TODO: 5
Different tools 5
Imports/Shorthand 6
Different Types of Variables 7
One Hot encoding 7
Machine Learning - Introduction 8
Data science questions 8
Variable types: 16
Probabilistic Modelling 16
Neural networks 16
Kernel methods 17
Tree-based models 18
Classification problem - example/Assessment task 19
Creating a training set and a test set 21
Scatter plot 21
k-Nearest Neighbours 22
Building the model 24
Making predictions 24
Evaluating the model 25
Entire training and evaluating procedure: recap 25
Confusion matrix 26
Supervised Learning (Chapter 2) 26
k-Nearest Neighbours 28
Linear Models 33
Linear Regression (ordinary least squares) 35
Ridge Regression 37

Lasso **41**

Elastic Net **44**

Evaluation metrics for regression **44**

THERE ARE MANY DIFFERENT LINEAR MODELS FOR CLASSIFICATION **44**

Logistic regression **45**
- Odds Ratio 46
- class_weight 46

Linear Support Vector Machines (linear SVMs) **46**

Naive Bayes Classifiers **53**

Evaluation Metrics and Scoring **55**

DummyClassifier **55**

Confusion matrices **56**

taking uncertainty into account **59**

Precision-recall curves + ROC curves **62**

ROC and AUC **63**

Evaluation metrics for Multiclass classification **66**

Cross-validation (Chapter 3) **67**
- KFold 69
- Leave-one-out cross-validation 70
- Shuffle-split cross-validation 70

Cross-validation with groups **71**

Grid Search **72**
- Simple Grid Search 72

Grid Search + Cross Validation **74**
- Analysing the result 76
- Conditional parameters 76
- Nested cross-validation 77
- Parallelising cross-validation + grid-search 77

Standardisation **78**

RandomSearchCV **78**

Nested CV **78**

Decision Trees (Week 4) **80**
Analyzing decision trees: 84
Feature importance 85
DecisionTreeRegressor/Tree Based Regression models 86

Ensembles of Decision Trees **86**
Random Forests 87
Analyzing random forests: 88
Gradient boosted regression trees (gradient boosting machines) 90

Kernelized Support Vector Machines (Week 5) **93**
parameters 100
Preprocessing data for SVMs 102
One-class Support Vector Machines 105

Artificial Neural Networks (Chapter 6) **108**
Data Representations 113
Manipulating Tensors 115
Data batches 116
(Vector data 116
Tensor operations 116
Keras 121
Neurons 122
Combining Neurons into a Neural Network 125
Training a neural network 127

Unsupervised Learning (Chapter 7) **133**
Preprocessing/Scaling 134
Dimensionality Reduction/Feature Extraction/Manifold Learning 135
Eigenfaces for feature extraction: 152
Autoencoders 157
Non-Negative Matrix Factorization (NMF) 165
Manifold Learning with t-SNE 168

Clustering (still Chapter 7) **172**
k means 176
Hierarchical clustering: 180
k-Means Clustering 187
Failure of k-means 189
Vector quantization, k-means as decomposition 190

Agglomerative Clustering 194
DBSCAN 197

Evaluating clustering algorithms (Chapter 7) 200

Sequential Data Models (Chapter 8) 202
Hidden Markov Model (HMM) 202
LSTM Networks 212
Variants of LSTM 217

Natural Language Processing (Chapter 9) 219
Document Vectors 220
Word embeddings 222

What justifies that the embedding of the word 'dog' should be closer to that of 'cat' than that of 'truck'? 228
Named-entity recognition 228
Topic modelling 235

TODO:

- you have to go through these notes and edit them repeatedly

Different tools

Python →
- lingua franca for data science applications
- many libraries → vast toolbox
- can interact directly with the code (jupyter notebook)
- GUIs + web services

Anaconda →
- comes with NumPy, SciPy, matplotlib, pandas, IPython, Jupyter Notebook, and scikit-learn

scikit-learn →
- open source project
- https://scikit-learn.org/stable/user_guide.html
- https://scikit-learn.org/stable/index.html
- depends on 2 other packages numpy and scipy (+matplotlib for plotting and interactive development)

jupyter notebook →
- open source
- “interactive environment for running code in the browser”

numpy →
- fundamental package for scientific computing in Python
- functionality for multidimensional arrays, high-level mathematical func- tions such as linear algebra operations and the Fourier transform, and pseudorandom number generators
- *the NumPy array is a fundamental data structure in scikit-learn*
- ndarray class, a multidimensional (n-dimensional) array

SciPy →
- collection of functions for scientific computing in Python

matplotlib →

- primary scientific plotting library in Python

pandas →
- Python library for data wrangling and analysis
- Series and DataFrames
- DataFrame ⇒ a table (similar to an Excel spreadsheet)

Imports/Shorthand

```
import numpy as np
import pandas as pd

import mglearn
import matplotlib.pyplot as plt
```

Datasets:

```
import sklearn.datasets
```
(then refer to the dataset by sklearn.datasets.name_of_dataset)
or
```
from sklearn.datasets import name_of_dataset
```
refer to dataset with just “name_of_dataset”

- iris dataset
- forge dataset → synthetic wave dataset
- Real world:
    - Wisconsin Breast Cancer dataset (cancer)

```
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()
```

    - boston housing dataset

```
from sklearn.datasets import load_boston
boston = load_boston()
```

Different Types of Variables

IMPORTANT:

continuous:

examples:

- age
- weight….

you can find:

- mean is in this world
- correlation is computed using these variables

categorical:

- sex
- eye colour
- ethnicity
- NEVER CALCULATE
    - MEAN, variance (PCA) or corr()

One Hot encoding

encoding = representing each piece of data in a way that the computer can understand (converting to computer code)
produces a vector with equal length to the number of categories
if a data point belongs to the nth category, then all the components of the vector are the value 0 except for the nth component, which is assigned a value of 1

```
[0,1,0,0] // female
[0,1,0,0]
[0,1,0,0]
[0,1,0,0]
[1,0,0,0] // male
[1,0,0,0]
[1,0,0,0]
[0,0,1,0] // gender-neutral
[0,0,0,1] // other
[0,0,0,1]
```

Machine Learning - Introduction

- What is Machine Learning (ML) and Data Mining (DM)?

Data science questions

<u>There are 3 categories of data science: there are only 3 things you can do with data:</u>

<u>the difference between the three categories is the question</u>

1. descriptive
    a. how many people over 65 have strokes? incidence (per year), prevalence (percentage)
    b. correlation
    c. risk
    d. clustering ⇒ unsupervised learning (main technique, others like PCA, LDA)
2. predictive/classification
    a. predictions
    b. don't really care about cause (e.g. previous hospitalisations, multiple hospital visits)
        i. logistic regression
        ii. linear regression
        iii. CNNs/deep learning

iv. KNN
v. trees (RF, gradient boosted trees)
vi. SVM

3. Causal inference = explanatory models
    a. epidemiology, risk
    b. looking for a cause
    c. treatment A vs treatment B (as in an actual RCT falls into this category)
    d. advantage over traditional RCT ⇒ *you can emulate a RCT (observational control trial)*
    e. whenever you can think of a RCT you can think of an emulated trial using data (rural vs urban, hip replacement vs none…)
        i. logistic regression → people use this for RCTs remember
        ii. BART - based on random forests
        iii. RF - causal inference
        iv. REINFORCEMENT LEARNING (e.g. neural networks)

t_0 = when the patients are randomised, divided into treatment and intervention groups, when the treatment is actually administered (e.g. drug A is actually injected into their veins) →

red = machine learning techniques you can use to answer a question in any of the three categories

PREDICTION ⇒

$\hat{Y} = \hat{f}(X)$ (where Y is the output/prediction, X is the input, the function f is the model - black box)

inference ⇒

the way that Y is affected as $X_1 \ldots X_P$ change

predictors associated with the response, the relationship between the response and the predictor,

read the paper

shallow learning → input -> output

- one layer
- we have input and we have output

deep learning = multiple layers

input → layer → layer → layer → output

but again it can be supervised, unsupervised (embedding), reinforcement (GAN)

if you're doing predictions w/ one hot encoding drop one of the columns (for linear or logistic regression)

medical record goes to medical coders → given an ICD 10 code (moving to ICD 11 soon) ⇒ starts with a letter (organ) ⇒ have a window of 28 days, ⇒ goes to NSW health (for billing)

ICD10s happen 28 days after discharge! you don't have available at the point of discharge

Artificial intelligence = the effort to automate intellectual tasks normally performed by humans

- encompasses machine learning
- which in turn encompasses deep learning
- also includes processes that don't involve any learning (e.g. automatic chess playing machine) = symbolic AI
- classical programming → humans input rules and data to be processed according to these rules and out come answers

Lady Lovelace → computers are incapable of originality or learning

Data mining =

- taking methods from statistics, computer science and database analytics and applies them to large data sets in order to extract useful information

Machine learning =
an approach to data mining
ML gives computers the ability to learn without being explicitly programmed
→ computer go beyond what we order it to perform and *learn on its own to perform a specified task*
automatically learn rules by looking at data

- humans input data as well as the answers expected from the data, and out come rules. rules can then be applied to new data to produce original answers/ a machine learning model transforms its input data into meaningful outputs (having 'learned' from exposure to known examples of inputs and outputs). = *learning from experience*
- trained rather than explicitly programmed.
- presented with many examples relevant to a task, and it finds statistical structure in these examples that eventually allows the system to come up with rules for automating the task

'Extracting knowledge from data"
Machine learning methods are ubiquitous → youtube recommendations

- Getting computers to perform tasks without explicit instructions/ Giving computers systems the ability to learn and improve from experience without being explicitly programmed.
- Algorithms learn from previous experience
- Looking for patterns and relationships

Most popular and successful subfield of AI

Statistical learning refers to a vast set of tools for understanding data

- What problems can ML solve? (Supervised Learning, Unsupervised Learning and Reinforcement Learning)

One setting:

Supervised learning = learning to predict
FOR EACH OBSERVATION OF TEH PREDICTOR MEASUREMENTS (x_1, x_2….x_n) there is AN ASSOCIATED RESPONSE MEASUREMENT
usually we are interested in prediction → once it has learnt, it is able to make predictions
we are always interested in finding a function y = f(x) where x is the input ! and y is the output! !!!!

- remember: the input X is a two-dimensional array (a NumPy array, two dimensions required by scikit-learn) → of features, with one row per data point and one column per feature
- array y is a one dimensional array, containing one class label, an integer (classification)

Think: "learning a task under supervision (e.g. Maths class) → someone is there to tell you whether you got the answer correct"

data is always labelled → some input and some label

- having a full set of labelled data while training an algorithm
- need: input data points, examples of expected output, a way to measure whether the algorithm is doing a good job

- supervised statistical learning involves building a statistical model for predicting, or estimating, an output based on one or more inputs.
- Machine learning algorithms that automate decision making processes by generalizing from known examples
- The user provides the algorithm with pairs of inputs and desired outputs, and the algorithm finds a way to produce the desired output given an input
- The algorithm is able to create an output for an input it has never seen before without any help from a human
- Machine learning algorithms that learn from input/output pairs are called "supervised learning algorithms" because a teacher provides supervision to the algorithms in the form of desired outputs for each example they learn from
- I.e. you create a dataset that includes the desired outcome

- Examples:
    - Identifying the zip code from handwritten digits on an envelope
        - (input is a scan of handwriting, the output is the actual digits in the zip code)
    - Determining whether a tumour is benign or malignant based on a medical image:
        - Input - image, output = whether the tumour is benign
    - Detecting fraudulent activity in credit card transactions:
        - Input = record of the credit card transaction, output = whether it is likely to be fraudulent or not
    - speech recognition (inputs: sound files, outputs: “malignant cancer tumour”, “benign cancer tumour”)
    - patient admission data (input data: number of surgeries, length of stay etc ⇒ these are the x), then the y (output data: is the patient going to be readmitted “yes” or “no”) ⇒ want to predict whether a patient will be readmitted or not
    - automate the task of tagging your vacation pictures, you could present a machine-learning system with many examples of pictures already tagged by humans, and the system would learn statistical rules for associating specific pictures to specific tags
    - duck photos → ducks and not ducks
        - dataset of animal images and their corresponding labels
        - the label tells the model which photos are ducks and are not ducks
        - we build the predictive/classification model using these labelled images (training set)
        - once the model is built, someone presents the model a completely new image
        - when shown this image, the model compares it to the training examples to predict the correct label (which images are ducks)
    - E.g. spam classification: the user provides the algorithm with a large number of emails (which are the input), together with information about whether any of these emails are spam (which is the desired output). Given a new email, the algorithm will then produce a prediction as to whether the new email is spam.

supervised machine learning is the most commonly used and successful
used when we want to predict a certain outcome from a given input AND we have examples of input/output pairs (make accurate predictions for new, never-before-seen data)
we build a machine learning model from these input/output pairs (the training set)

Unsupervised learning = finding patterns and clusters in data
VECTOR OF MEASUREMENTS (x_1,
we don't provide labels, we only have the input. We want to know what is going on with the data, what kind of patterns we have, what kind of clusters.
We want to distinguish between different patterns but we have so much data we can't do it ourselves.
clustering algorithm
Unsupervised algorithms = another type of algorithm
Only the input data is known, and no known output data is given to the algorithm
Usually harder to understand and evaluate

- unsupervised statistical learning, there are inputs but no supervising output; nevertheless we can learn relationships and structure from such data
- Examples
    - Identifying topics in a set of blog posts
    - Detecting abnormal access patterns to a website
    - clustering: algorithm learns the inherent groupings in the data
    - dimensionality reduction algorithms: : reducing the number of variables or features in a dataset for better understanding or to discover patterns that cannot be seen in high dimensions
    - anomaly detection: unusual patterns or unusual data points
    - malignant tumour: able to find clusters of unusual cells

searching for useful representations of some input data within a predefined space of possibilities. But this time, we are not using guidance from a feedback signal or labelled data. The machine learning algorithm then attempts to automatically model the underlying structure or distribution in the data by extracting useful features and analysing its structure.
there are no correct answers

<u>Reinforcement learning</u> = enabling a software agent/'artificial agent' to learn by trial and error using feedback from its own actions and experiences
exploration → try a new activity
exploitation → try already known activities
reward function = reward system
resultant action
want to take actions to maximise reward
don't need to provide input/output pairs
sub-optimal actions don't have to be explicitly corrected

- trying to make a machine learn through a reward system (e.g. learn a checkers game)
- machine is going to learn to make a series of actions (how to move the checkers)

- Data representation: it's important to have a representation of your input data that a computer can understand (Data, Tables, Continuous and Categorical features).

You have to represent your input data in a way that a computer can understand:

- Building a good representation of your data is called “feature extraction” or “feature engineering”

Each data point that you want to reason about is a row (each email, each customer, each transaction)
Each property that describes a data point is a column (age, gender, when they created an account, how often they have bought from your shop)

Each entity (row) here is called a “sample” or “data point” in machine learning, while the columns, the properties that describe these entities, are called “features”

data visualisation is step 2 of the machine learning and data mining work flow
must inspect the data before before building a machine learning model
best way is to visualise it → graphing with seaborn, matplotlib and plotnine (ggplot)

- What is the difference between ML and Classic Statistics (Explanatory, Causality Models)?

- How can we understand the data we are working with and how it relates to the problems we want to solve?

- Machine Learning and Data Mining Work Flow. Iterative data driven processes, in which the **data and the research question** drive the analysis

Modelling ⇒ when you apply the machine learning algorithm

whole process = data mining, machine learning is just inside step 4

Variable types:

Categorical ⇒ ‘yes’ or ‘no’

histogram = distribution of continuous numeric variable by grouping ranges of the data into bins
bar charts ⇒ plot categorical data

Probabilistic Modelling

One of the earliest forms of machine learning.
best known algorithm = the Naive Bayes algorithm

- machine learning classifyer
- based on applying Bayes theorem (18th century)
- assumes that all the features of the input data are independent (‘naive’)

closely related model to: logistic regression
- also a classification algorithm (not a regression algorithm = misnomer)

Neural networks

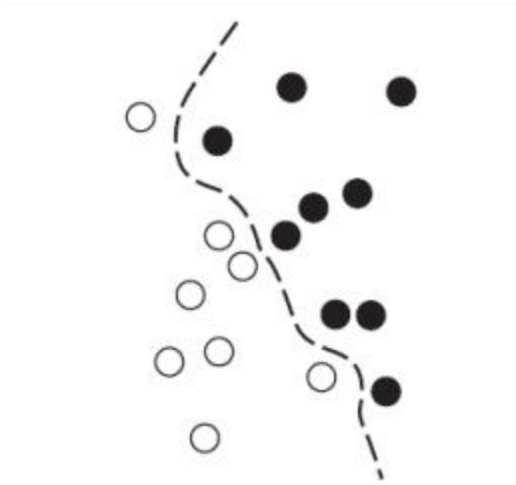
Backpropagation algorithm → a way to train chains of parametric operations using gradient descent optimization

shunned in 2010
deep neural networks ⇒ deep learning

Kernel methods

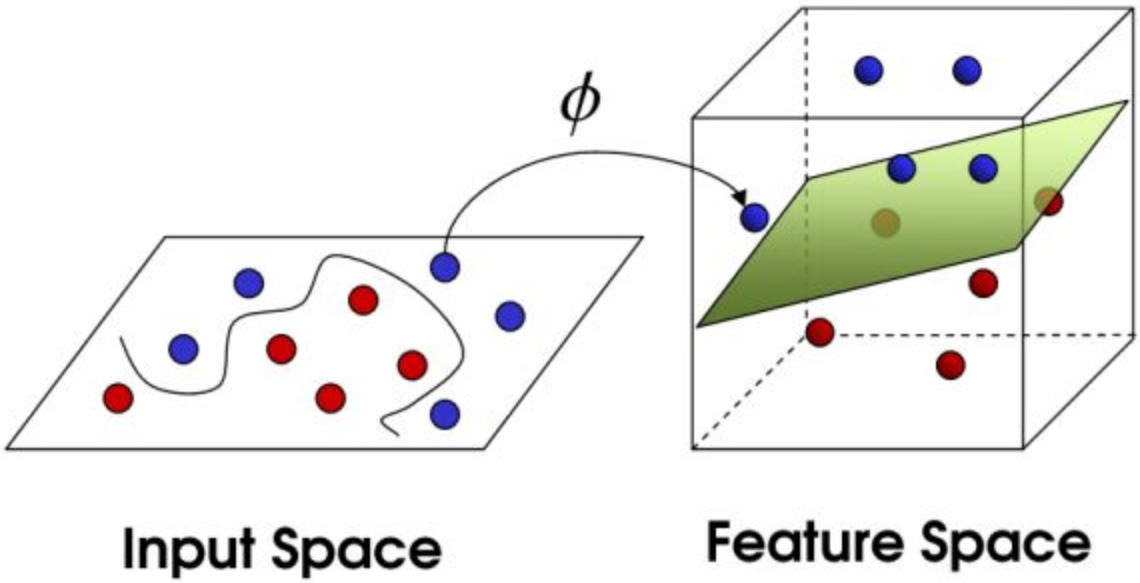
= a group of classification algorithms

best known = support vector machine (SVM)
SVMs aim at solving classification problems by finding good decision boundaries between two sets of points belonging to two different categories
decision boundary = a line separating your training data into two spaces corresponding to two categories



the data is “mapped” to a high-dimensional representation, decision boundary “is expressed as a hyperplane”
a good decision boundary/separation hyperplane maximises the distance between the hyperplane and the closest data points from each class

to classify new data points, simply check what side of the decision boundary they fall on

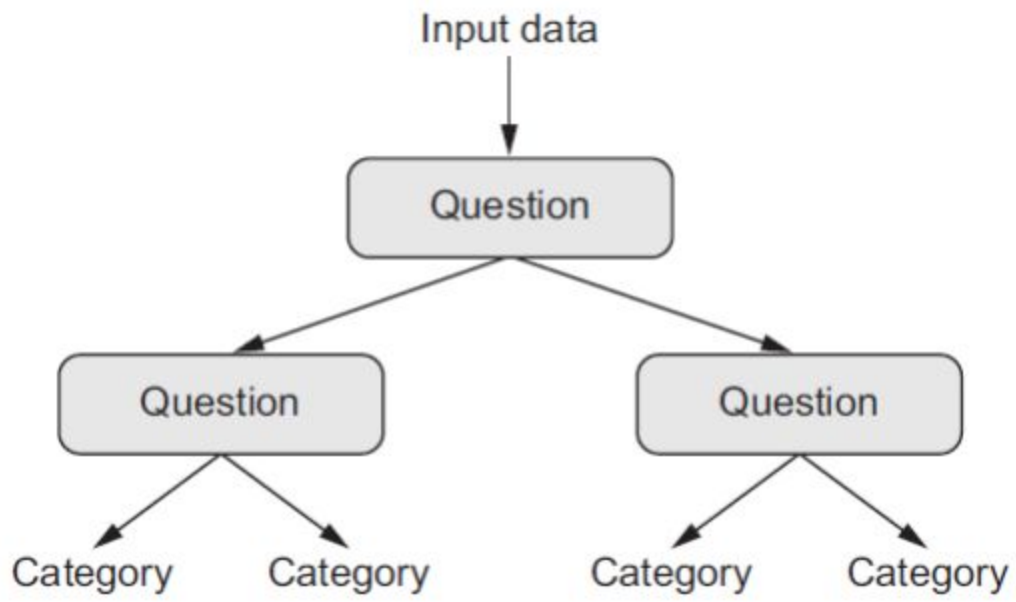


in 2D = straight line

mapping to a high dimensional representation is *computationally intractable* → kernel trick = don't explicitly compute the coordinates of the points in space, just compute the distance between pairs of points in that space (efficient)
kernel function → maps any two points in the initial space to the *distance between these points in the target representation space* (hence you don't have to explicitly compute the new representation)
kernel functions are "crafted by hand" → as in you design them, only the separation hyperplane is learned

- hard to scale to large datasets, shallow method, feature engineering

Tree-based models



decision trees
flowchart like

Random Forest algorithm ⇒ large number of specialised decision trees, ensembling their outputs

gradient boosting machine ⇒ ensembles weak prediction models (usually decision trees)
gradient boosting = iteratively training new models that address the weak points of previous models

- easy to visualise and interpret
- preferred to kernel methods

Classification problem - example/Assessment task

Classification problem: predicting one of several options (e.g. species of iris)

possible outputs (e.g. different species) are called “classes”

using a dataset of irises that’s included in scikit-learn

```
from sklearn.datasets import load_iris
iris_dataset = load_iris()
```

```
iris_dataset['data']
```

lists all the data (as a really big array, with each data point (i.e. flower) being an array containing multiple values.

```
iris_dataset['data'].shape
```

Output: (150, 4)
the dataset has 150 entries (measurements for 150 different flowers) with 4 measurements each (you can see what they are in feature names)

```
iris_dataset['feature_names']
```

Output:

['sepal length (cm)',
 'sepal width (cm)',
 'petal length (cm)',
 'petal width (cm)']

```
iris_dataset['target']
```

Outputs an array 150 elements long containing 0s, 1s and 2s. Each of these numbers represents the ‘output’ - i.e. what type of flower it is.

(in the assessment: ‘0’ = no readmission in 30 days, ‘1’ = readmission in 30 days)

```
iris_dataset['target_names']
```

Output:
array(['setosa', 'versicolor', 'virginica'], dtype='<U10')

Tells you what the numbers represent: #0 means setosa, 1 means versicolor, and 2 means virginica

Creating a training set and a test set

We split our dataset into a training set to build our model, and a test set to evaluate how well our model will generalise to new, previously unseen data

We decide which method fits our needs the best by using test set

scikit-learn contains train_test_split function. This function extracts 75% of the rows in the data as the training set, together with the corresponding labels for this data. The remaining 25% of the data, together with the remaining labels, is declared as the test set.

data is usually denoted with a capital X, while labels are denoted by a lowercase y. This is inspired by the standard formulation y = f(x) in mathematics, where x is the input to a function and y is the output. Following more conventions from mathematics, we use a capital X because the data is a two-dimensional array (a matrix) = required by scikit-learn and a lowercase y because the target is a one-dimensional array (a vector).

```
X_train, X_test, y_train, y_test = train_test_split(
 iris_dataset['data'], iris_dataset['target'], random_state=0)
```

shuffles the dataset using a
pseudorandom number generator. If we just took the last 25% of the data as a test set, all the data points would have the label 2, as the data points are sorted by the label (see the output for iris['target'] shown earlier).

Scatter plot

Before building a machine learning model you should inspect your data.

- see if task is solvable without ML
- desired information might not be in the data
- looking for abnormalities and peculiarities

A scatter plot of the data puts one feature along the x-axis and another along the y-axis, and draws a dot for each data point.

Computer screens have only two dimensions, which allows us to plot only two (or maybe three) features at a time

- workaround = *pair plot,* looks at all possible pairs of features

```
import pandas as pd
import mglearn

# create dataframe from data in X_train
# label the columns using the strings in iris_dataset.feature_names
iris_dataframe = pd.DataFrame(X_train, columns=iris_dataset.feature_names)
# create a scatter matrix from the dataframe, color by y_train
grr = pd.plotting.scatter_matrix(iris_dataframe, c=y_train, figsize=(15,
15), marker='o',
 hist_kwds={'bins': 20}, s=60, alpha=.8, cmap=mglearn.cm3)
```

k-Nearest Neighbours

To build a machine learning model - there are many classification algorithms in scikit-learn you can use (for a classification problem e.g. three class petals classification problem)

requires all variables to be numeric (convert categorical variables to numeric)

THE OUTPUT IS CLASS MEMBERSHIP
A new data point is classified by majority vote of its neighbours, assigned to the class most common among its k nearest neighbours If k = 2, then the object is simply assigned to the class of two nearest neighbour (Wikipedia).

(The number of neighours, **k** is what we call a parameter of the model (hyperparameter). A hyperparameter must be set by the user; it can also be selected via grid search. Grid-search is a way to find the best model, parametrized by a grid of parameters (Chapter 3).)

To build this model you only have to store the training dataset.

To make a prediction for a new data point, the algorithm finds the point in the training set that is closest to the new point.

- makes predictions for a new data point by considering its closest neighbour(s) in the training set

Then it assigns the label of this training point to the new data point.

The k in k-nearest neighbors signifies that instead of using only the closest neighbor to the new data point, we can consider any fixed number k of neighbors in the train- ing (for example, the closest three or five neighbors).
Then, we can make a prediction using the majority class among these neighbors.

All machine learning models in scikit-learn are implemented in their own classes, called Estimator classes.

The k-nearest neighbors classification algorithm is implemented in the KNeighborsClassifier class in the neighbors module.

Before we can use the model, we need to *instantiate* the class into an object.

This is when we will set any parameters of the model.

The most important parameter of KNeighborsClassifier is the number of neighbors (we set to 1)

```
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=1)
```

Now the knn object contains the algorithm that will be used to build the model from the training data, as well as the algorithm that will be used to make predictions on new data points.
(also holds information it has extracted from the dataset, but because we're only using 1st neighbour, this is just the dataset itself)

(in the assessment: we used n_neighbors = 6, 6 neighbours)

Building the model

We then built the model by calling the fit method, passing the training data (X_train) and training outputs (y_train) as arguments.

To build the model on the training set, call the fit method of the knn object → this method takes NumPy arrays for arguments, the first being the training data (X_train) and the second being the corresponding training labels (y_train)

```
knn.fit(X_train, y_train)
```

Output:
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski', metric_params=None, n_jobs=1, n_neighbors=1, p=2, weights='uniform')

the fit method modifies the knn object in place and returns it (shows us what parameters are used in creating the model)

Making predictions

we can now make predictions using this model on new data for which we do not know the correct labels

try testing new data ⇒ create a new NumPy array with random values for sepal length, sepal width, and petal length, petal width.
Must be a two-dimensional NumPy array, as scikit-learn expects 2D arrays for data

To actually make a prediction, you call the predict method of the knn object

```
prediction = knn.predict(X_new)
print("Prediction: {}".format(prediction))
print("Predicted target name: {}".format(
          iris_dataset['target_names'][prediction]))
```

Output:
Prediction: [0]
Predicted target name: ['setosa']

Evaluating the model

we evaluate the model using the score method (computing the accuracy of the model)

```
y_pred = knn.predict(X_test)
print("Test set predictions:\n {}".format(y_pred))
```

Test set predictions: [2 1 0 2 0 2 0 1 1 1 2 1 1 1 1 0 1 1 0 0 2 1 0 0 2 0 0 1 1 0 2 1 0 2 2 1 0 2]

```
print("Test set score: {:.2f}".format(np.mean(y_pred == y_test)))
```

Test set score: 0.97

Or you can use the score method of the knn object:

```
print("Test set score: {:.2f}".format(knn.score(X_test, y_test)))
```

Test set score: 0.97

test set accuracy = 0.97 or 97% correct predictions.

Entire training and evaluating procedure: recap

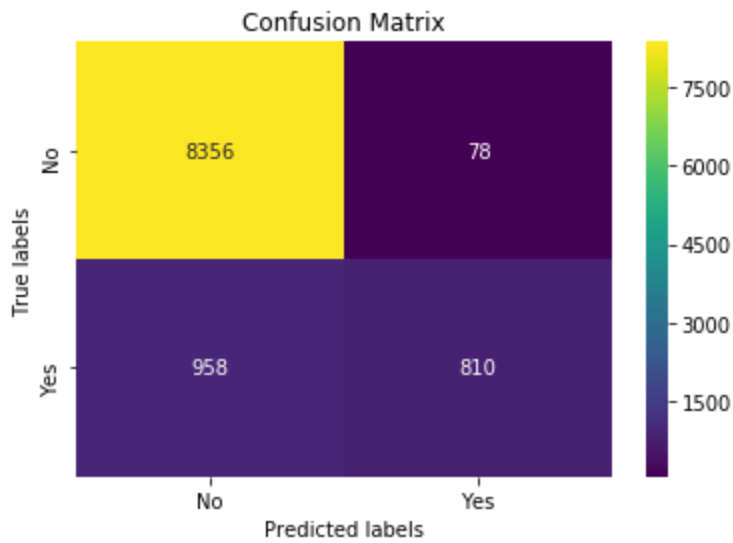
All you need is:

```
X_train, X_test, y_train, y_test = train_test_split(
```

```
     iris_dataset['data'], iris_dataset['target'], random_state=0)

knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(X_train, y_train)

print("Test set score: {:.2f}".format(knn.score(X_test, y_test)))
```

Chapter 1 book1 done!

Confusion matrix



shows how many 1 or YES readmissions are predicted by our model (y_pred) matched the truth (y_test)
= a punnet square, used to calculate sensitivity and specificity

Supervised Learning (Chapter 2)

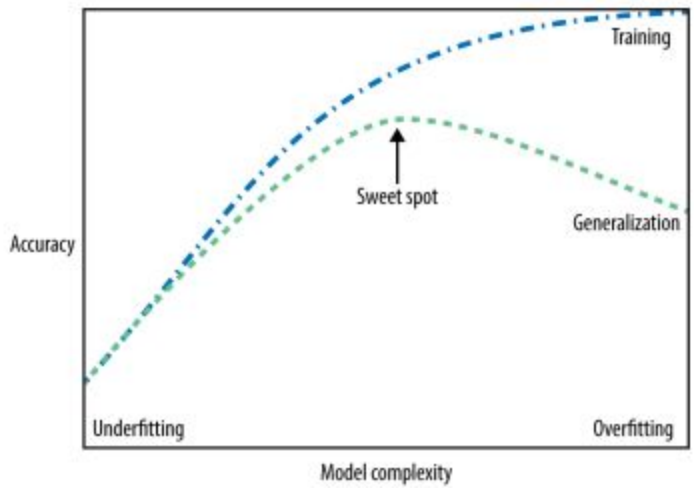
There are two (major) types of supervised machine learning problems:

- Classification
    - predict a *class* label
        - from predefined list of possibilities
    - no matter of degree
    - binary classification = distinguishing between two classes (spam or not spam). something that will give either a 1 or a 0
        - positive class = object of the study (e.g. positive = spam class)
        - e.g. predicting whether or not (YES/NO) a patient will be readmitted within 30 days
    - multiclass classification = classification between > 2 classes
    - e.g. classifying irises into one of three possible species, predicting language
- Regression
    - make a prediction of a variable that is continuous (a float) → an arbitrary number
    - if there is continuity between possible outcomes ⇒ the problem is a regression problem!
    - e.g. predicting income from education, age, neighbourhood
    - e.g. predicting farming yield given previous yields, weather, employee number
    - e.g. predict length of stay in hospital ⇒ a continuous variable

build a model on training data, make accurate predictions on new, unseen data (that has the same characteristics as the training set)

making accurate predictions from *unseen* data = ability to *generalize* ("generalization ability")
overfitting ⇒ building a model that is too complex for the amount of information we have (creating rules for everything) → "fitting a model too closely to the particularities of the training set, obtaining a model that works well on the training set but is not able to generalize to new data"
(fits the training set really well but not the test set)
underfitting ⇒ model is too simple, not able to capture all the variability in the data, does badly even on the training set

the more complex the model, the better it will be able to predict on training data, too complex, and the model will not generalize well to new data (you want low bias AND low variability)



must find the sweet spot between a simple model and a complex model

the larger variety of data points your data- set contains, the more complex a model you can use without overfitting

do this with: regularization, boosting and bagging.

Going through machine learning algorithms:

k-Nearest Neighbours

k-NN

- *to make a prediction for a new data point, the algorithm finds the closest data points in the training dataset - "nearest neighbours"*
    - *good*:
        - simplest ML algorithm
        - building this model: = just storing the training dataset, very fast
            - but when very large training set, prediction is slow
        - easy to understand
    - bad
        - have to preprocess your data
            - bad when datasets have hundreds of features
            - bad when most features are 0 most of the time (sparse dataset)

Classification:

simplest model (1 neighbour)
considers only one nearest neighbour (the closest data point in the training set to the one we want a prediction of) → the prediction is then simply the known output for this training point

you can *consider* an arbitrary number (k) of neighbours

- voting to assign a label (count how many neighbours belong to class a and how many neighbours belong to class b) → we then assign the class that is more frequent (the majority class among the k-nearest neighbours)

1st split data into a training and a test set

```
from sklearn.model_selection import train_test_split

X, y = mglearn.datasets.make_forge() #loading dataset input & output

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

Then import and instantiate the class (creating an object - called whatever you want), setting the number of neighbours to use as a parameter

- recall: All machine learning models in scikit-learn are implemented in their own classes, called Estimator classes.

Parameters:

- the number of neighbours
- how you measure distance between data points (beyond the scope of the course) → Euclidean distance is used by default

```
from sklearn.neighbors import KNeighborsClassifier
clf = KNeighborsClassifier(n_neighbors=3)
```

Then: you *build* the model by calling the fit method on the created object (clf), passing the training data (X_train) and training outputs (y_train) as arguments (modifies the object (clf) in place)

```
clf.fit(X_train, y_train)
```

To make predictions on the test dataset, call the predict method (for each data point in the test set i.e. X_test, it will compute its nearest neighbours in the training set and ouptut the most common class among these)

```
clf.predict(X_test)
```

To evaluate how well the model *generalize*, call the score method on the object:

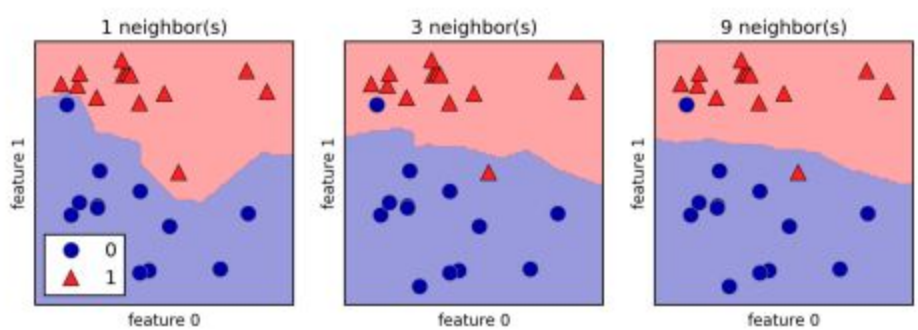
```
clf.score(X_test, y_test)
```

Out: test set accuracy 0.86

= 86% accurate (model predicted class correctly for 86% of the samples in the test dataset)

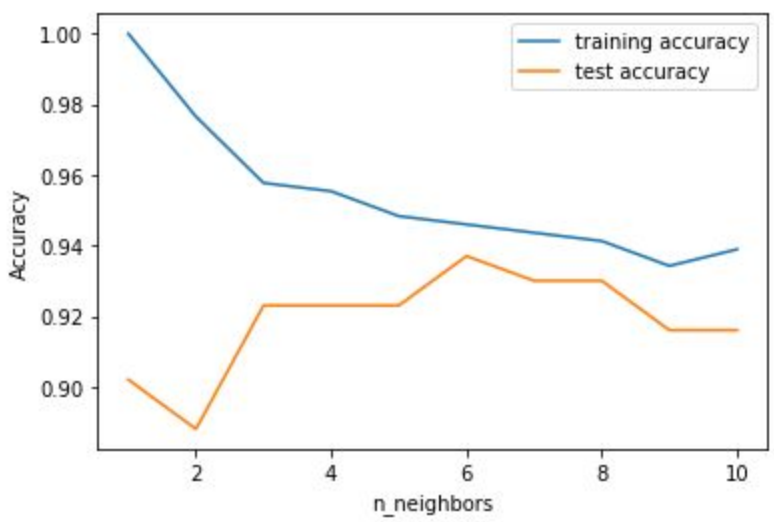
To view the decision boundary (for 2D datasets):

```
fig, axes = plt.subplots(1, 3, figsize=(10, 3))

for n_neighbors, ax in zip([1, 3, 9], axes):
    # the fit method returns the object self, so we can instantiate
    # and fit in one line
    clf = KNeighborsClassifier(n_neighbors=n_neighbors).fit(X, y)
    mglearn.plots.plot_2d_separator(clf, X, fill=True, eps=0.5, ax=ax,
alpha=.4)
    mglearn.discrete_scatter(X[:, 0], X[:, 1], y, ax=ax)
    ax.set_title("{} neighbor(s)".format(n_neighbors))
    ax.set_xlabel("feature 0")
    ax.set_ylabel("feature 1")
axes[0].legend(loc=3)
```



using few neighbours = high model complexity, using many neighbours = low model complexity
e.g to build the same model using the example dataset and an increasing k neighbours, from 1 neighbour to 11 neighbours: then plotting training set accuracy AND test set accuracy (y axis) against the number of neighbours on the x axis

```python
from sklearn.datasets import load_breast_cancer

cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, stratify=cancer.target, random_state=66)

training_accuracy = []
test_accuracy = []
# try n_neighbors from 1 to 10
neighbors_settings = range(1, 11)

for n_neighbors in neighbors_settings:
    # build the model
    clf = KNeighborsClassifier(n_neighbors=n_neighbors)
    clf.fit(X_train, y_train)
    # record training set accuracy
    training_accuracy.append(clf.score(X_train, y_train))
    # record generalization accuracy
    test_accuracy.append(clf.score(X_test, y_test))

plt.plot(neighbors_settings, training_accuracy, label="training accuracy")
plt.plot(neighbors_settings, test_accuracy, label="test accuracy")
plt.ylabel("Accuracy")
plt.xlabel("n_neighbors")
plt.legend()
```



the accuracy with which the model predicts the training set decreases with more neighbours, but the accuracy with which the model predicts the test set increases (ability to generalize) to a point (overfitting vs underfitting)
best performance around 6 neighbours

k-neighbors regression

regression variant of the k-nearest neighbours algorithm
prediction using a single neighbour ⇒ just the target value of the nearest neighboru
prediction using multiple neighbours ⇒ the average/mean of the relevant neighbours

implemented in the:
KNeighborsRegressor
class in scikit learn

Instantiated, trained in the same way:

```
from sklearn.neighbors import KNeighborsRegressor

X, y = mglearn.datasets.make_wave(n_samples=40)

# split the wave dataset into a training and a test set
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

# instantiate the model and set the number of neighbors to consider to 3
reg = KNeighborsRegressor(n_neighbors=3)
# fit the model using the training data and training targets
reg.fit(X_train, y_train)
```

Make predictions on the test set:

reg.predict(X_test)

Evaluate the model by calling the score method. For regressors → returns the R^2 score

R^2 score = *the coefficient of determination* = a measure of goodness of a prediction for a regression model, yields a score between 0 and 1
1 = perfect prediction
0 = constant model that just predicts the mean of the training set responses (y_train)

reg.score(X_test, y_test))
(for numeric/continuous)

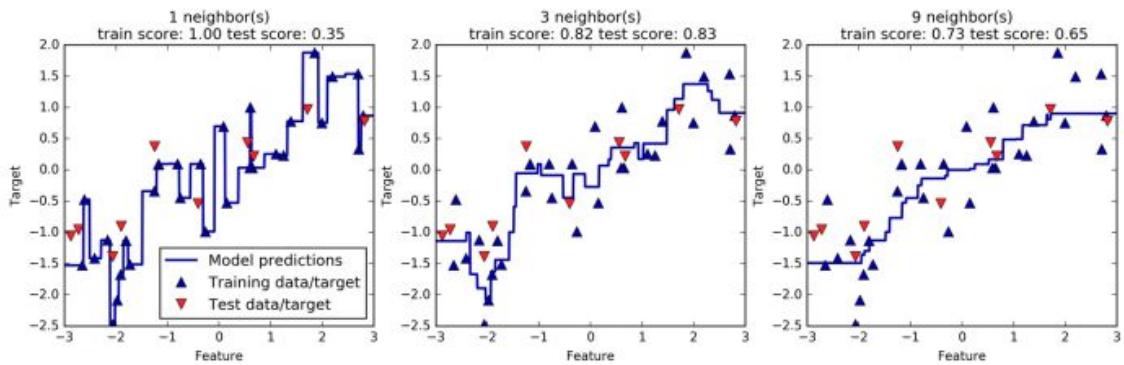


Figure 2-10. Comparing predictions made by nearest neighbors regression for different values of `n_neighbors`

more neighbours = smoother predictions, but these do not fit the training data as well

Linear Models

linear models ⇒ make a prediction using a *linear function* of the input features

THE ASSOCIATION BETWEEN THE INPUTS AND THE OUTPUT IS LINEAR AND ADDITIVE

linear models ⇒ ones that use a linear combination of their input features

- very easy to train
- fast to make predictions
- work very well with big data

linear models for classification, regression:

linear regression vs. linear classification?
linear models of regression: the output is ŷ is a linear function of the features (a line, plane, hyperplane in higher dimensions)
linear models for classification, the decision boundary is a linear function of the input.

linear model for regression:

general prediction formula for a linear model:

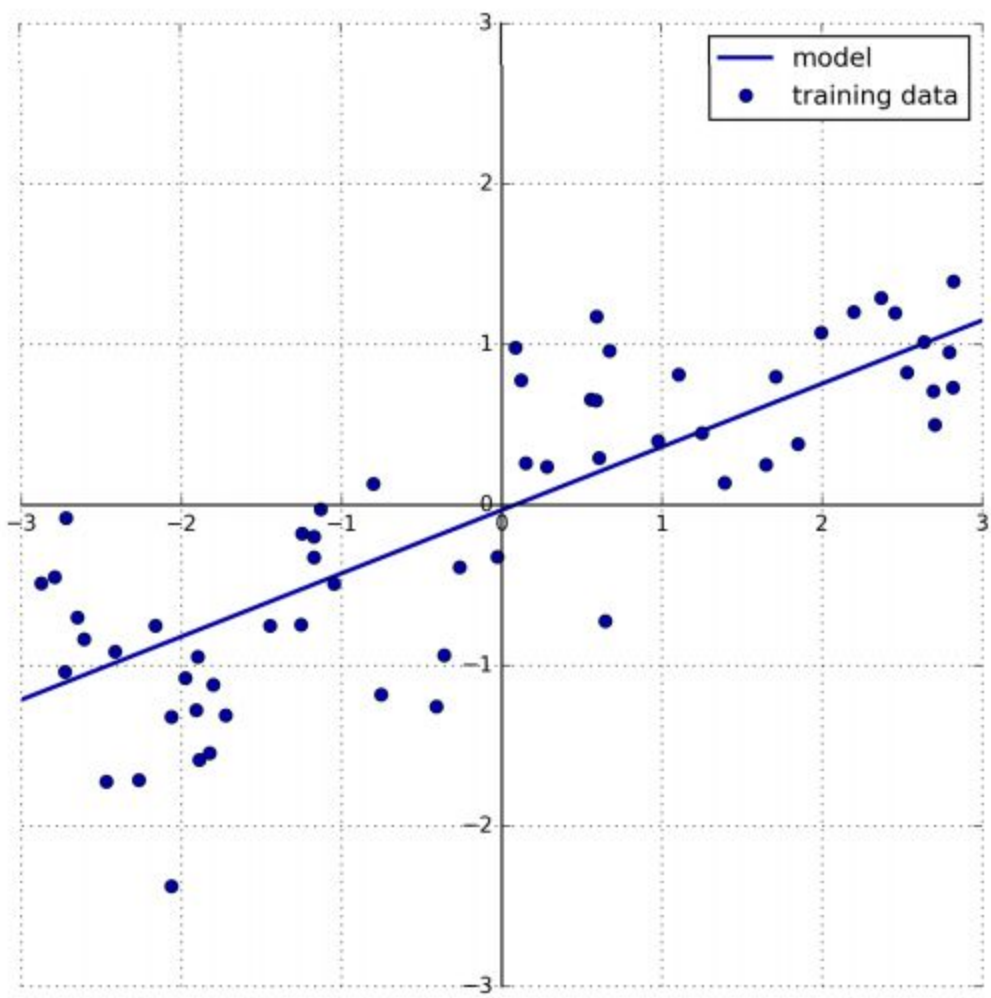
$\hat{y} = w[0] * x[0] + w[1] * x[1] + ... + w[p] * x[p] + b$

where x is a feature (p features), w and b are the parameters the model that are learned, y is the prediction the model makes i.e. for a single feature

$y = w[0]*x[0] + b$ (remember like the equation for a line $y = mx + b$)
m is the gradient, b is the y-axis offset

w contains the slopes along each feature axis, “weights”, can be negative, given by the entries of w

predicted response is a weighted sum of the input features



linear models for regression → prediction is a line for a single feature, a plane when using two features (or variables or whatever), a hyperplane in higher dimensions!!! (forms a cube with a plane through it → harder to visualise)

- if there's only one feature, then it's a line (y = w[0]*x[0] + b)
- more features then it produces some crazy geometry

LINEAR MODELS REQUIRE STRICTLY NUMERIC INPUTS
must convert categorical features into numerics
do this with dummy variables

THERE ARE MANY DIFFERENT LINEAR MODELS FOR REGRESSION:

- (of which, standard "linear regression" or LinearRegression is only one type)

Linear Regression (ordinary least squares)

simplest/most classic linear method for regression

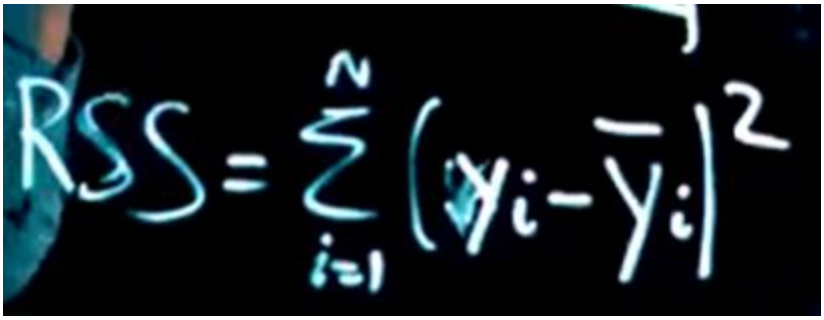
finds the parameters w and b that minimize the *mean squared error* between predictions and the true regression targets, y, on the training set

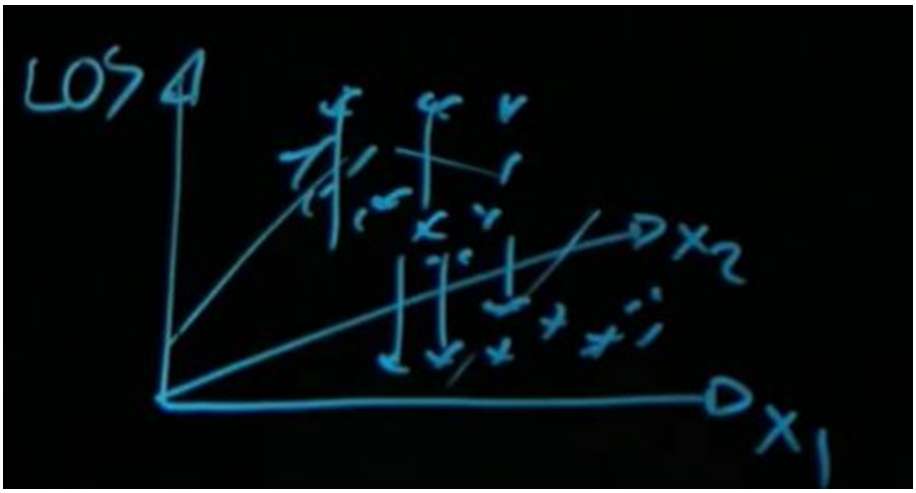
mean squared error = the sum of the (differences between the predictions and the true values)^2
MSE = 1/n(RSS) (the same as the RSS just with a coefficient)
(squared so negative distances do not cancel out positive distances)

- variance another error we introduce → taking a subset , introducing some variance the error we introduce by using a particular training set THE HIGHER THE VARIANCE THE WORSE GENERALIZATION
    - variance = the difference in fits between data sets
    - low variability = produces consistent predictions across datasets
- bias → by oversimplifying reality we have some error, the error we always introduce by the algorithm or function that we introduce in Machine learning
    - linear regression fits a straight line to the training set. bias = the inability of the machine learning method (like linear regression) to capture the 'true' relationship
    - low bias = accurately models the 'true' relationship
- noise = INCREASED BIAS, LESS FITTED

we are trying to make MSE (variance and bias) as small as possible
do not increase at the same time by the same number

residual sum of the squares → trying to minimize this value (distance between all the points and the line, plane or hyperplane (depending on the number of features))
keeps creating lines/functions, calculates this value (MSE) by adding all the distances from the line/prediction, creates a new line etc, picks the one with the smallest MSE → (uses the derivative = 0 in maths)





- has no parameters
- has no way to control model complexity

```
from sklearn.linear_model import LinearRegression

X, y = mglearn.datasets.make_wave(n_samples=60)

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

lr = LinearRegression().fit(X_train, y_train)
```

or

```
X, y = mglearn.datasets.load_extended_boston()

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

lr = LinearRegression().fit(X_train, y_train)
```

The “slope” parameters (w), also called weights or coefficients, are stored in the coef_ attribute, while the offset or intercept (b) is stored in the intercept_ attribute:

```
print(lr.coeff)
print(lr.intercept)
```

Output:

lr.coef_: [0.394]
lr.intercept_: -0.031804343026759746

Ridge Regression

Another linear model for regression
same formula (as ordinary least squares)
however, coefficients (w) → to predict well on training data + are chosen to fit an additional constraint
we want all entries of w to be as close to zero as possible
two competing interests: have to minimise RSS but also has to be in this circle
means each feature should have as little effect on the outcome as possible (small slope) while still predicting well
ALLOWS US TO CONTROL COMPLEXITY

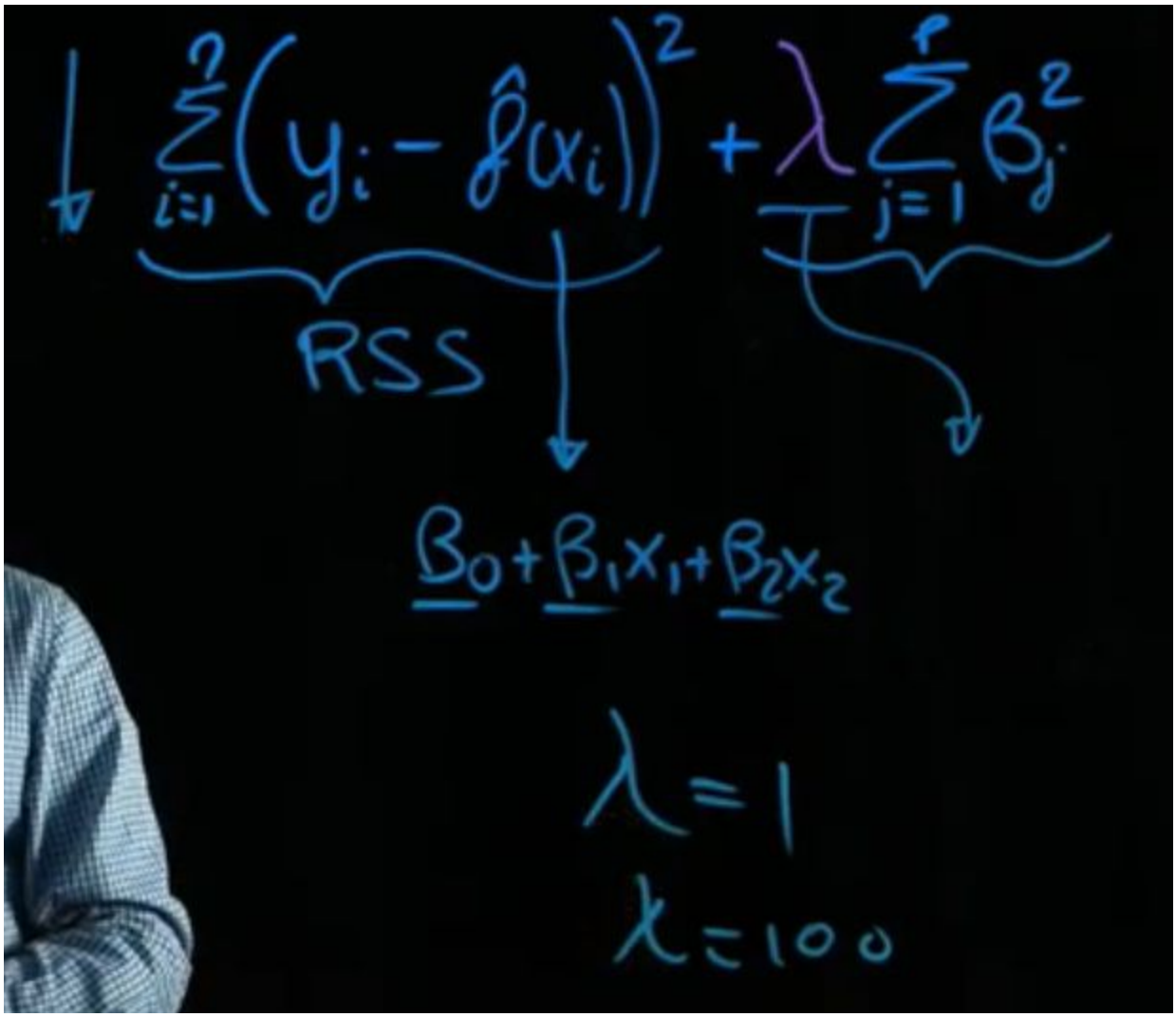
constraint is an example of REGULARIZATION (specifically L2 regularization)
L2 penalty
restrain the coefficients

REGULARIZATION = explicitly restricting a model to avoid overfitting (LinearRegression above has no regularization)

Betas (ws) can be anything ⇒
WHEN YOU CALCULATE MSE YOU ALSO ADD THE SUM OF ALL THE COEFFICIENTS SQUARED (L2 PENALTY) → BIASES MODELS WITH SMALLER COEFFICIENTS

$$\sum_{i=1}^{n}(y_i - \sum_{j=1}^{p} x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} \beta_j^2$$

restraining the betas (sigma $(B_i)^2$ <= s) ⇒ forms a circle go close to zero but mathematically proved to never go to zero



L1 IS THE SAME BUT YOU'RE ADDING THE ABSOLUTE VALUE → WHICH PUSHES COEFFICIENTS TO 0

implemented in the

linear_model.Ridge
class

```
from sklearn.linear_model import Ridge

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

print("Training set score: {:.2f}".format(ridge.score(X_train, y_train)))
print("Test set score: {:.2f}".format(ridge.score(X_test, y_test)))
```

Output:

Training set score: 0.89
Test set score: 0.75

Ridge regression is a more restricted model (linear model of regression) than linear regression
less likely to overfit (less complex)
less complex model = worse predictions for training set, better generalization

makes a trade off between the simplicity of the model (nearer zero coefficients) and performance on the training set:

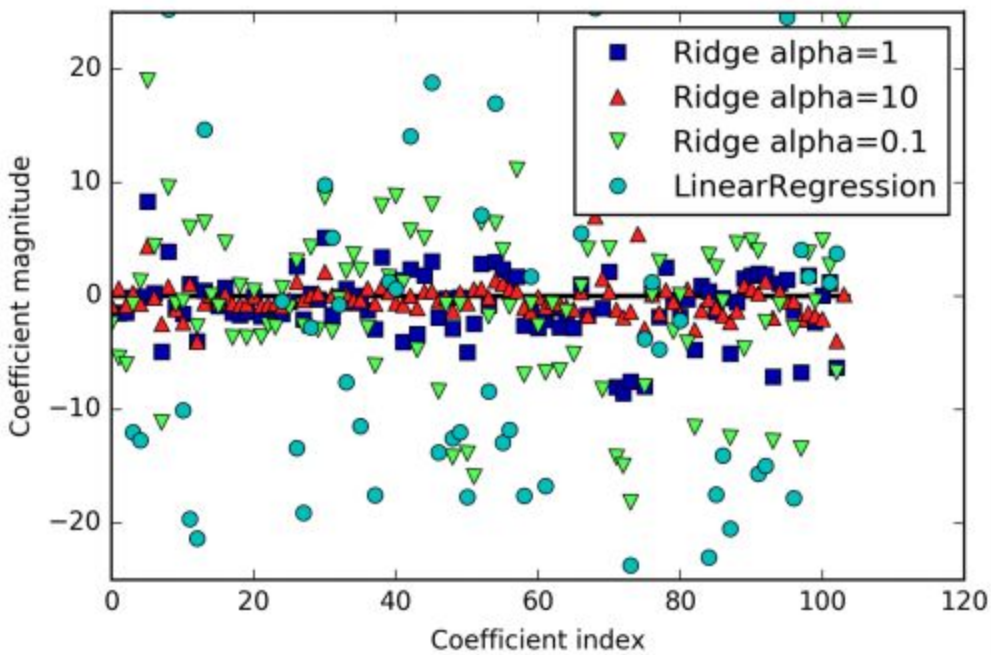
you can specify how much importance the model places on simplicity versus training set performance by passing in the alpha parameter (when you instantiate the class)

↑alpha = more restricted ⇒ the coefficients (w) are forced toward zero more, ↓training set performance, (potentially) ↑generalization

```
ridge10 = Ridge(alpha=10).fit(X_train, y_train)
print("Training set score: {:.2f}".format(ridge10.score(X_train, y_train)))
print("Test set score: {:.2f}".format(ridge10.score(X_test, y_test)))
```

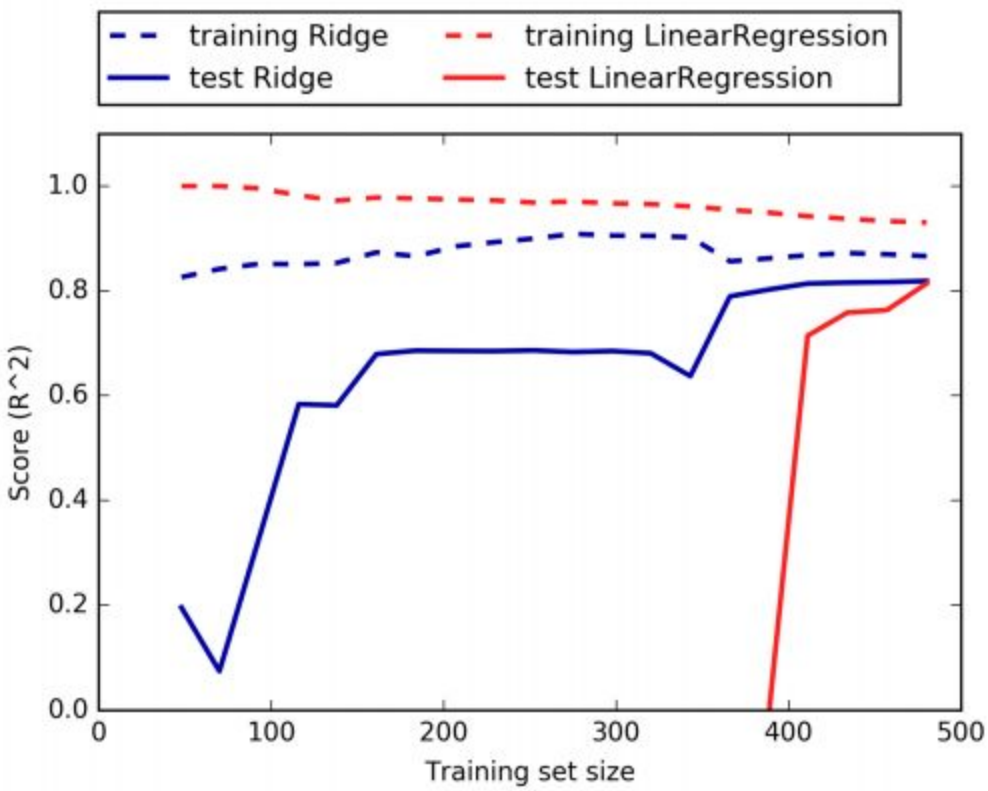
↓alpha = coefficients are less restricted ⇒ (very small values of alpha → model resembles LinearRegression)

```
ridge01 = Ridge(alpha=0.1).fit(X_train, y_train)
```



Ridge is better at predicting output (test score), particularly for small training sets

```
mglearn.plots.plot_ridge_n_samples()
```



- because ridge is regularized, the training score (score for predictions on the training set) of ridge is lower than the training score of linear regression
- test score for ridge is better, particularly for small training sets
- linear regression is not able to learn anything for less than 400 data points
- with enough training data, regularization becomes less important, linear regression catches up

Lasso

also restricts coefficients to be close to zero but by using L1 regularization (or L1 penalty)
BUT means that some coefficients can be *exactly zero* (some features entirely ignored by the model)
L2 = the square of the betas
L1 = absolute value of beta → sigma $|B_i|$<= s → forms a diamond → restrained to be close to zero + CAN actually be zero

$$\sum_{i=1}^{n}(y_i - \sum_{j=1}^{p} x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} \beta_j^2$$

FEATURE SELECTION

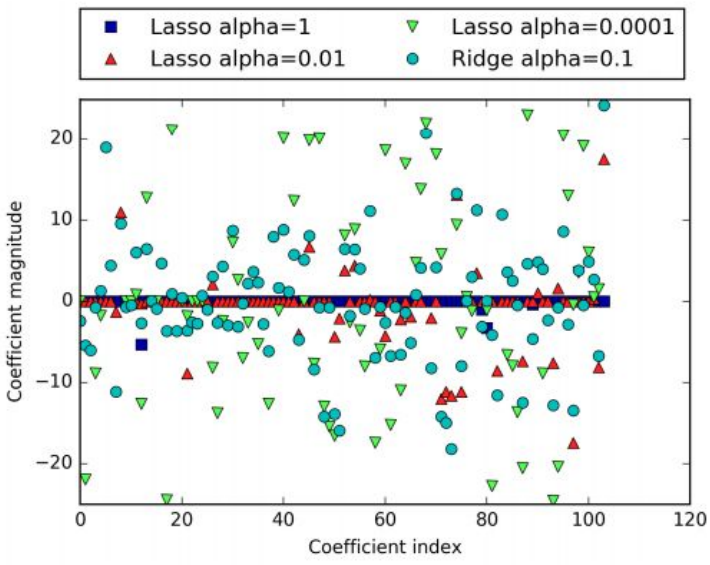
⇒ automatic feature collection

```
from sklearn.linear_model import Lasso

lasso = Lasso().fit(X_train, y_train)
print("Training set score: {:.2f}".format(lasso.score(X_train, y_train)))
print("Test set score: {:.2f}".format(lasso.score(X_test, y_test)))
print("Number of features used: {}".format(np.sum(lasso.coef_ != 0)))
```

also has a regularization parameter, alpha, that controls how strongly coefficients are pushed toward zero
default (no parameter) = 1.0

decreasing alpha → reduces underfitting
(may be warned you have to increase the maximum number of iterations)

```
# we increase the default setting of "max_iter",
# otherwise the model would warn us that we should increase max_iter.
lasso001 = Lasso(alpha=0.01, max_iter=100000).fit(X_train, y_train)
print("Training set score: {:.2f}".format(lasso001.score(X_train,
y_train)))
print("Test set score: {:.2f}".format(lasso001.score(X_test, y_test)))
print("Number of features used: {}".format(np.sum(lasso001.coef_ != 0)))
```


Lasso alpha=1
Lasso alpha=0.01
Lasso alpha=0.0001
Ridge alpha=0.1
Coefficient magnitude
Coefficient index
20
10
0
−10
−20
0
20
40
60
80
100
120

Elastic Net

ElasticNet class (imported from scikit-learn) combines the penalties of Lasso and Ridge
this combination works best
have to adjust two parameters: one for L1 regularization, one for L2 regularization

Evaluation metrics for regression

in most applications → default R^2 score is enough

= the proportion of the variance in the dependent variable that is predictable from the independent variable(s)

(NOT ACCURACY → BUT method is called score for both classification and regression models)

THERE ARE MANY DIFFERENT LINEAR MODELS FOR CLASSIFICATION

Binary classification:
prediction is made using the formula:
$\hat{y} = w[0] * x[0] + w[1] * x[1] + ... + w[p] * x[p] + b > 0$

instead of just returning the weighted sum of features,
the threshold of predicted features is 0
<0: we predict class -1
>0: we predict class +1

For linear models for regression, the output, $\hat{y}$, is a linear function of the features: a line, plane, or hyperplane (in higher dimensions).
For linear models for classification, the decision boundary is a linear function of the input.
a (binary) linear classifier is a classifier that separates two classes using a line, a plane, or a hyperplane.

the most common linear classification algorithms are:
probability of y given x (P(y/x))

the way the model is fitted is more complicated:
choose beta coefficients by *maximum likelihood* (such that the probability is close to 1 when the true value is 1, close to 0 when the true value is 0)

Logistic regression

implemented in the class:
linear_model.LogisticRegression

NB: misnomer - logistic regression is a classification algorithm (not a regression algorithm)
DON'T CONFUSE THIS WITH LinearRegression

used for binary classification - the response, y , is a categorical variable with 2 levels (e.g. readmission YES/NO)

Rather than modelling the response variable directly:
logistic regression → involves modelling/estimates the probability that the response variable belongs to a particular category (the positive class/object of the study), by the logistic function:

$P(Y=1|X) = e\beta 0+\beta 1X1+\cdots+\beta pXp/+e\beta 0+\beta 1X1+\cdots+\beta pXp=11+e-[\beta 0+\beta 1X1+\cdots+\beta pXp]$

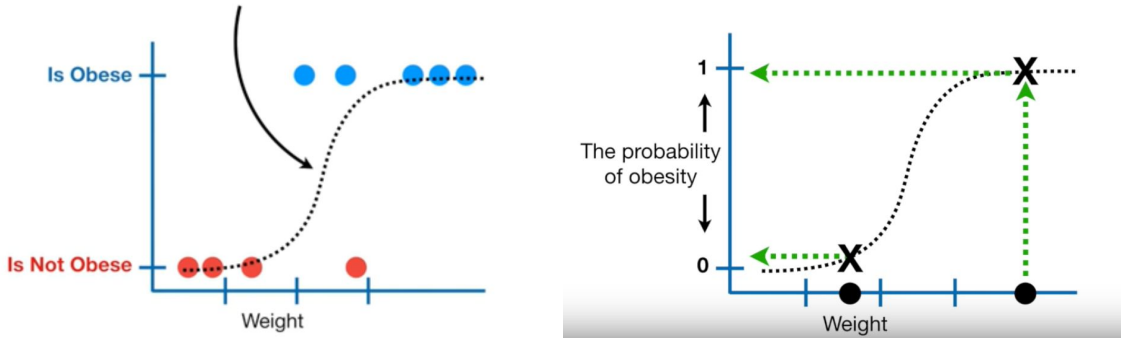
You can rearrange the logistic function into what is known as the logit function:

$log(P(y=YES)/1-P(y=YES))=\beta 0+\beta 1X1+\cdots+\beta pXp$

uses "maximum likelihood"

forms a sigmoid shape - fits an S-shaped "logistic function" (rather than a line) ⇒ 0 (not readmitted), 1 (readmitted)
threshold = 50 % (default)
y-axis is converted to the probability that an (input) is (class 1) - curve tells you the probability that the e.g. mouse is obese based on e.g. weight
prediction can be more complicated (weight, genotype, age)

can work with continuous data (weight, age) and discrete data (genotype) → even though it outputs a discrete variable



One way to classify: set a threshold at 0.5 (classify all _ as with a probability of being class 1 > 0.5 as being class 1, and everything with a probability <0.5 as being class 0)

Odds Ratio

the quantity $P(y=YES)/1-P(y=YES)$ has a special interpretation = odds ratio of an event occurring, and can take on any value between 0 and ∞ .

Notice that $1-P(y=YES)$ is the probability that y is not YES
so the odds ratio is the probability that $y=YES$ over the probability that $y \neq YES$.

logistic regression essentially involves modelling the log odds of an event via a linear regression model

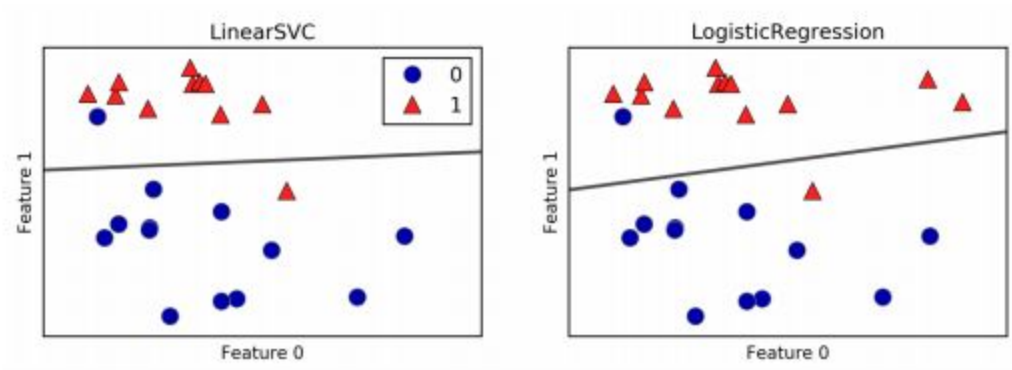
class_weight

the value you want the algorithm to put on a particular class output (e.g. if you have 90% class 0 and 10% class 1 → you want the algorithm to put more value on class 1 predictions, rather than just prediction class 0 all the time)

Linear Support Vector Machines (linear SVMs)

implemented in the class:
svm.LinearSVC

SVC = “support vector classifier”
⇒ these ones actually find a decision boundary that minimises the distance between the points and the line itself (not logistic regression)



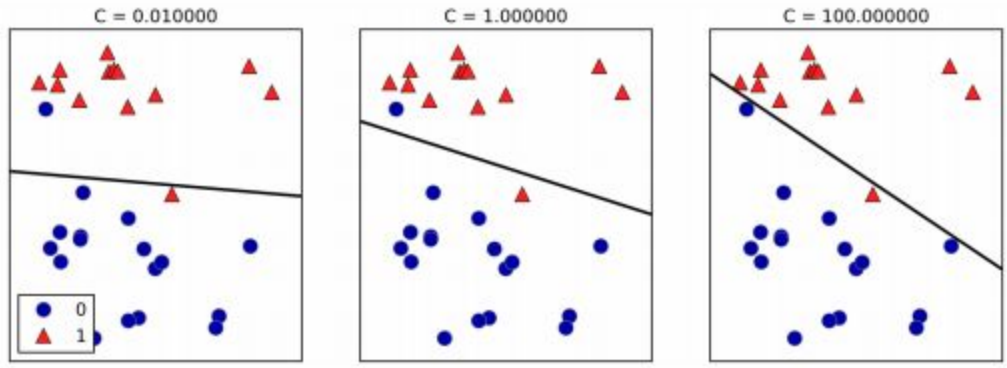
CREATED a decision boundary
any data point above the black line = classified as class 1
any point below the black line = classified as class 0

both models use L2 regularization by default

for LogisticRegression and LinearSVC ⇒ C = the trade off parameter that determines the strength of regularization
default value (no parameter supplied) of C = 1

$$(C = alpha = 1/\lambda)$$

```
mglearn.plots.plot_linear_svc_regularization()
```



↑C = ↓regularization, try to fit the training set as best as possible
stresses the importance of each individual data point to be classified correctly

↓C = ↑regularization, more emphasis on finding a coefficient vector (w) close to zero
try to adjust to the majority of data points

Using logistic regression (with default value of C = 1)

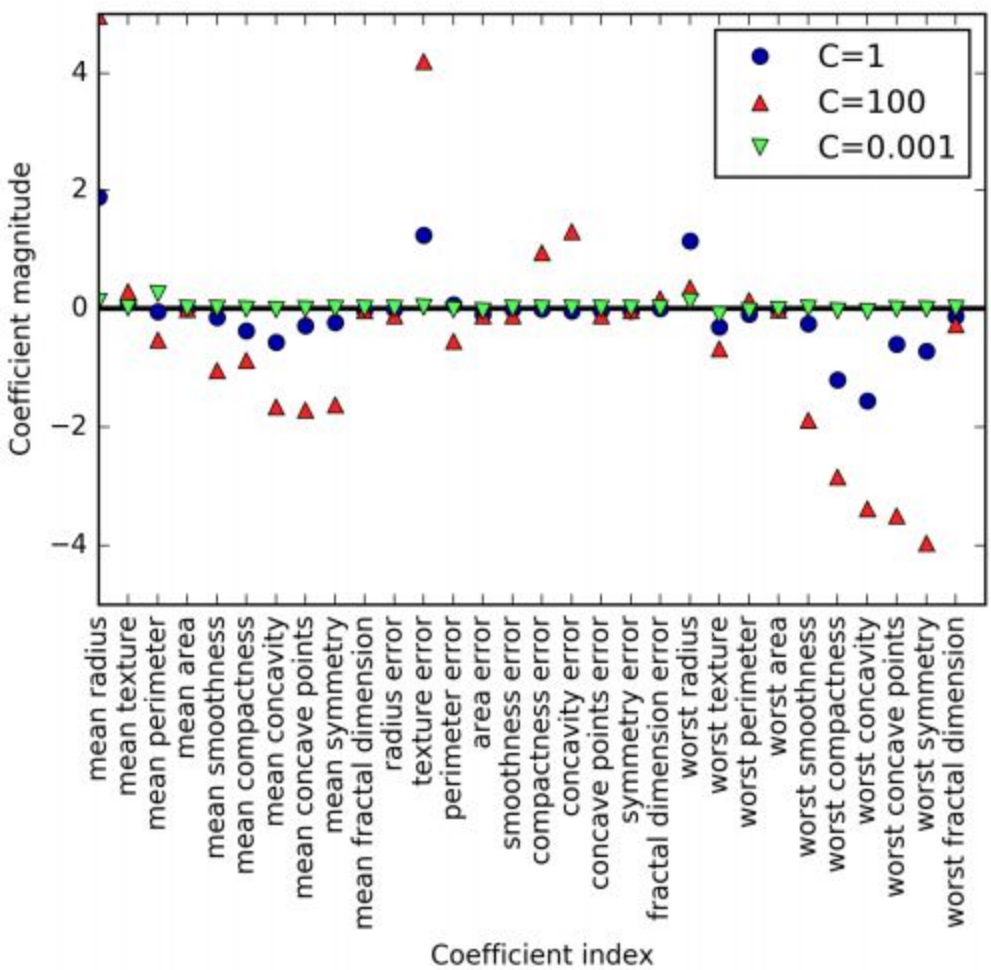
```
from sklearn.datasets import load_breast_cancer

cancer = load_breast_cancer()

X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, stratify=cancer.target, random_state=42)

logreg = LogisticRegression().fit(X_train, y_train)

print("Training set score: {:.3f}".format(logreg.score(X_train, y_train)))
print("Test set score: {:.3f}".format(logreg.score(X_test, y_test)))
```

using a more flexible model (higher value of C):

```
logreg100 = LogisticRegression(C=100).fit(X_train, y_train)
print("Training set score: {:.3f}".format(logreg100.score(X_train,
y_train)))
print("Test set score: {:.3f}".format(logreg100.score(X_test, y_test)))
```

using a more regularized model (low value of C)

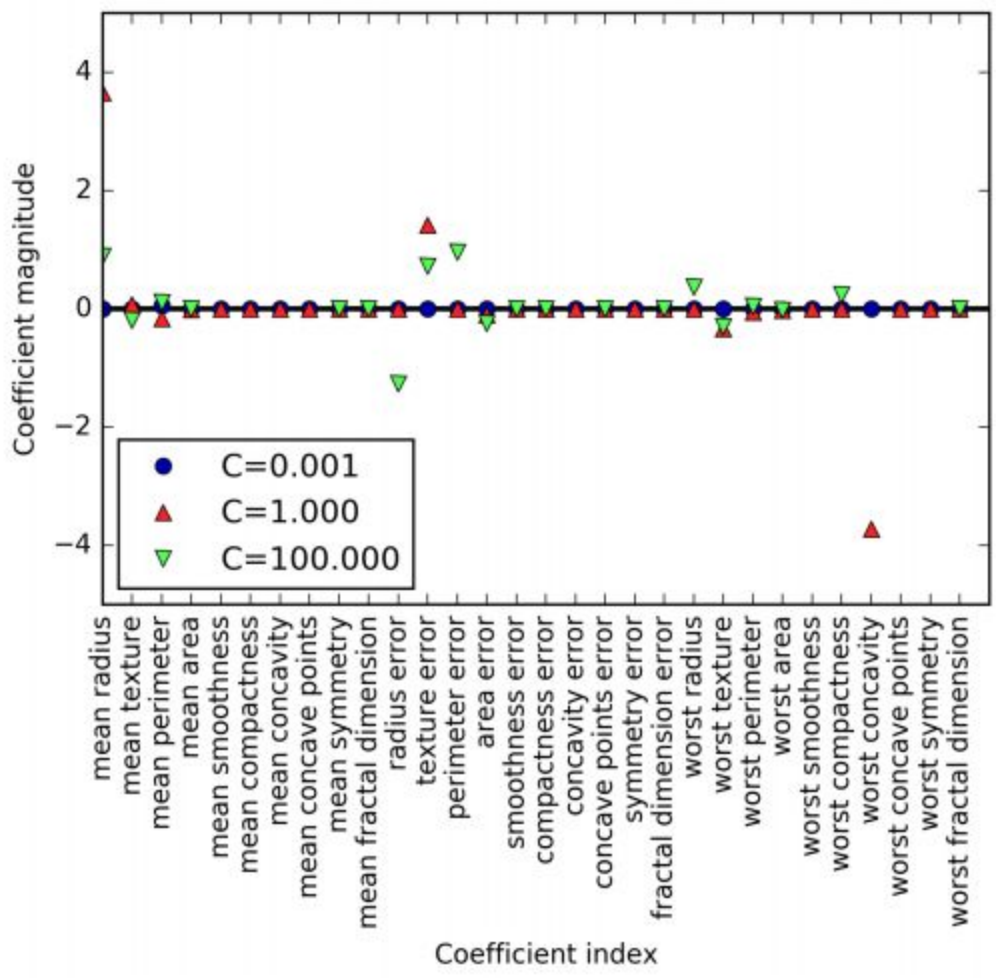
```
logreg001 = LogisticRegression(C=0.01).fit(X_train, y_train)
print("Training set score: {:.3f}".format(logreg001.score(X_train,
y_train)))
print("Test set score: {:.3f}".format(logreg001.score(X_test, y_test)))
```



To get the model to use L1 regularization, pass the constructor the argument penalty="L1"

```
lr_l1 = LogisticRegression(penalty="l1").fit(X_train, y_train)
```

will limit the model to using only a few features
L2 regularization = will use all available features, default for both
L1 regularization = will select only a subset of features



multiclass classification:
most linear models for classification are for binary only:

to extend a binary classification algorithm to a multiclass classification algorithm → use the one vs rest approach

- binary model learned for each class
- tires to separate that class form all other classes
- as many binary models as there are classes (one vector of coefficients, one intercept for each class)
- prediction: all binary classifiers are run on a test point, the classifier that has the highest score on its single class wins, this class label is the prediction

```
from sklearn.datasets import make_blobs
X, y = make_blobs(random_state=42)

mglearn.discrete_scatter(X[:, 0], X[:, 1], y)

plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
plt.legend(["Class 0", "Class 1", "Class 2"])
```

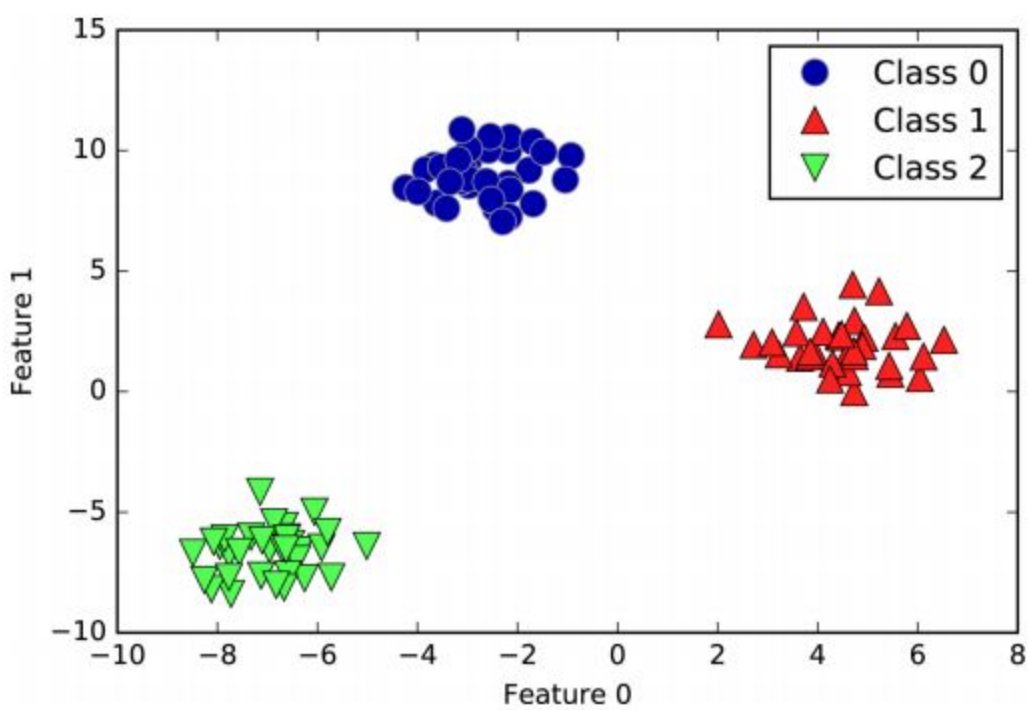


Figure 2-19. Two-dimensional toy dataset containing three classes

```
linear_svm = LinearSVC().fit(X, y)
print("Coefficient shape: ", linear_svm.coef_.shape)
print("Intercept shape: ", linear_svm.intercept_.shape)
```

output:
coefficient shape: (3, 2)
Intercept shape: (3,)

→ each row of coef_ contains the coefficient vector for one of three classes
→ each column of coef_ contains the coefficient value for a specific feature
→ intercepts are now a one-dimensional array, storing the intercepts for each class

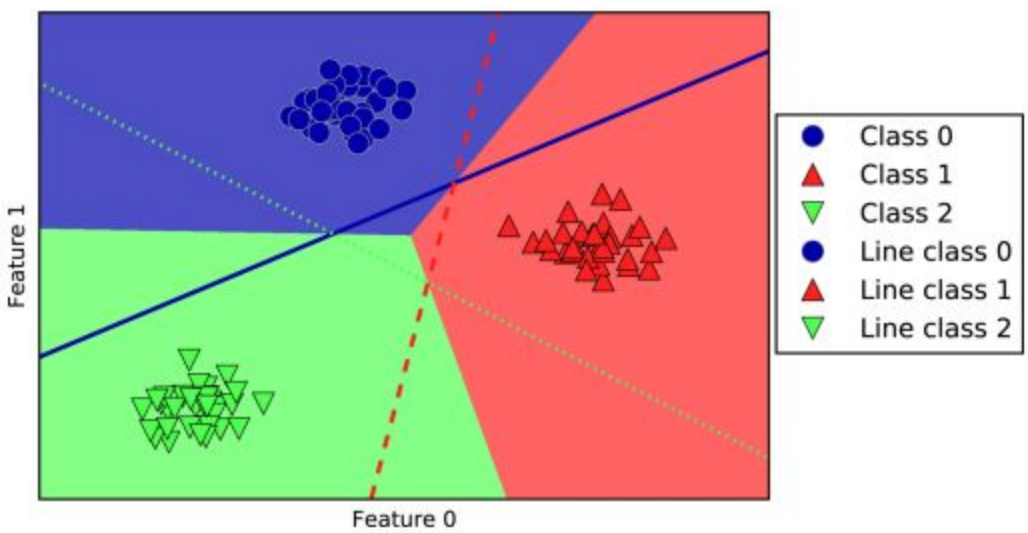


Figure 2-21. Multiclass decision boundaries derived from the three one-vs.-rest classifiers

RECAP OF LINEAR MODELS (for regression and classification):

Large values for alpha or small values for C mean simple models
(potentially underfitting?)

L1 regularization → can set some w coefficients as 0, only including a few features, easier to explain which features are important to the model, the effects of features

L2 regularization → uses all features

solver='sag' option in LogisticRegression and Ridge, which can be faster than the default on large datasets.

easy to understand how a prediction is made (using formula)
perform well when the number of features is large compared to the number of samples

NB:
(obvious)
remember you can chain methods:

you can do model instantiation, fitting (commonly done together), and predicting in one line:

y_pred = LogisticRegression().fit(X_train, y_train).predict(X_test)

not ideal, confusing

- when you instantiate (LogisticRegression()) the class it calls __init__ → remember

Naive Bayes Classifiers

similar to linear models
tend to be even faster to train
generalization performance slightly worse

learn parameters by looking at each feature individually
collect simple per-class statistics from each feature

there are three naive Bayes classifiers in scikit-learn:
- GuassianNB
    - can be applied to any continuous data
    - used on very high-dimensional data
- BernoulliNB
    - assumes binary data
    - counts how often every feature of each class is not zero
    - takes a single argument alpha (controls model complexity)

- MultinomialNB
    - assumes count data (integer counts of something)
    - takes alpha parameter (controls model complexity)

ceof_ means something different → not the same as w

will add alpha many virtual data points with positive values for all the features) → smoothing the statistics
large alpha → more smoothing → less complex models

Evaluation Metrics and Scoring

accuracy (% correctly classified samples)
regression performance (R^2)
= only two ways to evaluate a supervised model

business metric = high level goal of the application

- e.g. decreasing hospital admissions

want model/parameters to have best +ve impact on business metric

binary classification
accuracy not a good measure of predictive performance
false positive = type I error
false negative = type II error

consequences for false positive and false negative rarely the same:
e.g. breast cancer, false negative much worse than false positive

imbalanced dataset/dataset with imbalanced classes ⇒ dataset in which one class is much more frequent than the other

classifier with high accuracy on imbalanced dataset → doesn't take class imbalance into account
can achieve high accuracy without even building a machine learning model, just by always predicting the more frequent class

DummyClassifier

always predicts the majority class:

```
from sklearn.dummy import DummyClassifier dummy_majority =
DummyClassifier(strategy='most_frequent').fit(X_train, y_train)
```

accuracy is an inadequate measure for quantifying predictive performance in this imbalanced setting

Confusion matrices

- best way to represent result of evaluating binary classification

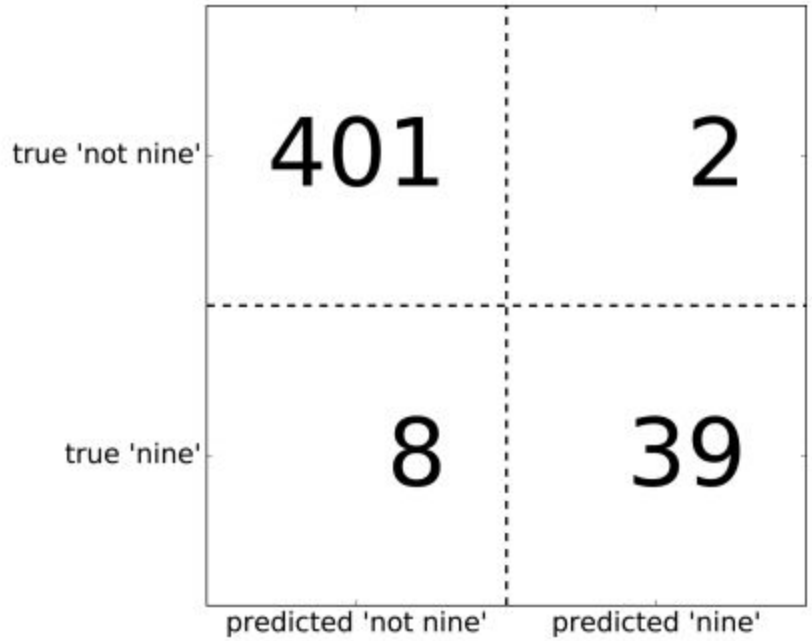
e.g.

```
from sklearn.linear_model import LogisticRegression

logreg = LogisticRegression(C=0.1).fit(X_train, y_train)
pred_logreg = logreg.predict(X_test)

from sklearn.metrics import confusion_matrix

confusion = confusion_matrix(y_test, pred_logreg)
print("Confusion matrix:\n{}".format(confusion))
```

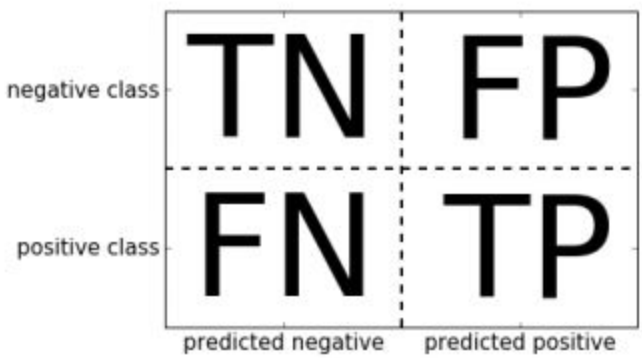
Output:

```
Confusion matrix:
[[401   2]
 [  8  39]]
```

```
mglearn.plots.plot_confusion_matrix_illustration()
```




```
mglearn.plots.plot_binary_confusion_matrix()
```



sensitivity = TP/ TP + FN
specificity = TN/ TN + FP
PPR = TP/ TP + FP

on an imbalanced dataset:

- dummy predictor has a very small number of true positives

$$\text{Accuracy} = \frac{TP+TN}{TP+TN+FP+FN}$$ (the number of correct predictions TP + TN divided by the number of all samples)

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

= *sensitivity*

- important performance metric when WE NEED TO IDENTIFY POSITIVES/AVOID FALSE NEGATIVES
- cancer diagnosis

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

= *positive predictive value* (PPV)

- important when you want to LIMIT THE NUMBER OF FALSE POSITIVES

<u>trade off between optimising recall and optimising precision</u>
perfect recall (sensitivity) when you predict everything as positive, but you will have more false positives, therefore precision will be low

e.g. for logistic regression: if you set the probability threshold very low (if it was super important you correctly identify every positive case)

- would result in correct classifications for all truly positive pts (↑true positives, ↑sensitivity)
- ↑ false positives (↓ PPV/precision)
- ↓false negatives (FN)
- ↓ true negatives (TN)

if you set the probability threshold very high:

- True positives the same
- ↓false positives
- ↑true negative
- false negatives the same

most commonly used measures for binary classification
to summarize them: use an *f-score (or f_1-score)*

$$F = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

better measure than accuracy on imbalanced datasets

To get a comprehensive summary of precision, recall and f-score, use the classification_report function

```
from sklearn.metrics import classification_report

print(classification_report(y_test, pred_most_frequent,
    target_names=["not nine", "nine"]))
```

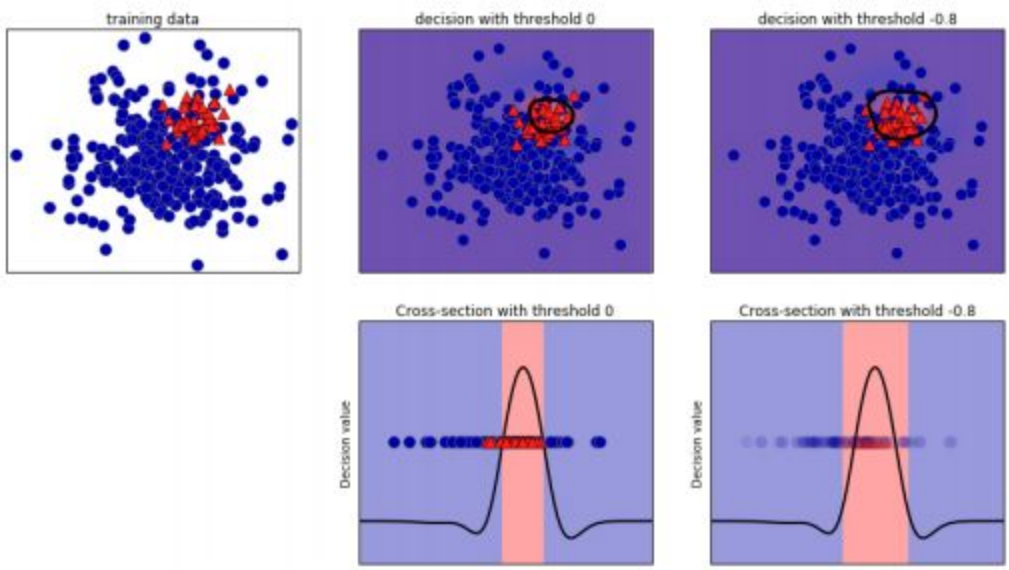
```
             precision    recall  f1-score   support

    not nine       0.90      1.00      0.94       403
        nine       0.00      0.00      0.00        47

avg / total       0.80      0.90      0.85       450
```

taking uncertainty into account

most classifiers provide a decision_function or a predict_proba method to assess degrees of certainty about predictions

prediction = thresholding the output of decision function (0) or predict_proba (0.5) (below this value = designated as class 0, above this value = designated as class 1)

threshold can be set to anything between 0 and 1 → how do we determine which threshold is the best?



e.g. an example:

- binary classification for
- imbalanced dataset
- 400 points in negative class
- 50 points in positive class

```
print(classification_report(y_test, svc.predict(X_test)))
```

Out[51]:

	precision	recall	f1-score	support
0	0.97	0.89	0.93	104
1	0.35	0.67	0.46	9
avg / total	0.92	0.88	0.89	113

is recall used for both sensitivity (class 1) and specificity (class 0)?

if e.g. need to have a high recall for class 1 (e.g. cancer screening) (ie. we're willing to risk more false positives, false class 1s, in exchange for more true positives, increasing the recall)

you can actually alter the default threshold for decision_function (default = 0) → want more points to be classified as class 1, so decrease the threshold

y_pred_lower_threshold = svc.decision_function(X_test) > -.8

```
In[53]:
    print(classification_report(y_test, y_pred_lower_threshold))
Out[53]:
             precision    recall  f1-score   support

          0       1.00      0.82      0.90       104
          1       0.32      1.00      0.49         9

avg / total       0.95      0.83      0.87       113
```

If you value precision over recall or the other way around, or your data is heavily imbalanced, changing the decision threshold is the easiest way to obtain better results.
e.g.

- classifying samples infected with ebola and not infected with ebola → must classify EVERY sample infected with ebola to stop an outbreak → LOWER the threshold

decision_function

- arbitrary ranges

predict_proba

- fixed 0 to 1 scale
- models probability
- default 0.5 = the model is 50% sure that a point is in the positive class
- increasing the threshold = the model has to be more confident/sure to make a positive prediction (and less confident to make a negative decision)
- more intuitive → BUT NOT ALL MODELS PROVIDE REALISTIC MODELS OF UNCERTAINTY (e.g. logistic regression does, tree-based models do NOT)
- calibrated model = provides an accurate measure of uncertainty

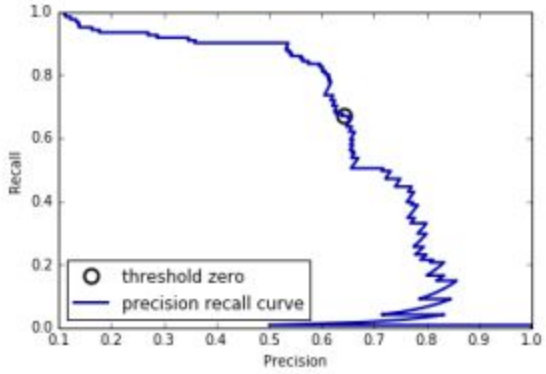
Precision-recall curves + ROC curves

If you want to look at all the possible thresholds, all the possible trade-offs of precision and recalls at once:
create a graph:
precision-recall curve

```
from sklearn.metrics import precision_recall_curve
precision, recall, thresholds = precision_recall_curve(
 y_test, svc.decision_function(X_test))
```

returns a list of precision and recall values for all possible thresholds in sorted order:
then plot a graph:

```
# Use more data points for a smoother curve
X, y = make_blobs(n_samples=(4000, 500), centers=2, cluster_std=[7.0, 2],
                                random_state=22)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
svc = SVC(gamma=.05).fit(X_train, y_train)
precision, recall, thresholds = precision_recall_curve(
 y_test, svc.decision_function(X_test))
# find threshold closest to zero
close_zero = np.argmin(np.abs(thresholds))
plt.plot(precision[close_zero], recall[close_zero], 'o', markersize=10,
            label="threshold zero", fillstyle="none", c='k', mew=2)


plt.plot(precision, recall, label="precision recall curve")
plt.xlabel("Precision")
plt.ylabel("Recall")
```



THE CLOSER THE CURVE STAYS TO THE UPPER RIGHT CORNER, THE BETTER THE CLASSIFIER

You can summarize the precision-recall curve by computing the integral or area under the curve (= the average precision)

use the average_precision_score function to compute the average precision

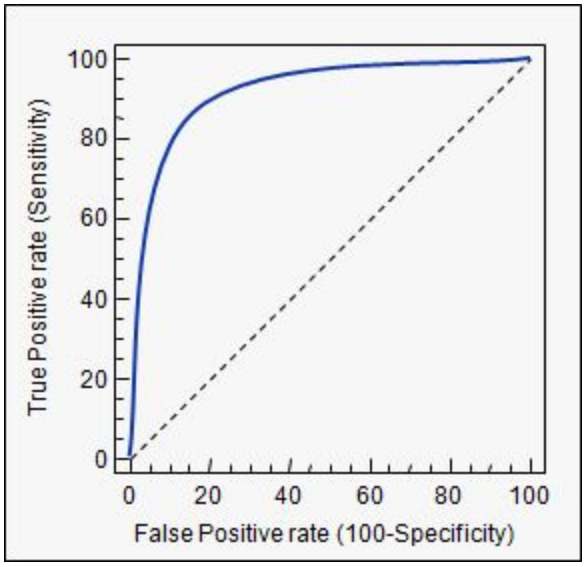
```
from sklearn.metrics import average_precision_score
ap_rf = average_precision_score(y_test, rf.predict_proba(X_test)[:, 1])
ap_svc = average_precision_score(y_test, svc.decision_function(X_test))
print("Average precision of random forest: {:.3f}".format(ap_rf))
print("Average precision of svc: {:.3f}".format(ap_svc))
```

Average precision of random forest: 0.666
Average precision of svc: 0.663

ROC and AUC

“receiver operating characteristics curve”

false positive rate against true positive rate
provides a graph to summarize all of the information you need (rather than being overwhelmed with confusion matrices)



false positive rate (FPR) against the true positive rate (TPR)

true positive rate (sensitivity/Recall)

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

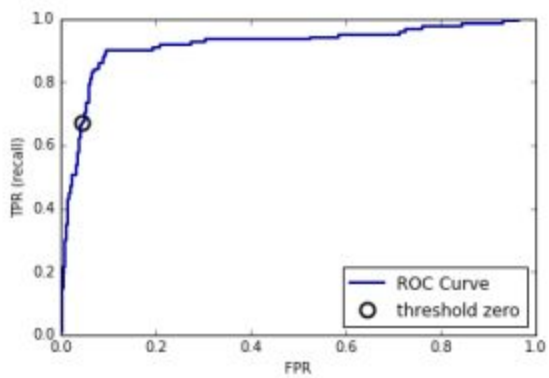
false positive rate = fraction of false positives out of all the truly negative samples

$$FPR = \frac{FP}{FP+TN}$$

$$\text{False Positive Rate} = (1 - \text{Specificity}) = \frac{\text{False Positives}}{\text{False Positives} + \text{True Negatives}}$$

To get the ROC curve use the roc_curve function

```
from sklearn.metrics import roc_curve
fpr, tpr, thresholds = roc_curve(y_test, svc.decision_function(X_test))
plt.plot(fpr, tpr, label="ROC Curve")
plt.xlabel("FPR")
plt.ylabel("TPR (recall)")
# find threshold closest to zero
close_zero = np.argmin(np.abs(thresholds))
plt.plot(fpr[close_zero], tpr[close_zero], 'o', markersize=10,
 label="threshold zero", fillstyle="none", c='k', mew=2)
plt.legend(loc=4)
```



ideal curve: close as possible to the top left
want classifier to produce a high sensitivity/recall (PPV) while keeping a low false positive rate
curve shows we can produce a higher recall by increasing FPR slightly

AUC = summarising the ROC curve using a single number, the area under the curve
compute the area under the curve using the roc_auc_score function

```
from sklearn.metrics import roc_auc_score
rf_auc = roc_auc_score(y_test, rf.predict_proba(X_test)[:, 1])
svc_auc = roc_auc_score(y_test, svc.decision_function(X_test))
print("AUC for Random Forest: {:.3f}".format(rf_auc))
print("AUC for SVC: {:.3f}".format(svc_auc))
```

AUC = between 0 (worst) and 1 (best)
predicting randomly always produces an AUC of 0.5 (no matter how imbalanced the classes in the dataset are)
= the ranking of positive samples
= *the probability that a randomly picked point of the positive class will have a higher score according to the classifier than a randomly picked point from the negative class*
perfect AUC of 1 = all positive points have a higher score than all negative points
FOR CLASSIFICATION PROBLEMS WITH IMBALANCED CLASSES, AUC FOR MODEL SELECTION = MUCH MORE MEANINGFUL THAN ACCURACY

Evaluation metrics for Multiclass classification

derived from binary classification metrics, but averaged over all classes

accuracy = percentage of correctly classified inputs?

metric for imbalanced datasets in the multiclass setting is the multiclass version of the f-score

macro averaging ⇒
weighted averaging ⇒
micro averaging

Cross-validation (Chapter 3)

https://machinelearningmastery.com/k-fold-cross-validation/
https://machinelearningmastery.com/train-final-machine-learning-model/
https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_val_score.html

when we're trying to find out which machine learning method would be best (KNN, SVM, logistic regression…)

interested in measuring how well our model generalizes to new, previously unseen data

Before ⇒ we split our data into a training and test set (partitioned the data into two sets, one for training and one for testing), measured how well our model generalized (called "the training test approach")

- simple to understand and implement
- estimates of test error can be highly variable

cross-validation = (a more robust way) method to assess generalization performance (how well a given algorithm will generalize on a specific dataset)

- data is split repeatedly
- multiple models trained

k-fold cross validation

- k is a user specified number (usually 5 or 10)
- the original dataset/sample is randomly partitioned into *k* parts of approximately equal size/k equal size subsamples (*folds*)
- a sequence of models is trained (a single fold is designated as the test set, and the remaining k-1 folds are used as training data)
- first model is trained using fold 1 as the test set, and 2 - k folds as the training set
- next model is built using fold 2 as the test set and folds 1, 3 - k as the training set
- process repeated using 3, 4, 5...k as test sets
- for each split, computes the accuracy
- hence: we train k models, each with a different test fold, and then average the performance to yield our final accuracy

cross_val_score(model you want to evaluate, training data, ground-truth labels) function

we can compare methods by seeing how well they performed

```
from sklearn.model_selection import cross_val_score
from sklearn.datasets import load_iris
from sklearn.linear_model import LogisticRegression

iris = load_iris()
logreg = LogisticRegression()
scores = cross_val_score(logreg, iris.data, iris.target)
print("Cross-validation scores: {}".format(scores))
```

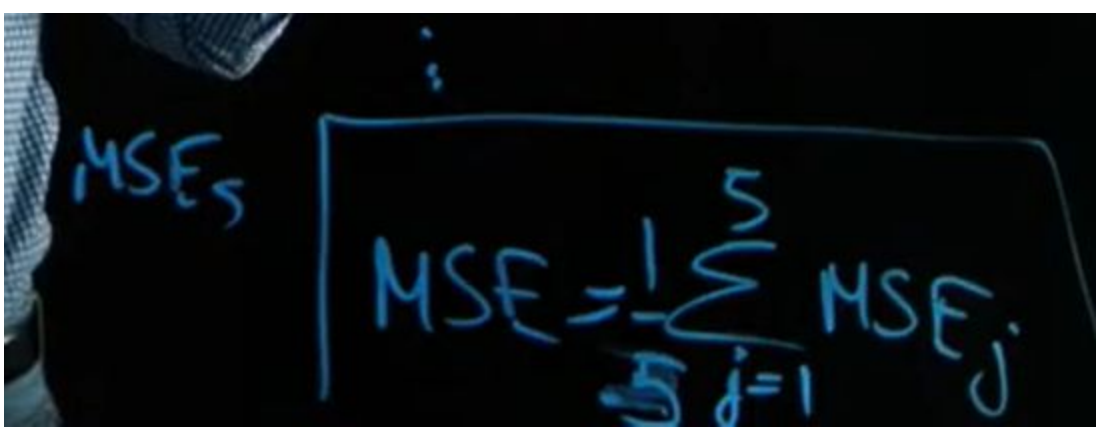
default, k = 3
change the number of folds using the “cv” parameter

```
scores = cross_val_score(logreg, iris.data, iris.target, cv=5)
```

mean cross validation accuracy:

```
print("Average cross-validation score: {:.2f}".format(scores.mean()))
```

- model needs to generalize well to all the folds for all the cross-validation accuracies (and the mean of the accuracies) to be high



- range = worst and best case
- disadvantage = increased computational cost

- DOES NOT BUILD A MODEL - only used to evaluate how well a given algorithm will generalize when trained on a specific dataset

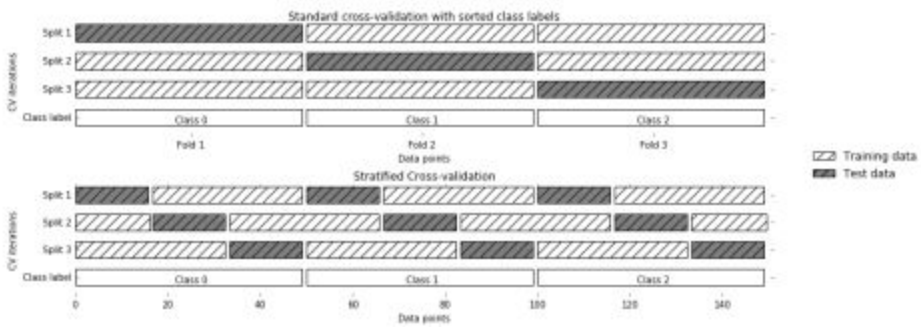
default for regression

should not used for classification

instead use:

stratified k-fold cross validation

- splits the dataset such that proportions between classes are the same in each fold as they are in the whole dataset



= MORE RELIABLE ESTIMATION OF GENERALIZATION PERFORMANCE (than k-fold cross validation) when evaluating a CLASSIFIER

KFold

Can pass a *cross validation splitter* as the cv parameter

```
from sklearn.model_selection import KFold

kfold = KFold(n_splits=5)
print("Cross-validation scores:\n{}".format(
        cross_val_score(logreg, iris.data, iris.target, cv=kfold)))
```

and again:

```
kfold = KFold(n_splits=3)
print("Cross-validation scores:\n{}".format(
      cross_val_score(logreg, iris.data, iris.target, cv=kfold)))
```

To shuffle the data:

```
kfold = KFold(n_splits=3, shuffle=True, random_state=0)
print("Cross-validation scores:\n{}".format(
       cross_val_score(logreg, iris.data, iris.target, cv=kfold)))
```

Leave-one-out cross-validation

each fold is a single sample
in an extreme case, when you call each individual patient (or sample) a block

```
from sklearn.model_selection import LeaveOneOut
loo = LeaveOneOut()
scores = cross_val_score(logreg, iris.data, iris.target, cv=loo)
print("Number of cv iterations: ", len(scores))
print("Mean accuracy: {:.2f}".format(scores.mean()))
```

Shuffle-split cross-validation

each split samples:

- train_size many points for the training set
- test_size many (disjoint) points for the test set

splitting is repeated n_iter times

```
from sklearn.model_selection import ShuffleSplit

shuffle_split = ShuffleSplit(test_size=.5, train_size=.5, n_splits=10)
scores = cross_val_score(logreg, iris.data, iris.target, cv=shuffle_split)
print("Cross-validation scores:\n{}".format(scores))
```

StratifiedShuffleSplit → for classification tasks

Cross-validation with groups

if you have:

- groups in the data that are highly related

to accurately evaluate generalization to new faces, must ensure that the training and test sets contain different people
(multiple samples from the same patient, but are interested in generalizing to new patients)

GroupKFold

- takes an array of groups as an argument
- groups in the data that should not be split when creating training and test sets

```
from sklearn.model_selection import GroupKFold
# create synthetic dataset
X, y = make_blobs(n_samples=12, random_state=0)
# assume the first three samples belong to the same group,
# then the next four, etc.
groups = [0, 0, 0, 1, 1, 1, 1, 2, 2, 3, 3, 3]
scores = cross_val_score(logreg, X, y, groups, cv=GroupKFold(n_splits=3))
print("Cross-validation scores:\n{}".format(scores))
```

Grid Search

hyperparameter tuning

grid search = trying all possible combinations of the parameters you're interested in

creates a grid of parameter settings

Simple Grid Search

You can create a simple grid search just using for loops

```
# naive grid search implementation

from sklearn.svm import SVC

X_train, X_test, y_train, y_test = train_test_split(
 iris.data, iris.target, random_state=0)
print("Size of training set: {} size of test set: {}".format(
 X_train.shape[0], X_test.shape[0]))

best_score = 0

for gamma in [0.001, 0.01, 0.1, 1, 10, 100]:
    for C in [0.001, 0.01, 0.1, 1, 10, 100]:

        # for each combination of parameters, train an SVC

        svm = SVC(gamma=gamma, C=C)
        svm.fit(X_train, y_train)
        # evaluate the SVC on the test set
        score = svm.score(X_test, y_test)

        # if we got a better score, store the score and parameters
```

```
        if score > best_score:
            best_score = score
            best_parameters = {'C': C, 'gamma': gamma}

print("Best score: {:.2f}".format(best_score))
print("Best parameters: {}".format(best_parameters))
```

BECAUSE you used the test data to adjust the parameters, you can no longer use it to assess how good the model is
INSTEAD split the model into 3 sets:

- training set to build the model
- validation set to determine the best parameters of the model
- test set to evaluate the performance of the selected parameters/model
- get evaluation metric (e.g. accuracy) on training set AND validation set, then score it on the test set
- remember: in compsci, validation set is usually called the development set

```
from sklearn.svm import SVC

# split data into train+validation set and test set

X_trainval, X_test, y_trainval, y_test = train_test_split(
    iris.data, iris.target, random_state=0)

# split train+validation set into training and validation sets

X_train, X_valid, y_train, y_valid = train_test_split(
    X_trainval, y_trainval, random_state=1)

print("Size of training set: {} size of validation set: {} size of test
set:"
 " {}\n".format(X_train.shape[0], X_valid.shape[0], X_test.shape[0]))


best_score = 0
for gamma in [0.001, 0.01, 0.1, 1, 10, 100]:
    for C in [0.001, 0.01, 0.1, 1, 10, 100]:
```

```
        # for each combination of parameters, train an SVC
        svm = SVC(gamma=gamma, C=C)
        svm.fit(X_train, y_train)
        # evaluate the SVC on the test set
        score = svm.score(X_valid, y_valid)
        # if we got a better score, store the score and parameters
        if score > best_score:
           best_score = score
           best_parameters = {'C': C, 'gamma': gamma}

# rebuild a model on the combined training and validation set,
# and evaluate it on the test set

svm = SVC(**best_parameters)
svm.fit(X_trainval, y_trainval)
test_score = svm.score(X_test, y_test)

print("Best score on validation set: {:.2f}".format(best_score))
print("Best parameters: ", best_parameters)
print("Test set score with best parameters: {:.2f}".format(test_score))
```

Grid Search + Cross Validation

using cross validation to evaluate the performance (of each parameter combination)

In a nutshell - you are testing multiple combinations of parameters (grid search) - but for each combination, you are building multiple models using different sections of the (training + validation) set to train and test the model.

```
for gamma in [0.001, 0.01, 0.1, 1, 10, 100]:
    for C in [0.001, 0.01, 0.1, 1, 10, 100]:
        # for each combination of parameters,
        # train an SVC
        svm = SVC(gamma=gamma, C=C)
        # perform cross-validation
        scores = cross_val_score(svm, X_trainval, y_trainval, cv=5)
        # compute mean cross-validation accuracy
        score = np.mean(scores)
```

```
        # if we got a better score, store the score and parameters
        if score > best_score:
           best_score = score
           best_parameters = {'C': C, 'gamma': gamma}

# rebuild a model on the combined training and validation set
svm = SVC(**best_parameters)
svm.fit(X_trainval, y_trainval)
```

GridSearchCV class implements grid search with cross-validation in the form of an estimator have to specify the parameters you want to search over AS A DICTIONARY

- dictionary keys = the names of the parameters you want to adjust

```
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100],
                        'gamma': [0.001, 0.01, 0.1, 1, 10, 100]}

from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC
grid_search = GridSearchCV(SVC(), param_grid, cv=5)

X_train, X_test, y_train, y_test = train_test_split(
       iris.data, iris.target, random_state=0)
```

when you call fit on the grid_search object, it runs cross-validation for each combination of parameters in “param_grid”

```
grid_search.fit(X_train, y_train)
```

access the model with the best parameters trained on the whole training set using the best_estimator_
attribute:

```
grid_search.best_estimator_
```

Fitting the GridSearchCV object not only searches for the best parameters, but also automatically fits a new model on the whole training dataset with the parameters that yielded the best cross-validation performance

You can call the predict(self, X) method
= Calls predict on the estimator with the best found parameters.

Analysing the result

cv_results_ attribute = a dictionary
best to convert this to a pandas DataFrame

```
import pandas as pd
# convert to DataFrame
results = pd.DataFrame(grid_search.cv_results_)
# show the first 5 rows
display(results.head())
```

each row = one particular parameter setting
for each setting,
the results of all cross-validation splits are recorded, the mean, the standard deviation

Conditional parameters

Sometimes your model will only use certain parameters
param_grid can be a list of dictionaries

```
param_grid = [{'kernel': ['rbf'],
               'C': [0.001, 0.01, 0.1, 1, 10, 100],
               'gamma': [0.001, 0.01, 0.1, 1, 10, 100]},
              {'kernel': ['linear'],
               'C': [0.001, 0.01, 0.1, 1, 10, 100]}]
```

In the first grid, the kernel parameter is always set to 'rbf' (not that the entry for kernel is a list of length one), and both the C and gamma parameters are varied.
In the second grid, the kernel parameter is always set to linear, and only C is varied.

Nested cross-validation

using multiple splits of cross-validation,

outer loop → loops over splits of the data into training and test sets
for each outer split, the test set score using the best settings is reported.

outer loop ⇒ e.g. 5 - CV, splitting the total dataset into folds
you now have the dataset divided into 5 folds
for fold 1 = test set, folds 2 - 5 as training set
then you do the gridsearchCV on the training set (as in splitting it again into folds.. building models with every parameter combination using a different fold as a testing fold etc etc)

then for fold 2 = test set, folds 1, 3 -5 as training set
do gridsearchCV on the training set (1, 3 - 5)

the result of this procedure is a list of scores—not a model, and not a parameter set- ting. The scores tell us how well a model generalizes, given the best parameters found by the grid.

```
iris.data, iris.target, cv=5)
print("Cross-validation scores: ", scores)
print("Mean cross-validation score: ", scores.mean())
```

Parallelising cross-validation + grid-search

Each time a model is built using a particular parameter setting on a particular cross-validation split can be done independently → ideal candidates for parallelization over multiple CPU cores or over a cluster

setting the n_jobs parameter to the number of CPU cores you want to use.
You can set n_jobs=-1 to use all available cores.

Standardisation

https://medium.com/greyatom/why-how-and-when-to-scale-your-features-4b30ab09db5e

since, most of the machine learning algorithms use Eucledian distance between two data points in their computations, this is a problem. If left alone, these algorithms only take in the magnitude of features neglecting the units.

The features with high magnitudes will weigh in a lot more in the distance calculations than features with low magnitudes.

for regularization to be meaningful - features must be standardised

fit the standardiser on the *training set only*, then apply it to the test set

With cross-validation, the training and test set changes at each iteration. This means we must fit the scaler for each new training set in each iteration. We will achieve this by using a pipeline.

Pipelines allow us to sequentially perform a list of transformations and a final estimator.

https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html
https://towardsdatascience.com/a-simple-example-of-pipeline-in-machine-learning-with-scikit-learn-e726ffbb6976

RandomSearchCV

Nested CV

Decision Trees (Week 4)

Strengths:
- easily visualised
- invariant to scaling ⇒ no normalization or standardization needed

Weaknesses:
- tend to overfit, poor generalization performance
- tree-based regression models can't extrapolate

learn a hierarchy of if/else questions, leading to a decision

top node = root, represents the whole dataset - begin with a root node
each node → represents a question ("*tests*") → represented as continuous features (is feature i larger than value a?)
EACH TEST CONCERNS ONLY A SINGLE FEATURE
we split the data based on the feature that provides the most information about the response variable (the form of test depends on the data type of the chosen variable)
- if the feature of choice is continuous, the test will be $X_i > a$ (is feature i larger than some value a)
- if the feature is categorical → test will be $X_i = c$?

result is two children nodes → one for a
terminal node/leaf = answer

"learn these models from data using supervised learning"
- searches over all possible tests
- picks the one that is most informative about the target variable

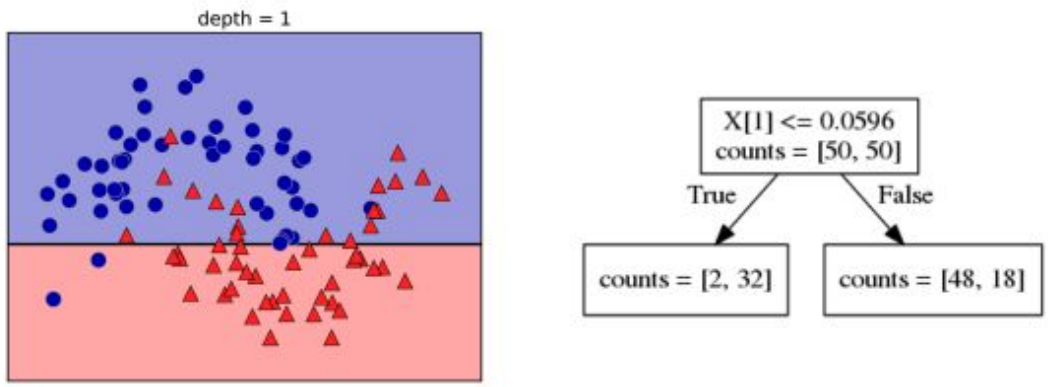


Figure 2-24. Decision boundary of tree with depth 1 (left) and corresponding tree (right)

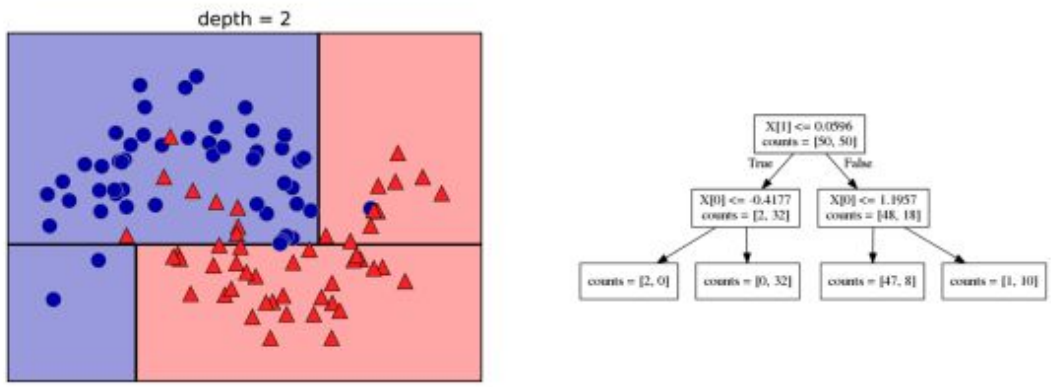


Figure 2-25. Decision boundary of tree with depth 2 (left) and corresponding decision tree (right)

recursive process !

each test splits the part of the data that is currently being considered along one axis

builds a hierarchical partition, hierarchy of if/else questions, segment the feature space

recursive partitioning of the data is repeated until each region in the partition (each leaf) only contains a single value (class or regression value)

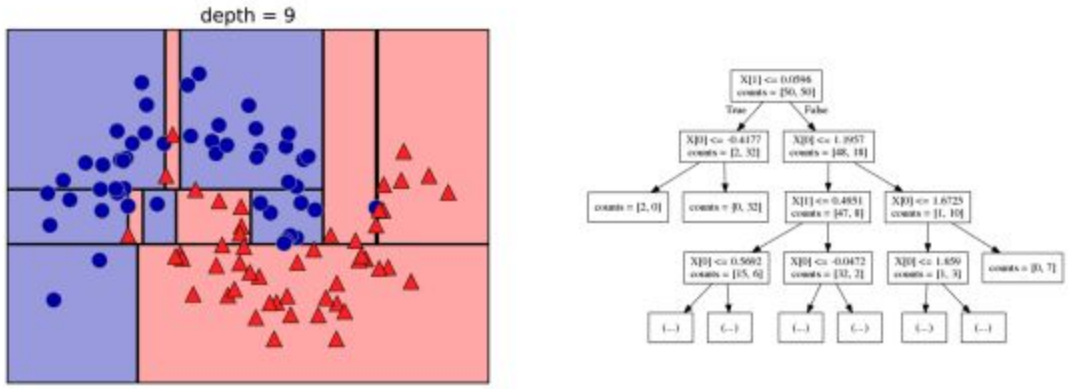


Figure 2-26. Decision boundary of tree with depth 9 (left) and part of the corresponding tree (right); the full tree is quite large and hard to visualize

Predictions on new data points:
checks which region of the partition of the feature space the point lies on, gives the majority class (or the mean of the regression values, for regression)

if you continue until all leaves are pure → models are very complex, overfit to training data
unprune trees are prone to overfitting and not generalizing well to new data
becomes arbitrarily deep and complex

To prevent overfitting:
stop the creation of the tree early = *pre-pruning*

- limiting the maximum depth of the tree
- limiting the maximum number of leaves

building the tree but then removing or collapsing nodes that contain little information = *post-pruning/pruning*

implemented in DecisionTreeRegressor, DecisionTreeClassifier classes
only implements pre-pruning

```
from sklearn.tree import DecisionTreeClassifier

cancer = load_breast_cancer()
```

```
X_train, X_test, y_train, y_test = train_test_split(
cancer.data, cancer.target, stratify=cancer.target, random_state=42)

tree = DecisionTreeClassifier(random_state=0)
tree.fit(X_train, y_train)

print("Accuracy on training set: {:.3f}".format(tree.score(X_train,
y_train)))
print("Accuracy on test set: {:.3f}".format(tree.score(X_test, y_test)))
```

Out[58]:
Accuracy on training set: 1.000
Accuracy on test set: 0.937
As expected, the accuracy on the trai

accuracy on the training set = 100%, because the leaves are pure
tree grown deep enough that it can perfectly memorize all the labels on the training data

to pre-prune
To limit the depth, use max-depth = the number of consecutive questions that can be asked
max_leaf_nodes
min_samples_leaf

In[59]:

```
tree = DecisionTreeClassifier(max_depth=4, random_state=0)
tree.fit(X_train, y_train)
print("Accuracy on training set: {:.3f}".format(tree.score(X_train,
y_train)))
print("Accuracy on test set: {:.3f}".format(tree.score(X_test, y_test)))
```

Out[59]:
Accuracy on training set: 0.988
Accuracy on test set: 0.951

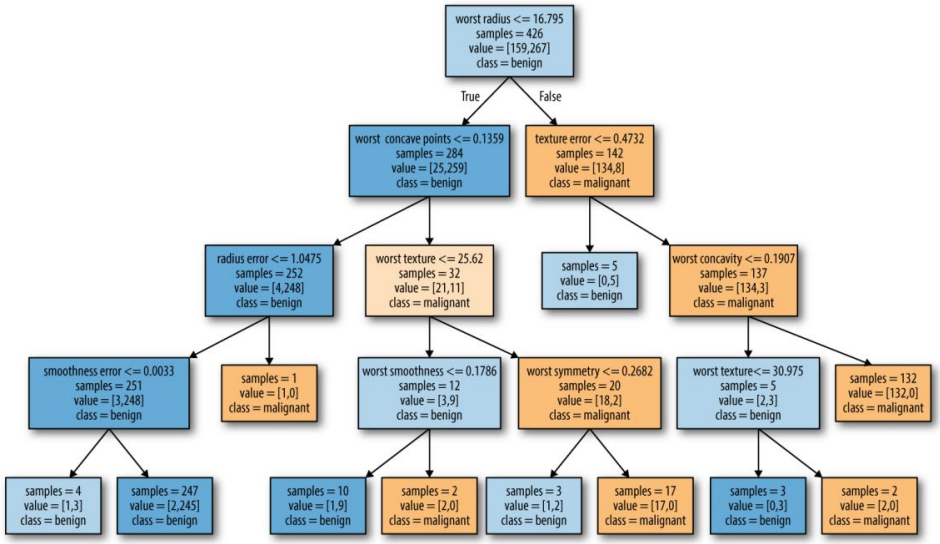
Analyzing decision trees:

use export_graphviz in sklearn.tree module
creates a file in the .dot format = text file format for storing graphs
colour the nodes based on the majority class in each node, pass the class and feature names

In[61]:
```
from sklearn.tree import export_graphviz
export_graphviz(tree, out_file="tree.dot", class_names=["malignant", "benign"],
feature_names=cancer.feature_names, impurity=False, filled=True)
```
We can read this file and visualize it, as seen in Figure 2-27, using the graphviz mod-

visualise with graphviz module

```
import graphviz
with open("tree.dot") as f:
dot_graph = f.read()
graphviz.Source(dot_graph)
```



n samples = number of samples
value = number of samples per class ([num malignant, num benign])

Feature importance

feature importance = how important each feature is for a decision the tree makes

0 = not used at all
1 = perfectly predicts the target

feature importances always sum to 1

In[62]:
print("Feature importances:\n{}".format(tree.feature_importances_))

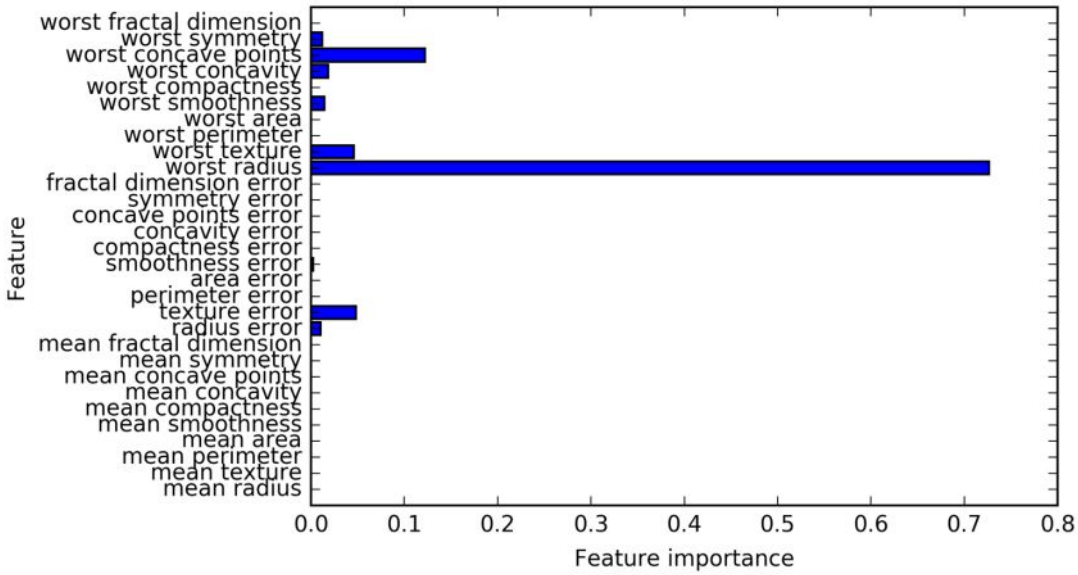
Out[62]:

```
Feature importances:
[ 0.     0.     0.     0.     0.     0.     0.     0.     0.     0.     0.01
  0.048  0.     0.     0.002  0.     0.     0.     0.     0.     0.727  0.046
  0.     0.     0.014  0.     0.018  0.122  0.012  0.   ]
```

visualise the feature importances like this:

In[63]:

```
def plot_feature_importances_cancer(model):
    n_features = cancer.data.shape[1]
    plt.barh(range(n_features), model.feature_importances_, align='center')
    plt.yticks(np.arange(n_features), cancer.feature_names)
    plt.xlabel("Feature importance")
    plt.ylabel("Feature")

plot_feature_importances_cancer(tree)
```



DecisionTreeRegressor/Tree Based Regression models

NB:
tree-based models for regression CANNOT *extrapolate*, make predictions outside the range of the training data

once you extend beyond the data range for which the model has data, the model simply keeps predicting the last known point.

The tree has no ability to generate “new” responses, outside of what was seen in the training data.

Ensembles of Decision Trees

Ensembles = methods that combine multiple machine learning models to create more powerful models

Random Forests

strengths;
- very powerful, work well without tuning parameters
- don't require scaling of the data

weaknesses:
- single decision tree easier to visualise to laypeople
- don't perform well on high-dimensional, sparse data (e.g. text data)
- require more memory, slower to train + predict than linear models

decision trees tend to overfit training data

random forest = a collection of decision trees, where each tree is slightly different

each tree will likely overfit on a part of the data, overfit in different ways

reduce the amount of overfitting by averaging their results

reduction in overfitting while retaining the predictive power of the trees

inject randomness into the tree-building

RandomForestRegressor/RandomForestClassifier
- n_estimators parameter = number of trees to build
- max_features parameter = number of features selected for each test node in each decision tree
    - if set to n_features, means that each split can look at all the features in the dataset, and no randomness will be injected in the feature selection
    - if set to 1: splits have no choice on which feature to test

to build a tree: take a *bootstrap sample* of the data (*bootstrap aggregation/bagging*)
from our n samples, repeatedly draw a sample randomly with replacement n times → creates a dataset as big as the original dataset, but with some points missing/repeated
then a decision tree is built based on this newly created dataset
each decision tree in the random forest
+

Instead of looking for the best test for each node → algorithm randomly selects a subset of the features, THEN looks for the best possible test involving one of these features
each split in each tree operates on a different subset of features (default sqrt(number of features))

overfitted to the randomised dataset, with features that have been taken randomly, on a bootstrapped dataset

To make a prediction:
algorithm makes a prediction for every tree in the forest
for regression: averages these predictions to get a final prediction
for classification:

- soft-voting
- each algorithm provides a probability for each possible output label
- probabilities predicted by all the trees are averaged, class with the highest probability is predicted

Analyzing random forests:

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=100, noise=0.25, random_state=3)
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y,

random_state=42)

forest = RandomForestClassifier(n_estimators=5, random_state=2)

forest.fit(X_train, y_train)
```

The trees built are stored in the estimator_ attribute
To visualise the decision boundaries learned by each tree:

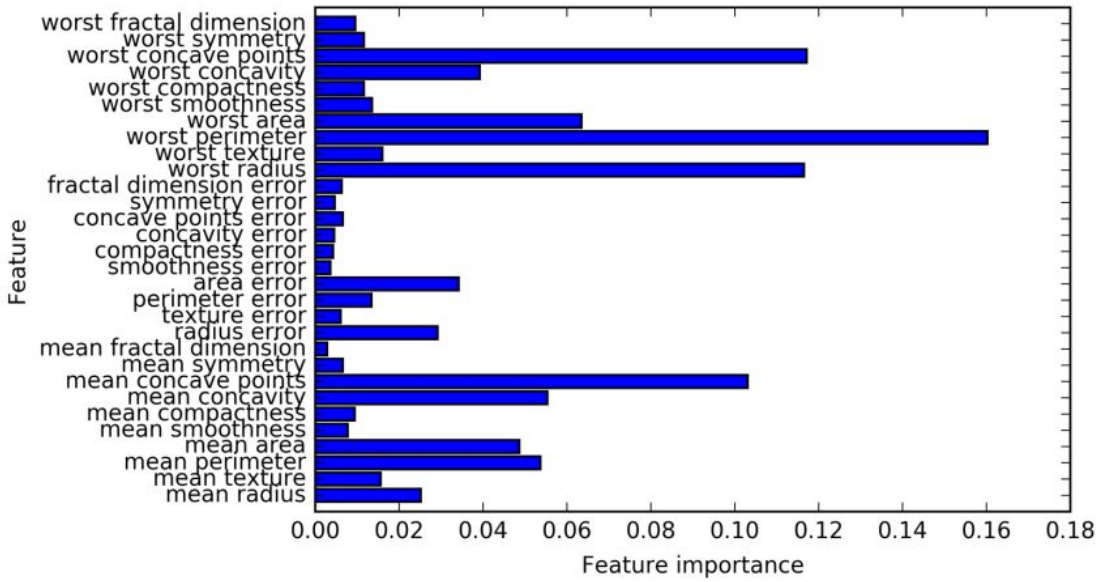
```
fig, axes = plt.subplots(2, 3, figsize=(20, 10))
```

```
for i, (ax, tree) in enumerate(zip(axes.ravel(), forest.estimators_)):
        ax.set_title("Tree {}".format(i))
        mglearn.plots.plot_tree_partition(X_train, y_train, tree, ax=ax)

mglearn.plots.plot_2d_separator(forest, X_train, fill=True, ax=axes[-1,
-1],
                                                        alpha=.4)

axes[-1, -1].set_title("Random Forest")
mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
```

[70]:

```
X_train, X_test, y_train, y_test = train_test_split(
     cancer.data, cancer.target, random_state=0)
forest = RandomForestClassifier(n_estimators=100, random_state=0)
forest.fit(X_train, y_train)

print("Accuracy on training set: {:.3f}".format(forest.score(X_train,
y_train)))
print("Accuracy on test set: {:.3f}".format(forest.score(X_test, y_test)))
```

```
plot_feature_importances_cancer(forest)
```



the randomness in building the random forest captures a much broader picture of data than a single tree

use n_jobs parameter => adjust the number of cores to use (if you have a multi-core processor)
n_jobs = -1 ⇒ uses all the cores in your computer

random_state

Gradient boosted regression trees (gradient boosting machines)

Strengths:

- no scaling

Weaknesses:

- requires careful tuning of the hyperparameters!
- take a long time to train

ensemble method
misnomer: can be used for regression or classification

“Gradient boosting works by building a large number of trees where each tree tries to correct the mistakes of the previous one. The way this is achieved is by fitting each subsequent tree on a modified version of the original dataset, depending on how the previous trees performed. Given the current model, we fit a decision tree to the residuals (the unexplained variation) from the model. That is, we fit a new tree using the current residuals, rather than the whole response Y, as the response. We then add this new decision tree into the fitted function in order to update the residuals.”

builds trees in a serial manner, where each tree tries to correct the mistakes of the previous one
no randomisation → instead strong pre-pruning, very shallow trees (1 - 5) (max_depth parameter is set very low, <= 5) = boosting
fit consecutive trees, each tree must solve for net error from the previous tree

calculates psuedo residuals ⇒
slow learning better ⇒ multiply by “learning rate” = hyperparameter between 0 and 1⇒
each model predicts the residuals produced by the previous model
75 + 8 + 2…..
times the residuals by the learning rate to slow it down
build weak classifiers that improve residuals of the previous models
proven mathematically

gradient boosting = gradient descent + boosting ⇒
- uses the gradient descent algorithm, which can optimise any differentiable loss function
- trees are built one by one and summed sequentially

combines many simple models (*weak learners*) i.e. shallow trees => each tree can only make god predictions on part of the data

learning_rate parameter = controls how strongly each tree tries to correct the mistakes of the previous trees

In[72]:

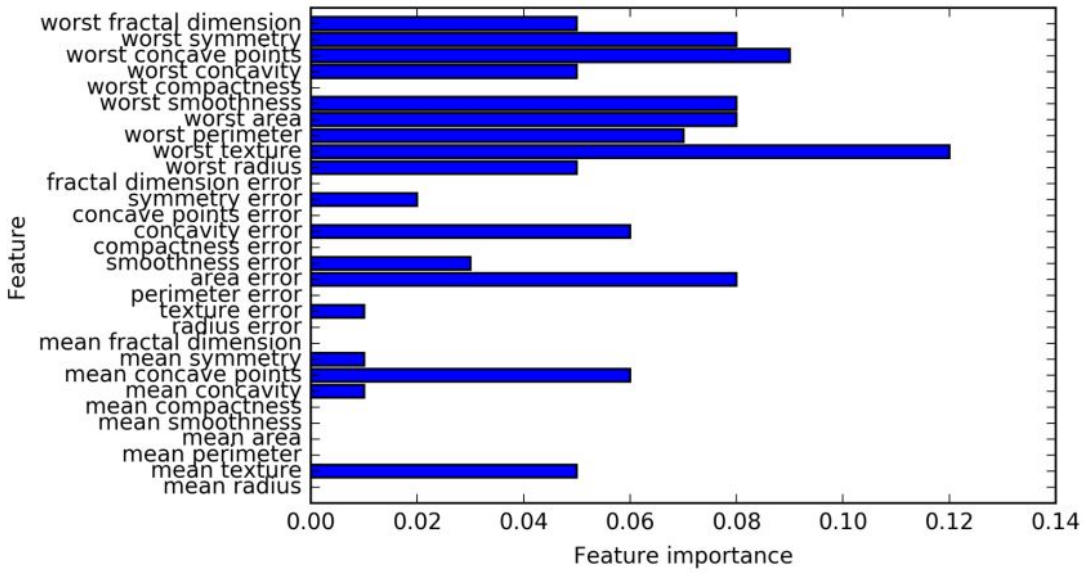
```
from sklearn.ensemble import GradientBoostingClassifier

X_train, X_test, y_train, y_test = train_test_split(
```

```
        cancer.data, cancer.target, random_state=0)

gbrt = GradientBoostingClassifier(random_state=0)
gbrt.fit(X_train, y_train)

print("Accuracy on training set: {:.3f}".format(gbrt.score(X_train,
y_train)))
print("Accuracy on test set: {:.3f}".format(gbrt.score(X_test, y_test)))
```

Out[72]:
Accuracy on training set: 1.000
Accuracy on test set: 0.958

To visualize:

In[75]:
gbrt = GradientBoostingClassifier(random_state=0, max_depth=1)
gbrt.fit(X_train, y_train)



Kernelized Support Vector Machines (Week 5)

strengths:

- powerful, complex decision boundaries

weaknesses:

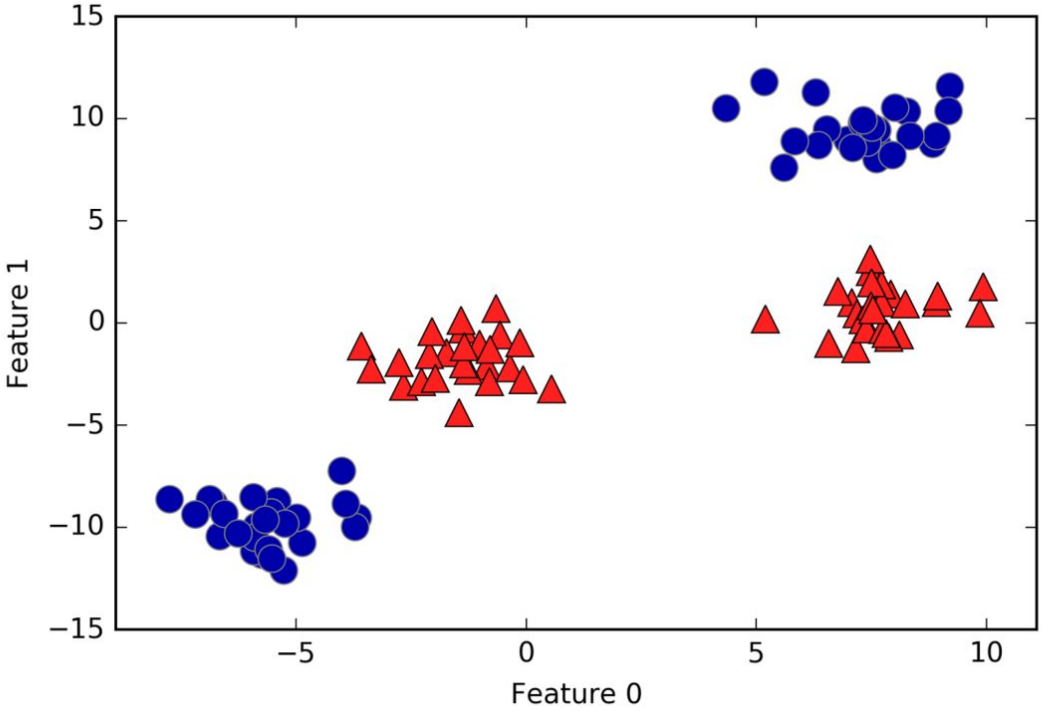
- can handle a large number of features, don't handle large number of samples well (>100,000) → computationally challenging
- requires preprocessing of data (scaling)!! → whereas tree-based models do not (because each decision is made based on a single input feature) ⇒ otherwise it has a huge effect on the boundary

SVMs → want to find a line or a decision boundary that maximises the distance between the closest points to the line and the line itself

kernelized SVMs = different to linear SVMs

instead of representing each data point as 2D
map it to a higher dimension (3d)

say we have a linear model (y = output), with two input features (e.g. height and weight)


15
10
5
0
−5
−10
−15
Feature 1
−5
0
5
10
Feature 0

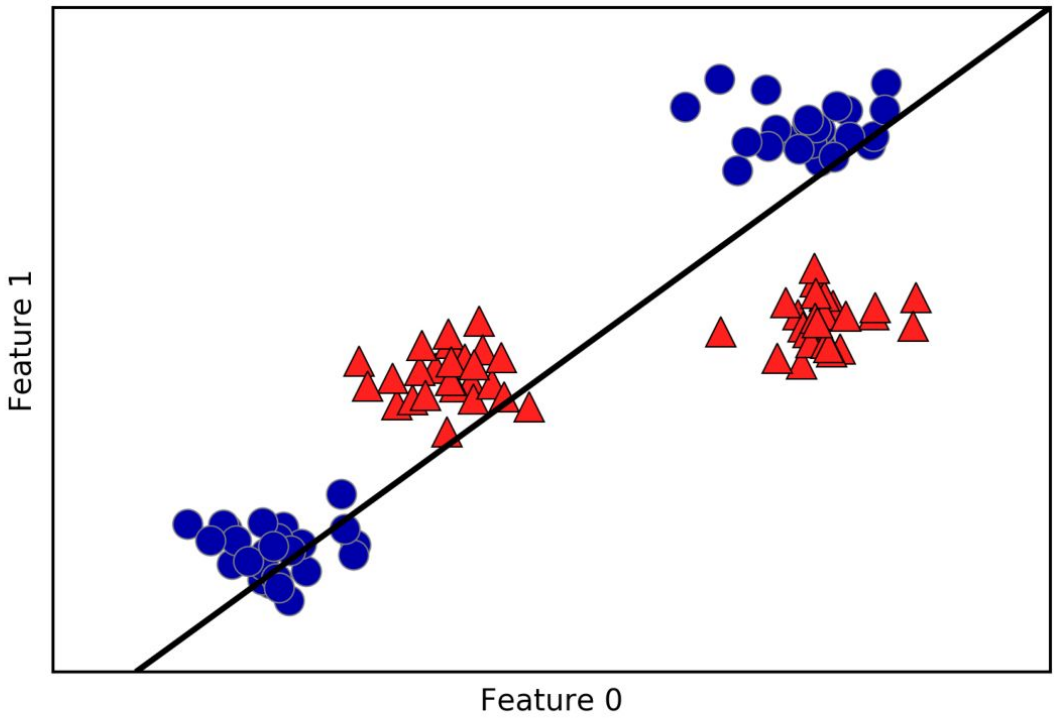


Figure 2-37. Decision boundary found by a linear SVM

you can expand the set of features e.g. by adding $(\text{feature1})^2$ as a new feature, then representing each data point instead of two dimensional (x, y) == (feature0, feature1) as a three-dimensional point (feature0, feature1, feature1^2)

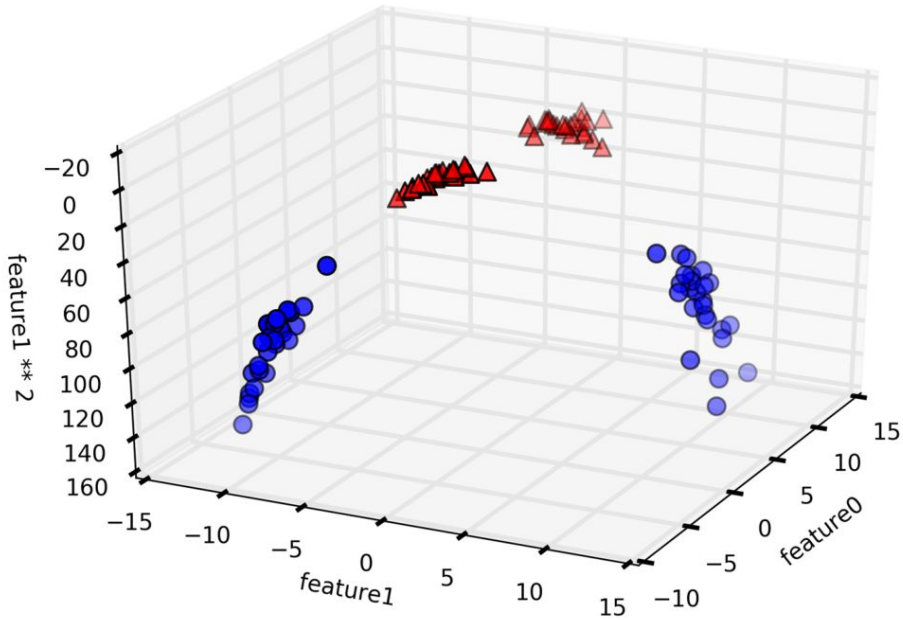


Figure 2-38. Expansion of the dataset shown in Figure 2-37, created by adding a third feature derived from feature1

then find a decision boundary (a plane)

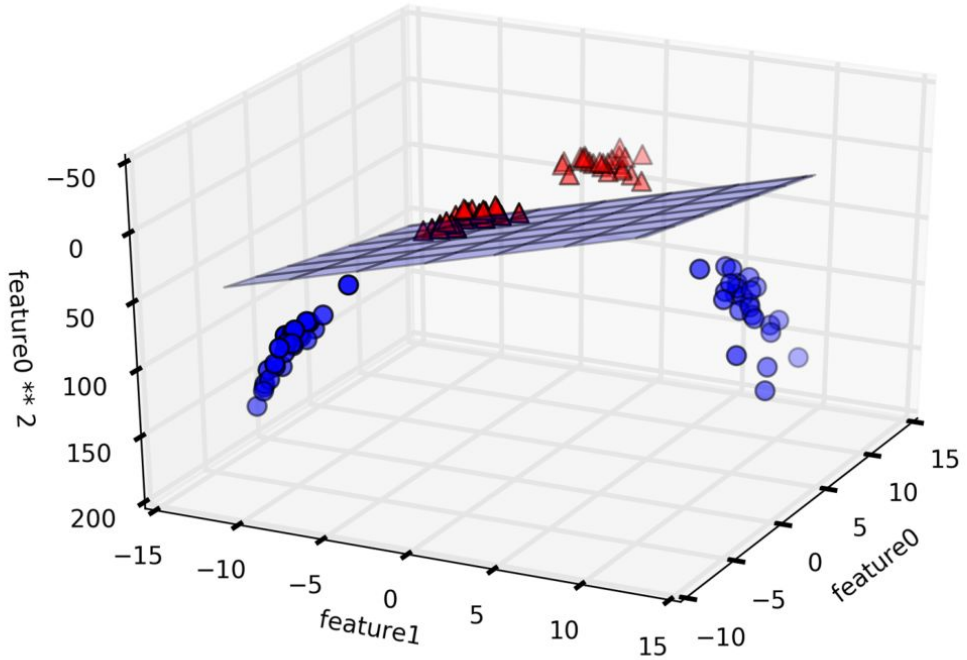


Figure 2-39. Decision boundary found by a linear SVM on the expanded three-dimensional dataset

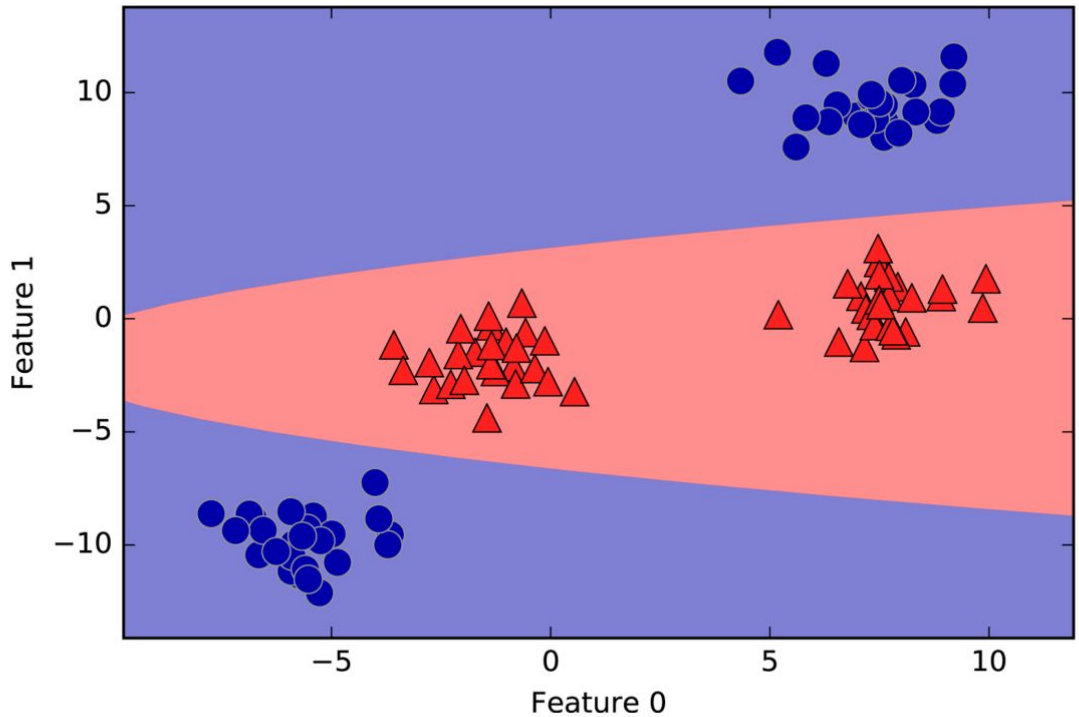


Figure 2-40. The decision boundary from Figure 2-39 as a function of the original two features

creates a non-linear decision boundary by projecting the data through a non-linear function to a space with a higher dimension. Data points which can't be separated by a straight line in their original space I are lifted to a feature space F where there can be a straight hyperplane that separates the data points of one class from another. When that hyperplane is projected back to the input space I, it has the form of a non-linear curve.

hyperplane is represented with the equation $w^T x+b=0$

all points for the class -1 are on one side, all data points for the class 1 are on the other
the closest point from each class to the hyperplane is equal
searches for the maximal margin (separating power) between the classes!

can't guess what feature to add, adding many features (e.g. all possible interactions in a 100-dimensional space) = very computationally expensive!
kernel trick = clever mathematical trick, means you don't have to compute the new, very very large representation → works by directly computing the distance (the scalar products) of the data points for the expanded feature representation (rather than computing the expansion itself)

remember ⇒ the kernel is generating extra features, so the polynomial adds features to a power, guassian kernel adds polynomials considering an infinite number….

<u>polynomial kernel</u> = computes all possible polynomials up a certain degree of the original features (feature1^2 x feature2)

<u>radial basis function/Gaussian kernel</u> = infinite dimensional feature space, considers all possible polynomials of all degrees, but the importance of the features decreases for higher degrees

$$K(x, x') = \exp\left(-\frac{\|x - x'\|^2}{2\sigma^2}\right)$$

while trained

- SVM learns how important each of the training data points is to represent the decision boundary between the two classes
- only the data points that lie on the border between the classes (a subset of all the data points) usually matter (*support vectors*)

to make a prediction for a new point →
the distance to each of the support vectors is measured
distance to the support vector * importance of the support vector learned during training (stored in the dual_coef_ attribute)

$$k_{\text{rbf}}(x_1, x_2) = \exp\left(\gamma\|x_1 - x_2\|^2\right)$$

x_1 - x_2 = Euclidean distance, gamma = controls the width of the Gaussian kernel

```
from sklearn.svm import SVC
X, y = mglearn.tools.make_handcrafted_dataset()
svm = SVC(kernel='rbf', C=10, gamma=0.1).fit(X, y)
mglearn.plots.plot_2d_separator(svm, X, eps=.5)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
# plot support vectors
```

```
sv = svm.support_vectors_
# class labels of support vectors are given by the sign of the dual
coefficients
sv_labels = svm.dual_coef_.ravel() > 0
mglearn.discrete_scatter(sv[:, 0], sv[:, 1], sv_labels, s=15,
markeredgewidth=3)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

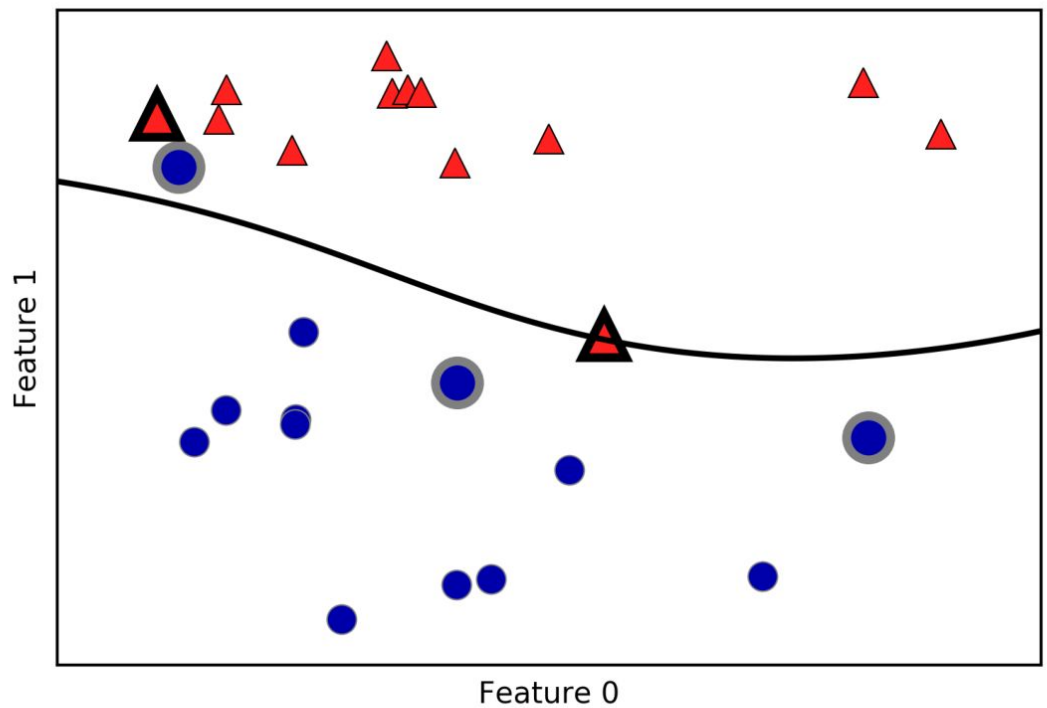


Figure 2-41. Decision boundary and support vectors found by an SVM with RBF kernel

parameters

gamma ⇒ controls the width of the Gaussian kernel
determines the scale of what it means for points to be close together

C ⇒ regularization parameter, limits the importance of each point (i.e. dual_coef_)

```
fig, axes = plt.subplots(3, 3, figsize=(15, 10))

for ax, C in zip(axes, [-1, 0, 3]):
for a, gamma in zip(ax, range(-1, 2)):
mglearn.plots.plot_svm(log_C=C, log_gamma=gamma, ax=a)

axes[0, 0].legend(["class 0", "class 1", "sv class 0", "sv class 1"],
ncol=4, loc=(.9, 1.2))
```

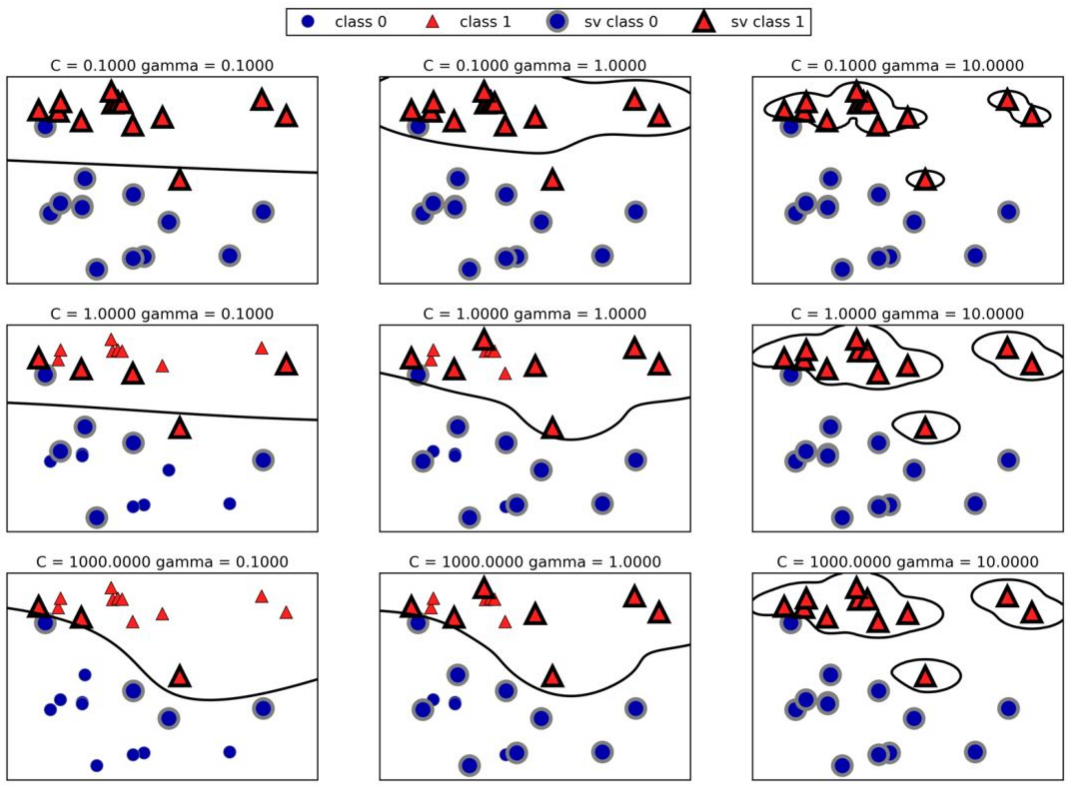


Figure 2-42. Decision boundaries and support vectors for different settings of the parameters C and gamma

small gamma = large radius for the Gaussian kernel, many points are considered close by, decision boundary will vary slowly, model will have a low complexity
high value of gamma = more complex model
smooth decision boundaries left, boundaries focusing on single points further to the right

small C = very restricted model, each data point has a restricted amount of influence
increasing C ⇒ each point has a stronger influence on the model, makes the decision boundary bend to classify them correctly

NB: features of different order of magnitude = problem for linear models, DEVASTATING for kernel SVM

Preprocessing data for SVMs

Rescale each feature so they are on the same scale

e.g. scale the data such that all features are between 0 and 1

by hand

```
# compute the minimum value per feature on the training set
min_on_training = X_train.min(axis=0)
# compute the range of each feature (max - min) on the training set
range_on_training = (X_train - min_on_training).max(axis=0)

# subtract the min, and divide by range
# afterward, min=0 and max=1 for each feature
X_train_scaled = (X_train - min_on_training) / range_on_training
print("Minimum for each feature\n{}".format(X_train_scaled.min(axis=0)))
print("Maximum for each feature\n {}".format(X_train_scaled.max(axis=0)))
```

Output:

```
Minimum for each feature
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
Maximum for each feature
 [ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
   1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]
```

In[86]:

```
# use THE SAME transformation on the test set,
# using min and range of the training set (see Chapter 3 for details)
X_test_scaled = (X_test - min_on_training) / range_on_training
```

In[87]:

```
svc = SVC()
svc.fit(X_train_scaled, y_train)

print("Accuracy on training set: {:.3f}".format(
svc.score(X_train_scaled, y_train)))
print("Accuracy on test set: {:.3f}".format(svc.score(X_test_scaled,
y_test)))
```

Out[87]:
Accuracy on training set: 0.948
Accuracy on test set: 0.951

do this automatically with MinMaxScaler

In[3]:

```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
cancer = load_breast_cancer()

X_train, X_test, y_train, y_test = train_test_split(cancer.data,
cancer.target,
random_state=1)

print(X_train.shape)
print(X_test.shape)
```

Out[3]:
(426, 30)
(143, 30)

import the class that implements the preprocessing (MinMaxScaler), then instantiate it:

```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
```

then fit the scaler using the *fit* method, to the training data
MinMaxScaler → fit method computes the minimum and maximum value of each feature on the training set, doesn't require y_train, only x_train

In[5]:
scaler.fit(X_train)
Out[5]:
MinMaxScaler(copy=True, feature_range=(0, 1))

then to actually scale the training data - use the transform method of the scaler

```
# transform data
X_train_scaled = scaler.transform(X_train)

# print dataset properties before and after scaling
print("transformed shape: {}".format(X_train_scaled.shape))
print("per-feature minimum before scaling:\n
{}".format(X_train.min(axis=0)))
print("per-feature maximum before scaling:\n
{}".format(X_train.max(axis=0)))
print("per-feature minimum after scaling:\n {}".format(
X_train_scaled.min(axis=0)))
print("per-feature maximum after scaling:\n {}".format(
X_train_scaled.max(axis=0)))
```

```
transformed shape: (426, 30)
per-feature minimum before scaling:
 [  6.98    9.71   43.79  143.50    0.05    0.02    0.      0.      0.11
    0.05    0.12    0.36    0.76    6.80    0.      0.      0.      0.
    0.01    0.      7.93   12.02   50.41  185.20    0.07    0.03    0.
    0.      0.16    0.06]
per-feature maximum before scaling:
 [  28.11    39.28   188.5   2501.0     0.16     0.29     0.43     0.2
    0.300    0.100    2.87     4.88    21.98   542.20     0.03     0.14
```

```
    0.400    0.050    0.06     0.03    36.04    49.54   251.20  4254.00
    0.220    0.940    1.17     0.29     0.58     0.15]
per-feature minimum after scaling:
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
   0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
per-feature maximum after scaling:
 [ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
   1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]
```

features are now between 0 and 1, as desired

To apply the SVM to the scaled data → you also need to transform the test set (call the transform method again on X_test)

In[7]:

```
# transform test data
X_test_scaled = scaler.transform(X_test)

# print test data properties after scaling
print("per-feature minimum after
scaling:\n{}".format(X_test_scaled.min(axis=0)))
print("per-feature maximum after
scaling:\n{}".format(X_test_scaled.max(axis=0)))
```

Out[7]:

```
per-feature minimum after scaling:
[ 0.034  0.023  0.031  0.011  0.141  0.044  0.     0.     0.154 -0.006
 -0.001  0.006  0.004  0.001  0.039  0.011  0.     0.    -0.032  0.007
  0.027  0.058  0.02   0.009  0.109  0.026  0.     0.    -0.    -0.002]
per-feature maximum after scaling:
[ 0.958  0.815  0.956  0.894  0.811  1.22   0.88   0.933  0.932  1.037
  0.427  0.498  0.441  0.284  0.487  0.739  0.767  0.629  1.337  0.391
  0.896  0.793  0.849  0.745  0.915  1.132  1.07   0.924  1.205  1.631]
```

some features are outside the 0 - 1 range → because all scalers always apply the exact same transformation to the training and test sets

One-class Support Vector Machines

http://rvlasveld.github.io/blog/2013/07/12/introduction-to-one-class-support-vector-machines/

when you only have one class and the goal is to test new data and find out whether it is alike or not to the training data/that class

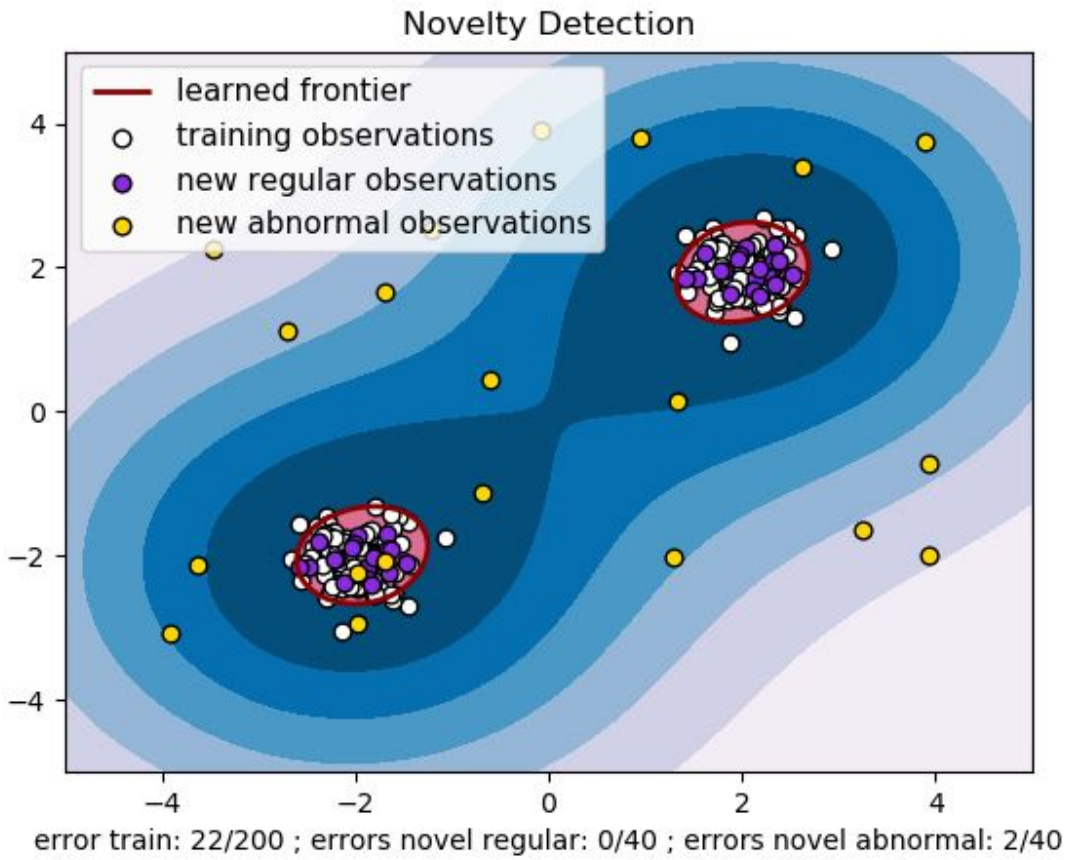
easy to gather training data of situations that are OK → e.g. in a factory
very difficult to gather example data of a faulty system, no way to guarantee that all faulty states are simulated and thus recognised in a two-class classification problem

algorithm creates a representational model of this data
if the newly encountered data is too different (according to some measurement) → it is labelled as out-of-class

separates ALL the data points from the origin (in the feature space F)
maximises the distance from this hyperplane to the origin

produces a
binary function which captures regions in the input space where the probability density of the data lives
function returns +1 in the small region (with all the training data points)
and -1 elsewhere

sklearn.svm.OneClassSVM
= unsupervised outlier detection



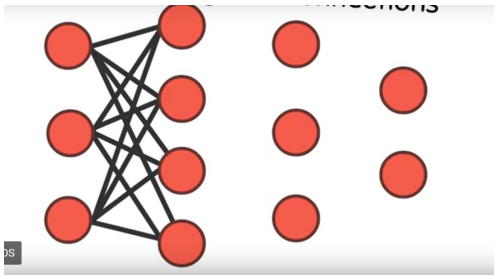
example in scikit-learn:

https://scikit-learn.org/stable/auto_examples/svm/plot_oneclass.html

Artificial Neural Networks (Chapter 6)

You have to edit these!
Question: why not take the double derivative to find the global minimum of the loss function? → it took years just to get the backpropagation algorithm working

neural network = black box
many neurons = basic unit
neurons are broken up into layers
connected →
every neuron of one layer interacts with the neurons of the next layer through weighted connections



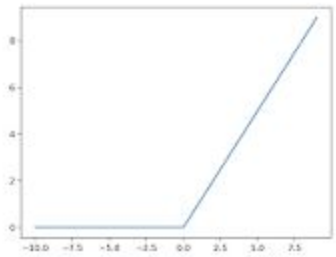
neuron takes the value of a connected neuron and multiplies it by the connection's weight (then passes it through an activation function, which converts it)

<u>Videos:</u>

Inside each neuron

1. each input is multiplied by a weight (w1 * x1, w2*x2)
2. All the weighted inputs are added together with a bias b ((w1*x1) + (w2*x2) + b)
3. the result is transformed through an activation function (relu, sigmoid, tahn…) = f(w1*x1 + w2*x2 + b)

ReLU = rectified linear unit



$$\sigma(x) = \max\{0, x\} \quad \text{- rectified linear unit (ReLU)} \tag{4.2a}$$

$$\sigma(x) = \frac{1}{1 + \exp(-2x)} \quad \text{- logistic} \tag{4.2b}$$

$$\sigma(x) = \tanh(x) \quad \text{- hyperbolic tangent.} \tag{4.2c}$$

plethora of other activation functions →
activation function replicates threshold for depolarisation to generate an action potential in a neuron → for values below a critical threshold, outputs remain at 0

neural network is made up of basic units called neurons →
neurons are grouped into layers
middle layers = hidden layers
layers are connected via weighted values (e.g. w1, w2)
each neuron receives a value which is multplied by weight etc (above). Activation function is just a non-linear function

weights are initialised as random
with the weights we're going to have some predictions
want to minimise the loss function to have the best classifier
loss functions = maximum likelihood, cross-entropy
go through optimiser, update the weights → iterate, each time hopefully we have better weights
ANN → rich in hyperparameters, one of the drawbacks of deep learning
learning rate = how fast I am learning

Deep learning →
- subfield of ML
- emphasis on learning SUCCESSIVE LAYERS OF INCREASINGLY MEANINGFUL REPRESENTATIONS (learned from exposure to data)
- depth of the model = number of layers (10000 + *successive layers of representations*)
- other ML models = 1 or 2 layers = shallow learning

neural networks → structured in literal layers stacked on top of each other

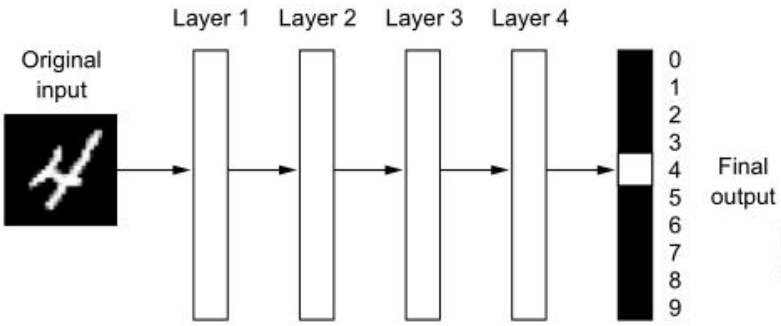
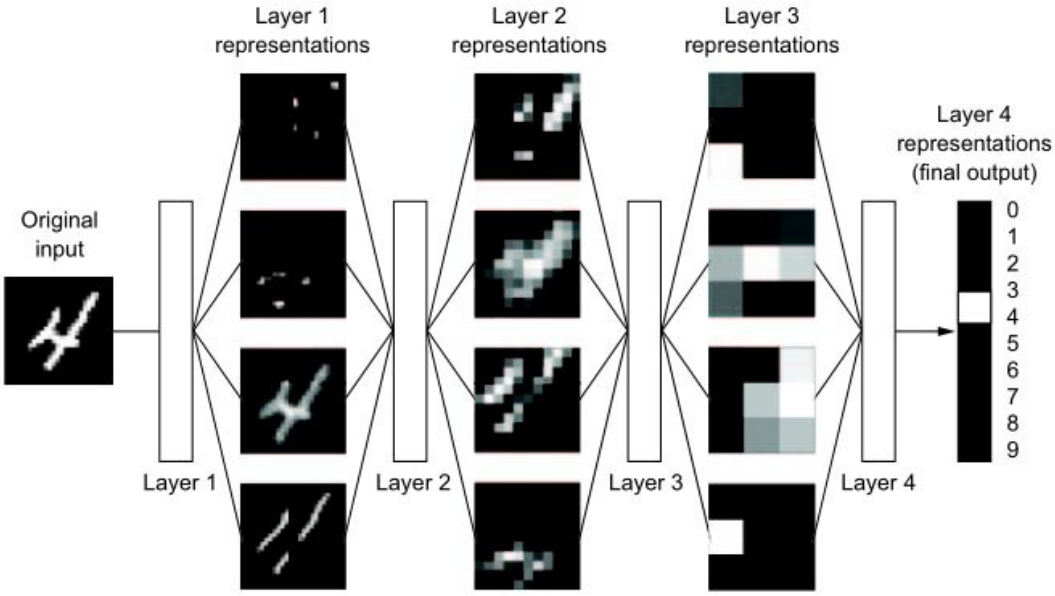


Figure 1.5 A deep neural network for digit classification

transforms input into representations that are increasingly different from the original, increasingly informative about the final result (the target/output)



input to target mapping via a deep sequence of simple data transformations (layers)

specification about what a layer does to its input data is stored in the layer's weights
= a bunch of numbers
transformation is parameterized by its weights

loss function/objective function → takes the predictions of the network and the true target/output, computes a distance score (how well the network has done)

$$\ell(x,y) = \frac{1}{2}(x-y)^2 \qquad \text{- quadratic loss } (\ell_2)$$
$$\ell(x,y) = |x-y| \qquad \text{- absolute loss } (\ell_1)$$
$$\ell(x,y) = \max\{0, 1-xy\} \qquad \text{- hinge loss.}$$

use the score as a feedback signal → to adjust the value of the weights in the direction that will lower the loss score = *optimiser* (implementing a backpropagation algorithm)

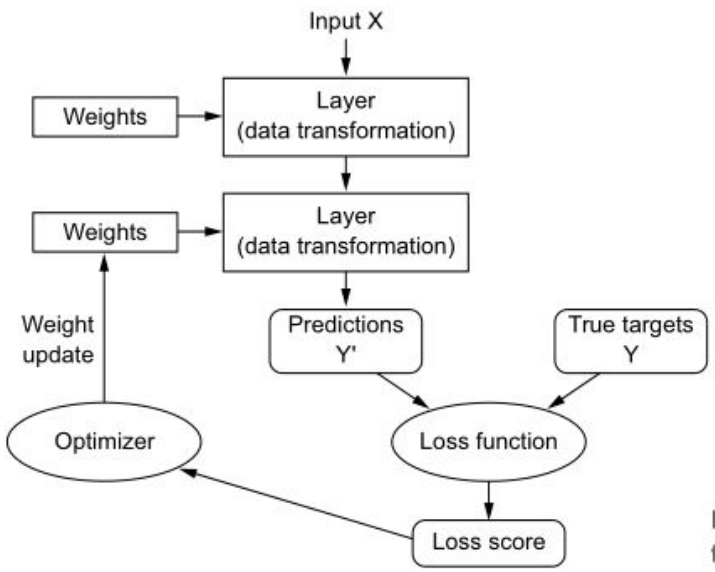


Figure 1.9 The loss score is used as a feedback signal to adjust the weights.

iterative
training loop
initially → weights are random, implements a series of random transformations
with every example, weights are adjusted in the correct direction

gradient propagation through deep stacks of layers of representations → feedback signal faded away as the number of layers increased
changed ~ 2009 - 2010 → activation functions, weight-initialization schemes → optimisation schemes

TensorFlow = symbolic tensor-manipulation frameworks for Python that supports auto differentiations (simplifying the implementation of new models)
Keras = user friendly library, makes deep learning as easy as manipulating LEGO bricks

MNIST dataset = “Hello World” of deep learning

To load the MNIST dataset:

```
from keras.datasets import mnist

(train_images, train_labels), (test_images, test_labels) =
mnist.load_data()
#training set                          #test set
```

images encoded as numpy arrays

To build the network:

```
from keras import models
from keras import layers

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

layer = data-processing module = filter for data
some data goes in → comes out in a more useful form (extract *representations* out f the data fed into them → like a sieve
chain together simple layers → *progressive data distillation*

two *Dense*/fully connected neural layers.

Last layer = 10-way softmax layer = will return an array of 10 probability scores (summing to 1) = each score is the probability that the current digit image belongs to one of our 10 digit classes

we will need:

- loss function → so the network can measure its own performance, steer itself in the right direction
- optimiser → how the neckwork updates itself based on data + loss function
- metrics to monitor (training + testing) ⇒ e.g. accuracy

```
network.compile(optimizer='rmsprop',
                loss='categorical_crossentropy',
                metrics=['accuracy'])
```

preprocess data: scaling all values to the [0, 1] interval

```
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype('float32') / 255

test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype('float32') / 255
```

categorically encode the labels:

from keras.utils import to_categorical

```
train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```

To actually train the network:

```
network.fit(train_images, train_labels, epochs=5, batch_size=128)
```

will return the loss of the network in training the data, and the training accuracy. To get test accuracy and test loss:

```
 test_loss, test_acc = network.evaluate(test_images, test_labels)
 print('test_acc:', test_acc)
```

Data Representations

Tensors

- all ML systems use tensors as their basic data structure
- = a container for data (numerical data) → number container →
- generalization of matrices to an arbitrary number of dimensions (aka axis)

Scalars = 0D tensors

- contains only one number
- float32 or float64 = scalara tensor/array

- ndim attribute = the number of axes (x = np.array(12), x.ndim #0)

Vectors = 1D tensors
- array of numbers
- exactly 1 axis
- e.g. vector data in research (samples, features)

```
>>> x = np.array([1, 2, 4, 20])
>>> x
array ([1, 2, 4, 20])
>>> x.ndim
1
```

NB: dimensionality = either the number of entries along a specific axis (above = 4 dimensional array) or the number of axes in a tensor

Matrices = 2D tensors
- two axes (rows and columns)
- visually interpreted as a rectangular grid of numbers

```
>>> x = np.array([[5, 78, 2, 34, 0],
                  [6, 79, 3, 35, 1],
                  [7, 80, 4, 36, 2]])
>>> x.ndim
2
```

3+D
- array of matrices = 3D tensor (cube of numbers)
- 3D tensors in an array = 4D tensor…

Image = 4D tensor (shape = (samples, height, width, channels))

ndim = number of axes, shape = tuple describing dimensions along each axis, data type (dtype) = type of data in the tensor (float32, uint8, float64, char…)

train_images = tensor

```
>>> print(train_images.ndim)
3
```

```
>>> print(train_images.shape)
(60000, 28, 28)
```

```
>>> print(train_images.dtype)
uint8
```

3D tensor of 8-bit images ⇒ array of 60 000 matrices of 28 x 8 integers (grascale image, coefficients from 0 - 255)

Manipulating Tensors

```
digit = train_images[4]
import matplotlib.pyplot as plt
plt.imshow(digit, cmap=plt.cm.binary)
plt.show()
```

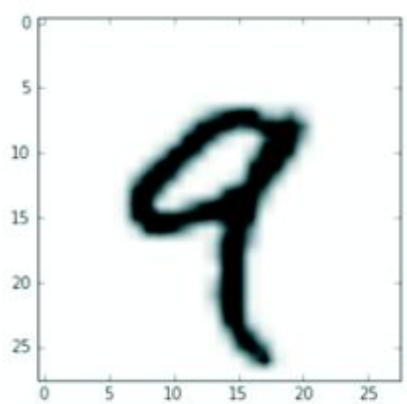


Figure 2.2 The fourth sample in our dataset

selecting a specific digit along an axis = *tensor slicing* (selects digits #10 to #99)

```
>>> my_slice = train_images[10:100]
>>> print(my_slice.shape)
(90, 28, 28)
```

: → equivalent to selecting the entire axis!!!!

```
>>> my_slice = train_images[10:100, :, :]
>>> my_slice.shape
(90, 28, 28)
>>> my_slice = train_images[10:100, 0:28, 0:28]
>>> my_slice.shape
(90, 28, 28)
```

Equivalent to the previous example

Also equivalent to the previous example

you can select any 2 indices along each tensor axis: e.g. to select 14 x 14 pixels in the bottom right corner of all images:

```
my_slice = train_images[:, 14:, 14:]
```

negative indices = position relative to the end of the current axis

```
my_slice = train_images[:, 7:-7, 7:-7]
```

Data batches

first axis = the samples axis (samples dimension)
deep-learning models don't process an entire dataset at once → break the data into small batches

(Vector data

each single data point (e.g. patient) → encoded as a vector
batch of data = 2D tensor (array of vectors)
first axis = samples axis, second axis = features axis)

Tensor operations

Each neural layer transforms its input data like this:

```
output = relu(dot(W, input) + b)
```

lost it after this point… who cares about tensor operations…

w & b = attributes of the layer
weights, trainable parameters (kernel and bias attributes)
weight matrices are filled with small random values = random initialization
resulting representations are meaningless
gradually adjust these weights based on feedback signal → graudal adjustment = training
training loop

1. draw a batch of training samples X_train and outputs y_train
2. run the network on X_train = forward pass, to obtain predictions y_pred
3. compute the loss of the network on the batch (difference between y_pred and y)
4. update all the weights of the network in a way that reduces the loss

evnetually → network with low loss on training data

ALL OPERATIONS ARE DIFFERENTIABLE
COMPUTE THE GRADIENT OF THE LOSS with respect to the networks coefficients
move the coefficients in the opposite direction from the gradient (decreasing loss)
(gradient = m = rise/run of the line that intersects with that point of the curve)

take the derivative of the tensor operation → f'(W) → move in the opposite direction to reduce f(W)
find all the points where the derivative = 0 = combination of weight values that yields the smallest possible loss function
f(W) = 0

1. Draw a batch of training samples `x` and corresponding targets `y`.
2. Run the network on `x` to obtain predictions `y_pred`.
3. Compute the loss of the network on the batch, a measure of the mismatch between `y_pred` and `y`.
4. Compute the gradient of the loss with regard to the network's parameters (a *backward pass*).
5. Move the parameters a little in the opposite direction from the gradient—for example `W -= step * gradient`—thus reducing the loss on the batch a bit.

= mini-batch stochastic gradient descent

- stochastic = each batch of data is drawn at random

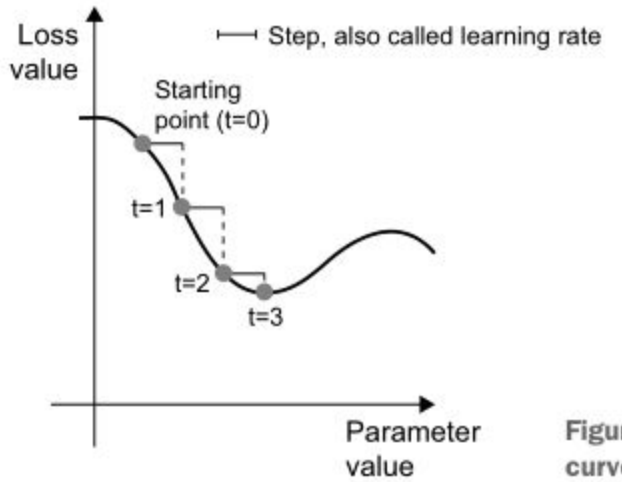


Figure 2.11 SGD down a 1D loss curve (one learnable parameter)

must pick a step

- too small → take too many iterations, stuck in local minimum

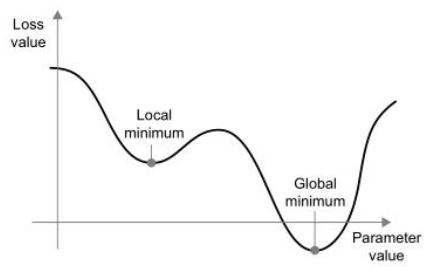


Figure 2.13 A local minimum and a global minimum

-

- too large → updates take you to completely random locations on the curve

true SGD = drawing a single sample and target at each iteration

batch SGD = run every step on the entire dataset

neural network = many tensor operations chained together → each with a simple, known derivative
chain of functions can be derived using the chain rule!!

dy/dx = dy/dv * dv/dx

```
: f(g(x)) = f'(g(x)) * g'(x)
```

vu' + uv'??

applying the chain rule to the chain of tensor operations → backpropagation/reverse mode differentiation
starts with final loss value

works backward from the top layers to the bottom layers, applying the chain rule to compute the contribution that each parameter had in the loss value

symbolic differentiation (in TensorFlow) → given a chain of operations with a known derivative, can compute a gradient function for the chain (by using the chain rule) that maps network parameter values to gradient values

Chapter summary

- *Learning* means finding a combination of model parameters that minimizes a loss function for a given set of training data samples and their corresponding targets.
- Learning happens by drawing random batches of data samples and their targets, and computing the gradient of the network parameters with respect to the loss on the batch. The network parameters are then moved a bit (the magnitude of the move is defined by the learning rate) in the opposite direction from the gradient.
- The entire learning process is made possible by the fact that neural networks are chains of differentiable tensor operations, and thus it's possible to apply the chain rule of derivation to find the gradient function mapping the current parameters and current batch of data to a gradient value.
- Two key concepts you'll see frequently in future chapters are *loss* and *optimizers*. These are the two things you need to define before you begin feeding data into a network.
- The *loss* is the quantity you'll attempt to minimize during training, so it should represent a measure of success for the task you're trying to solve.
- The *optimizer* specifies the exact way in which the gradient of the loss will be used to update parameters: for instance, it could be the RMSProp optimizer, SGD with momentum, and so on.

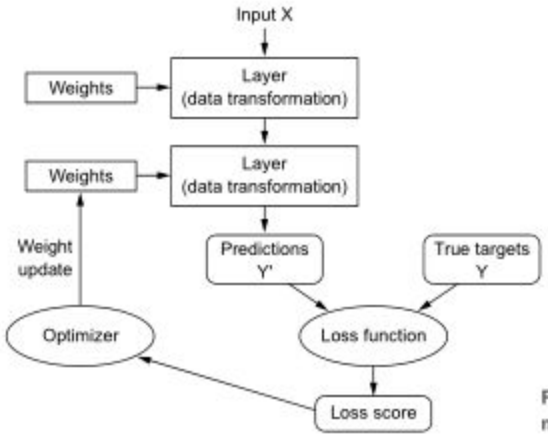


Figure 3.1 Relationship between the network, layers, loss function, and optimizer

layer = a data-processing module, takes tensors as an input and outputs one or more tensors
stateless/have weights as a state (one or several tensors learned with stochastic gradient descent) → together form the network's knowledge

layers combined into a network

build deep learning models in Keras by clipping together compatible layers (can only accept input tensors of a certain shape and will return output tensors of a certain shape) to form useful data-transformation pipelines

directed, acyclic graph of layers
most common = linear stack of layers
but also (two-branch networks, multihead networks, inception blocks)
hypothesis space/space of possibilities =

loss function/objective function = the quantity that will be minimised during training (represents a measure of success for the task at hand)
optimiser ⇒ determines how the network will be updated based on the loss function (implements a specific variant of stochastic gradient descent)
multiple outputs → multiple loss functions
Gd process must be based on a single scalar loss value (i.e. multiple → averaged into a single scalar quantity)

Guidelines:
- binary crossentropy → two class classification problem
- categorical crossentropy → many-class classification problem
- mean sqaured error → regression problem

Keras

deep learning framework for Python

model-level library
relies on specialised, well-optimised tensor libraries (as a backend engine of Keras) → can use 3 different backend engines (TensorFlow, Theano, CNTK)

Figure 3.3 The deep-learning software and hardware stack

Typical workflow:

Define your training data: input tensors and output tensors

Define a network of layers (=model) (mapping inputs to output)

- there are 2 ways to define a model
    - using the Sequential class (linear stacks of layers)

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(32, activation='relu', input_shape=(784,)))
model.add(layers.Dense(10, activation='softmax'))
```

    - using the functional API (directed acyclic graphs of layers)

```
input_tensor = layers.Input(shape=(784,))
x = layers.Dense(32, activation='relu')(input_tensor)
output_tensor = layers.Dense(10, activation='softmax')(x)

model = models.Model(inputs=input_tensor, outputs=output_tensor)
```

Configuration step: choose a loss function(s), an optimiser, metrics you want to monitor during training

```
from keras import optimizers

model.compile(optimizer=optimizers.RMSprop(lr=0.001),
              loss='mse',
              metrics=['accuracy'])
```

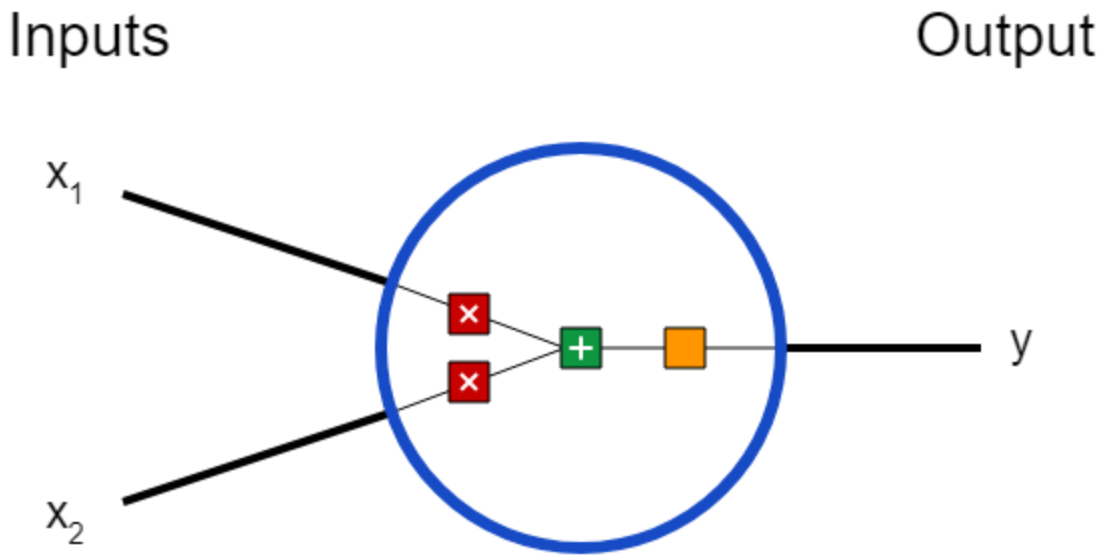
iterate on your training data by calling the fit() method, passing it Numpy arrays of training input and outputs

```
model.fit(input_tensor, target_tensor, batch_size=128, epochs=10)
```

Neurons

Neurons = basic unit

- takes inputs
- does some math with inputs
- produces one output



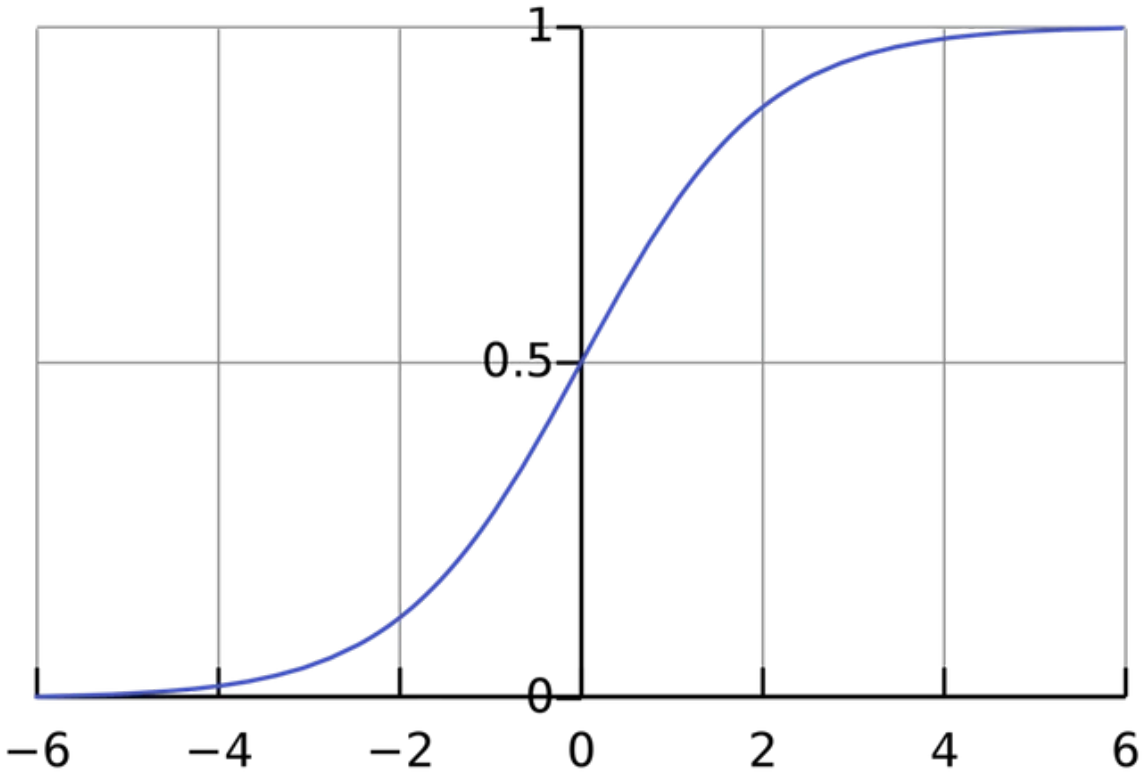
each input is multiplied by a weight
(x_1 * w_1)

all weighted inputs are added together with a bias b
x1 * w1 + x2*w2 + b

the sum is passed through an activation function

y = f(x1*w1 + x2*w2 + b)

turns “unbounded” input into an output with a nice, predictable form (e.g. sigmoid function)



2 input neuron with parameters (w = [0,1] ie w1 = 0, w2 = 1, and b = 4)
give the neuron an input of x = [2,3]

$$
\begin{aligned}
(w \cdot x) + b &= ((w_1 * x_1) + (w_2 * x_2)) + b \\
&= 0 * 2 + 1 * 3 + 4 \\
&= 7
\end{aligned}
$$

$$y = f(w \cdot x + b) = f(7) = \boxed{0.999}$$

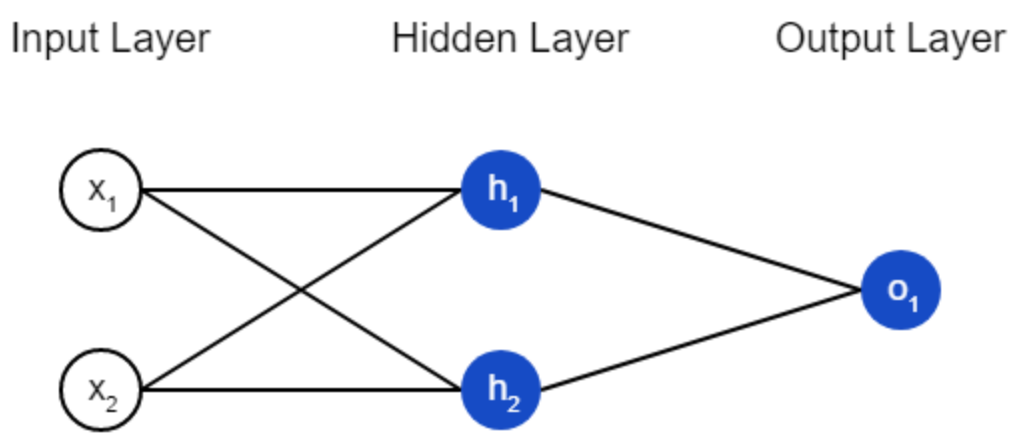
to implement a neuron in numpy

```
import numpy as np

def sigmoid(x):
  # Our activation function: f(x) = 1 / (1 + e^(-x))
  return 1 / (1 + np.exp(-x))
```

```python
class Neuron:
  def __init__(self, weights, bias):
    self.weights = weights
    self.bias = bias

  def feedforward(self, inputs):
    # Weight inputs, add bias, then use the activation function
    total = np.dot(self.weights, inputs) + self.bias
    return sigmoid(total)

weights = np.array([0, 1]) # w1 = 0, w2 = 1
bias = 4                   # b = 4
n = Neuron(weights, bias)

x = np.array([2, 3])       # x1 = 2, x2 = 3
print(n.feedforward(x))    # 0.9990889488055994
```

Combining Neurons into a Neural Network

neural network = a bunch of neurons connected together



- network has 2 inputs
- hidden layer with 2 neurons h_1 and h_2 = any layer between the input/first layer and the last/output layer → there can be multiple hidden layers
- output layer with 1 neuron (o_1)

e.g.

above, w/ all neurons with weights w = [0, 1] and bias b = 0, and the same sigmoid activation function
if you pass in the input x = [2,3]

$$
\begin{aligned}
h_1 = h_2 &= f(w \cdot x + b) \\
&= f((0 * 2) + (1 * 3) + 0) \\
&= f(3) \\
&= 0.9526
\end{aligned}
$$

$$
\begin{aligned}
o_1 &= f(w \cdot [h_1, h_2] + b) \\
&= f((0 * h_1) + (1 * h_2) + 0) \\
&= f(0.9526) \\
&= \boxed{0.7216}
\end{aligned}
$$

output of the neural network for the input x = [2, 3] is 0.7216

can have any number of layers with any number of neurons in those layers
feed the inputs forward through the neurons in the network to get the output at the end

To code the neural network:

import numpy as np

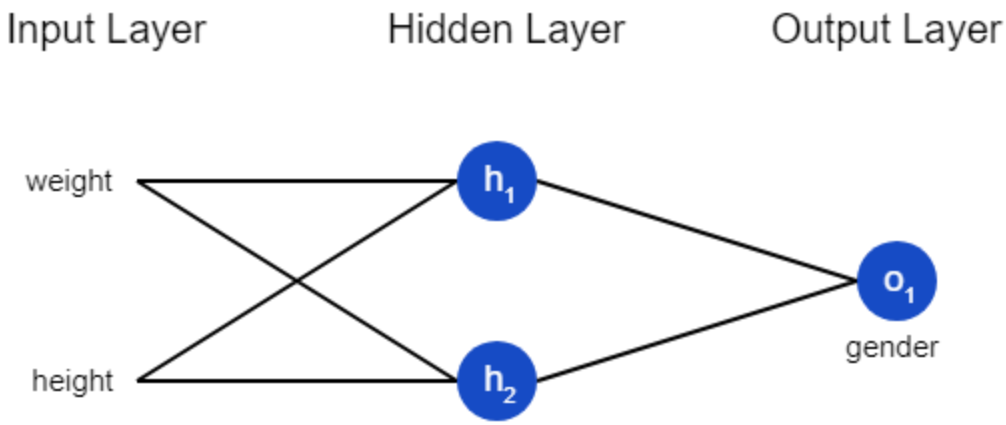
... code from previous section here

```
class OurNeuralNetwork:
  '''
  A neural network with:
    - 2 inputs
    - a hidden layer with 2 neurons (h1, h2)
    - an output layer with 1 neuron (o1)
  Each neuron has the same weights and bias:
    - w = [0, 1]
    - b = 0
```

```
  '''
  def __init__(self):
    weights = np.array([0, 1])
    bias = 0

    # The Neuron class here is from the previous section
    self.h1 = Neuron(weights, bias)
    self.h2 = Neuron(weights, bias)
    self.o1 = Neuron(weights, bias)

  def feedforward(self, x):
    out_h1 = self.h1.feedforward(x)
    out_h2 = self.h2.feedforward(x)

    # The inputs for o1 are the outputs from h1 and h2
    out_o1 = self.o1.feedforward(np.array([out_h1, out_h2]))

    return out_o1

network = OurNeuralNetwork()
x = np.array([2, 3])
print(network.feedforward(x)) # 0.7216325609518421
```

Training a neural network

Name	Weight (lb)	Height (in)	Gender
Alice	133	65	F
Bob	160	72	M
Charlie	152	70	M
Diana	120	60	F



before you train a network

need a way to quantify how “good” it is doing so that it can try to do “better” ⇒ LOSS

e.g. mean squared error (MSE) loss

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_{true} - y_{pred})^2$$

= the average over all squared errors (= meaned)

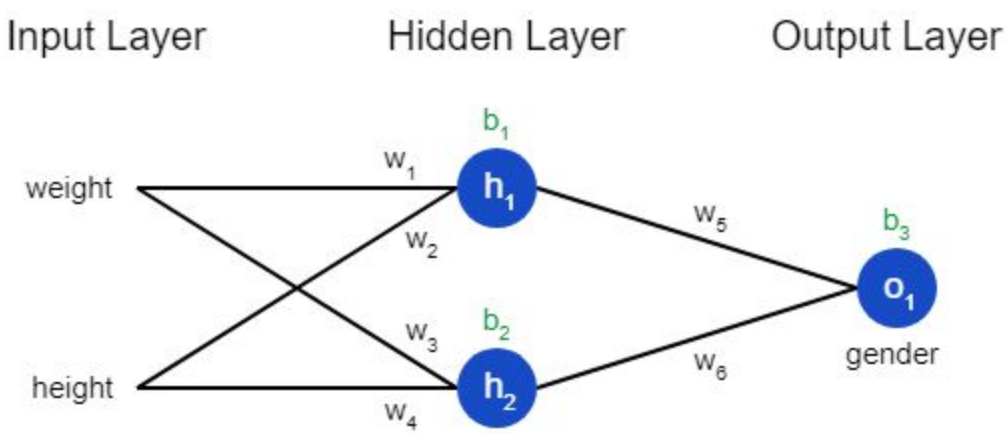
better predictions = lower loss

```
import numpy as np

def mse_loss(y_true, y_pred):
  # y_true and y_pred are numpy arrays of the same length.
  return ((y_true - y_pred) ** 2).mean()

y_true = np.array([1, 0, 0, 1])
y_pred = np.array([0, 0, 0, 0])

print(mse_loss(y_true, y_pred)) # 0.5
```

can also think about loss as a function of weights and biases (i.e. a multivariable function)

$$L(w_1, w_2, w_3, w_4, w_5, w_6, b_1, b_2, b_3)$$

label each weight and bias in the network



partial derivative $\frac{\partial L}{\partial w_1}$

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y_{pred}} * \frac{\partial y_{pred}}{\partial w_1}$$

$$\frac{\partial L}{\partial y_{pred}} = \frac{\partial (1 - y_{pred})^2}{\partial y_{pred}} = \boxed{-2(1 - y_{pred})}$$

$$y_{pred} = o_1 = f(w_5 h_1 + w_6 h_2 + b_3)$$

$$\frac{\partial y_{pred}}{\partial w_1} = \frac{\partial y_{pred}}{\partial h_1} * \frac{\partial h_1}{\partial w_1}$$

$$\frac{\partial y_{pred}}{\partial h_1} = \boxed{w_5 * f'(w_5 h_1 + w_6 h_2 + b_3)}$$

$$h_1 = f(w_1 x_1 + w_2 x_2 + b_1)$$

$$\frac{\partial h_1}{\partial w_1} = \boxed{x_1 * f'(w_1 x_1 + w_2 x_2 + b_1)}$$

$$f(x) = \frac{1}{1 + e^{-x}}$$

$$f'(x) = \frac{e^{-x}}{(1 + e^{-x})^2} = f(x) * (1 - f(x))$$

$$\boxed{\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y_{pred}} * \frac{\partial y_{pred}}{\partial h_1} * \frac{\partial h_1}{\partial w_1}}$$

= calculating partial derivatives by working backwards = backpropogation

Name	Weight (minus 135)	Height (minus 66)	Gender
Alice	-2	-1	1

Let's initialize all the weights to 1 and all the biases to 0. If we do a feedforward pass through the network, we get:

$$\begin{aligned} h_1 &= f(w_1 x_1 + w_2 x_2 + b_1) \\ &= f(-2 + -1 + 0) \\ &= 0.0474 \end{aligned}$$

$$h_2 = f(w_3 x_1 + w_4 x_2 + b_2) = 0.0474$$

$$\begin{aligned} o_1 &= f(w_5 h_1 + w_6 h_2 + b_3) \\ &= f(0.0474 + 0.0474 + 0) \\ &= 0.524 \end{aligned}$$

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y_{pred}} * \frac{\partial y_{pred}}{\partial h_1} * \frac{\partial h_1}{\partial w_1}$$

$$\begin{aligned}\frac{\partial L}{\partial y_{pred}} &= -2(1 - y_{pred}) \\ &= -2(1 - 0.524) \\ &= -0.952\end{aligned}$$

$$\begin{aligned}\frac{\partial y_{pred}}{\partial h_1} &= w_5 * f'(w_5 h_1 + w_6 h_2 + b_3) \\ &= 1 * f'(0.0474 + 0.0474 + 0) \\ &= f(0.0948) * (1 - f(0.0948)) \\ &= 0.249\end{aligned}$$

$$\begin{aligned}\frac{\partial h_1}{\partial w_1} &= x_1 * f'(w_1 x_1 + w_2 x_2 + b_1) \\ &= -2 * f'(-2 + -1 + 0) \\ &= -2 * f(-3) * (1 - f(-3)) \\ &= -0.0904\end{aligned}$$

$$\begin{aligned}\frac{\partial L}{\partial w_1} &= -0.952 * 0.249 * -0.0904 \\ &= \boxed{0.0214}\end{aligned}$$

i.e. if we were to increase w1, L would increase a tiny bit as a result

Unsupervised Learning (Chapter 7)

<u>NB: must go through and learn this stuff! especially Sebastian's lecture stuff: PCA, autoencoders, clustering !!!! learn it well</u>

supervised learning:
- set of features X_1, X_2 … X_p for each object
- response or outcome variable y
- goal is to predict Y using X_1, X_2 … X_p

Unsupervised learning:
- only observe the features X_1, X_2 … X_p
- do not have the associated response variable y

No known output
just shown the input data, extract knowledge from this data

visualisation, compact/more informative representation, subgroups

1. transformations of the dataset (unsupervised transformations)
    a. algorithms that create a new representation of the data (often in less dimensions)→ that might be easier to understand
    b. dimensionality reduction → high dimensional representation w/ many features → summarises the essential characteristics with fewer features
    c. finding the components that "make up" the data → = topic extraction groups of similar items nct into distion collections of documents
2. clustering algorithms
    a. partition the data
    b. discover unknown subgroups within the data

exploratory
preprocessing step for supervised algorithms

challenges:
hard to evaluate whether the model "did well"/learned anything useful

trickier to evaluate the outcome of your algorithm?/the quality of your unsupervised learning algorithms. was it successful or not?

advantages:

- much easier to obtain unlabeled data from a lab instrument or a computer than labelled data (which requires a human person!)
    - e.g. tumours vs healthy tissue

e.g.

- group breast cancer patients according to their genes
- group diseases based on pathophysiology, may help you to develop effective treatment
- identify subtypes of asthma→ woodruff
- growth abnormalities of children in rural Bangladesh → socioeconomic variables

Preprocessing/Scaling

I guess this is a form of unsupervised ML?

remember: SVMs, neural networks = very sensitive to scaling, must adjust the features so that data representation is more suitable for these algorithms (per-feature rescaling, shift of the data)

StandardScaler

- each feature, mean = 0, variance = 1, all features in the same magnitude
- does not ensure minimum or maximum values for the features

RobustScaler

- uses the median and quartiles (instead of mean and variance)
- ignores data points that are very different from the rest (i.e. measurement errors/outliers)

MinMaxScaler

- shifts the data o all features are exactly between 0 and 1
- Notes in SVM chapter

Normalizer

- scales each data point such that the feature vector has a Euclidean length of 1
- ie projects onto a circle with radius 1
- each data point is scaled by the inverse of its length

Must apply the same transformation to the training set and the test set

shortcut → to fit a model then transform it = scaler.fit_transform(X)

In[9]:

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
# calling fit and transform in sequence (using method chaining)
X_scaled = scaler.fit(X).transform(X)
# same result, but more efficient computation
X_scaled_d = scaler.fit_transform(X)
```

Dimensionality Reduction/Feature Extraction/Manifold Learning

when you have a lot of variables to consider
How do I take all of the variables I've collected and focus on only a few of them? i.e. = reduce the dimension of your feature space
by reducing the dimension of your feature space, you have fewer relationships between variables, less likely to overfit your model

feature elimination = eliminate features, drop all the features you don't think are important → but you gain no information from the features/variables you've dropped

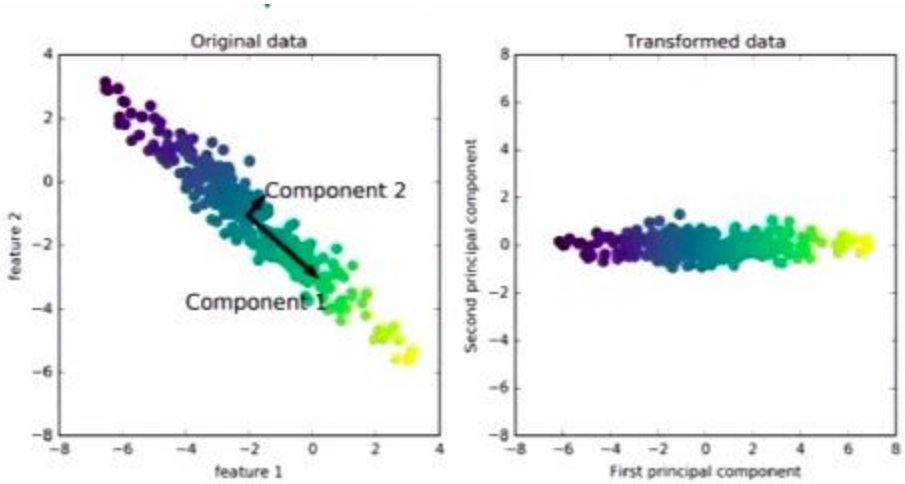
<u>feature extraction</u> = converting ten independent variables into ten new variables where each new variable is a combination of each of the tend "old" independent variables, but we create these new variables in a specific way, order them by how well they predict the dependent variable

- because the new features/variables are ordered by importance/how well they predict the dependent variable →
- you can drop the least important ones
- BUT
- the new features are a combination of the old features i.e. you're still keeping the most valuable parts of your old variables

PRINCIPAL COMPONENT ANALYSIS
= technique for feature extraction (produce a low dimensional representation of the dataset)

use cases:

- dimensionality reduction before applying supervised learning (reducing noise)
- data visualization
- data compression



e.g. feature 1 on the x-axis, feature 2 on the y-axis
you look at the direction or the vector in which the data varies the most

finds a sequence of linear combinations of the features that has: MAXIMAL VARIANCE, mutually uncorrelated (so each principal component tells you something new about the data)

each principal component is computed as a linear combination of your input features (first one on the x-axis, second principal component on the y-axis)

- Preprocessing:
  - Assume an $n \times p$ data set X with n data points and p features
  - Scale each feature $X_1, X_2, \ldots, X_p$ across data points to have mean zero and standard deviation one

scale your features to have 0 mean and SD of 1.
first principal component is just a linear combination of your features
in order to find the weights for this linear combination → trying to maximise the variance

- The first principal component is the linear combination of the features

$$z_{i1} = \phi_{11} x_{i1} + \phi_{21} x_{i2} + \ldots + \phi_{p1} x_{ip}$$

that has the largest sample variance, subject to the constraint $\sum_{j=1}^{p} \phi_{j1}^2 = 1$

why are we trying to maximise the variance? because if the variance is small, then all observations are similar and you don't get much information out of that

if you can choose your weighting of the features however you want, arbitrarily, you can make the variance as large as you want → hence why we have a constraint

- The constraint is necessary so that the variance doesn't get arbitrarily large

Since you scaled/normalized all your features so that they have a mean of 0, all your prinicipal components will also have a mean of 0.

- Since each of the x_{ij} has mean zero, so does z_{i1} (for any value of ϕ_{j1})

you can write the "sample variance" of the computed principal component z (called the principal component score) like this (which is just the usual way to calculate variance)

- The sample variance of the z_{i1} can be written as $\frac{1}{n}\sum_{i=1}^{n} z_{i1}^2$

therefore you're finding the weights, phi, that maximise the sample variance subject to the constraint of the weights having a norm of 1.

- Thus, we want to solve:

$$\underset{\phi_{11},\ldots,\phi_{p1}}{\text{maximize}} \frac{1}{n}\sum_{i=1}^{n}\left(\sum_{j=1}^{p}\phi_{j1}x_{ij}\right)^2 \text{ subject to } \sum_{j=1}^{p}\phi_{j1}^2 = 1$$

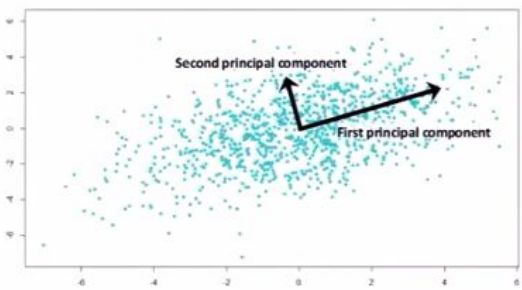
Z_1 = the first principal component, up to Z_{n1}

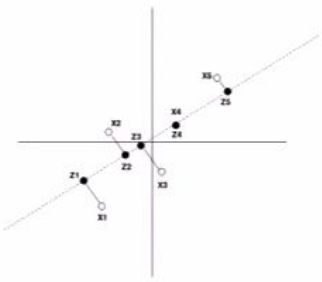
phis = the loadings of the first principal component, you can put them together into a "loading vector"

- Terminology:
  - Z_1 is the first principal component with scores $z_{11}, \ldots, z_{n1}$
  - $\phi_{11}, \phi_{21}, \ldots, \phi_{p1}$: loadings of the first principal component
  - $\phi_1 = (\phi_{11}\ \phi_{21}\ \ldots\ \phi_{p1})^T$: principal component loading vector

the first principal component/loading vector (arrow pointing to the riht) = the direction your data varies the most. Once you've found this vector, you project your data onto this vector, and find the principal component scores.

PCA: Interpretation





- The loading vector ϕ_1 with elements $\phi_{11}, \phi_{21}, \ldots, \phi_{p1}$ defines a direction in feature space along which the data vary the most
- If we project the n data points $x_1, \ldots, x_n$ onto this direction, the projected values are the principal component scores $z_{11}, \ldots, z_{n1}$

Then you go on to look for the second principal component, again a linear combination of features that has maximal variance along all linear combinations that are uncorrelated (= the two loading vectors are orthogonal with each other) with the first principal component you've found.

- The second principal component is the linear combination of $X_1, X_2, \ldots, X_p$ that has maximal variance among all linear combinations that are uncorrelated with Z_1

- The second principal component scores $z_{11}, \ldots, z_{n1}$ take the form

  $$z_{i2} = \phi_{12}x_{i1} + \phi_{22}x_{i2} + \ldots + \phi_{p2}x_{ip}$$

  where ϕ_2 is the second principal component loading vector with elements $\phi_{12}, \phi_{22}, \ldots, \phi_{p2}$

- Z_2 uncorrelated with Z_1 is equivalent to ϕ_2 orthogonal to ϕ_1 (and so on)

simplest way to compute PCA is by using singular-value decomposition of the matrix of X (the input features)
usually will be real valued, so all these features will be real valued as well (inverse of U will be equal to the transports of u)

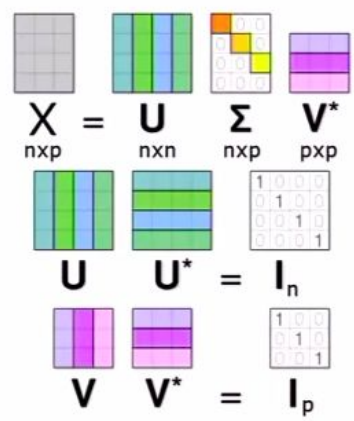
V has the principal vector loading vectors as columns
when you have more observations than features than you can have at most p principal components
n - 1 principal components → if you have more features than observations

PCA: Computation

- Principal components can be found by singular-value decomposition of the matrix X

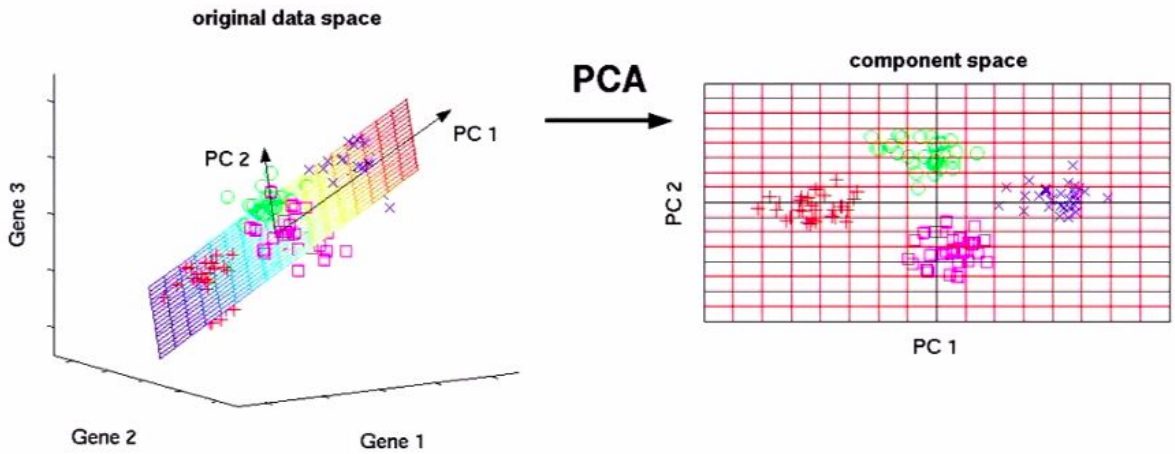
  $$X = U\Sigma V^*$$

  where
  - U is an $n \times n$ real or complex unitary matrix
  - Σ is a $n \times p$ rectangular diagonal matrix with non-negative real numbers on the diagonal (the singular values of X)
  - V is an $p \times p$ real or complex unitary matrix



- The principal component directions $\phi_1, \phi_2, \ldots$ correspond to the columns of V (the right-singular vectors)
- The variances of the PCA scores are 1/n times the squares of the singular values
- There are at most min(n-1, p) principal components

PCA: Example



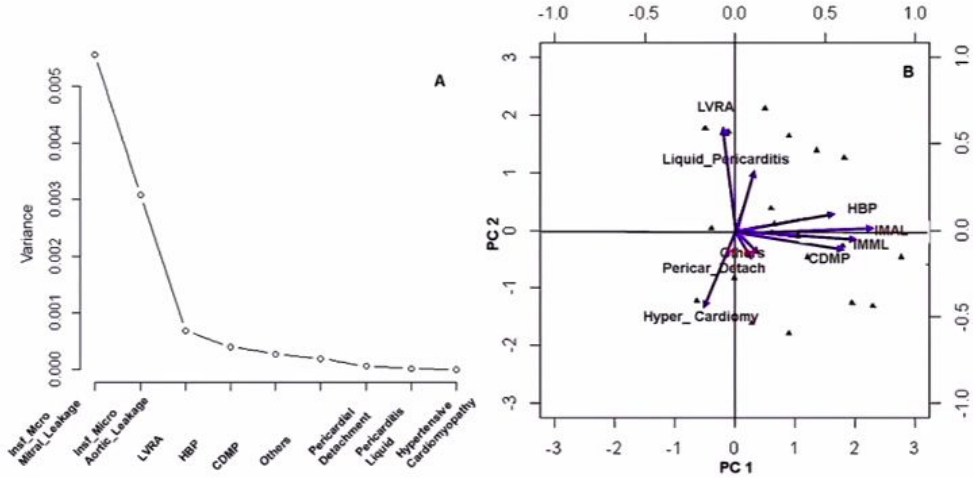
- Three-dimensional gene expression samples are projected onto a two dimensional component space that maintains the largest variance in the data [Matthias Scholz, Ph.D. thesis]

Biplots → can be interesting when you perform PCA, gives the information both on the position of the transformed data

+ vectors tell you the weighting of each feature for each principal component

PCA: Biplot

- Biplot: displays both the principal component scores (as data points) and the principal component loadings (as arrows)



Performance Assessment of the Recurrence of Cardiovascular Pathologies Symptoms in a Hypertensive Population [N.D. Dago et al., 2017]

Proportion of variance explained:
once you have transformed your data, how informative is your data, and how many principal components do you need to still have a good representation of your data?

ie which proportion of variance is explained by each principal component.

- Which proportion of variance is explained by each principal component?

In order to compute this, you start by looking at the total variance in your dataset, just the sum of the variance of each feature

- Total variance in a data set (assuming that the variables have been centered to have mean zero):

$$\sum_{j=1}^{p} \mathrm{Var}(X_j) = \sum_{j=1}^{p} \frac{1}{n} \sum_{i=1}^{n} x_{ij}^2$$

then you look at the variance explained by each principal component (so for a given set of scores z you compute the sample variance)

- Variance explained by the *m*-th principal component:

$$\mathrm{Var}(Z_m) = \frac{1}{n}\sum_{i=1}^{n} z_{im}^2$$

it can be shown that when you do PCA with the largest possible amount of principal components, then you are not losing any variance (the total variance of your initial data is equal to the total variance of your transformed data)

- It can be shown that $\sum_{j=1}^{p} \mathrm{Var}(X_j) = \sum_{m=1}^{M} \mathrm{Var}(Z_m)$ with M = min(n-1, p)

leads to a relatively simple way to compute proportion of variance explained by the mth principal component (basically just the variance of your scores divided by the total variance)

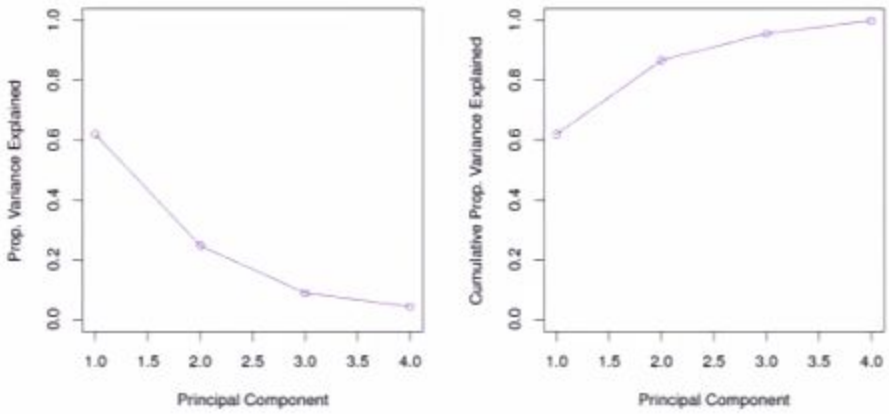
- Thus, the proportion of variance explained by the *m*-th principal component is

$$\frac{\sum_{i=1}^{n} z_{im}^2}{\sum_{j=1}^{p}\sum_{i=1}^{n} x_{ij}^2}$$

- This is a positive quantity between 0 and 1
- The proportions of variance explained sum to one

you can plot this: first one explains about 65% of your variance, the second about 25%.... allow you to select the proper amount of principal components sufficient to summarize your data (e.g. choose enouch principal components to represent 95% of the variance in your data)

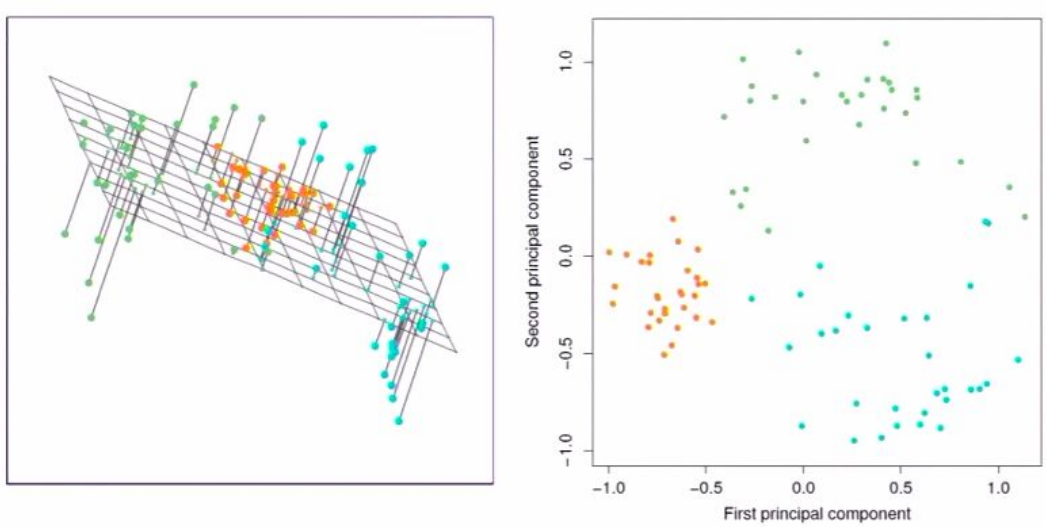
Plot can be used to select how many principal components are sufficient to summarize the data



Another interpretation of PCA:

- the principal components point in the direction of maximum variance of your data
- the first principal component defines the line that is closest to your data point (in terms of the average Euclidean distance)
- when you compute the second principal component the first and the second span the plane that is closest to your data points in higher dimensions

PCA: Another Interpretation



- The first principal components defines the line that is closest (in terms of average Euclidean distance) to the n data points
- The first two principal components span the plane that is closest to the n data points, and so on

computed by single-value decomposition → easiest way to do it

PCA: Conclusions

- Principal component loading vectors:
  - Point in the direction of maximal variance
  - Are orthogonal to one another
  - Span the "plane" that lies closest to the data

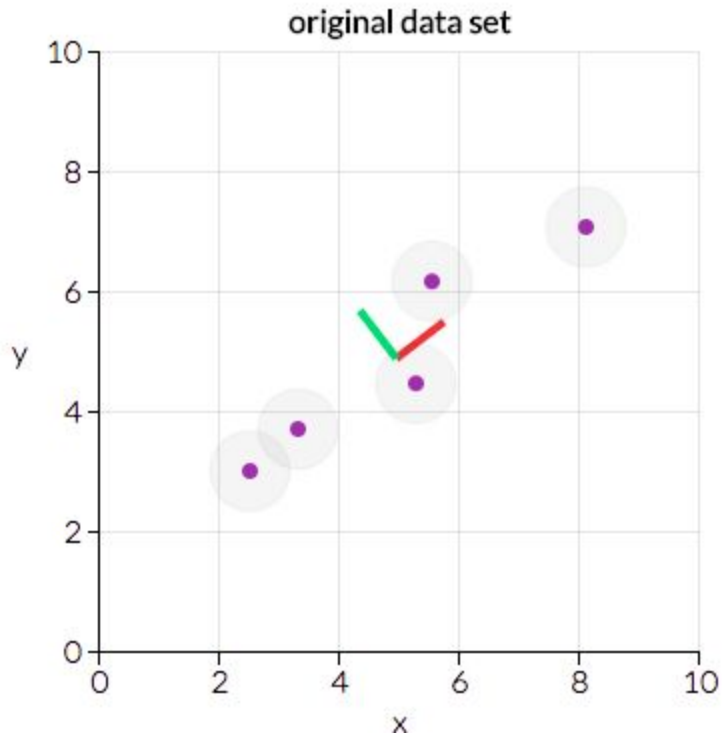
- Computed by SVD; Incremental PCA reduces memory requirements for large datasets

- Scaling of variables matters (especially if variables have different units)

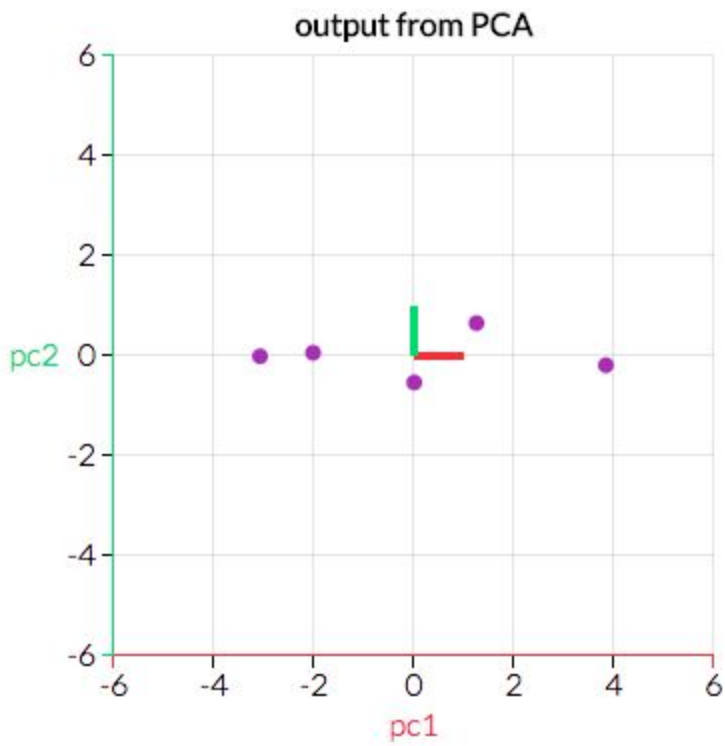
- Use proportion of variance explained plots to determine a suitable number of principal components

combines input variables, then you can drop the least important features while still retaining the most valuable parts of all the features
the new features are all still independent of one another (linear) → important
linear model: requires all features (independent variables) to be independent of one another

calculates a matrix that summarises how all features relate to each other
break this matrix down into separate components, direction and magnitude
directions of the data + magnitude (how important each direction is)



then transform the data to align with these important directions (combinations of the original features)



Our original data transformed by PCA. (Source.)

example is 2d (only two directions)
but you can have more dimensions

by identifying which directions are most important, compress your data into a smaller space by dropping the directions that are “least important” → reducing the dimensionality of the feature space

matrix of inputs X, subtract the mean for each feature (every feature now has a mean 0)
standardise?: are features with higher variance more important than features with lower variance, or not?
if not: divide each feature by that feature column’s SD (now each feature has a SD of 1)

take the matrix, transpose it, multiply the transposed matrix by the original matrix (covariance matrix of)

Calculate the eigenvectors and their corresponding eigenvalues of Z^TZ.

take the eigenvalues and sort them from largest to smallest, sort them accordingly

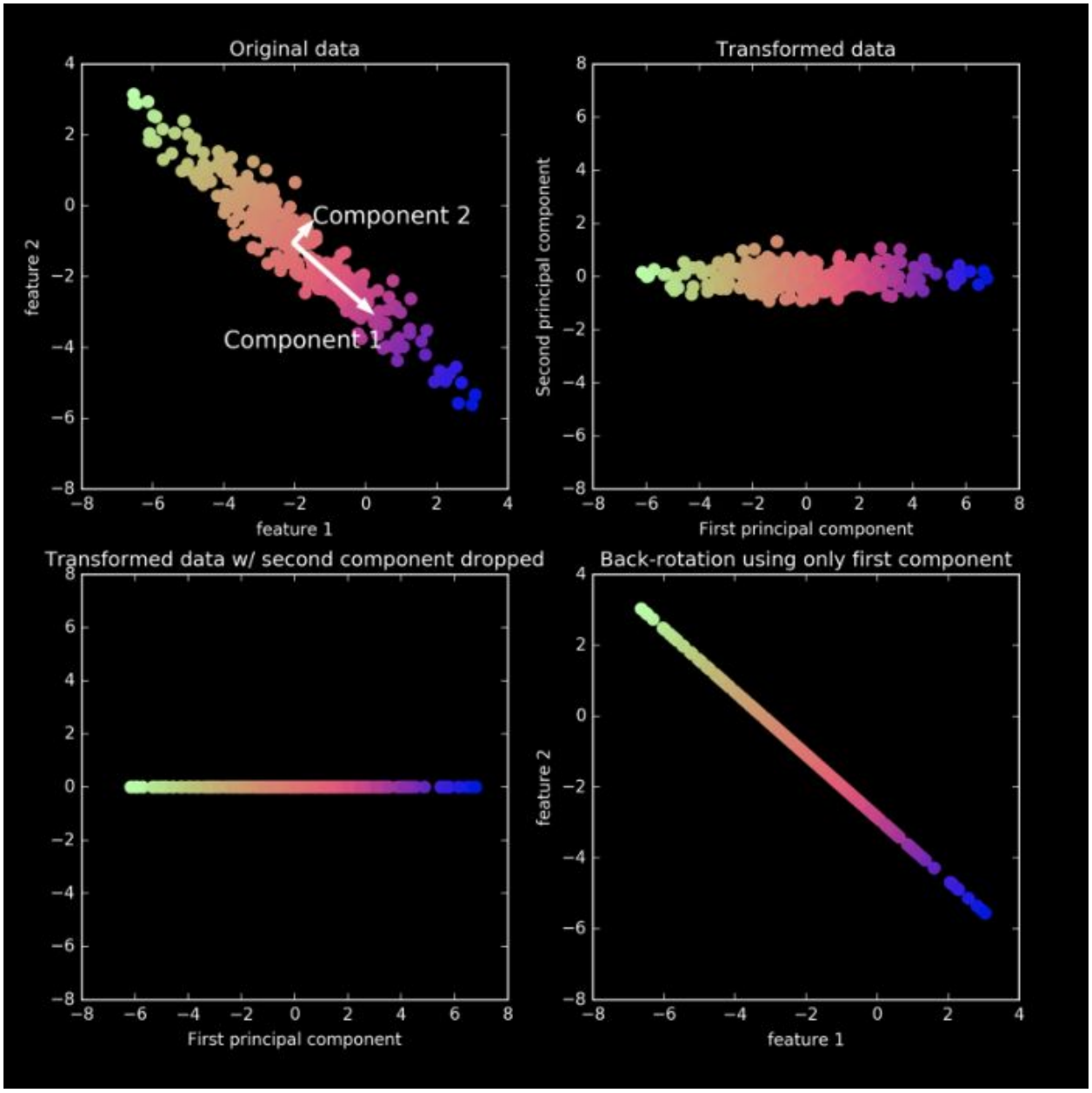
Principal component analysis

- rotates the dataset in a way such that the rotated features are statistically uncorrelated
- then selects only a subset of the new features, (import ones for explaining the data)
- unsupervised → does not use class information, simply looks at the correlations in the data

- *orthogonal linear transformation, transforms the data to a new coordinate system such that the greatest variance by some scalar projection of the data comes to lie on the first coordinate (the first principal component), the second greatest variance on the second coordinate*
- first subtract the mean of each variable from the dataset (to center the data around the origin)
- compute the covariance matrix of the data, calculate the eigenvalues + corresponding eigenvectors of this covariance matrix….
- fit an ellipsoid
- converts a set of observations of possibly correlated variables into a set of linearly uncorrelated variables → called principal components
- so that, the first principal component has the largest possible variance (accounts for as much of the variability in the data as possible), each succeeding component in turn has the highest variance possible (while staying orthogonal i.e. a right angle to the preceding components)
- ⇒ revealing the internal structure of the data in a way that best explains its variance
- by using only the first few principal components → multivariate dataset in a high-dimensional space → reduced to a lower dimensional picture

finds the direction of maximum variance = direction/vector in the data that contains most information/features are most correlated with each other
then the algorithm finds the direction that contains the most information that is orthogonal/at a right angle to the first direction

directions = principal components



rotated so that the first principal component aligns with the x-axis, the second principal component aligns with the y-axis
before the rotation, the mean was subtracted from the data (transformed data centered around 0)

For the breast cancer dataset:

- to create a histogram for each of the features, counting how often a data point appears with a feature in a certain range (bin)

```
fig, axes = plt.subplots(15, 2, figsize=(10, 20))
malignant = cancer.data[cancer.target == 0]
benign = cancer.data[cancer.target == 1]
ax = axes.ravel()
for i in range(30):
    _, bins = np.histogram(cancer.data[:, i], bins=50)
    ax[i].hist(malignant[:, i], bins=bins, color=mglearn.cm3(0), alpha=.5)
    ax[i].hist(benign[:, i], bins=bins, color=mglearn.cm3(2), alpha=.5)
    ax[i].set_title(cancer.feature_names[i])
    ax[i].set_yticks(())
ax[0].set_xlabel("Feature magnitude")
ax[0].set_ylabel("Frequency")
ax[0].legend(["malignant", "benign"], loc="best")
fig.tight_layout()
```

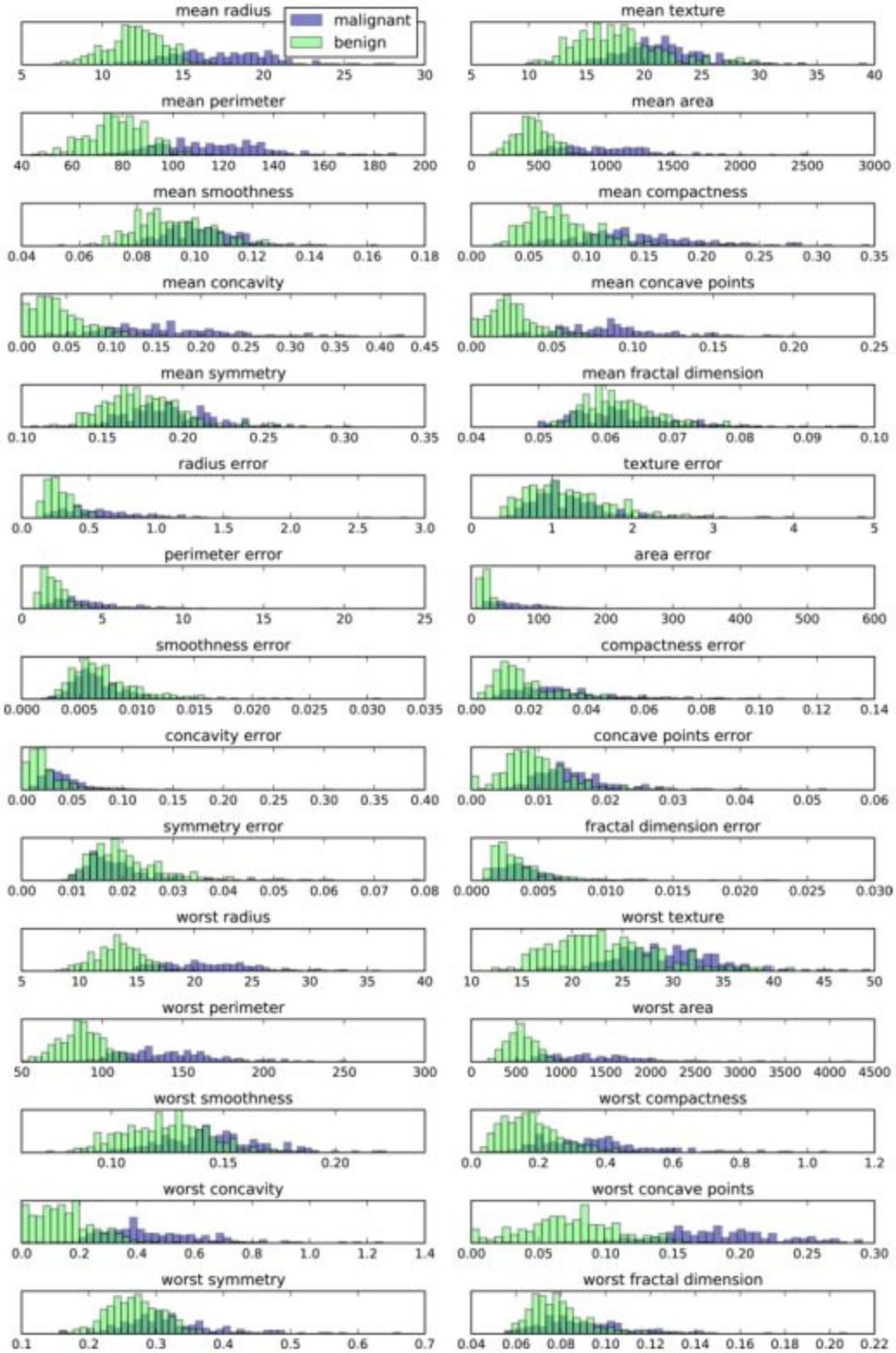
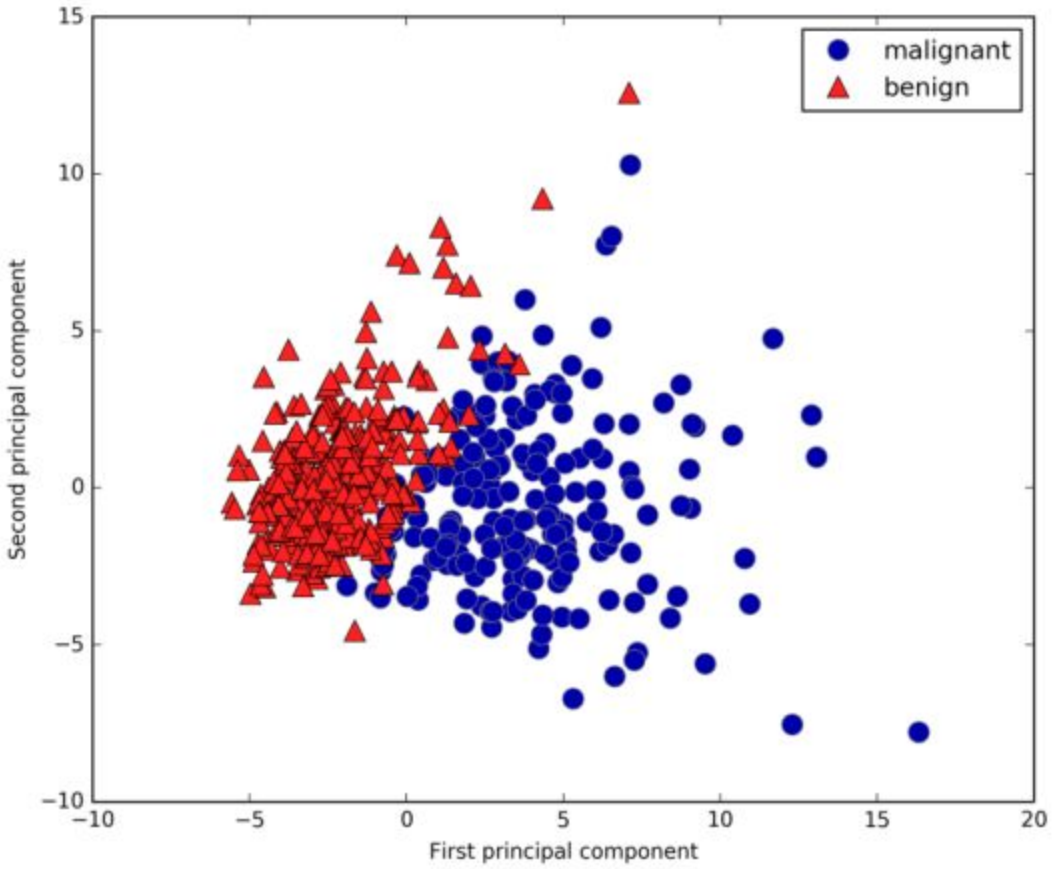


Figure 3-4. Per-class feature histograms on the Breast Cancer dataset

```
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()
scaler = StandardScaler()
scaler.fit(cancer.data)
X_scaled = scaler.transform(cancer.data)
```

[16]:

```
from sklearn.decomposition import PCA
# keep the first two principal components of the data
pca = PCA(n_components=2)
# fit PCA model to breast cancer data
pca.fit(X_scaled)
# transform data onto the first two principal components
X_pca = pca.transform(X_scaled)
print("Original shape: {}".format(str(X_scaled.shape)))
print("Reduced shape: {}".format(str(X_pca.shape)))


# plot first vs. second principal component, colored by class
plt.figure(figsize=(8, 8))
mglearn.discrete_scatter(X_pca[:, 0], X_pca[:, 1], cancer.target)
plt.legend(cancer.target_names, loc="best")
plt.gca().set_aspect("equal")
plt.xlabel("First principal component")
plt.ylabel("Second principal component")
```



Eigenfaces for feature extraction:

FACE RECOGNITION:
does a previously unseen face belong to a known person from the database?
rather than building a classifier where every person is a separate class → difficult
want to be able to add new people, without needing to retrain the model

find a representation of the data better suited to analysis
e.g. images, made up of pixels, stored as RGB intensities

```
from sklearn.datasets import fetch_lfw_people
people = fetch_lfw_people(min_faces_per_person=20, resize=0.7)
image_shape = people.images[0].shape
fix, axes = plt.subplots(2, 5, figsize=(15, 8),
```

```
                                subplot_kw={'xticks': (), 'yticks': ()})
for target, image, ax in zip(people.target, people.images, axes.ravel()):
    ax.imshow(image)
    ax.set_title(people.target_names[target])
```

In[22]:

```
print("people.images.shape: {}".format(people.images.shape))
print("Number of classes: {}".format(len(people.target_names)))
Out[22]:
people.images.shape: (3023, 87, 65)
Number of classes: 62
```

In[23]:

```
# count how often each target appears
counts = np.bincount(people.target)
# print counts next to target names
for i, (count, name) in enumerate(zip(counts, people.target_names)):
    print("{0:25} {1:3}".format(name, count), end='   ')
    if (i + 1) % 3 == 0:
```

```
        print()
```

Out[23]:

```
Alejandro Toledo           39   Alvaro Uribe               35
Amelie Mauresmo            21   Andre Agassi               36
Angelina Jolie             20   Arnold Schwarzenegger      42
Atal Bihari Vajpayee       24   Bill Clinton               29
Carlos Menem               21   Colin Powell              236
David Beckham              31   Donald Rumsfeld           121
George W Bush             530   George Robertson           22
Gerhard Schroeder         109   Gloria Macapagal Arroyo    44
Gray Davis                 26   Guillermo Coria            30
Hamid Karzai               22   Hans Blix                  39
Hugo Chavez                71   Igor Ivanov                20
[...]                           [...]
Laura Bush                 41   Lindsay Davenport          22
Lleyton Hewitt             41   Luiz Inacio Lula da Silva  48
Mahmoud Abbas              29   Megawati Sukarnoputri      33
Michael Bloomberg          20   Naomi Watts                22
Nestor Kirchner            37   Paul Bremer                20
Pete Sampras               22   Recep Tayyip Erdogan       30
Ricardo Lagos              27   Roh Moo-hyun               32
Rudolph Giuliani           26   Saddam Hussein             23
Serena Williams            52   Silvio Berlusconi          33
Tiger Woods                23   Tom Daschle                25
Tom Ridge                  33   Tony Blair                144
Vicente Fox                32   Vladimir Putin             49
Winona Ryder               24
```

In[24]:

```
mask = np.zeros(people.target.shape, dtype=np.bool)
for target in np.unique(people.target):
    mask[np.where(people.target == target)[0][:50]] = 1
X_people = people.data[mask]
y_people = people.target[mask]
# scale the grayscale values to be between 0 and 1
# instead of 0 and 255 for better numeric stability
X_people = X_people / 255.
A common task in face recognition is to ask if a previously unseen face
```

one nearest neighbour (kNN model) →

In[25]:

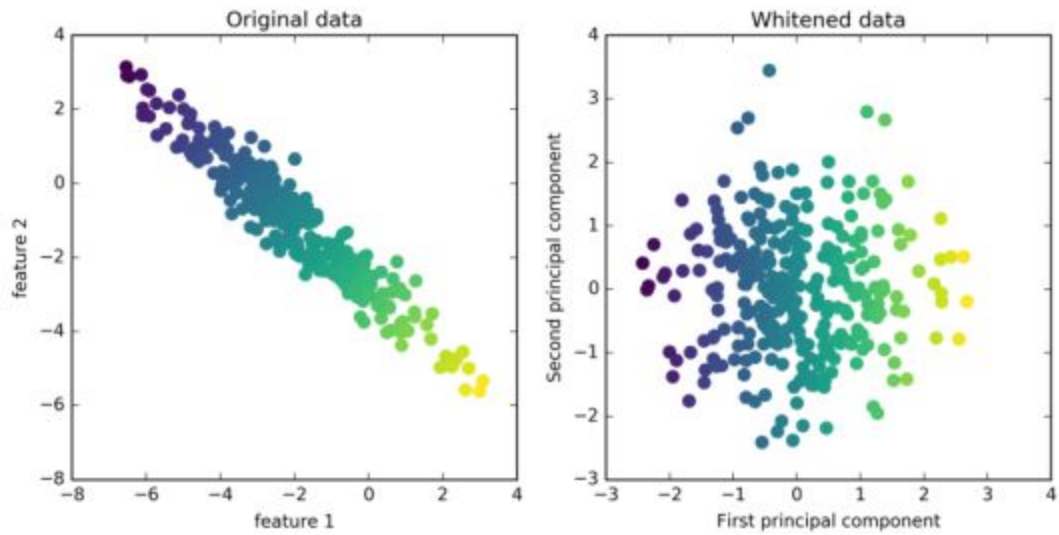
```
from sklearn.neighbors import KNeighborsClassifier
# split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(
    X_people, y_people, stratify=y_people, random_state=0)
# build a KNeighborsClassifier using one neighbor
knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(X_train, y_train)
print("Test set score of 1-nn: {:.2f}".format(knn.score(X_test, y_test)))
```

Out[25]:
Test set score of 1-nn: 0.27

only gives a 27% accuracy

PCA

- enable the whitening option, rescales the principal components to have the same scale




```
In[27]:
pca = PCA(n_components=100, whiten=True, random_state=0).fit(X_train)
X_train_pca = pca.transform(X_train)
X_test_pca = pca.transform(X_test)
print("X_train_pca.shape: {}".format(X_train_pca.shape))
```

Out[27]:
X_train_pca.shape: (1537, 100)

The new data has 100 features, the first 100 principal components.

```
knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(X_train_pca, y_train)
print("Test set accuracy: {:.2f}".format(knn.score(X_test_pca, y_test)))
```

Test set accuracy: 36%

```
fix, axes = plt.subplots(3, 5, figsize=(15, 12),
```

```
                          subplot_kw={'xticks': (), 'yticks': ()})
for i, (component, ax) in enumerate(zip(pca.components_, axes.ravel())):
    ax.imshow(component.reshape(image_shape),
              cmap='viridis')
    ax.set_title("{}. component".format((i + 1)))
hile we certainly cannot understand all aspects of these components, we can
guess
```

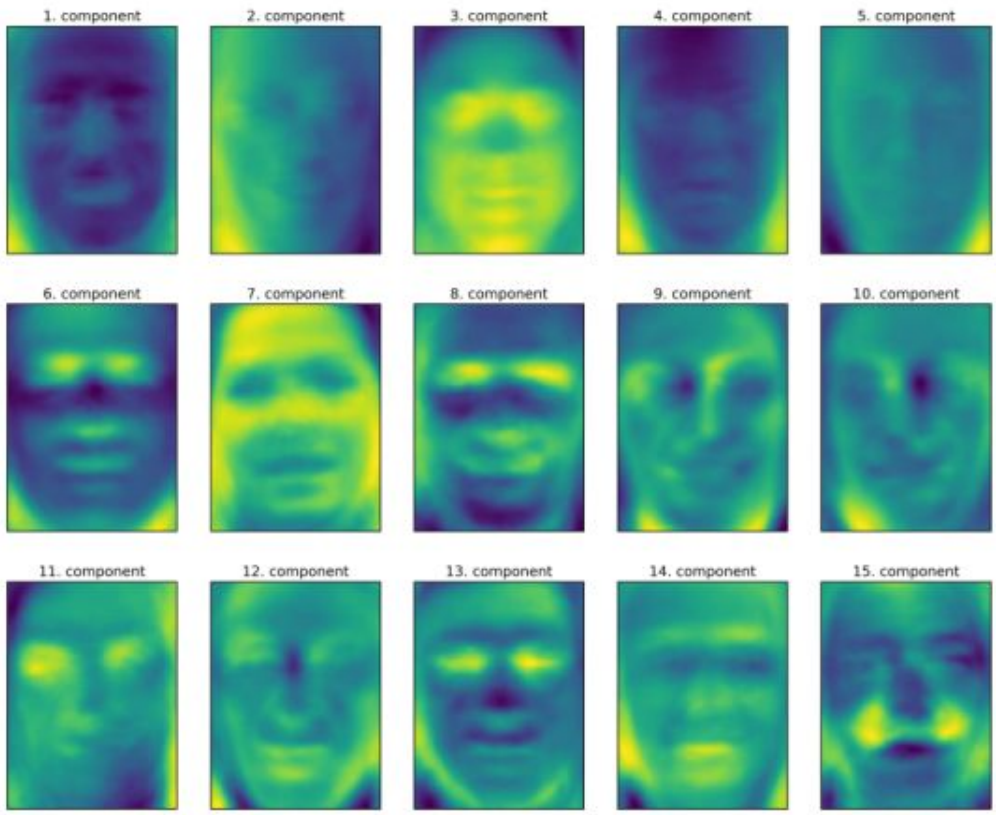


Figure 3-9. Component vectors of the first 15 principal components of the faces dataset

PCA transformation, rotating the data, then dropping the components with low variance

Autoencoders

very modern technique for dimensionality reduction

based on neural network

close link between autoencoders and principal component analysis

- Artificial neural networks capable of learning efficient representations of the input data, called codings, without any supervision
- Codings typically have a much lower dimensionality than the input data
- Uses:
  - Dimensionality reduction
  - Feature detection (unsupervised pretraining of deep neural networks)
  - Generative model (randomly generating new data that looks very similar to the training data)

typically the codings have a much lower dimensionality than the input data
feature detection → preprocessing before doing supervised learning
generative model → when you're trying to generate random samples that are not real but very similar to it

learn to copy their inputs to the output

- Autoencoders learn to copy their inputs to their outputs
- There are different constraints so that an autoencoder doesn't just learn the identity function, for example:
  - Limit the size of the internal representation
  - Add noise to the inputs and train the network to recover the original inputs

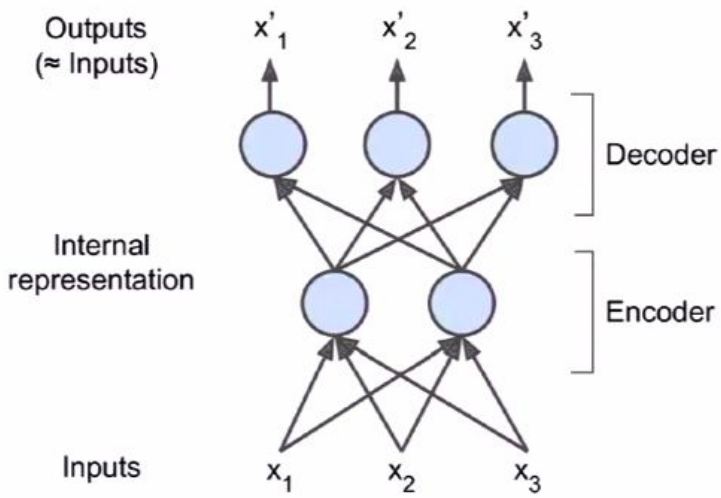
interesting → that they get different constraints
ie. limit the size, add noise/set some of your input data to 0, train the network to recover the original input

network architecture

always composed of two parts

encoder and a decoder
encoder/recognition network → converts the input/encodes it to an internal two-dimensional representation of the data
decoder/generative network → converts the two-dimensional representation to a three-dimensional output

- An autoencoder is always composed of two parts:
  - an encoder (or recognition network) that converts the inputs to an internal representation
  - a decoder (or generative network) that converts the internal representation to the outputs



same architecture as a neural network
but the number of neurons in the input and output layers must be the same

An autoencoder typically has the same architecture as a Multi-Layer Perceptron

But: the number of neurons in the output layer must be equal to the number of inputs

If the internal representation has a lower dimensionality than the input data, the autoencoder is undercomplete → it cannot trivially copy its inputs to the codings, yet it must find a way to output a copy of its inputs

The outputs are often called reconstructions

The cost function contains a reconstruction loss that penalizes the model when the reconstructions are different from the inputs

the cost function that you use when you train the function = reconstruction loss that penalises when the reconstructions are different from the inputs
the output must be as similar as possible to the input

link between autoencoders and PCA

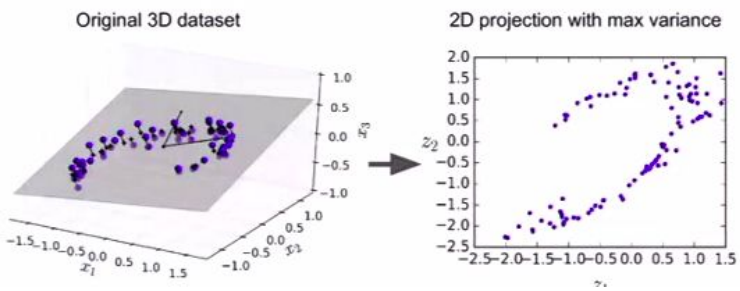
Autoencoder vs. PCA

- Remember that principal components span the plane that lies closest (i.e. minimizes the mean squared error) to the data

- An autoencoder with:
  - Only linear activations
  - Mean Squared Error (MSE) as the loss function

  projects the data onto the same plane as principal component analysis

- But: weight vectors might not be orthogonal, scores will tend to have equal variance

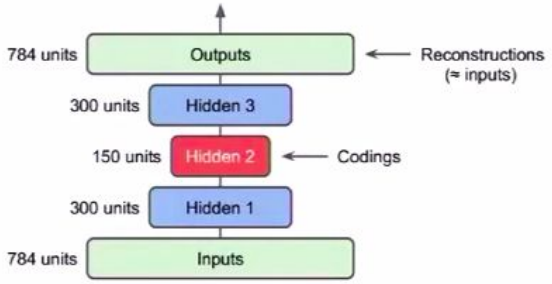


remember our interpretation of PCA: you're finding the plane that lies closest with average Euclidean distance
instead favour solutions/scores that have equal variance
in PCA → variance decaying
in autoencoders → variance of each dimension/across dimensions are all similar (advantage of autoencoders → you want to have similar variance across all your features in many cases)

basic structure (encoder and decoder with three layers above)
you can make this more complicated → use a number of stacked layers of your network, gradually decrease the dimensions of your data
have to be careful → more layers, many more parameters in your network to train
may lead to overfitting
what you can do to address this = tie the weights of the encoding and decoding layers → reduce your number of parameters by one half
then you use regularization techniques (drop out for example)

Stacked Autoencoders

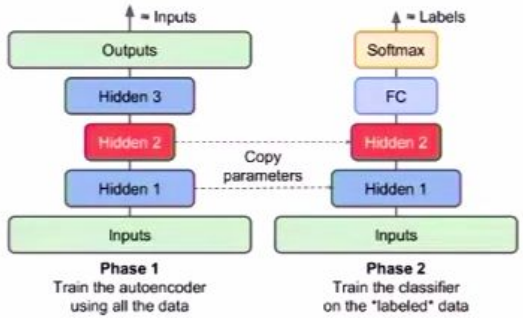
- Stacked autoencoders: autoencoders with multiple hidden layers
- Able to learn more complex codings
- Too many layers might lead to overfitting!
- Typically the architecture is symmetric with regards to the coding layer
- Weights can be "tied" between decoder and encoder layers (speeds up training and limits overfitting)



in some cases you might want to use unsupervised learning through autoencoders as a preprocessing step for supervised learning
e.g. you have a very complex supervised task, but you don't have a lot of labelled training data
first try google → reuse layers OR
use an autoencoder to learn useful features from your data

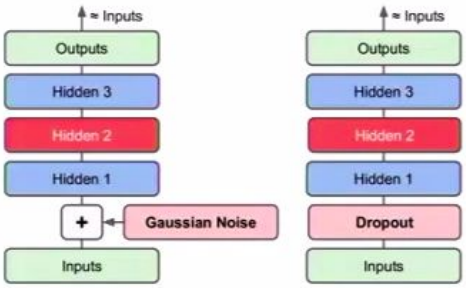
Unsupervised Pretraining Using Stacked Autoencoders

- Suppose you are tackling a complex supervised task but you do not have a lot of labelled training data
- One solution is to find a neural network that performs a similar task, and then reuse its lower layers
- Your neural network won't have to learn all the low-level features; it will just reuse the feature detectors learned by the existing net
- Similarly, you can first train a stacked autoencoder using all the data (labelled and unlabelled), then reuse the lower layers to create a neural network for your actual task, and train it using the labelled data



Take your input, then you somehow degrade it (by adding Gaussian noise, using dropout = set a random % of your data to 0), then encode it and try to learn network parameters such that the denoised output is similar to your original input

- Denoising autoencoders:
  - Add noise to its inputs, train it to recover the original, noise-free inputs

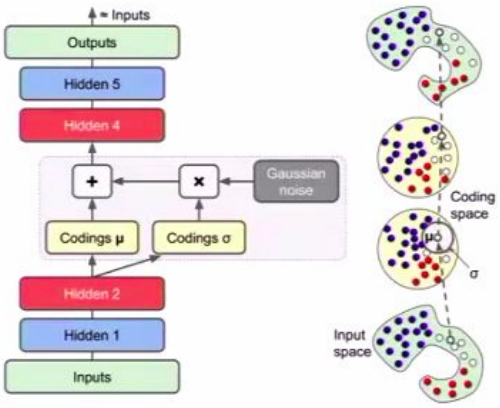


- Sparse autoencoders:
  - Force the autoencoder to represent each input as a combination of a small number of activations
  - Each neuron in the coding layer typically ends up representing a useful feature

Generative models: these are a bit different from standard autoencoders
in the middle layer → where you generally learn the low dimensional encoding of your data
in this case you learn a series of probability distributions, generally 1 dimensional guassians defined by a mean mu and a standard deviation sigma
the output of these types of networks is not deterministic - each time you feed an input to this type of network, they reach the middle layer, at which point you sample from the learned distributions that you have and then you send the sample data through the decoder so that you have a non-deterministic output that will be similar to your input but not exactly the same
use these generative models to produce data that is similar to your input
can be used for supervised learning → when your model is severely overfitting your data, and the normal ways you can use to reduce overfitting (e.g. drop out) is not sufficient, then you can use this

Autoencoder Variants

- Variational autoencoders:
  - Probabilistic: their outputs are partly determined by chance
  - Generative: they can generate new instances that look like they were sampled from the training set



- Many others!

Non-Negative Matrix Factorization (NMF)

again
unsupervised ML algorithm

feature extraction, dimensionality reduction
again: *decomposing each data point into a weighted sum of a fixed set of components*

trying to write each data point as a weighted sum of some components

wanting components and coefficients to be non-negative (>= 0)

can only be applied to data where each feature is NON-NEGATIVE

identify original components that make up the combined data
more interpetable than PCA

In[36]:

```
from sklearn.decomposition import NMF
nmf = NMF(n_components=15, random_state=0)
nmf.fit(X_train)
X_train_nmf = nmf.transform(X_train)
X_test_nmf = nmf.transform(X_test)
fix, axes = plt.subplots(3, 5, figsize=(15, 12),
                         subplot_kw={'xticks': (), 'yticks': ()})
for i, (component, ax) in enumerate(zip(nmf.components_, axes.ravel())):
    ax.imshow(component.reshape(image_shape))
    ax.set_title("{}. component".format(i))
```

```
compn = 3
# sort by 3rd component, plot first 10 images
inds = np.argsort(X_train_nmf[:, compn])[::-1]
fig, axes = plt.subplots(2, 5, figsize=(15, 8),
                         subplot_kw={'xticks': (), 'yticks': ()})
for i, (ind, ax) in enumerate(zip(inds, axes.ravel())):
    ax.imshow(X_train[ind].reshape(image_shape))
compn = 7
# sort by 7th component, plot first 10 images
inds = np.argsort(X_train_nmf[:, compn])[::-1]
fig, axes = plt.subplots(2, 5, figsize=(15, 8),
                         subplot_kw={'xticks': (), 'yticks': ()})
for i, (ind, ax) in enumerate(zip(inds, axes.ravel())):
    ax.imshow(X_train[ind].reshape(image_shape))
```

```
S = mglearn.datasets.make_signals()
plt.figure(figsize=(6, 1))
plt.plot(S, '-')
plt.xlabel("Time")
plt.ylabel("Signal")
```

In[39]:

```
# mix data into a 100-dimensional state
A = np.random.RandomState(0).uniform(size=(100, 3))
X = np.dot(S, A.T)
print("Shape of measurements: {}".format(X.shape))
```

Out[39]:
Shape of measurements: (2000, 100)
We can use NMF to recover the three signals:
In[40]:

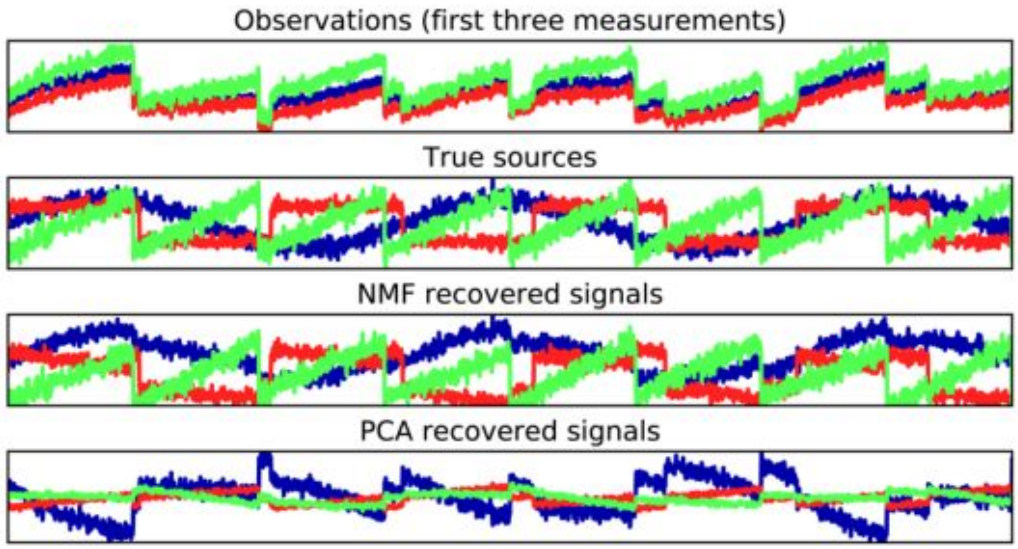
```
nmf = NMF(n_components=3, random_state=42)
S_ = nmf.fit_transform(X)
print("Recovered signal shape: {}".format(S_.shape))
```

Out[40]:
Recovered signal shape: (2000, 3)
For comparison, we also apply PCA:
In[41]:

```
pca = PCA(n_components=3)
H = pca.fit_transform(X)
```

Figure 3-19 shows the signal activity that was discovered by NMF and PCA:
In[42]:

```
models = [X, S, S_, H]
names = ['Observations (first three measurements)',
         'True sources',
         'NMF recovered signals',
         'PCA recovered signals']
fig, axes = plt.subplots(4, figsize=(8, 4), gridspec_kw={'hspace': .5},
                         subplot_kw={'xticks': (), 'yticks': ()})
for model, name, ax in zip(models, names, axes):
    ax.set_title(name)
    ax.plot(model[:, :3], '-')
```



Manifold Learning with t-SNE

one of a class of algorithms
visualisation

allow for complex mapping, provide better visualisations

find a 2d representation of the data the preserves the distances between the points as best as possible

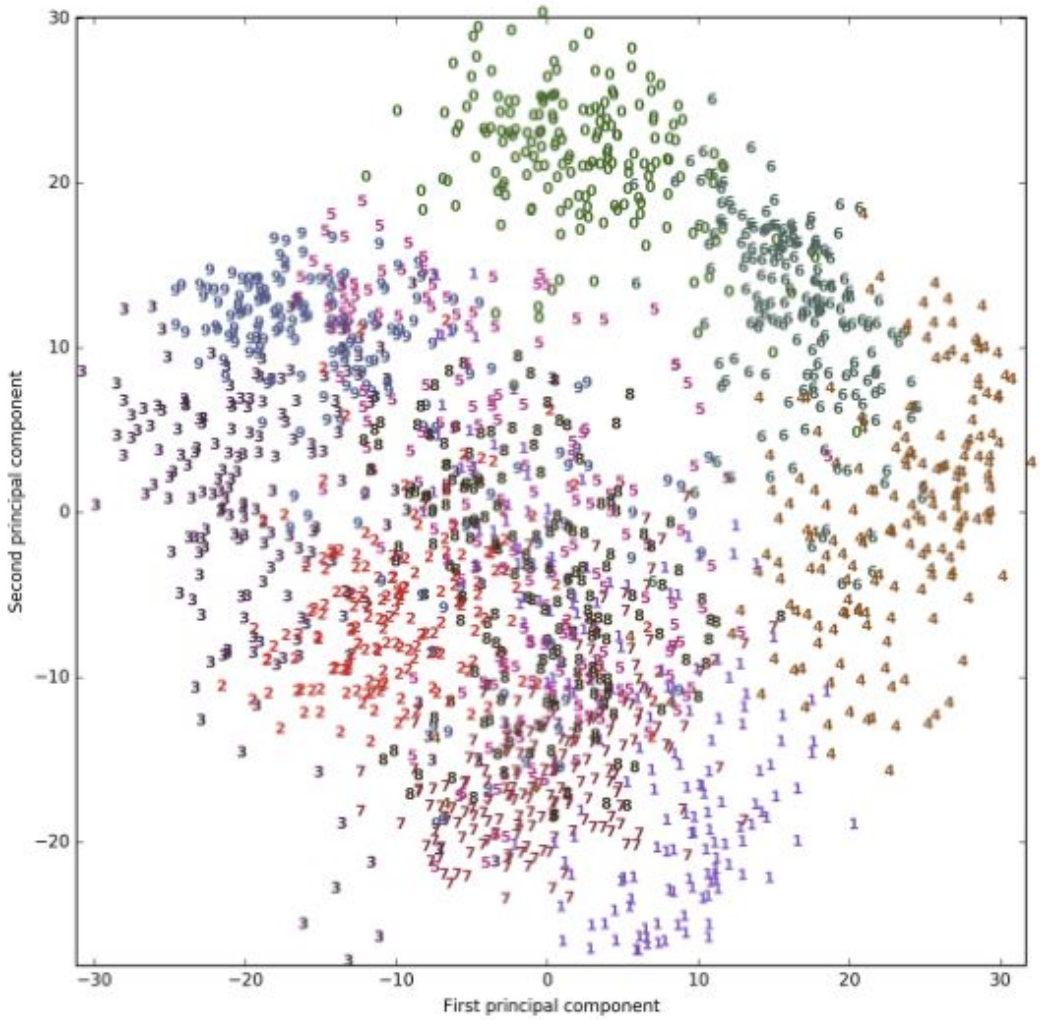
starts with a random 2D representation for each data point
then tries to make points that are close in the original feature space closer, points that are far apart in the original feature space farther apart
more emphasis on the points that are close by (*preserve information indicating which points are neighbors to each other)*

n[43]:
from sklearn.datasets import load_digits

```
digits = load_digits()
fig, axes = plt.subplots(2, 5, figsize=(10, 5),
                         subplot_kw={'xticks':(), 'yticks': ()})
for ax, img in zip(axes.ravel(), digits.images):
    ax.imshow(img)
```

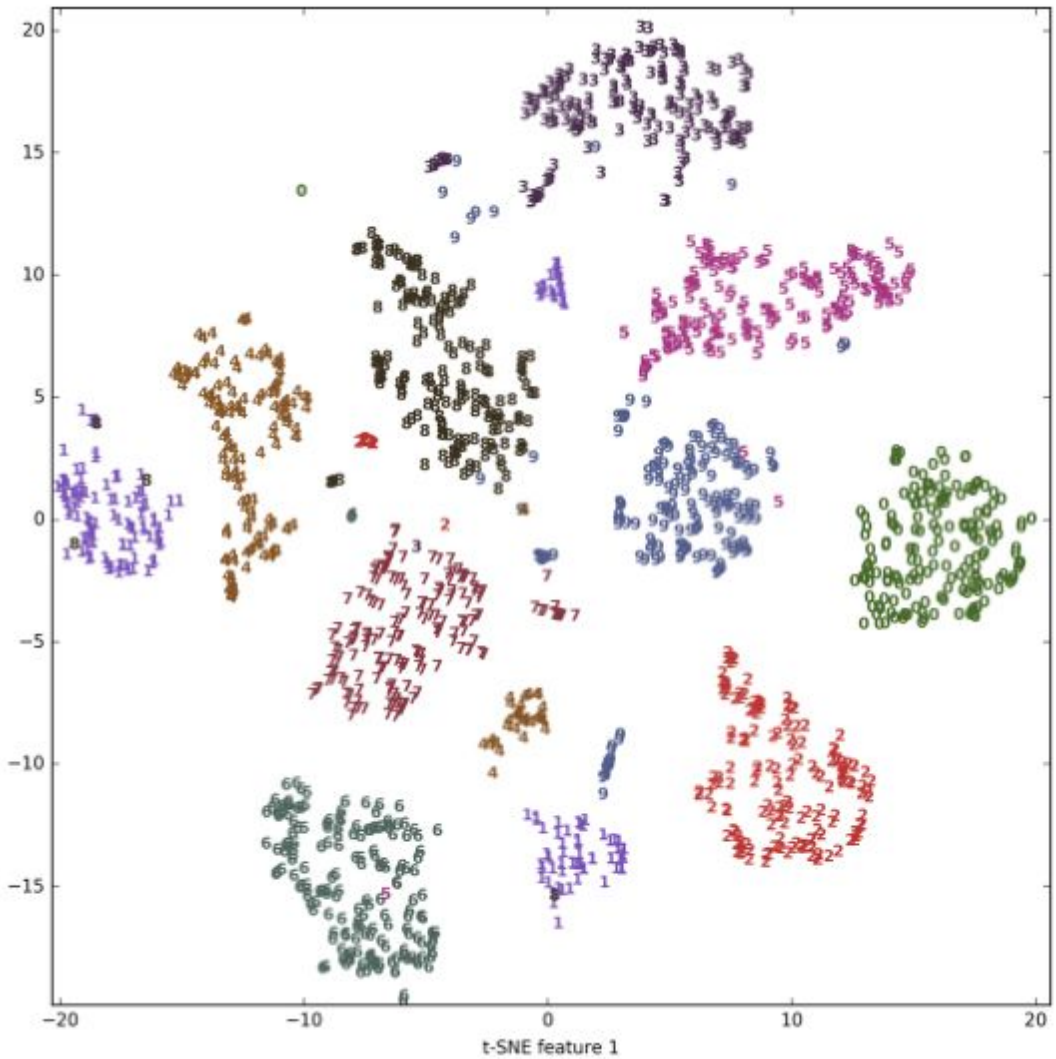
In[44]:

```
# build a PCA model
pca = PCA(n_components=2)
pca.fit(digits.data)
# transform the digits data onto the first two principal components
digits_pca = pca.transform(digits.data)
colors = ["#476A2A", "#7851B8", "#BD3430", "#4A2D4E", "#875525",
          "#A83683", "#4E655E", "#853541", "#3A3120", "#535D8E"]
plt.figure(figsize=(10, 10))
plt.xlim(digits_pca[:, 0].min(), digits_pca[:, 0].max())
plt.ylim(digits_pca[:, 1].min(), digits_pca[:, 1].max())
for i in range(len(digits.data)):
    # actually plot the digits as text instead of using scatter
    plt.text(digits_pca[i, 0], digits_pca[i, 1], str(digits.target[i]),
             color = colors[digits.target[i]],
             fontdict={'weight': 'bold', 'size': 9})
plt.xlabel("First principal component")
plt.ylabel("Second principal component")
```




```
from sklearn.manifold import TSNE
tsne = TSNE(random_state=42)
# use fit_transform instead of fit, as TSNE has no transform method
digits_tsne = tsne.fit_transform(digits.data)


plt.figure(figsize=(10, 10))
plt.xlim(digits_tsne[:, 0].min(), digits_tsne[:, 0].max() + 1)
plt.ylim(digits_tsne[:, 1].min(), digits_tsne[:, 1].max() + 1)
for i in range(len(digits.data)):
```

```
    # actually plot the digits as text instead of using scatter
    plt.text(digits_tsne[i, 0], digits_tsne[i, 1], str(digits.target[i]),
             color = colors[digits.target[i]],
             fontdict={'weight': 'bold', 'size': 9})
plt.xlabel("t-SNE feature 0")
plt.xlabel("t-SNE feature 1")
```



Clustering (still Chapter 7)

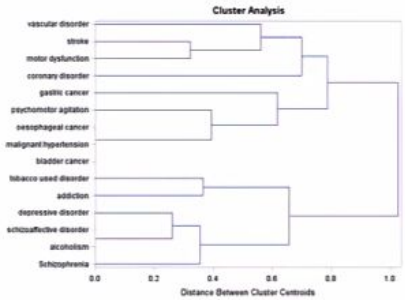
Introduction

Definition

- A cluster is a collection of observations that are “similar” between them and “dissimilar” to observations in other clusters

Motivation

- Find similar observations (e.g. patients)
- Generate hierarchical clusters (understand relationships, e.g. among diseases)



Dendrogram of disorders associated to the same genes as tobacco use disorder [de Viron S et al., 2014]

- Develop effective classifiers

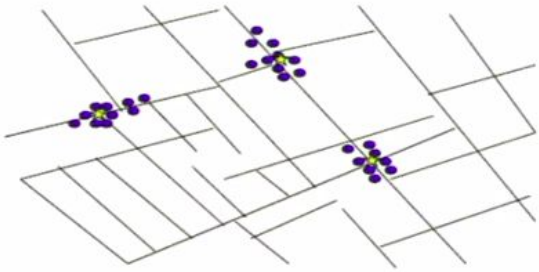
- PCA looks for a low-dimensional representation of the observations that explains a good fraction of the variance

- Clustering looks for homogeneous subgroups among the observations

hero of epidemiology: Jon Snow, plotted locations of cases on a map, all centered around infected wells.

Historic Application of Clustering

- John Snow, a physician from London, plotted the location of cholera deaths on a map during an outbreak in the 1850s
- The locations indicated that cases were clustered around certain intersections where there were polluted wells – thus exposing both the problem and the solution



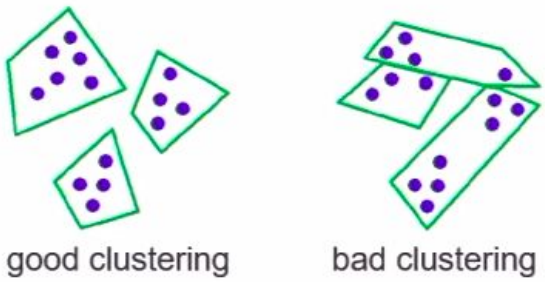
From: Nina Mishra HP Labs

you need:

- to define what observations are similar
- an algorithm to subdivide our data
- a way to evaluate

What Do We Need for Clustering?

- A measure of similarity
- A clustering algorithm
- A way to evaluate the quality of a clustering result



let's start with a similarity measure: what can we use here?
usually just squared eclidien distance
other metrics are possible: one norm (absolute value of the difference between your observations summed up= manhattan distance, cosine measure)

centroid = average for all your observations in your cluster
want to find the ideally the goal of clustering, you want to find a partition of the data that minimises the total distance between the data points and their assigned centroids

need to have some heuristic algorithms in order to minimise this cost.

Similarity Measure

- Usually: squared Euclidean distance

- Other metrics are possible
  - Manhattan distance
  - Cosine measure (dot product), e.g. for document clustering

- For a given a cluster D_j, define its centroid as

$$\mu_j = \frac{1}{|D_j|} \sum_{x \in D_j} x$$

- Ideally, we wish to find a partition that minimises the total distance between data points and their assigned centroids

$$\sum_{j=1}^{K} \sum_{\mathbf{x} \in D_j} \|\mathbf{x} - \boldsymbol{\mu}_j\|_2^2$$

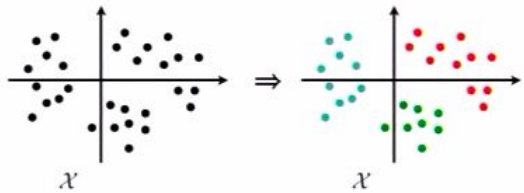
- Minimising this exactly is computationally difficult (NP-hard) so in practice we usually use heuristic algorithms

k-means → all the clusters are generated at once, need to know how many clusters you want a priori

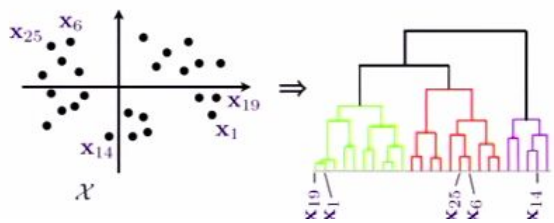
hierarchical clustering → don't need to know the number of clusters a priori

Clustering Algorithms

- Partitional (flat) clustering: k-means
  - all clusters are generated at once



- Hierarchical clustering
  - results in a tree-like visual representation of observations (dendrogram)
  - any "horizontal cut" of the dendrogram corresponds to a partitional clustering



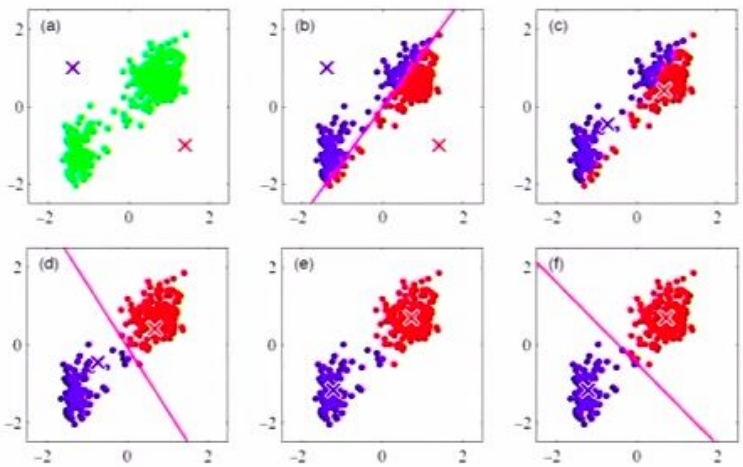
k means

k-means
takes two random points as a centroid
assign all the points to the closest centroid
recompute the centroid of each cluster
iterate it
and so on until a convergence

K-Means

- K-means (MacQueen, 1967) is a partitional clustering algorithm

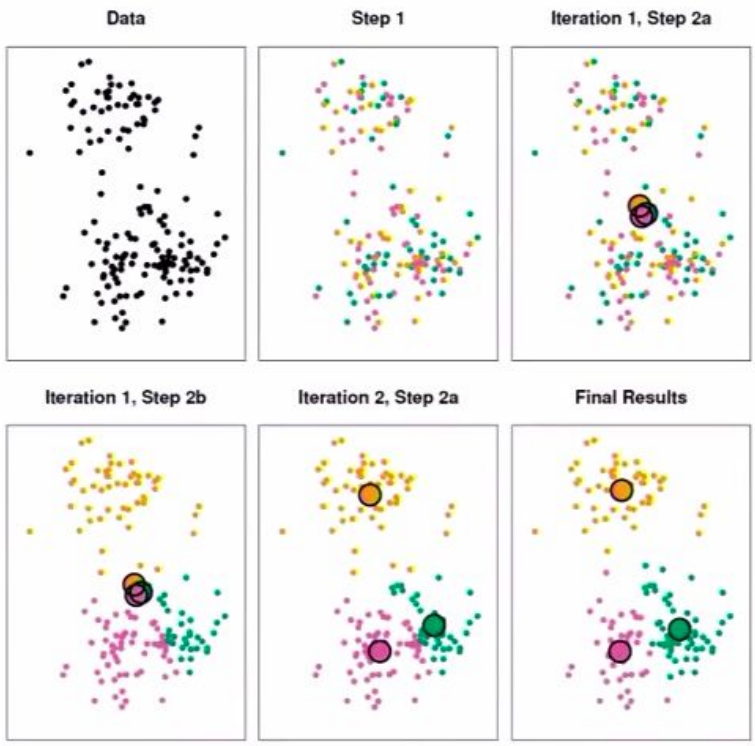
- Basic K-means algorithm for finding *K* clusters:
  1. Select *K* points as the initial centroids
  2. Assign all points to the closest centroid
  3. Recompute the centroid of each cluster
  4. Repeat steps 2 and 3 until the centroids don't change



alternative: start by assigning your data to a random cluster
compute centroids, assign data to the closest centroid and iterate until convergence, same as before

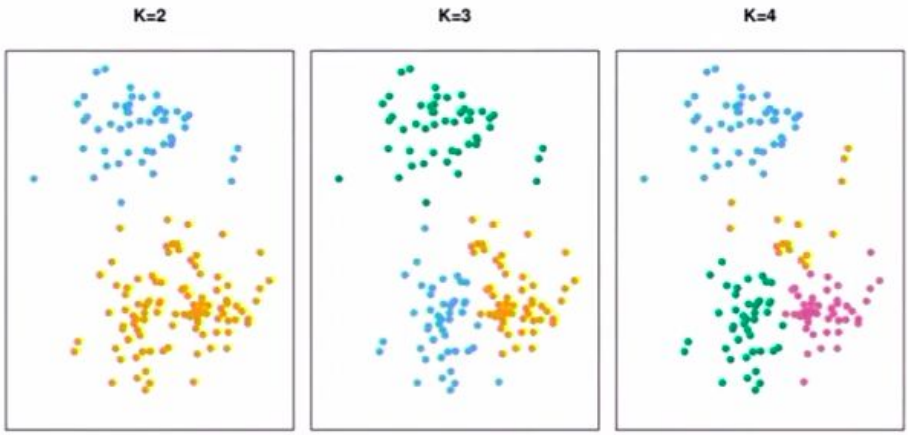
K-Means

- Equivalent alternative: start by randomly assigning each point to a cluster



how do you choose the number of clusters:
either have expert knowledge (know the number of clusters you will have)
or maximise some sort of clustering criteria (one example of this is the silhouette score, tells you how well each data point belongs to the assigned cluster)

How Many Clusters?



- Fix the number of clusters beforehand (expert knowledge required)
- Maximize clustering quality criterion (see e.g. *silhouette* score)

convergence is guaranteed → not posting the proof here,
important to note that the result of kmeans might not be the global optimum, because you're starting with a random initialisation → have to run it multiple times to find result that is closer to the global optimum
sensitive to outliers → remove from your data, or if you have a lot you can try subsampling
by far the most popular clustering algorithm, very efficient and simple
introduced in the 60s, still no clear evidence that the other, more sophisticated algorithms perform better

K-Means

- Guaranteed convergence after a finite number of steps
  (every step reduces the squared distance between data points and cluster centroids)
- Result might not be the global optimum
  → run multiple times with different random initial conditions

- Sensitive to outliers
  → remove outliers or perform subsampling of data

- Most popular clustering algorithm: efficient and simple
- No clear evidence that any other clustering algorithm performs better in general

clustering = partitioning the dataset into groups (clusters)

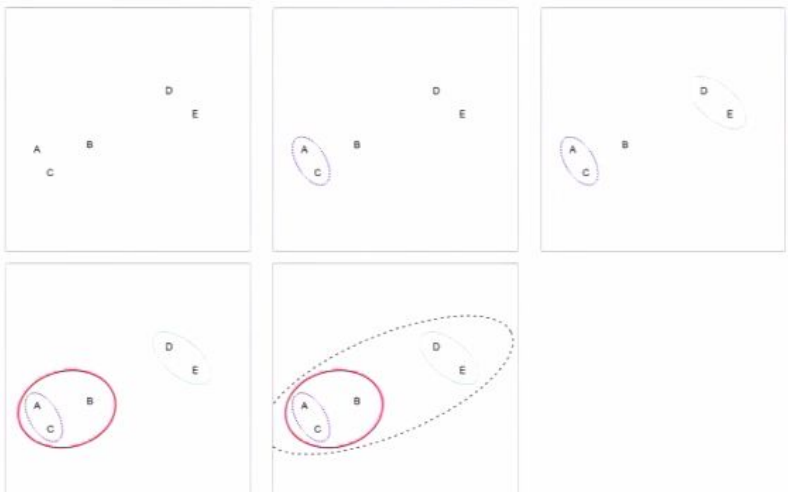
- points within a single cluster are very similar
- points in different clusters are different

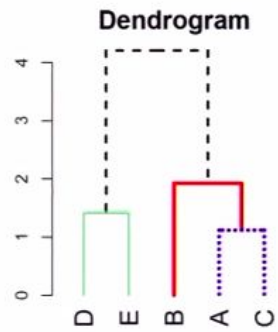
assign/predict a number to each data point (the cluster that point belongs to)

Hierarchical clustering:

Hierarchical Clustering

- Basic (agglomerative) hierarchical clustering algorithm:
  1. Start with each point as an individual cluster
  2. Compute the pairwise similarity between all clusters
  3. Merge the most similar two clusters
  4. Repeat steps 2 and 3 until only a single cluster remains



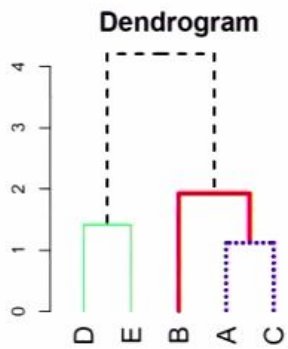


then you iterate

scale of features matters → normalise your data before applying

Hierarchical Clustering

- Dendrogram height corresponds to dissimilarity between clusters

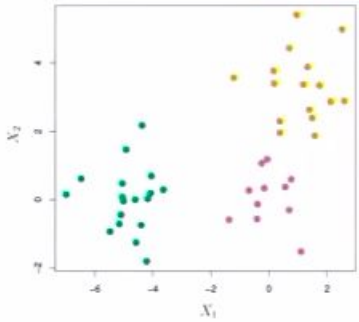


- Scale of features matters (→ normalize)
- There are different options to compute distances between clusters (see linkage functions)
- For example, centroid based linkage uses the distance between the clusters' centroids

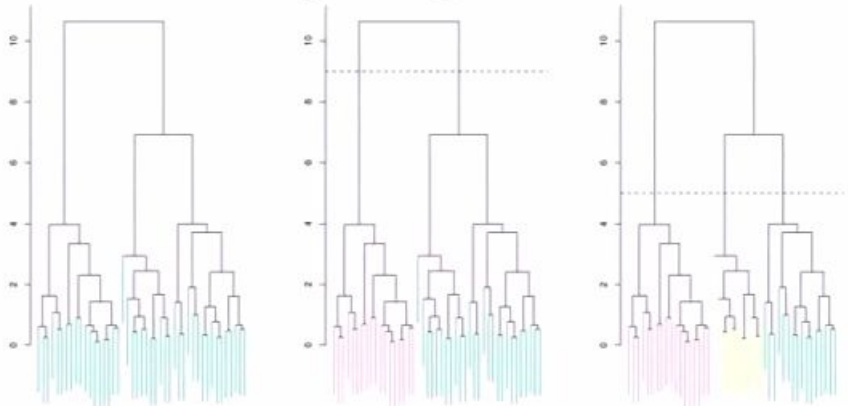
you can use centroids or data point distances

Hierarchical Clustering Example

- Observations from three classes generated in two-dimensional space



- Hierarchical clustering dendrogram



internal quality → doesn't require any expert knowledge
external quality→ comparing it to known classes

Evaluation of Cluster Quality

- Internal quality measure:
  - no external knowledge needed
  - e.g. overall similarity

- External quality measure:
  - compares clustering result to known classes
  - e.g. entropy, F-measure

overall similarity = basically the cost function that you're trying to minimise when you're doing clustering

Internal Quality: Overall Similarity (Inertia)

- Total distance between data points and their assigned centroids

$$\sum_{j=1}^{K} \sum_{x \in D_j} \|x - \mu_j\|_2^2$$

- Equal to the sum of pairwise distances between observations within a cluster divided by (two times) the number of observations

$$\sum_{j=1}^{K} \frac{1}{2|D_j|} \sum_{x,y \in D_j} \|x - y\|_2^2$$

for external quality ⇒ you have
entropy, concept that has been borrowed from physics
tells you the degree of disorder or randomness in the system
the lower the better →
start by computing the class distribution → the probability of each class being assigned to each cluster (remember external, have knowledge of the classes)

then you can compute the total entropy of the cluster, the sum of these probabilities multiplied by the log of these probabilities
total entropy = sum of the entropies of each cluster weighted by the size of each cluster

External Quality: Entropy

- Entropy: degree of disorder or randomness in the system (the lower the better)
- Start by computing the class distribution

$$p_{ij} = \frac{members\ of\ class\ i\ in\ cluster\ j}{size\ of\ cluster\ j}$$

- Entropy of cluster *j*

$$E_j = -\sum_i p_{ij} \log(p_{ij})$$

- Total entropy
  - sum of the entropies of each cluster weighted by the size of each cluster

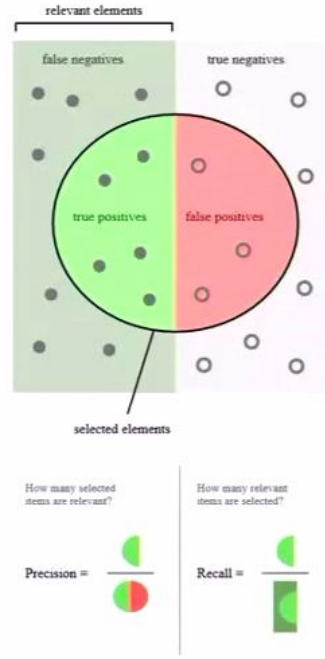
$$E_{CS} = \sum_{j=1}^{m} \frac{n_j * E_j}{n}$$

n_j: size of cluster *j*, *m*: number of clusters, *n*: total number of data points

F-measure → combines precision and recall into one single value
heirarchical clustering → can cut your dendrogram into different heights
take the maximum of each of your cuts

External Quality: F-Measure

- The F-measure combines the precision and recall ideas from information retrieval
- selected elements ≈ cluster *j*
- relevant elements ≈ class *i*
- $Precision(i,j) = \frac{members\ of\ class\ i\ in\ cluster\ j}{size\ of\ cluster\ j}$
- $Recall(i,j) = \frac{members\ of\ class\ i\ in\ cluster\ j}{size\ of\ class\ i}$
- $F(i,j) = 2\frac{Precision(i,j) * Recall(i,j)}{Precision(i,j) + Recall(i,j)}$



Source: wikipedia.org

- For hierarchical clustering the F-measure of class *i* is the maximum value it attains at any node in the tree
- Overall F-measure

$$F = \sum_i \frac{n_i}{n} \max\{F(i,j)\}$$

max over all clusters at all levels, n_i: size of class *i*,
n: total number of data points

Conclusions

- Hierarchical clustering groups nearest neighbours while k-means relies on global class properties

- However for k-means:
  - number of clusters needs to be defined a priori
  - sensitive to outliers
  - might require multiple runs to produce a good result

- K-means might produce clusters of widely different sizes
  → extensions such as bisecting k-means (practical exercise)

Conclusions (continued)

- Clustering is highly application dependent and to some extent subjective (no gold standard)

- Unsupervised learning is a very active area of research
- Improvements in unsupervised learning might lead to considerable progress in automated feature extraction and scientific discovery

- Reference: M Steinbach, G Karypis, V Kumar, "A Comparison of Document Clustering Techniques", Technical Report ##00-034, 2000
- Some graphics are from slides/books by Teemu Roos, Trevor Hastie, and Aurélien Géron

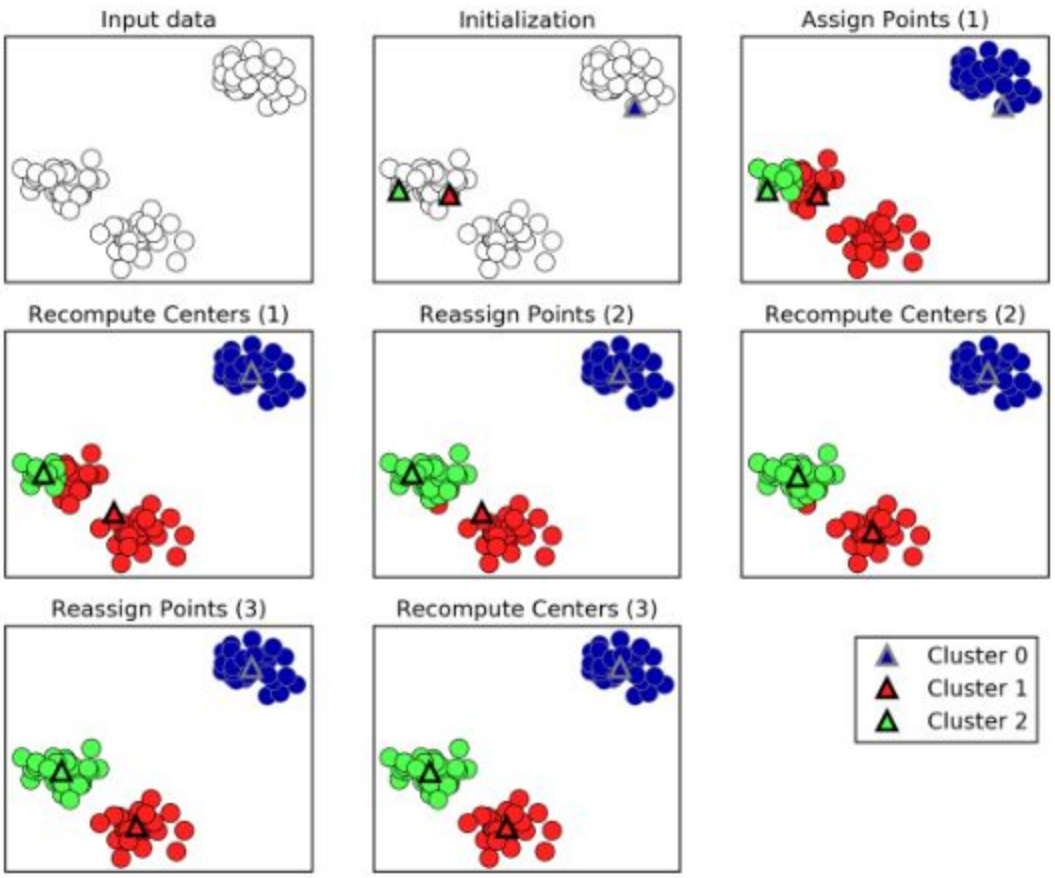
k-Means Clustering

finds cluster centers = certain regions of the data

alternates between:

1. assigning each data point to the closest cluster center
2. setting each cluster center as the mean of the data points assigned to it

finishes when: assignment of instances to clusters no longer changes



specify the number of clusters you're looking for (e.g. 3)
initialised: declares three data points randomly as cluster centers
iterative algorithm
each data point is assigned to the cluster center that is closest
then the cluster centers are updated to be the mean of the assigned poitns
repeated
on the third iteration, assignment of points to cluster centers remained unchanged, algorithm stops

In[49]:

```
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans

# generate synthetic two-dimensional data
X, y = make_blobs(random_state=1)

# build the clustering model
```

```
kmeans = KMeans(n_clusters=3)
kmeans.fit(X)
During the algorithm, each training data point in  is assigned a c
```

n_clusters is set to 8 by default !!

kmeans.labels_ attribute = the cluster labels for each of the data points (starts with 0)

```
In[50]:

    print("Cluster memberships:\n{}".format(kmeans.labels_))

Out[50]:

    Cluster memberships:
    [1 2 2 2 0 0 0 2 1 1 2 2 0 1 0 0 0 1 2 2 0 2 0 1 2 0 0 1 1 0 1 1 0 1 2 0 2
     2 2 0 0 2 1 2 2 0 1 1 1 1 2 0 0 0 1 0 2 2 1 1 2 0 0 2 2 0 1 0 1 2 2 2 0 1
     1 2 0 0 1 2 1 2 2 0 1 1 1 1 2 1 0 1 1 2 2 0 0 1 0 1]
```

you can assign cluster labels to new points, using the predict method

```
In[51]:

    print(kmeans.predict(X))

Out[51]:

    [1 2 2 2 0 0 0 2 1 1 2 2 0 1 0 0 0 1 2 2 0 2 0 1 2 0 0 1 1 0 1 1 0 1 2 0 2
     2 2 0 0 2 1 2 2 0 1 1 1 1 2 0 0 0 1 0 2 2 1 1 2 0 0 2 2 0 1 0 1 2 2 2 0 1
     1 2 0 0 1 2 1 2 2 0 1 1 1 1 2 1 0 1 1 2 2 0 0 1 0 1]
```

similar to classification! but, the labels have no a priori meaning

Failure of k-means

even if you know the right number of clusters for the dataset
kmeans might not always bea ble to recover them

each cluster is defined solely by its center (i.e. each cluster is a convex shape)

assumes all directions are equally important for each cluster

performs poorly if the clusters have complex shapes

Vector quantization, k-means as decomposition

k-means is a clustering algorithm!
represents each data point using a cluster center i.e. each data point is represented using a single component = vector quantization

In[57]:

```
X_train, X_test, y_train, y_test = train_test_split(
   X_people, y_people, stratify=y_people, random_state=0)
nmf = NMF(n_components=100, random_state=0)
nmf.fit(X_train)
pca = PCA(n_components=100, random_state=0)
pca.fit(X_train)
kmeans = KMeans(n_clusters=100, random_state=0)
kmeans.fit(X_train)
X_reconstructed_pca = pca.inverse_transform(pca.transform(X_test))
X_reconstructed_kmeans = kmeans.cluster_centers_[kmeans.predict(X_test)]
X_reconstructed_nmf = np.dot(nmf.transform(X_test), nmf.components_)
```

In[58]:

```
fig, axes = plt.subplots(3, 5, figsize=(8, 8),
                         subplot_kw={'xticks': (), 'yticks': ()})
fig.suptitle("Extracted Components")
for ax, comp_kmeans, comp_pca, comp_nmf in zip(
       axes.T, kmeans.cluster_centers_, pca.components_, nmf.components_):
   ax[0].imshow(comp_kmeans.reshape(image_shape))
   ax[1].imshow(comp_pca.reshape(image_shape), cmap='viridis')
   ax[2].imshow(comp_nmf.reshape(image_shape))
axes[0, 0].set_ylabel("kmeans")
axes[1, 0].set_ylabel("pca")
axes[2, 0].set_ylabel("nmf")
```

```
fig, axes = plt.subplots(4, 5, subplot_kw={'xticks': (), 'yticks': ()},
                         figsize=(8, 8))
fig.suptitle("Reconstructions")
for ax, orig, rec_kmeans, rec_pca, rec_nmf in zip(
        axes.T, X_test, X_reconstructed_kmeans, X_reconstructed_pca,
        X_reconstructed_nmf):
    ax[0].imshow(orig.reshape(image_shape))
    ax[1].imshow(rec_kmeans.reshape(image_shape))
    ax[2].imshow(rec_pca.reshape(image_shape))
    ax[3].imshow(rec_nmf.reshape(image_shape))
axes[0, 0].set_ylabel("original")
axes[1, 0].set_ylabel("kmeans")
axes[2, 0].set_ylabel("pca")
axes[3, 0].set_ylabel("nmf")
```

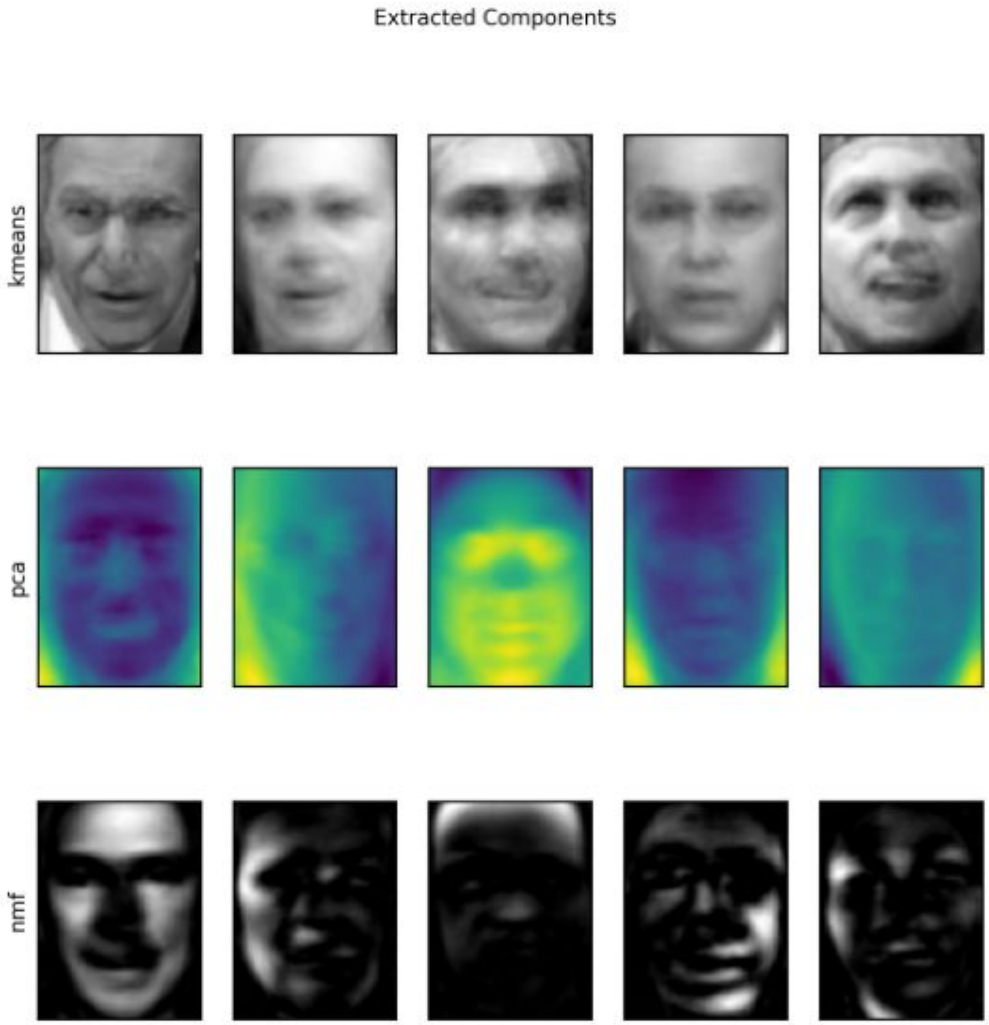


Figure 3-30. Comparing k-means cluster centers to components found by PCA and NMF

Figure 3-31. Comparing image reconstructions using k-means, PCA, and NMF with 100 components (or cluster centers)—k-means uses only a single cluster center per image

kmeans

- can use many more clusters than input dimension to encode the data

STRENGTHS:

- runs quickly
- easy to understand
- scales to large datasets

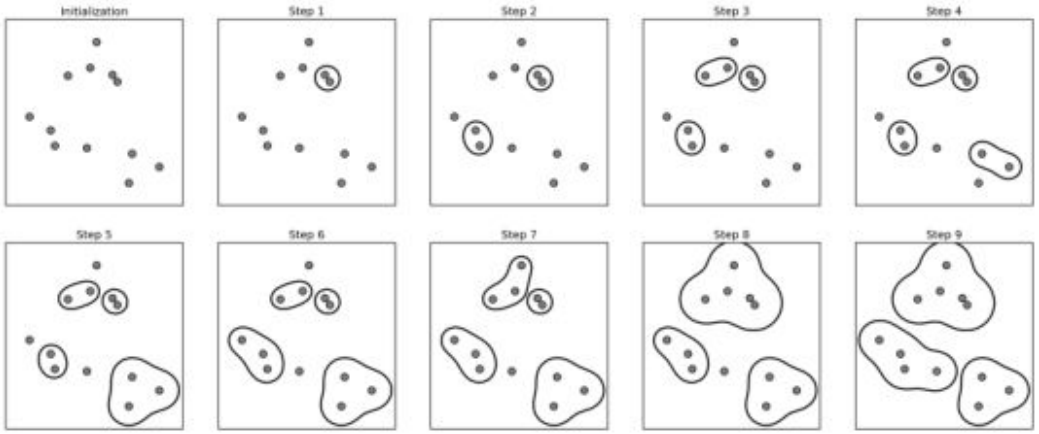
- MiniBatchKMeans class in scikit-learn = can handle very large datsets

WEAKNESSES
- relies on random initialization
- runs the algorithm 10 times with 10 different random initializations by default ⇒ returns the best result

Agglomerative Clustering

collection of clustering algorithms

algorithm
1. declares each point in its own cluster
2. merges the two most similar clusters (by linkage criteria) until some stopping criterion is satisfied (in scikit learn = the number of clusters) →
3. similar clusters merged until only the specified number of clusters are left



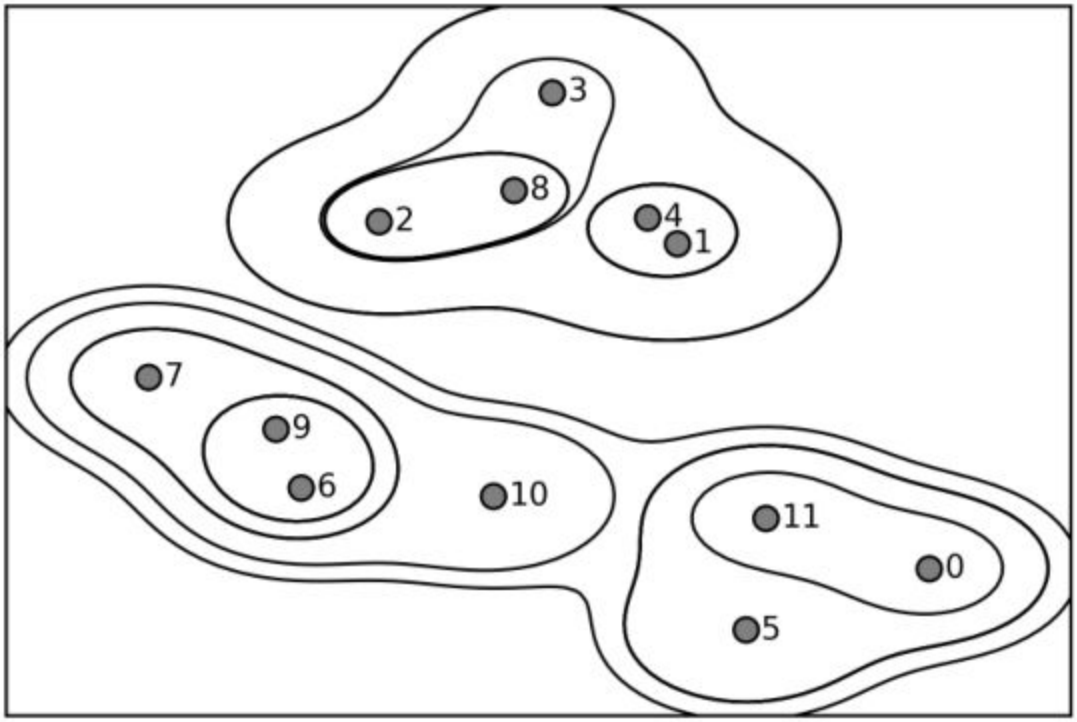
linkage criteria in scikit-learn:

a. ward = default, picks the two clusters to merge such that the variance within all clusters increases the least (clusters relatively equally sized)

b. average = merges the two clusters that have the smallest average distance between al their points
c. complete = merges the two clusters that have the smallest maximum distance between their points

```
from sklearn.cluster import AgglomerativeClustering
X, y = make_blobs(random_state=1)
agg = AgglomerativeClustering(n_clusters=3)
assignment = agg.fit_predict(X)
mglearn.discrete_scatter(X[:, 0], X[:, 1], assignment)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

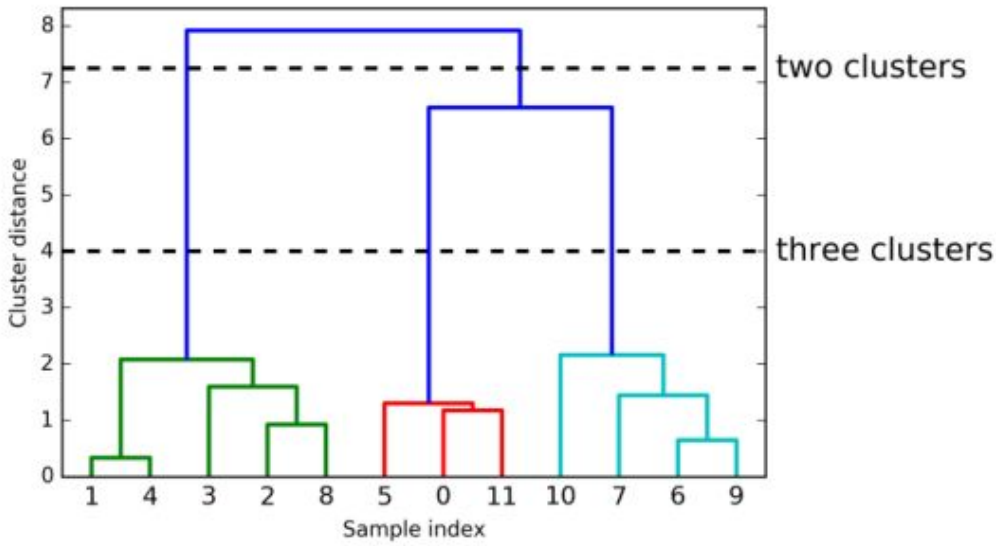
produces: hierarchical clustering



you can visualise this with a dendrogram → but scikit learn cannot draw them
you can do them with SciPy

```
# Import the dendrogram function and the ward clustering function from SciPy
from scipy.cluster.hierarchy import dendrogram, ward
X, y = make_blobs(random_state=0, n_samples=12)
# Apply the ward clustering to the data array X
# The SciPy ward function returns an array that specifies the distances
# bridged when performing agglomerative clustering
linkage_array = ward(X)
```

```
# Now we plot the dendrogram for the linkage_array containing the distances
# between clusters
dendrogram(linkage_array)
# Mark the cuts in the tree that signify two or three clusters
ax = plt.gca()
bounds = ax.get_xbound()
ax.plot(bounds, [7.25, 7.25], '--', c='k')
ax.plot(bounds, [4, 4], '--', c='k')
ax.text(bounds[1], 7.25, ' two clusters', va='center', fontdict={'size': 15})
ax.text(bounds[1], 4, ' three clusters', va='center', fontdict={'size': 15})
plt.xlabel("Sample index")
plt.ylabel("Cluster distance")
```



DBSCAN

“density based spatial clustering of applications with noise”

does not require you to set the number of clusters
can capture complex shapes
can identify points that are not part of any cluster

but it’s slower than agglomeratie clustering + k-means

- identifies points that are in crwoded regions of the feature space (where many data poins are close together)
- = dense regions in feature space
- clusters form dense regions, separated by regions that are relatively empty

points in dense region = core samples/points
parameters: min_samples, eps

If there are at least min_samples many data points within a distance of eps to a given data point, that data point is classified as a core sample.

Core samples that are closer to each other than the distance eps are put into the same cluster by DBSCAN.

picks an arbitrary point to begin
finds all the points with distance eps or less from that point
if there are less than min_samples points within distance eps of the starting point, point is labelled as *noise* (i.e. doest belong to any cluster)

if there are more than min_samples points within a distance of eps, the point is labelled as a core sample, assigned a cluster label

then → tries all points within eps of the point

visits all the neighbors of the core samples

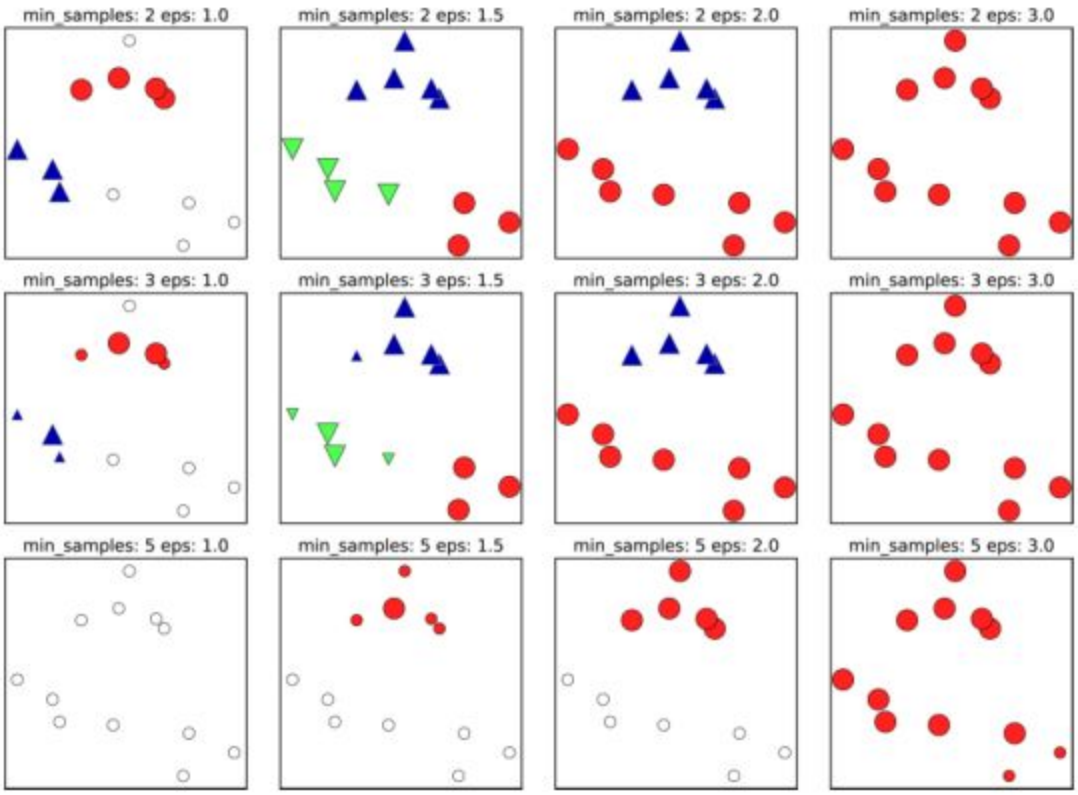
cluster grows until there are no more core samples within distance eps fo the cluster

then another point, not yet visited is picked, procedure repeats

in the end:
every data point is labelled as either a core point, points within distance eps of core points (boundary points), and noise

results always the same (core points, noise same) but boundary points might be neighbor to core points of more than one cluster

In[65]:
```
from sklearn.cluster import DBSCAN
X, y = make_blobs(random_state=0, n_samples=12)
dbscan = DBSCAN()
clusters = dbscan.fit_predict(X)
print("Cluster memberships:\n{}".format(clusters))
```
Out[65]:
```
Cluster memberships:
[-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1]
```
As you can see, all data points were assigned the label , whic



In[67]:

```
X, y = make_moons(n_samples=200, noise=0.05, random_state=0)
# rescale the data to zero mean and unit variance
scaler = StandardScaler()
scaler.fit(X)
X_scaled = scaler.transform(X)
dbscan = DBSCAN()
clusters = dbscan.fit_predict(X_scaled)
# plot the cluster assignments
plt.scatter(X_scaled[:, 0], X_scaled[:, 1], c=clusters, cmap=mglearn.cm2, s=60)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

Evaluating clustering algorithms (Chapter 7)

MUST USE A CLUSTERING METRIC, NOT ACCURACY_SCORE
accuracy requires the assigned cluster labels to match the ground truth → but the cluster labels are themselves meaningless!!

metrics:

adjusted rand index (ARI)

normalized mutual information (NMI)

give a measure between 0 and 1

```
from sklearn.metrics.cluster import adjusted_rand_score
X, y = make_moons(n_samples=200, noise=0.05, random_state=0)
# rescale the data to zero mean and unit variance
scaler = StandardScaler()
scaler.fit(X)
X_scaled = scaler.transform(X)
fig, axes = plt.subplots(1, 4, figsize=(15, 3),
                         subplot_kw={'xticks': (), 'yticks': ()})
# make a list of algorithms to use
algorithms = [KMeans(n_clusters=2), AgglomerativeClustering(n_clusters=2),
              DBSCAN()]
# create a random cluster assignment for reference
random_state = np.random.RandomState(seed=0)
random_clusters = random_state.randint(low=0, high=2, size=len(X))
# plot random assignment
axes[0].scatter(X_scaled[:, 0], X_scaled[:, 1], c=random_clusters,
                cmap=mglearn.cm3, s=60)
axes[0].set_title("Random assignment - ARI: {:.2f}".format(
        adjusted_rand_score(y, random_clusters)))
for ax, algorithm in zip(axes[1:], algorithms):
    # plot the cluster assignments and cluster centers
    clusters = algorithm.fit_predict(X_scaled)
```

```
ax.scatter(X_scaled[:, 0], X_scaled[:, 1], c=clusters,
           cmap=mglearn.cm3, s=60)
ax.set_title("{} - ARI: {:.2f}".format(algorithm.__class__.__name__,
                                       adjusted_rand_score(y, clusters)))
```

Sequential Data Models (Chapter 8)

We've only ever used "static" sets of data points (independent and identically distributed (i.i.d.)
likelihood function = product over all data points of the probability distribution

sequential data:

1. *time stamp matters* = temporal sequence (continuous measurements of vital signs, EEG, ECG, patients interacting with different GPs and specialists, rainfall measurements on successive days, daily values for currency exchange)
2. *order is important, but no time stamp* = ordered data = clinical notes, discharge summaries, GP records, genetic data, sequence of nucleotide base pairs along a strand of DNA, sequence of characters in a sentence
3. *fixed time-difference between the occurrence of successive data points* = time series

probabilistic sequential data models

- Markov Process
- Hidden Markov Model

neural networks for sequential data: recurrent neural networks (RNN)

- Long short-term memory

stationary → data evolves in time, distribution it is generated from remains the same
nonstationary → generative distribution itself is evolving in time

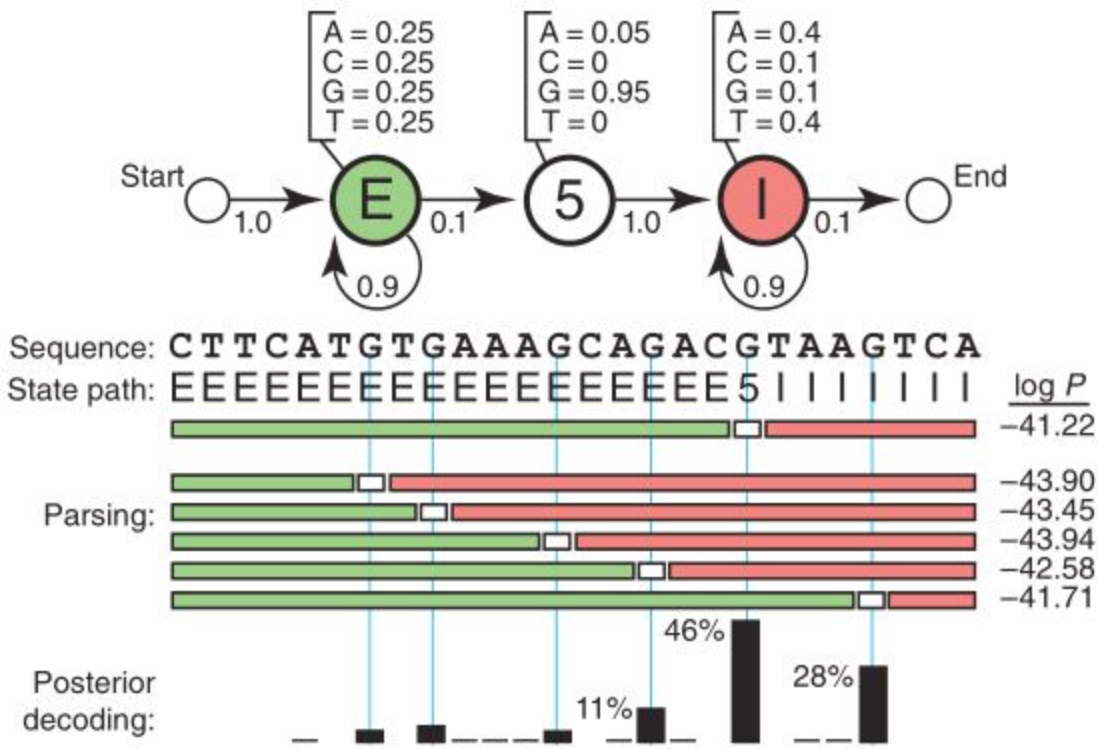
e.g financial forecasting → we wish to be able to predict the next value in a time series given observations of previous values

Hidden Markov Model (HMM)

- genefinding
- profile searches
- multiple sequence alignment
- regulatory site identification

= formal foundation for making probabilistic models of linear sequence labelling problems (supervised ML)

example:



on the 5' (5 PRIME) strand of DNA, trying to find where exon ends and becomes a non-coding intron

exons have a uniform base composition (25% for each of the nucleotide bases, UT CG)
introns = 40% A/T, 10% C/G
splice site (5'SS, where the exon ends) = consensus nucleotide is almost always a G (95% G, 5% A)

HMM invokes three states (one for the three labels, possible outputs you might give a nucleotide = exon (E), 5'SS (5), intron (I)

each state has its own *emission probabilities* → model the base composition (above the state on the diagram)

each state also has *transition probabilities* (arrows) → the probabilities of moving from this state to a new state = the linear order in which we expect the states to occur (one or more Es, one 5, one or more Is)

when you visit a state, you emit a residue from the state's emission probability distribution → then you choose which state to visit next according to the state's transition probability distribution
→ the model thus generates two strings of information

1. the underlying state path, = the outputs, those E, 5 or I labels as you transition from state to state = THE MARKOV CHAIN, and
2. the observed sequence = the DNA, the residue being emitted from that state in the state path

what state you go to next depends only on what state you're in
you're only given the observed sequence, the underlying state path (the markov chain) is hidden
→ hence hidden Markov chain
hidden chain/state path = the residue labels/outputs that you want to infer

the probability that a HMM with parameters **Θ** generates a state path **π** and an observed sequence S

$$P(S,\pi|\mathrm{HMM},\theta)$$

= the product of all the emission probabilities and transition probabilities that were used!!! THIS IS THE PRODUCT RULE (FROM THE TEXTBOOK BELOW)

e.g.
26-nucleotide base DNA sequence, same state path
there are 27 transitions, 26 emissions
multiply all 53 probabilities together (and in the book, they take the log → because these are small numbers), you calculate

$$\log P(S,\pi|\mathrm{HMM},\theta) = -41.22.$$

HMM = a full probabilistic model i.e.

- the model parameters, and the overall sequence scores are all probabilities
- therefore → you can use Bayesian probability theory, manipulate these numbers TO OPTIMISE THE PARAMETERS, INTERPRET THE SIGNIFICANCE OF SCORES

ACTUALLY FINDING THE CORRECT/BEST STATE PATH
now, you're given a sequence, you want to infer the hidden state path/markov chain
there are potentially many state paths that could generate the same sequence

you want to find the the state path WITH THE HIGHEST PROBABILITY (= best scoring alignment)

e.g.
you are given the HMM with 26-nucleotide sequence,
there are 14 possible paths that have non-zero probability (since the insertion, 5'SS must fall on either A or G, and there are max 14 As or Gs possible)
all 6 of the highest scoring paths have G at the insertion zone 5'SS
best has log probability of -41.22 (i.e. a probability of 10^{-42} or $1/10^{42}$), with 5'SS at the fifth G

we use the
Viterbi algorithm
to find the most probable state path, given a sequence and an HMM

POSTERIOR DECODING

how confident are you that the fifth G is the right choice?
because you're using probabilistic modelling, you can calculate your confidence directly
the probability that residue i was emitted by state k is the sum of the probabilities of all the state paths that use k to generate residue (that is, πi = k in the state path π), normalized by the sum over all possible state paths.

in the example = one state path/sum over 14 state paths
probability of 46% that the best-scoring fifth G is correct, 28% that the sixth G position is correct

done using two dynamic programming algorithms, Forward and Backward (like Biterbi but sum over possible paths instead of choosing the best)

TO MAKE A HMM

making an HMM means specifying four things
1. the symbol alphabet (K different symbols (e.g. ACGT, K = 4))
2. the number of states in the model, M
3. the emission probabilities $e_i(x)$ for each state i, that sum to one over K symbols
   $\sum_x e_i(x) = 1;$,
4. the transition probabilities $t_i(j)$ for each state i going to any other state j (including itself)

any model with these properties is an HMM

Advantages:
so you can always graph it out (like the figure above)

graphical simplicity

Disadvantages:
don't deal well with correlations between residues, because they assume that each residue depends only on one underlying state
e.g.
RNA secondary structure analysis,
conserved RNA base pairs induce long-range pairwise correlations
one position might produce any residue, but the base paired partner must be complementary!!!
the HMM state path/markov chain → has no way of remembering what a distant state generated

---

Markov property = the conditional probability distribution of futures states of the process (conditional on both past and present states) depends only on the present state, not the sequences of events before it.

A process with this property = Markov process
Markov assumption = a model where the Markov property is assumed to hold

states:
- hidden = we don't have access to, most often the values we actually want to know (exons, introns, splice sequence)
- observable = we have access to (the letters of the DNA sequence) A, T, G, C

you cannot see the hidden states BECAUSE THAT IS WHAT YOU'RE TRYING TO PREDICT
you use the observable states to make these predictions

transition probability = the probability of moving from one hidden state to another
emission probabilities = the probability of an observation given a specific hidden state, emitted from the hidden state

HMM = generative, probabilistic model, the sequence of observable x variables (your inputs, DNA sequence ACGTA…) is generated by a sequence of internal hidden states Z (= your classes)

Markov chain = a model describing a sequence of possible events in which the probability of each event depends only on the state attained in the previous event

pi = start possibility vector
A = transition probability matrix
emission probability of an observable = distribution with paramteres theta conditioned on the current hidden state
entirely determined by pi, A and theta

e.g.
bob and dylan, friends that live far away from each other

bob can see how Dylan is feeling, knows his mood is influenced by the weather
sunny → probability he is happy is 0.8/80%, sad = 20%
weather today is dependent on the weather yesterday
if it was sunny yesterday, it will be sunny tomorrow (probability = 90%)
if it was rain, 60% probability it will be rainy again today)

hidden states = weather condition (Bob cannot tell these directly, just their transition probability)
observable states =- Dylan's mood
emission probabilities = probabilities of Dylan being in a particular mood given the weather condition

YOU GENERATE THE HMM FROM A SEQUENCE OF OBSERVATIONS

- done using the Baum-Welch algorithm, an Expectation-Maximization algorithm \
- = getting the "local optimal parameters" = the emission probabilities, the transition probabilities etc

THEN YOU FIND THE PROBABILITY OF AN OBSERVED SEQUENCE = evaluation

- use the Forward algorithm
- = calculates the probability of an observation sequence given a particular HMM, then you choose the most likely HMM to be our model

FIND THE SEQUENCE OF HIDDEN STATES THAT MOST PROBABLY GENERATED AN OBSERVED SEQUENCE = decoding

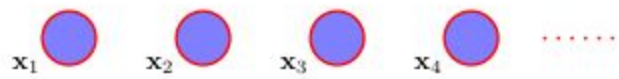
- use the Viterbi algorithm
- then you compare the decoded state path with the test set

---

recent observations more informative than historical observations?

assume that future predictions are independent of all but the most recent observations

Figure 13.2 The simplest approach to modelling a sequence of observations is to treat them as independent, corresponding to a graph without links.



simply ignore the sequential aspects, treat observations as independent, identically distributed however → fails to exploit the sequential patterns in the data!

relax the i.i.d assumption →

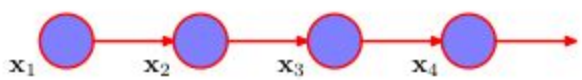
you can use the product rule to express the joint distribution for a sequence of observations in the form:

$$p(\mathbf{x}_1, \ldots, \mathbf{x}_N) = \prod_{n=1}^{N} p(\mathbf{x}_n | \mathbf{x}_1, \ldots, \mathbf{x}_{n-1}). \tag{13.1}$$

- product rule?

then assume that *each of the conditional distributions on the right-hand side is independent of all previous observations except the most recent, obtain*
first-order Markov chain →

A first-order Markov chain of observations $\{\mathbf{x}_n\}$ in which the distribution $p(\mathbf{x}_n|\mathbf{x}_{n-1})$ of a particular observation $\mathbf{x}_n$ is conditioned on the value of the previous observation $\mathbf{x}_{n-1}$.



now, the joint distribution for a sequence of N observations is:

$$p(\mathbf{x}_1, \ldots, \mathbf{x}_N) = p(\mathbf{x}_1) \prod_{n=2}^{N} p(\mathbf{x}_n | \mathbf{x}_{n-1}).$$

$$p(\mathbf{x}_n | \mathbf{x}_1, \ldots, \mathbf{x}_{n-1}) = p(\mathbf{x}_n | \mathbf{x}_{n-1})$$

thus: to predict the next observation in a sequence, the distribution of predictions will depend only on the value of the immediately preceding observation (which is independent of all earlier observations)

conditional distributions $p(x_n|x_{n-1})$ constrained to be equal (= stationary time sequence assumption) ⇒ *homogenous Markov chain*

predictions depending on the previous-but-one value ⇒ *second-order Markov chain*

$$p(\mathbf{x}_1, \ldots, \mathbf{x}_N) = p(\mathbf{x}_1) \prod_{n=2}^{N} p(\mathbf{x}_n|\mathbf{x}_{n-1}).$$

on property, we see that the conditional distri
the observations up to time n, is given by

$$p(\mathbf{x}_n|\mathbf{x}_1, \ldots, \mathbf{x}_{n-1}) = p(\mathbf{x}_n|\mathbf{x}_{n-1})$$

trends in the data over several, successive observations will provide important information in predicting the next value
→ move to higher-order Markov chains

$$p(\mathbf{x}_1, \ldots, \mathbf{x}_N) = p(\mathbf{x}_1)p(\mathbf{x}_2|\mathbf{x}_1) \prod_{n=3}^{N} p(\mathbf{x}_n|\mathbf{x}_{n-1}, \mathbf{x}_{n-2}).$$

A second-order Markov chain, in which the conditional distribution of a particular observation $\mathbf{x}_n$ depends on the values of the two previous observations $\mathbf{x}_{n-1}$ and $\mathbf{x}_{n-2}$.

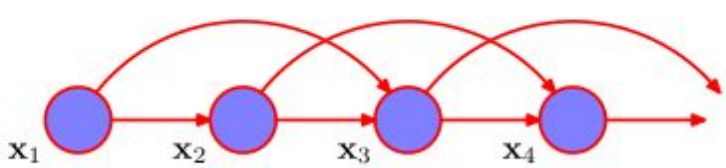
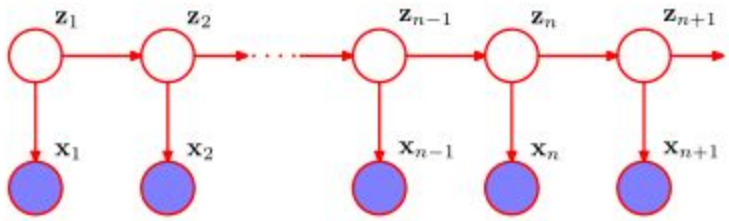


Figure 13.5 We can represent sequential data using a Markov chain of latent variables, with each observation conditioned on the state of the corresponding latent variable. This important graphical structure forms the foundation both for the hidden Markov model and for linear dynamical systems.



extend to the Mth order → the conditional distribution for a particular variable depends on the previous M variables BUT you pay a price, the number of parameters in the model is now much larger

continuous variables → you use linear-Guassian conditional distributions in which each node has a Guassian distribution whose mean is a linear function of its parents

or
use a neural network (tapped delay line)

if the latent variables are discrete/categorical ⇒ hidden Markov model

hidden Markov model = specific instance of the state space in which the latent variables are discrete

speech recognition, natural language modelling, on-line handwriting recognition, analysis of biological sequences such as proteins and DNA

Transition diagram showing a model whose latent variables have three possible states corresponding to the three boxes. The black lines denote the elements of the transition matrix A_{jk}.

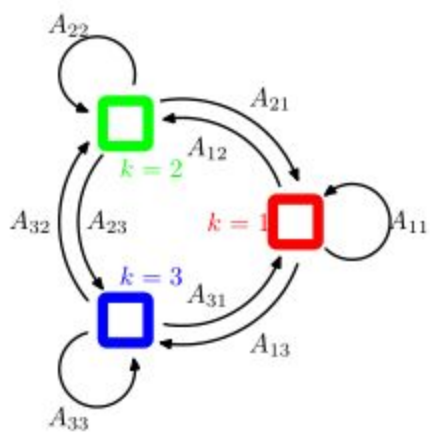
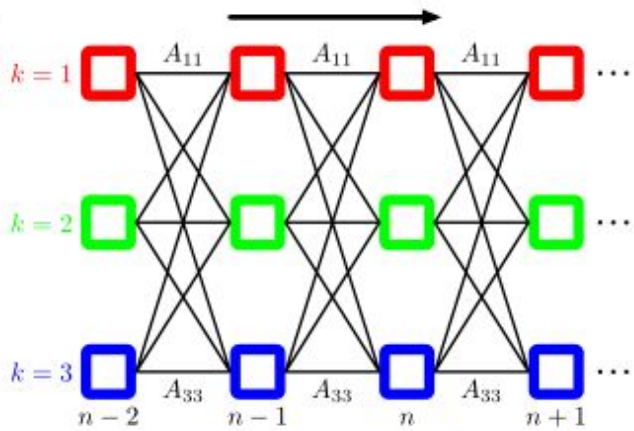


Figure 13.7 If we unfold the state transition diagram of Figure 13.6 over time, we obtain a lattice, or trellis, representation of the latent states. Each column of this diagram corresponds to one of the latent variables $\mathbf{z}_n$.



Bayes theorm
is just that the probability of a and b occuring (p(a,b))
= the probability of a x the probability of b given a has occurred (p(a) x p(b|a))

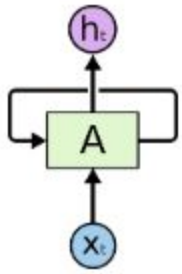
so you can do
p(x1, x2, x3….) = p(x1) x p(x2|x1, x2, x3, …)

LSTM Networks

Traditional neural networks → can't use reasoning about previous events to inform later ones

Recurrent neural network

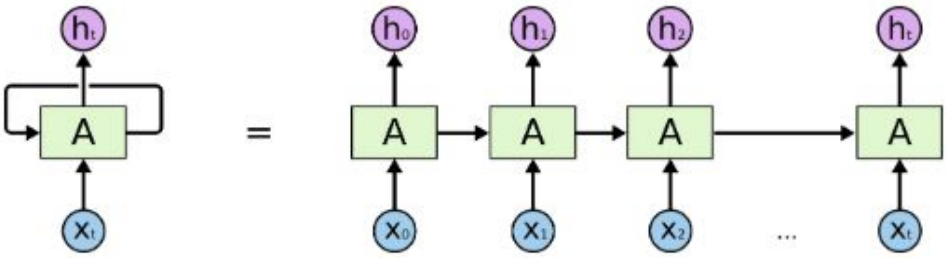
- networks with loops in them, allowing information to persist



Recurrent Neural Networks have loops.

input x, output h,

= multiple copies of the same network, each passing a message to a successor



An unrolled recurrent neural network.

connect previous information to the present task, previous video frames to inform the understanding of the present frame

“The clouds are in the sky”

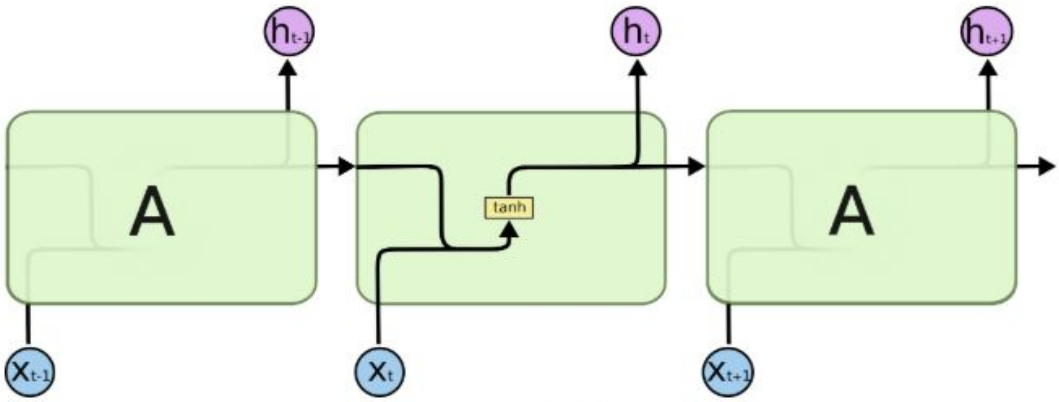
“I grew up in France… I speak fluent *French*”

gap between the relevant information, and the point where it is need, cab become very large (= “long-term dependency”, RNN unable to connect the information)

Long Short Term Memory networks (LSTMs) = special kind of RNN, capable of learning long-term dependencies

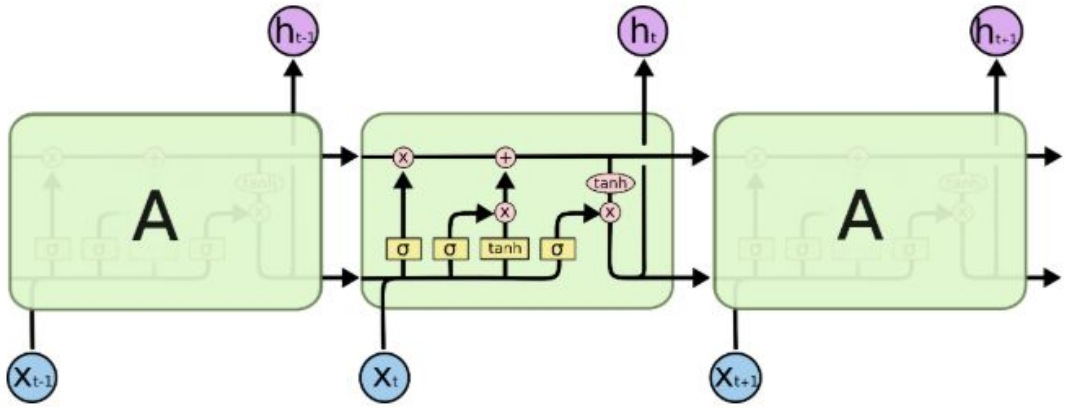
explicitly designed to avoid the long-term dependency problem
remember information for long periods of time

standard RNN = chain of repeating modules of neural network (with a simple structure, like a single tanh layer)

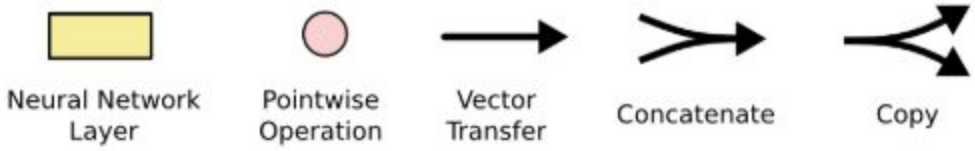


The repeating module in a standard RNN contains a single layer.

LSTM = also chain structure, but the repeating module has a different structure → instead of having a single neural network layer, there are four, interacting in a special way)

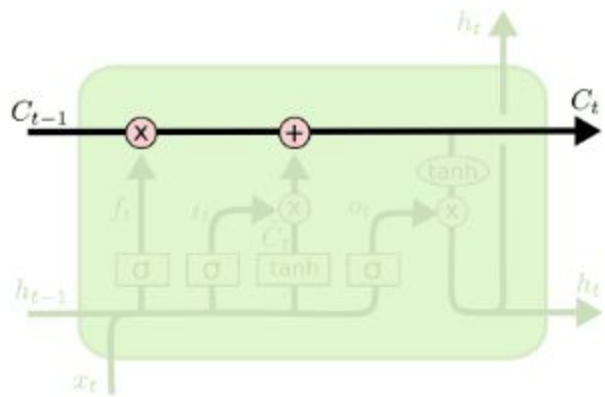


The repeating module in an LSTM contains four interacting layers.

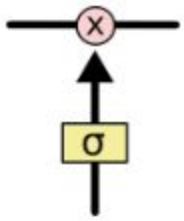


vector = output from one node/neuron to the inputs of others
pointwise operation = vector addition…
yellow box = learned neural network layers

cell state = horizontal line running through the top of the diagram, conveyor belt that runs straight down the entire chain, with only some minor linear interactions → information flows along it mostly unchanged

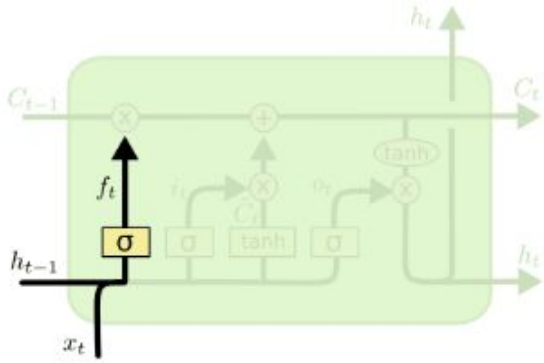


LSTM can add or remove information to the cell state, carefully regulated by “gates”
“gates” = sigmoid neural net layer, + a pointwise multiplication operation → sigmoid layer outputs numbers between 0 and 1 (i.e. how much of each component should be let through, 0 = let nothing though, 1 = let everything through) → optionally let information through



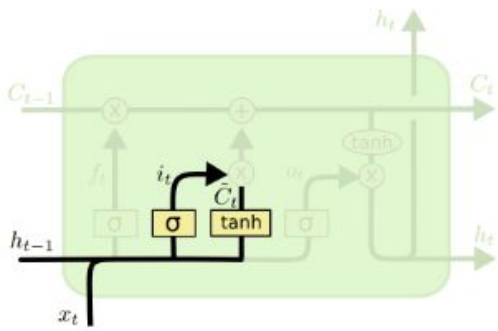
the LSTM will have 3 gates, PROTECT AND CONTROL THE CELL STATE

1. decide what information to throw away from the cell state =
    a. done by the “forget gate layer” (sigmoid layer, i.e. has a sigmoid activation function)→ looks at input x_t and previous output h_{t-1}, outputs a value between 0 and 1 *for each number in the cell state* (C_{t-1}). 1 = “completely keep this”, 0 = “completely get rid of this”
    b. e.g. language model, trying to predict the next word based on all the previous ones. Cell state might include the gender of the present subject (so that you use the correct pronouns, adjectives agree properly in some languages). When you see a new subject, you want to forget the gender of the old subject.



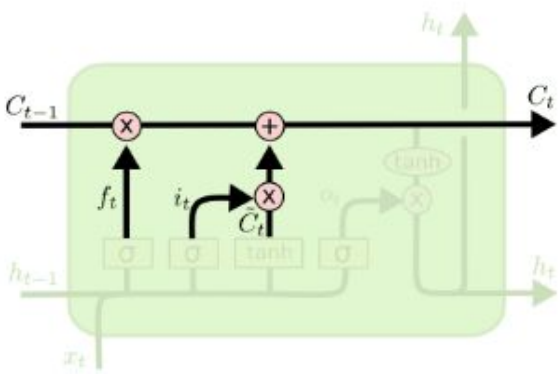
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \ + \ b_f\right)$$

2. Next: decide what new information to store in the cell state
    a. first, tanh layer creates a vector of new candidate values (C_t) to add to the state
        i. e.g. adding the gender of the new subject to the cells tate, to replace the old one



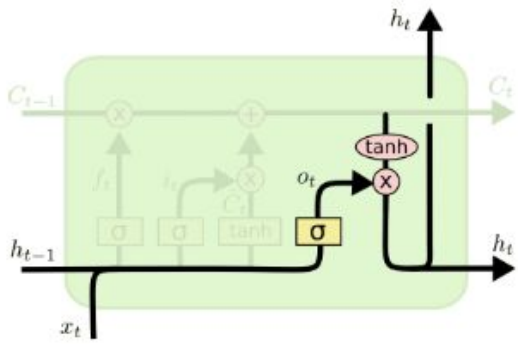
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

    b. second, update the old cell state, C_{t-1}, into the new cell state C_t → multiply the old state by f_t, forgetting the things we decided to forget earlier, then add i_t x C_t (the new candidate values, scaled by how much you decided to update each state value)
        i. e.g. in the language model, actually dropping the information about the old subject's gender and adding the new information (as you decided in step 1)



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

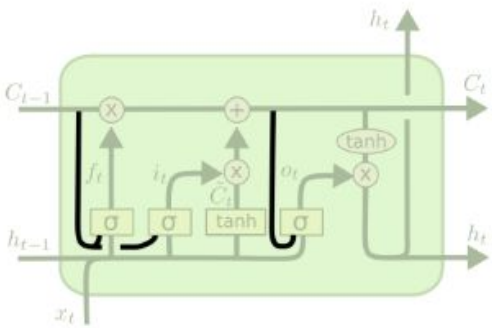
3. Finally: decide what you're going to output. Output is *based* on the cell state, but it is a filtered version.
    a. First: run a sigmoid layer that decides what parts of the cell state you're going to output
    b. Then: put the cell state through tanh (to push the values to be between - 1 and 1), multiply it by the output of the sigmoid gate, so that you only output the parts you decided on
    c. e.g. for the language model example: since you've seen a new subject, might want to output information relevant to a verb, in case that's what is coming next (e.g. whether the subject is singular or plural, so that you know what form a verb should be conjugated into if that's what follows next)



$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

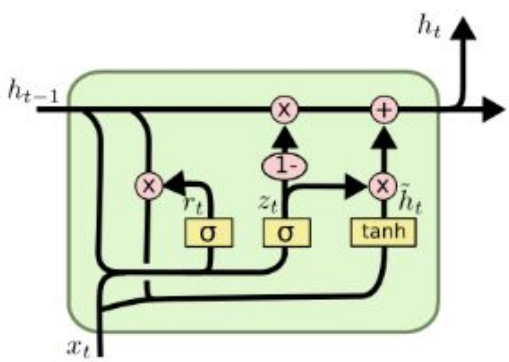
Variants of LSTM

Gers and Schmidhuber (200)
adding peepholed connections
= let the gate layers look at the cell state



$$f_t = \sigma\left(W_f \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \ + \ b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \ + \ b_i\right)$$
$$o_t = \sigma\left(W_o \cdot [\boldsymbol{C_t}, h_{t-1}, x_t] \ + \ b_o\right)$$

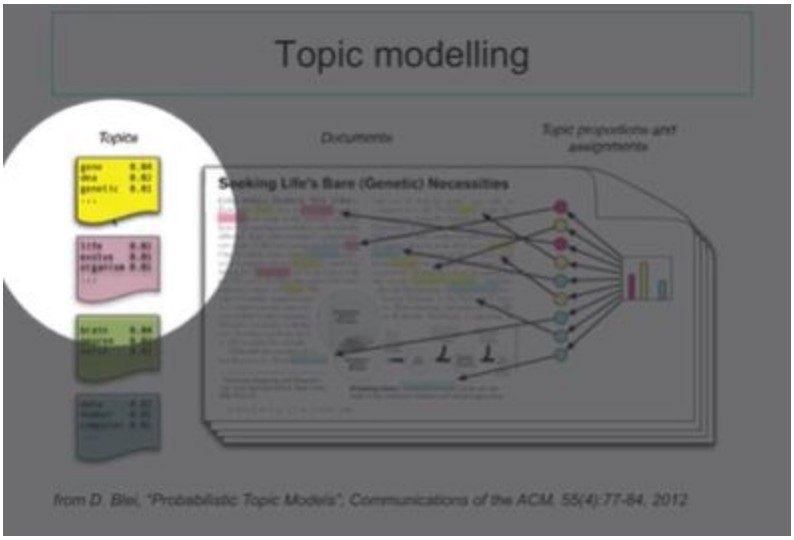
Gated recurrent Unit (Cho et al 2014)
combines forget and input gates into a single update gate
merges cell state and hidden state



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Natural Language Processing (Chapter 9)

To be able to understand, interpret, analyse, talk back - generate language

- Named-entity recognition
    - aims to detect all references in a text to entities that have a name (people, organisations, events…)
    - very mature technology
    - form of text tagging (can be tagged with colours, so that a human can quickly understand and review)
    - Can form the first step for high level tasks (chat bots…)
- Sentiment analysis
    - aims to classify the “sentiment” of a particular text (tweet, email)
    - this can be done by looking/recognising at specific keywords (together = the main polarity of the test)
    - trend these days is to take the whole text, turn it into a numerical representation (a single vector) so that you can use standard classifiers
- Topic modelling
    - do typically with a collection of documents
    - want to understand if these documents revolve around a certain number of topics
    - you also want to understand each topic, which proportion they represent
    - statistical/probablistic model - observed the documents themselves, the indexes, but topics = latent variables that have to be inferred
    - *latent Dirichlet allocation* → proposed by Blei, Ng, Jordan in 2003 = very highly cited
    - 

        - distributions over words
        - once you've discovered the topics → able to infer
- Machine translation
    - most successful application
    - very solid application that's already being used
    - available for many languages
    - can always use a pivot language even if there's no direct translation
    - google translate can translate between 183 different languages
- Dialogue systems
    - extremely important nowadays
    - probably the most challenging task
    - divided into two types
        - task-oriented dialogue systems: short, targeted conversations to extract extra info from a user needed to complete a task
        - chatbots: extended, engaging human-like conversations
            - Izzy = app for managing period pain
                - just becomes another one of your contacts
    - includes *question-answering systems:* "Who was the king of Spain in 1492?"; "What is the product of 2 and 3.5?"

(Many others: automatic summarization, textual entailment recognition, relationship extraction, wikification…)

Document Vectors

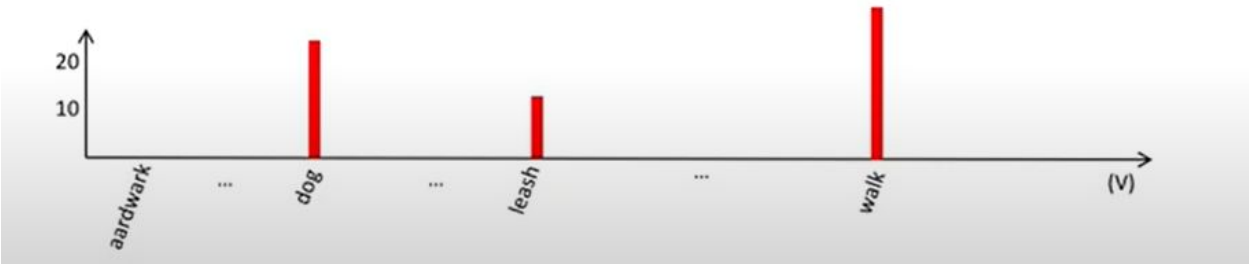
document vector = *numerical vector for representing a whole document for various purposes*

first need to define your vocabulary (let's say of V distinct words)

idea of the document vector is to represent the entire document in a single vector representation

first thing you think of:

GIVEN A VOCABULARY OF SIZE V, A BASIC DOCUMENT VECTOR = just counts of how often each word in the vocabulary appears in the document



= term frequency, *a histogram with V bins*
often normalized by dividing it by the length of the document - if you want you can normalise these counts by simply dividing by the length of the document

- tend to be dominated by words that are not very informative “I”, “to”, “a”
    - “the” appears most frequently in Shakespeare’s play is “the” (28, 944)
- main goal is to maybe reweight the term frequencies to be able to discount or appropriately proportion the words based on their meaningfulness or importance/remove the words altogether from the counts (in preprocessing steps called stop words elimination)
- but you want to have something more general that can actually do that for every word in your vocabulary
- common way to do that is to take the entire document collection, look at the words that appear in the documents, and if a word appears every document it is “uninformative’ and given a weight of 0
    - because the word “the” appears in all plays, its idf value is 0
- if the word appears in fewer documents and is at the same is frequent in those documents, it is likely instead to give a higher informative value it is given a higher weight
    - the most important word = the protagonist of each play (Prospero in the The Tempest) - very frequent in the document and doesn’t appear at all in the rest of the collection
- = *inverse document frequency = become the weight by which the term frequencies are reweighted*
- so together the product of these two terms (term frequency x inverse document frequency) is called **tf-idf**

Medicine:

- Application to clinical notes → attempt to classify them into medical subdomains
- UMLS concepts → large dictionary of medical terms

- in both these things, the computation of tf-idf can be done off the shelf with Apache Ctakes

Word embeddings

When you want to change the focus from documents to single words → for tasks when you're required not to look at the document at the whole, but the individual words (keyword identification, parsing,)

"distributional word representations"/word embeddings

conversion of words to vectors = word embedding

= *learned representation for text where words that have the same meaning have a similar representation.*

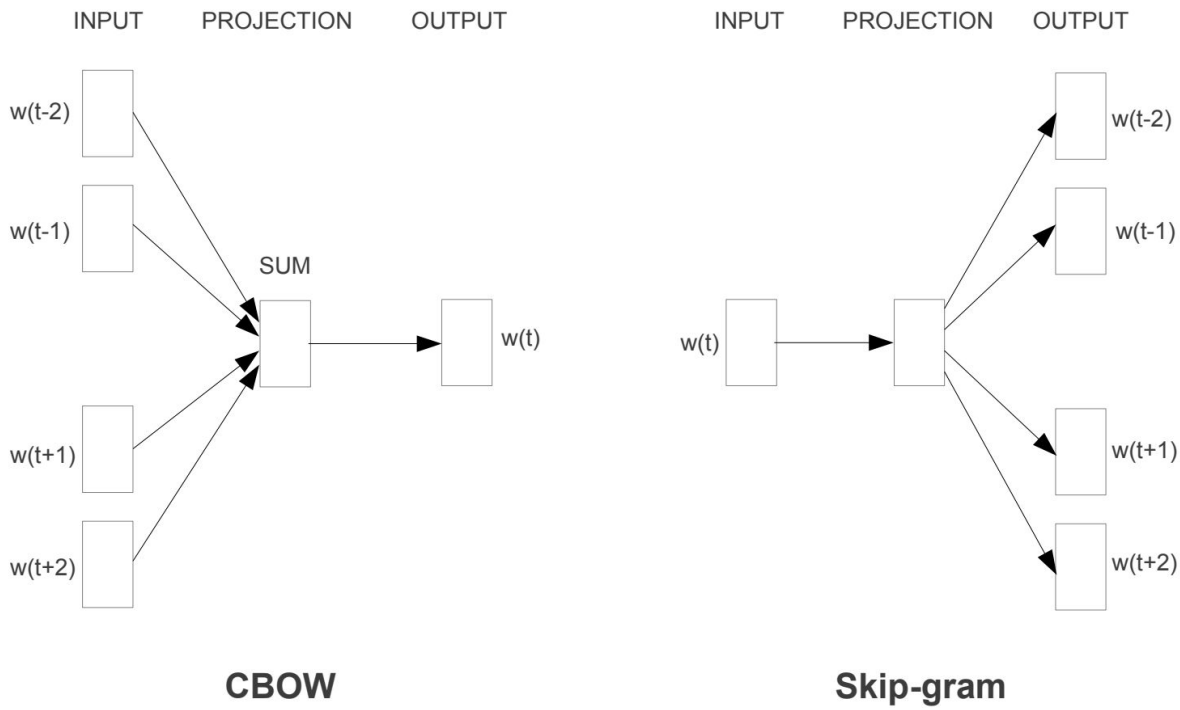
- class of techniques where individual words are represented as real-valued vectors in a predefined vector space
- each word is mapped to one vector and the vector values are learned in a way that resembles a neural network
- (dense and low-dimensional vector) → as opposed to the thousands or millions of dimensions required for sparse word representations (one-hot encoding) / different to bag of words model where different words have different representations regardless of how they are used
- dense distributed representation for each word
- each word is represented by a real-valued vector, often tens or hundreds of dimensions
- *associate with each word in the vocabulary a distributed word feature vector - the feature vector represents different aspects of the word: each word is associated with a point in a vector space: the number of features is much smaller than the size of the vocabulary*
- distributed representation is learned based on the usage of words → result is that words that are used in similar ways have similar representations (naturally capturing their meaning)
- *distributional hypothesis* (Zellig Harris, 1956) → words that have similar context will have similar structure/ let the usage of the word define its meaning!

LEARN A REAL-VALUED VECTOR REPRESENTATION FOR A PREDEFINED FIXED SIZE VOCABULARY FROM A CORPUS OF TEXT

"learn a word embedding from text data"

supervised classification problem → learned jointly with neural network model OR unsupervised (document statistics)

1. embedding layer
    a. word embedding learned jointly with a neural network model on a specific task (language modelling, document classification)
    b. document text cleaned and each word one-hot encoded
    c. size of the vector space is specified as part of the model (50, 100, 300 dimensions)
    d. vectors are initialized with small random numbers
    e. embedding layer is used at the start of the neural network
    f. fit in a supervised way using the backpropogation algorithm
    g. inputs → symbolic categorical features, associate each possible feature value (= word in the vocabulary) with a d-dimensional vector for some d → vectors are then considered parameters of the model, and are trained jointly with the other parameters
    h. one-hot encoded words are mapped to the word vectors
    i. multilayer Perceptron model → word vectors are concatenated before being fed as input to the model
    j. recurrent neural network → each word may be taken as one input in a sequence
    k. requires a lot of training data, can be slow, but will learn an embedding both targeted to the specific text data and the NLP task
2. Word2Vec
    a. standalone word embeddings from text corpus (2013, Google, standard for pre-trained word embedding)
    b. if you do vector math with the learned vectors → “King” - “man-ness” + “women-ness” = vector similar to “Queen”
    c. good at capturing syntactic and semantic regularities in language,
    d. each relationship is characterized by a relation-specific vector offset
    e. Continuous Bag-of-Words (CBOW) model
        i. learns the embedding by predicting the current word based on its context
    f. Continuous Skip-Gram model
        i. learns by predicting the surrounding words given a current word
    g. both → learning about words given their local usage context (= the window of neighbouring words) → size of the “sliding window” = tunable hyperparameter! (large windows = more topical similarities, smaller windows = more functional and syntactic similarities)
    h. advantage = high-quality word embeddings can be learned efficiently, allowing larger embeddings to be learned from much larger text corpora



3. GloVe
    a. extension to word2vec (more efficient, Pennington, Stanford)
    b. classical vector space model representations of words used matrix factorisation techniques (Latent Semantic Analysis)
    c. marries global statistics of matrix factorisation techniques (LsA) with local context-based learning in word2vec
    d. constructs an explicit word-context or word co-occurrence matrix using statistics across the whole text corpus
    e. global logbilinear regression model for the unsupervised learning of word representations that outperforms other models (generally better word embeddings)

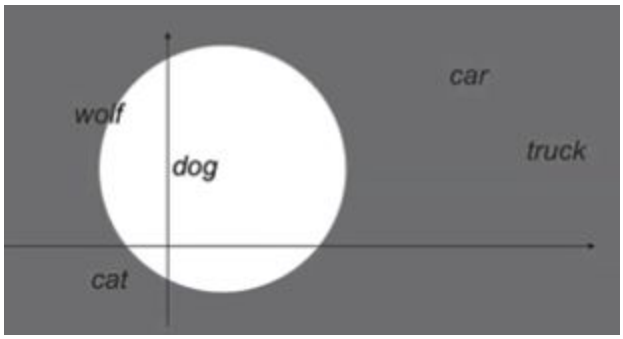
need to look at numerical representations not of the document but of the individual word itself
individual words need to be represented by a set of features (a feature vector)

In the past, for many years, these were simple binary features (capitalised? end with an s? does it contain digits? long word? does it have a suffix?)
= a thing of the past

from the last few years, people have moved from binary representations into

every distinct word in our vocabulary is mapped to a point in a what is typically a high dimensional space (50 to 100 dimensions typically)

mapping is not random
mapping is done under an expectation that words that are semantically similar to each other will also be close in that vector space



not difficult to compute
require large amounts of text to be computed reliably
such large amounts of text are easy to find these days (wikipedia etc) → available in digital form, don't require any annotation

The principle here to create a vector for a specific word is to look at the frequency of the words that appear in the context of that word
→ GloVe, word2vec, a few more

what does that look like?

trees and pines → obvious that pines are a subcategory of trees
words will likely be used in text that describes natural scenery (words such as mountains, rivers, plants) → likely to appear in the proximity, neighbourhood, around the word that we are trying to represent numerically.

For each word, the counts of its surrounding words form the *co-occurrence matrix*

- Example corpus:
  - I like deep learning.
  - I like NLP.
  - I enjoy flying.

counts	I	like	enjoy	deep	learning	NLP	flying	.
I	0	2	1	0	0	0	0	0
like	2	0	0	1	0	1	0	0
enjoy	1	0	0	0	0	0	1	0
deep	0	1	0	0	1	0	0	0
learning	0	0	0	1	0	0	0	1
NLP	0	1	0	0	0	0	0	1
flying	0	0	1	0	0	0	0	1
.	0	0	0	0	1	1	1	0

Those three sentences on the left, we are trying to compute co-occurrences for (entire set of text is on the left)

take the words in our vocabulary, put them here, organised by rows
then for each of these words, we look at the words in the text, +/- 1

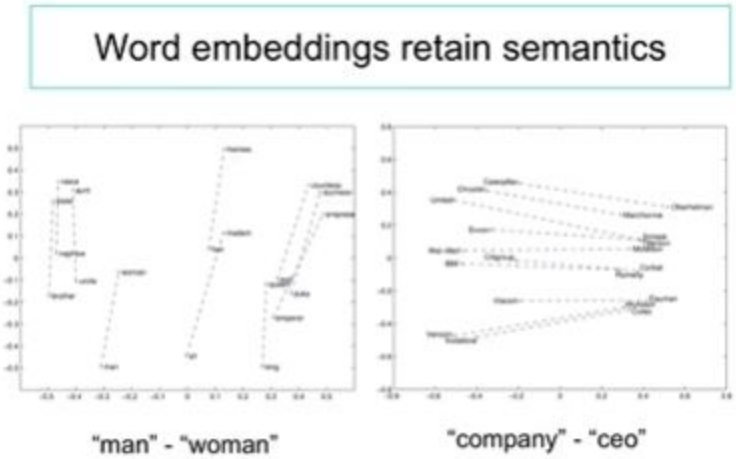
e.g. I appears followed by like 2 times, and enjoy once, so we put these counts in the first row

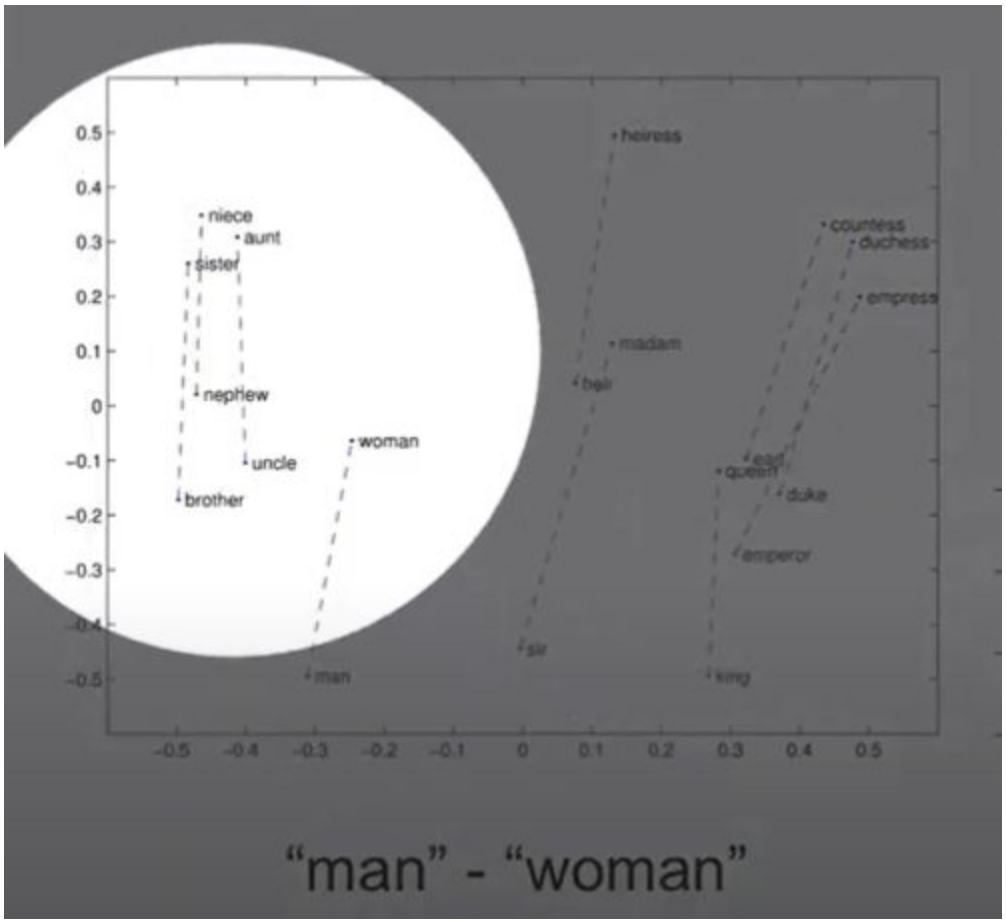
no other words in our vocabulary appear next to it

then we go to the word like (preceded by the word I twice, and followed by the words deep and NLP, so then we go to those words and do those counts)

can see “enjoy” and “like” have fairly similar rows, and those rows can be nothing else but the numerical representation of the at stake

so this idea of the co-occurrence matrix is the founding idea of how we embed words into a numerical representation → can become very big
number of ways to tame that (reduce the number of columns)calculate these embeddings on the fly)





left = famous example, shows that words that exchange only by gender in this caase (the gender attribute) have similar distance between each other
distance between these words is kind of similar in the embedded space, which is an impressive result

another example: the instance between the company name and the company's CEO is very similar
multi-dimensional scaling, t-SNE

word-embeddings are useful for every task in natural language processing

question:

What justifies that the embedding of the word 'dog' should be closer to that of 'cat' than that of 'truck'?

distributional structure (occurrence of parts relative to other parts) → sum of all its environments or (co-occurrents) → elements which the word yields an utterance

parts of a language do not occur arbitrarily relative to each other
each element occurs in certain positions relative to certain other elements
chose members of those classes that regularly occur together, in the order in which these classes occur

all elements in a language can be grouped into classes whose relative occurrence can be stated exactly

Zellig Harris, 1954 → distributional semantics → laid out the theoretical foundations for word embeddings

In a large sample of text/text corpus, similar words will have similar distributions

Named-entity recognition

named entity = *an entity in a text with a specific name (person, location, event etc)*

often capitalised but not always

the current trend to performed named entity recognition is to classify one by one individual words (convert the word into some form of numerical representation, pass that representation to a classifier → get some classifications (entity, not an-entity)) , outside
for the entity type → people location and the like
but in addition to this you have to provide labels for when you're at the beginning of an entity

have to cement the beginning of each entitiy to cement

- The current trend is to tackle named-entity recognition (NER) as **classification of individual words**. Each word has to be assigned to the correct *segmentation prefix* (**B**egin, **I**nside and **O**utside) and *class* (aka "type"). Example:

We	visited	Lemington	Spa	and	Stratford-upon-Avon
Outside	Outside	B-Location	I-Location	Outside	B-Location

very good and simple baseline is to make use of the word embeddings that we just described
convert each word in the text to its embedding
common practice -> to not just consider just the embedding of the word, but also the embeddings of the words just before and after (= in order to capture a little bit of context around that word) → often the context is able to improve the accuracy of the context

example: take the word at stake (w_t = tf in text), and also concatenate with its embeddings the embeddings of the words that come just before and just after

assume the word is XYZ, and you've never met it during the training stage of our named entity recogniser → Paris but we've never met it before, even during the embeddings (outside of our vocabular) → have to flag it as an unknown word)

I visited… last year = suggest that that XYZ is a location
we would be able by forming that vector around word XYZ with embeddings of that word that often belong and complement a location name, if we're lucky this embedding overall will still fall in the correct boundary of the classification and we will still be able to classify that word as B location if we've never met it in the training set (or even word in the word-embedding stage)

so after forming feature vector we use a very good classifier (like SVM, recurrent neural networks …)

Evaluation has to be performed with dedicated measures → first thing you have to keep in mind, entities are kind of sparse in the text (so you cannot use accuracy)

use something similar to the f1 score, dedicated f1 scores that take into account the fact that entities can be multi words, = MUC-7 or CoNLL (names of conferences in the field)

- Convert each word to its embedding
- As feature for the word, concatenate its embedding and those of a few words before and after (*n-gram*);
  e.g., $[w_{t-2}\ w_{t-1}\ w_t\ w_{t+1}\ w_{t+2}]$

  In this way, word XYZ in a sentence like "I visited XYZ last year" may be correctly classified as *B-Location* even if it was never encountered in the training set
- Use a good classifier (SVM, CRF, RNNs…)
- Evaluate with an appropriate measure (MUC-7, CoNLL…)

specialised example of Named Entity Recognition:

clinical named entity recognition

in clinical named entity recognition the netities aren't people or locations, they are names of drugs, illnesses, morbidities, names of treatments

we need to be able to parse the text, and recognise each occurance, of each of these types of entitites

fortunately we have lots of data collected, in many cases publically available → DDI (collected in Madrid), the i2b2 datasets (which are many and varied)
we can use them to test our approach, compare them with what has been done with other researchers before us
2 main/immediate challenges → first of all, the entities are sparse
there are just a few entities, and a lot of text around them
and that makes a dataset that is naturally highly imbalanced

second challenge → typical performance metrics (CoNLL) → expect us to identify the entire entity that is exactly correct → for a true positive in the final

e.g. “recently diagnosed abdominal carcinomatosis” must completely be recognised correctly to be counted as a True Positive in the performance measure (to get a brownie point)

Specialised NER: example

- **Drug name recognition** is a specialised NER to recognise entities of type *drug*, *drug_n* (i.e., not approved), *group*, *brand*
- **Clinical concept extraction** is a specialised NER to recognise *problem*, *test*, *treatment* entities
- Popular datasets: DDI-DrugBank, DDI-MedLine, 2010 i2b2/VA. Challenges:
  - highly imbalanced (entities are sparse)
  - the CoNLL score is very strict (e.g., “recently diagnosed abdominal carcinomatosis” must all be recognised correctly to be counted as a True Positive)

use instead a state of the art classifier: bidirection LSTM-CRF
consists of two separate classifiers: LSTM (long-shert term memory) → variation of recurrrent neural network (sequential classifier)
CRF = conditional random field, well-known sequential classifier (advantage over LSTM) → uses the Viterbi algorithm → can decode a whole sentence in one go

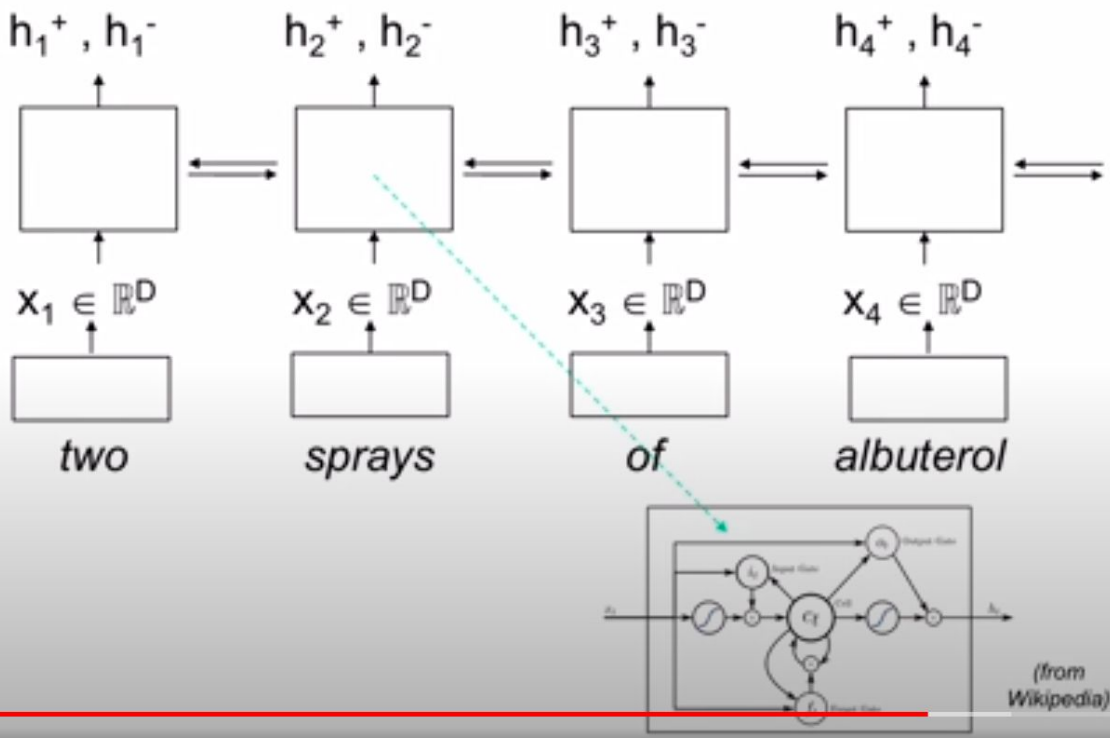
two techniques complement each other very nicely → for that reason, they’ve been adopted jointly into these LSTM-CRF framework which we have used for clinical named entity recognition

Clinical NER with the BiLSTM-CRF

- The highest accuracy in named-entity recognition (NER) is held by the **bidirectional LSTM-CRF** and its variants (Lample *et al*. 2016)
- **LSTM** (long short-term memory): a performing *recurrent neural network*, i.e., a sequential classifier
- **CRF**: another sequential classifier that can provide globally-optimal predictions using the Viterbi algorithm; it can improve the prediction of the LSTM
- We have recently used it for *clinical NER*

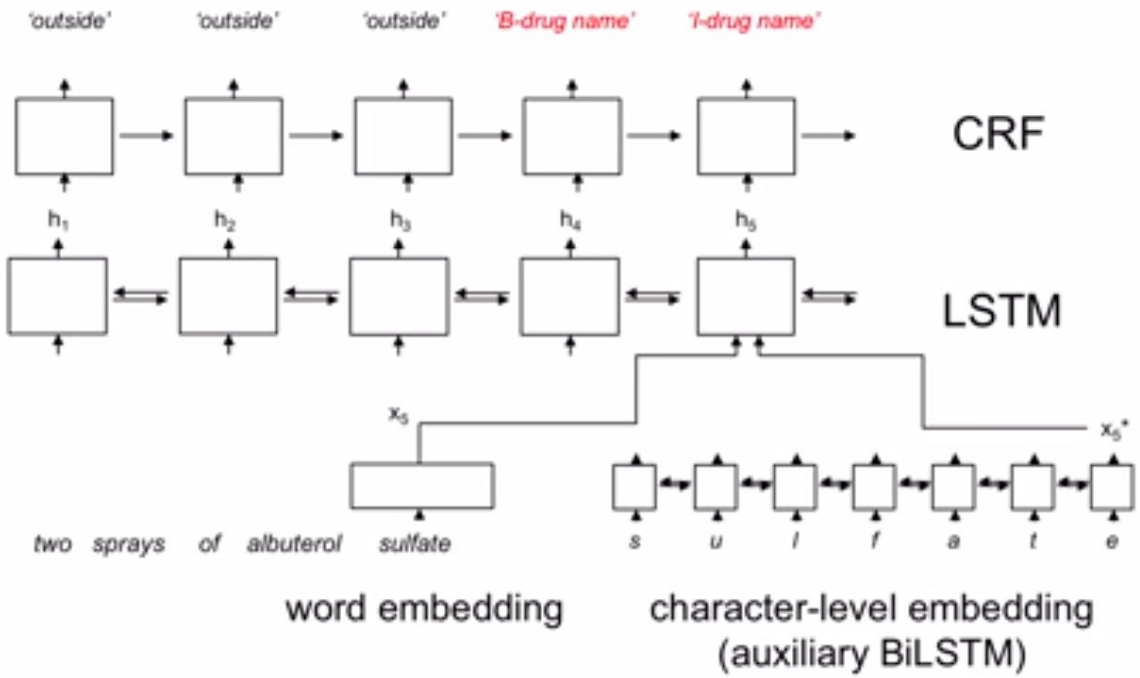
remember, the LSTM is a recurrent neural network that uses repeatedly a cell that has the schematic, contains three gates that change the store, (buliding block of “recurrent neural network”)
each word is converted to its corresponding embedding
and then the numerical representation goes into this cell
produces an output to the next stage and so on and so on → until the exhaustion of the sentence
no reason to do this in the opposite direction (left-to right, right-to-left), concatenate the outputs = bidirectional LSTM

The bidirectional LSTM



(from Wikipedia)

The bidirectional LSTM-CRF for NER



- Drug name recognition:

Model	DDI-DrugBank	DDI-MedLine
	F1 score (%)	F1 score (%)
WBI-NER	87.80	58.10
Hybrid-DDI	80.00	37.00
Word2Vec+DINTO	75.00	57.00
BiLSTM-CRF (cc/mimic)[1]	**88.38**	**60.66**

a long way to go to improve these approaches (lack of word-embeddings, vocabulary is very big)

Topic modelling

topic modelling is about finding topics in a collection of documents
large, typically becoming larger and largers these days

can look at these documents 1 by 1 and calculate the tf-idf for each one
and stack them together → in what people call utter document matrix (the stacked representation/document vector for each of the documents in the vector, and is our starting point)

Is it possible to find a topic set shared by this collection of documents that can allow us to reduce significantly the degrees of freedom in the “document matrix” and still be able to describe the documents effectively and the frequencies/occurances of the words in the documents by means of this intermediate layer of topics

in more precise probablistic words: if I have this stacked matrix that represents the probability of the word in each document in the collection
can i factorise that into the product of two terms: the probability of the word given the topic, and the probability of the topic given the document, in the marginalised topic, and be able to achieve an approximation of this

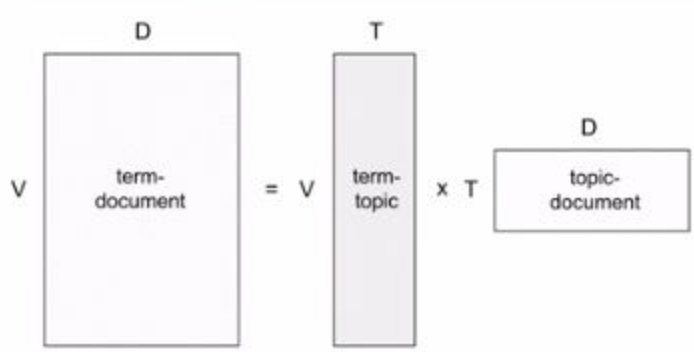
Topic modelling

- Is it possible to identify a set of **shared topics** in the document collection (T < D)?
- Can the probability of the terms be well modelled by these topics?
- In other words, can we effectively break down probability p(w|d) over two factors, p(w|t) and p(t|d), where t is a latent variable? As in:

$$p(w \mid d) = \sum_t p(w \mid t)\, p(t \mid d)$$

may be easier to be understood if we also look at this as a matrix representation:

Topic modelling as matrix factorisation



dimensions of the document matrix:

document number width, the number of documents = D
the size of our vocabulary (could be typically 100 000, 200 000) = V

- the vocabulary of specialised medical language can reach 5 million distinct words

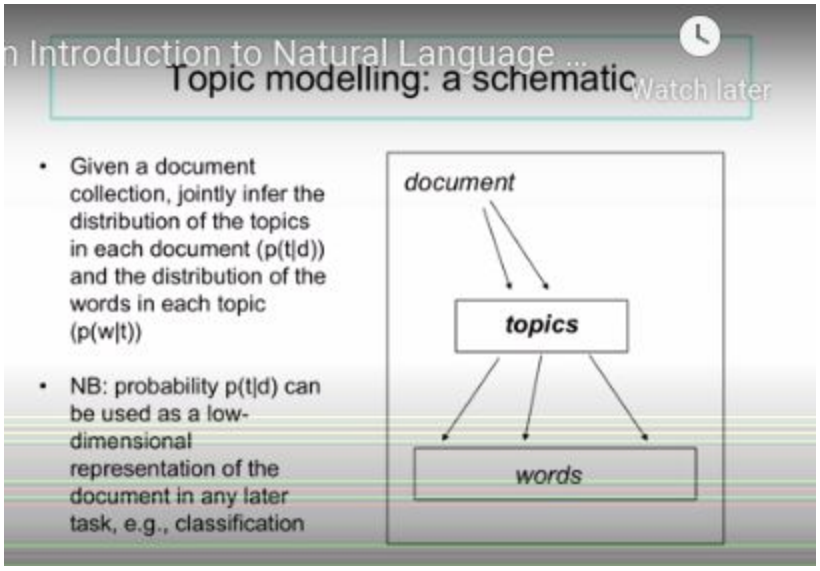
goal is to find a factorisation that will approximate that by far fewer degrees of freedom
term topic matrix

- for each word in the vocabulary, you would have for every topic a distribution of words
- e.g. how likely in this topic, a subfield of medicine, words typically occur

and then for each individual document = the distribution of topics

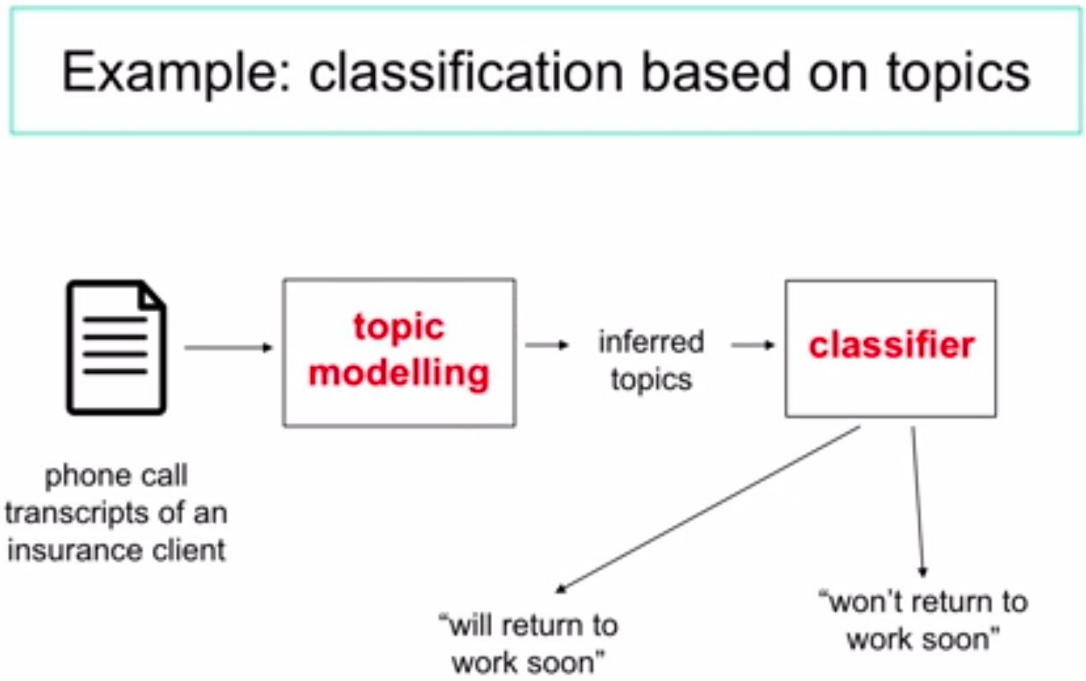
the product of these two matrixes (term topic x topic-document) will give an approximation of the document matrix

trying to make this approximation as closely as possible.



1st → topics distribution
2nd → distribution of topics within documents. Can be used for classification
taking the documents, finding for each document its set of topics → can become the feature vector that you use for classifying the document itself

example: transcript of phone calls being client and insurance company
for each of these phone calls we have extracted these topics



the model we have described: is called *probablistic latent semantic indexing* → as you can see, from the date 1999, these techniques are twenty years old.

an improvement over this, the most widely used technique for topic modelling → = *Latent Dirichlet allocation (LDA)*

- adds a probabilistic modeling of the distributions' parameters which brings a number of advantages/ just adds a number of other terms to the equation, to the model, so higher probabilities in the distribution of the parameters, the distribution of the topics, the distribution of the words in the topic
- widely used
- available in every package or language used in data science: R, scikit-learn, gensim

the number of the topics is typically heuristic: typically something you decide with typical hyperparameter selection approaches

CS224N → Stanford, offered for free !!! Most popular MOOC in this field!!!!