Hands-on Machine Learning with sklearn, numpy, matplotlib, pandas, keras and TensorFlow 2 + RAPIDS.

Table of contents

[1 Prerequisites 5](#_Toc66140822)

[1.1 Python 5](#_Toc66140823)

[1.1.1 Data Structures 5](#_Toc66140824)

[1.1.2 Functions and lambdas 6](#_Toc66140825)

[1.1.3 Error handling 6](#_Toc66140826)

[1.1.4 String operations 7](#_Toc66140827)

[1.1.5 IO 7](#_Toc66140828)

[1.1.6 Classes 8](#_Toc66140829)

[1.1.7 Pickle – serialization framework – 9](#_Toc66140830)

[1.1.8 Generator, yield, async-await and coroutines 9](#_Toc66140831)

[1.1.9 Threading 9](#_Toc66140832)

[1.2 Numpy 9](#_Toc66140833)

[1.2.1 Arrays (vectors and matrices) 9](#_Toc66140834)

[1.2.2 Shaping 10](#_Toc66140835)

[1.2.3 Stacking 10](#_Toc66140836)

[1.2.4 Linalg 10](#_Toc66140837)

[1.2.5 Miscellaneous (random, linspace, math) 10](#_Toc66140838)

[1.3 Pandas 11](#_Toc66140839)

[1.3.1 Series 11](#_Toc66140840)

[1.3.2 DataFrame 11](#_Toc66140841)

[1.4 Matplotlib 13](#_Toc66140842)

[1.4.1 Lines and 2D functions 13](#_Toc66140843)

[1.4.2 Multiple plots on a figure 14](#_Toc66140844)

[1.4.3 Scatter plots, histograms and images 15](#_Toc66140845)

[1.4.4 Texts and Legends 16](#_Toc66140846)

[1.4.5 ... 16](#_Toc66140847)

[1.5 Jupyter notebook 16](#_Toc66140848)

[1.6 R 17](#_Toc66140849)

[1.6.1 Attributes, Basic types, NA and NaN 17](#_Toc66140850)

[1.6.2 Complex types 17](#_Toc66140851)

[1.6.3 Subsetting, indexing 19](#_Toc66140852)

[1.6.4 Date & time 20](#_Toc66140853)

[1.6.5 Reading data, Files 21](#_Toc66140854)

[1.6.6 Control structures 21](#_Toc66140855)

[1.6.7 Functions 22](#_Toc66140856)

[1.6.8 Loop functions (apply-s) 22](#_Toc66140857)

[1.7 SQL 23](#_Toc66140858)

[1.8 Matlab 23](#_Toc66140859)

[2 Machine Learning 24](#_Toc66140860)

[2.1 Types of Machine Learning systems 24](#_Toc66140861)

[2.1.1 Supervised / Unsupervised / Semisupervised / Reinforcement 24](#_Toc66140862)

[2.1.2 Batch / Online 24](#_Toc66140863)

[2.1.3 Instance-based / Model-based 24](#_Toc66140864)

[2.2 Challenges of Machine Learning 24](#_Toc66140865)

[2.3 Testing and Validating 25](#_Toc66140866)

[2.4 Scaling 25](#_Toc66140867)

[3 Data Science workflow 26](#_Toc66140868)

[3.1 Panelek 26](#_Toc66140869)

[3.2 Kényelmi funkciók 26](#_Toc66140870)

[4 Scikit-learn utilities 27](#_Toc66140871)

[4.1 Datasets 27](#_Toc66140872)

[4.2 Test-train split and cross validation 27](#_Toc66140873)

[4.3 Clustering 27](#_Toc66140874)

[4.3.1 Agglomerative Clustering 27](#_Toc66140875)

[4.3.2 K-Means Clustering 27](#_Toc66140876)

[4.3.3 DBSCAN (Density-Based) Clustering 27](#_Toc66140877)

[4.4 Egyéb 27](#_Toc66140878)

[4.5 Egyéb 27](#_Toc66140879)

[4.6 Egyéb 27](#_Toc66140880)

[5 Classification 28](#_Toc66140881)

[5.1 Linear model 28](#_Toc66140882)

[5.2 Confusion Matrix and ROC Curve 28](#_Toc66140883)

[5.2.1 Precision 29](#_Toc66140884)

[5.2.2 Recall 29](#_Toc66140885)

[5.2.3 F1 Score and Precision-Recall tradeoff 29](#_Toc66140886)

[5.3 Multiclass Classification 31](#_Toc66140887)

[5.3.1 One versus All (One versus Rest) 31](#_Toc66140888)

[5.3.2 One versus One 31](#_Toc66140889)

[5.4 Egyéb 31](#_Toc66140890)

[6 Classification 33](#_Toc66140891)

[7 Neural Networks 34](#_Toc66140892)

[7.1 Forward and Backpropagation 34](#_Toc66140893)

[7.1.1 Math 34](#_Toc66140894)

[7.1.2 Forward propagation on XOR N.N. 34](#_Toc66140895)

[7.1.3 Backpropagation on XOR N.N. 35](#_Toc66140896)

[7.2 Egyéb 36](#_Toc66140897)

[8 Classification 37](#_Toc66140898)

[9 Classification 38](#_Toc66140899)

# Prerequisites

The book assumes basic programming knowledge as well familiarity with the most often used scientific libraries such as numpy, pandas and matplotlib.

## Python

Modern programming language, where classes passed as *reference copy* and primitives passed by value.

*Reference copy:* Assigning new objects to this won’t affect the original object’s state. Moreover, modifying the object is only feasible by calling operations (add,pop etc.) on this copy.

### Data Structures

#### List

lst = [1, 2, 3]

lst.append(4) #4 to the end

lst.insert(0, 0) #0 to the beginning

lst.remove(obj)

lst.pop(index) #remove element

lst.copy()

lst.extend(lst\_other)

primes = list(filter(lambda num: num.is\_prime, numbers))

#method called list comprehension:

primes = [num.value **for** num **in** numbers **if** num.is\_prime]

lst[-1] gives you the last element

#### Tuple

Similar to list but unchangeable during runtime.

tpl = (1, 2, 3)

zip() function for pairing lists/tuples into a single tuple

#### Set (unordered)

st = {1, 2, 3}

st.add(4)

st.update(list)

st.remove(obj) #will raise Error if not found

st.discard(obj) #no Error thrown

st3 = st.union(st2)

#### Dictionary

dct = { ”a” : 1, ”b”: 2, ”c” : 3 }

dct[”d”] = 4

dct.pop(”c”)

#no error thrown when key doesnt exists, instead it gets #initialized with 0

**from** collections **import** defaultdict  
dct = defaultdict(int)

**for** key **in** dct:  
 **pass  
  
for** value **in** dct.values():  
 **pass  
  
for** (key, value) **in** dct.items():  
 **pass**

### Functions and lambdas

**def** func(p1, p2, \*args, \*\*kwargs):  
 print(args[0], args[1], kwargs[**"kwparam"**])

#will print 3 4 cc  
  
**def** main():  
 func(1, 2, 3, 4, kwparam=**"cc"**)

lbd = **lambda** x : np.exp(x)

### Error handling

**try**:  
 **raise** NotImplementedError(**"test"**)  
**except** NotImplementedError **as** err:  
 print(err)  
**except**:  
 print(**"wont be printed"**)

### String operations

#### String interpolation

n = **"john"**a = 30  
str1 = **"name: {} age: {:0.2f}"**.format(n, a)  
str2 = **"name: {name} age: {age}"**.format(name=n, age=a)

#### ...

### IO

IO modes:

|  |  |
| --- | --- |
| r | reading  raises error if file’s missing |
| a | append  creates the missing file or writes to the end |
| w | writing  create the missing file or overwrites |
| x | create  error if file exists |

|  |  |
| --- | --- |
| t | text |
| b | binary |

file = open(path, mode) #”rt” ”xb” etc.

for line in file:

**pass**

file.close()

file.write(...)

RAII for IO (will automatically close, even on raised errors):

**with** open(**"path"**, **"mode"**) **as** f:  
 **pass**

### Classes

#### Defining a class and achieve inheritance

**class** BASE:  
 **def** \_\_init\_\_(self, x):  
 self.x = x  
 **def** GetX(self):  
 **return** self.x  
  
**class** DERIVED(BASE):  
 **def** \_\_init\_\_(self, x, y):  
 super().\_\_init\_\_(x) #BASE.\_\_initself.y = y  
 **def** GetY(self):  
 **return** self.y  
  
**def** main():  
 d = DERIVED(10, 20)  
 print(d.GetX())  
 print(d.GetY())  
  
main()

#### Static variables

**class** C:  
 static = **"ABC"  
  
 def** \_\_init\_\_(self):  
 **pass**

#### Class and Static methods

**class** C:  
 static = **"ABC"  
  
 def** \_\_init\_\_(self, name, age):  
 self.name = name  
 self.age = age  
  
 @classmethod  
 **def** John(cls):  
 **return** cls(**"John"**, 32)  
  
 @classmethod  
 **def** Eva(cls):  
 **return** cls(**"Eva"**, 22)  
  
 @staticmethod  
 **def** set\_static(val):  
 C.static = val  
  
**def** main():  
 c\_john = C.John()

**@classmethod** decorator is used for returning objects constructed by a set of predefined parameters. Also, it is the most pythonic way to achive multiple constructors as you can create only one \_\_init\_\_ definition.

**@staticmethod** decorator marks the function as a static one.

#### ...

### Pickle – serialization framework –

**if** os.path.exists(**"ser.data"**):  
 obj = pickle.load(open(**"ser.data"**, **"rb"**))  
**else**:  
 obj = **None** *# some long processing task..* pickle.dump(obj, open(**"ser.data"**, **"wb"**))

### Generator, yield, async-await and coroutines

### Threading

## Numpy

Math library with operations faster than standard library.

**import** numpy **as** np

### Arrays (vectors and matrices)

np.empty(5)

np.zeros((4,4))

np.full((4,4), np.pi)

np.array([[...], [...], ..., [...]])

np.arange(lower\_bound\_inc, upper\_bound\_exc, step=1) #iota

np.random.rand(4) #4 long vector with rands between 0,1

np.random.randn(4) #4 long vector with N(0,1) distribution

np.fromfunction(**lambda** i, j: i\*j, (4, 4)) #ij can be 0

### Shaping

arr.shape = (4,4)

arr = np.reshape(arr, (16, 1)) #-1 calculates the other dim

x = np.atleast\_2d(x) #converts lower dimensions to 2D, useful for reshaping vector type np.arrays to matrix type (for dotprod)

arr.ravel() #to vector

arr.flatten() #copies, and converts copy to vector

### Stacking

qv1, qv2: [[1 1 1]], [[2 2 2] [2 2 2]]

qv3 = np.vstack((qv1, qv2))

qv3 = [[1 1 1] [2 2 2] [2 2 2]]

qh1, qh2: [[1 1 1] [1 1 1]], [[2 2] [2 2]]

qh3 = np.hstack((qh1, qh2))

qh3 = [[1 1 1 2 2] [1 1 1 2 2]]

### Linalg

**import** numpy.linalg **as** linalg

mtx.transpose()

mtx.T

mtx\_mult = mtx1.dot(mtx2)

inverse = linalg.inv(mtx)

pinverse = linalg.pinv(mtx)

identity\_mtx = np.eye(n)

det = linalg.det(mtx)

eigenvalues, eigenvectors = linalg.eig(mtx)

### Miscellaneous (random, linspace, math)

axis=0 refers to row

axis=1 refers to column

np.linspace(min, max, num): splits the interval (min, max) into num equal parts

r1, r2, r3 = np.random.rand(3, 100): creates a 3x100 random matrix, ri are the ith row

np.random.normal(0, 1, 100): creates a 100 long vector with normally distributed vals

a[i, j] indexing a multidiminensional array

np.sum(), np.max(), np.argmax() and other unary operators

np.squeeze(X, axis=1): removes the axis if it is single dimensioned

np.append(X, y): append element y to array X, returns a new object

## Pandas

Library for messing around with data.

**import** pandas **as** pd

### Series

Equivalent of Excel’s column.

|  |  |
| --- | --- |
| s\_indexed | |
| „a” | 30 |
| „b” | 27 |
| „c” | 22 |
| „d” | 41 |

#### Creating Series

|  |  |
| --- | --- |
| s\_base | |
| 0 | 2 |
| 1 | 3 |
| 2 | 4 |
| 3 | 5 |

predefined\_dic = {**"c"** : 22, **"d"** : 40}  
  
s\_base = pd.Series([2, 3, 4, 5])

s = np.exp(s\_base)  
s = s\_base + pd.Series([4, 3, 2, 1])  
s = s\_base + 1  
s = s\_base <= 3

s = pd.Series([30, 27, 22, 41], index=[**"a"**, **"b"**, **"c"**, **"d"**])  
s = pd.Series(predefined\_dic, index=[**"c"**]) #filtering by indices

s = pd.Series([22, 23], index=[**"alice"**, **"bob"**], name=**"ages"**)

s = pd.Series(np.nan, index=[**"def1"**, **"def2"**]) #def1: Nan, def2: NaN

#### Plotting Series

plt.scatter(s.index, s.values)   
#plt.plot would connect the points with linesplt.show()

### DataFrame

Equivalent of Excel’s spreadsheet.

#### Creating DataFrames

predefined\_dic = {  
 **"in letters"** : pd.Series([1, 2, 3]),  
 **"w numbers"** : pd.Series([**"ein"**, **"zwei"**, **"drei"**]),  
 **"rand"**: pd.Series(np.full(3, np.nan))  
 }  
df = pd.DataFrame(predefined\_dic) #if some columns couldn't be matched, those will become NaNsprint(df[[**"in letters"**, **"w numbers"**]])

#filtering  
df = pd.DataFrame(predefined\_dic, columns=[**"w numbers"**], index=[1, 2]) np\_datamatrix = np.array([  
 [**"alice"**, 33, **"mercedes"**],  
 [**"bob"**, 40, **"cadillac"**],  
 [**"chad"**, 25, **"bmw"**],  
])  
df = pd.DataFrame(np\_datamatrix, columns=[**"name"**, **"age"**, **"car"**], index=[**"a"**, **"b"**, **"c"**])

#### Accessing rows

|  |  |  |  |
| --- | --- | --- | --- |
|  | name | age | car |
| a (iloc 0) | alice | 33 | mercedes |
| b (iloc 1) | bob | 40 | cadillac |
| c (iloc 2) | chad | 25 | bmw |

#row of chaddf.loc[**"c"**]  
df.iloc[2]  
#row of alice and chaddf[np.array([**True**, **False**, **True**])]  
#row of alice and chaddf[pd.to\_numeric(df[**"age"**], errors=**"raise"**) <= 35]  
#inserting a new rowdf.loc[**"d"**] = np.array([**"dave"**, 17, np.nan])

df.drop(index=[**"d"**], inplace=**True**) #deleting the new row

#### Accessing columns

df = df.astype({**"age"** : **"int32"**})  
df[**"over 18"**] = [**False**] \* len(df.index)  
df[**"over 18"**] = pd.Series({**"a"** : **True**}) *#other will default to np.nan*df[**"over 18"**] = df[**"age"**] >= 18  
popped\_col = df.pop(**"over 18"**)  
df.drop(columns=[**"over 18"**])  
df.insert(0, **"ID card num"**, pd.Series({**"a"** : **"0x00"**, **"b"** : **"0xcf"**, **"c"** : **"0xfd"**}))  
evdf1 = df.eval(**"age\*\*2 + sin(age)"**)  
age\_limit = 21  
evdf2 = df.eval(**"age > @age\_limit"**)

#### Querying

queried\_df = df.query(**"age > @age\_limit and name.str.contains('a')"**)

#### Sorting

df.sort\_values(**"car"**, ascending=**True**, inplace=**True**)  
df.sort\_values([**"car"**, **"age"**], ascending=[**True**, **True**], inplace=**True**)  
df.sort\_index(ascending=**False**)

#### Handle missing data

df.fillna(**"no data"**, inplace=**True**)  
df.dropna(axis=0, how=**"any"**, inplace=**True**) #drop row if any of its value is np.nan

#### Pandas Utilities for reading structured data and misc

axis 0: rows

axis 1: columns

housing = pd.read\_csv(**"housing.dat"**, sep=**"..."**)  
housing[**"age\_category"**] = pd.cut(  
 housing[**"house age"**],  
 bins = (5, 20, 40, 80, np.inf),  
 labels = (**"new"**, **"renovated"**, **"mid"**, **"old"**, **"for demolition"**)  
)

#for printing a subset of the dataframe

df.head(num)

df.tail(num)

## Matplotlib

Scientific python library for showing graphs.

**import** matplotlib.pyplot **as** plt

### Lines and 2D functions

plt.figure(**"figure\_title"**, figsize=(10, 12))

plt.axis(**"off"**)



plt.plot([2, 1, 2, 4, 1])  
plt.show()



plt.plot([-3,-2, 1, -1], [2, 4, 1, 2])  
plt.axis([-3, 1, 1, 5]) *# xe[-3, 1], ye[1, 5]*plt.show()



x = np.linspace(-1.4, 1.4, 30)  
plt.plot(x, x, **"g--"**, linewidth=3)  
plt.plot(x, x\*\*2, **"r:"**)  
plt.plot(x, x\*\*3, **"b^"**, alpha=0.1)  
plt.show()



x = np.linspace(-2, 2, 500)  
y = x\*\*2  
plt.plot(x, y)  
plt.title(**"x squared"**)  
plt.xlabel(**"x"**)  
plt.ylabel(**"y"**)  
plt.grid(**True**)  
plt.show()

### Multiple plots on a figure



x = np.linspace(-1.4, 1.4, 30)  
plt.subplot(2,2,1)  
plt.plot(x,x)  
plt.subplot(2,2,2)  
plt.plot(x, x\*\*2)  
plt.subplot(2,1,2)  
plt.plot(x, x\*\*3)  
plt.show()

*subplot(2,2,2):*

*A table made up by two rows and two columns. We put the figure in the 2nd „cell”.*

### Scatter plots, histograms and images



mu, sigma = 100, 15  
x = mu + sigma \* np.random.randn(10000)  
plt.grid(**True**)  
plt.hist(x, bins=50, facecolor=**'green'**, edgecolor=**"black"**, alpha=0.75)  
plt.show()



x = np.linspace(-1.4, 1.4, 30)  
y = x\*\*2  
plt.scatter(x, y, s=200.0, c=**"red"**, alpha=0.9, edgecolors=**"green"**)  
plt.show()

plt.imshow(pic, cmap=**"binary"**)

plt.axis(**"off"**)  
plt.show() *# pic: NxM bitmap*

### Texts and Legends



x = np.linspace(-1.4, 1.4, 30)  
y = x\*\*2  
px = 0  
py = px\*\*2  
  
plt.plot(x, y)  
plt.plot(px, py, **"ro"**)  
plt.text(px, py + 0.05, **"minima"**, fontsize=10, color=**"red"**, horizontalalignment=**"center"**)  
plt.show()



plt.scatter([0.1, 0.15], [0.1, 0.08], color=**"red"**, marker=**"x"**, label=**"safe"**)  
plt.scatter([1, 0.89], [0.95, 0.99], color=**"green"**, marker=**"s"**, label=**"dangerous"**)  
plt.legend(loc=**"upper left"**)  
plt.show()

### ...

## Jupyter notebook

*pip install notebook*

*jupyter notebook (localhost:8888)*



## R

In R console we can include libraries (e.g.: library(datasets))

### Attributes, Basic types, NA and NaN

atomic types: character, numeric, integer, complex, logical

x <- 1 #numeric

x <- 1L #integer

x <- 1/0 #numeric, Inf

x <- 0/0 #numeric, NaN

types gets printed on using class(object) and can be checked with is.numeric(obj)..

type casts: as.numeric(x), as.character(x)

operations with non-ordinary notations: %% (mod), %/% (integer division), ^

logical operations (and, or) are single digit: & |

check if two objects are identical: identical(x, y)

format string (3 to 003): sprintf("%03d", 3)

NA: not available, missing value

NaN: not a number, counts as an NA as well

|  |  |  |
| --- | --- | --- |
|  | **NA** | **NaN** |
| is.na(x) | TRUE | TRUE |
| is.nan(x) | FALSE | TRUE |

### Complex types

vector (for storing elements with the same class):

x <- rep(NA, times=5) #NA NA NA NA NA

x <- rep(c(1, 2), each=2) #1 1 2 2

x <- c("numeric", length=5) #0 0 0 0 0

x <- c(1 + 0i, 2 + 3i) #explicitly telling the elements

x <- c(x, 3 + 2i) #appending to vector

x <- c(col1 = 1, col2 = 2) #named vector (names(object))

x <- 1:5 #int sequence: 1 2 3 4 5

x <- seq(0, 2, by=0.5) #0.0 0.5 1.0 1.5 2.0

x <- seq(0, 1, length=5) #0.0 0.25 0.5 0.75 1

x <- seq\_along(x) #1 2 3 ... length(x)

x <- rnorm(1000) #1000 normally distributed value

x <- sample(x, 100) #selects 100 elements randomly

length of vector: length(x)

list (for storing elements even with different types):

x <- list(1, TRUE, 1+2i)

x <- list(c1=1, c2=TRUE, c3=1+2i)

matrix (vector with set dimension attribute):

x <- matrix(nrow=2, ncol=3) # [NA NA NA]

# [NA NA NA]

x <- matrix(1:6, nrow=2, ncol=3) #column major filling

# [1 3 5]

# [2 4 6]

x <- 1:3 #1 2 3

y <- 6:8 #6 7 8 [1 6]

m1 <- cbind(x, y) #column-wise merge [2 7]

[3 8]

m2 <- rbind(x, yÖ #row-wise merge [1 2 3]

[6 7 8]

x <- 1:10 #redefining the dimension of a vector

dim(x) <- c(2, 5) [1 2 3 4 5]

[6 7 8 9 10]

inverse matrix: solve(m)

transpose matrix: t(m)

true matrix multiplication: m1 %\*% m2

factor (for categorizing data):

dirs <- c("left", "up", "up", "down", "up", "right", "left")

fdirs <- factor(dirs) # left up up down up right left

# Levels: down left right up

# levels are automatically ordered

fdirs2 <- factor(dirs, levels=c("up", "down", "left", "right")

# levels are manually ordered

Factor level generation, gl function (num\_leves, replications, seq):

f <- gl(3, 2, c("NY", "BP", "LA"))

# NY NY BP BP LA LA

# Levels: NY BP LA

On loading tabular data, columns of strings will be interpreted as factors.

data frame (handling tabular data with different types):

empty\_df <- data.frame(matrix(nrow=0, col=3))

colnames(empty\_df) <- c("col1", "col2", "col3")

df <- data.frame(col1 = 1:4, col2 = c(T,F,F,F))

add rows by calling: df <- rbind(df, list(col1 = 1, col2 = F))

add columns by calling: df <- cbind(df, col3 = c("a", "b"))

printing dataframe: head(df, n) tail(df, n)

dimensions of dataframe: nrow(df) ncol(df)

### Subsetting, indexing

[] : return an object of the same class, can be used to select multiple elements

[[]] : can only extract a single element from a list or dataframe, might return another type of object than is was called on

$: same as [[]], can helps us extract a single element by name

vector examples:

x <- c("a", "b", "c", "b", "d", "a")

x[1] #a, indexing starts at 1 in R!

x[1:3] #a b c

y[x %in% 1:3] #every element in y in [1, 3]

x[x < "c"] #a b b a

list examples:

x <- list(foo=1:4, bar=0.6)

x[1] #returns 1:4 sequence as LIST!

x[[1]] #returns 1:4 sequence as raw vector

x$foo #returns 1:4 sequence as raw vector

x[c(1,2)] #returns foo and bar as list

x[c("foo", "bar")] #returns foo and bar as list

x$f #partial matching, autocompleted to foo

matrix examples:

x <- matrix(rep(1, 6), 3, 2)

x[1,1] #returns 1 as numeric

x[1, ] #returns the first row as vector of numerics

x[1, , drop=FALSE] #returns the first row as a matrix

x[c(1, 2), ] #returns the 1st and 2nd row as a matrix

data frame examples:

df <- df[!is.na(df), c1] #returns every non-NA elem under c1

df <- df[complete.cases(df),] #returns NA-free rows

### Date & time

time:

t <- Sys.time() #class(t) POSIXct ("2021-03-08 22:43:50 CET") #unclass(t) integer (seconds since 1970)

t <- as.POSIXlt(t) #class(t) POSIXlt ("2021-03-08 22:43:50 CET")

#unclass(t) list with names: sec, min, ...

t <- strptime("October 17, 1999 08:00", "%B %d, %y %H:%M")

#will create a POSIXlt object

date:

d <- Sys.Date() #class(d) Date ("2021-03-08")

#unclass(d) integer (days since 1970)

d <- as.Date("1969-01-01) #unclass(d) integer (-365)

weekdays(d/t): "Monday"

months(d/t) "March"

quarters(d/t) "Q1"

### Reading data, Files

file.path(path1, path2) #will join path1 and path2

paste(str1, str2, ...) #concatenates strings

read.table ( filename, header=FALSE, sep="\tab", colClasses, nrows,

commentChar="#", skipBegin=0, stringAsFactors=TRUE)

read.csv is the same as read.table but sep is ",".

getwd() return the currect working directory.

source – dump:

y <- data.frame(a=1, b="str")

dput(y, "ser.R")

y <- dget("ser.R")

dget – dput:

Can handle multiple data.

x <- "foo"

y <- c(1, 2, 3)

dump(c(x, y), file="ser.R") #serialization

rm(x, y) #clears variable from enviroment

source("ser.R") #deserialization

### Control structures

if() {} else if {} else {}

y <- if (x < 3) { 10 } else { -10 }

for (i in 1:10)

for (elem in x)

for (i in seq\_along(x)) { next }

while(cnt < 10) { cnt <- cnt + 1}

repeat { ... break ... } #creates infinite loop

### Functions

Usually functions are written in a .R file which is then included to the current environment with sourcing (source("XYZ.R")). The last row of function is the return statement.

Complex examples:

f <- function(x=NA, fun, ...) {

args <- list(...)

y <- if (args[[using.fun]] == TRUE) { fun(args[[data]]) }

else { 0 }

x + y

}

f(c(1, 2), mean, using.func=TRUE, data=c(1, 1))

% bin % <- function(lhs, rhs) {

lhs %% rhs

}

x %bin% y

### Loop functions (apply-s)

lapply (x, fun, ...): Evaluates function on a list’s/vector’s each element.

Always returns a list.

sapply (x, fun, ...): Same as lapply but return the simplest possible class. By simplest class we mean either a single number or a vector.

vapply (x, fun, template, ...): Same as sapply, but checks whether the return type matches the template. If not it halts the program with an error. Template can be numeric(5) for example.

apply (x, margin, fun, ...): Allows us to call functions other than on arrays. The margin parameter’s counterpart in pandas is axis.

apply(m, 1, mean): mean of every row in matrix m (use rowMeans!)

mapply (fun, ..., more.args=NULL, simplitfy=TRUE, use.names=TRUE):

With mapply we can use a function with multiple sets of arguments.

mapply(rep, 1:3, 3:1) #list(rep(1, 3), rep(2, 2), rep(3, 1))

tapply (x, index, fun, ..., simplify=TRUE):

Useful when we need to break up the vector to groups defined by some classifying factor.

x <- c(rnorm(10), runif(10), rnorm(10, 1))

f <- gl(3, 10)

tapply(x, f, mean) #mean of rnorm, runif, rnorm1 in a list

split (x, f, drop=FALSE):

Will break up the vector to grups defined by the classifying factor f.

Will return a list of (# of levels in f) vectors. Can be used to simulate the group by statement.

## SQL (Microsoft Server SQL)

SQL is a Data Query/Definition/Control/Manipulation language which allows us to issue Create, Read, Update, Delete (CRUD) commands to a relational database system.

### Data types

INT(size=255)

DEC(size=10, precision=0)

BOOL

VARCHAR(length)

BLOB # Binary Large OBject

DATA # YYYY-MM-DD

TIMESTAMP # YYYY-MM-DD HH:MM:SS

### Create and alter schemes

CREATE TABLE student (

id INT, -- IDENTITY(1,1) for autoincrement

-- teacher\_id INT,

name VARCHAR(20), -- NOT NULL, UNIQUE

major VARCHAR(20), -- DEFAULT ’unknown’

PRIMARY KEY(student\_id),

-- FOREIGN KEY (teacher\_id) REFERENCES teacher(id)

-- ON DELETE SET NULL

-- ON DELETE CASCADE

);

ALTER TABLE ADD gpa DEC(3,2);

ALTER TABLE DROP COLUMN gpa;

DROP TABLE student;

### Insert rows into table

INSERT INTO student VALUES(1, ’Jack’, ’CS’);

INSERT INTO student(id, name) VALUES(2, ’Kate’);

INSERT INTO student(id) VALUES (3), (4), (5), ... ;

### Delete rows from table

### Data types

## Matlab

// TODO

# Machine Learning

## Types of Machine Learning systems

### Supervised / Unsupervised / Semisupervised / Reinforcement

Supervised: Training set includes the solutions (aka the labels) – Classification –

Unsupervised: Training set is unlabeled – Clustering & Anomaly Detection –

Semisupervised: Combined version of the previous types

Reinforcement learning: Reward based training of an agent

### Batch / Online

Batch: The system must be retrained from scratch with the full dataset on every new version

Online: The system can be trained incrementally by feeding it data instances either individually or in mini-batches

### Instance-based / Model-based

Instance-based: Similarity to the already learnt examples is measured on new cases

Model-based: Build a model from the learnt examples and make predictions according to the model.

## Challenges of Machine Learning

Insufficient quantity of training data

Nonrepresentative training data: training data must be representative of the new cases

Poor-quality: outliers, noise, errors, n/a-s

Redundant features

Overfitting training data (regularization helps)

Underfitting training data (model with more params helps)

## Testing and Validating

Testing how well the model will generalize to new cases is done by splitting the data set into a *training set* and a *test set*. We then train our model with the training set and test its error rate on never previously seen cases from the testing set. This error rate is called the *generalization error*. If we evaluate our model on the training set and get a much lower training error than the generalization error of the corresponding test set then we have the problem of overfitting, meaning that our model only performs well on the training data and the handling of new instances is quite flawed.

Often we hold out another set called the *validation set* to avoid refining (experimenting with regularization hyperparameters) a model to perform well solely on a particular training set. We train the model with the reduced training set (full traning set minus the validation set) and choose the model that produces the lowest generalization error on the validation set. After this process we retrain this model with the full training set and lastly evaluate the final model on the test set to get an estimate of the generalization error.

Cross-validation: The idea is to use multiple validation sets. Each model candidate (SVM, logistic regression etc) is evaluated once per validation set and trained on the rest of the training set. By averaging the evaulation results we get a realistic picture of the performance of out model at the cost of training time.

## Scaling

Machine learning models work better when features are on a similar scale. The most often used scaling method are MinMax scaling [0, 1] and Standard scaling N(0, 1).

**from** sklearn.preprocessing **import** MinMaxScaler, StandardScaler  
  
X = np.array([[1], [2], [3], [4]])

scaler = MinMaxScaler()  
X = scaler.fit\_transform(X)

# Data Science workflow

## Panelek

## Kényelmi funkciók

# Scikit-learn utilities

## Datasets

**from** sklearn.datasets **import** fetch\_xxx / load\_xxx

After fetching use the suitable keys (keys()) to extract the data and the labels.

## Test-train split and cross validation

**from** sklearn.model\_selection **import** cross\_val\_score  
**from** sklearn.model\_selection **import** train\_test\_split  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)  
val\_scores = cross\_val\_score(sgd\_classifier, X\_train, y\_train, cv=3, scoring=**"accuracy"**)

## Clustering

### Agglomerative Clustering

// TODO

### K-Means Clustering

// TODO

### DBSCAN (Density-Based) Clustering

// TODO

## Egyéb

## Egyéb

## Egyéb

# Classification

During classification we are predicting classes.

## Linear model

**from** sklearn.linear\_model **import** SGDClassifier

sgd\_classifier = SGDClassifier()

# Y\_train is an array of True-Falses

sgd\_classifier.fit(X\_train, Y\_train)

Y\_preds = sgd\_classifier.predict(X\_test)

## Confusion Matrix and ROC Curve

**from** sklearn.metrics **import** confusion\_matrix  
**from** sklearn.metrics **import** precision\_score  
**from** sklearn.metrics **import** recall\_score  
**from** sklearn.metrics **import** f1\_score  
  
cfm = confusion\_matrix(y\_train, y\_preds)  
ps = precision\_score(y\_train, y\_preds)  
rs = recall\_score(y\_train, y\_preds)  
fs = f1\_score(y\_train, y\_preds)

Given the task of deciding whether a handwritten digit is 5 or not:

**Precision (3 / 4)**

**Recall (3 / 5)**

\*TN: The letter is actually other than 5 and fortunately the model says the same.

\*\*FP: The letter is actually other 5 but we accidentally classify it as 5.

\*\*\*FN: The letter is actually 5, but we accidentally classify it as other than 5.

\*\*\*\*TP: The letter is actually 5 and luckily the model says so as well.

### Precision

Amongst all the cases the model predicted positive, how many of them is actually positive. (Collecting games that can be played by children. Gather less, but guarantee that those games wont show any gore / adult content)

### Recall

How many amongst all the positive cases in the real world got found by our model. (For example we’d like to get all the patients with viral infections from a group of people. It’s definitely better if someone has to go to the hospital redundatly rather than missing someone with dangerous disease.)

### F1 Score and Precision-Recall tradeoff

Usually, the lower the precision is, the higher the recall is. Thus, we combined these two measures into ’F1 Score’ by taking the harmonic mean of them.

Ideally, we’d like to maximize this.

**from** sklearn.metrics **import** precision\_recall\_curve  
  
y\_scores = sgd\_classifier.decision\_function(X\_test)  
precisions, recalls, thresholds = precision\_recall\_curve(Y\_test\_5, y\_scores)  
plt.title(**"PR curve"**)  
plt.plot(thresholds, recalls[:-1], **"g-"**, label=**"Recall"**)  
plt.plot(thresholds, precisions[:-1], **"b--"**, label=**"Precision"**)  
plt.legend(loc=**"center right"**)  
plt.ylim([0, 1])  
plt.xlabel(**"Threshold"**)  
plt.grid(**"on"**)  
plt.show()

*Receiver operation characteristic curve* (**ROC**) is another tool used with binary classifiers.

ROC curve plots the True Positive Rate (TPR a.k.a. recall or sensitivity) against the False Positive Rate (FPR). FPR is equal to (1 – True Negative Rate). True Negative Rate (specificity) is the ratio of the actually negatively marked instances over all the negative instances of the real world.



We can see the tradeoff here as well. As we find more and more actually positive

cases the false positive rate increases simultaneously. One way to compare classifiers is to measure the AUC (area under the curve). An ideal classifier’s AUC would be close to 1 because we want a classifier that almost touches the upper left corner.

**from** sklearn.metrics **import** roc\_curve  
**from** sklearn.metrics **import** roc\_auc\_score  
  
fpr, tpr, thresholds = roc\_curve(Y\_test\_5, Y\_preds\_5)  
auc\_score = roc\_auc\_score(Y\_test\_5, Y\_preds\_5)  
plt.title(**"Receiver Operating Characteristic"**)  
plt.plot(fpr, tpr, **"b"**, label=**"AUC = {:0.2f}"**.format(auc\_score))  
plt.legend(loc=**"lower right"**)  
plt.plot([0, 1], [0, 1], **"r--"**)  
plt.xlim([0, 1])  
plt.ylim([0, 1])  
plt.ylabel(**"True Positive Rate"**)  
plt.xlabel(**"False Positive Rate"**)  
plt.grid(**"on"**)  
plt.show()

## Multiclass Classification

Doing multiclass classification is not always supported natively by the most frequent classification algorithms. Thus, the problem of multiclass classificiation is often reduced to multiple binary classification problems.

The examples below will show two different approach to solve the 10-digit handwriting recognition problem.

### One versus All (One versus Rest)

Make 10 binary classifier that tells whether the given digit is 0, 1, ..., 9 or not.

On a newly introduced digit we run all these 10 classifiers and select the class whose classifier outputs the highest score.

### One versus One

This method trains a binary classifier for all possible pairs (45 for the current problem)

Run all 45 classifiers on a digit never seen before and choose the class that wins the most.

## Egyéb

# Classification

# Neural Networks

## Forward and Backpropagation

### Math

chain rule:

### Forward propagation on XOR N.N.



### Backpropagation on XOR N.N.

## Egyéb

# Classification

# Classification