update $a_i = f(z_i)$, we need to update the weights of the previous layer
  - We want to nudge a_i in the right direction by updating $w_{i,j}$:

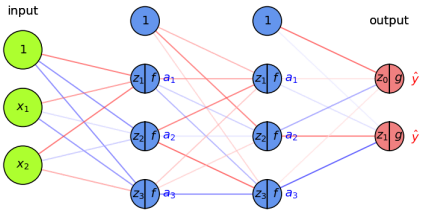
$$\frac{\partial\mathcal{L}}{\partial w_{i,j}} = \frac{\partial\mathcal{L}}{\partial a_i}\frac{\partial a_i}{\partial z_i}\frac{\partial z_i}{\partial w_{i,j}} = \left(\frac{\partial\mathcal{L}}{\partial g}\frac{\partial g}{\partial z_0}\frac{\partial z_o}{\partial a_i}\right)\frac{\partial a_i}{\partial z_i}\frac{\partial z_i}{\partial w_{i,j}}$$

  - We know $\frac{\partial\mathcal{L}}{\partial g}$ and $\frac{\partial g}{\partial z_0}$ from the previous step, $\frac{\partial z_o}{\partial a_i} = w_i$, $\frac{\partial a_i}{\partial z_i} = f'$ and $\frac{\partial z_i}{\partial w_{i,j}} = x_j$



Backpropagation (3)

- With multiple output nodes, $\mathcal{L}$ is the sum of all per-output (per-class) losses
  - $\frac{\partial\mathcal{L}}{\partial a_i}$ is sum of the gradients for every output
- Per layer, sum up gradients for every point $\mathbf{x}$ in the batch: $\sum_j \frac{\partial\mathcal{L}(\mathbf{x}_j, y_j)}{\partial W}$
- Update all weights of every layer l
  - $W^{(l)}_{(i+1)} = W^{(l)}_{(i)} - \eta \sum_j \frac{\partial\mathcal{L}(\mathbf{x}_j, y_j)}{\partial W^{(l)}_{(i)}}$
- Repeat with a new batch of data until loss converges
- Nice animation of the entire process



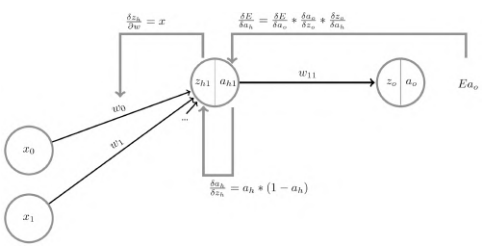
Backpropagation (summary)

- The network output a_o is defined by the weights $W^{(o)}$ and biases $\mathbf{b}^{(o)}$ of the output layer, and
- The activations of a hidden layer h_1 with activation function a_{h_1}, weights $W^{(1)}$ and biases $\mathbf{b}^{(1)}$:

$$a_o(\mathbf{x}) = a_o(\mathbf{z_0}) = a_o(W^{(o)}a_{h_1}(z_{h_1}) + \mathbf{b}^{(o)}) = a_o(W^{(o)}a_{h_1}(W^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) + \mathbf{b}^{(o)})$$

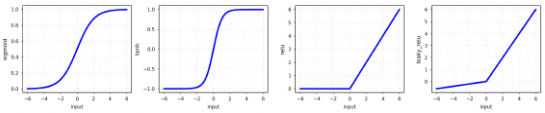
- Minimize the loss by SGD. For layer l, compute $\frac{\partial\mathcal{L}(a_o(x))}{\partial W_l}$ and $\frac{\partial\mathcal{L}(a_o(x))}{\partial b_{l,i}}$ using the chain rule
- Decomposes into gradient of layer above, gradient of activation function, gradient of layer input:

$$\frac{\partial\mathcal{L}(a_o)}{\partial W^{(1)}} = \frac{\partial\mathcal{L}(a_o)}{\partial a_{h_1}}\frac{\partial a_{h_1}}{\partial z_{h_1}}\frac{\partial z_{h_1}}{\partial W^{(1)}} = \left(\frac{\partial\mathcal{L}(a_o)}{\partial a_o}\frac{\partial a_o}{\partial z_o}\frac{\partial z_o}{\partial a_{h_1}}\right)\frac{\partial a_{h_1}}{\partial z_{h_1}}\frac{\partial z_{h_1}}{\partial W^{(1)}}$$



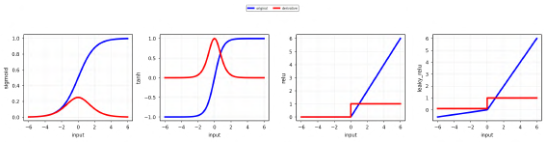
Activation functions for hidden layers

- Sigmoid: $f(z) = \frac{1}{1+e^{-z}}$
- Tanh: $f(z) = \frac{2}{1+e^{-2z}} - 1$
  - Activations around 0 are better for gradient descent convergence
- Rectified Linear (ReLU): $f(z) = max(0, z)$
  - Less smooth, but much faster (note: not differentiable at 0)
- Leaky ReLU: $f(z) = \begin{cases} 0.01z & z < 0 \\ z & otherwise \end{cases}$



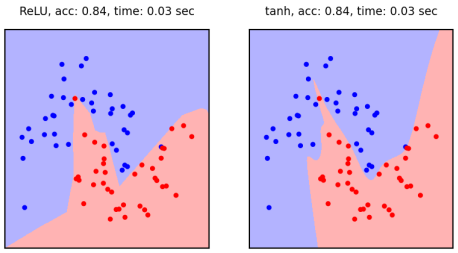
Effect of activation functions on the gradient

- During gradient descent, the gradient depends on the activation function a_h: $\frac{\partial \mathcal{L}(a_o)}{\partial W^{(l)}} = \frac{\partial \mathcal{L}(a_o)}{\partial a_{h_l}} \frac{\partial a_{h_l}}{\partial z_{h_l}} \frac{\partial z_{h_l}}{\partial W^{(l)}}$
- If derivative of the activation function $\frac{\partial a_{h_l}}{\partial z_{h_l}}$ is 0, the weights w_i are not updated
  - Moreover, the gradients of previous layers will be reduced (vanishing gradient)
- sigmoid, tanh: gradient is very small for large inputs: slow updates
- With ReLU, $\frac{\partial a_{h_l}}{\partial z_{h_l}} = 1$ if $z > 0$, hence better against vanishing gradients
  - Problem: for very negative inputs, the gradient is 0 and may never recover (dying ReLU)
  - Leaky ReLU has a small (0.01) gradient there to allow recovery



ReLU vs Tanh

- What is the effect of using non-smooth activation functions?
  - ReLU produces piecewise-linear boundaries, but allows deeper networks
  - Tanh produces smoother decision boundaries, but is slower

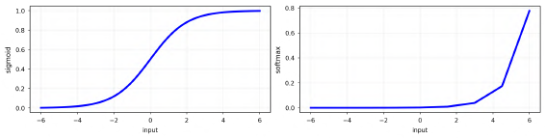


Activation functions for output layer

- *sigmoid* converts output to probability in [0,1]
  - For binary classification
- *softmax* converts all outputs (aka 'logits') to probabilities that sum up to 1
  - For multi-class classification (k classes)
  - Can cause over-confident models. If so, smooth the labels: $y_{smooth} = (1-\alpha)y + \frac{\alpha}{k}$

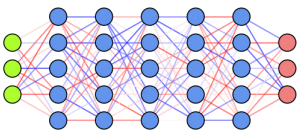
$$\text{softmax}(\mathbf{x}, i) = \frac{e^{x_i}}{\sum_{j=1}^{k} e^{x_j}}$$

- For regression, don't use any activation function, let the model learn the exact target



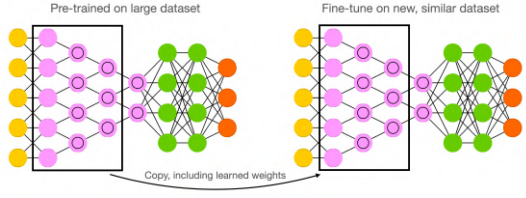
Weight initialization

- Initializing weights to 0 is bad: all gradients in layer will be identical (symmetry)
- Too small random weights shrink activations to 0 along the layers (vanishing gradient)
- Too large random weights multiply along layers (exploding gradient, zig-zagging)
- Ideal: small random weights + variance of input and output gradients remains the same
  - Glorot/Xavier initialization (for tanh): randomly sample from $N(0, \sigma), \sigma = \sqrt{\frac{2}{\text{fan\_in} + \text{fan\_out}}}$
    - fan_in: number of input units, fan_out: number of output units
  - He initialization (for ReLU): randomly sample from $N(0, \sigma), \sigma = \sqrt{\frac{2}{\text{fan\_in}}}$
  - Uniform sampling (instead of $N(0, \sigma)$) for deeper networks (w.r.t. vanishing gradients)



Weight initialization: transfer learning

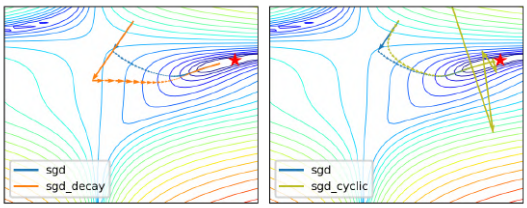
- Instead of starting from scratch, start from weights previously learned from similar tasks
  - This is, to a big extent, how humans learn so fast
- Transfer learning: learn weights on task T, transfer them to new network
  - Weights can be frozen, or finetuned to the new data
- Only works if the previous task is 'similar' enough
  - Meta-learning: learn a good initialization across many related tasks



Optimizers

SGD with learning rate schedules

- Using a constant learning η rate for weight updates $\mathbf{w}_{(s+1)} = \mathbf{w}_s - \eta \nabla \mathcal{L}(\mathbf{w}_s)$ is not ideal
- Learning rate decay/annealing with decay rate k
  - E.g. exponential ($\eta_{s+1} = \eta_s e^{-ks}$), inverse-time ($\eta_{s+1} = \frac{\eta_s}{1+ks}$),...
- Cyclical learning rates
  - Change from small to large: hopefully in 'good' region long enough before diverging
  - Warm restarts: aggressive decay + reset to initial learning rate



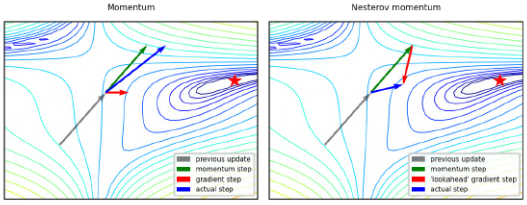
Momentum

- Imagine a ball rolling downhill: accumulates momentum, doesn't exactly follow steepest descent
  - Reduces oscillation, follows larger (consistent) gradient of the loss surface
- Adds a velocity vector $\mathbf{v}$ with momentum γ (e.g. 0.9, or increase from $\gamma = 0.5$ to $\gamma = 0.99$)

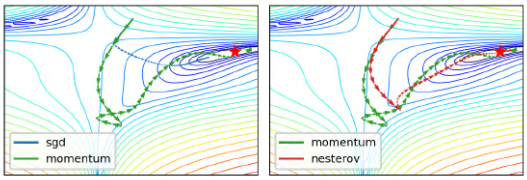
$$\mathbf{w}_{(s+1)} = \mathbf{w}_{(s)} + \mathbf{v}_{(s)} \qquad \text{with} \qquad \mathbf{v}_{(s)} = \gamma \mathbf{v}_{(s-1)} - \eta \nabla \mathcal{L}(\mathbf{w}_{(s)})$$

- Nesterov momentum: Look where momentum step would bring you, compute gradient there
  - Responds faster (and reduces momentum) when the gradient changes

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



Momentum in practice



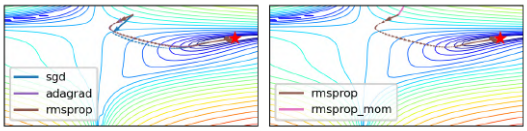
Adaptive gradients

- 'Correct' the learning rate for each w_i based on specific local conditions (layer depth, fan-in,...)
- Adagrad: scale η according to squared sum of previous gradients $G_{i,(s)} = \sum_{t=1}^{s} \mathcal{L}(w_{i,(t)})^2$
  - Update rule for w_i. Usually $\epsilon = 10^{-7}$ (avoids division by 0), $\eta = 0.001$.

$$w_{i,(s+1)} = w_{i,(s)} - \frac{\eta}{\sqrt{G_{i,(s)} + \epsilon}} \nabla \mathcal{L}(w_{i,(s)})$$

- RMSProp: use *moving average* of squared gradients $m_{i,(s)} = \gamma m_{i,(s-1)} + (1-\gamma) \nabla \mathcal{L}(w_{i,(s)})^2$
  - Avoids that gradients dwindle to 0 as $G_{i,(s)}$ grows. Usually $\gamma = 0.9, \eta = 0.001$

$$w_{i,(s+1)} = w_{i,(s)} - \frac{\eta}{\sqrt{m_{i,(s)} + \epsilon}} \nabla \mathcal{L}(w_{i,(s)})$$



Adam (Adaptive moment estimation)

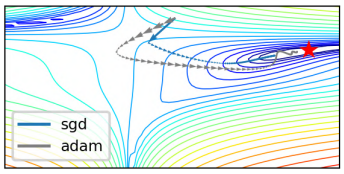
- Adam: RMSProp + momentum. Adds moving average for gradients as well (γ_2 = momentum):
  - Adds a bias correction to avoid small initial gradients: $\hat{m}_{i,(s)} = \frac{m_{i,(s)}}{1-\gamma}$ and $\hat{g}_{i,(s)} = \frac{g_{i,(s)}}{1-\gamma_2}$

$$g_{i,(s)} = \gamma_2 g_{i,(s-1)} + (1-\gamma_2) \nabla \mathcal{L}(w_{i,(s)})$$

$$w_{i,(s+1)} = w_{i,(s)} - \frac{\eta}{\sqrt{\hat{m}_{i,(s)} + \epsilon}} \hat{g}_{i,(s)}$$

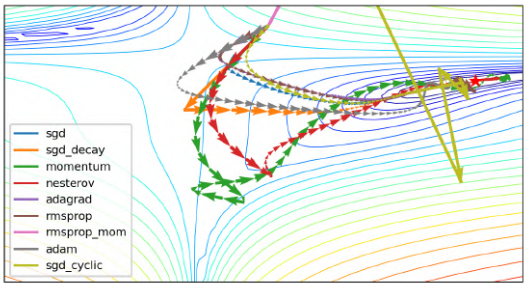
- Adamax: Idem, but use max() instead of moving average: $u_{i,(s)} = max(\gamma u_{i,(s-1)}, |\mathcal{L}(w_{i,(s)})|)$

$$w_{i,(s+1)} = w_{i,(s)} - \frac{\eta}{u_{i,(s)}} \hat{g}_{i,(s)}$$



SGD Optimizer Zoo

- RMSProp often works well, but do try alternatives. For even more optimizers, see here.



Neural networks in practice

- There are many practical courses on training neural nets. E.g.:
  - With TensorFlow: https://www.tensorflow.org/resources/learn-ml
  - With PyTorch: fast.ai course, https://pytorch.org/tutorials/
- Here, we'll use Keras, a general API for building neural networks
  - Default API for TensorFlow, also has backends for CNTK, Theano
- Focus on key design decisions, evaluation, and regularization
- Running example: Fashion-MNIST
  - 28x28 pixel images of 10 classes of fashion items

Building the network

- We first build a simple sequential model (no branches)
- Input layer ('input_shape'): a flat vector of 28*28=784 nodes
  - We'll see how to properly deal with images later
- Two dense hidden layers: 512 nodes each, ReLU activation
  - Glorot weight initialization is applied by default
- Output layer: 10 nodes (for 10 classes) and softmax activation

```
network = models.Sequential()
network.add(layers.Dense(512, activation='relu',
kernel_initializer='he_normal', input_shape=(28 * 28,)))
network.add(layers.Dense(512, activation='relu',
kernel_initializer='he_normal'))
network.add(layers.Dense(10, activation='softmax'))
```

Model summary

- Lots of parameters (weights and biases) to learn!
  - hidden layer 1 : (28 *28 + 1)* 512 = 401920
  - hidden layer 2 : (512 + 1) * 512 = 262656
  - output layer: (512 + 1) * 10 = 5130

```
network.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 512)               401920

 dense_1 (Dense)             (None, 512)               262656

 dense_2 (Dense)             (None, 10)                5130

=================================================================
Total params: 669,706
Trainable params: 669,706
Non-trainable params: 0
_________________________________________________________________
```

Choosing loss, optimizer, metrics

- **Loss function**
  - Cross-entropy (log loss) for multi-class classification (y_{true} is one-hot encoded)
  - Use binary crossentropy for binary problems (single output node)
  - Use sparse categorical crossentropy if y_{true} is label-encoded (1,2,3,...)
- **Optimizer**
  - Any of the optimizers we discussed before. RMSprop usually works well.
- **Metrics**
  - To monitor performance during training and testing, e.g. accuracy

```
# Shorthand
network.compile(loss='categorical_crossentropy',
optimizer='rmsprop', metrics=['accuracy'])
# Detailed
network.compile(loss=CategoricalCrossentropy(label_smoothing=0.01),
                optimizer=RMSprop(learning_rate=0.001, momentum=0.0)
                metrics=[Accuracy()])
```

Preprocessing: Normalization, Reshaping, Encoding

- Always normalize (standardize or min-max) the inputs. Mean should be close to 0.
  - Avoid that some inputs overpower others
  - Speed up convergence
    - Gradients of activation functions $\frac{\partial a_h}{\partial z_h}$ are (near) 0 for large inputs
    - If some gradients become much larger than others, SGD will start zig-zagging
- Reshape the data to fit the shape of the input layer, e.g. (n, 28*28) or (n, 28,28)
  - Tensor with instances in first dimension, rest must match the input layer
- In multi-class classification, every class is an output node, so one-hot-encode the labels
  - e.g. class '4' becomes [0,0,0,0,1,0,0,0,0,0]

```
X = X.astype('float32') / 255
X = X.reshape((60000, 28 * 28))
y = to_categorical(y)
```

Choosing training hyperparameters

- Number of epochs: enough to allow convergence
  - Too much: model starts overfitting (or just wastes time)
- Batch size: small batches (e.g. 32, 64,... samples) often preferred
  - 'Noisy' training data makes overfitting less likely
    - Larger batches generalize less well ('generalization gap')
  - Requires less memory (especially in GPUs)
  - Large batches do speed up training, may converge in fewer epochs
- Batch size interacts with learning rate
  - Instead of shrinking the learning rate you can increase batch size

```
history = network.fit(X_train, y_train, epochs=3, batch_size=32);
```

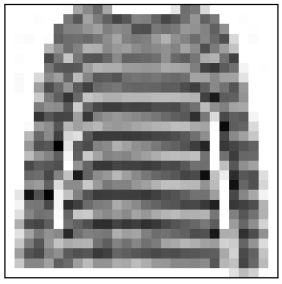
```
Epoch 1/3
1875/1875 [==============================] - 24s 13ms/step - loss: 0.4331 - accuracy: 0.8529
Epoch 2/3
1875/1875 [==============================] - 25s 13ms/step - loss: 0.4242 - accuracy: 0.8568
Epoch 3/3
1875/1875 [==============================] - 26s 14ms/step - loss: 0.4183 - accuracy: 0.8573
```

Predictions and evaluations

We can now call `predict` to generate predictions, and evaluate the trained model on the entire test set

```
network.predict(X_test)
test_loss, test_acc = network.evaluate(X_test, y_test)
```

```
[0.0240177 0.0001167 0.4472437 0.0056629 0.057807  0.000094  0.4632739
 0.0000267 0.0017463 0.0000112]
```

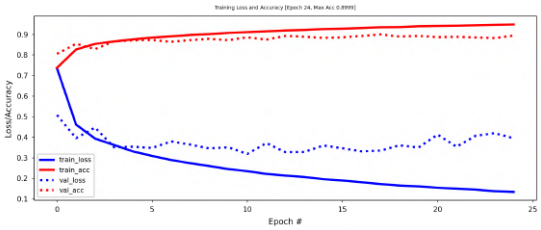


True label: [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]

```
313/313 [==============================] - 2s 7ms/step - loss: 0.3845 - accuracy: 0.8636
Test accuracy: 0.8636000156402588
```

Model selection

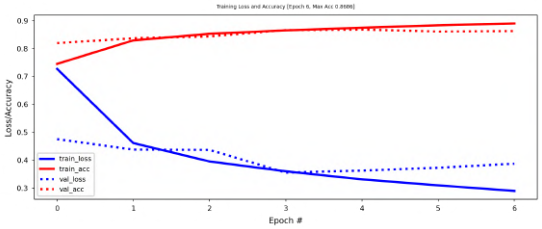
- How many epochs do we need for training?
- Train the neural net and track the loss after every iteration on a validation set
  - You can add a callback to the fit version to get info on every epoch
- Best model after a few epochs, then starts overfitting



Early stopping

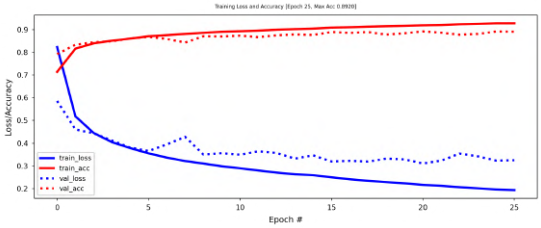
- Stop training when the validation loss (or validation accuracy) no longer improves
- Loss can be bumpy: use a moving average or wait for k steps without improvement

```
earlystop = callbacks.EarlyStopping(monitor='val_loss', patience=3)
model.fit(x_train, y_train, epochs=25, batch_size=512, callbacks=
[earlystop])
```



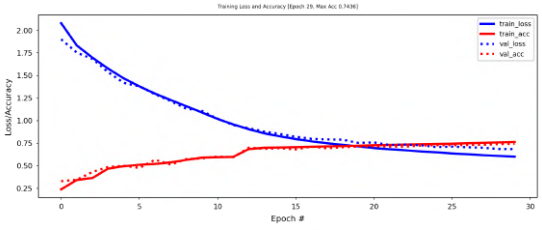
Regularization and memorization capacity

- The number of learnable parameters is called the model *capacity*
- A model with more parameters has a higher *memorization capacity*
  - Too high capacity causes overfitting, too low causes underfitting
  - In the extreme, the training set can be 'memorized' in the weights
- Smaller models are forced it to learn a compressed representation that generalizes better
  - Find the sweet spot: e.g. start with few parameters, increase until overfitting stars.
- Example: 256 nodes in first layer, 32 nodes in second layer, similar performance



Information bottleneck

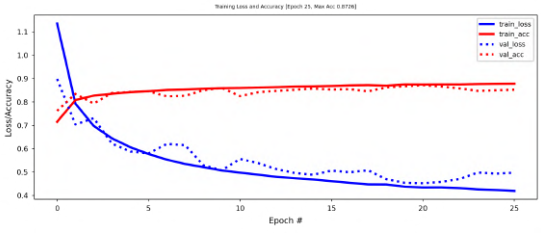
- If a layer is too narrow, it will lose information that can never be recovered by subsequent layers
- *Information bottleneck* theory defines a bound on the capacity of the network
- Imagine that you need to learn 10 outputs (e.g. classes) and your hidden layer has 2 nodes
  - This is like trying to learn 10 hyperplanes from a 2-dimensional representation
- Example: bottleneck of 2 nodes, no overfitting, much higher training loss



Weight regularization (weight decay)

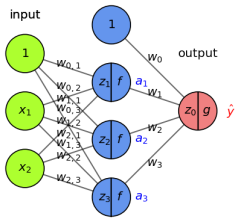
- As we did many times before, we can also add weight regularization to our loss function
- L1 regularization: leads to *sparse networks* with many weights that are 0
- L2 regularization: leads to many very small weights

```
network = models.Sequential()
network.add(layers.Dense(256, activation='relu',
kernel_regularizer=regularizers.l2(0.001), input_shape=(28 * 28,)))
network.add(layers.Dense(128, activation='relu',
kernel_regularizer=regularizers.l2(0.001)))
```



Dropout

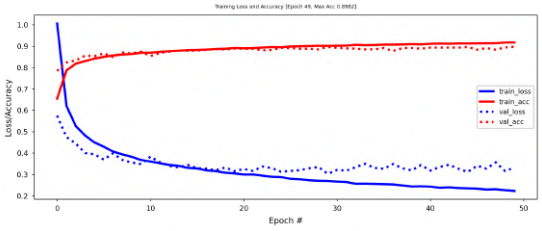
- Every iteration, randomly set a number of activations a_i to 0
- *Dropout rate* : fraction of the outputs that are zeroed-out (e.g. 0.1 - 0.5)
- Idea: break up accidental non-significant learned patterns
- At test time, nothing is dropped out, but the output values are scaled down by the dropout rate
    - Balances out that more units are active than during training



Dropout layers

- Dropout is usually implemented as a special layer

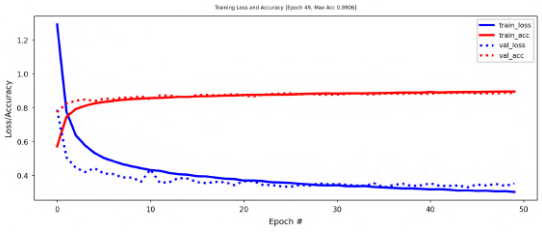
```
network = models.Sequential()
network.add(layers.Dense(256, activation='relu', input_shape=(28 *
28,)))
network.add(layers.Dropout(0.5))
network.add(layers.Dense(32, activation='relu'))
network.add(layers.Dropout(0.5))
network.add(layers.Dense(10, activation='softmax'))
```



Batch Normalization

- We've seen that scaling the input is important, but what if layer activations become very large?
    - Same problems, starting deeper in the network
- Batch normalization: normalize the activations of the previous layer within each batch
    - Within a batch, set the mean activation close to 0 and the standard deviation close to 1
        - Across badges, use exponential moving average of batch-wise mean and variance
    - Allows deeper networks less prone to vanishing or exploding gradients

```
network = models.Sequential()
network.add(layers.Dense(512, activation='relu', input_shape=(28 *
28,)))
network.add(layers.BatchNormalization())
network.add(layers.Dropout(0.5))
network.add(layers.Dense(256, activation='relu'))
network.add(layers.BatchNormalization())
network.add(layers.Dropout(0.5))
network.add(layers.Dense(64, activation='relu'))
network.add(layers.BatchNormalization())
network.add(layers.Dropout(0.5))
network.add(layers.Dense(32, activation='relu'))
network.add(layers.BatchNormalization())
network.add(layers.Dropout(0.5))
```



Tuning multiple hyperparameters

- You can wrap Keras models as scikit-learn models and use any tuning technique
- Keras also has built-in RandomSearch (and HyperBand and BayesianOptimization - see later)

```
def make_model(hp):
    m.add(Dense(units=hp.Int('units', min_value=32, max_value=512,
step=32)))
    m.compile(optimizer=Adam(hp.Choice('learning rate', [1e-2, 1e-3,
1e-4])))
    return model
```

```
from tensorflow.keras.wrappers.scikit_learn import KerasClassifier
clf = KerasClassifier(make_model)
grid = GridSearchCV(clf, param_grid=param_grid, cv=3)

from kerastuner.tuners import RandomSearch
tuner = keras.RandomSearch(build_model, max_trials=5)
```

Summary

- Neural architectures
- Training neural nets
    - Forward pass: Tensor operations
    - Backward pass: Backpropagation
- Neural network design:
    - Activation functions
    - Weight initialization
    - Optimizers
- Neural networks in practice
- Model selection
    - Early stopping
    - Memorization capacity and information bottleneck
    - L1/L2 regularization
    - Dropout
    - Batch normalization

Lecture 9: Convolutional Neural Networks

Handling image data

Joaquin Vanschoren, Eindhoven University of Technology

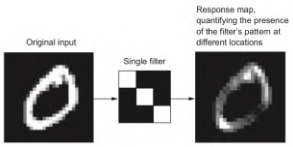
Overview

- Image convolution
- Convolutional neural networks
- Data augmentation
- Model interpretation
- Using pre-trained networks (transfer learning)

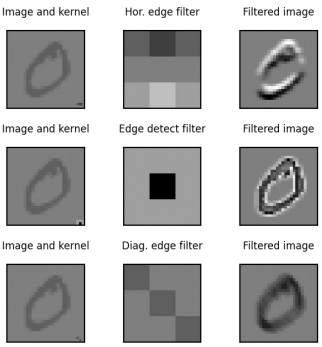
Convolution

- Operation that transforms an image by sliding a smaller image (called a *filter* or *kernel*) over the image and multiplying the pixel values
  - Slide an n x n filter over n x n *patches* of the original image
  - Every pixel is replaced by the *sum* of the *element-wise products* of the values of the image patch around that pixel and the kernel

```
# kernel and image_patch are n x n matrices
pixel_out = np.sum(kernel * image_patch)
```



- Different kernels can detect different types of patterns in the image



Demonstration on Google streetview data

House numbers photographed from Google streetview imagery, cropped and centered around digits, but with neighboring numbers or other edge artifacts.

For recognizing digits, color is not important, so we grayscale the images

Demonstration

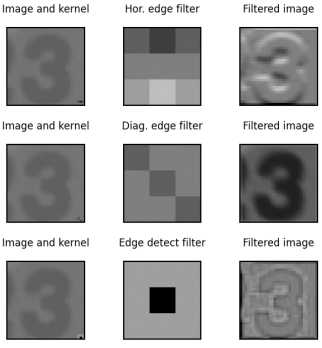
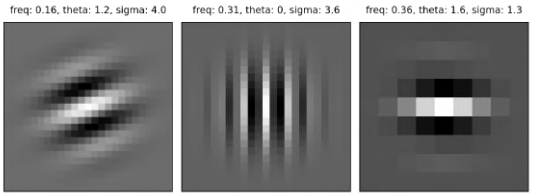


Image convolution in practice

- How do we know which filters are best for a given image?
- *Families* of kernels (or *filter banks*) can be run on every image
  - Gabor, Sobel, Haar Wavelets,...
- Gabor filters: Wave patterns generated by changing:
  - Frequency: narrow or wide ondulations
  - Theta: angle (direction) of the wave
  - Sigma: resolution (size of the filter)

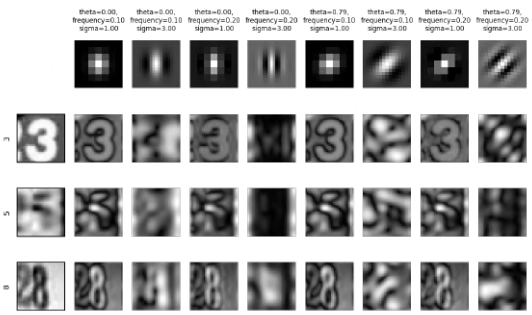
Demonstration



Demonstration on the streetview data

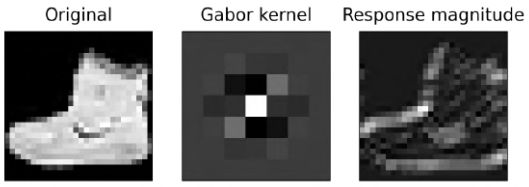
Filter banks

- Different filters detect different edges, shapes,...
- Not all seem useful

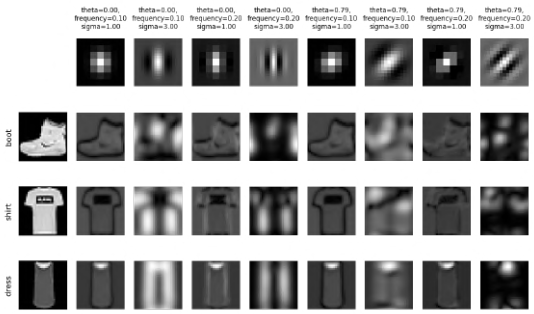


Another example: Fashion MNIST

Demonstration

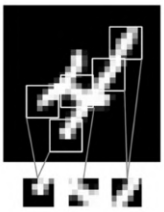


Fashion MNIST with multiple filters (filter bank)



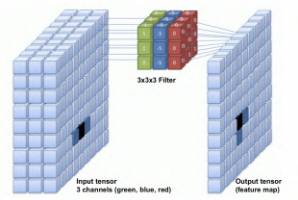
Convolutional neural nets

- Finding relationships between individual pixels and the correct class is hard
- We want to discover 'local' patterns (edges, lines, endpoints)
- Representing such local patterns as features makes it easier to learn from them
- We could use convolutions, but how to choose the filters?



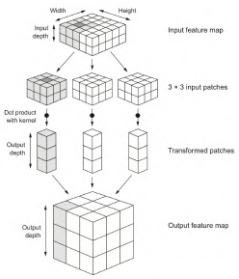
Convolutional Neural Networks (ConvNets)

- Instead of manually designing the filters, we can also *learn* them based on data
  - Choose filter sizes (manually), initialize with small random weights
- Forward pass: Convolutional layer slides the filter over the input, generates the output
- Backward pass: Update the filter weights according to the loss gradient
- Illustration for 1 filter:



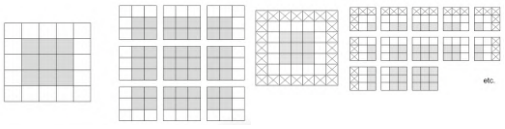
Convolutional layers: Feature maps

- One filter is not sufficient to detect all relevant patterns in an image
- A convolutional layer applies and learns d filter in parallel
- Slide d filters across the input image (in parallel) -> a (1x1xd) output per patch
- Reassemble into a *feature map* with d 'channels', a (width x height x d) tensor.



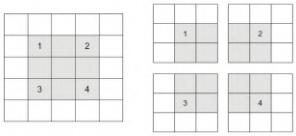
Border effects (zero padding)

- Consider a 5x5 image and a 3x3 filter: there are only 9 possible locations, hence the output is a 3x3 feature map
- If we want to maintain the image size, we use *zero-padding*, adding 0's all around the input tensor.



Undersampling (striding)

- Sometimes, we want to *downsample* a high-resolution image
  - Faster processing, less noisy (hence less overfitting)
- One approach is to *skip* values during the convolution
  - Distance between 2 windows: *stride length*
- Example with stride length 2 (without padding):



Max-pooling

- Another approach to shrink the input tensors is *max-pooling* :
  - Run a filter with a fixed stride length over the image
    - Usually 2x2 filters and stride lenght 2
  - The filter simply returns the *max* (or *avg*) of all values
- Agressively reduces the number of weights (less overfitting)
- Information from every input node spreads more quickly to output nodes
  - In `pure` convnets, one input value spreads to 3x3 nodes of the first layer, 5x5 nodes of the second, etc.
  - Without maxpooling, you need much deeper networks, harder to train
- Increases *translation invariance* : patterns can affect the predictions no matter where they occur in the image

Convolutional nets in practice

- ConvNets usually use multiple convolutional layers to learn patterns at different levels of abstraction
  - Find local patterns first (e.g. edges), then patterns across those patterns
- Use MaxPooling layers to reduce resolution, increase translation invariance
- Use sufficient filters in the first layer (otherwise information gets lost)
- In deeper layers, use increasingly more filters
  - Preserve information about the input as resolution descreases
  - Avoid decreasing the number of activations (resolution x nr of filters)

Example with Keras:

- `Conv2D` for 2D convolutional layers
  - 32 filters (default), randomly initialized (from uniform distribution)
  - Deeper layers use 64 filters
  - Filter size is 3x3
  - ReLU activation to simplify training of deeper networks
- `MaxPooling2D` for max-pooling
  - 2x2 pooling reduces the number of inputs by a factor 4

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
                        input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
```

Observe how the input image on 28x28x1 is transformed to a 3x3x64 feature map

- Convolutional layer:
  - No zero-padding: every output 2 pixels less in every dimension
  - 320 weights: (3x3 filter weights + 1 bias) * 32 filters
- After every MaxPooling, resolution halved in every dimension

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 26, 26, 32)        320

 max_pooling2d (MaxPooling2D  (None, 13, 13, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 11, 11, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 5, 5, 64)         0
 2D)

 conv2d_2 (Conv2D)           (None, 3, 3, 64)          36928

=================================================================
Total params: 55,744
Trainable params: 55,744
Non-trainable params: 0
_________________________________________________________________
```

Completing the network

- To classify the images, we still need a Dense and Softmax layer.
- We need to flatten the 3x3x64 feature map to a vector of size 576

```
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

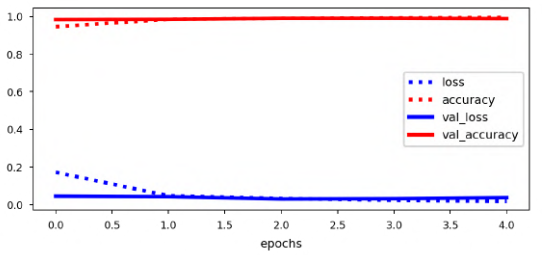
Complete network

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 26, 26, 32)        320

 max_pooling2d (MaxPooling2D  (None, 13, 13, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 11, 11, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 5, 5, 64)         0
 2D)

 conv2d_2 (Conv2D)           (None, 3, 3, 64)          36928

 flatten (Flatten)           (None, 576)               0

 dense (Dense)               (None, 64)                36928

 dense_1 (Dense)             (None, 10)                650

=================================================================
Total params: 93,322
Trainable params: 93,322
Non-trainable params: 0
_________________________________________________________________
```

Run the model on MNIST dataset

- Train and test as usual: 99% accuracy
    - Compared to 97,8% accuracy with the dense architecture

```
Accuracy:  0.9888000488281250
```



Tip:

- Training ConvNets can take a lot of time
- Save the trained model (and history) to disk so that you can reload it later

```
model.save(os.path.join(model_dir, 'mnist.h5'))
with open(os.path.join(model_dir, 'mnist_history.p'), 'wb') as
file_pi:
    pickle.dump(history.history, file_pi)
```

Cats vs Dogs

- A more realistic dataset: Cats vs Dogs
    - Colored JPEG images, different sizes
    - Not nicely centered, translation invariance is important
- Preprocessing
    - Create balanced subsample of 4000 colored images
        - 3000 for training, 1000 validation
    - Decode JPEG images to floating-point tensors
    - Rescale pixel values to [0,1]
    - Resize images to 150x150 pixels

Data generators

- `ImageDataGenerator`: allows to encode, resize, and rescale JPEG images
- Returns a Python *generator* we can endlessly query for batches of images
- Separately for training, validation, and validation set

```
train_generator =
ImageDataGenerator(rescale=1./255).flow_from_directory(
        train_dir,              # Directory with images
        target_size=(150, 150), # Resize images
        batch_size=20,          # Return 20 images at a time
        class_mode='binary')    # Binary labels
```

Since the images are larger and more complex, we add another convolutional layer and increase the number of filters to 128.

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
                        input_shape=(150, 150, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_3 (Conv2D)            (None, 148, 148, 32)      896
_________________________________________________________________
max_pooling2d_2 (MaxPooling  (None, 74, 74, 32)        0
2D)
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 72, 72, 64)        18496
_________________________________________________________________
max_pooling2d_3 (MaxPooling  (None, 36, 36, 64)        0
2D)
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 34, 34, 128)       73856
_________________________________________________________________
max_pooling2d_4 (MaxPooling  (None, 17, 17, 128)       0
2D)
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 15, 15, 128)       147584
_________________________________________________________________
max_pooling2d_5 (MaxPooling  (None, 7, 7, 128)         0
2D)
_________________________________________________________________
flatten_1 (Flatten)          (None, 6272)              0
_________________________________________________________________
dense_2 (Dense)              (None, 512)               3211776
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 513
=================================================================
Total params: 3,453,121
Trainable params: 3,453,121
Non-trainable params: 0
_________________________________________________________________
```

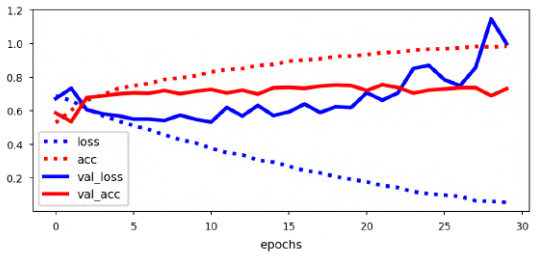
Training

- The `fit` function also supports generators
  - 100 steps per epoch (batch size: 20 images per step), for 30 epochs
  - Provide a separate generator for the validation data

```
model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(lr=1e-4),
              metrics=['acc'])
history = model.fit(
      train_generator, steps_per_epoch=100,
      epochs=30, verbose=0,
      validation_data=validation_generator,
      validation_steps=50)
```

Results

- The network seems to be overfitting. Validation accuracy is stuck at 75% while the training accuracy reaches 100%
- There are many things we can do:
  - Regularization (e.g. Dropout, L1/L2, Batch Normalization,...)
  - Generating more training data
  - Meta-learning: Use pretrained rather than randomly initialized filters



Data augmentation

- Generate new images via image transformations
  - Images will be randomly transformed *every epoch*
- We can again use a data generator to do this

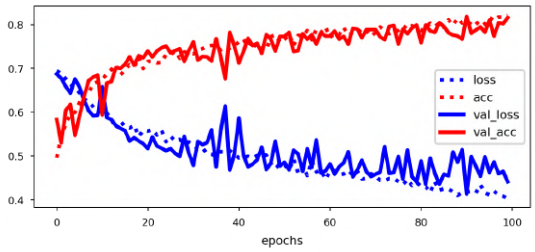
```
datagen = ImageDataGenerator(
      rotation_range=40,      # Rotate image up to 40 degrees
      width_shift_range=0.2, # Shift image left-right up to 20% of
image width
      height_shift_range=0.2,# Shift image up-down up to 20% of
image height
      shear_range=0.2,        # Shear (slant) the image up to 0.2
degrees
      zoom_range=0.2,         # Zoom in up to 20%
      horizontal_flip=True,  # Horizontally flip the image
      fill_mode='nearest')
```

Example

We also add Dropout before the Dense layer

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
                        input_shape=(150, 150, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dropout(0.5))
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

(Almost) no more overfitting!



Interpreting the model

- Let's see what the convnet is learning exactly by observing the intermediate feature maps
  - A layer's output is also called its *activation*
- We can choose a specific test image, and observe the outputs
- We can retrieve and visualize the activation for every filter for every layer

- Layer 0: has activations of resolution 148x148 for each of its 32 filters
- Layer 2: has activations of resolution 72x72 for each of its 64 filters
- Layer 4: has activations of resolution 34x34 for each of its 128 filters
- Layer 6: has activations of resolution 15x15 for each of its 128 filters

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_10 (Conv2D)           (None, 148, 148, 32)      896
_________________________________________________________________
max_pooling2d_8 (MaxPooling  (None, 74, 74, 32)        0
2D)
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 72, 72, 64)        18496
_________________________________________________________________
max_pooling2d_9 (MaxPooling  (None, 36, 36, 64)        0
2D)
_________________________________________________________________
conv2d_12 (Conv2D)           (None, 34, 34, 128)       73856
_________________________________________________________________
max_pooling2d_10 (MaxPoolin  (None, 17, 17, 128)       0
g2D)
_________________________________________________________________
conv2d_13 (Conv2D)           (None, 15, 15, 128)       147584
_________________________________________________________________
max_pooling2d_11 (MaxPoolin  (None, 7, 7, 128)         0
g2D)
_________________________________________________________________
flatten_2 (Flatten)          (None, 6272)              0
_________________________________________________________________
dropout (Dropout)            (None, 6272)              0
_________________________________________________________________
dense_4 (Dense)              (None, 512)               3211776
_________________________________________________________________
dense_5 (Dense)              (None, 1)                 513
=================================================================
Total params: 3,453,121
Trainable params: 3,453,121
Non-trainable params: 0
_________________________________________________________________
```

- To extract the activations, we create a new model that outputs the trained layers
  - 8 output layers in total (only the convolutional part)
- We input a test image for prediction and then read the relevant outputs

```
layer_outputs = [layer.output for layer in model.layers[:8]]
activation_model = models.Model(inputs=model.input,
outputs=layer_outputs)
activations = activation_model.predict(img_tensor)
```

Output of the first Conv2D layer, 3rd channel (filter):

- Similar to a diagonal edge detector
- Your own channels may look different

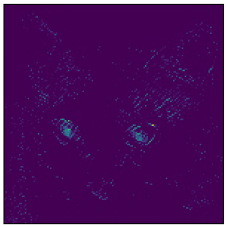
Input image

Activation of filter 6

Output of filter 16:

- Cat eye detector?

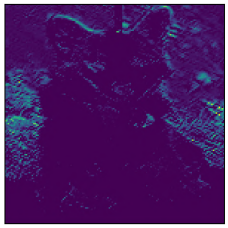
Input image



Activation of filter 25

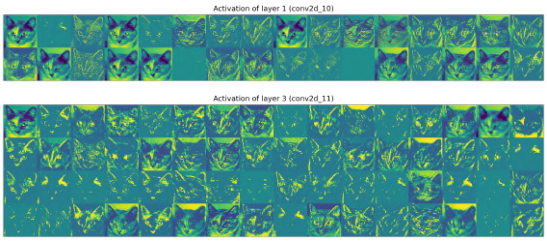
The same filter responds quite differently for other inputs (green detector?).

Input image

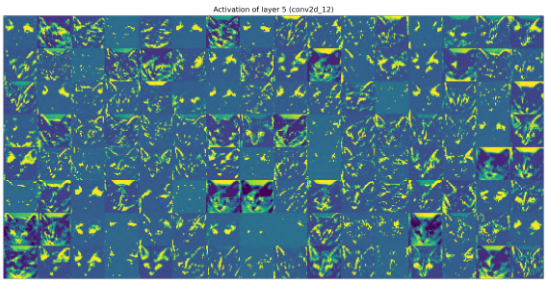


Activation of filter 25

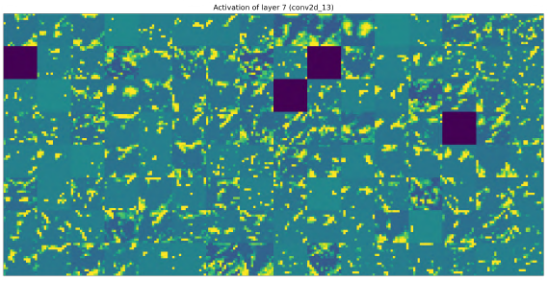
- First 2 convolutional layers: various edge detectors



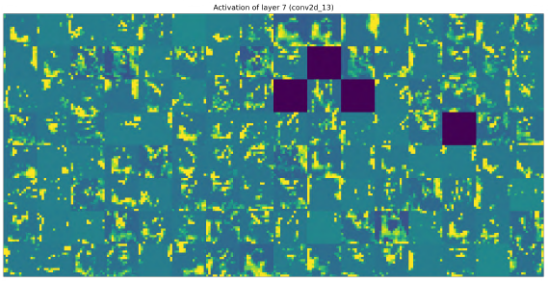
- 3rd convolutional layer: increasingly abstract: ears, eyes



- Last convolutional layer: more abstract patterns
- Empty filter activations: input image does not have the information that the filter was interested in

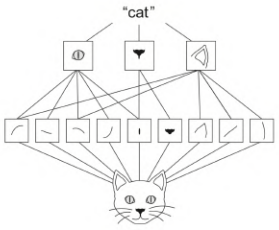


- Same layer, with dog image input
    - Very different activations



Spatial hierarchies

- Deep convnets can learn *spatial hierarchies* of patterns
    - First layer can learn very local patterns (e.g. edges)
    - Second layer can learn specific combinations of patterns
    - Every layer can learn increasingly complex *abstractions*



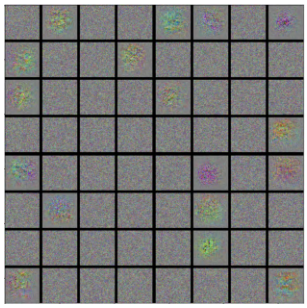
Visualizing the learned filters

- The filters themselves can be visualized by finding the input image that they are maximally responsive to
- *gradient ascent in input space* : start from a random image, use loss to update the pixel values to values that the filter responds to more strongly

```
from keras import backend as K
    input_img = np.random.random((1, size, size, 3)) * 20 + 128.
    loss = K.mean(layer_output[:, :, :, filter_index])
    grads = K.gradients(loss, model.input)[0] # Compute gradient
    for i in range(40): # Run gradient ascent for 40 steps
        loss_v, grads_v = K.function([input_img], [loss, grads])
        input_img_data += grads_v * step
```

- Learned filters of second convolutional layer
- Mostly general, some respond to specific shapes/colors

- Learned filters of last convolutional layer
- More focused on center, some vague cat/dog head shapes

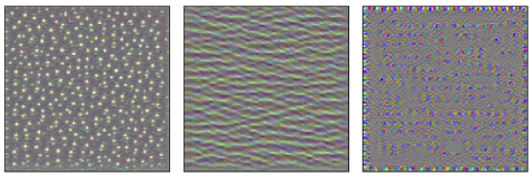


Let's do this again for the `VGG16` network pretrained on `ImageNet` (much larger)

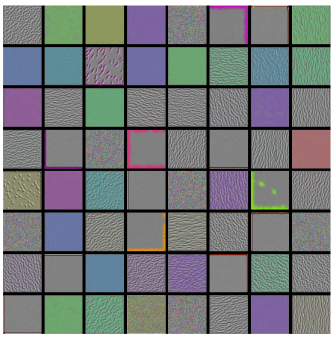
```
model = VGG16(weights='imagenet', include_top=False)
```

```
Model: "vgg16"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, None, None, 3)]   0
block1_conv1 (Conv2D)        (None, None, None, 64)    1792
block1_conv2 (Conv2D)        (None, None, None, 64)    36928
block1_pool (MaxPooling2D)   (None, None, None, 64)    0
block2_conv1 (Conv2D)        (None, None, None, 128)   73856
block2_conv2 (Conv2D)        (None, None, None, 128)   147584
block2_pool (MaxPooling2D)   (None, None, None, 128)   0
block3_conv1 (Conv2D)        (None, None, None, 256)   295168
block3_conv2 (Conv2D)        (None, None, None, 256)   590080
block3_conv3 (Conv2D)        (None, None, None, 256)   590080
block3_pool (MaxPooling2D)   (None, None, None, 256)   0
block4_conv1 (Conv2D)        (None, None, None, 512)   1180160
block4_conv2 (Conv2D)        (None, None, None, 512)   2359808
block4_conv3 (Conv2D)        (None, None, None, 512)   2359808
block4_pool (MaxPooling2D)   (None, None, None, 512)   0
block5_conv1 (Conv2D)        (None, None, None, 512)   2359808
block5_conv2 (Conv2D)        (None, None, None, 512)   2359808
block5_conv3 (Conv2D)        (None, None, None, 512)   2359808
block5_pool (MaxPooling2D)   (None, None, None, 512)   0
=================================================================
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
_________________________________________________________________
```

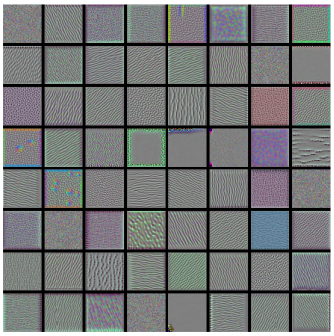
- Visualize convolution filters 0-2 from layer 5 of the VGG network trained on ImageNet
- Some respond to dots or waves in the image



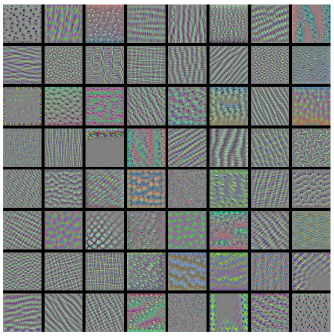
First 64 filters for 1st convolutional layer in block 1: simple edges and colors



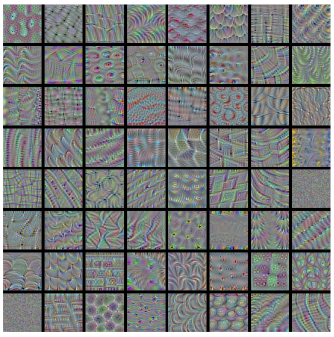
Filters in 2nd block of convolution layers: simple textures (combined edges and colors)



Filters in 3rd block of convolution layers: more natural textures



Filters in 4th block of convolution layers: feathers, eyes, leaves,...



Visualizing class activation

- We can also visualize which part of the input image had the greatest influence on the final classification
  - Helpful for interpreting what the model is paying attention to
- *Class activation maps* : produce heatmap over the input image
  - Take the output feature map of a convolution layer (e.g. the last one)
  - Weigh every filter by the gradient of the class with respect to the filter

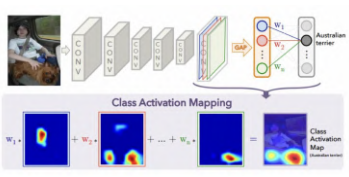
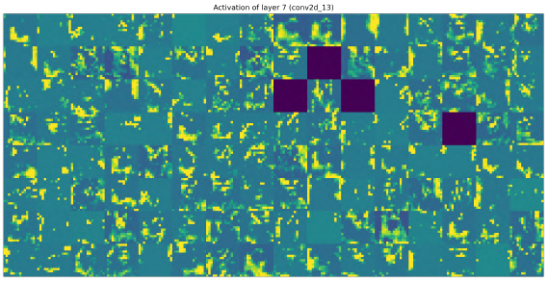


Illustration (cats vs dogs)

- These were the output feature maps of the last convolutional layer
    - These are flattened and fed to the dense layer
- Compute gradient of the 'cat' node output wrt. every filter output (pixel) here
    - Average the gradients per filter, use that as the filter weight
- Take the weighted sum of all filter maps to get the class activation map



More realistic example:

- Try VGG (including the dense layers) and an image from ImageNet

```
model = VGG16(weights='imagenet')
```

Preprocessing

- Load image
- Resize to 224 x 224 (what VGG was trained on)
- Do the same preprocessing (Keras VGG utility)

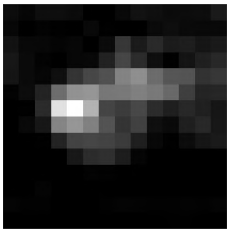
```
from keras.applications.vgg16 import preprocess_input
img_path = '../images/10_elephants.jpg'
img = image.load_img(img_path, target_size=(224, 224))
x = image.img_to_array(img)
x = np.expand_dims(x, axis=0) # Transform to batch of size (1, 224, 224, 3)
x = preprocess_input(x)
```

- Sanity test: do we get the right prediction?

```
preds = model.predict(x)
```

```
Predicted: [('n02504458', 'African_elephant', 0.90988594), ('n01871265', 'tusker', 0.085724816), ('n02504013', 'Indian_elephant', 0.0043471307)]
```

Visualize the class activation map

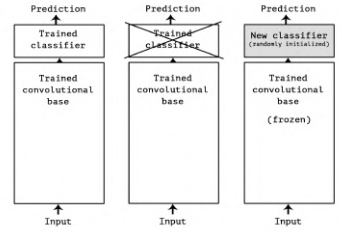


Superimposed on the original image

Class activation map

Using pretrained networks

- We can re-use pretrained networks instead of training from scratch
- Learned features can be a generic model of the visual world
- Use *convolutional base* to contruct features, then train any classifier on new data
- Also called *transfer learning* , which is a kind of *meta-learning*



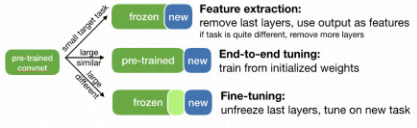
- Let's instantiate the VGG16 model (without the dense layers)
- Final feature map has shape (4, 4, 512)

```
conv_base = VGG16(weights='imagenet', include_top=False,
input_shape=(150, 150, 3))
```

```
Model: "vgg16"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 150, 150, 3)]     0
block1_conv1 (Conv2D)        (None, 150, 150, 64)      1792
block1_conv2 (Conv2D)        (None, 150, 150, 64)      36928
block1_pool (MaxPooling2D)   (None, 75, 75, 64)        0
block2_conv1 (Conv2D)        (None, 75, 75, 128)       73856
block2_conv2 (Conv2D)        (None, 75, 75, 128)       147584
block2_pool (MaxPooling2D)   (None, 37, 37, 128)       0
block3_conv1 (Conv2D)        (None, 37, 37, 256)       295168
block3_conv2 (Conv2D)        (None, 37, 37, 256)       590080
block3_conv3 (Conv2D)        (None, 37, 37, 256)       590080
block3_pool (MaxPooling2D)   (None, 18, 18, 256)       0
block4_conv1 (Conv2D)        (None, 18, 18, 512)       1180160
block4_conv2 (Conv2D)        (None, 18, 18, 512)       2359808
block4_conv3 (Conv2D)        (None, 18, 18, 512)       2359808
block4_pool (MaxPooling2D)   (None, 9, 9, 512)         0
block5_conv1 (Conv2D)        (None, 9, 9, 512)         2359808
block5_conv2 (Conv2D)        (None, 9, 9, 512)         2359808
block5_conv3 (Conv2D)        (None, 9, 9, 512)         2359808
block5_pool (MaxPooling2D)   (None, 4, 4, 512)         0
=================================================================
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
_________________________________________________________________
```

Using pre-trained networks: 3 ways

- Fast feature extraction (similar task, little data)
  - Call `predict` from the convolutional base to build new features
  - Use outputs as input to a new neural net (or other algorithm)
- End-to-end tuning (similar task, lots of data + data augmentation)
  - Extend the convolutional base model with a new dense layer
  - Train it end to end on the new data (expensive!)
- Fine-tuning (somewhat different task)
  - Unfreeze a few of the top convolutional layers, and retrain
    - Update only the more abstract representations



Fast feature extraction (without data augmentation)

- Run every batch through the pre-trained convolutional base

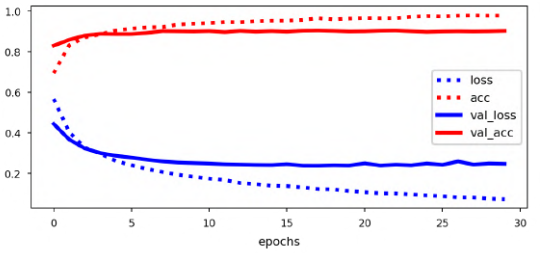
```
generator = datagen.flow_from_directory(dir, target_size=(150, 150),
        batch_size=batch_size, class_mode='binary')
for inputs_batch, labels_batch in generator:
    features_batch = conv_base.predict(inputs_batch)
```

- Build Dense neural net (with Dropout)
- Train and evaluate with the transformed examples

```
model = models.Sequential()
model.add(layers.Dense(256, activation='relu', input_dim=4 * 4 *
512))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(1, activation='sigmoid'))
```

- Validation accuracy around 90%, much better!
- Still overfitting, despite the Dropout: not enough training data

```
Max val_acc 0.90500003
```



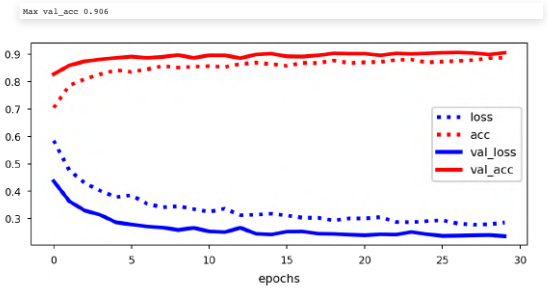
Fast feature extraction (with data augmentation)

- Simply add the Dense layers to the convolutional base
- *Freeze* the convolutional base (before you compile)
    - Without freezing, you train it end-to-end (expensive)

```
model = models.Sequential()
model.add(conv_base)
model.add(layers.Flatten())
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
conv_base.trainable = False
```

We now get about 90% accuracy again, and very little overfitting

```
Max val_acc 0.906
```

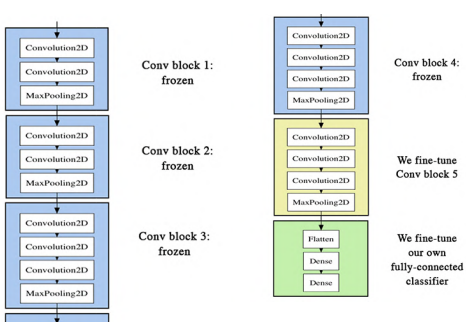


Fine-tuning

- Add your custom network on top of an already trained base network.
- Freeze the base network, but unfreeze the last block of conv layers.

```
for layer in conv_base.layers:
    if layer.name == 'block5_conv1':
        layer.trainable = True
    else:
        layer.trainable = False
```

Visualized

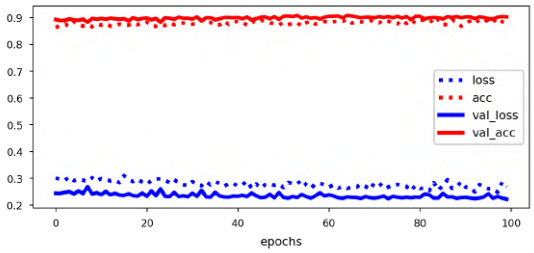


- Load trained network, finetune
  - Use a small learning rate, large number of epochs
  - You don't want to unlearn too much: *catastrophic forgetting*

```
model = load_model(os.path.join(model_dir,
'cats_and_dogs_small_3b.h5'))
model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(lr=1e-5),
              metrics=['acc'])
history = model.fit(
      train_generator, steps_per_epoch=100, epochs=100,
      validation_data=validation_generator,
      validation_steps=50)
```

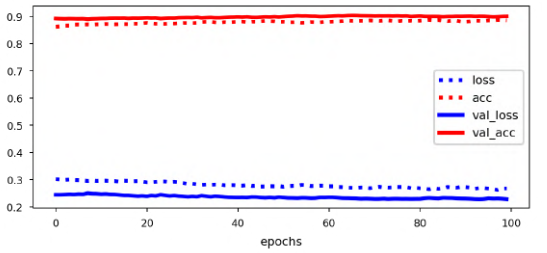
Almost 95% accuracy. The curves are quite noisy, though.

```
Max val_acc 0.90800005
```



- We can smooth the learning curves using a running average

```
Max val_acc 0.9039536851123335
```



Take-aways

- Convnets are ideal for attacking visual-classification problems.
- They learn a hierarchy of modular patterns and concepts to represent the visual world.
- Representations are easy to inspect
- Data augmentation helps fight overfitting
- You can use a pretrained convnet to build better models via transfer learning