Overview of Statistical Learning Methods

Steve Jeffrey, 2018/9

Introduction

Given the task of classifying an object onto two classes 0 and 1, the Bayes decision rule can be used if the prior probabilities P(0) and P(1) are known, and the conditional probabilities P(**x**|0) and P(**x**|1) are also known. In this situation:

$$\mathrm{P}(0|\mathbf{x}) = \frac{\mathrm{P}(0)\,.\,\mathrm{P}(\mathbf{x}|0)}{\mathrm{P}(\mathbf{x})}$$

$$\mathrm{P}(1|\mathbf{x}) = \frac{\mathrm{P}(1).\,\mathrm{P}(\mathbf{x}|1)}{\mathrm{P}(\mathbf{x})}.$$

Given a discrete-valued feature vector **x**, the object is classified as class 0 if P(0|**x**) > P(1|**x**) (and vice versa). In the continuum case P(**x**) is zero, so probability densities are used:

$$\mathrm{P}(0|\mathbf{x}) = \frac{\mathrm{P}(0)\,.\,p(\mathbf{x}|0)}{p(\mathbf{x})}$$

$$\mathrm{P}(1|\mathbf{x}) = \frac{\mathrm{P}(1).\,p(\mathbf{x}|1)}{p(\mathbf{x})}.$$

The probability that the feature vector (now taking on real values) for an object of class 0 falls in a given volume *V* is:

$$P(\mathbf{x} \in V|0) = \int_V p(\mathbf{z}|0)\, d\mathbf{z}$$

and similarly for P(**x** ∈ *V*|1).

In general the prior and conditional probabilities (or densities) are not known. If the training set is large compared to the dimensionality of the problem, it may be possible to estimate the probabilities from the data. However in general this is not feasible and consequently methods must be adopted for learning decision rules from the data. This document summarises a subset of such methods.

Learning criteria

In the classification problem the minimum error R^* is obtained by using the Bayes' rule. However this approach is not possible in most cases because the conditional distributions $P(\boldsymbol{x}|0)$ and $P(\boldsymbol{x}|1)$ (or densities $p(\boldsymbol{x}|0)$ and $p(\boldsymbol{x}|1)$) are unknown. If we are restricted to using decision rules from a given class C, the optimal error rate that can be achieved is:

$$R_C^* = \min_{c \in C} R(c)$$

where $R_C^* \geq R^*$. We should also note that we cannot compute the error rates because the distributions are unknown.

Given a finite amount of data, it may not be possible to find the best rule in C, so the goal is reduced to finding the *approximately optimal* rule. We therefore aim to select an hypothesis h from C such that the error rate is:

$$R(h) \leq R_C^* + \epsilon$$

where ϵ is an accuracy parameter. However as the training data are typically random, we may not always obtain representative data, so we cannot expect to always identify a good hypothesis. The best we can aim for is to identify a good hypothesis (with accuracy ϵ) with high probability:

$$P\{R(h) \leq R_C^* + \epsilon\} \geq 1 - \delta$$

where δ is a confidence parameter. This is the Probably Approximate Correct (PAC) criterion. A class C is PAC learnable if there is some training set of finite size such that $P\{R(h) \leq R_C^* + \epsilon\} \geq 1 - \delta$ for all $\epsilon, \delta > 0$ (regardless of the underlying distributions).

To determine if a class is PAC learnable, we need the following definitions:

- A set of feature vectors $\boldsymbol{x}_i, i = 1, \ldots, n$ is *shattered* by a class of decision rules C if all the 2^npossible labellings can be generated from C.
- The Vapnik-Chervonenkis dimension of a class C is the largest integer Vsuch that some set of V feature vectors is shattered by C.

C is PAC learnable *iff* the VC dimension of C is finite. In this case, theoretical lower and upper bounds on the required sample size can be obtained.

It should be noted there is a trade-off in learning:

- a very rich class C is useful for reducing *approximation error* - the inability of rules in C to approximate the minimal Bayes' error.
- if $C$is very rich, it will be difficult to identify which rule will perform best on future (*i.e.* not included in the training set) examples – the *estimation* error.

The preceding results for classification problems can be adapted to estimation problems. However in this case, decision rules are no longer partitions, but are functions $f(\boldsymbol{x})$ mapping the *d*-dimensional feature vector to a real-value output *i.e.* $\boldsymbol{R}^d \rightarrow \boldsymbol{R}$. The error estimate is also adapted, with the squared error $(y - f(\boldsymbol{x}))^2$ commonly being used because it has a smooth derivative. The best estimator (function) is one which minimizes the expected error. If the dependent variable y is drawn from the conditional distribution $p(y|\boldsymbol{x})$ the expected error is:

$$R(f) = \int_{\boldsymbol{R}^d} R(f, \boldsymbol{x})\, p(\boldsymbol{x}) d\boldsymbol{x} = \int_{\boldsymbol{R}^d} \int_{\boldsymbol{R}} (y - f(\boldsymbol{x}))^2\, p(y|\boldsymbol{x})\, p(\boldsymbol{x})\, dy\, d\boldsymbol{x}$$

The *regression function* is the function we seek – it is the function that minimises the expected error. In a manner analogous to the classification case, this is the Bayes error rate and denoted R^*. In a manner analogous to that used above for the classification problem, we can investigate the requirements for PAC learning. In this case, we have a class of functions $\mathcal{F}$ which map from feature space to desired output *i.e.* $\boldsymbol{R}^d \rightarrow \boldsymbol{R}$. The minimum error error is obtained from the function in $\mathcal{F}$ which minimises the error rate: $R_{\mathcal{F}}^* = \min_{f \in \mathcal{F}} R(f)$. The class $\mathcal{F}$ is PAC learnable if there is some function $h \in \mathcal{F}$ such that:

$$P\{R(h) > R_{\mathcal{F}}^* + \epsilon\} < \delta$$

for all $\epsilon, \delta > 0$ and a training data set of finite size. The class of functions $\mathcal{F}$ is PAC learnable if the pseudo-dimension of $\mathcal{F}$ is finite. The pseudo-dimension is simply the real-valued generalization of the VC-dimension: it is the largest integer V such that some set of V feature vectors is shattered by $\mathcal{F}$.

References

An elementary introduction to statistical learning theory, Kulkarni and Harman, Chs 11-15.

Nearest neighbours

Description

An elementary method which classifies a given feature vector to the same class as the nearest feature vector in the training data. When using the single nearest neighbour it is essentially Voronoi tessellation, but it can be extended to use *k* nearest neighbours. If *k* neighbours are being used to classify a given feature vector $\boldsymbol{x}$, the classes associated with the *k* neighbours are identified and the majority vote taken *i.e.* the input sample is assigned to the most common class appearing in the *k* nearest neighbours. For regression problems using *k* neighbours, the predicted value for the sample could be the mean or median of the neighbours' values.

The expected error rate *R* is:

$$R^* \leq R_\infty \leq 2R^*(1 - R^*)$$

where R^* is the optimal Bayes error rate and R_n is the expected error rate of NN after n training samples. Since R^* lies between 0 and 1, $1 - R^* \leq 1$, so the upper bound is $2R^*$.

There is no universal guide for selecting the optimal number of nearest neighbours. However the error rate approaches the optimal Bayes error rate R^* as $n \to \infty$ if k_n is chosen such that $k_n \to \infty$ and $k_n/\infty \to 0$. One such choice is $k_n = \sqrt{n}$.

No knowledge about the underlying distribution is required as the training data provide the classification rule.

The nearest neighbours algorithm can also be used for regression problems, in which case the error rate is equal to $2R^*$ (in the classification case the upper bound is $2R^*$). The estimated value for a given sample is given by the mean of the *k* nearest neighbours (in the classification case the sample was assigned by taking a majority vote of the labels assigned to the *k* nearest neighbours).

References

An elementary introduction to statistical learning theory, Kulkarni and Harman, Ch 7.

Potential application areas

- Classification for data with discrete labels.
- Regression (or estimation) for data with continuous labels.
- Discrete and continuum feature vectors
- Suited to problems where the data are naturally clustered in groups having the same classification.

Advantages

- Simple, fast
- Completely unbiased *i.e.* no prior assumptions about the underlying data distribution
- Relatively good performance given the simplicity, providing the input sample is close to the nearest neighbour(s)
- Only makes mild assumptions about the data.

Disadvantages

- Prediction error can be relatively high
- May perform poorly in high dimensions
- Results can be unstable.

Example(s)

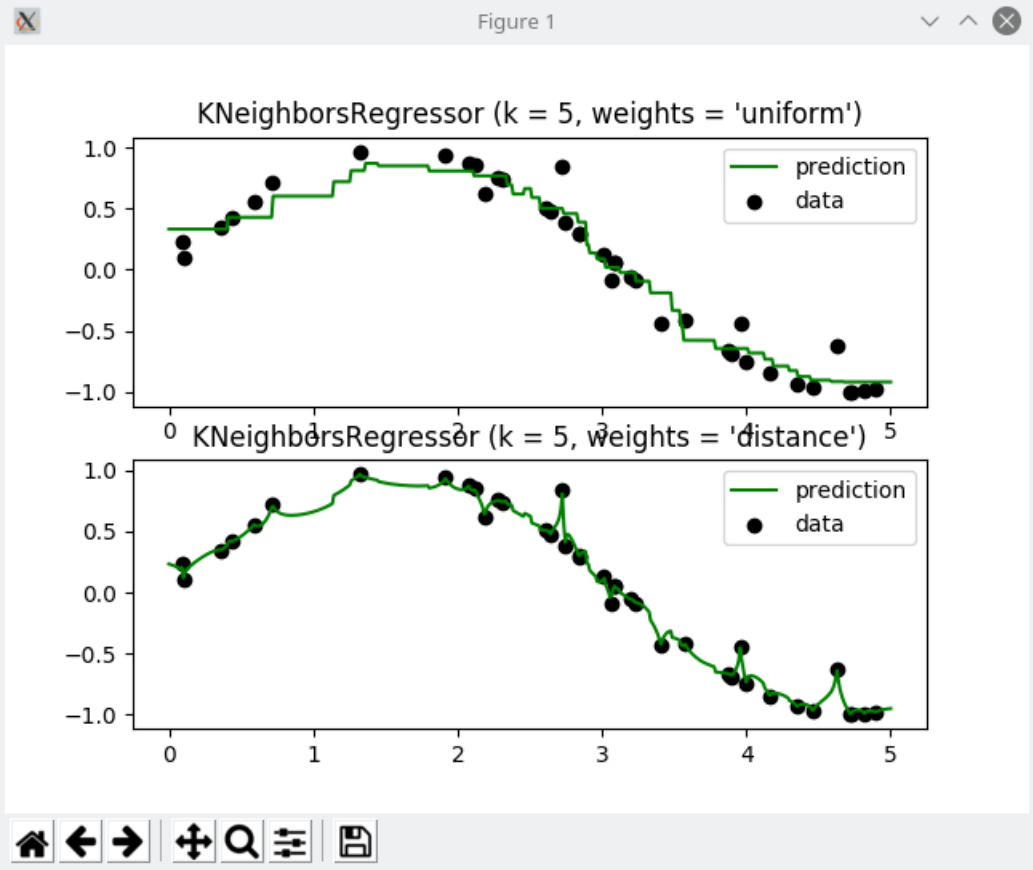
1. Using Python

This example generates a test dataset $y_i = \sin x_i + N$, $i = 1, \ldots, 40$, where *N* is random noise and the x_i are randomly chosen on $[0, 5)$. The scikit-learn *k*-NN algorithm is then used to estimate values for $x_j, j = 1, \ldots, 500$ uniformly distributed on $[0, 5]$.

```
"""
      ============================
      Nearest Neighbors regression
      ============================

      Demonstrate the resolution of a regression problem
      using a k-Nearest Neighbor and the interpolation of the
      target using both barycenter and constant weights.

"""
print(__doc__)

# Author: Alexandre Gramfort <alexandre.gramfort@inria.fr>
#         Fabian Pedregosa <fabian.pedregosa@inria.fr>
# License: BSD 3 clause (C) INRIA

###############################################################################
# Generate sample data
import numpy as np
import matplotlib.pyplot as plt
from sklearn import neighbors

np.random.seed(0)
X = np.sort(5 * np.random.rand(40, 1), axis=0)
T = np.linspace(0, 5, 500)[:, np.newaxis]
y = np.sin(X).ravel()

# Add noise to targets
y[::5] += 1 * (0.5 - np.random.rand(8))

###############################################################################
# Fit regression model
n_neighbors = 5

for i, weights in enumerate(['uniform', 'distance']):
    knn = neighbors.KNeighborsRegressor(n_neighbors,weights=weights)
    y_ = knn.fit(X, y).predict(T)

    plt.subplot(2, 1, i + 1)
    plt.scatter(X, y, c='k', label='data')
    plt.plot(T, y_, c='g', label='prediction')
```

```
    plt.axis('tight')
    plt.legend()
    plt.title("k-NN Results(k=%i, weights='%s')" % (n_neighbors, weights))

plt.show()
```

The output is:



Source: http://scikit-learn.org/stable/auto_examples/neighbors/plot_regression.html

2. Using R

This example classifies a small (*n*=100) cancer dataset as either benign (B) or malignant (M), based on eight features (*d*=8). The test data are split into training (65%) and test (35%) datasets, with the number of nearest neighbours $k_n = \sqrt{n}$ (n is the size of the training subset).

```
# Load the data
setwd(".")

# Imports the required data set and saves it to the prc data frame.
prc <- read.csv("Prostate_Cancer.csv",stringsAsFactors = FALSE)

# Show the structure of the data
str(prc)

# Remove the first variable(id) from the data set
prc <- prc[-1]

# Summarise the "diagnosis_result" variable - shows the number of
# samples under each classificiation (B = benign, M = malignant)
```

```
table(prc$diagnosis_result)

# Rename the classifications: B -> benign, M -> malignant
prc$diagnosis <- factor(prc$diagnosis_result, levels = c("B", "M"), labels
= c("Benign", "Malignant"))

# Summarise the "diagnosis_result" variable, but convert to a percentage,
rounded to 1 decimal place
round(prop.table(table(prc$diagnosis)) * 100, digits = 1)

# Define a function for normalising data
# - it simply scales the data to [0,1]
normalize <- function(x) {
  return ((x - min(x)) / (max(x) - min(x)))
}

# Normalise all variables except variable 1 (which is a class [malignant
or benign], not a numeric value)
prc_n <- as.data.frame(lapply(prc[2:9], normalize))

# Check the normalising by outputting stats around one of the variables
summary(prc_n$radius)

# Split the data in training and test sets
# - the target variable was removed in the normalisation step
prc_train <- prc_n[1:65,]
prc_test <- prc_n[66:100,]
# - target variable, "diagnosis_result"
prc_train_labels <- prc[1:65, 1]
prc_test_labels <- prc[66:100, 1]

# Load the "class" library
# - library has classification algorithms such as k-nearest neighbour,
#   Learning Vector Quantization and Self-Organizing Maps
# - if a library is not installed, it can be installed in R using:
#   install.packages("name_of_library")
library("class")

# Classify the data
# - set the no. of neighbours:
no_neighbours <- round(sqrt( nrow(prc_train) ))
prc_test_pred <- knn(train = prc_train, test = prc_test,cl =
prc_train_labels, k=no_neighbours)

# Load the "gmodels" library (it has tools for model fitting)
library("gmodels")

# Use the model to predict the classes in the training set
CrossTable(x=prc_test_labels, y=prc_test_pred, prop.chisq=FALSE)
```

The output is:

```
   Cell Contents
|-------------------------|
|                       N |
|           N / Row Total |
|           N / Col Total |
|         N / Table Total |
|-------------------------|


Total Observations in Table:  35


                | prc_test_pred
prc_test_labels |         B |         M | Row Total |
----------------|-----------|-----------|-----------|
              B |        10 |         9 |        19 |
                |     0.526 |     0.474 |     0.543 |
                |     0.909 |     0.375 |           |
                |     0.286 |     0.257 |           |
----------------|-----------|-----------|-----------|
              M |         1 |        15 |        16 |
                |     0.062 |     0.938 |     0.457 |
                |     0.091 |     0.625 |           |
                |     0.029 |     0.429 |           |
----------------|-----------|-----------|-----------|
   Column Total |        11 |        24 |        35 |
                |     0.314 |     0.686 |           |
----------------|-----------|-----------|-----------|
```

Note: the number of correct classifications is (true positives + true negatives) / total, which is (10+15) / 35 or 60%.

Source: http://www.analyticsvidhya.com/blog/2015/08/learning-concept-knn-algorithms-programming/

Kernel rules / potential functions / discriminant functions

Description

Kernel rules are a broad class of methods which can be viewed as an extension of the Nearest Neighbours algorithm. Instead of selecting the nearest *k* neighbours when evaluating the target sample, a broad range of functions (or *kernels, potential functions or discriminant functions*) can be used.

The kernel is constructed to select and weight feature vectors $\boldsymbol{x_i}$ that are nearest to the sample having feature vector $\boldsymbol{x}$, based on some distance measure. Given the kernel $K(\cdot)$ and indicator function:

$$I_A = f(\boldsymbol{x}) = \begin{cases} 1, & A\ is\ true \\ 0, & otherwise \end{cases}$$

a feature vector can be classified into either class 0 or class 1 by counting the number of training samples in each class:

$$v_n^0(\boldsymbol{x}) = \sum_{i=0}^{n} I_{\{y_i=0\}} K\left(\frac{\boldsymbol{x} - \boldsymbol{x}_i}{h}\right)$$
$$v_n^1(\boldsymbol{x}) = \sum_{i=0}^{n} I_{\{y_i=1\}} K\left(\frac{\boldsymbol{x} - \boldsymbol{x}_i}{h}\right)$$

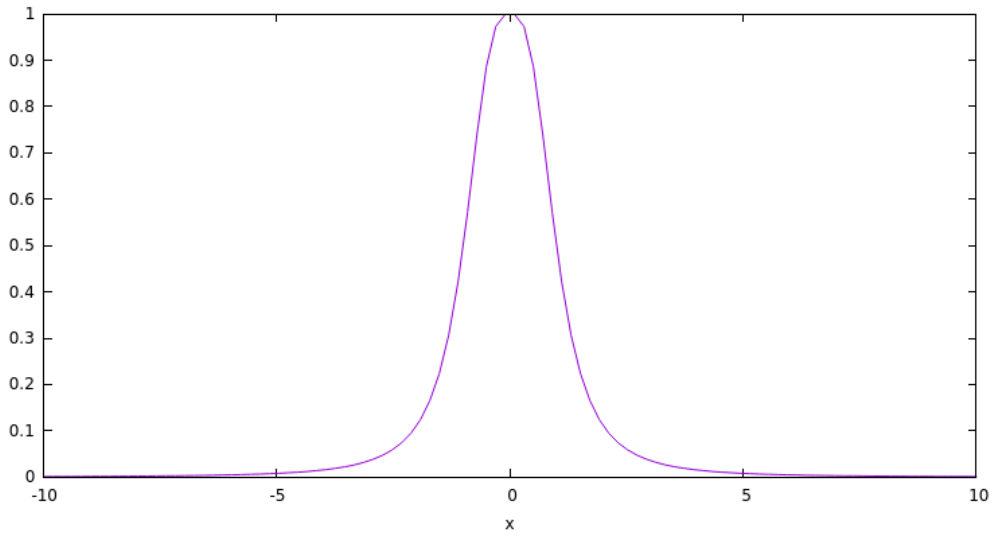
with training data $(\boldsymbol{x}_i, y_i), i = 1, \ldots, n$, and scaling parameter h. The sample can then be classified: based on the majority vote:

$$g_n(\boldsymbol{x}) = \begin{cases} 0, & if\ v_n^0(\boldsymbol{x}) \geq v_n^1(\boldsymbol{x}) \\ 1, & otherwise. \end{cases}$$

The parameter h effectively controls the smoothing. To ensure performance approaches that of the optimal Bayes decision rule, we require $lim_{n\to\infty} h_n = 0$ and $lim_{n\to\infty} n\, h_n^d = \infty$ (*i.e.* this ensures locality). The kernel maps $\boldsymbol{R}^d \rightarrow \boldsymbol{R}$ and needs to be non-negative over the neighbourhood of interest. For example, the Cauchy kernel:

$$K(\boldsymbol{x}, \boldsymbol{x_i}) = \frac{1}{(1 + \|\boldsymbol{x} - \boldsymbol{x_i}\|^{d+1})}$$

with $d = 2$ decays with distance:



The distance metric can be adapted as needed. While the Euclidean distance is common, other metrics can be used when dealing with scalars and vectors (such as similarity measures), and measures can also be constructed for dealing with sets and strings.

The kernel may also be able to take advantage of any additional information known about the underlying distribution. For example, if the distribution is not spatially uniform, the kernel could include both distance/similarity and also *direction*.

Performance may depend strongly on the choice of features chosen *i.e.* the elements of the feature vector. While the method will converge to the optimal Bayes error rate with increasing number of training examples, convergence may be very slow if a poor choice was made when selecting features. If the training data are fixed (*i.e.* feature selection is beyond your control), you may be able to transform the feature vectors into another format or modify the training set. For example, the original training data $(\boldsymbol{x}_i, y_i), i = 1, \dots, n$, could be clustered to form a new dataset $(\boldsymbol{z}_i, w_i), i = 1, \dots, k$ with weights $a_i, i = 1, \dots, k$. Computational time may be significantly reduced if $k \ll n$.

Kernel methods can also be used for regression problems, in which case the estimated value for a given sample is given by the weighted average of the *k* nearest neighbours. The weights are usually some measure of the distance between sample and the *k* nearest neighbours.

References

An elementary introduction to statistical learning theory, Kulkarni and Harman, Ch 8.

Potential application areas

- Classification for data with discrete labels
- Regression for data with continuous labels
- Discrete and continuum feature vectors, but also other inputs such as strings because of the flexibility in computing distance measures.

Advantages

- Relatively simple
- Relatively good performance given the simplicity, providing the input sample is close to the nearest neighbour(s).

Disadvantages

- Prediction error can be relatively high
- May perform poorly in high dimensions

Example(s)

1. Using R

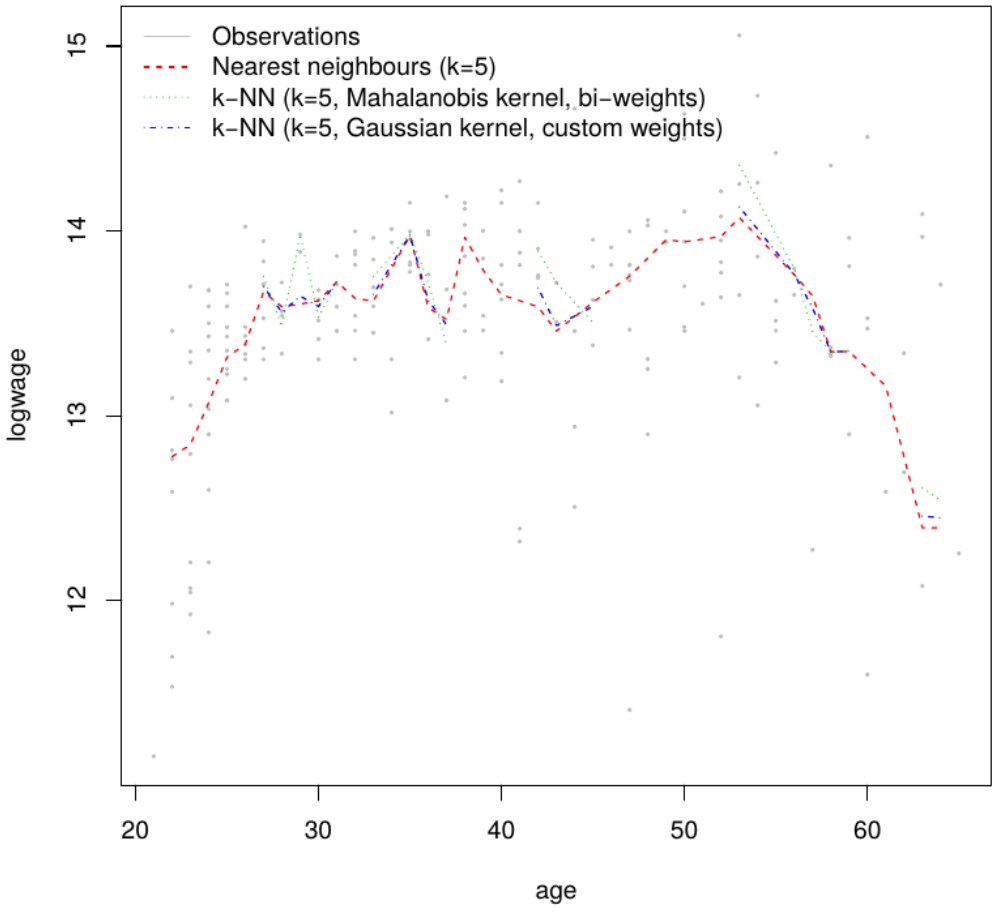
This example uses kernel methods to predict the wages from a person's age (single predictor), using the CPS71 Canadian wages dataset.

```
# Load the library
library('KernelKnn')

# To get help, at the interactive prompt, type: ?KernelKnn::KernelKnn

# Load the cps71 "Canadian wage" dataset contained in the np package
library(np)
```

```
data(cps71)

# Attach to the data so variables can be called directly
attach(cps71)

# Show the structure
str(cps71)

# When using an algorithm where the ouput depends on distance calculation
# (as is the case in k-nearest-neighbours) it is recommended to first
# scale the data,
x = scale(age)   # or, if we did not "attach", use: x = scale(cps71[,"age"])
y = logwage      # or, if we did not "attach", use: y = cps71[,"logwage"]
summary(age)
summary(logwage)
#print(age)
#print(logwage)

# Randomly split the data into train and test sets
# spl_train and spl_test are array indices, referencing the rows in
# our full dataset that will be used for testing and training
spl_train = sample(1:length(y), round(length(y) * 0.75))
spl_test = setdiff(1:length(y), spl_train)

# Predict the test data using simple k-nearest neighbours:
# - use our index arrays to pass in the training and test subsets
# x is a 1-column matrix, but x[spl_train] gets converted to a vector
# (this doesn't happen if the original matrix 'x' has more than one
# column). The KernelKnn requires the input data to be a matrix
# or data frame so we need to convert the x[spl_train] vector to a matrix
print(is.vector(x[spl_train]), is.matrix(x[spl_train]),
is.data.frame(x[spl_train]))
x_train = matrix(x[spl_train])
x_test = matrix(x[spl_test])
print(is.vector(x_train), is.matrix(x_train), is.data.frame(x_train))

preds_kNN = KernelKnn(x_train, TEST_data = x_test, y[spl_train], k = 5,
            method = 'euclidean', weights_function = NULL, regression = T)

# Predict the test data using a kernel method (mahalanobis distance
# metric for selecting the neighbours to use when predicting the output
# for a given sample), and non-uniform weights for weighting the
# selected neighbours (bi-weights)
preds_kMB = KernelKnn(x_train, TEST_data = x_test, y[spl_train], k = 5,
    method = 'mahalanobis', weights_function = 'biweight', regression = T)

# Predict the test data using a kernel method (euclidean distance metric)
# and a custom weights function for weighting the selected neighbours
norm_kernel = function(W) {
  W = dnorm(W, mean = 0, sd = 1.0)
  W = W / rowSums(W)
  return(W)
}
preds_kGN = KernelKnn(x_train, TEST_data = x_test, y[spl_train], k = 5,
    method = 'euclidean', weights_function = norm_kernel, regression = T)
```

```
# Plot the results
# - default output is Rplots.pdf
# - cex=0.25 uses circles that are 1/4 the default size

plot(age,logwage,cex=0.25,col="grey")
lines(age[spl_test],preds_kNN,col=2,lty=2)    # Colour 2, line type 2
lines(age[spl_test],preds_kMB,col=3,lty=3)    # Colour 3, line type 3
lines(age[spl_test],preds_kGN,col=4,lty=4)    # Colour 4, line type 4

# Add a legend
legend("topleft",
       c("Observations","Nearest neighbours (k=5)",
         "k-NN (k=5, Mahalanobis kernel, bi-weights)",
         "k-NN (k=5, Gaussian kernel, custom weights)"),
       lty=c(1,2,3,4),
       col=c("grey",2,3,4),
       bty="n")
```

The output is:

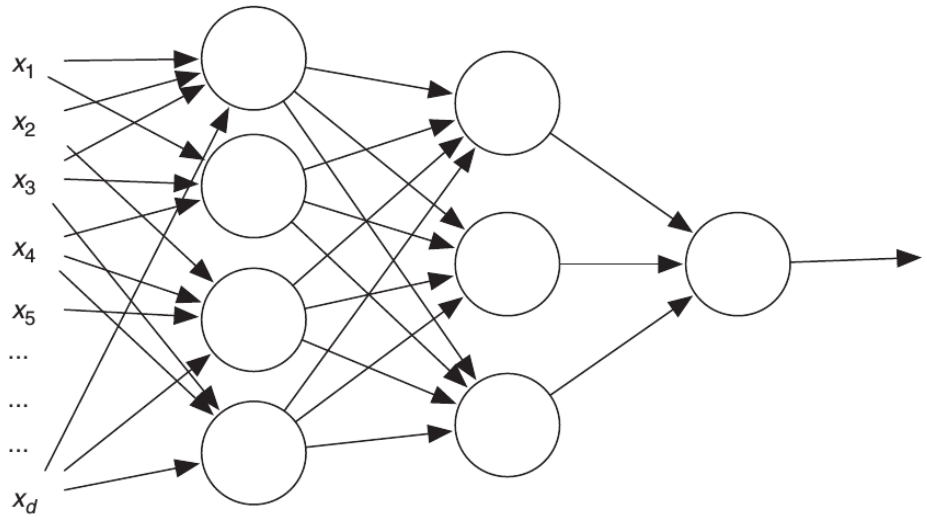


Source: adapted from the "Regression using the Housing data" vignette at https://cran.r-project.org/web/packages/KernelKnn/vignettes/regression_using_the_housing_data.html.

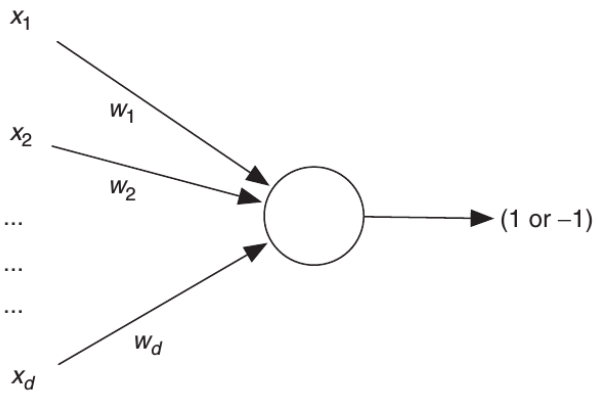
Neural Networks - multilayer networks

Description

A multilayer feedforward neural network:



is typically assembled from individual units known as perceptrons. A perceptron:



takes inputs $(\boldsymbol{x}_i, y_i), i = 1, \ldots, d$, and associated weights $w_i, i = 1, \ldots, d$, and outputs a single value, a:

$$a = sign(x_1 w_1 + x_2 w_2 + \cdots + x_d w_d)$$

where:

$$sign(u) = \begin{cases} -1, & if\ u < 0 \\ 1, & otherwise. \end{cases}$$

The output as shown above is either –1 or 1 where the threshold is 0. It is also common to use a threshold of 0.5 in which case the output is either 0 or 1.

Perceptron Convergence Procedure

Given a sample vector ***x***, the perceptron can be used to classify the sample into two classes. The classification rule is determined by the weights, and the rule can be progressively "learned" by adapting the weights based on the training dataset. Given the training dataset and some initial set of

weights, the learning rule can be refined by repeatedly cycling through the samples in the training dataset, updating the weights after each training sample. Each weight is adjusted as follows:

$$\Delta w_j = c(t - a)x_j$$

where *c* is a convergence parameter, *t* is the true classification, and *a* is the perceptron output.

After sufficient training the perceptron will correctly classify the training data, provided the data are of a type *which can be classified by a perceptron*.

The perceptron classifies a given feature vector as follows:

$$x_1 w_1 + x_2 w_2 + \cdots + x_d w_d \geq 0 \ \rightarrow class\ 1$$

and

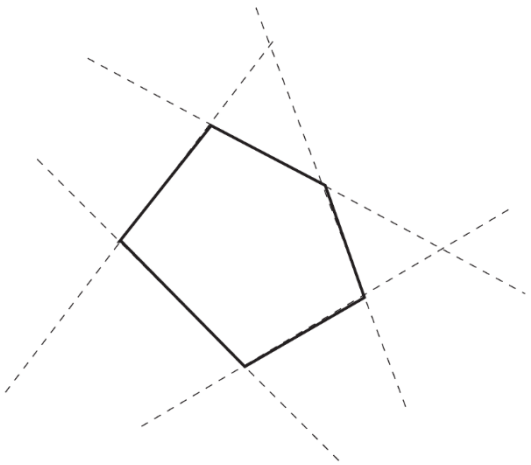
$$x_1 w_1 + x_2 w_2 + \cdots + x_d w_d < 0 \ \rightarrow class\ 0.$$

The boundary between the two classes is given by $x_1 w_1 + x_2 w_2 + \cdots + x_d w_d = 0$ which is a hyperplane in R^d. The perceptron can only correctly classify the training data if the examples are linearly separable. In other words, the hyperplane in Rd must separate the class 0 and class 1 examples, so the two classes must be *completely disjoint*.

Multilayer networks

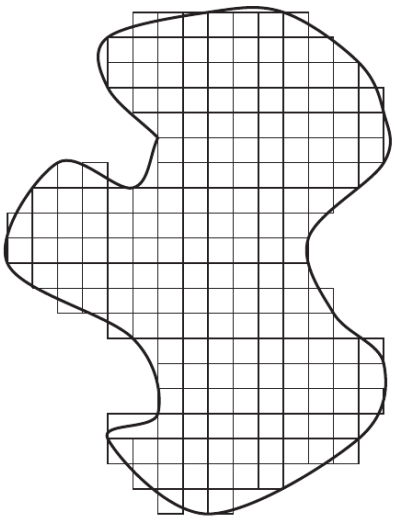
The perceptron's restriction to linearly separable classes makes it impractical for use in most situations. However, this limitation can be overcome by combining multiple units in multiple layers. With just three layers and sufficient units, it is possible to learn any decision rule.

A decision rule to classify a feature vector into a convex set can be approximated via a series of half spaces approximating the convex set. Each half space (with a linear decision boundary) is implemented by a single perceptron. For example, the convex set below is approximated using 5 half spaces:

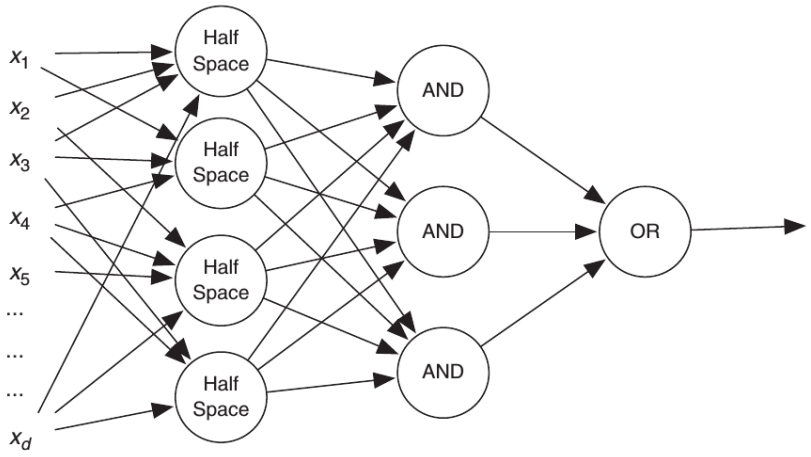


In the above example, the output from the five perceptrons is then passed to a perceptron in the second layer, which implements a logical AND of the five inputs. The output is 1 if and only if all five inputs are 1's. In other words the first layer perceptrons classify the sample against a series of half spaces, and the second layer perceptron determines if the sample lies within the convex set defined by the half spaces.

The accuracy of the half-space approximation to the convex set can be improved by taking more inputs in the first layer. The restriction to convex sets can be overcome by adding a third layer, which essentially builds up an approximation to the non-convex set using a series of convex sets:



In the first layer we again have perceptrons implementing half-spaces (but in this case the perceptrons can be thought of as belonging to groups – each group defining the half spaces for a given convex set). The second layer has a series of perceptrons, each one implementing the logical AND of its input to determine if the sample is within a given convex set. The third layer computes the logical OR of the outputs from the second layer. If the sample was within one of the convex sets (*i.e.* the output from at least one of the second layer perceptrons was 1), then the sample lies within the non-convex region.



The multilayer network can be trained using a gradient descent approach similar to the single perceptron. However due to the non-linear connectivity, the training procedure is adjusted as:

- the threshold function is replaced by a sigmoidal function. The sigmoidal function is differentiable and prevents large changes cascading through the network which occurs when a threshold function switches from 0 to 1 (or vice-versa). One commonly used sigmoidal function is:

$$\sigma(y) = \frac{1}{1 + e^{-y}}$$

where the input $y = x_1 w_1 + x_2 w_2 + \cdots + x_d w_d$. The range is 0.0 to 1.0.

Other popular activation functions are:

- Rectified linear unit (ReLU):
$$ReLU(y) = \max(y, 0)$$
ReLU is continuous but not differentiable at $y = 0$, which can make gradient descent bounce around. However it is fast, works well in practice and doesn't have a maximum (which can help gradient descent in some situations).
- Hyperbolic tangent. The range is -1.0 to 1.0, so each layer's output tends to be centred around 0.0 at the start of training, which can speed up convergence.

- to minimise the training error we use sum the squares of the errors $(t - a)^2$ over all training examples. The squared error is used (instead of, for example, the absolute error) because it is differentiable.
- the gradient descent method for updating network weights is implemented via a back-propagation procedure.

Neural networks can also be used for regression problems, in which case the final output unit is not passed through a threshold (as is done with classification problems). The final output can be the weighted sum of the inputs (*i.e.* linear activation) in which case the output can take on values from $-\infty$ to ∞. Alternatively the output can be restricted to a finite interval by passing it through a sigmoidal function.

Back propagation

The back propagation algorithm for training a network with layers $l = 1, 2, \ldots, L$, is implemented as follows:

1. Initialised the network weights
2. Repeatedly cycle through the training samples. For each sample:
    a. Compute the total input $u_i(l)$ for each neuron l in all layers, $l = 1, 2, \ldots, L$
    b. Compute the network output $a(L)$
    c. Compute the errors $\delta_i(l)$ at each neuron i, starting at layer L and propagating backwards to the second layer, where:

$$\delta_i(l) = \sigma'(u_i(l)) \sum_k \delta_k(l+1) w_{ki}(l+1) \quad \text{for } l = 1, 2, \ldots, L-1$$

    where σ' is the derivative of σ and

$$\delta_i(L) = \sigma'(u_i(L))\,(t_i - a_i(L)).$$

    d. Update the weights using:
$$\Delta w_{ij}(l) = \eta\, \delta_i(l) a_j(l-1)$$
    where η is a convergence parameter, $u_i(l)$ is the total input to neuron i in layer l, $a_i(l)$ is the output of neuron i in layer l, and $w_{ij}(l)$ is the weight from neuron j in layer $(l-1)$ to neuron i in layer l.

Back propagation is a sequential implementation of a gradient descent type algorithm. While the weights are locally optimal (instead of globally), the algorithm works well in most applications and is widely used.

References

An elementary introduction to statistical learning theory, Kulkarni and Harman, Chs 9 and 10.

Potential application areas

- Classification for data with discrete labels (with a threshold function on the final node)
- Regression for data with continuous labels (with a linear or sigmoidal function on the final node).
- Discrete and continuum feature vectors
- Convolutional networks are commonly used for visual tasks
- Recurrent networks are commonly used for time-series forecasting and tasks with arbitrary length input sequences *e.g.* natural language processing.

Advantages

- Relatively simple architecture
- Backpropagation provides a reliable method for training the network.

Disadvantages

- Difficult to understand the physical basis of the network, particularly as the number of layers increases.
- Difficult to put in a Bayesian framework.

Example(s)

1. Using R

This example uses backpropagation to train a multilayer network with two hidden layers. It is a regression task where 13 independent variables are used to predict a real-valued output. The output is however subsequently rounded to an integer allowing the results to be conveniently viewed as a 0, 1 classification problem.

```
#!/usr/bin/env Rscript

# Initialise the random number generator so that results are reproducible
set.seed(1)

# Load the data from CSV
data = read.csv(file="wine_data.csv", header=TRUE, sep=",")

# Normalise the data
normalize_to_unit_range <- function(x) {
  return ((x - min(x)) / (max(x) - min(x)))
}
maxmindf <- as.data.frame(lapply(data, normalize_to_unit_range))

# Split the data into training and test sets
no_train = round(nrow(maxmindf) * 0.8)
# - this randomly samples the data. spl_train & spl_test are array indices,
#   referencing rows in the dataset that will be used for testing & training
```

```
spl_train = sample(1:nrow(maxmindf), no_train)
spl_test = setdiff(1:nrow(maxmindf), spl_train)
trainset = maxmindf[spl_train, ]      # This creates a dataframe
testset = maxmindf[spl_test, ]

# Train the neural network
# - network has c(2,1) architecture in the hidden layers (2 nodes in the
#   first hidden layer, 1 node in the second hidden layer)
library(neuralnet)
nn <- neuralnet(Winery ~
                Alcohol + Malic.Acid + Ash + Ash.Alcalinity + Magnesium +
                Total.Phenols + Flavanoids + Nonflavanoid.Phenols +
                Proanthocyanins + Color.Intensity + Hue +
                OD280.OD315.of.dedulted.wines + Proline,
                data=trainset, hidden=c(2,1), linear.output=FALSE, threshold=0.01)
nn$result.matrix
plot(nn)

# Test the resulting output
# - remove the predictand either by selecting all desired columns by name:
#temp_test <- subset(testset, select = c("Alcohol", "Malic.Acid", "Ash",
   "Ash.Alcalinity", "Magnesium", "Total.Phenols", "Flavanoids",
   "Nonflavanoid.Phenols", "Proanthocyanins", "Color.Intensity",
   "Hue", "OD280.OD315.of.dedulted.wines", "Proline"))
# - alternatively we can simply drop the last column
no_predictors <- ncol(testset) - 1
temp_test <- testset[c(1:no_predictors)]
nn.results <- compute(nn, temp_test)

# Compute the accuracy
# - construct a data frame containing the actual and predicted values
results <- data.frame(actual = testset$Winery, prediction = nn.results$net.result)

# - print all results
print("Actual vs. predicted values:")
results

# - round the data so it can be treated as a 0,1 classification problem
roundedresults<-sapply(results,round,digits=0)
roundedresultsdf=data.frame(roundedresults)
# - output the confusion matrix
attach(roundedresultsdf)
table(actual,prediction)
```

The output is:

Actual	Prediction
0.32596291013	0.4889405599
0.72539229672	0.4398252431
0.62196861626	0.7604123303
0.90513552068	0.7476329291

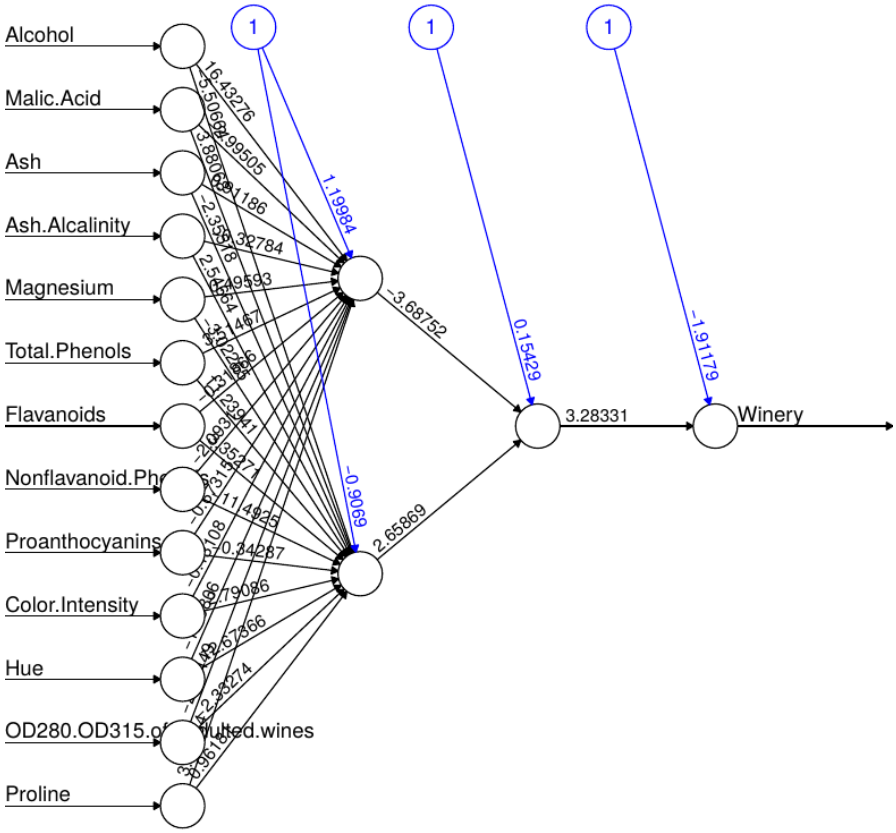
```
0.71469329529        0.7042546463
                  …
```

which, when rounded to integers, yields:

```
                  prediction
actual             0  1
     0            24  0
     1             4  8
```

showing 24 true negatives, 8 true positives, 0 false positives and 4 false negatives.

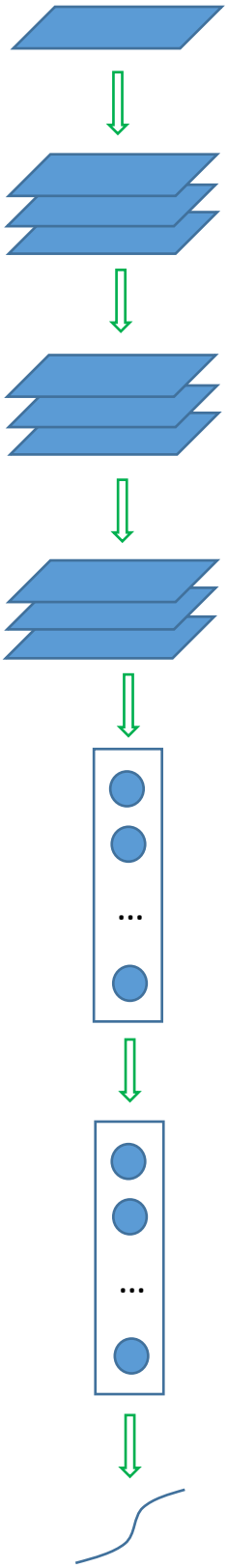
The network structure with weights is:



Source: Adapted from http://www.michaeljgrogan.com/neural-network-modelling-neuralnet-r/.

2. Using TensorFlow/Python

This example uses a convolutional neural network to classify the digit (0-9) in a series of images. The code builds a network with structure:


Input layer, 28 x 28
Convolutional layer, 28 x 28
32 feature maps
Convolutional layer, 14 x 14
64 feature maps
Max() pooling layer, 7 x 7
64 feature maps
Fully connected layer,
64 x 7 x 7 neurons
Fully connected layer,
10 neurons
Softmax output

```
import numpy as np
import os,sys
import tensorflow as tf
from sklearn.datasets import load_sample_image

def main():
    # Seed the random number generator
    np.random.seed(42)

    # MNIST dataset
    height = 28
    width = 28
    channels = 1
    n_inputs = height * width

    conv1_fmaps = 32
    conv1_ksize = 3
    conv1_stride = 1
    conv1_pad = "SAME"

    conv2_fmaps = 64
    conv2_ksize = 3
    conv2_stride = 2
    conv2_pad = "SAME"

    pool3_fmaps = conv2_fmaps

    n_fc1 = 64
    n_outputs = 10

    with tf.name_scope("inputs"):
        X = tf.placeholder(tf.float32, shape=[None, n_inputs], name="X")
        X_reshaped = tf.reshape(X, shape=[-1, height, width, channels])
        y = tf.placeholder(tf.int32, shape=[None], name="y")

    conv1 = tf.layers.conv2d(X_reshaped, filters=conv1_fmaps, kernel_size=conv1_ksize,
                             strides=conv1_stride, padding=conv1_pad,
                             activation=tf.nn.relu, name="conv1")
    conv2 = tf.layers.conv2d(conv1, filters=conv2_fmaps, kernel_size=conv2_ksize,
                             strides=conv2_stride, padding=conv2_pad,
                             activation=tf.nn.relu, name="conv2")

    with tf.name_scope("pool3"):
        pool3 = tf.nn.max_pool(conv2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding="VALID")
        pool3_flat = tf.reshape(pool3, shape=[-1, pool3_fmaps * 7 * 7])

    with tf.name_scope("fc1"):
        fc1 = tf.layers.dense(pool3_flat, n_fc1, activation=tf.nn.relu, name="fc1")

    with tf.name_scope("output"):
        logits = tf.layers.dense(fc1, n_outputs, name="output")
        Y_proba = tf.nn.softmax(logits, name="Y_proba")

    with tf.name_scope("train"):
        xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(logits=logits, labels=y)
        loss = tf.reduce_mean(xentropy)
        optimizer = tf.train.AdamOptimizer()
        training_op = optimizer.minimize(loss)

    with tf.name_scope("eval"):
        correct = tf.nn.in_top_k(logits, y, 1)
        accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))

    with tf.name_scope("init_and_save"):
        init = tf.global_variables_initializer()

    from tensorflow.examples.tutorials.mnist import input_data
    mnist = input_data.read_data_sets("/tmp/data/")
```

```
    n_epochs = 10
    batch_size = 100

    with tf.Session() as sess:
        init.run()
        for epoch in range(n_epochs):
            for iteration in range(mnist.train.num_examples // batch_size):
                X_batch, y_batch = mnist.train.next_batch(batch_size)
                sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
            acc_train = accuracy.eval(feed_dict={X: X_batch, y: y_batch})

            acc_test = 0.0
            for iteration in range(mnist.test.num_examples // batch_size):
                X_batch, y_batch = mnist.test.next_batch(batch_size)
                acc_test_batch = accuracy.eval(feed_dict={X: X_batch, y: y_batch})
                acc_test += acc_test_batch * y_batch.shape[0]

            acc_test /= float(mnist.test.num_examples)
            print("Test accuracy:", acc_test)


if __name__ == "__main__":
    main()
```

The output is:

```
Test accuracy: 0.9815000081062317
Test accuracy: 0.9823000115156174
Test accuracy: 0.9844000118970871
Test accuracy: 0.9862000095844269
Test accuracy: 0.9886000102758408
Test accuracy: 0.9870000106096267
Test accuracy: 0.9857000106573105
Test accuracy: 0.9890000081062317
Test accuracy: 0.9892000079154968
Test accuracy: 0.9898000085353851
```

Source: Adapted from an example in Ch 13 of Hands-On Machine Learning with Scikit-Learn & TensorFlow (Géron, 2017), available at: https://github.com/ageron/handson-ml/blob/master/13_convolutional_neural_networks.ipynb

Support Vector Machines

Description

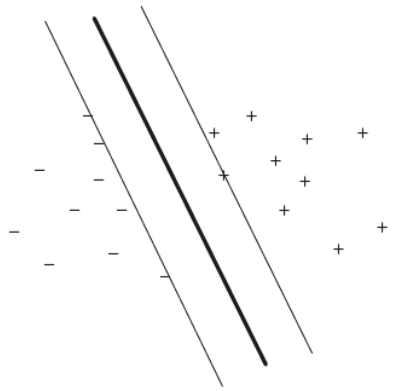
Support Vector Machines (SVMs) overcome the difficulty of separating feature vectors by mapping them to a high dimensional space, where it may be possible to separate them using linear methods (ie. a hyperplane in the transformed space). The SVM approach effectively results in the use of non-linear classifiers in the original space, thereby overcoming the representational limitations of linear classifiers.

Support vector machines have been successfully applied to problems where the transformed data are separated by a large margin. While many separating hyperplanes may perform equally well on the training dataset, a hyperplane with large margin may provide the best generalized performance on unseen data.

In the classification problem, the task is to separate a given sample into class -1 or 1. Given training data $(\boldsymbol{x}_i, y_i), i = 1, \ldots, n$, where $\boldsymbol{x} \in \boldsymbol{R}^d$, we map the original data:

$$\Phi: \boldsymbol{R}^d \rightarrow \mathcal{H}$$

to the new feature space $\mathcal{H}$. The transformed data become $\Phi(\boldsymbol{x}_i, y_i)$ and we seek the hyperplane which separates the transformed data with largest margin. If the transformed data are linearly separable, the two hyperplanes of maximum margin (passing through the two samples) are called the support vectors:



Note: some texts refer to the samples defining the largest margin hyperplanes as being the support vectors, as opposed to hyperplanes themselves.

The method can be used for handling outliers by choosing whether the margin is hard or soft:

- Hard margin classification: the classes must be linearly separable. This method may be adversely affected by outliers in the data
- Soft margin classification: margin violations are tolerated to cater for outliers. A trade-off must be defined between the margin width and minimising the number of margin violations.

The SVM approach of transforming the feature space and finding a large margin classifier can be formulated as an optimisation problem. The resulting classifier can, under certain conditions, take the form of kernel methods which can be efficiently implemented. Three scenarios are possible:

- The Φ mapping is not used:

    - The training data are linearly separable. The optimisation problem is finding the largest margin classifier
    - The training data are not linearly separable. The optimisation problem is finding the hyperplane that separates the data *as much as possible*, by introducing slack variables
- The Φ mapping is used:
    - The training data are not linearly separable. Again the optimisation seeks the hyperplane that separates the data as much as possible, but in this case the transformed case results in *kernel methods*.

If the equation for the hyperplane we seek is given by $\boldsymbol{w}$ and some $\Phi(\boldsymbol{x})$ lies in the hyperplane, then:

$$\boldsymbol{w} \,.\, \Phi(\boldsymbol{x}) + b = 0$$

where $b/\|\boldsymbol{w}\|$ is the distance from the hyperplane to the origin. If the hyperplane separates the training data:

$$\begin{aligned} \boldsymbol{w} \,.\, \Phi(\boldsymbol{x}_i) + b \; > 0 \quad \text{if} \;\; y_i = 1 \\ \boldsymbol{w} \,.\, \Phi(\boldsymbol{x}_i) + b < 0 \quad \text{if} \;\; y_i = -1 \end{aligned}$$

which we can renormalise and express as:

$$\begin{aligned} \boldsymbol{w} \,.\, \Phi(\boldsymbol{x}_i) + b \geq \; 1 \quad \text{if} \;\; y_i = 1 \\ \boldsymbol{w} \,.\, \Phi(\boldsymbol{x}_i) + b \leq -1 \quad \text{if} \;\; y_i = -1 \end{aligned}$$

We may not be able to separate all the samples, so we introduce slack variables ξ_i, $\xi_i \; \geq 0$. The above equations then become:

$$\begin{aligned} \boldsymbol{w} \,.\, \Phi(\boldsymbol{x}_i) + b \geq \; 1 - \xi_i \quad \text{if} \;\; y_i = 1 \\ \boldsymbol{w} \,.\, \Phi(\boldsymbol{x}_i) + b \leq -1 + \xi_i \quad \text{if} \;\; y_i = -1 \end{aligned}$$

To obtain the largest margin classifier, we need to construct the optimisation problem. The margin is the shortest distance from a positive example to the hyperplane (d_+) plus the shortest distance from a negative example to the hyperplane (d_-), which is given by:

$$\text{margin} = \; d_+ + \; d_- = \frac{2}{\|\boldsymbol{w}\|}.$$

Therefore to maximise the margin we need to minimise $\|\boldsymbol{w}\|$ or $\|\boldsymbol{w}\|^2$. The optimisation problem is then:

$$\begin{gathered} \text{minimise} \; \|\boldsymbol{w}\|^2 + C \sum_i \xi_i \\ \text{subject to} \; y_i(\boldsymbol{w} \,.\, \Phi(\boldsymbol{x}_i) + b) - \; 1 + \xi_i \geq 0 \;\; \text{for} \;\; i = 1, \ldots, n \\ \xi_i \geq 0 \;\; \text{for} \;\; i = 1, \ldots, n \end{gathered}$$

for the soft margin problem. For the hard margin problem we don't need the slack variables, and the optimisation problem is simply:

$$\begin{gathered} \text{minimise} \; \|\boldsymbol{w}\|^2 \\ \text{subject to} \; y_i(\boldsymbol{w} \,.\, \Phi(\boldsymbol{x}_i) + b) \geq 0 \;\; \text{for} \;\; i = 1, \ldots, n \end{gathered}$$

The training process involves solving the optimisation problem to obtain $\boldsymbol{w}$ and b. Both optimisation problems are Quadratic Programming problems.

Once $\boldsymbol{w}$ and b are known, a given sample can then be classified by determining on which side of the hyperplane it lies. If

$$\boldsymbol{w} \,.\, \Phi(\boldsymbol{x}) + b = \sum_{i=1}^{n} \alpha_i y_i \left(\Phi(\boldsymbol{x}_i) \cdot \Phi(\boldsymbol{x}) \right) + b$$

is > 0, $\boldsymbol{x}$ is classified as 1, and -1 otherwise.

The constrained optimisation problem (primal problem) can be expressed in a different form, the dual problem. Under certain conditions (which are satisfied by SVM), the primal and dual problems have the same solution. (If not, the solution of the dual problem is typically a lower bound to the solution of the primal problem.) The dual problem is preferred because: (i) it is faster to solve if the number of training instances is less than the number of features; and (ii) it enables the mapping $\Phi(\boldsymbol{x})$ to be replaced by a kernel function $K(\boldsymbol{x}_i, \boldsymbol{x})$.

The mapping $\Phi(\boldsymbol{x})$ may be difficult to compute, however under certain conditions[1] the dot product $\Phi(\boldsymbol{x}_i) \cdot \Phi(\boldsymbol{x})$ can be replaced by a kernel function $K(\boldsymbol{x}_i, \boldsymbol{x})$ which is easy to compute. In practice the user chooses the kernel function and doesn't need to know the corresponding Φ and $\mathcal{H}$. However the choice of kernel can strongly influence the generalized performance. The most common kernels are:

$$\begin{aligned}
\text{Linear:} \quad & K(\boldsymbol{a}, \boldsymbol{b}) = \boldsymbol{a}^T \cdot \boldsymbol{b} \\
\text{Polynomial:} \quad & K(\boldsymbol{a}, \boldsymbol{b}) = (\gamma \boldsymbol{a}^T \cdot \boldsymbol{b} + r)^d \\
\text{Gaussian RBF:} \quad & K(\boldsymbol{a}, \boldsymbol{b}) = e^{-\gamma \|\boldsymbol{a} - \boldsymbol{b}\|^2} \\
\text{Sigmoid:} \quad & K(\boldsymbol{a}, \boldsymbol{b}) = tanh(\gamma \boldsymbol{a}^T \cdot \boldsymbol{b} + r)
\end{aligned}$$

where $\boldsymbol{a}$ and $\boldsymbol{b}$ are two instances.

Note: SVM is sensitive to scaling, so the input samples must be scaled (preferable to have mean=0 and variance=1).

If the data are not linearly separable, you could try:

- adding polynomial features
  - with low degree it cannot deal with complex data
  - with high degree it adds many features, so the method is slow.
- adding features. For example, use one or more similarity functions to transform the original data into new dimensions. A radial basis function centred on a given landmark can be used to compute a new feature. However adding a lot of new features can make the method slow.
- using non-linear SVM. For example, use a polynomial kernel or a RBF kernel which overcome the performance issues.

The Gaussian RBF and polynomial kernels are the most common (after linear SVM).

References

An elementary introduction to statistical learning theory, Kulkarni and Harman, Ch 17.

Potential application areas

- Linear and non-linear classification
- Linear and non-linear regression

[1] The kernel must satisfy Mercer's condition.

- Outlier detection.

Advantages

- Has shown very good results on some problems
- Well suited to classification of complex but small-medium sized datasets.

Disadvantages

- Performance can be influenced by the choice of kernel
- Does not scale well with the size of the input dataset
- Does not handle multiclass classifications (but this can be overcome by using a one-vs-one strategy of binary classifications)
- Can be unstable when used with boosting and the AdaBoost algorithm.

Example(s)

1. Using R

This example compares linear regression and SVM for predicting values from a simple set of two-dimensional data. The SVM parameters are then tuned to improve the fit.

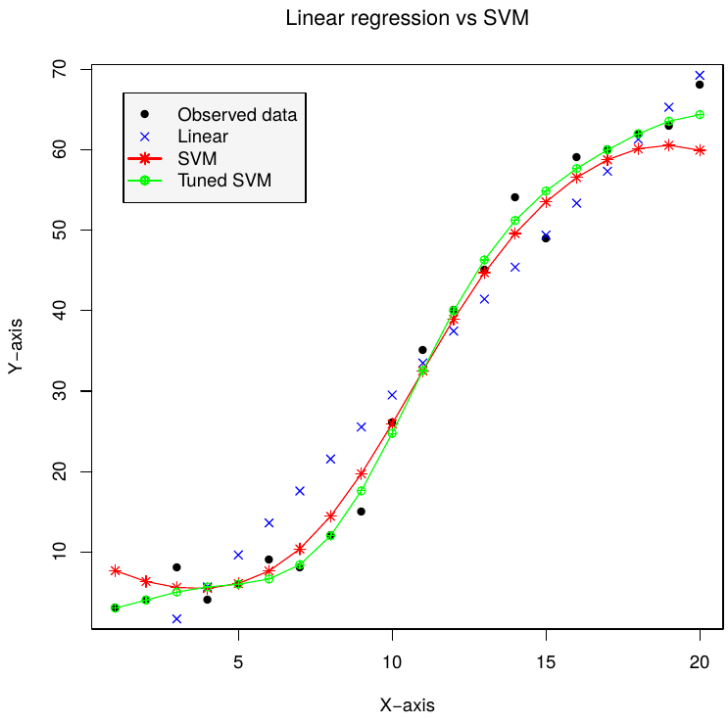
```
# Load the data from CSV
dataDirectory <- "."
data <- read.csv(paste(dataDirectory, 'data.csv', sep="/"), header = TRUE)

# Simple linear regression
# ------------------------

# Plot the data using point symbol 16 (pch=16)
pdf("my_plots.pdf")                                    # Set the filename
plot(data, pch=16,
     main=expression("Linear regression vs SVM"),
     xlab="X-axis",
     ylab="Y-axis")
# To insert superscripts, use: main=expression("graph title"^2)
# To insert subscripts,   use: main=expression("graph title"[2])

# Create a linear regression model (format: response ~ terms where "response"
# is the dependent variable and "terms" are the independent variables)
model <- lm(Y ~ X, data)

# Add the fitted line to the current plot
abline(model)

# Make a prediction for each X
# - we could pass in the orginal data frame as it has X,Y labels:
#     predictedY <- predict(model, data)
#   or we could extract the X column and create a new dataframe:
Xdf = data.frame(data$X)
colnames(Xdf) <- c("X")
predictedY <- predict(model, Xdf)

# Display the predictions
points(data$X, predictedY, col = "blue", pch=4)
```

```
# Compute the RMSE
rmse <- function(error)
{
  sqrt(mean(error^2))
}

error <- model$residuals  # same as data$Y - predictedY
predictionRMSE <- rmse(error)
cat(sprintf("RMSE from linear regression: %.4f\n", predictionRMSE))

# Support Vector Regression
# -------------------------

library(e1071)
model <- svm(Y ~ X , data)
predictedY <- predict(model, data)
points(data$X, predictedY, col = "red", pch=8)
lines(data$X, predictedY, col = "red")

error <- data$Y - predictedY
svrPredictionRMSE <- rmse(error)
cat(sprintf("RMSE from SVM: %.4f\n", svrPredictionRMSE))

# Add a legend
legend("topleft",
       inset=.05,
       cex = 1,
       #title="Legend",
       c("Observed data", "Linear","SVM", "Tuned SVM"),
       horiz=FALSE,
       lty=c(0,0,1,1),
       lwd=c(0,1,2,2),
       pch=c(16,4,8,10),
       col=c("black","blue", "red", "green"),
       bg="grey96")

# Tune the SVM fit over a wide range of epsilon and cost values
# -------------------------------------------------------------
# perform a grid: search
# - epsilon is varied from 0 to 1 in increments on 0.1, while the
#   cost is varied from 2^2, 2^3, ..., 2^9
coarseTuneResult <- tune(svm, Y ~ X,  data = data,
                         ranges = list(epsilon = seq(0,1,0.1), cost = 2^(2:9))
)
print(coarseTuneResult)

# The coarse tuning indicated epsilon should be around 0 - 0.2, so we
investigate this area
tuneResult <- tune(svm, Y ~ X,  data = data,
                   ranges = list(epsilon = seq(0,0.2,0.01), cost = 2^(2:9))
)
print(tuneResult)

# Automatically select the best model
# -----------------------------------
```

```
tunedModel <- tuneResult$best.model
tunedModelY <- predict(tunedModel, data)

error <- data$Y - tunedModelY
tunedModelRMSE <- rmse(error)
cat(sprintf("RMSE from tuned SVM: %.4f\n", tunedModelRMSE))

# Plot the tuned results
points(data$X, tunedModelY, col = "green", pch=10)
lines(data$X, tunedModelY, col = "green")

# Draw the tuning graphs
plot(coarseTuneResult)
plot(tuneResult)
```

The output is:

```
RMSE from linear regression: 5.7038
RMSE from SVM: 3.1571

Parameter tuning of 'svm':
- sampling method: 10-fold cross validation
- best parameters:
 epsilon cost
       0    4
- best performance: 9.815355

RMSE from tuned SVM: 1.9977
```



Source: adapted from https://www.svm-tutorial.com/2014/10/support-vector-regression-r/.

Boosting

Description

Boosting is an iterative method for improving the performance of one or more learning methods. In each round a learning method is used to produce a new learning rule that focusses attention on samples which were misclassified (or had larger error, in the case of estimation) in the previous round. The final estimate or classification for a given sample is a weighted average of the individual learning rules constructed in each round.

For the classification problem, given a set of classification rules $h_1(\boldsymbol{x}), h_2(\boldsymbol{x}), \dots, h_T(\boldsymbol{x})$ from each round $1, 2, \dots, T$, the new (or combined) classification rule is:

$$\text{sign}\left(\sum_{i=1}^{T} \alpha_i \, h_i(\boldsymbol{x})\right)$$

where the α_i are weights and $h_i(\boldsymbol{x})$ is the classification of $\boldsymbol{x}$ from the rule obtained in round i.

There are two common boosting methods:

1. AdaBoost

   In order to iteratively improve the learning rules, we assign greater weights to those samples that were misclassified in the previous round. If $D_t(i)$ is the weight assigned to sample i in round t, the weak learning algorithm used in each round attempts to minimise the weighted error:

   $$\epsilon_t = \sum_{i=1}^{T} D_i(i) \; I_{\{h_t(x_i) \neq y_i\}}$$

   where $I_{\{h_t(x_i) \neq y_i\}}$ counts the number of misclassifications. At the end of each round, the error rate ϵ_t is calculated and used to compute the weight α_t and the distribution to be used in the next round, D_{t+1}. There are various techniques for computing the α_t and D_{t+1}. In AdaBoost the weight assigned to a given predictor is:

   $$\alpha_t = \eta \log \frac{1 - \epsilon_t}{\epsilon_t}$$

   where η is a learning rate hyperparameter and the weights are updated as follow:

   $$w_i \leftarrow \begin{cases} w_i & \text{if } \hat{y}_i^t = y_i \\ w_i . e^{\alpha_t} & \text{if } \hat{y}_i^t \neq y_i \end{cases}$$

2. Gradient boosting

   Instead of modifying the weights to focus attention on samples that were misclassified in the previous round, Gradient boosting fits a predictor to the residual errors made in the previous round.

   A learning rate η is used to scale the contribution of each successive tree. If the learning rate is low, a lot of trees will be required but the generalized performance will be better than if η was high. This method of regularization is called *shrinkage*.

Early stopping can be used to determine how many decision trees are needed. Early stopping stops learning when the validation error stops decreasing (normally the MSE of the training set is the criteria for stopping).

The error on the training data approaches zero exponentially with the number of rounds, but the generalization error of the combined classifier H:

$$R(H) \leq \frac{1}{n}\sum_{i=1}^{T} I_{\{h_t(x_i) \neq y_i\}} + O\left(\sqrt{\frac{TV}{n}}\right)$$

grows with the number of rounds T, where V is the Vapnik-Chervonenkis dimension. This suggest boosting may over-fit the training data if run for too many rounds, but in practice the generalization error commonly decreases with the number of rounds.

Boosting is an example of an *ensemble method*, whereby a collection of classifiers (or estimators) are combined in the hope that the combined method outperforms any of the individual methods. Ensemble methods can be constructed by combining different learning algorithms or by manipulating the training data. For example:

- *Bagging* is a method of repeatedly sampling (with replacement) from a set of *n* sample to create a new training set. The new set may have multiple occurrences of some samples, while others will be omitted. The learning algorithm is then used on each of the new datasets.
- *Disjoint subsets* can be created by systematically taking a series of disjoint subsets from the full training dataset, and using the learning algorithm on each new subset. This is a type of cross-validation approach.

References

An elementary introduction to statistical learning theory, Kulkarni and Harman, Ch 18.

Potential application areas

Advantages

- Has shown very good results on some problems.

Disadvantages

- Computational cost

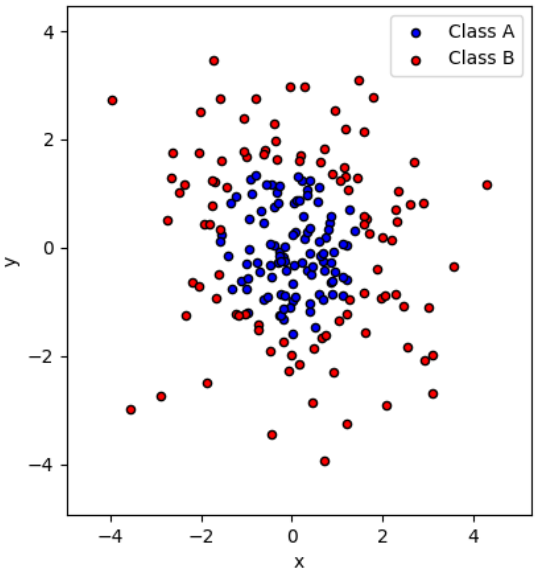
Example(s)

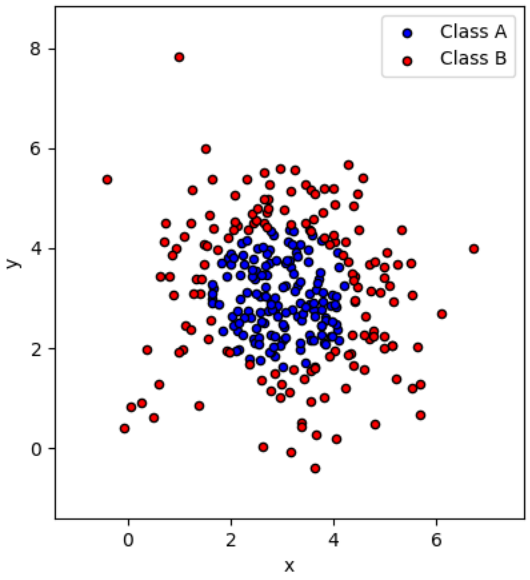
1. Using Python

This example uses the AdaBoost algorithm to separate samples labelled as either class A or class B. The samples are drawn from two bi-normal distributions. Each set has half of its members assigned to class A and the remainder to class B. Samples are randomly selected from the bi-normal distribution and then separated into two equal-sized classes – the group closest to the centre of the distribution is assigned to class A and the other half to class B.

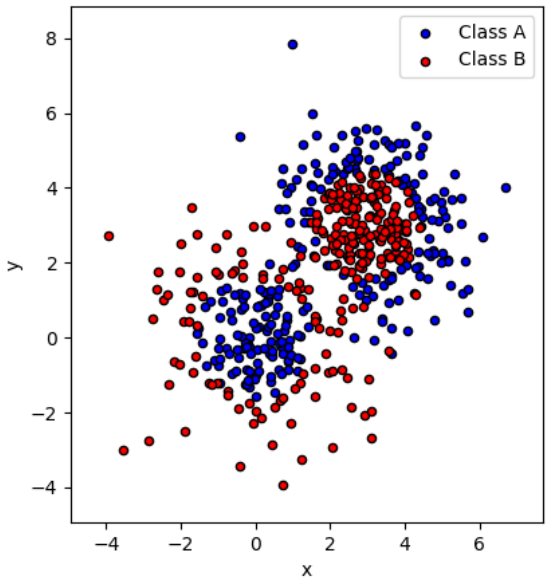
The first distribution has mean (0.0, 0.0):

The second distribution has mean (3.0, 3.0):





The combined dataset is:



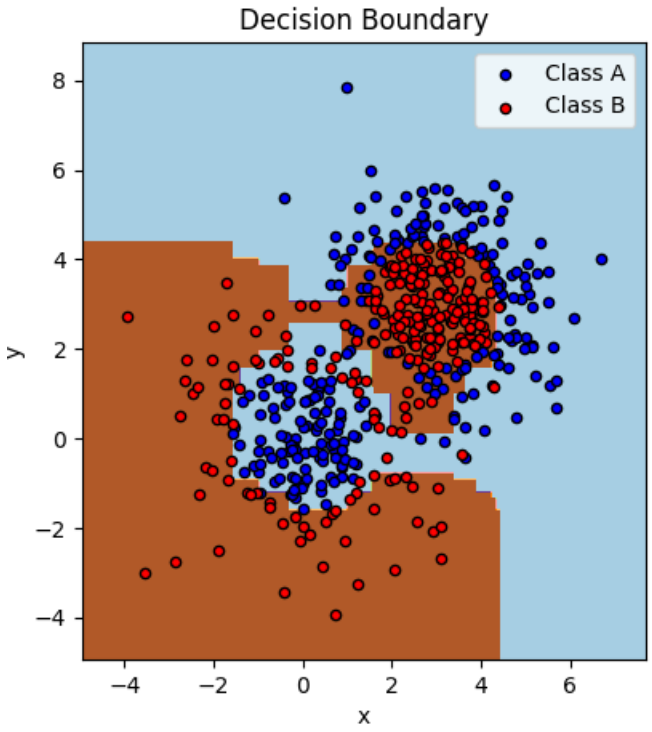
Note the classification has been inverted for those samples from the second distribution. This was done to make the task more challenging. The resulting classes are not linearly separable and the task is to determine the decision boundary.

```
import numpy as np
import matplotlib.pyplot as plt

from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import make_gaussian_quantiles

# Construct datasets
# - 2D bi-normal distributions with mean = (0,0), with samples
#   assigned to one of two classes based on their "quantile" distance
#   from the centre of the distribution ie. the samples are ranked in order
#   of distance from the centre: the closest half are assigned to class A
```

```
#   and the other half to class B
X1, y1 = make_gaussian_quantiles(cov=2.0,
                                 n_samples=200, n_features=2,
                                 n_classes=2, random_state=1)
# - simple 2D bi-normal distributions with mean = (3,3)
X2, y2 = make_gaussian_quantiles(mean=(3, 3), cov=1.5,
                                 n_samples=300, n_features=2,
                                 n_classes=2, random_state=1)

# Join the datasets together (the classes are inverted in the 2nd set
# to make the task more challenging)
X = np.concatenate((X1, X2))
y = np.concatenate((y1, - y2 + 1))

# Create and fit an AdaBoosted decision tree
bdt = AdaBoostClassifier(DecisionTreeClassifier(max_depth=1),
                         algorithm="SAMME",
                         n_estimators=200)

bdt.fit(X, y)

plt.figure(figsize=(10, 5))

# Plot the decision boundaries
plt.subplot(121)
# - construct an equally spaced grid spanning the domain of interest
plot_step = 0.02
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, plot_step),
                     np.arange(y_min, y_max, plot_step))

# Use the fitted model to classify every point on the grid
Z = bdt.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
cs = plt.contourf(xx, yy, Z, cmap=plt.cm.Paired)
plt.axis("tight")

# Plot the training points
plot_colors = "br"
class_names = "AB"

# This loop runs twice:
# 1st iteration it plots the training samples labelled as class A:
#      i=0, n='A' (i.e. class name), c='b' (i.e. colour blue)
# 2nd iteration it plots the training samples labelled as class B:
#      i=1, n='B' (i.e. class name), c='r' (i.e. colour red)
for i, n, c in zip(range(2), class_names, plot_colors):
    idx = np.where(y == i)   # Select the points in either class A or B
    plt.scatter(X[idx, 0], X[idx, 1],
                c=c, cmap=plt.cm.Paired,
                s=20, edgecolor='k',
                label="Class %s" % n)
plt.xlim(x_min, x_max)
plt.ylim(y_min, y_max)
plt.legend(loc='upper right')
```

```
plt.xlabel('x')
plt.ylabel('y')
plt.title('Decision Boundary')

plt.show()
```

The output is:



Source: adapted from http://scikit-learn.org/stable/auto_examples/ensemble/plot_adaboost_twoclass.html#sphx-glr-auto-examples-ensemble-plot-adaboost-twoclass-py.

Decision trees

Description

Decision trees are simple tree-like structures that partition the training data into "groups" using the Classification and Regression Tree (CART) algorithm.

When applied to classification problems, the CART algorithm constructs the tree by progressively splitting the training data into two subsets using a single feature k and threshold t_k. At each step it searches for the pair (k, t_k) which minimises the cost function:

$$J(k, t_k) = \frac{m_{left}}{m} G_{left} + \frac{m_{right}}{m} G_{right}$$

where $G_{left/right}$ is the impurity of the left or right subset, and $m_{left/right}$ is the number of instances in the left or right subset. The impurity measure is usually either the Gini impurity:

$$G_i = 1 - \sum_{k=1}^{n} p_{i,k}^2$$

(where $p_{i,k}$ is the proportion of instances assigned to node i that belong to class k) , or the entropy:

$$H_i = - \sum_{\substack{k=1 \\ p_{i,k} \neq 0}}^{n} p_{i,k} \, log(p_{i,k})$$

When applied to regression problems, the CART algorithm splits each region in a way that makes most instances as close as possible to the predicted value for that subset:

$$J(k, t_k) = \frac{m_{left}}{m} MSE_{left} + \frac{m_{right}}{m} MSE_{right}$$

where

$$MSE_{node} = \sum_{i \in node} (\hat{y}_{node} - y^i)^2$$

$$\hat{y}_{node} = \frac{1}{m_{node}} \sum_{i \in node} y^i$$

The predicted regression value for a given leaf node is simply the average of all instances in that node.

Random forests are an ensemble method using decision trees. They are very powerful learning algorithms.

References

Hands-On Machine Learning with Scikit-Learn and TensorFlow, Géron, Ch 6.

Potential application areas

- classification
- multi-output classification
- regression

Advantages

- easy to understand
- predictions are very fast
- random forests are very powerful
- random forests can be used to indicate which features are the most important.

Disadvantages

- the data may need to be rotated as the algorithm splits data perpendicular to the feature axis being used for the split
- the algorithm can be sensitive to small variations in the data
- regularization is usually required to prevent over-fitting the training data
- training can be slow.
- range of predictions is limited to the range of the training data
- specifically in regard to random forests:
    - don't train well on smaller datasets
    - not easily interpreted like individual decision trees
    - training can be very slow (it trains multiple decision trees)

Example(s)

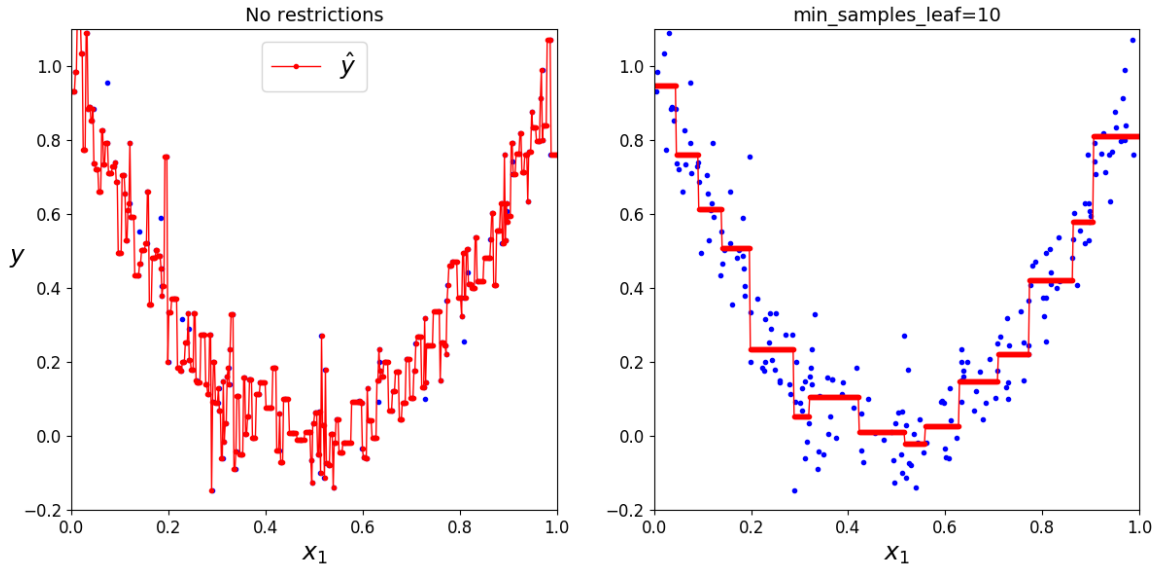
1. Using Python

This example uses a decision tree for regression. It is applied to a randomly generated dataset having a parabolic shape.

```
import numpy as np
import os
import matplotlib
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeRegressor

np.random.seed(42)

# Create a dataset with an underlying parabolic distribution plus
noise
m = 200
X = np.random.rand(m, 1)
y = 4 * (X - 0.5) ** 2
y = y + np.random.randn(m, 1) / 10

# Fit the decision tree
tree_reg1 = DecisionTreeRegressor(random_state=42)
tree_reg1.fit(X, y)

# Fit a decision tree with regularization
tree_reg2 = DecisionTreeRegressor(random_state=42,
min_samples_leaf=10)
tree_reg2.fit(X, y)

# Make predictions for values uniformly distributed between 0,1
x1 = np.linspace(0, 1, 500).reshape(-1, 1)
y_pred1 = tree_reg1.predict(x1)
```

```
y_pred2 = tree_reg2.predict(x1)

plt.figure(figsize=(11, 4))
plt.rcParams['axes.labelsize'] = 14
plt.rcParams['xtick.labelsize'] = 12
plt.rcParams['ytick.labelsize'] = 12

plt.subplot(121)
plt.plot(X, y, "b.")
plt.plot(x1, y_pred1, "r.-", linewidth=1, label=r"$\hat{y}$")
plt.axis([0, 1, -0.2, 1.1])
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", fontsize=18, rotation=0)
plt.legend(loc="upper center", fontsize=18)
plt.title("No restrictions", fontsize=14)

plt.subplot(122)
plt.plot(X, y, "b.")
plt.plot(x1, y_pred2, "r.-", linewidth=1, label=r"$\hat{y}$")
plt.axis([0, 1, -0.2, 1.1])
plt.xlabel("$x_1$", fontsize=18)
plt.title("min_samples_leaf={}".format(tree_reg2.min_samples_leaf),
fontsize=14)
plt.show()
```

The output is:



Source: Examples in Ch 6, Hands-On Machine Learning with Scikit-Learn and TensorFlow, Géron.

Principal Component Analysis

Description

PCA aims to identify a set of axes which explain a desired portion of the variance in the original dataset. The approach is:

1. Identify the 1st axis which explains most the variance
2. Identify the 2nd axis (orthogonal to the 1st axis) which explains most the remaining variance
3. … further axes are progressively identified.

Singular Value Decomposition is used to decompose the matrix of training samples $\boldsymbol{X}$:

$$\boldsymbol{X} = \boldsymbol{U} \cdot \boldsymbol{\Sigma} \cdot \boldsymbol{V}^T$$

where the principal components are in the columns of $\boldsymbol{V}$. Given the principal components, the training data are then projected onto the d-dimensional hyperplane:

$$\boldsymbol{X}_{d-proj} = \boldsymbol{X} \cdot \boldsymbol{W}_d$$

where $\boldsymbol{W}_d$ contains the first d columns of $\boldsymbol{V}$.

References

Hands-On Machine Learning with Scikit-Learn and TensorFlow, Géron, Ch 8.

Potential application areas

- Compression
- Dimensionality reduction

Advantages

- Stochastic version is fast
- Incremental version can be applied to large datasets and online training

Disadvantages

- Loses information, but this is controlled by the user (the number of dimensions)
- Prediction/classification etc. is usually simpler in the lower dimensional subspace, but in some cases it may be more complex.

Example(s)

1. Using Python

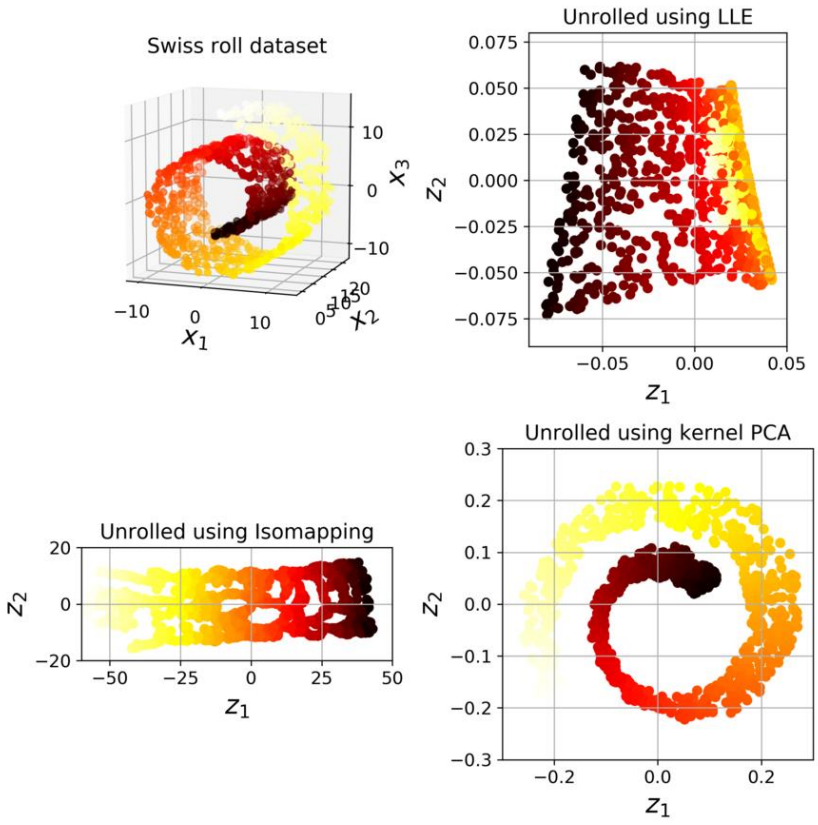
This example uses three dimension reduction techniques to reduce the dimension of a 3D dataset to 2D. The original data appear to lie approximately on a 2D manifold, but the data cannot simply be flattened because the manifold “curls over”.

```
import numpy as np
import os,sys
```

```
import matplotlib
import matplotlib.pyplot as plt

# Seed the random number generator
np.random.seed(42)

# To plot pretty figures
import matplotlib
import matplotlib.pyplot as plt
plt.rcParams['axes.labelsize'] = 14
plt.rcParams['xtick.labelsize'] = 12
plt.rcParams['ytick.labelsize'] = 12

# Construct the dataset
from mpl_toolkits.mplot3d import proj3d
from sklearn.datasets import make_swiss_roll
X, t = make_swiss_roll(n_samples=1000, noise=0.2, random_state=42)

fig = plt.figure(figsize=(8, 8))
fig.tight_layout()

# Display the original dataset
axes = [-11.5, 14,   -2, 23,   -12, 15]
original_plot = fig.add_subplot(221, projection='3d')
original_plot.set_title("Swiss roll dataset", fontsize=14)
original_plot.set_aspect('equal')
original_plot.scatter(X[:, 0], X[:, 1], X[:, 2], c=t, cmap=plt.cm.hot)
original_plot.view_init(10, -70)
original_plot.set_xlabel("$x_1$", fontsize=18)
original_plot.set_ylabel("$x_2$", fontsize=18)
original_plot.set_zlabel("$x_3$", fontsize=18)
original_plot.set_xlim(axes[0:2])
original_plot.set_ylim(axes[2:4])
original_plot.set_zlim(axes[4:6])

# Unroll using locally linear embedding
from sklearn.manifold import LocallyLinearEmbedding
lle = LocallyLinearEmbedding(n_components=2, n_neighbors=14, random_state=4)
X_reduced_LLE = lle.fit_transform(X)

unrolled_lle_plot = fig.add_subplot(222)
unrolled_lle_plot.set_title("Unrolled using LLE", fontsize=14)
unrolled_lle_plot.set_aspect('equal')
unrolled_lle_plot.scatter(X_reduced_LLE[:, 0], X_reduced_LLE[:, 1], c=t, cmap=plt.cm.hot)
unrolled_lle_plot.set_xlabel("$z_1$", fontsize=18)
unrolled_lle_plot.set_ylabel("$z_2$", fontsize=18)
unrolled_lle_plot.axis([-0.09, 0.05,  -0.09, 0.08])
unrolled_lle_plot.grid(True)

# Unroll using an isomap
from sklearn.manifold import Isomap
isomap = Isomap(n_components=2)
X_reduced_iso = isomap.fit_transform(X)

unrolled_iso_plot = fig.add_subplot(223)
unrolled_iso_plot.set_title("Unrolled using Isomapping", fontsize=14)
```

```
unrolled_iso_plot.set_aspect('equal')
unrolled_iso_plot.scatter(X_reduced_iso[:, 0], X_reduced_iso[:, 1], c=t, cmap=plt.cm.hot)
unrolled_iso_plot.set_xlabel("$z_1$", fontsize=18)
unrolled_iso_plot.set_ylabel("$z_2$", fontsize=18)
unrolled_iso_plot.axis([-60.0, 50.0,  -20.0, 20.0])
unrolled_iso_plot.grid(True)

# Unroll using kernel PCA (sigmoid kernel)
from sklearn.decomposition import KernelPCA
sig_pca = KernelPCA(n_components = 2, kernel="sigmoid", gamma=0.001, coef0=1)
X_reduced_sig_pca = sig_pca.fit_transform(X)

unrolled_pca_plot = fig.add_subplot(224)
unrolled_pca_plot.set_title("Unrolled using kernel PCA", fontsize=14)
unrolled_pca_plot.set_aspect('equal')
unrolled_pca_plot.scatter(X_reduced_sig_pca[:, 0], X_reduced_sig_pca[:, 1],
c=t, cmap=plt.cm.hot)
unrolled_pca_plot.set_xlabel("$z_1$", fontsize=18)
unrolled_pca_plot.set_ylabel("$z_2$", fontsize=18)
unrolled_pca_plot.axis([-0.3, 0.3,   -0.3, 0.3])
unrolled_pca_plot.grid(True)

plt.show()
```

The output is:



Source: Examples in Ch 8, Hands-On Machine Learning with Scikit-Learn and TensorFlow, Géron.

Notes from Hands-On Machine Learning with Scikit-Learn & TensorFlow

(Géron, 2017)

Chapter 1. Overview of learning types

- Learning types:
  - Supervised eg. *k*-nearest neighbours, regression, SVMs, decision trees, random forests, neural networks.
  - Unsupervised learning is useful for clustering, visualization, anomaly detection, dimensionality reduction, and association rule learning
  - Semi-supervised (usually a combination of supervised and unsupervised)
  - Reinforcement learning: the learning system (or agent) adapts its policies based on previous experience (rewards for successes and penalties for failures)
- Training can be:
  - batch: essentially the model is static after learning has finished
  - online: the model can evolve with incoming samples. A learning rate must be set to control how quickly the model adapts to the new data.
- Learning can be:
  - Instance based: the prediction for a new sample is based in its similarity to samples in the training dataset
  - Model based: the learning algorithm builds a model for predicting the output for new unseen samples.
- Dataset limitations:
  - It may not be representative of the samples you wish to generalize to
  - It may be too small or biased
  - It may not contain the necessary features, or contain irrelevant features
  - It may contain errors or missing values
- An attribute is a data type. A feature is the attribute plus its value
- Dimensionality reduction: aim is to simplify the data without losing too much information e.g. merge correlated features
- Feature extraction: aim is create new features with improved modelling power
- Feature selection: aim is to select the features with the best modelling power
- Overfitting occurs when the model is too complex relative to the amount or noisiness of the data
- Under-fitting occurs when the model is too simple to learn the underlying structure of the data
- Regularization: constraining a model to make it simpler and reduce the risk of overfitting.
- Hyperparameter is a parameter of the learning algorithm, not the model.
- Tuning hyperparameters is an important part of building a machine learning system:
  - Split the available data into training, test and validation sets
  - Use the training set to train multiple models with various hyperparameters
  - Use the validation set to select the best-performing model
  - Use the test set to estimate the generalization error of the selected model
- No free lunch theorem: if you make no assumptions about the data, no model is guaranteed to work better than others *i.e.* you need to evaluate them all. In practice you make assumptions about the data and evaluate a subset of models which may be appropriate based on your assumptions.

Chapter 2. End-to-end example

- For very large datasets, batch learning could be split across multiple servers using MapReduce
- Error estimates typically use the "distance" between two vectors e.g. the vector of predictions and the vector of target values. The l_k norm is:

$$\|\boldsymbol{v}\|_k = \left(|v_0|^k + |v_2|^k + \cdots + |v_n|^k\right)^{1/k}$$

  l_0 is the number of non-zero elements in the vector, l_1 is the sum of absolute errors (not MAE), l_2 is the sum of squared errors (not RMSE), and l_∞ is the maximum absolute value in the vector. The higher the norm index, the more it focusses on the larger values.

- Use python's "virtualenv" to create an isolated environment. When the environment is activated and you install a package, it is only installed in that environment, not globally, enabling you to create isolated workspaces. See page 41.
- Standard deviation is the square root of the variance
- If the data have been pre-processed, check if any values have been capped (ie. truncated to a given range) – these may need to be removed from the dataset so the algorithm doesn't learn to restrict its projections to the same range.
- Features may need to be scaled (so they have similar magnitudes) and transformed (e.g. transforming a distribution with a long tail to something that is closer to a normal distribution)
- Stratified sampling must be used to ensure the test set is representative of the true population *i.e.* the proportion of samples in each sub-group is the same as in the true population.
- Correlation matrices can help identify potential relationships
- Combinations of existing attributes may have greater predictive power than the existing attributes.
- Missing values can be treated using any of the following:
  - Delete all samples that have data missing for any attribute
  - Delete the attribute
  - Set missing values to some value eg. median of that attribute
- Text and categorical attributes need to be converted to numerical values. The numerical values should use one-hot encoding, so the learning algorithm doesn't weight categories by their numerical value. One-hot encoding generates a binary value for each possible category: the binary value is 0 for all categories except the category assigned to that sample. For example, if the categories are [low, medium, high], one-hot encoding generates 3 binary vectors (one element for each sample). If a given sample was assigned "low", the binary values would be [1, 0, 0] ie "low" is hot or on, but "medium" and "hot" are cold or off.
- Scikit-Learn consists of
  - estimators – function that can estimate something from the data. The main function is `fit()` which returns nothing (ie. it returns `self`).
  - transformers – estimators that can also transform the data. The main function is `transform()` which returns a transformed dataset. As a transformer is also an estimator, it must have a `fit()` function. You should use `fit_transform()` when possible.
  - predictors – main functions are `predict()` and `score()`

Note:

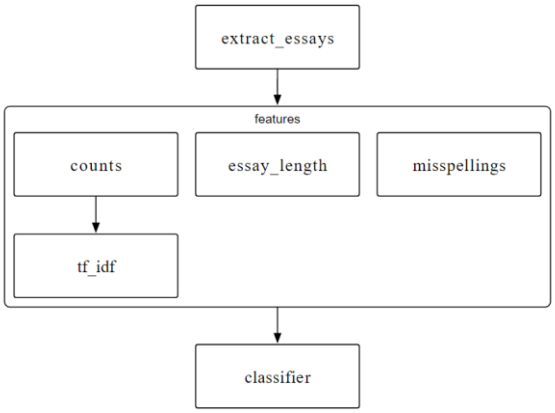
- `fit()` and `transform()` only take one parameter: the dataset to be used for training (or dataset + labels for supervised learning)
- `predict()` only takes one parameter: the dataset to be used for prediction

You should use inheritance to build on these three base classes, so your customised transformations easily fit into the pipeline.

- Features need to be scaled so they have similar scales. The target values usually doesn't need to be scaled. Scaling methods are:
  - min-max
    - data are scaled to span [0, 1]
    - adversely affected by outliers
  - standardisation (normalisation).
    - not greatly affected by outliers
    - not suitable in some cases as some algorithms require the data to span [0,1]
- Transformations should be done on the training data (eg. computing the mean and variance for standardisation) and the same transformations then applied to the test data. For example, if standardising the data, compute the mean and variance on the training set and standardise the training set, and then use the same mean and variance to standardise the test data.
- Scikit-Learn has a Pipeline class, where you specify a series of steps: all steps must be transformers (ie. have a `fit_transform()` function), but the last must be an estimator (ie. have a `fit()` function) , but it can also be a transformer.
- Pipelines can be chained together in serial and parallel. For example:

```
pipeline = Pipeline([
  ('extract_essays', EssayExractor()),
      ('features', FeatureUnion([
            ('ngram_tf_idf', Pipeline([
                  ('counts', CountVectorizer()),
                  ('tf_idf', TfidfTransformer())
            ])),
            ('essay_length', LengthTransformer()),
            ('misspellings', MispellingCountTransformer())
      ])),
  ('classifier', MultinomialNB())
])
```

achieves this workflow:



Source: http://zacstewart.com/2014/08/05/pipelines-of-featureunions-of-pipelines.html

- When tuning a model, you can split the training data into a smaller training set and a validation set. A better approach is *k*-fold cross validation: the training data are split into *k* "folds" and cross validation performed *k* times. Each time the model is model is evaluated using one fold after being trained on the remaining *(k-1)* data folds. The variance of the *k* cross validation scores also indicates how precise the error estimate is.
- If the RMSE on the training set is much lower than on the validation set, the model is over-fitting the data
- Random forests are an ensemble method: they train many decision trees on random subsets of features.
- A grid search or randomized search can be used to search parameter space for optimal values of the hyperparameters.
- After a grid or random search, you should examine which features are important

Chapter 3. Classification

- Some learning algorithms are sensitive to the order of the training samples – they may perform poorly if they get too many similar instances in a row. In most cases you should shuffle the data (except time-series data) and also stratify.
- A stochastic gradient descent classifier can handle large datasets as it trains on one sample at a time – well suited to online learning.
- Evaluating classifiers is more difficult than evaluating regressors
- Prediction accuracy (number of correct predictions / total predictions) is a poor performance measure, especially when the dataset is skewed *i.e.* some classes are more common than others (e.g. the performance may appear to be high simply because the target class is more common than the rest)

Binary classifiers

- Confusion matrix shows actual vs predicted results:

		Predictions	
		Negative	*Positive*
Actual	*Negative*	True negative	False positive
	Positive	False negative	True positive

↑Precision is proportion of positive predictions that are correct

← Recall is proportion of positives that were predicted correctly

- Precision (how many of your positive predictions were actually correct):

$$\frac{TP}{TP + FP}$$

- Recall (how many of the positives do you actually predict correctly):

$$\frac{TP}{TP + FN}$$

- F_1 score is a harmonic mean of precision and recall. It is high only when both precision and recall are high:

$$\frac{TP}{TP + \frac{FN + FP}{2}}$$

- F_1 score favours classifiers that have similar precision and recall, which may not always be the case (depends on the application).
- There is a trade-off between precision and recall. To view the trade-off, plot recall vs precision
- For binary classification, a given test sample is assigned a score. If the score is above a fixed threshold it is assigned to the positive class; otherwise it is assigned to the negative class.
- To select the threshold, plot the recall vs threshold and precision vs threshold curves and select the threshold which provides the desired recall or precision
- Performance can also be assessed using a Receiver Operating Characteristic (ROC) curve, which shows false positive rate vs true positive rate.
- The area under the ROC curve is useful for comparing algorithms. A perfect classifier has false positive rate = 0 and true positive rate = 1, so area under the curve = 1. A random classifier has false positive rate = true positive rate, so area under the curve = 0.5.
- The curve to use depends on the application
    - Precision vs recall curve is preferred when the positive class is rare, or when false positives are more important than false negatives
    - ROC curve is preferred otherwise

Multiclass classifiers

- Some classifiers (eg. Random Forest, naïve Bayes) can handle multiple classes
- Binary classifiers (e.g. SVMs, linear classifiers, Stochastic Gradient Descent classifiers) cannot handle multiple classes directly. To handle multiple classes, there are two options:
    - One vs all strategy: given *k* classes, the evaluation is done k times, in each iteration evaluating a single class against all the remaining classes *i.e.* evaluate class 1 against classes 2…k, evaluate class 2 against class 1 and classes 3…k, etc.
    - One vs one strategy: given *k* classes, the evaluation is done $k(k-1)/2$ times, in each iteration evaluating a single class against one other class e.g. evaluate class 1 against classes 2, evaluate class 1 against class 3 etc.

      If the algorithm scales poorly with the size of the dataset (e.g. SVM) it may be better to use the one vs one strategy as it uses a smaller dataset (*i.e.* it only uses data from the two classes, not data from all classes)

- A confusion matrix can be used to identify errors in multiclass classification.
- Stochastic Gradient Descent is a simple linear model
- Multi-label classification: where the classifier outputs more than one binary label. For example, a system trained to recognise Alice, Bob and Charlie in a photograph, it might identify Alice=yes, Bob=no, Charlie=yes.
- Not all classifiers support multi-label classification (k-neighbours does)
- Multi-output classification: where the classifier outputs more than one label, and the label can have more than one class. For example, a system for modifying a photo can be multi-output: one label for each pixel, where each label can have multiple values (0-255 intensity).

Chapter 4. Training models

- Linear regression models:

$$\hat{y} = \theta_0 + \theta_1 x_1 + \cdots + \theta_n x_n$$
$$= h_\theta(\boldsymbol{x}) = \theta^T . \boldsymbol{x_i}$$

can be trained using an analytic method:

$$\hat{\theta} = (\boldsymbol{X}^T . \boldsymbol{X})^{-1} . \boldsymbol{X}^T . \boldsymbol{Y}$$

which minimises the cost function:

$$MSE(\boldsymbol{X}, h_\theta) = \frac{1}{m} \sum_{i=1}^{m} (\theta^T . \boldsymbol{x_i} - y_i)^2$$

where the training data $\boldsymbol{X}$ contains m samples each having n features.
This method becomes slow as the number of features gets large

- In general a closed form expression for updating the parameters will not be available, so gradient descent methods will be required. Gradient descent methods use partial derivatives of the cost function to iteratively improve the parameters θ_j. The parameters are updated:

$$\theta_j^{next\ step} = \theta_j - \eta . \frac{\partial}{\partial \theta_j}$$

where η is the learning rate. There are three approaches:

a) Batch Gradient Descent:
The parameters θ_j are updated using the entire dataset. For linear regression:

$$\frac{\partial}{\partial \theta_j} MSE(\boldsymbol{X}, h_\theta) = \frac{2}{m} \sum_{i=1}^{m} (\theta^T . \boldsymbol{x_i} - y_i)\, x_j^i$$

for a single parameters θ_j, or for all parameters:

$$\nabla_\theta MSE(\theta) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} MSE(\theta) \\ \frac{\partial}{\partial \theta_1} MSE(\theta) \\ \vdots \\ \frac{\partial}{\partial \theta_n} MSE(\theta) \end{pmatrix} = \frac{2}{m} \boldsymbol{X}^T . (\boldsymbol{X} . \boldsymbol{\theta} - \boldsymbol{y})$$

Notes:

- The entire training set (m samples) is used
- Is slow when number of samples (m) is large
- Scales well with the number of features (n)
- Learning continues until the gradient vector is negligible (e.g. its norm)
- If the learning rate is too low convergence will be slow
- If the learning rate is too high the cost may oscillate or diverge
- Algorithm will reach the optimal solution (if it converges).

b) Stochastic Gradient Descent

Instead of computing the gradients based on the entire dataset, this method randomly selects a single instance and computes the gradients $\partial/\partial\theta_j$ based on that single instance.

Notes:

- Inputs must be scaled (if not the cost function will be elongated in the direction of features with smaller values, so training will be slower for those features)
- In each iteration only a single sample is used
- Is fast even when the number of samples (m) and the number of features (n) are large
- The cost function will not smoothly decrease, but will decrease on average
- Algorithm will reach a good solution, but not the optimal solution (if it converges) as the cost bounces around
- Well suited to cases where the cost function has local minima, as the randomness helps it escape local minima
- A simulated annealing method can be used to find the global minimum: the learning rate is gradually decreased.

c) Mini-batch Gradient Descent
A hybrid of batch and stochastic methods: the gradients are computed on small random subsets of instances. The convergence is less erratic than the pure stochastic method.

When applied to linear regression, gradient descent methods are guaranteed to converge to the global minimum cost because the MSE cost function for linear regression is a convex function *i.e.* no local minima.

- Gradient descent methods are widely used for finding the parameters θ_j that minimise the cost function for arbitrary models *i.e.* they can be used on a wide range of models, not just linear regression.
- Polynomial regression: using linear regression methods applied to polynomial powers of the original data:
  - The original data may have features a and b. By using polynomial powers of a and b, we can use linear techniques to fit non-linear data, and also find relationships between the features e.g. the original data may not be a linear combination of a and b, but they may be will fitted by a polynomial in a and b:
    $$a + b + a^2 + b^2 + ab + a^2b + ab^2 + a^3 + b^3$$
  - High degree polynomials may overfit the training data
- Learning curves
  - If a model performs well on the training data but generalizes poorly, it may be over-fitting the training data (model is too complex – it needs to be regularized)
  - If a model performs poorly on both training and test data, it is under-fitting (the model is too simple or you need more data or more features)
  - Learning curves plot the error on the training set and validation set as a function of training set size:
    - If both curves are close and plateau, the model is under-fitting the data
    - If there is a gap between the curves (results on training set is significantly better than the validation set), the model is over-fitting the data

- A model's generalization error is the sum of:
    - Bias (due to incorrect assumptions e.g. assuming the data are linear). High bias indicates under-fitting
    - Variance (the model is highly sensitive to input data, or has too many degrees of freedom). High variance indicates over-fitting.
    - Irreducible error (due to noise in the data, so it can only be reduced by improving the data quality)
- Bias/variance trade-off: as model complexity increases, the bias usually decreases but the variance usually increases.
- Regularisation methods used to constrain linear regression:
    a) Ridge regression: a regularisation term:

$$\alpha \sum_{i=1}^{n} \theta_i^2$$

    is added to the cost function to minimise the weights. α is a hyperparameter that controls the amount of regularisation: as α increases the regularisation becomes stronger.

    Note:
        - the bias term ($\theta_{i=0}$) is not regularised (it is the bias or "intercept" in the linear model).
        - Input features should be scaled

    b) Least Absolute Shrinkage and Selection Operator (Lasso) regression uses a regularisation term:

$$\alpha \sum_{i=1}^{n} |\theta_i|$$

    Lasso tends to eliminate the weights for the least important features.

    c) Elastic Net regression is a combination of Ridge and Lasso regularisation, controlled by a mixing ratio r. The total cost function is:

$$MSE(\theta) = r\,\alpha \sum_{i=1}^{n} |\theta_i| + \frac{1-r}{2}\,\alpha \sum_{i=1}^{n} \theta_i^2$$

    Note:

        - Ridge regularisation is a good default
        - Lasso can be unstable if the number of features exceeds the number of training samples, or several features are highly correlated.
        - Elastic Net is good if you suspect some features may not be important.
- Training is usually run until the error on the training set is minimised. *Early stopping* is an alternative where training stops when the error on the validation stop stops decreasing. The validation error usually drops as the model is fit to the data, but then increases when over-fitting starts to occur.
- Logistic regression: using a linear regression model for **binary classification**. The output is the probability that a sample belongs to a given class:

$$\hat{p} = h_\theta(\boldsymbol{x}) = \sigma(\theta^T \cdot \boldsymbol{x})$$

    where the sigmoid function is:

$$\sigma(t) = \frac{1}{1 + e^{-t}}$$

- Logistic regression can be trained using a cost function that is high if the training sample is classified incorrectly and low if the classification is correct. The cost function:

$$c(\theta) = \begin{cases} -\log(\hat{p}), & \text{if } y = 1 \\ -\log(1 - \hat{p}), & \text{if } y = 0 \end{cases}$$

  is convex so gradient descent can be used to find the global minimum.
- The decision boundary is the threshold between classifying a sample into two different classes. This can be plotted by computing the probability of a set of samples where you have set the input vector ranging over the domain of interest.
- Softmax or Multinomial Logistic regression uses a linear regression model for classification between **multiple classes**. For a given sample, you compute the probability of the sample belonging to each class. The sample is assigned to the class with the highest probability.
- Cross entropy is used for training Multinomial Logistic regression. Cross entropy measures how well the estimated class probabilities match the target classes. The cost function is:

$$J(\Theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^i \log\left(\hat{p}_k^i\right)$$

  where there are K classes, y_k^i is 1 if sample i belongs to class k (and 0 otherwise), $\hat{p}_k^i$ is the probability that sample i belongs to class k, and Θ is the matrix of parameters (has one row θ_k for each class k).

Overview of binary/multiclass classification:

	Binary classification • Algorithm predicts if an instances belongs to a class or not (there is only one class to choose from)	*Multiclass classification* • Algorithm predicts the class an instance belongs to • Algorithm can choose from more than one class • Classes are mutually exclusive
Prediction	Logistic regression: • compute the weighted sum of the inputs+bias • use the logistic (sigmoid) function to compute the probability of the instance belong to the class. This is called the *logit.* (Page 136).	Softmax regression: • for each possible class, compute the weighted sum of the inputs+bias. • Use the softmax function to compute the probability of the instance belonging to each class. • Instance is assigned to the class with the highest probability (Page 142).
Training	Log-loss: • cost is high when the logit (*i.e.* probability) is low for positive instances, and vice versa. Page 137.	Cross entropy: • cost is high when the target class has low probability. Page 143.

- The learning algorithm will usually be fitted using an error function that is different to the performance measure used to evaluate the final model. For example, when using a grid search to locate the best parameters for a random forest classifier:

1. The grid (or random search) tries many different models, choosing a different set of parameters for each model. The models are ranked based on a score computed for each model. The score may be computed using cross validation. The scoring method is some form of accuracy estimate, such as Jaccard Similarity for classification, negative mean square error for regression etc. The scoring method will be an argument of the search (not the model).
2. Each individual model tested during the grid/random search is fitted by optimising an error metric for that model. For example, with a random forest model the splits may be chosen to minimise the Gini impurity. The metric to optimise will be an argument of the model (not the grid search).
3. The best model found during the grid/random search may also provide an "Out-of-Bag" score. This is the accuracy of the OOB predictions for that single model. Even if the methods used for scoring in #1 is the same as the method used for assessing accuracy of the OOB predictions, the values may differ (i.e the score from #1 may differ from the OOB score in #3), because they will almost certainly have been computed on different subsets. The search (#1) will have used its own cross validation subsets (via pasting), while the fitting of the individual model will have used different subsets (via bagging).

- Out-of-bag evaluation is desirable because OOB instances are not seen during training, so they can be used for evaluation. Consequently the original dataset does not need to be split into training and test sets. However in the above case, a separate test set is necessary: all of the training data were used when searching for the best parameters. Even though the OOB score for the best model was computed without seeing the training data, the parameters used for selecting that model were based on all the training data. Consequently a completely independent test set is required (obtained by splitting the original set into test and train sets with the search method only accessing the training test).
- Contamination (notes taken from https://rapidminer.com/blog/validate-models-training-test-error/):
    - Training error: predictive error from predictions made on the training data (*i.e.* same data that was used to train the model)
    - Test error: predictive error from predictions made on test data *i.e.* data not used in training.
    - Training error should never be used to infer how a model will perform on unseen data.
    - The best way to assess test error is *k*-fold cross validation:
        - All data can be used for training
        - No contamination between test and training datasets
        - The *k* error estimates provide you with a mean error and also standard deviation of the error, indicating the likely range of model accuracy when put into production.
    - It is critical that information about the test data does not leak into the training data. If it does, the test error will be artificially low. Examples showing how contamination can occur:
        - Preprocessing the entire dataset prior to cross validation:
          For example, if the data are normalised prior to cross validation, information about the range of the samples in the held-out dataset is carried into the data used in training. The correct approach is:
          `for each fold (ie. in the k-fold CV):`

```
        normalise the training data and save the parameters (eg. range)
        fit the model
        normalise the test data, using the norm. params from the training set
        make predictions using the normalised test set
    compute the test error on the test set
compute the mean CV error and standard deviation
```

- Parameter optimisation (including feature selection):
  It is quite common to use a grid or random search to compare a range of algorithms or parameters for a given algorithm. The two most common approaches are:
  - Separate the data into training, test and validation sets. The test set is used compare a range of models that were trained using the training set. The performance of the best model (as determined from the model with the lowest test error) is then evaluated using the validation set. The problem with this approach is it severely reduces the size of the training set.
  - Separate the data into training and validation sets. Cross validation is used to evaluate the various models, and the best model selected based on the model with the lowest cross validation error. This overcomes the need for a test dataset, however it still requires a validation dataset (thereby reducing the size of the training dataset).

  The best approach is to use cross validation on the outer "parameter optimisation step", so the validation dataset is not required and all data can be used for training. Note: cross validation is also used on the inner step – evaluating each individual model. There are now inner and outer processes:

  - Inner cross validation: evaluate the accuracy of the individual models and thereby choose the best one
  - Outer cross validation: to validate the best model chosen

Chapter 5. Support vector machines

- Powerful methods for linear and non-linear classification, regression and outlier detection
- Suited to small-medium datasets that are complex
- Sensitive to feature scale, so training data need to be scaled
- May also be necessary to centre the data (*i.e.* so each feature has mean zero)
- Can have soft or hard margins
- Approaches to classification:
  1. Linear e.g. $ax_1 + bx_2$
  2. Non-linear: linear SVM with polynomial features e.g. $ax_1 + bx_2 + cx_1^2 + dx_2^2 + ex_1x_2 + \cdots$
  3. Manually add similarity features eg. how similar samples are to selected *landmarks*
  4. Use non-linear kernels
- Classification: the aim is to find the largest margin separation between classes (with minimal samples between the support vectors *i.e.* margin violations)
- Regression: the aim is the opposite, to have the maximum number of samples lie between the support vectors (in this case margin violations are those samples that don't lie within the support vectors).

Chapter 6. Decision trees

- Powerful methods for fitting complex datasets
- Used for classification, regression and multi-output regression
- Don't require scaling or centring
- Requires little data preparation, however:
    - the data may need to be rotated as the algorithm splits data perpendicular to the feature axis being used for the split
    - the algorithm can be sensitive to small variations in the data.
- Can output the probability of a given sample lying in a given class
- The tree is constructed using the Classification and Regression Tree (CART) algorithm.
- Gini impurity tends to isolate the most common class in its own branch of the tree, while entropy leads to more balanced trees.
- CART is a greedy algorithm: it finds the best split at each level, but it does not check if the split leads the lowest impurity at levels further down. Consequently the resulting tree may not be optimal overall.
- Decision trees are "non-parametric" because the number of parameters is not determined prior to training (there can be a lot of parameters eg. the minimum number of instances to be assigned to a leaf node). Without proper constraints decision trees can tightly fit the training data.
- Many other algorithms are parametric in that some parameters are fixed (e.g. polynomial degree), so the degree of freedom can be limited, thereby reducing the risk of overfitting (and increasing the risk of underfitting)
- Regularization is usually needed to prevent overfitting.

Chapter 7. Ensemble learning and random forests

- Ensemble methods combine the results from multiple *weak* learners to create a stronger learner.
- Ensemble methods are powerful if there are a sufficient number of weak learners and they are diverse. Diversity can be obtained by:
    - Use different algorithms eg. one weak learner could be SVM, another could be a decision tree etc.
    - Train the various learners on different features
    - Train the various learners on different subsets of the training data
- The ensemble output can be obtained from:
    - Hard voting: output (for classification) is simply the class that gets the most votes from the individual learners
    - Soft voting: the output is the class with the highest probability (but requires all the individual learners to provide a probability estimate)
    - Stacking: another learning algorithm is stacked on top of the weak learners. For a given sample it takes the output from all the weak learners and make a prediction.

      Soft voting can outperform hard voting in some cases because it gives greater weight to prediction made with greater confidence.

- The ensemble output typically has similar bias to the individual learners, but lower variance.

- The training dataset can be split into subsets for use by the individual learners. The split can be done by samples, features, or both:
  - By sample: For each individual learner, a subset is created by randomly *selecting training samples* from the full dataset. There are two approaches:
    - bagging or bootstrap aggregation: samples are randomly selected with replacement. Consequently, in a given subset any given sample may appear more than once.
    - Pasting: sampling is done without replacement.

    Note: these approaches are similar to *k*-fold cross validation used for evaluating a single algorithm (see notes above)

  - By feature: For each individual learner, a subset is created by randomly *selecting features* from all samples in the full dataset. Features can be randomly selected using either bagging or pasting. This method is called *random subspaces*.
  - By sample and feature: sampling by both instances and features. This method is called *random patches*.
- Evaluation approaches:
  - use cross validation to compare results from bagging and pasting
  - out of bag (OOB) evaluation: the subset obtained by bagging will typically only contain 63% of the unique samples in the full training set (if the size of the subset obtained by bagging is the same size as the full training set)[2]. Consequently the weak learners can use the remaining 37% of the date for evaluation as they have not seen these samples before. Note: as the bagging subsets are randomly selected for each weak learner, the instances used for training and evaluation are different for each weak learner.
- Random forests use feature bagging. Decision trees take input samples that contain the full feature set and at each stage in constructing the tree, they split a node by searching for the best feature to use for making the split. Random forests introduce additional diversity by using an ensemble of decision trees: each decision tree is trained using a training set where the samples only contain a randomly selected *subset of the features*.

  In both decision trees and random forests, the feature and threshold used for splitting each node is chosen to minimise the impurity (see notes above)
- Extremely randomized trees (Extra-trees) are similar to random forests, but the threshold using for splitting a node is randomly selected.
- Extra-trees are faster to train than random forests
- The only way to compare performance of extra-trees and random forests is to test them using cross validation and grid search for tuning the hyperparameters.
- A feature's importance is indicated by how much the impurity is reduced by that feature.
- Boosting is an ensemble method where each successive learner is directed to focus on samples that were misclassified in the previous learner. AdaBoost and Gradient boosting are the two most popular boosting methods.
- Stochastic gradient boosting is the same as gradient boosting except that instead of using the full training set in each successive round, a randomly selected training subset is used in each successive round.

[2] As the size of the dataset increases, the fraction approaches $1 - e^{-1} \approx 63\%$.

- Stacking can be used instead of hard or soft voting to combine the results from an ensemble of weak learners. The final predictor (or *blender*) is another learner that takes the inputs from the individual weak learners to produce the final output.
- Multilayer stacking takes stacking to a second level: multiple blenders are trained and a second level blender is used to blend the outputs of the first level blenders.

Chapter 8. Dimensionality reduction

- Aim is to make an intractable problem tractable
- Reducing dimensionality may filter out noise and unnecessary details, thereby improving performance. However in general is doesn't improve performance, it simply speeds up training.
- In high dimensional spaces:
    - Most instances are close to the border
    - The distance between instances increases with dimensionality

    Therefore:

    - Datasets are sparse
    - The sample to be predicted is unlikely to be near an training instance, so predictions are less reliable
    - Higher risk of overfitting
- The number of training instances required to reach a given density grows exponentially with the number of dimensions. It is therefore not realistic to seek dense datasets in high dimensions.
- Common assumptions:
    - Some features are constant, while others are highly correlated. Consequently training instances may lie within a lower dimensional subspace of the original high-dimensional space
    - Prediction will be simpler in the lower dimensional space

    Note: these assumptions do not apply in some cases.

- Two main approaches:
    - Projection
      The aim is to project the training data onto a reduced number of axes.

    - Manifold learning.
      The aim is to model the manifold on which the training instances lie, where it is assumed that real-world high-dimensional datasets lie close to a much lower dimensional manifold.

      A d-dimensional manifold is part of an n-dimensional space ($d < n$) that locally resembles a d-dimensional hyperplane.
- Principal Component Analysis:
    - Successively identify a series of axes which explain the largest proportion of the variance not explained by the previously identified axes:
        1. Identify the 1st axis which explains most the variance

2. Identify the 2nd axis (orthogonal to the 1st axis) which explains most the remaining variance
3. … further axes are progressively identified.

- Singular Value Decomposition is used to decompose the matrix of training samples:

$$\boldsymbol{X} = \boldsymbol{U} \cdot \boldsymbol{\Sigma} \cdot \boldsymbol{V}^T$$

where the principal components are in the columns of $\boldsymbol{V}$.

- Data must be centred before performing PCA
- The training data are projected onto the d-dimensional hyperplane:

$$\boldsymbol{X}_{d-proj} = \boldsymbol{X} \cdot \boldsymbol{W}_d$$

where $\boldsymbol{W}_d$ contains the first d columns of $\boldsymbol{V}$.

- The number of dimensions d can be chose:
  - To explain a desired fraction of variance (*i.e.* keep adding dimensions until you have explained x% of the variance)
  - Plot variance explained vs number of dimensions, and then select the optimum number of dimensions.
- PCA is approximately reversible (but there will be some loss as the original PCA lost some information by setting $d < n$). The reconstruction error is the mean squared distance between the original dataset and the reconstructed dataset.
- Incremental PCA splits the training set into subsets, each being processed sequentially:
  - Used for large datasets that don't fit into memory
  - Used for online training systems
- Randomized PCA is a stochastic method that finds approximations to the principal components. It is faster than traditional PCA.
- Kernel PCA uses a kernel to perform complex non-linear projections for dimensionality reduction:
  - The approach is similar that used in Support Vector Machines, where a linear decision boundary in high dimensional space to corresponds to a complex non-linear decision boundary in the original space.
  - It is often good at preserving clusters of instances after projection, and unrolling datasets that lie on a twisted manifold
  - Is an unsupervised method, so a grid search approach may be needed to find the optimal parameters (dimension d and choice of kernel) that results in the best performance. In other works, choose the kernel and d, fit the *k*PCA, estimate performance (eg. classification error), and then repeat until you find the optimal kernel and dimension.

- Manifold learning

  Locally Linear Embedding (LLE) is a powerful non-linear method that looks for a low-dimensional representation of the training set that preserves the distances between the instances in the original space.

  Fitting is one in a two step process:

  1. For each training instance $\boldsymbol{x}_i$, it identifies the k nearest neighbours, and computes weights $w_{i,j}$ that minimise the squared distance between the instance $\boldsymbol{x}_i$ and an approximate reconstruction of $\boldsymbol{x}_i$ through linear combination of the k neighbours:

$$\sum_{j=1}^{m} w_{i,j}\boldsymbol{x}_j$$

where the weights are zero for those instances not in the list of k nearest neighbours. The matrix of weights is obtained by solving the constrained minimisation problem:

$$\widehat{\boldsymbol{W}} = \underset{\boldsymbol{W}}{\operatorname{argmin}} \sum_{i=1}^{m} \left(\boldsymbol{x}_i - \sum_{j=1}^{m} w_{i,j}\boldsymbol{x}_j \right)^2$$

subject to:

$$\begin{cases} w_{i,j} = 0, & \text{if } \boldsymbol{x}_{i,j} \text{ is not in the } k - \text{nearest neighbours of } \boldsymbol{x}_i \\ \sum_{j=1}^{m} w_{i,j} = 1 \text{ for i} = 1,2,\dots,m \end{cases}$$

2. Map the n-dimensional instances $\boldsymbol{x}_i$ into d-dimensional instances $\boldsymbol{z}_i$ by solving the unconstrained minimisation problem: (where the weights are held fixed):

$$\widehat{\boldsymbol{Z}} = \underset{\boldsymbol{Z}}{\operatorname{argmin}} \sum_{i=1}^{m} \left(\boldsymbol{z}_i - \sum_{j=1}^{m} w_{i,j}\boldsymbol{z}_j \right)^2$$

Note: LLE scales poorly with the size of the dataset, but can unwrap complex manifolds.

- Other methods:
  Other popular methods include:
  - Multidimensional scaling: aim is to preserve the distance between instances
  - Isomap: creates a graph connecting each instances to its nearest neighbours, and tries to reduce dimensionality while preserving the geodesic[3] distance between instances
  - t-distributed stochastic neighbour embedding: aim is to keep "similar" instances close and dissimilar instances apart (mainly used for visualisation)
  - Linear Discriminant Analysis: aim is to learn the most descriptive axes and then project the data onto a hyperplane defined by those axes.

Chapter 9. Introduction to Tensorflow

- Has a Python API called TF.Learn which is compatible with Scikit-learn.
- Has an API called TF.slim for building simple networks.
- Has a C++ API for performance critical operations
- Tensorboard can be used to visualise networks and also display model results
- Overall process consists of 2 phases:
  1. Construction (building the computation graph)
  2. Execution (running it)

  Steps are:

  1. Define variables and nodes
  2. Build the graph

[3] Number of nodes along the shortest path between two nodes.

3. Create a session
    a) Initialise the variables
    b) Evaluate the graph

- You can have multiple graphs. New nodes are added to the graph that is currently the default
- You can have multiple sessions. In single process TF, the multiple sessions do not share any state. In distributed TF, the various sessions can share variables.
- Variables:
    - Lifetime is from when the variable is initiliased to when the session is closed
    - Value is maintained across graph runs
- All other nodes:
    - Value is NOT maintained across graph runs
- When a node is evaluated, it will determine what nodes it depends upon and evaluate those nodes in the order of dependency. It will not reuse the results from previous evaluations. If you wish to evaluate multiple nodes that have common dependencies, you can:
    - evaluate each node sequentially – however the dependent nodes will be evaluated multiple times (as the results from previous evaluations are not reused, so the dependent nodes are re-evaluated for each parent node)
    - run the session, and flag which nodes can be evaluated in a single graph run
- Placeholder nodes don't do any computation – they simply output the requested data for use by the connected node. A given node can obtain input (*i.e.* be "fed" data) from:
    - A placeholder node
    - Any operation
- Node types:
    - X = tf.constant(…) value is constant and initialised by call to init()
    - y = tf.Variable(…) value is not constant and initialised by call to init()
    - error = y_pred – y evaluated at run time by call to eval()
- Large models should be periodically saved so training can be resumed after a crash, or the training results can be loaded. Both the graph and the graph's state (ie. variable values) can be saved and restored.
- To use Tensorboard, start it using

  `tensorboard –logdir <directory_containing_logfiles>`

  and then view the results in your browser at http://localhost:6006 (or http://0.0.0.0:6006)
- To improve clarity, related nodes can be grouped inside a namescope.
- Variables can be shared in various ways:
    - Define the shared variable first, and then use as required. This is not practical when there is a large number of shared variables
    - Store all shared variables in a dictionary, and pass the dictionary around as required
    - Use python classes, with class variables used to hold the shared parameter
    - If the variable is used in a function within a namespace, use the `hasattr()` function to determine if the variable already exists:
        - The first time the function is called, `hasattr()` returns false, so the variable is created.
        - In all subsequent calls to the function, returns true so the pre-existing variable is re- used
    - TensorFlow's variable scope method:
        - The variable is created in a variable scope

        - Whenever the shared variable is required:
            - Switch to the variable scope in which the variable was first defined
            - "Retrieve" the variable using `get_variable()` and then use as normal
- In numpy, a 1D array is neither a row nor a column vector *i.e.* $\boldsymbol{a} = \boldsymbol{a}^T$ if $\boldsymbol{a}$ is a 1D vector. This means you can take the dot product of $\boldsymbol{a}$ with itself, without transposing the second argument.
- When reloading a previously-saved model:
    - You may need to access the operations and tensors in the model. Operations can be recalled using the `get_operation_by_name()` function. Tensors can be recalled using the `get_tensor_by_name()` function, where the tensor name is the name of the operation that outputs it, followed by a digit indicating which output it is e.g. `X:0` if you are seeking the first output from operation `X`.
    - To easily recall all the operations and tensors that are needed, you should add them to a collection when writing out the model. They can then be easily recalled using the `get_collection()` function.
- To reload a previously-saved model (e.g. to continue training or to adapt the trained model to a new task), you can do it two ways:
    - If you have the python code that built the original graph, to resume training all you need to do is restore the model state:

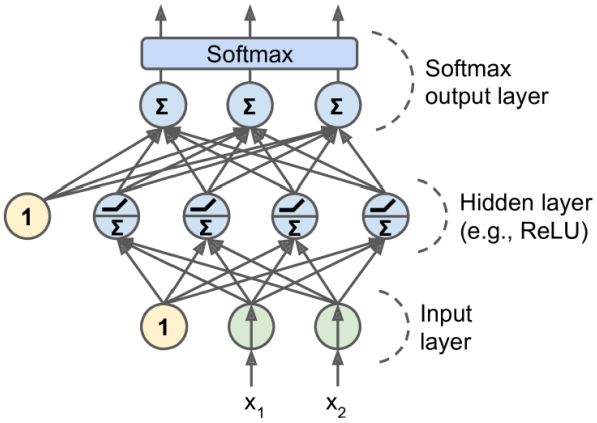
```
… existing Python code builds the graph…
saver.restore(sess, "./name_of_checkpoint_file.ckpt") – restores the state
…continue training…
```

    - If you don't have the original python code, you need to restore the graph and model state:

```
saver = tf.train.import_meta_graph("./name_of_checkpoint_file.ckpt.meta")
saver.restore(sess, "./name_of_checkpoint_file.ckpt")
…continue training…
```

Chapter 10. Introduction to Artificial Neural Networks

- An artificial neuron has one or more binary inputs and one binary output
- A perceptron (Linear Threshold Unit) passes weighted inputs into a threshold function to produce a binary output
- Logistic regression is preferred over perceptrons for classification because perceptrons output a prediction based on a hard threshold; logistic regression outputs a class probability so soft voting can be used.
- Perceptrons are linear classifiers. The limitations of linear classifiers can be overcome by stacking multiple perceptrons together.
- Backpropagation is the same as reverse mode auto diff (automatic differentiation).
- A multi-layer perceptron has an input (pass through) layer, one or more layers of linear translation units (hidden layers) and an output layer of LTU's. The image below illustrates a typical feedforward network:



Taken from Hands-On Machine Learning with Scikit-Learn & TensorFlow, Ch 10.

Notes:

- A deep neural network has two or more hidden layers
- When used for classification, there is usually one output LTU for each possible class. This enables multi-class classification.
- If the classes are mutually exclusive, the outputs can be passed through a softmax function to provide a probability for each class.
- Every layer except the output layer has a bias neuron. In a given layer, the bias + the weighted inputs are passed as input to each neuron in the next layer.

- A network can be built using the helper functions in the high-level API TF.learn. For more control you can tailor a network using the lower level native TensorFlow API.
- The weights between each input and and each neuron should be initialised randomly using a truncated normal distribution:
  - The weights must be selected randomly to avoid any symmetries that would call the gradient descent algorithm to fail.
  - The weights matrix is sometimes called the kernel.
  - The distribution must be truncated to ensure there are no large weights which would slow down training
  - The normal distribution should have a standard deviation of:

    $$2 \Big/ \sqrt{n_{inputs} + n_{neurons}}$$

    to improve the rate of convergence.
- A part of a graph can be evaluated using either `Session.run` on a list of variables or `Tensor.eval`:
  - Calling `t.eval()` is equivalent to calling `tf.get_default_session().run(t)` where `t` is a tensor.
  - each call to `eval()` and `run()` will execute the whole graph from scratch.
  - to cache the result of a computation, assign it to a `tf.Variable`.
- A multilayer perceptron with just one hidden layer can model very complex functions, provided it has neurons.
- Deep networks have higher parameter efficiency: they can model complex functions with exponentially fewer neurons, making them faster to train.
- Deep networks are successful because they model the hierarchical structure commonly seen in real datasets. The high level layers of the network (*i.e.* those near the output layer) model the high level structure, with the lower layers (*i.e.* those near the input layer) modelling the

finer details. This also assists in generalizing to new datasets: instead of starting the training from scratch, you can start by reusing the weights and biases from the original network: your new problem may share some of the fine structure of the original problem, so you may be able to reuse the weights and biases for the lower layers, an only need to start training from scratch for the higher layers (which need to learn the higher level structure of the new problem).

- For most problems, start with 1 or 2 hidden layers and gradually increase the number of layers until you start overfitting the data.
- Very deep networks (with dozens of layers) usually aren't fully connected and require huge amounts of training data. However you usually don't train from scratch: you reuse parts of a pretrained network that performs a similar task.
- Number of neurons in each layer:
    - Input layer: number if fixed by the number of input features
    - Output layer: number is fixed by the number of outputs
    - Hidden layers: lower layers near the input require more neurons (to model the fine structure), progressively decreasing in number for the higher layers near the output (to model the high-level structure). You should start with a small number of neurons and gradually increase the number of neurons until you start overfitting the data.
- In general you will get better performance by adding layers rather than adding neurons.
- Activation functions:
    - Hidden layers: ReLU works well and is fast. Because it is unbounded [0,max] it doesn't saturate like the sigmoidal functions, so gradient descent works better with large input values
    - Output layer: softmax is used for mutually exclusive classes. For binary tasks or when the classes are not mutually exclusive, use the logistic function.

Chapter 11. Training deep Neural Networks

- Deep networks require a long time and a lot of data to train. To improve performance:
    - Use a good initialisation strategy
    - Use a good activation function
    - Use Batch Normalisation
    - Reuse parts of an existing network
    - Use an optimiser that is faster than gradient descent
- Key problems when training deep networks:
    - Vanishing or exploding gradients
    - Time and data required to train them
    - They have millions of parameters, increasing the risk of overfitting.
- Exploding gradients is mostly encountered in recurrent neural networks.
- Vanishing gradients is encountered when the gradients progressively get smaller as the back propagation algorithm progresses down to the lower layers. As a consequence the connection weights for the lower layers never change, so training never converges. The problem arises because a sigmoidal activation function saturates at the upper and lower ends and the gradient is almost zero. Furthermore the gradient gets further diluted as it propagated down to the lower layers.
- Vanishing gradients can be alleviated by correct initialisation, enabling the signal to flow forward (when making predictions) and backwards (when back propagating gradients). For the signal to flow properly, we need:

    - Variance of the output of each layer to be equal to the variance of its inputs
    - Gradients before entering a layer to have equal variance to the gradients after leaving the layer (during back propagation)

  There are now standard methods for initialising weights from both uniform and normal distributions, for use with various activation functions. Using these methods can speed up training considerably.

- Activation functions for deep networks (in order of preference, best is last):
    - Sigmoidal – not favoured as they saturate and result in vanishing gradients.
    - ReLU – better than sigmoidal as they don't saturate, but they *die* (especially with large learning rates). If the weighted sum of inputs to ReLU is negative, the output is zero and the gradient is zero, so the unit usually doesn't come back to life.
    - Leaky ReLU – similar to ReLU but has a hyperparameter that controls the leakage: $LeakyReLU(z) - \max(\alpha z, z)$. The small slope ensures the gradient is non-zero for negative inputs, so Leaky ReLUs don't die
    - Exponential linear units (ELU) – it is smooth everywhere (so gradient descent doesn't bounce around near $z = 0$, and it can output negative values (so the average output is closer to zero, alleviating the vanishing gradients problem). It is defined by:
$$ELU_\alpha(z) = \begin{cases} \alpha(e^z - 1), & \text{if } z < 0 \\ z, & \text{if } z \geq 0 \end{cases}$$
    - Scaled exponential linear units (SELU) – outperforms the ELU. It is defined by:
$$SELU_\alpha(z) = \begin{cases} \alpha(e^z - \alpha), & \text{if } z \leq 0 \\ z, & \text{if } z > 0 \end{cases}$$
- Using the best activation and initialisation methods can reduce the problem of vanishing/exploding weights at the beginning of training, but they can back later duing training. This can be overcome using Batch normalisation.
- Batch normalisation overcomes the problems that the distribution of each layer's inputs changes during training. A pre-processing step is inserted before each layer's input to the activation function for that layer. The pre-processing:
    1. Centres and normalises the inputs
    2. Scales and shifts the output from 1, using defined shift and scaling parameters.

  While batch normalisation adds complexity and is slow, it has numerous advantages:
    - Strongly reduces the vanishing gradients, allowing a wide variety of activation functions to be used
    - Networks are less sensitive to weight initialisation
    - Larger learning rates can be used
    - Acts like a regulariser, reducing the need for other regularisation methods.
- Gradient clipping can be used to alleviate exploding gradients by simply clipping the gradients to a pre-defined range. Clipping has largely been replaced by batch normalisation.
- Transfer learning is the term used when an existing (trained) network is adapted to a new task. It only works if the original and new tasks have similar low level features, however:
    - It reduces the amount of training data required
    - Can dramatically reduce the training time
- Typically you will reuse only part of the original network (usually the lower layers), so you will reload the lower layers and weights from the original network, and then add the new higher level layers. To train the weights for the new layers, you can:

- Freeze the weights of the lowest layers, so they don't change. It will be easier and faster to train the weights for the remaining layers (the new layers and the highest 1 or 2 layers from the original model).
- Cache the output from the highest layer that is frozen. As the training goes through the entire dataset multiple times, you can save a lot of time by caching the output from the highest frozen layer during the first cycle, and then simply reusing this output in all subsequent cycles. (The output won't change because the weights are fixed.)

- Model *zoos* contain pre-trained models that you may be able to adapt to your task.
- If you cannot train a deep network on the target problem (for instance, you may not have enough training data, and/or you cannot adapt a trained model from a zoo), you may be able:
  - Train a network on a similar task for which you can easily generate or obtain training data, and then use this model to initialise your target model.
  - Use unsupervised pre-training. If you don't have a lot of labelled training data, but can access a lot of unlabelled training data, you may be able manually pre-train each layer using the output from the previous layer. Because the training data are not labelled, unsupervised feature detection methods (such as autoencoders or Restricted Boltzmann Machines) are used.
- Several optimisers are faster than Gradient Descent:
  - Momentum Optimisation:
    The correction applied to the weights (*i.e.* parameter vector θ) is a weighted combination of the previous adjustments (the momentum vector, $\boldsymbol{m}$) and the computed update (learning rate × gradient of the cost function):
    $$\boldsymbol{m} \leftarrow \beta \boldsymbol{m} - \eta \nabla_\theta J(\theta)$$
    $$\theta \leftarrow \theta + \boldsymbol{m}$$

    Notes:

    - It can escape plateaus faster than gradient descent
    - Improves speed when the inputs have very different scales
    - Can roll past local optima, but also tends to overshoot
  - Nesterov Accelerated Gradient
    Identical to Momentum Optimisation except the gradient is evaluated at $\theta + \beta \boldsymbol{m}$ instead of θ:
    $$\boldsymbol{m} \leftarrow \beta \boldsymbol{m} - \eta \nabla_\theta J(\theta + \beta \boldsymbol{m})$$
    $$\theta \leftarrow \theta + \boldsymbol{m}$$

    Notes:

    - Compared to Momentum Optimisation it converges faster and effect of oscillations is reduced.
  - AdaGrad
    Simple Gradient Descent quickly descends down the steepest slope, then slowly goes down the valley (where the inputs have widely differing scales). AdaGrad scales down the gradient vector along the steepest dimensions so it points more toward the global optimum:
    $$\boldsymbol{s} \leftarrow \boldsymbol{s} + \nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$$
    $$\theta \leftarrow \theta - \eta \nabla_\theta J(\theta) \oslash \sqrt{\boldsymbol{s} + \epsilon}$$

Notes:

- Works well on simple problems
- Should not be used on deep neural networks (it slows down too fast and doesn't converge)

- RMSProp
  This method overcomes the problem of AdaGrad stopping early by only accumulating the gradients from the most recent iterations:

$$\boldsymbol{s} \leftarrow \beta \boldsymbol{s} + (1-\beta)\, \nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$$
$$\theta \leftarrow \theta - \eta \nabla_\theta J(\theta) \oslash \sqrt{\boldsymbol{s} + \epsilon}$$

- Adam Optimisation
  Adaptive moment estimation uses an exponentially decaying average of past gradients (like Momentum Optimisation), and exponentially decaying average of past gradients squared (like RMSProp).

- Adaptive optimisers (such as AdaGrad, RMSProp and Adam) are generally faster, but can lead to solutions that generalize poorly.
- Note: the above optimisers only require the first order partial derivatives (Jacobians). Better optimisers are available which require second order partical derivatives (Hessians), but they are not practical given the size of the Hessian matrix.
- If you need a model that is fast at prediction (*i.e.* a *sparse* model, where most parameters are zero), you can:
  - Train the model as usual, and then set any small weights to zero
  - Apply strong $l_1$regularization to zero out small weights (see previous notes on Lasso)
  - *Dual averaging*, sometimes called Follow The Regularized Leader (FTRL).
- The learning rate controls how fast the parameters are updated:
  - If it is too high, training may diverge
  - If it is too low, training will take a long time
  - If it slightly too high, it may oscillate around the optimum.
- A learning schedule can be used to adjust the learning rate. There are various methods:
  - Piecewise constant
  - Performance (learning rate is reduce when the validation error stops falling)
  - Exponential and power decay (learning rate is a function of the iteration number).

  Notes:
  - Performance and exponential scheduling usually work well
  - Adaptive optimisers automatically reduce the learning rate as training proceeds, so a learning schedule is not needed.
- Deep neural networks can have millions of parameters, so they can be prone to overfitting. Common regularization methods are:
  - Early stopping
    Training is stopped when the performance on the validation set stops improving. This method works well, but better results can be obtained by combining it with other regularization methods

  - l_1 and $l_2$regularization
    You can sum the l_1 or l_2 norms of each layers' weight vector, and add this sum to the entropy. The total cost being minimised (*i.e.* passed to the optimiser) is then the entropy + the scaled sum of norms.) is then pa regularization terms to the

    - Dropout
      Widely used method that drops out a fixed proportion of nodes in each iteration during training. For each training sample, nodes in all layers (except the output) are temporarily dropped out with a given probability.

      This method makes the resulting more robust because it is effectively averaging an ensemble of smaller networks. It slows down training, but results in a much better model. If the model is still overfitting after using dropout, the dropout rate needs to be increased, and similarly, the dropout rate can be reduced is the model is underfitting.
    - Max-norm regularization
      The l_2 norm of the weights' vector for each layer, $\|\boldsymbol{w}\|_2$ can be constrained to be less than a given threshold. As the threshold decreases, regularization increases and overfitting is reduced.
    - Data augmentation
      It may be possible to derive additional training data from the existing set by applying simple transformations. For example, rotating, scaling and translating images, changing the colour intensity etc.
- Summary
  The following methods work well in the most applications:
    - He initialisation
    - ELU activation function
    - Batch normalisation
    - Dropout regularisation
    - Data augmentation (where possible)
    - Nesterov Accelerated Gradient optimiser
    - Fixed learning rate *i.e.* no schedule.

Chapter 12. Distributing TensorFlow across devices and servers

- Use the CUDA_VISIBLE_DEVICES to restrict access to selected GPUs (on your local machine)
- TensorFlow will take all memory on a GPU (so you can't run a second program), but you can change this by setting the amount of memory TensorFlow takes.
- When a graph is evaluated, it is placed on the default device (GPU0), unless you pin it to a specific device
- You can choose the device type used for pinning an operation to, based on the type of operation *e.g.* variable might be pinned to the CPU, multiplication pinned to GPUs.
- A dependency queue is used to control the order in which nodes are evaluated.
- It may be beneficial delay evaluation of a given operation (subject to the depencies) for various reasons. For example, if it use a lot of memory, you wish to delay evaluation until the results are really needed (instead of early evaluation whereby the results are unused and consuming memory while awaiting other operations – which may be slowed down by the reduced amount of memory available)
- Multiple devices spread over multiple machines can be used by setting up a *cluster*. A cluster consists of multiple servers or tasks. A *job* is a named group of tasks.
- The tasks running a given machine must listen on different ports, so those ports must be accessible through firewalls.

- Operations can be pinned to very specific hardware by specifying the job name (in the cluster), the task index (also in the cluster), the device type and device index.
- The typical configuration is:
    - Model parameters (eg variables) are stored on tasks in a "ps" (parameter server) job
    - All other operations are performed in tasks in a "worker" job.
- In a standalone local session, each variable's state is stored in the session. When the session ends, all variable values are lost. In a distributed session, the variable state is managed in a resource container on the cluster, so the variables can be shared across multiple sessions.
- Queues are widely used for communicating data
- When defining a variable, you may wish to `specify collections=[]`, to prevent it being added to the `GraphKeys.GLOBAL_VARIABLES` collection. This list is used for determining what is saved at checkpoints (*i.e.* you don't want to save large amounts of static (input) training data in a checkpoint)
- There are two approaches used for training multiple networks on multiple networks (for example, an ensemble prediction system that aggregates the predictions from multiple networks):
    - In graph replication
      One large graph is used, containing all networks. Each network is evaluated (potentially in parallel on different devices), and then the results aggregated.
    - Between-graph replication
      A separate graph is used for each network, and a queue is used for handling the aggregation at the end
- Model parallelism refers to running a single network on multiple devices:
    - Fully connected networks cannot be run in this manner because there is too much communication between the layers
    - Partially connected networks can be run across multiple devices, but the efficiency will depend on how much communication is required. Ideally the network would be constructed so that nodes that communicate all reside on the same device (but this won't be possible in most cases, so a trade-off will be required). Devices that need to communicate the most should reside on the same machine.
- Data parallelism refers to training a network in parallel;
    - When using synchronous updates, each device training on a different mini-batch. At the end of each step, the results (*i.e.* gradients) from each device are aggregated the model parameters updated. This can be inefficient if fast devices are idle while waiting for slower devices to finish.
    - When using asynchronous updates the model parameters are updates as soon as a given device finishes. This can introduce stability issues because of stale gradients: when a device has finished computing gradients based on some parameter values, it then attempts to update the parameters but they may have already been changed by another device.
- Bandwidth requirements need to be considered in all operations to prevent saturation.

Chapter 13. Convolutional neural networks

- CNNs are widely used for visual tasks such as image recognition, however they are also used in voice recognition and natural language processing.

- Layers are typically two dimensional for processing 2D images. One dimensional methods are used for language processing.
- CNNs typically consist of convolutional layers and pooling layers:
    - Convolutional layers:
      Each neuron in layer *j* is connected to all neurons in a rectangular sub-region (called a *receptive field*) in layer *j*-1. Padding may be used if the two layers (*i.e.* layers *j*-1 and *j*, respectively) are to be the same width and height.

      The receptive field can move over the input layer *j* using horizontal and vertical strides greater than 1, thereby reducing the size of the output layer *j*-1.

      The 2D matrix of weights (or kernel) is called a filter. Typically the same weights are used for all neurons, and the output is a feature map. A feature map highlights regions in the input layer *j*-1 that are most similar to the filter. During training, the network finds the most useful filters and combines them into complex patterns. Using the same weights for all neurons:
        1. Drastically reduces the number of parameters to train
        2. Means that once the network has learned to recognise a pattern in one location, it can also recognise it in any other location.

      A convolutional layer usually consists of multiple feature maps. A single feature map is obtained by passing a filter (with the same weights for all neurons) over the input layer *j*-1. Multiple feature maps are obtained by passing multiple filters over the input layer *j*-1, each with different weights. However a single feature map in the output layer *j* typically includes weighted contributions from all the feature maps in the input layer *j-1*. Specifically:

$$z_{i,j,k} = b_k + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{\acute{k}=0}^{f_{\acute{n}}-1} x_{\acute{i},\acute{j},\acute{k}} . w_{u,v,\acute{k},k} \text{ with } \begin{cases} \acute{i} = i \times s_h + u \\ \acute{j} = j \times s_w + v \end{cases}$$

      Where:
        - $z_{i,j,k}$ is the output assigned to neuron in row *i*, column *j* in feature map *k* of layer *l*
        - s_h and s_w are the horizontal and vertical strides
        - f_h and f_w are the height and width of the receptive field
        - b_k is the bias for feature map *k* of layer *l*
        - $f_{\acute{n}}$ is the number of feature maps in the previous layer, *l-1*
        - $x_{\acute{i},\acute{j},\acute{k}}$ is the output from neuron in row $\acute{i}$, column $\acute{j}$ in feature map $\acute{k}$ of layer *l*-1 (or channel $\acute{k}$ if the previous layer is the input layer *i.e. j*-1 = 0)
        - $w_{u,v,\acute{k},k}$ are the connection weights between feature map *k* in layer *l* and feature map $\acute{k}$ *in layer l*-1. Note how the summation is over all feature maps in the previous layer.

      Note: the input data may also have multiple channels eg. red, green, blue.

      For example, if we have m training instances, each having 3 channels (RGB), and wish to apply f_n filters:

        - We need $3 \times f_n$ filters (we need one filter for each channel in the input *i.e.* $f_{\acute{n}} = 3$)
        - We obtain $m \times f_n$ output feature maps *i.e.* for each instance we obtain f_n output feature maps (we sum over of the channels/feature maps in the input – see the $\sum_{\acute{k}=0}^{f_{\acute{n}}-1}$ summation in the expression above)

    - Pooling layers:
      Pooling layers sub-sample the input feature maps to reduce their size, thereby reducing the computational load, memory requirements and number of parameters (reducing the risk of overfitting).

      Pooling neurons have no weights. Typical aggregation functions are the maximum or mean. *i.e.* the output is simply the maximum (or mean) of the neuron values within the kernel.

      The pooling layer typically works on each input channel/feature map independently, so the depth of the input and output layers (layers *j*-1 and *j*, respectively) are the same. Alternatively you can pool over the depth dimension, in which case the spatial dimension is not changed but the number of channels/feature maps is reduce.
- Cross validation can be used to find the optimal values for the hyperparameters, but this is very time consuming.
- Large amounts of RAM are required for training because the reverse pass of back propagation requires all the intermediate values from the forward pass.
- Some architectures use:
    - Regularization, such as dropout (temporarily removing neurons) and data augmentation (transforming the input instances to create more training data, eg translating or rotating images)
    - Techniques to encourage different feature maps to specialise *e.g.* local response normalisation identifies neurons strongly responding in a given feature map, and then inhibits the corresponding neurons in the neighbouring feature maps *i.e.* neighbouring in the depth dimension.
    - Techniques that enable very deep networks to be trained *e.g.* skipping connections can assist the signal in moving through the network, particularly when training is just starting.
- In general:
    - The image gets smaller as it progresses through the network, but also gets deeper. At the top of the stock there is usually a few fully connected feed-forward layers, with a softmax layer outputting estimate class probabilities)
    - It is more efficient (less compute load and less parameters) to use multiple layers with smaller kernels than a single layer with a larger kernel.

Chapter 14. Recurrent neural networks

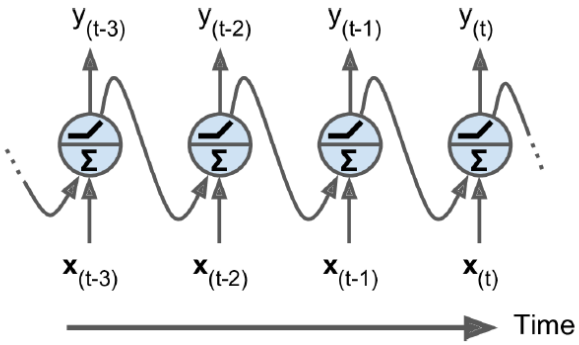
- RNNs are widely used for:
    - Time series data (*e.g.* forecasting a vehicle's trajectory)
    - Input sequences of arbitrary length (sentences, documents, audio samples etc.)

They are commonly used for natural language processing (translation), speech-to-text encoding and sentiment analysis.

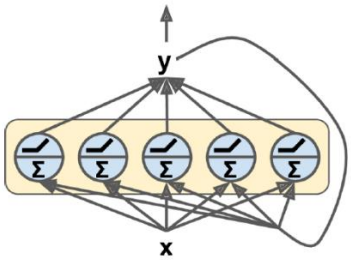
- The basic building block is the recurrent neuron, which takes two inputs at each time step: $\boldsymbol{x}_t$ (the input vector at time t) and y_{t-1} (the scalar output from the previous time step):

The output is: $y_t = \phi(\boldsymbol{x}_t.\boldsymbol{w} + b)$, where ϕ is the activation function.

The output is therefore a function of the inputs at *all* previous time steps. A recurrent network is created by chaining multiple cells together. If the network is unrolled along the time axis, we have:



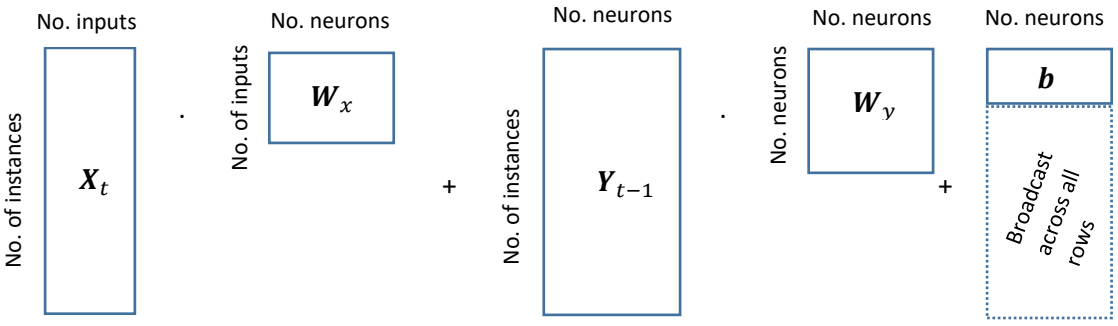
- A *layer* of recurrent neurons is created by feeding the input into multiple recurrent neurons. However, the output is now a vector $\boldsymbol{y}_t$ and the input consists of $\boldsymbol{x}_t$ (the input vector at time t) and $\boldsymbol{y}_{t-1}$ (the *vector* output from the previous time step).



The output is now:

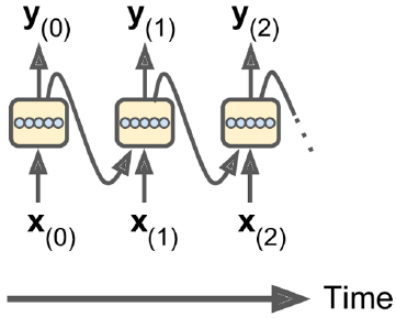
$$\boldsymbol{Y}_t = \phi\left(\boldsymbol{X}_t \cdot \boldsymbol{W}_x + \boldsymbol{Y}_{t-1} \cdot \boldsymbol{W}_y + \boldsymbol{b}\right)$$

or



And the output for each instance is a vector of length $n_{neurons}$. The number of neurons is not related to the length of the input sequence.

The recurrent network is again created by chaining multiple layers together. When unrolled through time we have:



- A *cell* or *memory cell* is a neuron that preserves some state across time steps *i.e.* it has some form of memory.
- RNNs can take various inputs and outputs:
    - Sequence to sequence (*i.e.* the input is a sequence such as a sentence) and the output is also a sequence
    - Sequence to vector (*e.g.* movie review (sequence of words) and output a rating)
    - Vector to sequence (*e.g.* input could be an image and the output a caption)
    - Delayed sequence to sequence, consisting of an encoder (sequence to vector) followed by a decoder (vector to sequence)
- An RNN can be created by:
    - Static unrolling: building a series of basic cells – one cell for each time step.
    - Dynamic unrolling: instead of building a series of cells, it builds a single cell and during execution it loops over it the required number of times (*i.e.* the number of input time steps). This method is capable of dealing with variable length inputs and outputs.

  The same weights and bias are used in all cells.

- Back propagation through time (BPTT) is used to train an RNN:
    - Step 1. Forward pass through the unrolled network
    - Step 2. Output sequence is evaluated using a cost function that considers the cost over a given number of time steps (*i.e.* not just the last time step):

      $$C(\boldsymbol{y}_{t_{min}}, \boldsymbol{y}_{t_{min}+1}, \cdots, \boldsymbol{y}_{t_{max}})$$

      Step 3. The gradients of the cost function are propagated backward through the unrolled network, updating the model parameters (weights and bias) for the cells which output $\boldsymbol{y}_{t_{min}}, \boldsymbol{y}_{t_{min}+1}, \cdots, \boldsymbol{y}_{t_{max}}$.
- RNN performance can be improved using methods applicable to other network types, such as: tuning hyperparameters with cross validation, optimal initialization of the parameters (*e.g.* He initialization), regularization such as dropout, longer training.
- An RNN can be used to classify a sequence. Given a set of training instances, each having length n, the network should consist of
    - a series of n cells. Each cell can have $n_{neurons}$ neurons (*i.e.* a layer)

    - the output is passed into a fully connected layer with $n_{neurons}$ inputs and $n_{classes}$ outputs, where the sequence could be classified into $n_{classes}$ possible classes
    - a softmax layer to select the predicted class.
- An RNN can be used to forecast time series data.
    - Given a long time series of data, training instances are created by randomly selecting sequences on n consecutive values, where n is the number of time steps you wish to use for training. Increasing the number of time steps may help capture long term behaviour, but it will reduce the number of training instances, and the effectiveness of long sequences is reduced by memory loss (unless special memory-preserving cells are used).
    - In general, you may have more than one input feature
    - The network can be assembled in two ways:
        - Each of the n cells could be attached to a fully connected layer with $n_{neurons}$ inputs and $n_{out-features}$ outputs (where $n_{out-features}$ is the feature length of the output time series)
          or,
        - The output from all n cells is reshaped into a tensor of size $(n_{neurons}, n)$ which is passed into a single fully connected layer. The output of the fully connected layer is subsequently reshaped into a vector of length $n_{out-features}$ for each of the n outputs. This approach is more efficient (but care must be taken to check the results are as good as the less efficient approach).
- Deep RNNs are created by stacking multiple layers of cells:
    - To prevent overfitting, dropout is commonly used. Note: dropout is only used during training, not during prediction.
    - Training deep RNNs over long time series is difficult because:
        - Vanishing/exploding gradients
          Techniques used on other network types can alleviate these issues, such as appropriate initialisation, batch normalisation, clipping gradients, faster optimisers, non-saturating activation functions.
        - Training is slow
          During training, unroll the network by a limited number of time steps. However the model will not be able to learn long-term patterns. To overcome this you could change the temporal resolution of the input data: each instance could contain some annual values, followed by some monthly values, followed by daily values.
        - Memory fade
          Some information is lost in each time step, so the memory of the first inputs gradually fades away.
- To prevent memory fade, use a cell with long term memory:
    - LSTM (Long-Short Term Memory) cell:
      Instead of the output $\boldsymbol{y}_{t-1}$ being fed into the next unit, the state is split into a short term and long term component. The input $\boldsymbol{x}_t$ and short term memory $\boldsymbol{h}_{t-1}$ are fed into four fully connected units:
        - Forget gate, controls what is dropped from the long term memory $\boldsymbol{c}_{t-1}$
        - Input gate, controls what is added to the long term memory $\boldsymbol{c}_{t-1}$

        - Output gate, given the updated long term memory $\boldsymbol{c}_t$, it controls what is sent to output $\boldsymbol{y}_t$ and the updated short term memory $\boldsymbol{h}_t$
        - Main gate, analyses the current input (the same as in the basic cell).

      The role of the input gate is to recognise important input and store it in the long term state.

      Peephole connections can be used to provide the gates with information from the long term state $\boldsymbol{c}_{t-1}$ (by default they can only see the current input $\boldsymbol{x}_t$ and short term state, $\boldsymbol{h}_{t-1}$).

    - GRU (Gated Recurrent Unit), which is a simplified version of the LSTM cell but offers similar performance.

    Improvements in applications such as natural language processing that use RNNs is largely due to the use of LSTMs and GRUs.

- Natural language processing (machine translation, summarisation, parsing, sentiment analysis etc.) uses an encoder/decoder architecture:
    - Each element (*e.g.* word) in the vocabulary is represented using a dense vector, called embedding.
    - Pre-trained embeddings are publicly available, which you can use as-is, or start training with a existing embedding and fine-tune in your application.
    - Embeddings are useful for categorical attributes that can take on many values, especially when there are complex similarities between them.

Chapter 15. Autoencoders

- Autoencoders generate efficient representations of the input data. They are widely used for:
    - Dimensionality reduction
    - Feature detection
    - Unsupervised pre-training of other networks
    - Generating new data sets.
- Autoencoders generally work on unlabelled data
- Autoencoders typically consist of:
    - An encoder (or recognition network), which converts the input data into an internal representation *i.e.* the coding
    - A decoder (or generative network) which generates output approximating the input.
- The internal representation *i.e.* coding has lower dimensionality than the input data, the encoder is forced to learn the most important features and discard less important features.
- An autoencoder with:
    - Reduced dimensionality
    - Linear activation functions (*i.e.* no activation function; the outputs are not modified)
    - Mean Square Error cost function

  is the same as principal component analysis.

- Stacked or deep Autoencoders consist of multiple hidden layers:
    - They are usually symmetric in the number of neurons in the hidden layers
    - The encoder and decoder weights are usually tied to each other (this halves the number of weights and speeds up training)

- They can lean complex encodings, however it is important that the encoder is not too powerful otherwise it will overfit the training data
- When the weights are tied, they are commonly regularized
- Biases are never tied and never regularized.

- Encoders with multiple layers can be trained one layer at a time. For example, if the encoder has 3 hidden layers:

$$Input \rightarrow \underset{n\ neurons}{hidden\ layer\ 1} \rightarrow \underset{m\ neurons}{hidden\ layer\ 2} \rightarrow \underset{n\ neurons}{hidden\ layer\ 3} \rightarrow output$$

it can be trained in two steps:

1. Train:

$$Input \rightarrow hidden\ layer\ 1 \rightarrow output$$

2. Given the outputs of $hidden\ layer\ 1$, train:

$$hidden\ layer\ 1 \rightarrow hidden\ layer\ 2 \rightarrow hidden\ layer\ 3$$

Note: the outputs of $hidden\ layer\ 1$ can be cached after Step 1, and reused in Step 2 *i.e.* the outputs from $hidden\ layer\ 1$ don't need to be constantly re-evaluated while during Step 2.

- Autoencoders can be used for pre-training networks. Given a complex supervised classification task, you can reduce the amount of labelled training data required to train the network (and also training time), by reusing the lower layers of a network that has already been trained on a similar task. This approach works because the lower layers learn the low level features which are common to both tasks. An autoencoder can be used to pretrain a network for the same reason: the autoencoder can learn low-level features using unlabelled data, and these layers can then be reused in a network designed for supervised classification. The target network can then be refined using a relatively small amount of labelled data (required to learn the high level features *i.e.* train the remaining upper layers). Note: it is common to use a smaller learning rate when fine-tuning pre-trained weights, in comparison to using (randomly-initialized) weights.
- The recent rapid advancement in deep networks has arisen from the discovery that deep networks can be pretrained in an unsupervised fashion, thereby enabling much deeper networks to be trained.
- Denoising autoencoders: Gaussian noise is added to the input samples to prevent the autoencoder trivially copying the input to the output. Dropout can be used in a similar fashion.
- Sparse autoencoders: a constraint on the number of active neurons (sparsity) is used to force neurons to learn useful features. Sparsity can be enforced by including a sparsity term in the cost function. The sparsity measures how many neurons are active in the encoding layer, thereby forcing the autoencoder to represent inputs using a small number of activations.

  The loss measure to be minimised will then consist of:

$$loss_{reconstruction} + loss_{sparsity}$$

  where:

  - The reconstruction loss is the difference between the input and output. This can be the Mean Square Error, but cross entropy is preferred because it has larger gradients and will therefore converge faster
  - The sparsity loss can be computed from the Kullback-Leibler divergence of the desired activation rate and the actual rate.

- Variational autoencoders: these are probabilistic autoencoders that are used for generating new training instances *i.e.* they are used for augmenting training datasets. Variational autoencoders differ from other autoencoders:
    - Instead of producing an encoding for each input, it produces a mean encoding μ and a standard deviation σ. The output from the encoder is randomly sampled from a Gaussian distribution, $N(\mu, \sigma)$.
    - The loss measure to be minimised consists of:
      $$loss_{reconstruction} + loss_{latent}$$
      where the latent loss forces the autoencoder to have encodings that appear to have been sampled from a simple Gaussian distribution. The Kullback-Leibler divergence between the target (*i.e.* Gaussian) distribution and the actual distribution is used for this term.

Chapter 16. Reinforcement learning

- In reinforcement learning, the object is to learn to act in a way that will maximize its long term rewards.
- A software *agent* makes *observations* and takes *actions* within an *environment* and in turn receives *rewards*.
- The *policy* is the algorithm used by the agent to determine its actions.
- The policy parameters can be obtained by:
    - Policy search: exhaustive search but this is not feasible if the policy space is large
    - Genetic algorithms: randomly start with n policies and evaluate them. The worst (say) 80% are killed off, and the remaining 20% are used to derive offspring. The offspring are derived from the parent(s) with some random mutation added. The process continues for multiple generation.
    - Policy Gradients: see below.
    - Markov Decision Processes: see below.
- Credit assignment problem: should the reward be assigned to the most recent action, or any of the previous actions that lead up to it?
- To overcome the credit assignment problem, an action is evaluated from the sum of all rewards that come after it, typically applying a discounting rate at each step.
- To assist in training you should inject as much prior knowledge as possible.
- There are two fundamental approaches to learning policies:
    - Policy Gradients: the policy is optimised to increase rewards
    - Markov Decision processes: the agent learns to estimate the expected sum of discounted future rewards for: (i) each state; or (ii) each action in each state.
- Policy Gradients
    - The algorithm compute the gradients of the reward *wrt* the policy parameters, and modify the parameters in the direction that increases the reward *i.e.* gradient ascent.
    - A neural network can learn a policy using Policy Gradients: compute the probability for each possible action. The action to take is then obtained by randomly selecting an action, according to the probability of each action. This approach (as opposed to selecting the action with highest probability) lets the agent find a balance between *exploring* new actions and *exploiting* actions that are known to work well.
    - Training via neural network is difficult because at any given step we don't know what the best action to take is:

        - If we knew the best action, we could train the network by minimizing the cross entropy between the estimated and target probabilities.
        - The only information we have are the rewards, and the reward may be delayed.
- Markov Decision Processes
    - Markov chains are stochastic processes with no memory. The process has a fixed number of states and it randomly evolves from one state to another at each step.
    - Markov decision processes: the same as Markov chains but the transition probabilities depend on the action taken.
    - The optimal state value $V^*(s)$ of a given state s is the sum of the discounted future rewards the agent can expect *on average*, assuming it acts optimally. It can be computed using the Bellman Optimality Equation:

      $$V^*(s) = \max_a \sum_{\acute{s}} T(s,a,\acute{s})[R(s,a,\acute{s}) + \gamma.V^*(\acute{s})]$$

      where $T(s,a,\acute{s})$ is the transition probability from state s to state $\acute{s}$ following action a, $R(s,a,\acute{s})$ is the associated reward and γ is the discounting rate.

      The optimal state values $V^*(s)$ can be computed iteratively using the *Value Iteration* algorithm:

      $$V_{k+1}(s) \leftarrow \max_a \sum_{\acute{s}} T(s,a,\acute{s})[R(s,a,\acute{s}) + \gamma.V_k(\acute{s})] \quad \text{for all } s.$$

      Optimal state values can be used to evaluate a policy, but cannot tell an agent what to do.
    - Q-values (state-action pairs) are required for decision making. The optimal Q-value for the state-action pair (s,a) is denoted $Q^*(s,a)$. It is the sum of the discounted future rewards the agent can expect *on average*, after it reaches state s and choses action a (but before seeing the outcome of this action) assuming it acts optimally. The optimal Q-values for each state-action pair can be computed iteratively using the Q-value iteration algorithm:

      $$Q_{k+1}(s,a) \leftarrow \sum_{\acute{s}} T(s,a,\acute{s})\left[R(s,a,\acute{s}) + \gamma.\max_{\acute{a}} Q_k(\acute{s},\acute{a})\right] \quad \text{for all } s,a.$$

      The optimal Q-values tell us the optimal action a to take from state s.
- A reinforcement learning problem with discrete actions can commonly be modelled using a Markov Decision Process, however the transition probabilities $T(s,a,\acute{s})$ and rewards $R(s,a,\acute{s})$ are usually unknown, so we can't compute the $Q^*(s,a)$:
    - To learn the rewards, the agent must experience each state-action pair at least once; and
    - To learn the transition probabilities it needs to experience them multiple times.

  By randomly exploring the MDP, we can iteratively compute:
    1. The state values using the Temporal Difference learning algorithm:

       $$V_{k+1}(s) \leftarrow (1-\alpha)V_k(\mathrm{s}) + \alpha[\mathrm{r} + \gamma.V_k(\acute{s})] \quad \text{for all } s.$$

    2. The Q-values using the Q-Learning algorithm:

       $$Q_{k+1}(s,a) \leftarrow (1-\alpha)Q_k(\mathrm{s},\mathrm{a}) + \alpha\left[\mathrm{r} + \gamma.\max_{\acute{a}} Q_k(\acute{s},\acute{a})\right] \quad \text{for all } s,a.$$

- Q-learning will compute the optimal Q-values given enough iterations *i.e.* if it thoroughly explores the policy space. The random policy search can be improved by:
    - ε-greedy policy: at each step it acts randomly with probability ε, or greedily with probability $1-\varepsilon$ (in which case it chooses the action with the highest Q-value). As

the Q-value estimates improve, the parameter ε is gradually reduced. The algorithm works by focussing on high-value parts of the MDP (environment), but also occasionally visiting unknown regions of the MDP.
    - Steering: use an exploration policy that encourages testing of actions that have not been tried much before.
- Q-learning scales poorly as the number of possible states and actions increases. Approximate Q-learning overcomes this problem:
    - Instead of trying to compute $Q^*(s, a)$, we find a function $Q_\theta(s, a)$ that approximates the Q-value for any state-action pair (s, a) using a small number of parameters θ.
    - A neural network can be used to compute this function. Deep Q-learning uses a deep neural network to estimate Q-values.
- A deep Q-learning network can be trained by gradient descent, minimizing the square error between the target Q-value and the estimated Q-value. The target value:

$$y(s, a) = r + \gamma . \max_{\acute{a}} Q_\theta(\acute{s}, \acute{a})$$

  where r is the reward from playing action a from state s, and the resulting being state $\acute{s}$. The target value is computed

- A deep Q-learning system usually consists of two networks:
    - A target DQN whose only role is to estimate the next state's Q-values for each possible action
    - An online DQN that learns. Its parameters are copied to the target DQN at regular intervals.

Appendix 1. Distributions

Binomial distribution

The binomial distribution with parameters n and p is the discrete probability distribution of the number of successes in a sequence of n independent experiments, each asking a yes–no question, and each with its own boolean-valued outcome: a random variable containing single bit of information: success/yes/true/one (with probability p) or failure/no/false/zero (with probability $q = 1 - p$). A single success/failure experiment is also called a Bernoulli trial or Bernoulli experiment and a sequence of outcomes is called a Bernoulli process.

The binomial distribution is frequently used to model the number of successes in a sample of size n drawn with replacement from a population of size N.

If the random variable X follows the binomial distribution with parameters $n \in \mathbb{N}$ and $p \in [0,1]$, the probability of getting exactly k successes in n trials is given by the probability mass function:

$$\Pr(k; n, p) = \binom{n}{k} p^k (1-p)^{n-k} \quad for\ k = 0,1,2,\ldots,n$$

where

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

is the binomial coefficient.

Source: https://en.wikipedia.org/wiki/Binomial_distribution

Multinomial distribution

The multinomial distribution is the generalization of the binomial distribution to cases where there are more than two possible outcomes.

Appendix 2. Implementation notes

- NumPy's reshape() function allows one dimension to be –1, which means "unspecified": the value is inferred from the length of the array and the remaining dimensions.