Deree

School of Graduate and Professional Education

THE AMERICAN COLLEGE OF GREECE
Pierce
Deree
Alba
1875

ITC6103B1 - Applied Machine Learning

Machine Learning Revision

Dr. Elena Chatzimichali

Deree

School of Graduate and Professional Education

THE AMERICAN COLLEGE OF GREECE
Pierce
Deree
Alba
1875

Pandas cheat sheet

Pandas useful functions

Data Wrangling

with pandas Cheat Sheet

http://pandas.pydata.org

Pandas API Reference Pandas User Guide

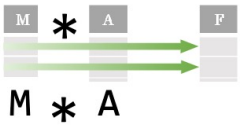
Tidy Data – A foundation for wrangling in pandas

In a tidy data set:

Each **variable** is saved in its own **column**

Each **observation** is saved in its own **row**

Tidy data complements pandas's vectorized operations. pandas will automatically preserve observations as you manipulate variables. No other format works as intuitively with pandas.



Creating DataFrames

	a	b	c
1	4	7	10
2	5	8	11
3	6	9	12

```
df = pd.DataFrame(
        {"a" : [4, 5, 6],
         "b" : [7, 8, 9],
         "c" : [10, 11, 12]},
       index = [1, 2, 3])
```
Specify values for each column.

```
df = pd.DataFrame(
    [[4, 7, 10],
     [5, 8, 11],
     [6, 9, 12]],
    index=[1, 2, 3],
    columns=['a', 'b', 'c'])
```
Specify values for each row.

		a	b	c
N	v			
D	1	4	7	10
	2	5	8	11
e	2	6	9	12

```
df = pd.DataFrame(
        {"a" : [4 ,5, 6],
         "b" : [7, 8, 9],
         "c" : [10, 11, 12]},
index = pd.MultiIndex.from_tuples(
        [('d', 1), ('d', 2),
         ('e', 2)], names=['n', 'v']))
```
Create DataFrame with a MultiIndex

Method Chaining

Most pandas methods return a DataFrame so that another pandas method can be applied to the result. This improves readability of code.

```
df = (pd.melt(df)
        .rename(columns={
                'variable':'var',
                'value':'val'})
        .query('val >= 200')
    )
```

Reshaping Data – Change layout, sorting, reindexing, renaming

pd.melt(df)
Gather columns into rows.

df.pivot(columns='var', values='val')
Spread rows into columns.

pd.concat([df1,df2])
Append rows of DataFrames

pd.concat([df1,df2], axis=1)
Append columns of DataFrames

df.sort_values('mpg')
Order rows by values of a column (low to high).

df.sort_values('mpg', ascending=False)
Order rows by values of a column (high to low).

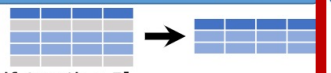
df.rename(columns = {'y':'year'})
Rename the columns of a DataFrame

df.sort_index()
Sort the index of a DataFrame

df.reset_index()
Reset index of DataFrame to row numbers, moving index to columns.

df.drop(columns=['Length', 'Height'])
Drop columns from DataFrame

Subset Observations - rows



df[df.Length > 7]
Extract rows that meet logical criteria.

df.drop_duplicates()
Remove duplicate rows (only considers columns).

df.sample(frac=0.5)
Randomly select fraction of rows.

df.sample(n=10) Randomly select n rows.

df.nlargest(n, 'value')
Select and order top n entries.

df.nsmallest(n, 'value')
Select and order bottom n entries.

df.head(n)
Select first n rows.

df.tail(n)
Select last n rows.

Subset Variables - columns

df[['width', 'length', 'species']]
Select multiple columns with specific names.

df['width'] *or* **df.width**
Select single column with specific name.

df.filter(regex='*regex*')
Select columns whose name matches regular expression *regex*.

Using query

query() allows Boolean expressions for filtering rows.

```
df.query('Length > 7')
df.query('Length > 7 and Width < 8')
df.query('Name.str.startswith("abc")',
         engine="python")
```

Subsets - rows and columns

Use **df.loc[]** and **df.iloc[]** to select only rows, only columns or both.
Use **df.at[]** and **df.iat[]** to access a single value by row and column.
First index selects rows, second index columns.

df.iloc[10:20]
Select rows 10-20.

df.iloc[:, [1, 2, 5]]
Select columns in positions 1, 2 and 5 (first column is 0).

df.loc[:, 'x2':'x4']
Select all columns between x2 and x4 (inclusive).

df.loc[df['a'] > 10, ['a', 'c']]
Select rows meeting logical condition, and only the specific columns .

df.iat[1, 2] Access single value by index

df.at[4, 'A'] Access single value by label

Logic in Python (and pandas)

<	Less than	!=	Not equal to
>	Greater than	df.column.isin(*values*)	Group membership
==	Equals	pd.isnull(*obj*)	Is NaN
<=	Less than or equals	pd.notnull(*obj*)	Is not NaN
>=	Greater than or equals	&,|,~,^,df.any(),df.all()	Logical and, or, not, xor, any, all

regex (Regular Expressions) Examples

'\.'	Matches strings containing a period '.'
'Length$'	Matches strings ending with word 'Length'
'^Sepal'	Matches strings beginning with the word 'Sepal'
'^x[1-5]$'	Matches strings beginning with 'x' and ending with 1,2,3,4,5
'^(?!Species$).*'	Matches strings except the string 'Species'

Cheatsheet for pandas (http://pandas.pydata.org/ originally written by Irv Lustig, Princeton Consultants, inspired by Rstudio Data Wrangling Cheatsheet

Pandas useful functions

Summarize Data

```
df['Length'].value_counts()
```
Count number of rows with each unique value of variable

```
len(df)
```
# of rows in DataFrame.

```
len(df['w'].unique())
```
# of distinct values in a column.

```
df.describe()
```
Basic descriptive statistics for each column (or GroupBy)

pandas provides a large set of **summary functions** that operate on different kinds of pandas objects (DataFrame columns, Series, GroupBy, Expanding and Rolling (see below)) and produce single values for each of the groups. When applied to a DataFrame, the result is returned as a pandas Series for each column. Examples:

`sum()` Sum values of each object.

`count()` Count non-NA/null values of each object.

`median()` Median value of each object.

`quantile([0.25,0.75])` Quantiles of each object.

`apply(function)` Apply function to each object.

`min()` Minimum value in each object.

`max()` Maximum value in each object.

`mean()` Mean value of each object.

`var()` Variance of each object.

`std()` Standard deviation of each object.

Group Data

`df.groupby(by="col")` Return a GroupBy object, grouped by values in column named "col".

`df.groupby(level="ind")` Return a GroupBy object, grouped by values in index level named "ind".

All of the summary functions listed above can be applied to a group. Additional GroupBy functions:

`size()` Size of each group.

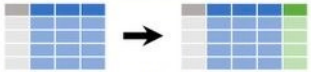
`agg(function)` Aggregate group using function.

Handling Missing Data

```
df=df.dropna()
```
Drop rows with any column having NA/null data.

```
df=df.fillna(value)
```
Replace all NA/null data with value.

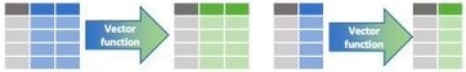
Make New Variables



```
df=df.assign(Area=lambda df: df.Length*df.Height)
```
Compute and append one or more new columns.

```
df['Volume'] = df.Length*df.Height*df.Depth
```
Add single column.

```
pd.qcut(df.col, n, labels=False)
```
Bin column into n buckets.



pandas provides a large set of **vector functions** that operate on all columns of a DataFrame or a single selected column (a pandas Series). These functions produce vectors of values for each of the columns, or a single Series for the individual Series. Examples:

`max(axis=1)` Element-wise max.

`min(axis=1)` Element-wise min.

`clip(lower=-10,upper=10)` Trim values at input thresholds

`abs()` Absolute value.

The examples below can also be applied to groups. In this case, the function is applied on a per-group basis, and the returned vectors are of the length of the original DataFrame.

`shift(1)` Copy with values shifted by 1.

`rank(method='dense')` Ranks with no gaps.

`rank(method='min')` Ranks. Ties get min rank.

`rank(pct=True)` Ranks rescaled to interval [0, 1].

`rank(method='first')` Ranks. Ties go to first value.

`shift(-1)` Copy with values lagged by 1.

`cumsum()` Cumulative sum.

`cummax()` Cumulative max.

`cummin()` Cumulative min.

`cumprod()` Cumulative product.

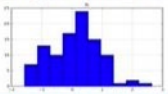
Windows

```
df.expanding()
```
Return an Expanding object allowing summary functions to be applied cumulatively.

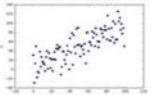
```
df.rolling(n)
```
Return a Rolling object allowing summary functions to be applied to windows of length n.

Plotting

`df.plot.hist()` Histogram for each column



`df.plot.scatter(x='w',y='h')` Scatter chart using pairs of points



Combine Data Sets

adf

x1	x2
A	1
B	2
C	3

+

bdf

x1	x3
A	T
B	F
D	T

=

Standard Joins

x1	x2	x3
A	1	T
B	2	F
C	3	NaN

```
pd.merge(adf, bdf,
         how='left', on='x1')
```
Join matching rows from bdf to adf.

x1	x2	x3
A	1.0	T
B	2.0	F
D	NaN	T

```
pd.merge(adf, bdf,
         how='right', on='x1')
```
Join matching rows from adf to bdf.

x1	x2	x3
A	1	T
B	2	F

```
pd.merge(adf, bdf,
         how='inner', on='x1')
```
Join data. Retain only rows in both sets.

x1	x2	x3
A	1	T
B	2	F
C	3	NaN
D	NaN	T

```
pd.merge(adf, bdf,
         how='outer', on='x1')
```
Join data. Retain all values, all rows.

Filtering Joins

x1	x2
A	1
B	2

```
adf[adf.x1.isin(bdf.x1)]
```
All rows in adf that have a match in bdf.

x1	x2
C	3

```
adf[~adf.x1.isin(bdf.x1)]
```
All rows in adf that do not have a match in bdf.

ydf

x1	x2
A	1
B	2
C	3

+

zdf

x1	x2
B	2
C	3
D	4

=

Set-like Operations

x1	x2
B	2
C	3

```
pd.merge(ydf, zdf)
```
Rows that appear in both ydf and zdf (Intersection).

x1	x2
A	1
B	2
C	3
D	4

```
pd.merge(ydf, zdf, how='outer')
```
Rows that appear in either or both ydf and zdf (Union).

x1	x2
A	1

```
pd.merge(ydf, zdf, how='outer',
         indicator=True)
.query('_merge == "left_only"')
.drop(['_merge'],axis=1)
```
Rows that appear in ydf but not zdf (Setdiff).

http://pandas.pydata.org/ This cheat sheet inspired by Rstudio Data Wrangling Cheatsheet (https://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf) Written by Irv Lustig, Princeton Consultants

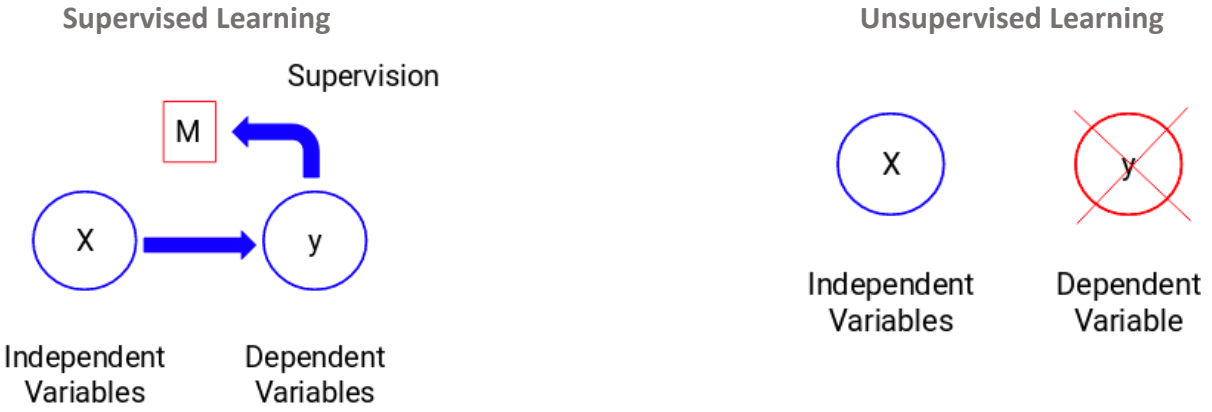
Deree

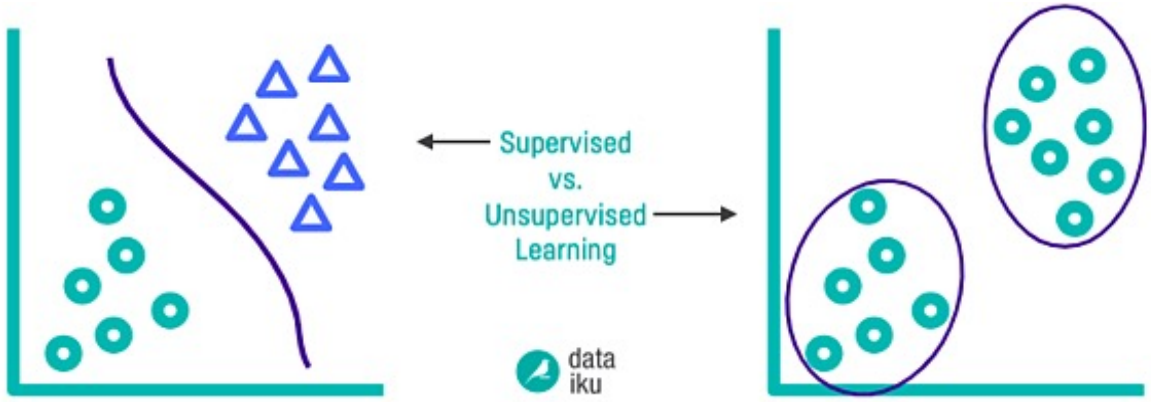
School of Graduate and Professional Education

THE AMERICAN COLLEGE OF GREECE
Pierce
Deree
Alba
1875

Supervised vs. Unsupervised Learning

Supervised vs. Unsupervised Learning





Source https://www.analyticsvidhya.com/blog/2019/05/beginners-guide-hierarchical-clustering/ and https://www.altexsoft.com/blog/unsupervised-machine-learning/

Supervised vs. Unsupervised Learning

A recap of scikit-learn's estimator interface

Scikit-learn strives to have a uniform interface across all methods, and we'll see examples of these below. Given a scikit-learn *estimator* object named `model`, the following methods are available:

In all Estimators:

- `model.fit()` : fit training data. For supervised learning applications, this accepts two arguments: the data `X` and the labels `y` (e.g. `model.fit(X, y)`). For unsupervised learning applications, this accepts only a single argument, the data `X` (e.g. `model.fit(X)`).

In supervised estimators:

- `model.predict()` : given a trained model, predict the label of a new set of data. This method accepts one argument, the new data `X_new` (e.g. `model.predict(X_new)`), and returns the learned label for each object in the array.
- `model.predict_proba()` : For classification problems, some estimators also provide this method, which returns the probability that a new observation has each categorical label. In this case, the label with the highest probability is returned by `model.predict()`.
- `model.score()` : for classification or regression problems, most (all?) estimators implement a score method. Scores are between 0 and 1, with a larger score indicating a better fit.

In unsupervised estimators:

- `model.transform()` : given an unsupervised model, transform new data into the new basis. This also accepts one argument `X_new`, and returns the new representation of the data based on the unsupervised model.
- `model.fit_transform()` : some estimators implement this method, which more efficiently performs a fit and a transform on the same input data.

Source https://scipy-lectures.org/packages/scikit-learn/index.html

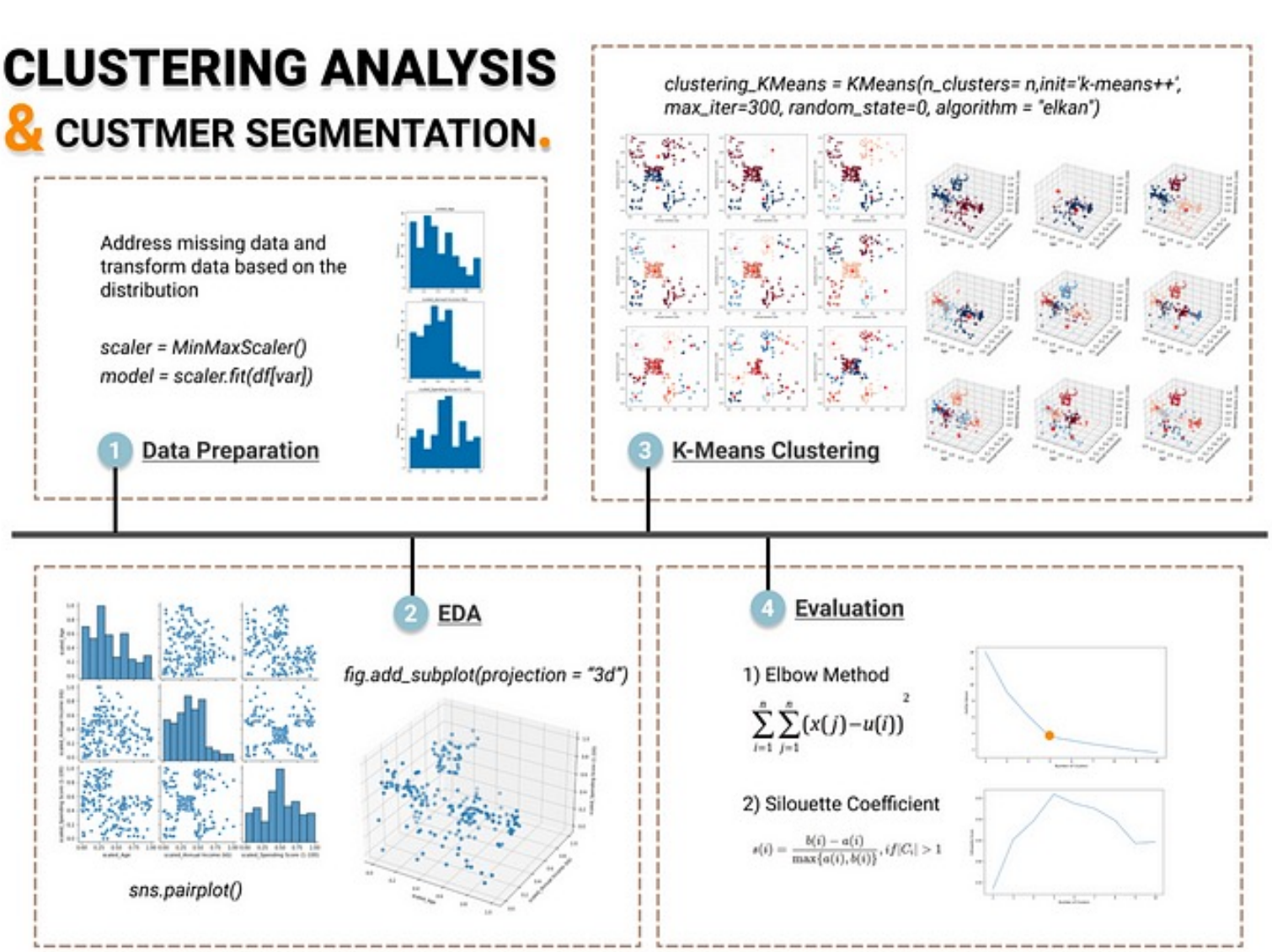
Deree

School of Graduate and Professional Education

THE AMERICAN COLLEGE OF GREECE
Pierce
Deree
Alba
1875

Unsupervised Learning

Unsupervised Learning steps



Source https://towardsdatascience.com/clustering-algorithm-for-customer-segmentation-e2d79e28cbc3

Unsupervised Learning steps

Step1: Loading (Libraries), dataset ingestion and data overview

```
df = pd.read_csv("../input/heart-disease-uci/heart.csv")
df.head()
```

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
0	63	1	3	145	233	1	0	150	0	2.3	0	0	1	1
1	37	1	2	130	250	0	1	187	0	3.5	0	0	2	1
2	41	0	1	130	204	0	0	172	0	1.4	2	0	2	1
3	56	1	1	120	236	0	1	178	0	0.8	2	0	2	1
4	57	0	0	120	354	0	1	163	1	0.6	2	0	2	1

Use **df.info()** to have a summarized view of dataset, including **data type, missing data and number of records.**

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 303 entries, 0 to 302
Data columns (total 14 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   age       303 non-null    int64
 1   sex       303 non-null    int64
 2   cp        303 non-null    int64
 3   trestbps  303 non-null    int64
 4   chol      303 non-null    int64
 5   fbs       303 non-null    int64
 6   restecg   303 non-null    int64
 7   thalach   303 non-null    int64
 8   exang     303 non-null    int64
 9   oldpeak   303 non-null    float64
 10  slope     303 non-null    int64
 11  ca        303 non-null    int64
 12  thal      303 non-null    int64
 13  target    303 non-null    int64
dtypes: float64(1), int64(13)
memory usage: 33.3 KB
```

Source https://towardsdatascience.com/top-machine-learning-algorithms-for-classification-2197870ff501

Unsupervised Learning steps

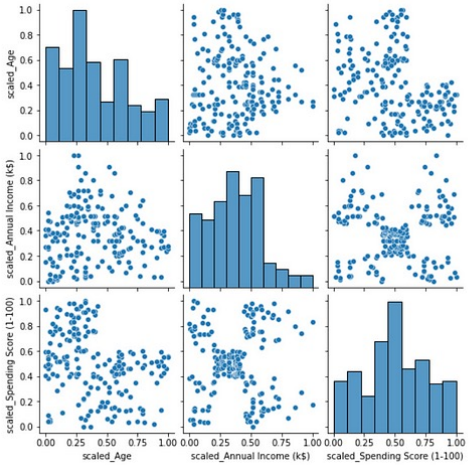
Step2: Exploratory Data Analysis (get a sense of the data using – mostly – visualisations)

```
df.describe()
```

	CustomerID	Age	Annual Income (k$)	Spending Score (1-100)
count	200.000000	200.000000	200.000000	200.000000
mean	100.500000	38.850000	60.560000	50.200000
std	57.879185	13.969007	26.264721	25.823522
min	1.000000	18.000000	15.000000	1.000000
25%	50.750000	28.750000	41.500000	34.750000
50%	100.500000	36.000000	61.500000	50.000000
75%	150.250000	49.000000	78.000000	73.000000
max	200.000000	70.000000	137.000000	99.000000

```
# 2D scatter plot
import seaborn as sns
columns = ["scaled_Age","scaled_Annual Income (k$)", "scaled_Spending Score (1-100)"]
sns.pairplot(df[columns])
```

```
<seaborn.axisgrid.PairGrid at 0x7fd170290850>
```



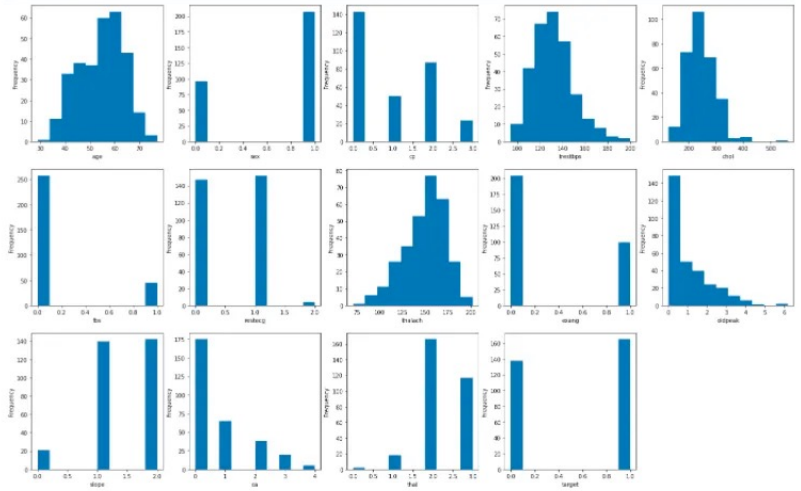
Source https://towardsdatascience.com/clustering-algorithm-for-customer-segmentation-e2d79e28cbc3

Supervised Learning steps

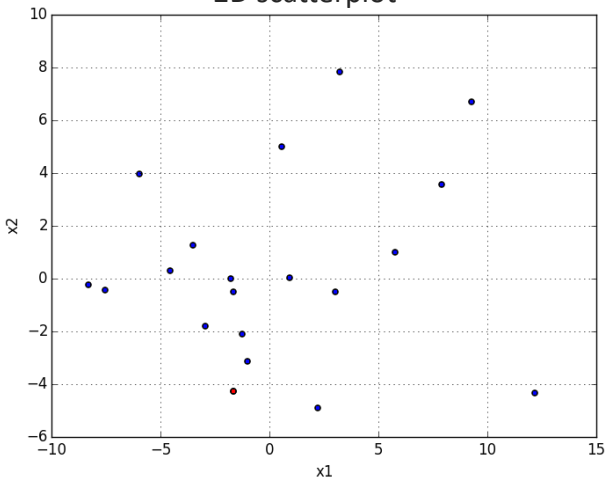
Step2: Exploratory Data Analysis (get a sense of the data using – mostly – visualisations)

Histograms

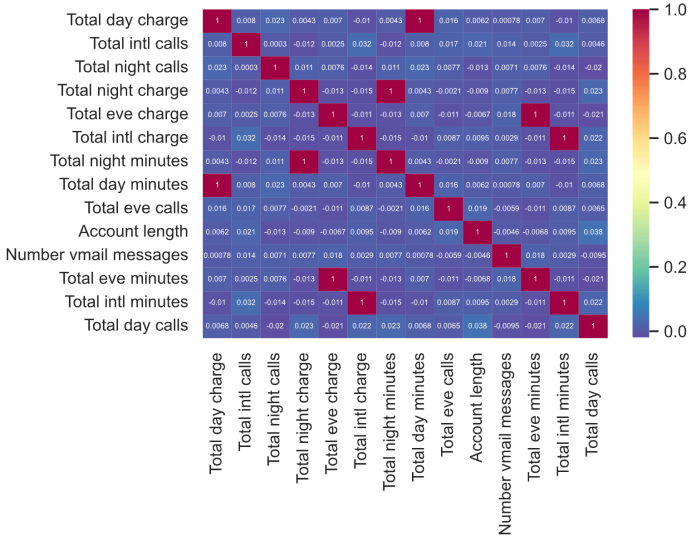
```
fig = plt.figure(figsize=(24, 15))
i = 0
for column in df:
    sub = fig.add_subplot(3, 5, i + 1)
    sub.set_xlabel(column)
    df[column].plot(kind = 'hist')
    i = i + 1
```



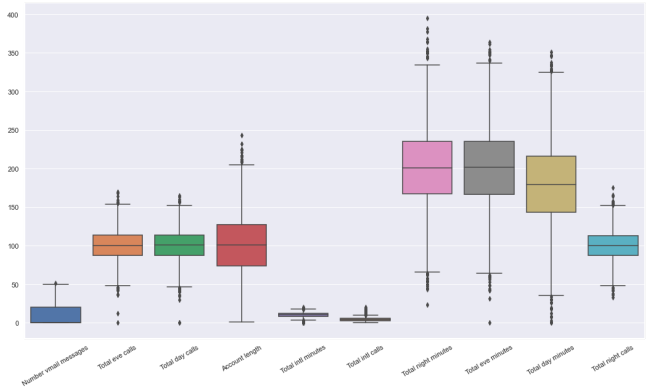
2D scatterplot



Heatmap of correlations



Boxplot



Unsupervised Learning

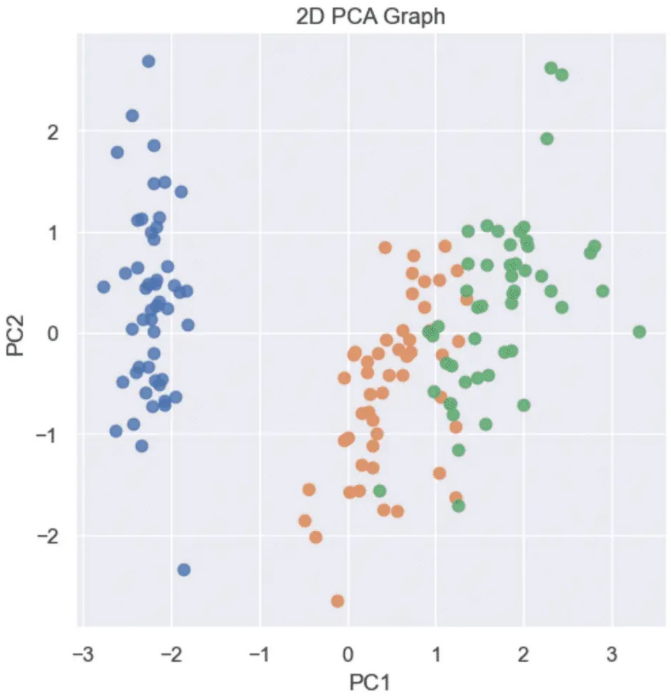
Step3: Data pre-processing

Some of the most commonly-applied steps of data pre-processing, are:

- Drop / filter unnecessary features that may not be relevant or do not contribute to the analysis (e.g. IDs)
- Check and handle missing values and/or outliers
- Encode categorical variables
- Scale the data
- PCA: (if needed, in cases of high-dimensional, highly-correlated features) Apply PCA (after scaling!)

```
from sklearn.decomposition import PCA

pca = PCA(n_components=3)

pca_features = pca.fit_transform(x_scaled)

print('Shape before PCA: ', x_scaled.shape)
print('Shape after PCA: ', pca_features.shape)

pca_df = pd.DataFrame(
    data=pca_features,
    columns=['PC1', 'PC2', 'PC3'])
```

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()

sns.lmplot(
    x='PC1',
    y='PC2',
    data=pca_df,
    hue='target',
    fit_reg=False,
    legend=True
    )

plt.title('2D PCA Graph')
plt.show()
```



Source https://towardsdatascience.com/clustering-algorithm-for-customer-segmentation-e2d79e28cbc3

Unsupervised Learning

Step4: Apply a clustering model

```python
# define K means algorithm function that return inertia, label, centroids and silhouetee score

from sklearn.cluster import KMeans
from sklearn import metrics

def KMeans_Algorithm(dataset, n):
    clustering_KMeans = KMeans(n_clusters= n,init='k-means++', max_iter=300, random_state=0, algorithm = "elkan")
    clustering_KMeans.fit(dataset)

    # create data frame to store centroids
    centroids  = clustering_KMeans.cluster_centers_
    centroids_df = pd.DataFrame(centroids, columns =['X', 'Y'])

    # add cluster label for each data point
    label = clustering_KMeans.labels_
    df["label"] = label

    # evaluation metrics for clustering - inertia and silhouette score
    inertia = clustering_KMeans.inertia_
    silhouette_score = metrics.silhouette_score(dataset, label)

    return inertia, label, centroids_df, silhouette_score
```

This function calls the *KMeans()* from sklearn library. Some key parameters:

- ***n_clusters:*** the number of clusters
- ***init:*** it controls initialization techniques. “k-means++” is used to select initial centroids in a smart way to speed up convergence
- ***max_iter:*** specify the maximum number of iterations allowed for each run
- ***algorithm****:* there are several options to choose from, “auto”, “full” and “elkan” and we chose “elkan” because it is a K Means algorithm variation that uses triangle inequality to make it more time efficient.
- ***random_state:*** use an integer to determine the random generation of initial centroids

The output will generate the following attributes and evaluation metrics:

- ***cluster_centres_:*** returns an array of the centroid locations
- ***inertia_:*** sum of squared distances of data points to their closest centroid
- ***labels_:*** the cluster label assigned to each data point
- ***silhouette_score:*** the distance between the data point to other data points in the same cluster compares to the data points in the nearest neighbour cluster

Source https://towardsdatascience.com/clustering-algorithm-for-customer-segmentation-e2d79e28cbc3

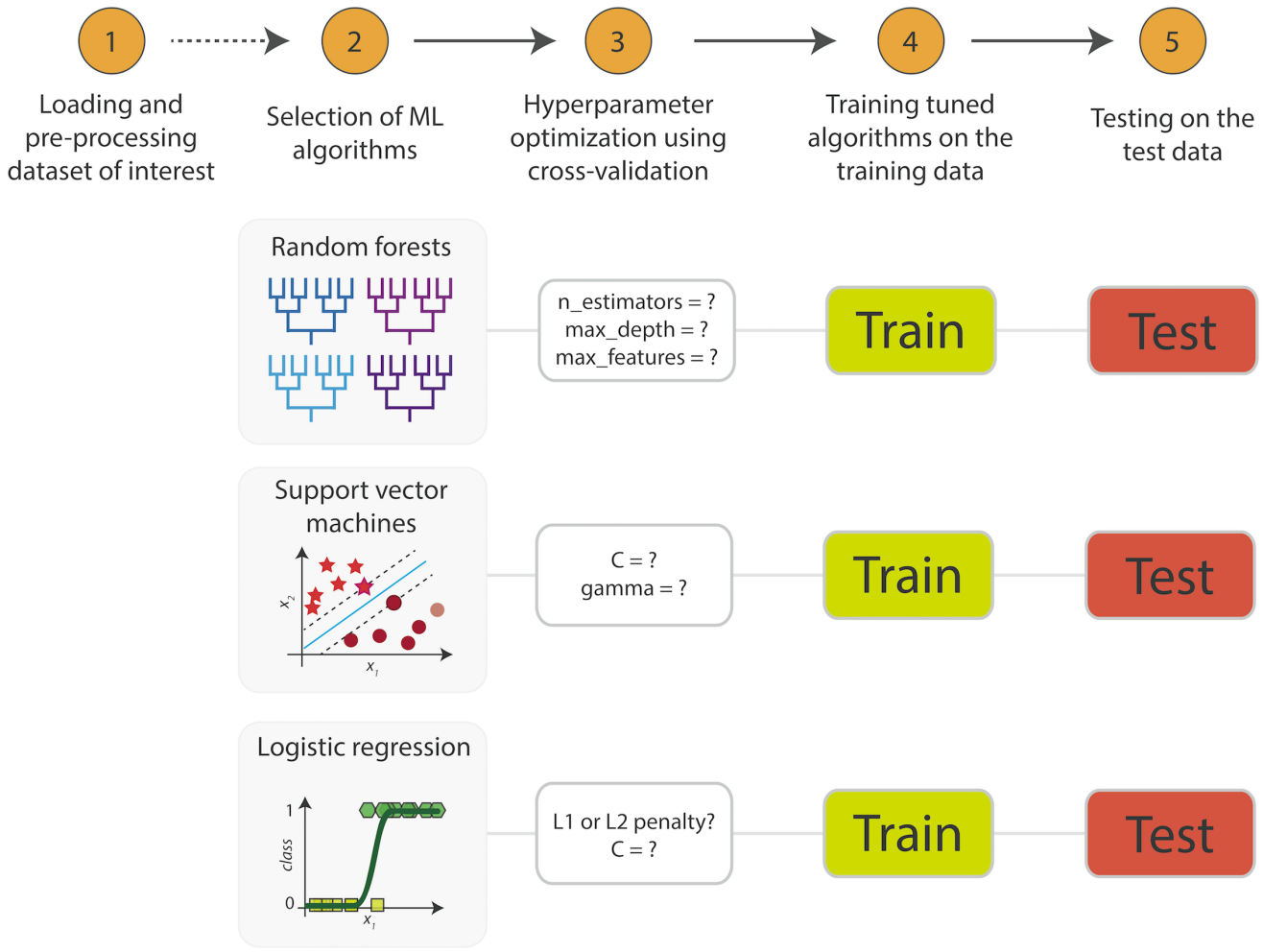
Deree

School of Graduate and Professional Education

THE AMERICAN COLLEGE OF GREECE
Pierce
Deree
Alba
1875

Supervised Learning

Supervised Learning steps



Source https://cambridgecoding.wordpress.com/2016/05/16/expanding-your-machine-learning-toolkit-randomized-search-computational-budgets-and-new-algorithms-2/

Supervised Learning steps

Step1: Loading (Libraries), dataset ingestion and data overview

```
df = pd.read_csv("../input/heart-disease-uci/heart.csv")
df.head()
```

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
0	63	1	3	145	233	1	0	150	0	2.3	0	0	1	1
1	37	1	2	130	250	0	1	187	0	3.5	0	0	2	1
2	41	0	1	130	204	0	0	172	0	1.4	2	0	2	1
3	56	1	1	120	236	0	1	178	0	0.8	2	0	2	1
4	57	0	0	120	354	0	1	163	1	0.6	2	0	2	1

Use **df.info()** to have a summarized view of dataset, including **data type, missing data and number of records.**

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 303 entries, 0 to 302
Data columns (total 14 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   age       303 non-null    int64
 1   sex       303 non-null    int64
 2   cp        303 non-null    int64
 3   trestbps  303 non-null    int64
 4   chol      303 non-null    int64
 5   fbs       303 non-null    int64
 6   restecg   303 non-null    int64
 7   thalach   303 non-null    int64
 8   exang     303 non-null    int64
 9   oldpeak   303 non-null    float64
 10  slope     303 non-null    int64
 11  ca        303 non-null    int64
 12  thal      303 non-null    int64
 13  target    303 non-null    int64
dtypes: float64(1), int64(13)
memory usage: 33.3 KB
```

Source https://towardsdatascience.com/top-machine-learning-algorithms-for-classification-2197870ff501

Supervised Learning steps

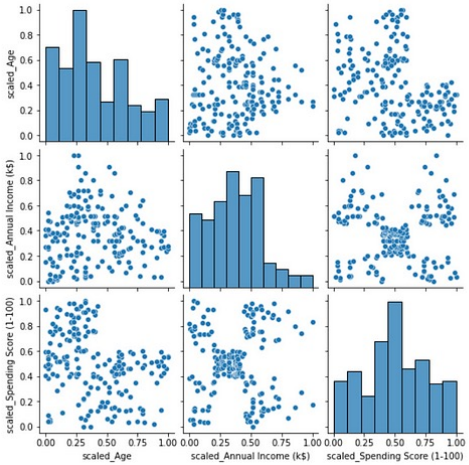
Step2: Exploratory Data Analysis (get a sense of the data using – mostly – visualisations)

```
df.describe()
```

	CustomerID	Age	Annual Income (k$)	Spending Score (1-100)
count	200.000000	200.000000	200.000000	200.000000
mean	100.500000	38.850000	60.560000	50.200000
std	57.879185	13.969007	26.264721	25.823522
min	1.000000	18.000000	15.000000	1.000000
25%	50.750000	28.750000	41.500000	34.750000
50%	100.500000	36.000000	61.500000	50.000000
75%	150.250000	49.000000	78.000000	73.000000
max	200.000000	70.000000	137.000000	99.000000

```
# 2D scatter plot
import seaborn as sns
columns = ["scaled_Age","scaled_Annual Income (k$)", "scaled_Spending Score (1-100)"]
sns.pairplot(df[columns])
```

```
<seaborn.axisgrid.PairGrid at 0x7fd170290850>
```



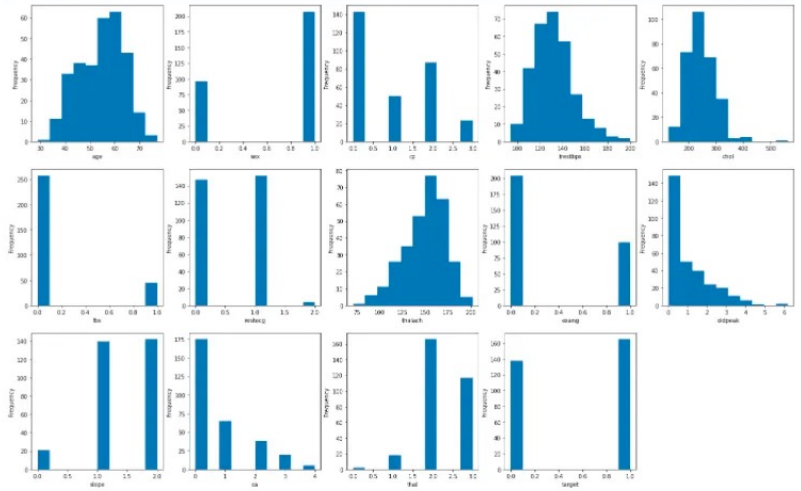
Source https://towardsdatascience.com/clustering-algorithm-for-customer-segmentation-e2d79e28cbc3

Supervised Learning steps

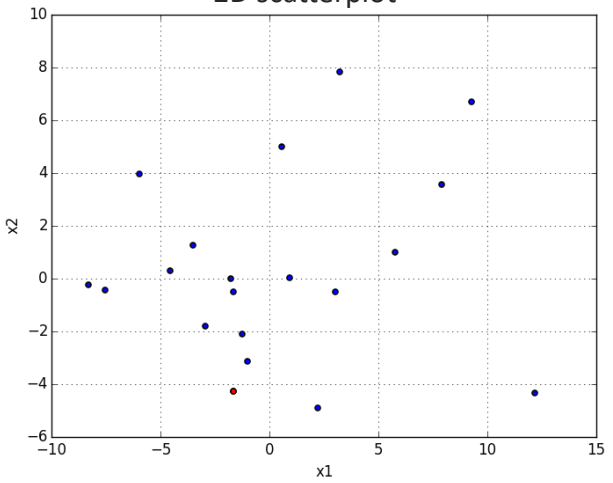
Step2: Exploratory Data Analysis (get a sense of the data using – mostly – visualisations)

Histograms

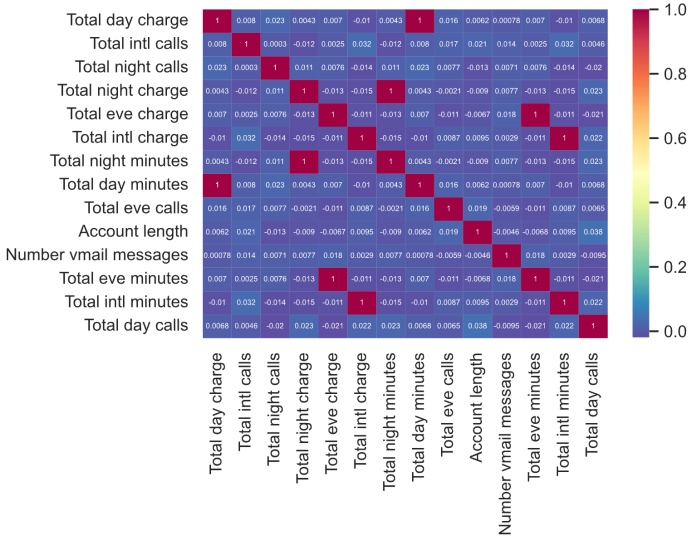
```
fig = plt.figure(figsize=(24, 15))
i = 0
for column in df:
    sub = fig.add_subplot(3, 5, i + 1)
    sub.set_xlabel(column)
    df[column].plot(kind = 'hist')
    i = i + 1
```



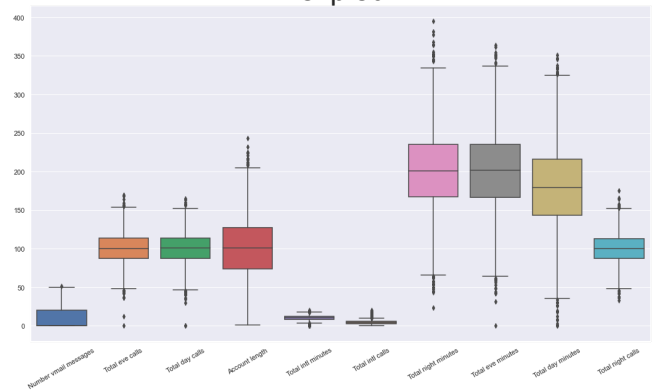
2D scatterplot



Heatmap of correlations



Boxplot



Supervised Learning steps

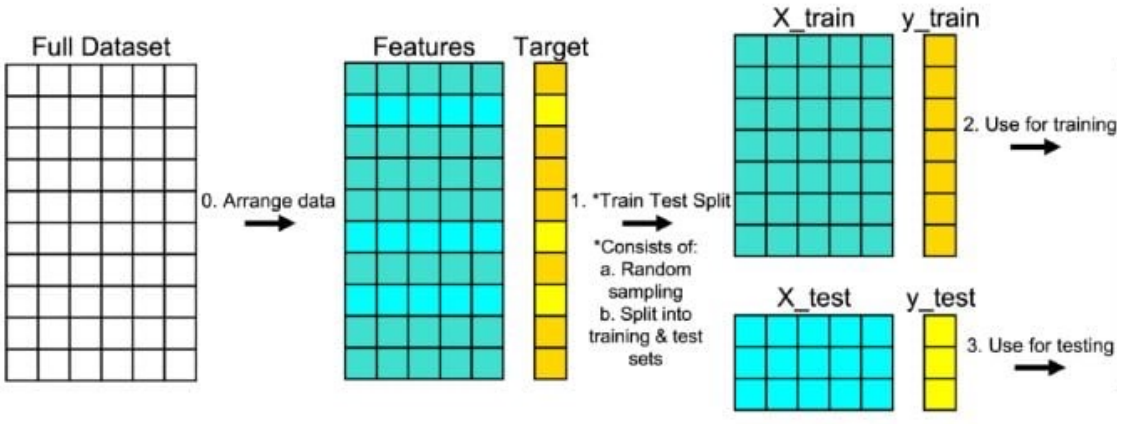
Step3: Split into independent (X) and dependent (y) variables

Next, we want separate the independent predictor variables and the target variable into X and y respectively. This step could be done prior to (or after) conducting EDA and visualizations.

Note: remember to **encode** (if needed in cases of non-numeric values) the y variable as well as screen the **class frequencies for imbalances**.

Step4: Train/Test split

Classification algorithm falls under the category of supervised learning, so dataset needs to be split into a subset for training and a subset for testing (to check the **generalization** performance). The model is trained on the training set and then examined using the test set.



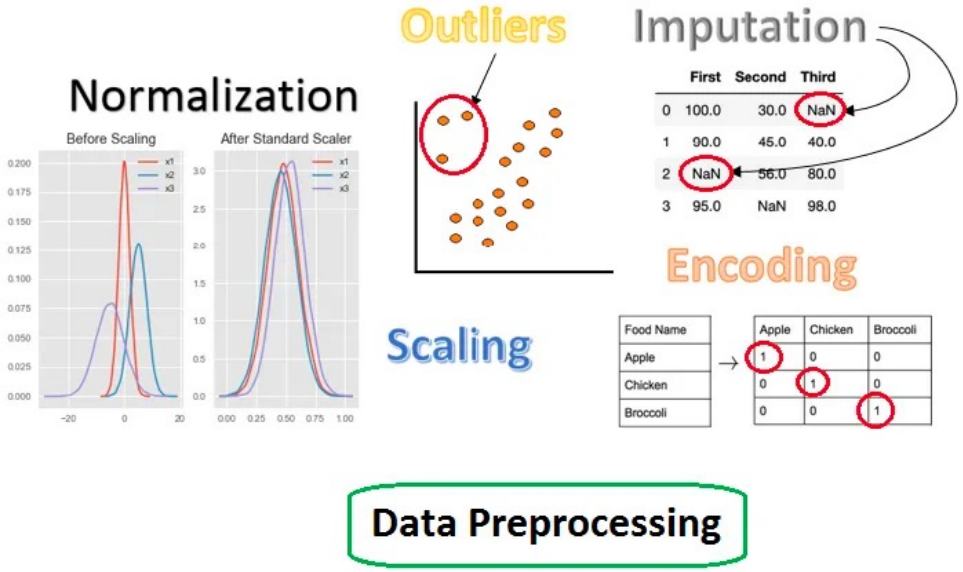
Source https://builtin.com/data-science/train-test-split

Supervised Learning steps

Step5: Data pre-processing

Note: There are some data fixes (e.g. dropping unnecessary features) and pre-processing techniques that can be applied early on in the analysis. However, for data pre-processing techniques that make use of the statistical properties of our data (such as scaling), it is crucial to apply them after the train/test split to avoid data leakage.

In this case, we **fit_transform (fit = learn) the training set** and **only transform the test set.**



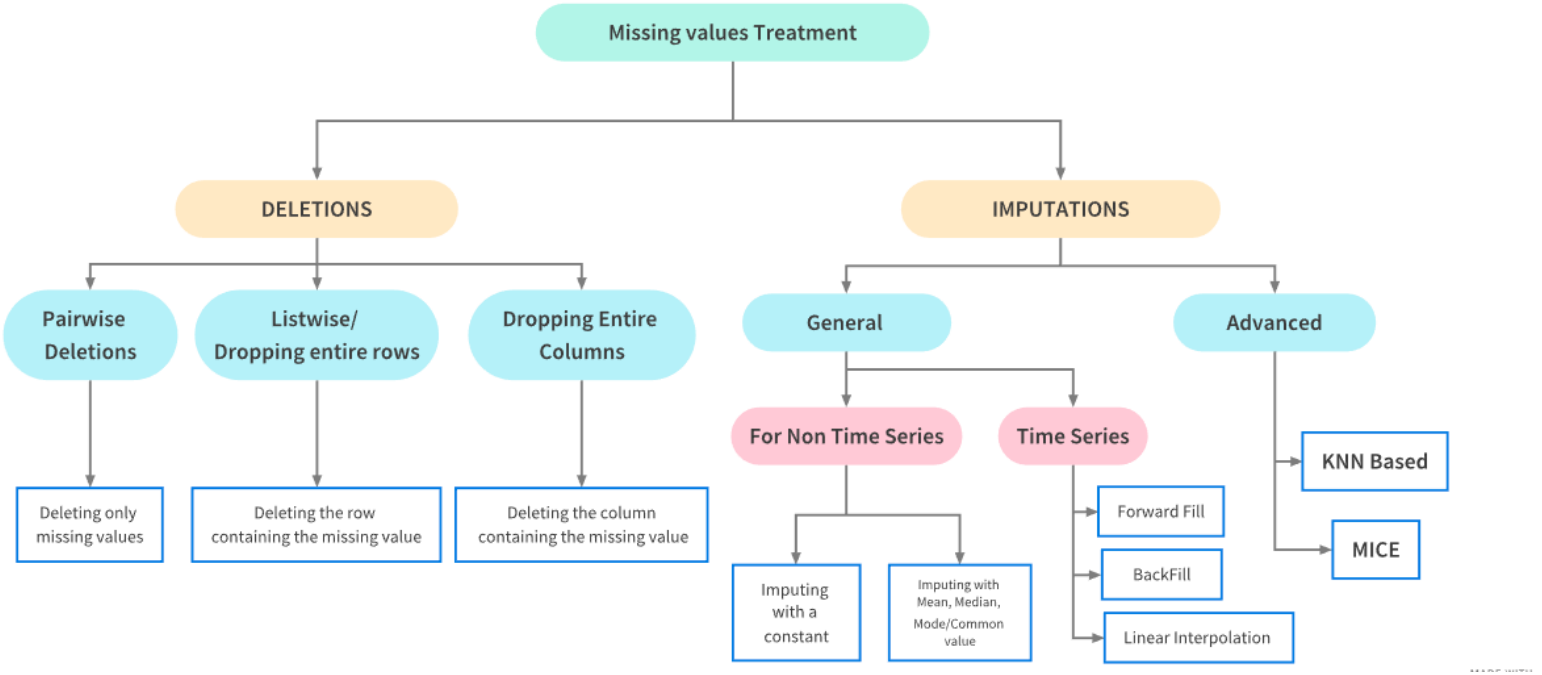
Source https://www.kaggle.com/code/parulpandey/a-guide-to-handling-missing-values-in-python

Supervised Learning steps

Step5: Data pre-processing

1. **Drop redundant columns (or set as index in cases where it makes sense) such as Customer IDs**

2. **Missing value treatment (dealing with NAs)**



Source https://www.kaggle.com/code/parulpandey/a-guide-to-handling-missing-values-in-python

Supervised Learning steps

Step5: Data pre-processing

2. Missing value treatment (continued)

- Remove rows/columns with missing values (naïve but simple)
- Impute missing values
  - Fill in with prior knowledge
  - Fill in with a constant (such as 0, distinct from all other values)
  - A value from another randomly selected record
  - A mean, median or mode value for the column
  - A value estimated by another predictive model

Any imputing performed on the **training dataset** will also have to be performed **on new data** in the future when predictions are needed from the finalized model. This needs to be taken into consideration when choosing how to impute the missing values.

PassengerId	Survived	Pclass	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
1	0	3	male	22	1	0	A/5 21171	7.25		S
2	1	1	female	38	1	0	PC 17599	71.2833	C85	C
3	1	3	female	26	0	0	STON/O2. 3101282	7.925		S
4	1	1	female	35	1	0	113803	53.1	C123	S
5	0	3	male	35	0	0	373450	8.05		S
6	0	3	male		0	0	330877	8.4583		Q

Supervised Learning steps

Step5: Data pre-processing

3. Scaling

Many machine learning algorithms perform better or converge faster when features are on a relatively similar scale and/or close to normally distributed. **Remember(!), the scaler needs to be fitted only on the train set and sequentially used to transform both the train and test sets to avoid data leakage.** Some of the commonly used scaling techniques are:

Min Max Scaler — normalization

MinMaxScaler() is usually applied **when the dataset is not distorted.** It normalizes the data into a range between 0 and 1 based on the formula:

$$x' = (x - min(x)) / (max(x) - min(x))$$

Standard Scaler — standardization

We use standardization when the dataset conforms to **normal distribution.** *StandardScaler()* converts the numbers into the standard form of **mean = 0 and variance = 1** based on z-score formula:

$$x' = (x - mean) / standard\ deviation.$$

Supervised Learning steps

Step5: Data pre-processing

3. PCA: when developing a Supervised Learning model, if PCA is needed as a pre-processing step (due to high-dimensionality and highly-correlated data), it should be applied after the train/test split and scaling. PCA is instantiated and **fitted only on the train data set**, followed by transformations in both the train and test set, which are then fed into a classifier / regressor.

```
# Split the dataset
X_train, X_test, y_train, y_test = model_selection.train_test_split(X_array, y_array, test_size=0.3, random_state=42)

# Instantiate the StandardScaler
scaler = StandardScaler()

# Fit on the train set only
scaler.fit(X_train) # could be done in one step as fit_transform (only for train)

# Apply to both the train set and the test set.
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)

# Instantiate a PCA object
pca = PCA(n_components=2)

# Fit on the train set only (after scaling!)
pca.fit(X_train) # could be done in one step as fit_transform (only for train)

# Apply transform to both the train set and the test set (after scaling!)
X_train_pca = pca.transform(X_train)
X_test_pca   = pca.transform(X_test)
explained_variance = pca.explained_variance_ratio_

# Feed to a classifier e.g. SVMs
clf = svm.SVC(C=5., gamma=0.001)
clf.fit(X_train_pca, y_train)
```

Supervised Learning steps

Step5: Data pre-processing and data leakage

From the official scikit-learn documentation:

Data leakage occurs when information that would not be available at prediction time is used when building the model. This results in overly optimistic performance estimates and thus poorer performance when the model is used on actually novel data, for example during production.

A common cause is not keeping the test and train data subsets separate. Test data should never be used to make choices about the model. **The general rule is to never call fit (or fit_transform) on the test data.** While this may sound obvious, this is easy to miss in some cases, for example when applying certain pre-processing steps.

Although both train and test data subsets should receive the same pre-processing transformation, it is important that these transformations are only learnt from the training data.

For example, if you have a normalization step where you divide by the average value, the average should be the average of the train subset, **not** the average of all the data. If the test subset is included in the average calculation, information from the test subset is influencing the model.

Source https://scikit-learn.org/stable/common_pitfalls.html

Supervised Learning steps

Step5: Data pre-processing and data leakage

Below are some tips on avoiding data leakage:

- Always split the data into train and test subsets first, **particularly before any pre-processing steps.**
- **Never include test data when using the fit() and fit_transform() methods**. Using all the data, e.g., fit(X), can result in data leakage and overly optimistic scores.
- Conversely, the **transform() method should be used on both train and test subsets as the same preprocessing should be applied to all the data**. This can be achieved by using fit_transform() on the train subset and transform() on the test subset.
- The scikit-learn pipeline is a great way to prevent data leakage as it ensures that the appropriate method is performed on the correct data subset. The pipeline is ideal for use in cross-validation and hyper-parameter tuning functions.

Source https://scikit-learn.org/stable/common_pitfalls.html

Supervised Learning steps

Step6: Choosing the classification model and creating a benchmark (baseline) model

Create a benchmark model of a classifier using either:

- the default hyperparameters
- OR a set of pre-defined hyperparameters

to get a sense of the model performance prior to optimizing/tuning it.
Remember: we ALWAYS **fit()** on the training data and **predict()** the test data!

4 main steps for the creation of a Supervised Learning model

1. Instantiate a model with the default hyperparameters

rf_model = RandomForestClassifier()

2. Fit (learn) on the training data

rf_model.fit(X_train,y_train)

3. Predict the test data

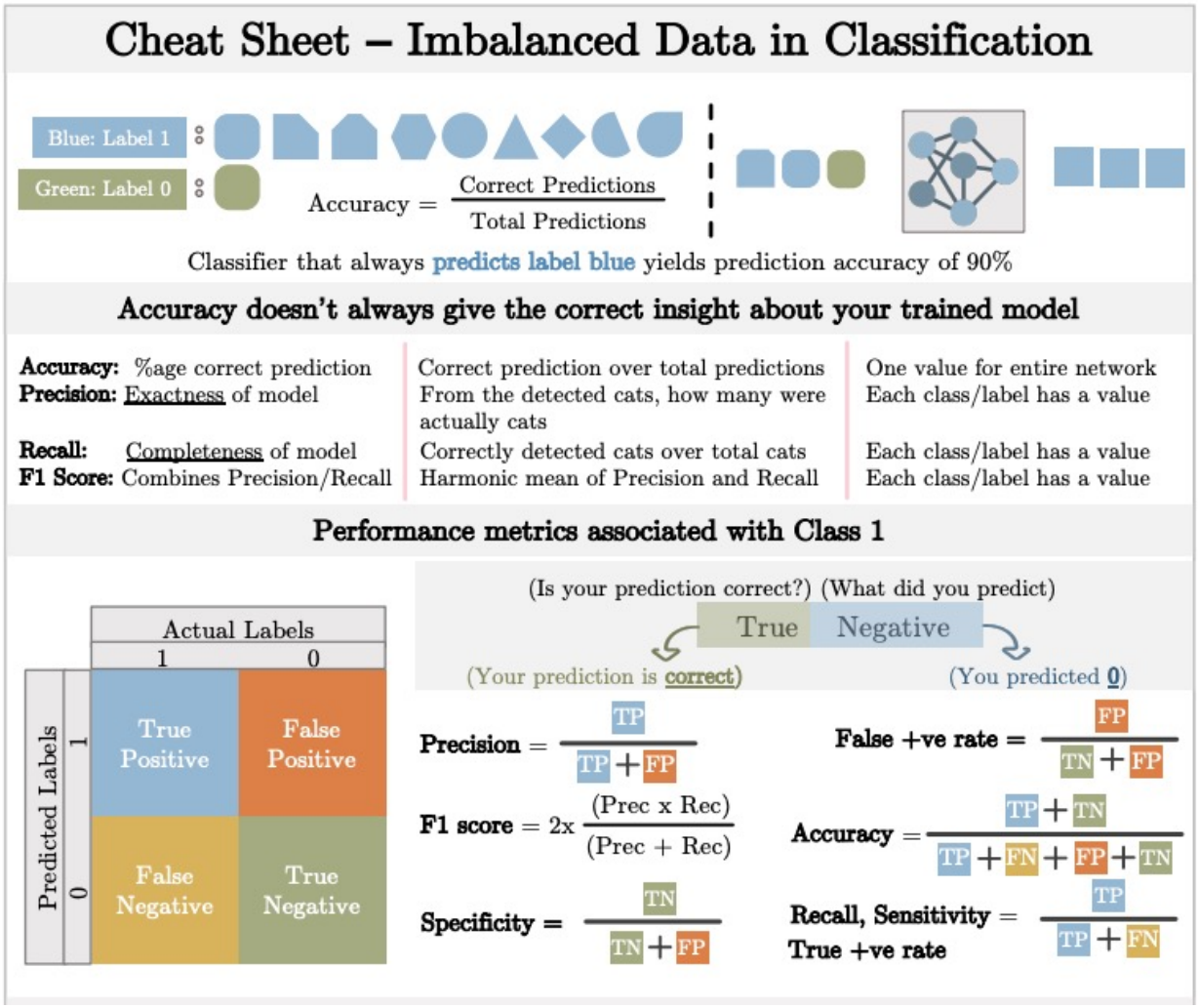
y_pred = rf_model.predict(X_test)

4. Report the classification performance (metrics)

metrics.accuracy_score(y_test, y_pred)

Supervised Learning steps

Step7: Performance metrics



Source https://medium.com/analytics-vidhya/hyper-parameter-tuning-gridsearchcv-vs-randomizedsearchcv-499862e3ca5

Supervised Learning steps

Step8: Hyperparameter optimization (tuning)

Hyperparameter optimization is required to get the most out of your ML models.

What are hyperparameters?
Hyper parameters are like “handles” available to control the output or the behavior of the algorithm used for modelling. They can be supplied to algorithms as arguments.
For example, here the criterion entropy is the hyperparameter passed.

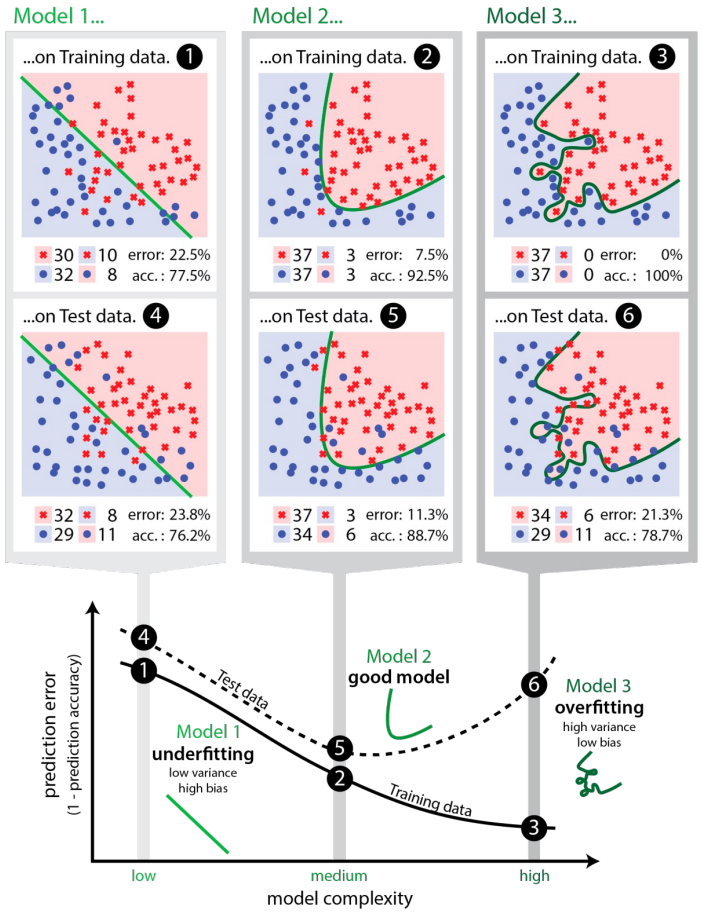
model= DecisionTreeClassifier(criterion=’entropy’)

Source https://medium.com/analytics-vidhya/hyper-parameter-tuning-gridsearchcv-vs-randomizedsearchcv-499862e3ca5

Supervised Learning steps

Bias-Variance trade-off

High bias (underfitting): if the model is too simple (overly-simplified model), the solution is biased and does not fit the data well. In this case, we have high error on both train and test data.



	Bias	Variance	Complexity	Flexibility	Generalizability
Underfitting: you have an overly simple model	High	Low	Low	Low	High
Overfitting: your model is modelling the noise	Low	High	High	High	Low

High variance (overfitting): if the model is too (overly) complex, then it is very sensitive to small changes in the data (starts modelling the noise in the input). In this case, the train error is low and the test error high.

Source https://cambridgecoding.wordpress.com/2016/03/24/misleading-modelling-overfitting-cross-validation-and-the-bias-variance-trade-off/

Supervised Learning steps

Step8: Hyperparameter optimization (tuning)

In order to search the best values in hyper parameter space, we can use:

- **GridSearchCV** (considers **exhaustively all possible combinations** of hyperparameters)
  Grid Search is one of the most basic hyper parameter technique used and so their implementation is quite simple. All possible permutations of the hyper parameters for a particular model are used to build models. The performance of each model is evaluated and the best performing one is selected. Since GridSearchCV uses each and every combination to build and evaluate the model performance, this method is highly computational expensive.

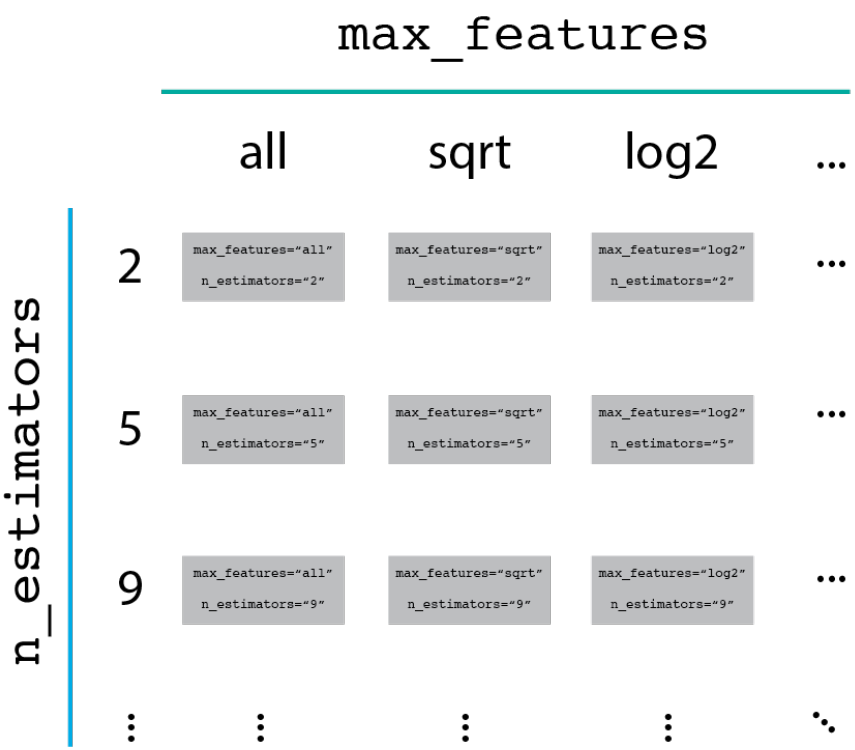
- **RandomizedSearchCV** (**only few samples are randomly selected**)
  In RandomizedSearchCV, instead of providing a discrete set of values to explore on each hyperparameter, we provide a statistical distribution or list of hyper parameters. Values for the different hyper parameters are picked up at random from this distribution. RandomizedSearchCV is very useful when we have many parameters to try and the training time is very long (for example, in the case of Random Forests) but the problem that it doesn't guarantee to give the best parameter combination because not all parameter values are tried out

Both techniques evaluate models for a given hyperparameter vector using **cross-validation, hence the "*CV*"** suffix of each class name.

Supervised Learning steps

Step8: Hyperparameter optimization (tuning)

Define a **dictionary of hyperparameters** that you will pass to Grid/RandomizedSearchCV.
For example:

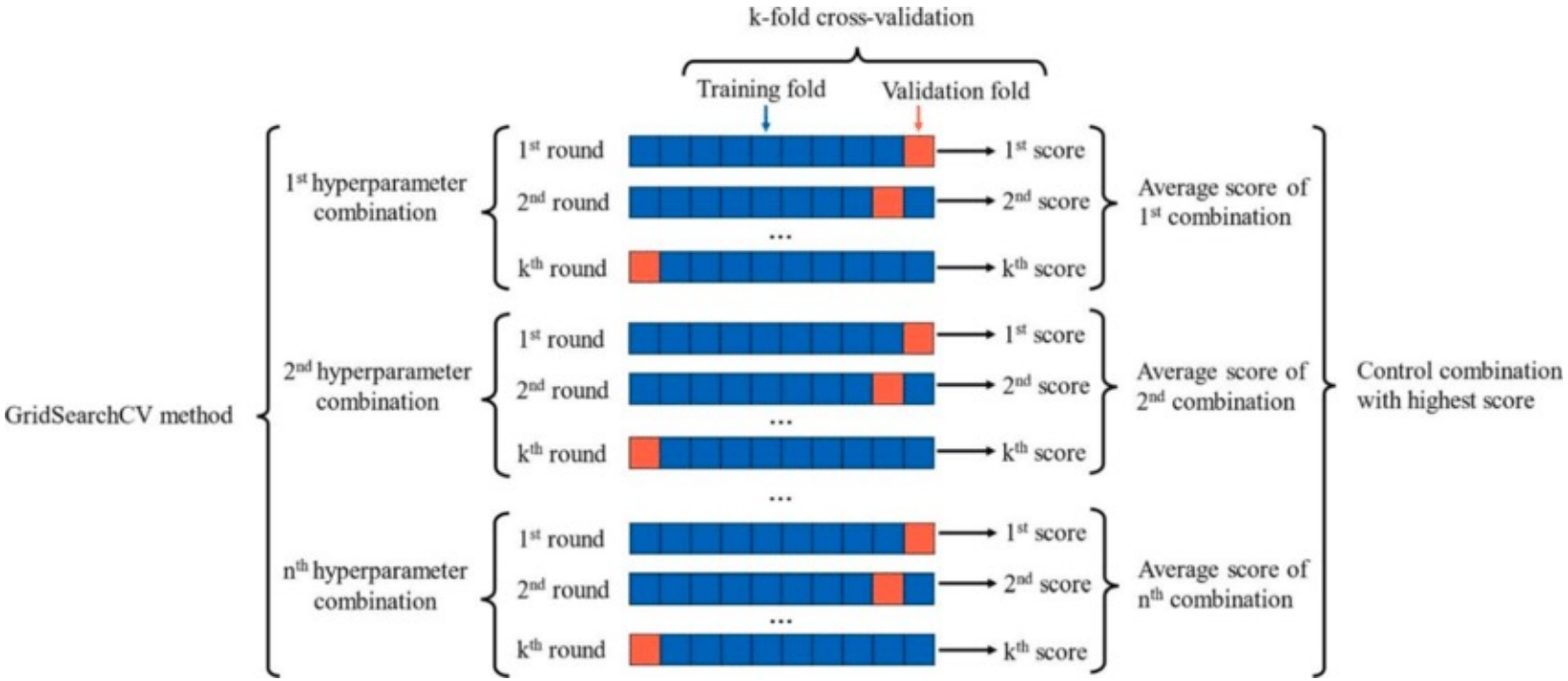


Source https://cambridgecoding.wordpress.com/2016/04/03/scanning-hyperspace-how-to-tune-machine-learning-models/

Supervised Learning steps

Step8: Hyperparameter optimization (tuning)
GridSearchCV breakdown

The grid search algorithm trains multiple models (one for each combination of hyperparameters), calculates the average cross-validation score (e.g. accuracy) for each hyperparameter combination and finally retains the ***best (optimal)*** set of hyperparameter values that achieves the highest score.

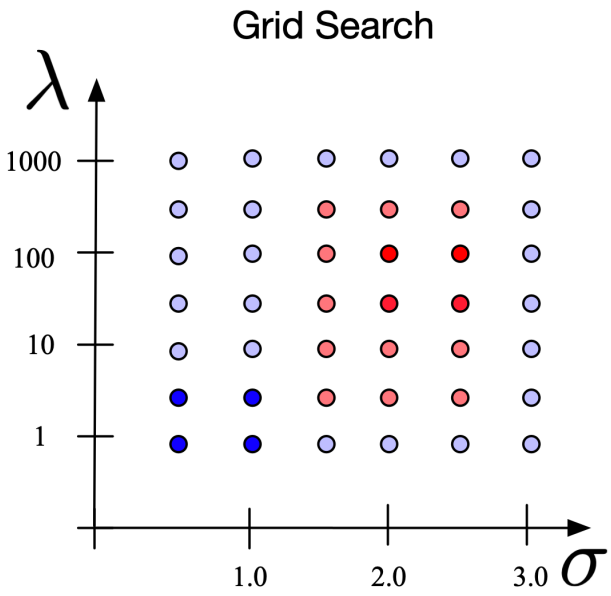


Source https://www.researchgate.net/figure/Hyperparameter-tuning-using-five-fold-cross-validation-GridSearchCV_fig1_360189662

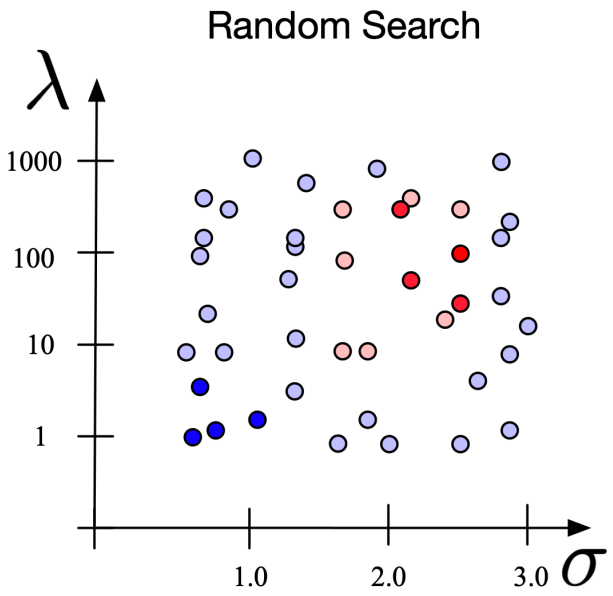
Supervised Learning steps

Step8: Hyperparameter optimization (tuning)
GridSearchCV vs. RandomizedSearchCV

The point of the grid that **maximizes the average cross-validation score (e.g. accuracy)**, is the optimum combination of values for the hyperparameters.



GridSearchCV: Exhausts all combinations of parameters



RandomizedSearchCV: Randomly selects the combinations of parameters

Source https://www.cs.cornell.edu/courses/cs4780/2021fa/lectures/lecturenote11.html

Supervised Learning steps

Step8: Hyperparameter optimization (tuning)
GridSearchCV vs. RandomizedSearchCV parameters

Common parameters of GridSearchCV:

- **estimator**: Our model instance.
- **params_grid**: A dictionary that holds the hyperparameters we wish to experiment with.
- **scoring**: Evaluation metric that we want to implement, e.g. Accuracy, F1macro, F1micro.
- **cv**: The total number of cross-validations we perform for each hyperparameter.
- **n_jobs**: number of processes you wish to run in parallel for this task. If it is -1 it will use all available processors.

Common parameters of RandomizedSearchCV:

- **estimator**: Our model instance.
- **param_distributions:** Dictionary with parameters names as keys and distributions or lists of parameters to search.
- **scoring:** Evaluation metric that we want to implement, e.g. Accuracy, F1macro, F1micro.
- **n_iter**: It specifies the number of combinations to try randomly. Selecting too low of a number will decrease our chance of finding the best combination. Selecting too large of a number will increase the processing time. So, it trades off run time vs quality of the solution.
- **cv:** The total number of cross-validations we perform for each hyperparameter.

Deree

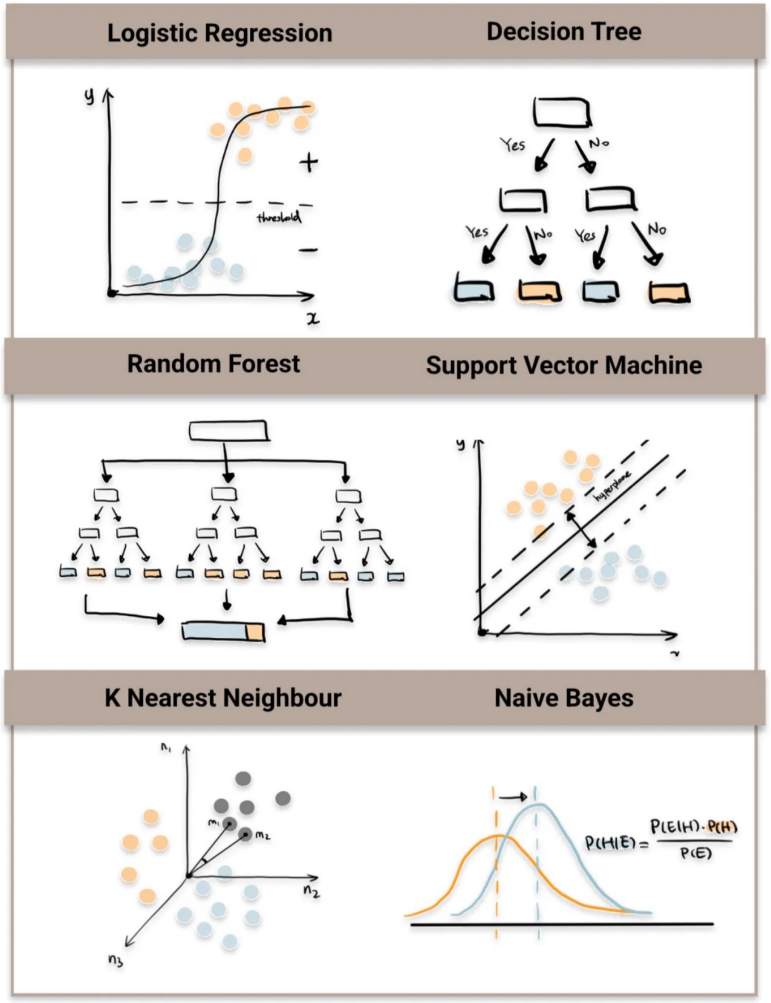
School of Graduate and Professional Education

THE AMERICAN COLLEGE OF GREECE
Pierce
Deree
Alba
1875

Classifiers in detail

Classifiers

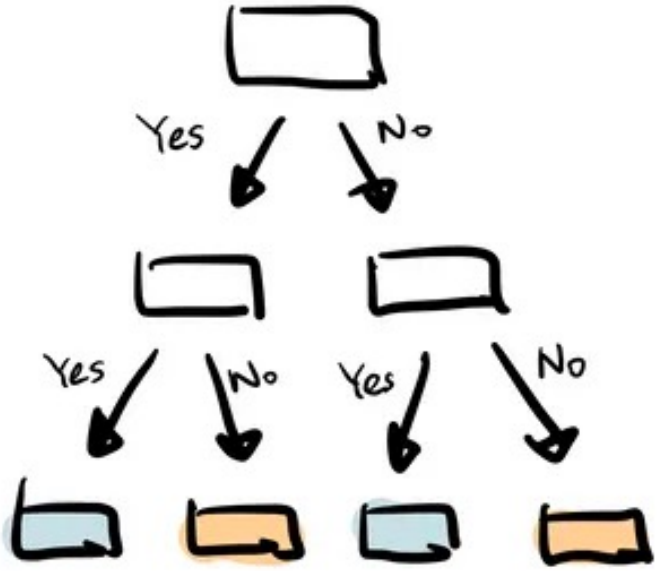
There's a huge selection of classifiers. Some of the most commonly-used are:



Source https://towardsdatascience.com/top-machine-learning-algorithms-for-classification-2197870ff501

Decision Trees

Decision tree builds tree branches in a hierarchical approach, where each branch can be considered as an if-else statement. The branches develop by partitioning the dataset into subsets based on most important features. Final classification happens at the leaves of the decision tree. Decision trees are prone to over-fitting.




```
from sklearn.tree import DecisionTreeClassifier
dtc = DecisionTreeClassifier()
dtc.fit(X_train, y_train)
y_pred = dtc.predict(X_test)
```

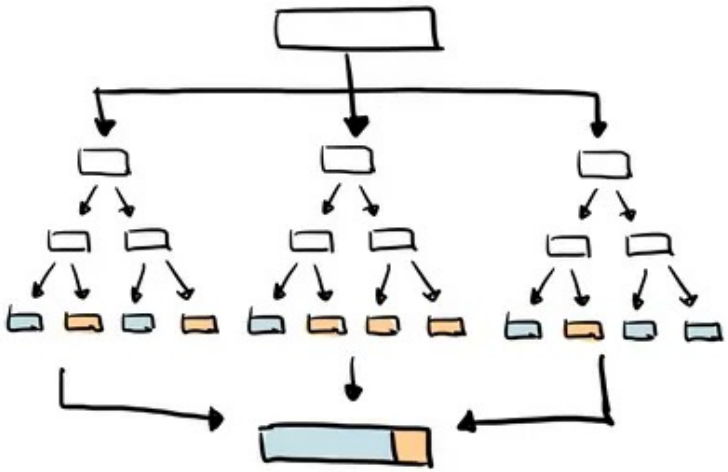
decision tree common hyperparameters: criterion, max_depth, min_samples_split, min_samples_leaf; max_features

Note: start by tweaking at least max_depth

Source https://towardsdatascience.com/top-machine-learning-algorithms-for-classification-2197870ff501

Random Forests

Random forests is a collection of decision trees. It is a common type of ensemble methods which aggregate results from multiple predictors. Random forests additionally utilize bagging technique that allows each tree to be trained on a random sampling of original dataset and takes the majority vote from trees. Compared to decision trees, it has better generalization but it is less interpretable since more layers are added to the model.



```
from sklearn.ensemble import RandomForestClassifier
rfc = RandomForestClassifier()
rfc.fit(X_train, y_train)
y_pred = rfc.predict(X_test)
```

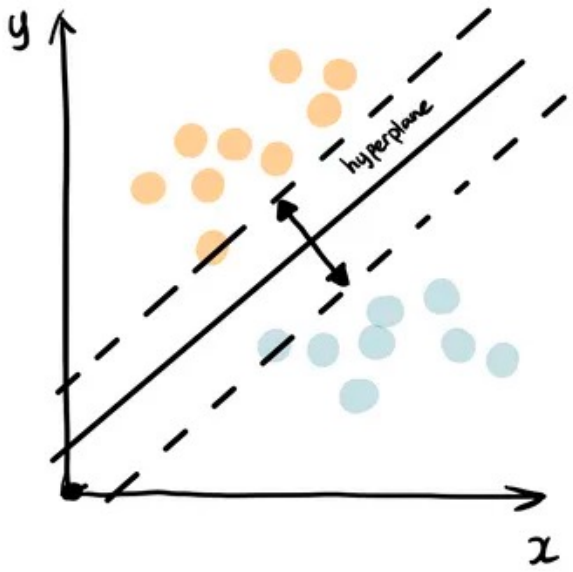
random forest common hyperparameters: n_estimators, max_features, max_depth, min_samples_split, min_samples_leaf, boostrap

Note: start by tweaking at least n_estimators and secondly max_depth

Source https://towardsdatascience.com/top-machine-learning-algorithms-for-classification-2197870ff501

Support Vector Machines

Support vector machines attempts to find the best way to classify the data by constructing a boundary (hyperplane) that is maximising the distance (margin) between the closest data points from different classes. Similar to decision tree and random forest, support vector machine can be used in both classification and regression, SVC (support vector classifier) is for classification problem




```
from sklearn.svm import SVC
svc = SVC()
svc.fit(X_train, y_train)
y_pred = svc.predict(X_test)
```

support vector machine common hyperparameters: c, kernel, gamma

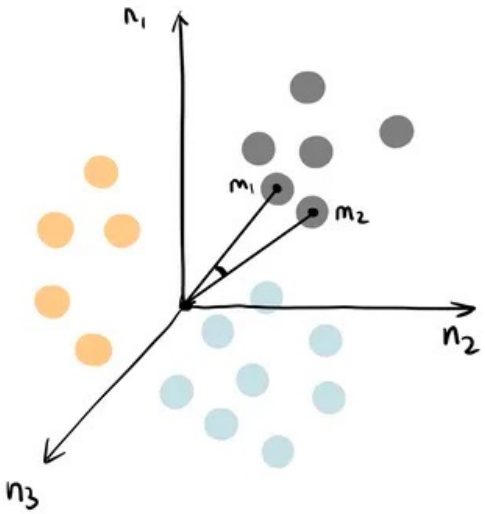
Note: with the right combination of C and gamma, the RBF kernel (default in sklearn) can approximate the linear kernel (however, the linear kernel is much faster and only has the hyperparameter C to tune). So RBF as a kernel is always a good choice to try in SVMs; in this case, we need to tweak C and gamma. Other kernels like polynomial and sigmoid are harder to tune.

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

Source https://towardsdatascience.com/top-machine-learning-algorithms-for-classification-2197870ff501

K Nearest Neighbors (KNN)

KNN classifiers are instance-based (lazy learners). Defer the decision to generalize until a new test instance in encountered. They do not construct models using boundaries (unlike eager learners such as decision trees etc.). Instead, it stores instances of the training data in memory and computes the classification based on a simple or weighted majority vote. The main drawback of KNN is the high computational cost. This is mainly due to calculations of the nearest neighbors for each new sample.




```
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier()
knn.fit(X_train, y_train)
y_pred = knn.predict(X_test)
```

KNN common hyperparameters: n_neighbors, weights, leaf_size, p

Note: use an odd number for n_neighbors. Weights can be 'uniform' or 'distance', and you can also set the hyperparameter 'metric' to other distance types/metrics.

Source https://towardsdatascience.com/top-machine-learning-algorithms-for-classification-2197870ff501