Machine Learning

Data Preprocessing

Machine learning (ML) models need to be trained on big collections of data (**datasets**). Usually, the bigger the dataset available to train a model, the better the model is. In Python, ML algorithms usually implemented with scikit-learn↗ library.

Dataset loading

Datasets usually organized as a matrix x where each row is a **sample** (N samples) and each column corresponds to a **feature** (M features). They may also contain a column y with **target** values or labels (supervised learning). In this case, values in y are the property we want to correlate with features in x.

```
# Import dataset from Python library
from sklearn.datasets import load_diabetes
dataset = load_diabetes()
X = dataset.data
Y = dataset.target
feature_names = dataset.feature_names
```

Dataset partition

ML models are developed in stages that use different subsets of original dataset:

- **Training**: model uses data to learn relationships between features and targets, usually defined by some parameters (weights and bias).
- **Validation**: model is improved by tuning *hyperparameters* using subset of training data. Hyperparameters are parameters that control the learning process of the model.
- **Testing**: data used is unseen for the model during previous stages, provides an estimate of the model performance. The model should be unchanged by these data (no tuning or improving).

```
# Separate dataset into training and testing partitions
from sklearn.model_selection import train_test_split
X_trn, X_tst, Y_trn, Y_tst = train_test_split(X, Y, test_size=0.4)
```

Data imputation

When database contains incomplete entries (missing values, not-a-number, NaN), they are substituted by a valid value taken from the dataset (mean, mode, closest value, probabilistic model…).

Data normalization

Usually data in x needs to be normalized (not y). Both training and testing subsets are normalized with respect to training subset. Each feature normalized independently (subtract mean and divide by standard deviation).

```
# Normalize data
from sklearn.preprocessing import StandardScaler
scl = StandardScaler().fit(X_trn) # scaler object
X_trn_norm = scl.transform(X_trn)
X_tst_norm = scl.transform(X_tst)
```

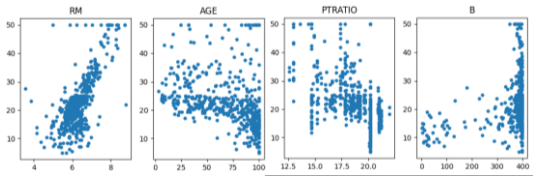
Streamlined preprocessing

- **Pipeline**: Different preprocessing and training steps of a ML model are streamlined into a single process.

```
# Implement pipeline process
from sklearn.pipeline import Pipeline
polynomial = ('pol', PolynomialFeatures(include_bias=False))
scaler     = ('scl', StandardScaler())
selector   = ('sel', SelectKBest(f_regression))
regressor  = ('reg', Lasso())
pipe = Pipeline([polynomial, scaler, selector, regressor])
pipe.fit(X_trn, Y_trn)  # find best features, learn normalization params and train regressor
f = pipe.predict(X_tst) # expand features, normalize data, select best features and predict output
```

FEATURE ANALYSIS

Features are studied and compared to find relevant or redundant ones. *Correlation* between values of a feature and the target is proportional to the feature's relevance. Each feature can be plotted against target to visually assess their correlation.




```
# Obtain correlation score
from sklearn.feature_selection import f_regression
score,_ = f_regression(X_trn, Y_trn)
```

Feature selection

Irrelevant features are removed to reduce amount of data. To find them, features are ranked by how *informative* they are and the most informative ones are selected. In regression, features selected should contribute to define the target value. In classification, features selected should differentiate among classes and have low in-class variability and large between-class variability. Information theory (entropy, mutual information) can quantify relevance of each feature.

```
# Select best features
from sklearn.feature_selection import SelectKBest
k = 3                                # number of features to select
sel = SelectKBest(f_regression, k=k) # feature selector object
sel.fit(X_trn, Y_trn)                # selector learns feature scores
X_trn_sel = sel.transform(X_trn)     # best features are selected in both X datasets
X_tst_sel = sel.transform(X_tst)
```

- **Forward selection**: start with best single feature and add subsequent best features one by one. Used to select a few features.
- **Backward selection**: start with whole set and remove the worst features one by one. Used to select most of the features.

Feature extraction

Features are *transformed* into a lower-dimensional, more informative set with new, uncorrelated features that explain data in a more efficient manner. See dimensionality reduction in **Unsupervised Learning**.

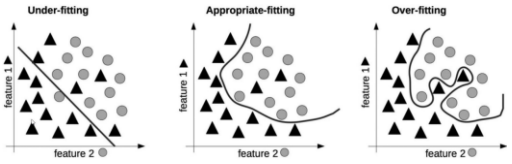
Feature expansion

Features are *combined* among them to obtain a new feature space with more dimensions that can be used to find non-linear relationships between input and target. See kernel methods in **Advanced ML Algorithms**.

Supervised Learning

Training data is labeled with known target values or classes, which are used to assess accuracy of predictions. An appropriate model minimizes regression/classification error while avoiding the following:

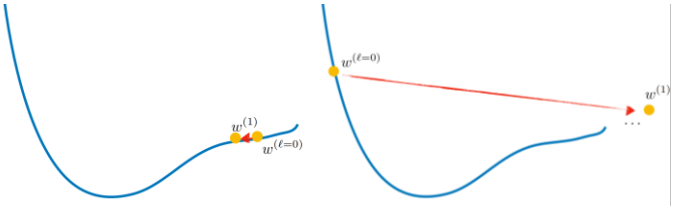
- **Underfitting**: model is not flexible so it doesn't adapt enough to nature of variables, fails to make good predictions.
- **Overfitting**: model adapts so much to training set that it becomes sensitive to small fluctuations and may not generalize correctly (it does not perform well with new data).



In parametric algorithms, some parameters (usually **weights** w and **bias** w_0) are introduced to define the relationship between input features and target value. A *cost* or *loss function* $\mathcal{L}(w, w_0)$ is defined to compare predictions with true targets. $\mathcal{L}$ is optimized w.r.t. model parameters, and those parameters are updated so that loss function is close to 0 (prediction coincides with target).

- **Gradient descent**: method to numerically minimize $\mathcal{L}$. Weights are randomly initialized and then updated in the opposite direction of the gradient of $\mathcal{L}$. Step size η needs to be appropriate: if too short, we don't move fast to convergence (solution); if too large, we may overshoot the solution.

$$w^{(t)} = w^{(t-1)} - \eta \nabla \mathcal{L}\left(w^{(t-1)}\right)$$



- **Stochastic gradient descent** (SGD): introduces mini-batches, used with large databases to improve efficiency (convergence is faster).
- **Adaptive learning rate**: step size can be adjusted according to gradient steepness ($\uparrow \nabla\mathcal{L} \rightarrow \downarrow \eta$), weights size, how fast the model is learning…

- ***K*-fold cross-validation**: training set divided in *k* partitions (folds), each of them is used once as validation set with all remaining folds used to train model. Results are averaged, improving model robustness. Used in hyperparameter tuning.

- **Hyperparameter tuning**: model is trained for a range of values of the parameter to study (grid search). The value with lowest error is chosen as the optimum and used to retrain the model on the original training set. Implemented using a validation set or with *k*-fold cross-validation.

```
# Implement grid search and k-fold cross-validation
from sklearn.model_selection import GridSearchCV
clf = KNeighborsClassifier()            # model to train
n_fold = 10                             # number of folds
range = [10, 20, 30, 40]                # range of values to test
grid = {'n_neighbors':range}            # hyperparameter to cross-validate
clf_cv = GridSearchCV(clf, grid, cv=F) # classifier with k-fold CV
clf_cv.fit(X_trn, Y_trn)
clf_cv.best_params_                     # best parameter value
clf_cv.best_estimator_                  # model trained with best parameter value
clf_cv.cv_results_                      # scores for all parameter values
f = clf_cv.predict(X_tst)
```

  - Hyperparameter tuning can be applied on a pipeline, cross-validation is done on hyperparameters at any step of the pipeline. Grids of parameters defined as dictionary: `{'step__parameter':range}`.

```
# Implement cross-validation on previous pipeline
rang_D = np.array([1,5,10,15,20]) # degree of polynomial
rang_F = np.arange(1,11)          # number of features to select
rang_A = np.logspace(-3,3,7)      # regularization coefficient alpha
grid = {'pol__degree':rang_D, 'sel__k':rang_F, 'reg__alpha':rang_A}
pipe_cv = GridSearchCV(pipe, grid)
```

REGRESSION

Data to predict are numerical values, which should be related to input data through some mathematical relationship that the model aims to approximate.

Parametric regression

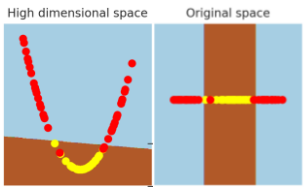
Target values are predicted by combining features according to some weights w and bias w_0.

- **Linear regression**: prediction f is a linear combination of all features weighted. Dataset needs to be extended with a column of 1s to accommodate intercept (bias). Parameters are fitted by minimizing loss function $\mathcal{L}$ (least squares or MSE) that defines our error in estimating the training set. After training, optimized weights $w^\star$ are used to predict new samples from test dataset.

$$f = w_0 + w_1x_1 + w_2x_2 + \cdots + w_Mx_M = w^\top x \qquad \mathcal{L} = \frac{1}{N}\sum\left(y_i - f(x_i)\right)^2 \qquad w^\star = (x^\top x)^{-1}x^\top y$$

```
# Implement linear regressor
from sklearn.linear_model import LinearRegression
reg = LinearRegression()       # linear regressor model
reg.fit(X_trn, Y_trn)          # fit regressor to training data
f = reg.predict(X_tst)         # predict target values
err = reg.score(X_tst, Y_tst) # predict and compute error
w = reg.coef_                  # get weights
w0 = reg.intercept_            # get bias
```

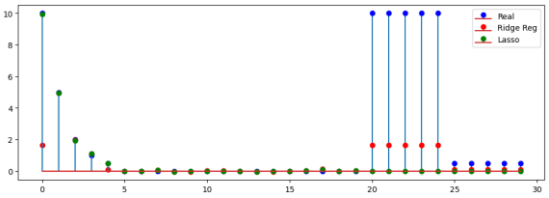
- **Polynomial expansion** (particular case of kernel method): features are linearly combined among themselves to create a higher-dimensional feature space which is then used in a regression model, allowing non-linear relationship between x and y. Data normalization is needed after expansion. Recommended to reduce number of features first to reduce computational costs. Risk of overfitting for high polynomial degree. Although final prediction is non-linear, the model is considered linear regression because weight vector is linear.



$$x = \{x_1, x_2\} \longrightarrow x_{exp} = \{1, x_1, x_2, x_1^2, x_2^2, x_1x_2\} \longrightarrow f = w_0 + w_1x_1 + w_2x_2 + w_3x_1^2 + w_4x_2^2 + w_5x_1x_2$$

```
# Expand features of dataset
from sklearn.preprocessing import PolynomialFeatures
degree = 2                                          # degree of polynomial
pol = PolynomialFeatures(degree, include_bias=False) # polynomial constructor object
X_trn_exp = pol.fit_transform(X_trn)                # polynomial version of training set
X_tst_exp = pol.transform(X_tst)                    # polynomial version of testing set
```

- **Ridge regularization**: to avoid overfitting, loss function incorporates a penalization term that prevents model from converging to large weight values. Hyperparameter α controls regularization strength, should be cross-validated. Ridge regularization introduces L_2 penalization, which maintains linearity and closed-form solution. Weights are proportional to feature importance, but relevance of redundant features is distributed among them. N is number of samples and M is number of features.



$$\mathcal{L} = \frac{1}{N}\left(\sum_{i=1}^{N}(y_i - w^\top x_i)^2 + \underbrace{\alpha \sum_{j=1}^{M} w_j^2}_{L_2}\right)$$

```
# Implement Ridge linear regressor
from sklearn.linear_model import Ridge
a = 2                 # regularization parameter
reg = Ridge(alpha=a) # Ridge regressor
```

- **LASSO regularization**: same as Ridge, but introduces L_1 penalization (no closed-form solution, global minimum convergence no longer guaranteed). Weights are proportional to feature importance, but only the first of many redundant features is given importance (others are ignored).

$$\mathcal{L} = \frac{1}{N}\left(\sum_{i=1}^{N}(y_i - w^\top x_i)^2 + \underbrace{\alpha \sum_{j=1}^{M} |w_j|}_{L_1}\right)$$

```
# Implement LASSO linear regressor
from sklearn.linear_model import Lasso
a = 2                 # regularization parameter
reg = Lasso(alpha=a) # LASSO regressor
```

- **Elastic net**: combines both L_2 and L_1 penalization terms from Ridge and LASSO, implemented through `sklearn.linear_model.ElasticNet`.

Non-parametric regression

The algorithm does not use parameters, although there may be hyperparameters.

- ***k*-nearest neighbors** (*k*-NN) **regressor**: new sample prediction is the average of the *k* nearest neighbors (*k* is hyperparameter). To find the *k* nearest neighbors, distances from new sample to all labeled samples are computed and the minimum distances are used. High-dimensional feature space and large *k* values lead to high computational costs (grows with $\mathcal{O}(N \cdot M)$).

```
# Implement k-nearest neighbors regressor
from sklearn.neighbors import KNeighborsRegressor
k = 5                          # number of neighbours
reg = KNeighborsRegressor(k) # k-NN regressor
```

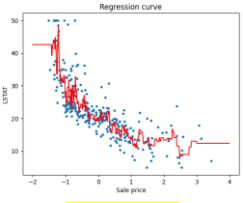
- **Gaussian processes**: see Gaussian processes in **Advanced ML Algorithms**.

Regression evaluation

Model predictions f are compared to real targets y of the test subset in order to evaluate the performance of the regression model.

- **Mean absolute error** (MAE) $\longrightarrow \frac{\Sigma|y_i - f_i|}{N}$
- **Mean square error** (MSE) $\longrightarrow \frac{\Sigma(y_i - f_i)^2}{N}$
- **Coefficient of determination** (R2 score or R2S) $\longrightarrow R_2 = 1 - \frac{\Sigma(y_i - f_i)^2}{\Sigma(y_i - \bar{y})^2}$

```
# Compute error manually
MAE = np.mean(np.abs(Y_tst - f))
MSE = np.mean((Y_tst - f)**2)

# Compute error with metrics library
from sklearn import metrics
MAE = metrics.mean_absolute_error(Y_tst, f)  # mean absolute error
MSE = metrics.mean_squared_error(Y_tst, f)   # mean squared error
R2S = metrics.r2_score(Y_tst, f)             # R2 score
R2S = reg.score(X_tst, Y_tst)                # predicts and computes R2 score
```



- **Regression curve**: prediction f as a function of input data x (f is obtained for an interval of values of x), can be plotted on top of training data to visually assess its performance. Usually done with one single feature.
- **Learning curve**: error or accuracy of model is plotted for training and testing datasets and for several values of a hyperparameter, allows to visualize its effect on the model (underfitting, overfitting…). If randomness is present, several iterations can be run for each value and averaged to provide a smooth curve. Add picture

CLASSIFICATION

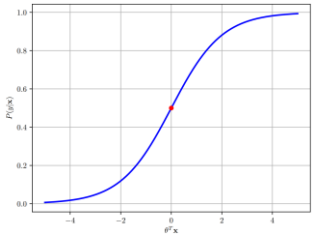
Target data to predict are the class (or classes) to which input sample belongs. Output can be given as hard output (predicted labels, usually integer values) or as soft output (probability of belonging to each class). In parametric classification, model is trained to find threshold (*decision boundary*) between classes, according to a set of discriminative features.

- **Dummy classifier**: ignores input features (e.g., always returns same class), used as baseline to compare.

```
# Implement dummy classifier
from sklearn.dummy import DummyClassifier
clf = DummyClassifier(strategy='most_frequent') # dummy classifier model
clf.fit(X_trn, Y_trn)     # train model
clf.predict(X_tst)        # predict labels
clf.score(X_tst, Y_tst)   # predict and compute accuracy
clf.predict_proba(X_tst) # soft output (posterior probability)
clf.coef_                 # get coefficients w (boundary vector)
clf.intercept_            # get bias w0
```

Binary classification

Target value y is a *binary* label (0 or 1). A loss function $\mathcal{L}$ compares predicted and real classes to determine model performance (success of prediction). Parameters w (weights) and w_0 (bias) are tuned during training according to optimization of $\mathcal{L}$.



- **Logistic regression** (LR): input features are linearly combined, so classification boundary separating both classes ($f = 0$) is a line. After trained, model uses *sigmoid function* to predict probability of a new test sample x_t of belonging to class 1.

$$f(x) = w_0 + x_1 w_1 + \cdots + x_M w_M = w^T x + w_0 \qquad p(y = 1|x_t) = \frac{1}{1 + e^{-(w^T x_t + w_0)}} = \sigma(w^T x_t + w_0)$$

$$f = 0 \longrightarrow p(y = 1|x) = p(y = 0|x) = 0.5 \qquad p(y = 0|x_t) = 1 - p(y = 1|x_t)$$

```
# Define logistic regression classifier
from sklearn.linear_model import LogisticRegression
C = 0.1 # (inverse) regularization parameter
I = 1e4 # number of iterations (default 100 may not be enough for convergence)
clf = LogisticRegression(C=C, max_iter=I)
```

- **Binary cross entropy**: loss function accumulates the probabilities of correct classes ($\mathbb{I}[\text{cond}]$ is 1 if $\text{cond} = \text{true}$, 0 otherwise). Higher model performance → better predictions → $\uparrow \sum \rightarrow \downarrow \mathcal{L}$. When minimized, it returns new weights that induce f to give more confident predictions. $\mathcal{L}$ is convex → ∃ global minimum, but solution is no closed-form (∄ analytical solution) because sigmoid is non-linear.

$$\mathcal{L} = -\sum \mathbb{I}[y_i == 1] \log p(y_i = 1|x_i) + \mathbb{I}[y_i == 0] \log p(y_i = 0|x_i)$$

- **Binomial deviance**: another loss function, not based on probabilities.

$$\mathcal{L} = \sum \log\left(1 + e^{w^\top x_i}\right) - y_i w^\top x_i$$

- Since the definition of f is similar to a linear regression, features can be expanded with polynomial expansion and regularization can be introduced with Ridge or LASSO penalization terms (solver type needs to be changed).

```
# Define LR classifier with regularization
clf = LogisticRegression(penalty='l1', solver='liblinear') # Ridge LR classifier
clf = LogisticRegression(penalty='l2', solver='lbfgs')     # LASSO LR classifier (default)
```

Multiclass classification

Target value y is a *multiclass* label (can take value from 1 to K). Input sample $x_{M\times 1}$ (one value for each feature) is linearly transformed into *logits* $z_{K\times 1}$ (one value for each class) according to some weight matrix $W_{K\times M}$ and bias vector $w_{0_{K\times 1}}$. After trained, model uses *softmax function* to predict probability of new sample x of belonging to each class k by transforming each component z_k into a soft output. Sum of all f_k for a given sample is 1.

$$z = Wx + w_0 \qquad f_k = p(y = k|\boldsymbol{x}) = \frac{e^{z_k}}{\sum_i^K e^{z_i}}$$

- **Cross entropy**: loss function defined for N training samples and K classes, accumulates probabilities of belonging to true class.

$$\mathcal{L} = -\sum^{N}\sum^{K} \mathbb{I}[y_i == k] \log p(y_i = k|x)$$

- **Decision tree** (DT): data is split into two categories by a threshold applied to a feature. These threshold and feature are chosen by trying all possible splits (all combinations of feature and threshold) and choosing the one with lowest *Gini index*, which indicates a pure split (potentially separating classes). The split creates a node ("tree branch") and the process is repeated until classes have been separated. Model hyperparameters used to control overfitting. Picture of DT results

```
# Define decision tree classifier
from sklearn.tree import DecisionTreeClassifier
D = 4  # maximum depth of the tree
S = 10 # minimum number of samples required to split a node
L = 5  # minimum number of samples required to be at a leaf
clf = DecisionTreeClassifier(max_depth=D, min_samples_split=S, min_samples_leaf=L)
```

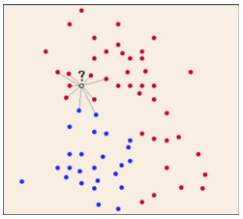
- **Random forest** (RF): particular case of bagging ensemble; several decision trees are trained with different subsets of samples and features, and results are averaged to improve accuracy. Because each sample will only be used in some trees, the model can be validated by classifying samples with trees where they did not participate, obtaining the *out-of-bag* error. RF can also be used to select relevant features.

```
# Define decision tree classifier
from sklearn.ensemble import RandomForestClassifier
N = 100                       # number of decision trees
D = 4                         # depth of trees
S = 0.5                       # percentage of samples used in a tree
clf = RandomForestClassifier(n_estimators=N, max_depth=D, max_samples=S).fit(X_trn, Y_trn)
imp = clf.feature_importances_ # relevance of each feature
```

- **Extremely randomized trees**: forces additional diversity by using random thresholds for each candidate feature and the best of these randomly-generated thresholds is picked as splitting rule.

Multi-output classification

Each sample may be classified into one or several classes. Code.



Non-parametric classification

The algorithm does not use parameters, although there may be hyperparameters.

- ***k*-nearest neighbors** (*k*-NN) **classifier**: new sample is assigned the most frequently occurring label in the *k* nearest neighbors (*k* is hyperparameter). Neighbors are found like in *k*-NN regressor.

Classification evaluation

There are metrics to evaluate performance of a classification algorithm.

- **Accuracy**: percentage of correctly classified samples over total number of samples. Default score using `.score` method.
- **Confusion matrix** (CM): rows correspond to real classes and columns to predicted ones. CM_{ij} is the number of samples belonging to class *i* that are predicted to be in class *j*. A perfect classification CM would have all zeros except for the diagonal, meaning that all predicted classes correspond to the real ones.

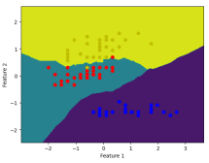
```
# Compute and show CM
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
cm = confusion_matrix(Y_tst, f)    # confusion matrix
ConfusionMatrixDisplay(cm).plot() # show CM
acc = np.trace(cm)/np.sum(cm)      # accuracy
```

- Metrics for binary classification: assuming classes are positive (P) or negative (N), a sample can be predicted as true P/N (correct) or false P/N (incorrect).
  - **Accuracy**: $acc = \frac{TP+TN}{TP+TN+FP+FN}$
  - False negative rate (FNR) or **missing ratio**: $FNR = \frac{FN}{FN+TP}$
  - False positive rate (FPR) or **false alarm ratio**: $FPR = \frac{FP}{FP+TN}$
  - True negative rate (TNR) or **specificity**: $TNR = \frac{TN}{TN+FP}$
  - True positive rate (TPR), **recall**, **sensitivity** or **detection ratio**: $TPR = \frac{TP}{TP+FN}$
  - Positive predictive value (PPV) or **precision**: $precision = \frac{TP}{TP+FP}$
  - **F-score**: $F\text{-}score = 2\frac{precision \cdot recall}{precision + recall}$

```
# Compute TN, TP, FN, FP from binary CM
```

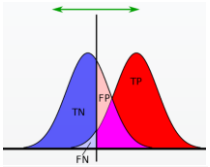
```
TN = cm[0,0] # 0 predicted as 0
TP = cm[1,1] # 1 predicted as 1
FN = cm[1,0] # 1 predicted as 0
FP = cm[0,1] # 0 predicted as 1

# Compute binary scores
from sklearn import metrics
acc = metrics.accuracy_score(Y_tst, f)
prec, recall, fscore, _ = metrics.precision_recall_fscore_support(Y_tst, f)
```

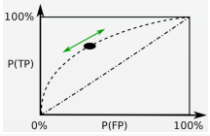
- **Classification boundary**: map of target labels assigned according to (usually) two features, plotted on top of training data to visually assess performance.



- **Receiver operating characteristic** (ROC) **curve**: shows performance of a binary classifier for different decision thresholds. A curve is plotted with FPR (1 – specificity) on x-axis and TPR (sensitivity) on y-axis, for several decision threshold values.

```
# Compute and plot ROC
from sklearn.metrics import roc_curve
o = clf.predict_proba(X_tst)       # soft output
fpr, tpr, _ = roc_curve(Y_tst, o) # false and true positive rates
plt.plot(fpr, tpr)                 # plot ROC
```





- **Area under the curve** (AUC): area under ROC curve, proportional to performance. Threshold that falls closer to upper left corner yields better results. Dummy classifier ROC will be diagonal and AUC will be 0.5.

```
# Compute AUC
from sklearn.metrics import auc
roc_auc = auc(fpr, tpr)
```

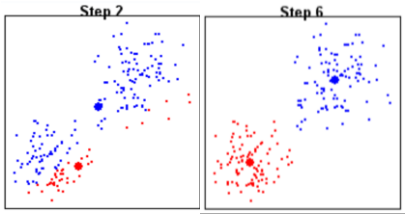
Unsupervised Learning

Labels of training data are unknown. Dataset is not separated into training, validation and test partitions.

Clustering

Model separates dataset in classes by finding natural *clusters* (groups) of samples. Distance between samples within same class should be minimized (high similarity) while maximizing distance to samples in other classes.

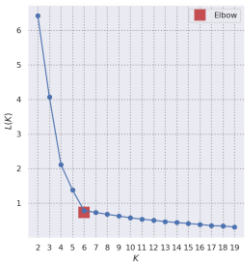
- ***k*-means clustering**: *k* random classes are created and iteratively adjusted to fit the real data distribution (*k* is specified by user).

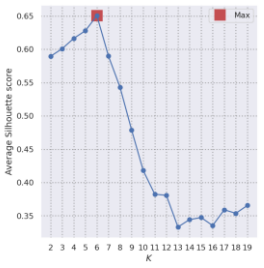


1. Randomly choose *k* objects from the training set (prototypes).
2. Create *k* classes by assigning all other objects to nearest prototype.
3. Compute new prototype as the center of mass of each class.
4. Repeat the process until some stopping criteria (number of iterations, convergence or threshold) is fulfilled.

```
# Implement a k-means clustering classifier
from sklearn.cluster import KMeans
K = 6                                # number of clusters
I = 10                               # number of initializations
clf = KMeans(n_clusters=K, n_init=I) # k-means classifier
f = clf.fit_predict(X)               # train model and predict cluster labels
c = clf.cluster_centers_             # get centroids
```

- Effect of `n_init`: solution depends on *initialization* and may converge to a local minimum instead of the global minimum that best represents the ideal solution, so it should be run several times with different initial prototypes to increase probability of finding best solution.
- Effect of `n_clusters`: if number of clusters is unknown, several values of *k* are tested and the loss function plotted. Ideal *k* will be at *elbow* of the curve (where it starts to decrease more softly).
- **Silhouette score**: measures similarity of each sample to its own cluster compared to the next closest cluster, used to determine ideal number of clusters (maximum silhouette score). Values from 0 to 1.






```
from sklearn.metrics import silhouette_score
```

```
SS = silhouette_score(X, f) # compute average silhouette score
```

- Spectral clustering:

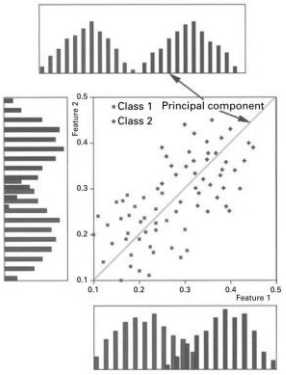
Dimensionality reduction

Features of the dataset are *transformed* into a lower-dimensional, more informative set with new, uncorrelated features that explain data in a more efficient manner. Removes irrelevant, noisy, correlated features, reducing amount of data to process and improving computation costs. Some information will be lost, but usually this will be noise.

- **Principal component analysis** (PCA): axes of original features X are rotated to obtain principal components aligned in the directions of *maximum variability*. PC are orthogonal, linear combinations of original axes that define new, uncorrelated features X', and they correspond to eigenvectors of covariance matrix of the feature vector X. These new features X' are ranked by the amount of class variability they explain, and only the first ones (highest eigenvalues λ) are retained. PCA does not consider any target (unsupervised), only distribution.

$$X' = U^\top X^\top \quad U = \text{eigs}(C_{XX})$$

  - Ex.: using original features (two dimensions) to sort a dataset into two classes may lead to overlapping of these classes, incorrectly classifying some samples (false positives). However, projecting data onto the principal component allows to optimally separate data into the two classes, reducing the required features to one (one dimension). In this case, the PC lies along the regression line.



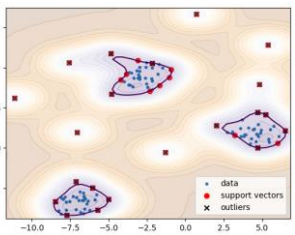
- **Autoencoder**: see autoencoder in **Deep Learning**.

Novelty detection

Novelty detection focuses on learning limits of data distribution. Used to clean *outliers* (observations that deviate significantly from rest of dataset), detect *model drift* (data statistics change enough to require model update).

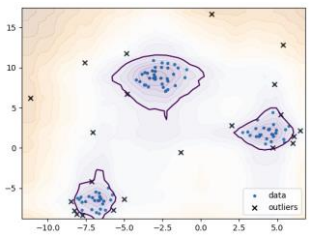
- **One-class SVM**: uses SVM methods but all samples are considered to belong to same class (positive), and a reference is used as negative class. Model is trained to learn boundary separating inliers (samples close to all others) and outliers (isolated samples). Kernel methods and slack variables are introduced, outliers are samples falling at wrong side of decision boundary.
  - **OCSVM against origin**: model finds the hyperplane with maximum margin that can separate observations from origin of coordinates in feature space. Formula depends on bias (distance to origin).
  - **OCSVM with hypersphere**: decision boundary defines hypersphere with minimum radius that encloses inliers, while outliers fall outside. Formula depends on sphere center and radius.

```
# Implement one-class SVM novelty detector
from sklearn.svm import OneClassSVM
nu = 0.1                        # percentage of support vectors (upperbound on
number of outliers)
det = OneClassSVM(nu=nu)        # OCSVM detector model
f = det.fit_predict(X)          # train detector and classify samples
inliers = np.where(f==1)[0]     # inliers are classified as 1
outliers = np.where(f==-1)[0] # outliers are classified as -1
```



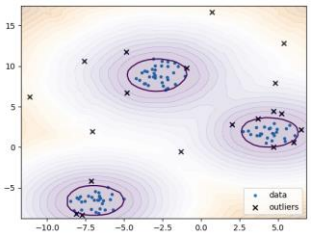
- **Isolation forest**: dataset is split as in a random forest (random threshold is defined between minimum and maximum value of a random feature). Results of several random trees are averaged. Because of their isolated nature, outliers will more likely be separated sooner from the rest of the dataset, while inlier observations will remain together for longer.

```
# Define novelty detector with isolation forest
from sklearn.ensemble import IsolationForest
det = IsolationForest(contamination=0.2)
```

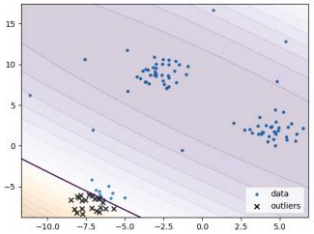


- **Local outlier factor**: density is estimated at each data point by measuring distance to nearby neighbors (as in *k*-NN), isolated observations (with less density) are classified as outliers. Suffers from *curse of dimensionality* (not reliable when number of features is high).

```
# Define novelty detector with local outlier factor
from sklearn.neighbors import LocalOutlierFactor
det = LocalOutlierFactor(novelty=True, contamination=0.2)
```

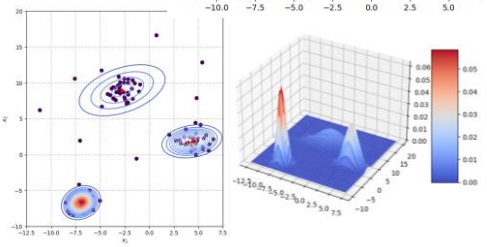


- **Minimum covariance determinant** or **elliptic envelope**: based on statistical properties of data, assumes data follows a multivariable Gaussian distribution in the feature space and so a hyperellipsoid can be defined around data. Observations falling out of this hyperellipsoid are considered outliers. Constrained to a single Gaussian distribution.




```
# Define novelty detector with elliptic envelope
from sklearn.covariance import EllipticEnvelope
det = EllipticEnvelope(contamination=0.2)
```

- **Gaussian mixture model**: used when data is distributed in more than one component (clusters), each following a different Gaussian distribution. Model learns parameters of each distribution iteratively by expectation-maximization algorithm. After trained, model can return probability of a given sample (lowest probabilities correspond to outliers).

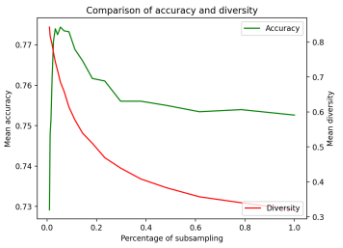


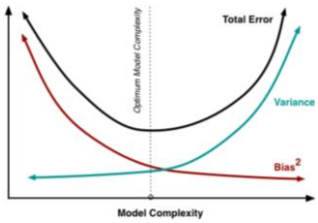
```
# Implement a Gaussian mixture model as novelty detector
from sklearn.mixture import GaussianMixture
K = 3                                   # number of components in the mixture
clf = GaussianMixture(n_components=K)   # Gaussian mixture model
clf.fit(X)                              # train model
p = clf.score_samples(X)                # compute log-likelihood (probability) of each sample
perc = 0.2                              # percentage of outliers
outliers = np.argsort(p)[:int(X.shape[0]*perc)] # obtain index of outliers
```

Advanced ML Algorithms

Ensembles

Several models (base learners) are combined to increase **diversity** and improve prediction of classification. Diversity is exploited in several ways. Training several ensembles and averaging their output allows to obtain smoother accuracy curves and to visualize variance.





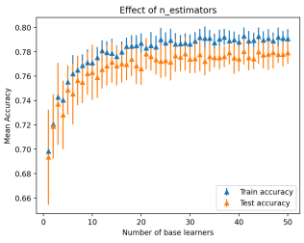
- **Bias**: expected difference between prediction of the designed model and real value. Simple models usually present high bias, tend to underfit.
- **Variance**: variability of prediction for several realizations of the model. Complex models usually present high variance, tend to overfit. There is a trade-off between bias and variance.
- **Noise**: random noise $\epsilon \sim N(0, \sigma_\epsilon^2)$ affecting generation of the model data → $Y = f(X) + \epsilon$

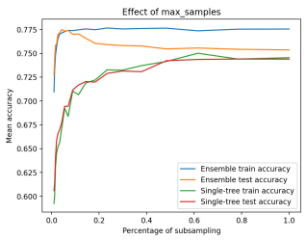
Bagging

Each base learner uses a different partition of training dataset, generating different predictions that can be averaged (regression) or aggregated (classification, usually majority vote). Both samples (data entries) and features can be subsampled. Subsampling is random and with replacement (*bootstrap sampling*). Samples and features may be repeated in same base learner and in more than one. E.g.: random forests.

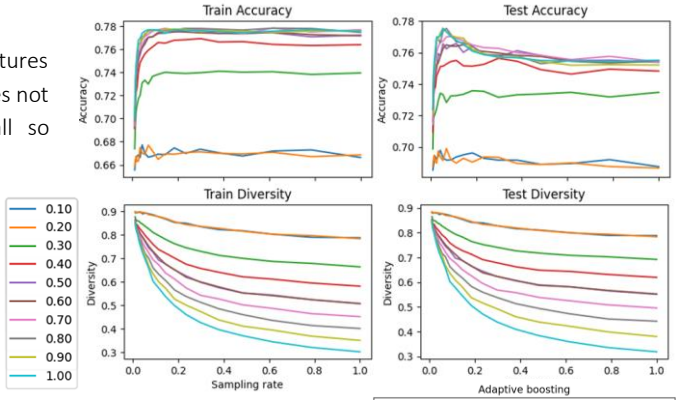
```
# Define a bagging ensemble
from sklearn.ensemble import BaggingClassifier
clf = DecisionTreeClassifier() # base learner
n_estimators = 20              # number of base learners
s_perc = 0.25                  # percentage of data entries to subsample
f_perc = 0.5                   # percentage of features to subsample
ens = BaggingClassifier(clf, n_estimators, max_samples=s_perc, max_features=f_perc) # bagging ensemble model
```

- Effect of `n_estimators`: adding base learners reduces *variance*. Accuracy of ensemble improves at first but stabilizes at some point (bias no longer decreases, no risk of overfitting). If `n_estimators` is further increased, the variance of mean accuracy is reduced, which improves performance.



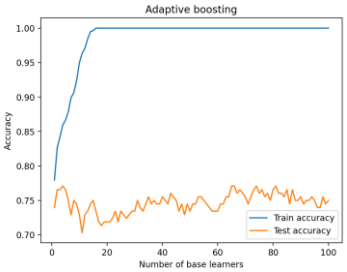


- Effect of `max_samples`: ensemble performs better than single tree. Using too small partitions (with few samples) does not produce accurate predictions, and using partitions of too many samples makes all base learners more similar, reducing diversity and test accuracy.
- Effect of `max_features`: smaller percentage of features improves diversity among base learner, but accuracy does not improve too much (maybe because dataset is small so subsampling leads to too few features).



Boosting

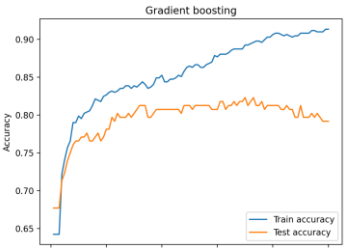
A sequence of T weak learners (slightly better than dummy model) is trained, each learner uses a modified version of the training dataset to try to correct error in previous one. Final output is a weighted combination of all, according to their performance.



- **Adaptive boosting** (AdaBoost): progressively focuses on samples wrongly classified by previous learner, thanks to an emphasis function $D_t(x)$ that changes for each learner and observation (initially all samples weight equally, $D_1(x_i) = \frac{1}{N}$). In *real AdaBoost*, base learner returns soft output $o_t(x)$ between –1 and 1 and ensemble output f_T is a weighted sum of all base learners' output. Output weights α_t are optimized according to exponential cost function to give higher influence to more accurate learners.

$$f_T(x_i) = \sum_t^T \alpha_t o_t(x_i) \qquad D_{t+1}(x_i) = \frac{D_t(x_i) e^{-\alpha_t o_t(x_i) y_i}}{Z_t} \qquad \alpha_t^\star = \min_\alpha \sum_i^N e^{-f_T(x_i) y_i} = \frac{1}{2} \ln\left(\frac{1 + \sum_{i=1}^N D_t o_t y}{1 - \sum_{i=1}^N D_t o_t y}\right)$$

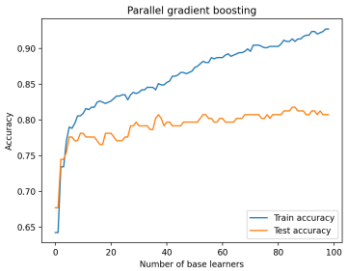
```
# Define real AdaBoost ensemble
from sklearn.ensemble import AdaBoostClassifier
clf = DecisionTreeClassifier() # base learner
T = 50                         # number of base learners
ens = AdaBoostClassifier(clf, n_estimators=T)
```



- **Gradient boosting**: real AdaBoost is reformulated as a gradient descent algorithm: adding at each step a new learner that minimizes the cost function is equivalent to moving in the direction of its gradient. Soft output o_t of each learner (decision trees by default) is seen as a new feature that improves this minimization. An additional optimization step is performed to find $o_t(t)$ that provides largest cost reduction up to step t, which is equivalent to maximizing emphasized version of accuracy of base learner t.

$$\min_{o_t(x)} \sum_i^N \exp\left(-\sum_{t'}^t \alpha_{t'} o_{t'}(x_i)\, y_i\right) = \max_{o_t(x)} \sum_i^N D_t(x_i) o_t(x_i) y_i$$

```
# Define gradient boosting ensemble
from sklearn.ensemble import GradientBoostingClassifier
T = 50               # number of decision trees
cost = 'exponential' # cost function
ens = GradientBoostingClassifier(n_estimators=T, loss=cost)
```



- **Parallel gradient boosting**: applies gradient boosting with parallel capabilities, uses early stopping (stops adding trees before overfitting occurs), useful for large databases. Implemented through XGBoost↗ library.

```
# Define parallel gradient boosting ensemble
from xgboost import XGBClassifier
T = 50 # number of decision trees
ens = XGBClassifier(n_estimators=T)
```

- Effect of `n_estimators`: adding base learners reduces *bias*. In adaptive boosting, training accuracy quickly reaches 100% but testing accuracy does not improve (overfitting). In gradient boosting, increasing number of base learners leads to a decrease of test accuracy. In both cases, model focuses too much on misclassified samples (which may be outliers), resulting in overfitting. Parallel gradient boosting reduces slightly this effect.
- Effect of `max_depth`: increasing depth of base learner trees helps reduce bias but increases variance.

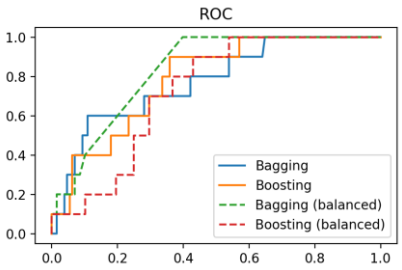
Stacking

Learners based on different algorithms (*k*-NN, logistic regression, DTs…) are combined to predict output. Because each model works differently, there will be diversity in their predictions and the final error variance will be reduced. Combination of outputs is done through a final estimator, usually logistic regression.

```
# Define stacking ensemble
from sklearn.ensemble import StackingClassifier
clf = [('KNN5', KNeighborsClassifier(5)), ('KNN10', KNeighborsClassifier(10))] # base estimators
clf_final = LogisticRegression()                                                # final estimator
ens = StackingClassifier(clf, final_estimator=clf_final)
```

Imbalanced problems

Class imbalance occurs when most of training samples belong to one class, while the other is not represented enough. A 90/10 imbalanced problem solved with a dummy classifier will achieve accuracy of 90%. Expected accuracy in imbalanced problems should always be higher than dummy, or use another metric (e.g., AUC). Two ways to correct imbalance:



- **Data resampling**: randomly either include repeated samples of minority class (oversampling) or undersample majority class (undersampling) so that same number of both is used. Implemented with bagging ensemble. Disadvantage: we lose samples in undersampling.

```
# Solve imbalance by data resampling
Nsub = n_perc*X_trn.shape[0]//2                          # number of samples to include in each class
samp_0 = np.random.choice(X_trn[Y_trn==0].shape[0], Nsub) # index to resample class 0
samp_1 = np.random.choice(X_trn[Y_trn==1].shape[0], Nsub) # index to resample class 1
samp = np.concatenate((samp_0, samp_1))                  # resampling index
ens = BaggingClassifier(clf, n_estimators=T)
ens.fit(X_trn[samp,:], Y_trn[samp])                      # train with resampled dataset
```

- **Cost function weighting**: classes are assigned different weights to balance the effect they have on the cost function. A class with N_i samples in the training dataset is assigned a weight of $\frac{1}{N_i}$. Implemented with boosting ensemble. Advantage: all samples are used.

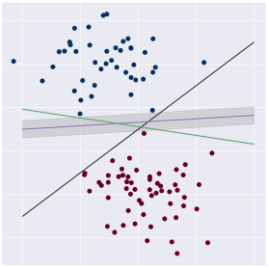
```
# Solve imbalance by cost function weighting
weight_0 = 1/np.sum(Y_trn==0)                    # initial weight for class 0
weight_1 = 1/np.sum(Y_trn==1)                    # initial weight for class 1
weights = np.where(Y_trn==0, weight_0, weight_1) # cost function weighting
ens = AdaBoostClassifier(clf, n_estimators=T)
ens.fit(X_trn, Y_trn, sample_weight=weights)     # assign initial weights
```

SUPPORT VECTOR MACHINES

In support vector machine (SVM) algorithms, classification boundary is defined by some samples (support vectors) that lie closest to the boundary. All other samples do not contribute to define the boundary → solution will be *sparse* (most samples with weight 0). Often combined with kernel methods.

SVM on separable classes

If classes are well separated, there are many linear solutions to the classification problem. SVM models try to find the best decision boundary to separate classes by applying a margin around it and finding the boundary that maximizes this margin. Wider margin means better generalization of the model.



- **Classification boundary**: $f(x) = w^{\mathrm{T}}x + w_0$ verifying that $f > 0$ for samples in class 1, $f < 0$ for samples in class –1 (in binary classification) and $f = 0$ is the *hyperplane* dividing both classes.
- **Functional margin**: distance from each sample to hyperplane, $y_i f(x_i)$, should be positive for all samples if predictions are correct → $y_i f(x_i) \geq 0$ (constrain). Negative functional margin would mean that sample is at wrong side of hyperplane (has been misclassified).
- **Geometrical margin**: functional margin of closest sample x^*, scaled by the norm of the weight vector $\frac{yf(x^*)}{\|w\|}$. This makes the margin invariant to hyperplane scale and provides a reference for the margin width since now the distance between classification boundary and closest sample is $\frac{y^*f(x^*)}{\|w\|} = \frac{1}{\|w\|}$. Constrain for samples being out of margin is $y_i f(x_i) \geq 1$.

- **Primal formulation**: since margin width is inversely proportional to the norm of the weight vector, maximizing margin is the same as minimizing $\|w\|$. Cost function is solved for w, w_0, complexity depends on number of dimensions, $\mathcal{O}(D^3)$. Convergence to unique solution is assured and efficient.

$$\begin{cases} \text{minimize} & \mathcal{L}(w, w_0) = \frac{1}{2}\|w\|^2 \\ \text{s.t.} & y_i f(x_i) \geq 1 \end{cases}$$

- **Dual formulation**: cost function expressed in terms of Lagrange multipliers a_i to introduce constrain. Solving for w, w_0, we get expressions that can be introduced in $\mathcal{L}$ to make it only depend on a_i. Complexity depends on number of samples, $\mathcal{O}(N^3)$. Selection of primal or dual will depend on size of database.

$$\begin{cases} \text{minimize} & \mathcal{L} = \frac{1}{2}\|w\|^2 - \sum a_i(y_i f(x_i) - 1) \\ \text{where} & a_i \geq 0 \end{cases} \xrightarrow[w,w_0]{\text{minimize}} \begin{cases} w = \sum a_i y_i x_i \\ 0 = \sum a_i y_i \end{cases}$$

$$\max_a \left(\sum a_i - \frac{1}{2} \sum_i \sum_j a_i a_j y_i y_j x_i^\top x_j \right) \text{ s.t.} \begin{cases} a_i \geq 0 \\ \sum a_i y_i = 0 \end{cases}$$

- **Sparsity**: for all samples, either their Lagrange multiplier a_i is 0 (and they do not contribute to solution) or they lie exactly on the margin, $y_i f(x_i) = 1$ (they are *support vectors* and contribute to solution). Weights of the model are defined only by these SVs.

$$a_i(y_i f(x_i) - 1) = 0 \longrightarrow \begin{cases} a_i = 0 & \longrightarrow \text{does not contribute} \\ y_i f(x_i) = 1 & \longrightarrow \text{support vector} \end{cases}$$

$$w = \sum_{i:a_i>0}^{N} a_i y_i x_i \qquad\qquad w_0 = \frac{1}{N_{SVs}} \sum_{i:a_i>0}^{N} (y_i - w x_i)$$

```
# Define SVM classifier
from sklearn.svm import LinearSVC            # SVM using primal formulation
from sklearn.svm import SVC                  # SVM using dual formulation
clf = SVC(kernel='linear').fit(X_trn, Y_trn) # SVM classifier
clf.n_support_                               # number of SVs for each class
clf.support_                                 # index of SVs
clf.suport_vectors_                          # support vectors
clf.dual_coef_                               # Lagrange multipliers multiplied by label (a*y)
```

SVM on non-separable classes

If classes overlap or if we want to avoid overfitting caused by outliers, previous formulation will not work. Some samples should be allowed to be misclassified, achieved by introducing *slack variable* ξ_i in the constraint, defined for each sample. A regularization parameter C appears in loss function to balance margin maximization and error minimization (larger C forces ξ_i to be 0, preventing samples to be misclassified).

$$y_i f(x_i) \geq 1 - \xi_i \longrightarrow \begin{cases} \xi_i = 0 & \longrightarrow & y_i f(x_i) \geq 1 & x_i \text{ correctly classified, outside or on margin} \\ 0 < \xi_i \leq 1 & \longrightarrow & 0 < y_i f(x_i) < 1 & x_i \text{ correctly classified, inside margin} \\ \xi_i > 1 & \longrightarrow & y_i f(x_i) < 0 & x_i \text{ misclassified} \end{cases}$$

$$\mathcal{L} = \frac{1}{2}\|w\|^2 + C \sum \xi_i - \sum a_i(y_i f(x_i) - 1 + \xi_i) - \sum b_i \xi_i$$

```
# Define SVM classifier (non-separable classes)
from sklearn.svm import SVC
C = 1e3                          # regularization parameter
clf = SVC(kernel='linear', C=C) # SVM classifier
```

- **Dual formulation**: similar to SVM on separable classes, optimization problem becomes:

$$\max_a \left(\sum a_i - \frac{1}{2} \sum_i \sum_j a_i a_j y_i y_j x_i^\top x_j \right) \text{ s.t.} \begin{cases} 0 \leq a_i \leq C \\ \sum a_i y_i = 0 \end{cases}$$

- **Hinge loss**: loss function, defines upper bound of classification error $\longrightarrow w^\star = \min_w \sum [1 - y_i f(x_i)]_+ + \frac{\|w\|^2}{2C}$

SVM on multiclass classification

Multiclass classification problem (several target classes) is converted into a set of binary problems. For a multiclass problem of K classes:

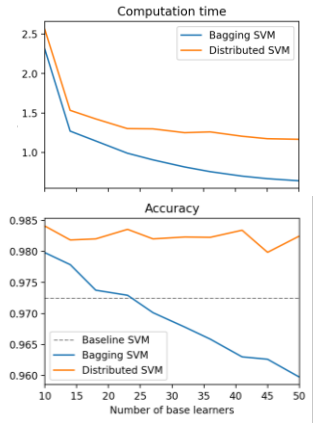
- **One-vs.-one strategy**: several classifiers ($K(K-1)/2$) are trained on pairs of classes, so that every class is compared with all others independently. Not very efficient for large K, and each binary classifier uses a small subset of training data (only uses samples corresponding to two classes).

```
# Define multiclass SVM (one-vs.-one)
from sklearn.multiclass import OneVsOneClassifier
clf = OneVsOneClassifier(SVC())
```

- **One-vs.-rest strategy**: each class is fitted against all remaining classes together. Only requires K classifiers and all training data is used in all of them. Used by default.

```
# Define multiclass SVM (one-vs.-rest)
from sklearn.multiclass import OneVsRestClassifier
clf = OneVsRestClassifier(SVC())
```

SVM on large databases



Training dataset is divided into subsets, each one used to train a separate SVM (bagging ensemble). Reduces computation costs.

- **Bagging SVM**: predictions of all SVM in the ensemble are combined by majority voting. Adding more base learners to ensemble (each of them with less number of samples) → less computation time but reduced accuracy.
- **Distributed SVM**: each SVM provides support vectors that are then combined in a new training dataset, only made of support vectors, used to train a new global model. Due to sparsity properties of SVM, support vectors are the only samples needed to find classification boundary between classes. Adding more base learners to ensemble → less computation time (although longer than bagging SVM due to training global model) and accuracy remains constant.

SVM on regression

Support vector regression (SVR) applies the SVM approach to a regression problem, instead of classification. Finish

KERNEL METHODS

In kernel methods, a non-linear mapping function $\phi(x)$ transforms input features into a high-dimensional feature space (feature expansion), where an inner product is applied to the dataset. Model is solved linearly in high-dimensional space but solution is non-linear in original space, allowing for complex relationships between input and target.

Kernel function

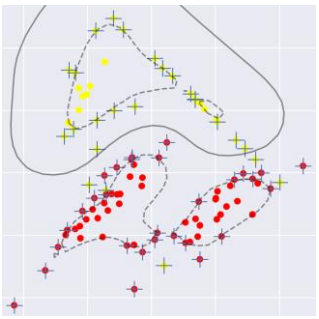
Kernel function is the operation of mapping function + scalar product, applied to each pair of samples.

$$k(x_1, x_2) = \phi(x_1)^\top \phi(x_2)$$

Kernel matrix has size $N_{x_1} \times N_{x_2}$, complexity grows with number of samples N, not features. Whenever an algorithm is formulated such that x only appears as scalar products, the *kernel trick* can be applied (replace scalar product by kernel function), converting algorithm into non-linear problem and reducing overfitting. If formulation is expressed in terms of kernel, we don't need to know ϕ (but required to recover weights: $w = \sum a_i y_i \phi(x_i)$). Often used in SVM where only support vectors need to be involved (low computational cost, solution maintains sparsity). x_i is training sample and x_t is test sample.

$$f(x_t) = \sum_{i \in SVs} a_i y_i x_i^\top x_t + w_0 \xrightarrow{\text{kernel trick}} f(x_t) = \sum a_i y_i k(x_i^\top, x_t) + w_0 = \sum a_i y_i \phi(x_i)^\top \phi(x_t) + w_0$$

Examples of kernel functions:



- **Linear kernel** → $k(x_1, x_2) = x_1^\top x_2$

```
from sklearn.metrics.pairwise import linear_kernel
K = linear_kernel(X_1, X_2) # compute linear kernel
```

- **Polynomial kernel** → $k(x_1, x_2) = (x_1^\top x_2 + c)^d$, where d is degree of polynomial (see polynomial expansion in **Supervised Learning**).
- **Gaussian kernel** or ***radial basis function*** (RBF) → $k(x_1, x_2) = e^{-\frac{\|x_1 - x_2\|^2}{2\sigma^2}} = e^{-\gamma \|x_1 - x_2\|^2}$, where γ defines influence of each training sample on decision boundary: ↑ γ → ↓ σ (Gaussian shape is narrow) → more influence (more samples are SVs) → model more complex, risk of overfitting. γ values often divided by number of features to reduce effect of size of feature space (since RBF is a distance-based kernel).

```
from sklearn.metrics.pairwise import rbf_kernel
K = rbf_kernel(X_1, X_2) # compute RBF kernel
```

- **Dual kernel** → $k = \lambda k_1 + (1-\lambda)k_2$, feature space is divided in subsets, each subset is processed with different kernels which are then combined into a global kernel. A parameter λ gives more weight to one or another subset.

```
lambda = 0.5                      # weight parameter
K_1 = linear_kernel(X_1, X_1)     # linear kernel of subset 1
K_2 = rbf_kernel(X_2, X_2)        # RBF kernel of subset 2
K = lambda*K_1 + (1 - lambda)*K_2 # dual kernel
```

- See more kernels in **Gaussian processes**.

Representer theorem

Solution to any algorithm involving minimization of a regularized empirical cost function can be expressed as a weighted combination of training data x_i by introducing dual variables, and the prediction to new samples x_t can then be defined with a scalar product that can be replaced with kernel functions.

$$\underset{\text{Primal formulation}}{\begin{cases} f(x) = w^\top x \\ w^\star = \min\limits_w \left(\sum \mathcal{L}(f(x_i)y_i) + \lambda \|w\|^2 \right) \end{cases}} \longrightarrow \underset{\text{Dual formulation}}{\begin{cases} w^\star = \sum a_i x_i \\ f(x_t) = \sum a_i x_i^\top x_t \longrightarrow f(x_t) = \sum a_i k(x_i, x_t) \end{cases}}$$

- **Primal formulation**: solving for w, w_0, complexity depends on number of dimensions, $\mathcal{O}(D^3)$.
- **Dual formulation**: solving for a_i, complexity depends on number of samples, $\mathcal{O}(N^3)$.
- **Kernel Ridge regression**: representer theorem applied on linear Ridge regression (dual formulation + kernel trick). θ are weights in the high-dimensional feature space, $K = k(x, x)$ is kernel between all training samples. Solution is not sparse.

$$\mathcal{L} = \frac{1}{N}\left(\sum_{i=1}^{N} (y_i - w^\top x_i)^2 + \alpha \sum_{j=1}^{M} w_j^2 \right) \xrightarrow{\text{mapping function}} \mathcal{L} = \frac{1}{N}\left(\sum_{i=1}^{N} \left(y_i - \theta^\top \phi(x_i)\right)^2 + \alpha \sum_{j=1}^{M} \theta_j^2 \right)$$

$$f(x_t) = w^\top x_t \xrightarrow{\text{dual formulation}} f(x_t) = \sum a_i x_i^\top x_t \xrightarrow{\text{kernel trick}} f(x_t) = \sum a_i k(x_i, x_t) \qquad a = (K - \lambda I)^{-1} y$$

```
# Define kernel Ridge regressor
from sklearn.kernel_ridge import KernelRidge
a = 0.01                                    # regularization strength
k = 'rbf'                                   # type of kernel
g = 0.1                                     # gamma parameter in kernel
reg = KernelRidge(alpha=a, kernel=k, gamma=g) # kernel Ridge regressor
```

- **Kernel logistic regression**: representer theorem applied on logistic regression with regularized binomial deviance as loss function. $k(x, x_i)$ is a column vector between all training data x and training sample x_i, $K = k(x, x)$ is kernel between all training samples.

$$\mathcal{L} = \sum \left(\log\left(1 + e^{w^\top x_i}\right) - y_i w^\top x_i \right) + \lambda \|w\|_2^2 \xrightarrow[\text{+ kernel trick}]{\text{mapping function}} \mathcal{L} = \sum \log\left(1 + e^{a^\top k(x, x_i)}\right) - y_i a^\top k(x, x_i) + \lambda a^\top K a$$

- **Kernel SVM**: representer theorem applied on SVM, see dual formulation in **SVM**.
- **Kernel Bayesian regression**: kernel trick applied on predictive function f of Bayesian regression. The following kernels are defined as $k(x_1, x_2) = \phi(x_1)^\top \Sigma_p \phi(x_2)$, between training ($x$) and test ($x_t$) datasets: $K = k(x, x)$, $K_t = k(x, x_t)$ and $K_{tt} = k(x_t, x_t)$.

$$y = w^\top \phi(x) + \epsilon \qquad \mu_w = K_t^\top (K + \sigma_n^2 I)^{-1} y \qquad \Sigma_w = K_{tt} - K_t^\top (K + \sigma_n^2 I)^{-1} K_t \qquad f(x_t) \sim \mathcal{N}(\mu_w, \Sigma_w)$$

- **Gaussian processes**: kernel used to define covariance in Gaussian processes distributions.

PROBABILISTIC MODELS

In probabilistic models, the output is provided as a *probabilistic* distribution, rather than a *deterministic* value. This allows to also obtain the uncertainty around the prediction.

Bayesian regression

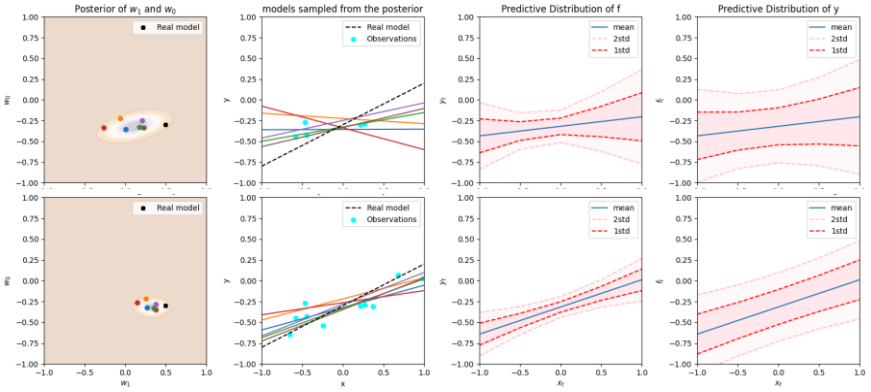
Target data is considered to be generated from x by a weighted linear model plus some noise ϵ following Gaussian distribution with zero mean and variance σ_n^2. Parameters of the model (w) are unknown but can be estimated from Bayes' theorem, which provides a probabilistic estimation in the form of distribution. Once estimated, w distribution characterizes the behavior of data in a probabilistic

manner and, for a test observation x_t, the model returns a *predictive distribution* f of the target (its most likely value and also how *confident* we are). Increasing the number of training samples improves this prediction and its certainty.

$$y = w^\top x + \epsilon \qquad \epsilon \sim \mathcal{N}(0, \sigma_n^2) \qquad p(w|x,y) = \frac{p(y|x,w) \cdot p(w)}{p(y|x)} \qquad f(x_t) = w^\top x_t \sim \mathcal{N}(\mu_w^\top x_t, x_t^\top \Sigma_w x_t)$$

- **Posterior probability** $\longrightarrow p(w|x,y) \sim \mathcal{N}(\mu_w, \Sigma_w)$, estimated distribution of parameter values, combines information of observations (x, y) with prior information on w.
- **Likelihood** $\longrightarrow p(y|x,w) \sim \mathcal{N}(xw, \sigma_n^2 I)$, probability of obtaining output y, knowing the input x and underlying parameters of the model. Represents information contained in the observations.
- **Prior probability** $\longrightarrow p(w) \sim \mathcal{N}(0, \Sigma_p)$, based on any previous knowledge about the parameter's behavior.
- **Evidence** $\longrightarrow p(y|x)$ probability for output in general. Same for all samples, it does not affect relative probabilities (scaling factor).

When no prior knowledge is available, maximum likelihood estimators (MLE) of w and σ_n^2 are obtained by solving $\max_{w,\sigma_n^2} p(y|x,w,\sigma_n^2)$ ($\hat{w}_{MLE}$ becomes solution of linear regression). When prior knowledge available, posterior distribution is Gaussian defined by a mean vector μ_w (one mean per feature) and covariance matrix Σ_w (μ_w is similar to solution of Ridge regression, Σ_p is equivalent to regularization coefficient α).



No prior	With prior
$\hat{w}_{MLE} = (x^\top x)^{-1} x^\top y$	$\mu_w = \sigma_n^{-2}\left(\sigma_n^{-2} x^\top x + \Sigma_p^{-1}\right)^{-1} x^\top y$
$\hat{\sigma}^2_{n\,MLE} = \frac{1}{N}\sum (y_i - \hat{w}_{MLE}^\top x_i)^2$	$\Sigma_w = \left(\sigma_n^{-2} x^\top x + \Sigma_p^{-1}\right)^{-1}$

In absence of training data or when training data is too noisy, we can still make predictions based solely on prior distribution. σ_n^2 will balance the relevance of prior and likelihood:

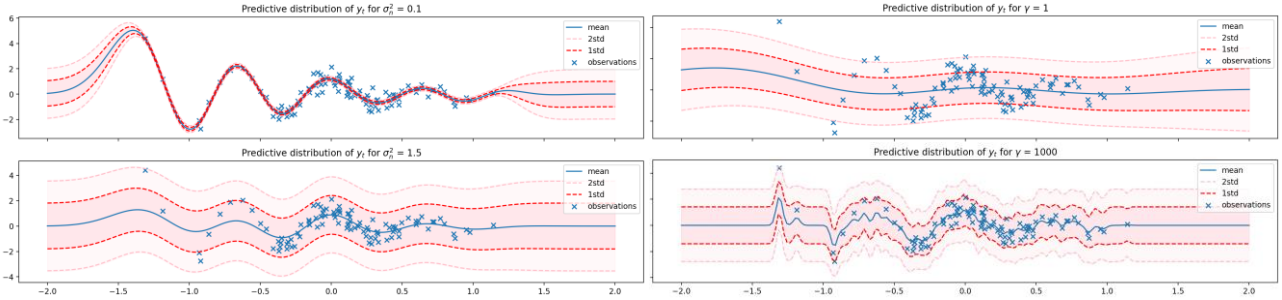
- $\sigma_n^2 \to \infty \longrightarrow$ observations are noisy (not trustworthy), posterior converges to prior $\longrightarrow \begin{cases} \Sigma_w = \left(\sigma_n^{-2} x^\top x + \Sigma_p^{-1}\right)^{-1} \longrightarrow \Sigma_p \\ \mu_w = \sigma_n^{-2} \Sigma_w x^\top y \longrightarrow \sigma_n^{-2} \Sigma_p x^\top y \longrightarrow 0 \end{cases}$
- $\sigma_n^2 \to 0 \longrightarrow$ data not noisy (trustworthy), more weight given to likelihood $\longrightarrow \begin{cases} \Sigma_w = \left(\sigma_n^{-2} x^\top x + \Sigma_p^{-1}\right)^{-1} \longrightarrow \sigma_n^2 (x^\top x)^{-1} \text{ narrow} \\ \mu_w = \sigma_n^{-2} \Sigma_w x^\top y \longrightarrow (x^\top x)^{-1} x^\top y = \hat{w}_{MLE} \end{cases}$

Once trained, model can be used to find predictive distribution of new test observations (*inference*). Each test sample has different variance: test samples close to many training points will have low predictive uncertainty, while test samples in regions with few (or none) training samples will have large predictive variance. Output can be of two types:

- **Noiseless output**: used when making predictions over test data $\longrightarrow f_t = w^\top x_t \sim \mathcal{N}(\mu_w^\top x_t, x_t^\top \Sigma_w x_t)$
- **Noisy output**: used to compare predictions over training data $\longrightarrow y_t = f_t + \epsilon \sim \mathcal{N}(\mu_w^\top x_t, x_t^\top \Sigma_w x_t + \sigma_n^2)$

If hyperparameters θ (noise variance, kernel parameters…) of real model are unknown, different values can be tested and the confidence interval (CI) of results is compared with true observations (95% samples should lie within 2σ).

- Noise variance σ_n^2: using a value higher (lower) than true one, CI will be wide (narrow) and most samples will fall inside (outside).
- Kernel parameter γ: using higher (lower) value allows to obtain a more non-linear (smoother) output.



If we have prior knowledge about θ (*hyperprior*), their distribution can be found by maximizing the marginal likelihood (evidence) $p(y|x,\theta)$ (negative log-marginal likelihood used as loss function; K is kernel between training samples).

$$\mathcal{L} = -\log p(y|x,\theta) = -\frac{1}{2}y^\top (K+\sigma_n^2)^{-1}y - \frac{1}{2}\log \overbrace{|K+\sigma_n^2 I|}^{\text{determinant}} - \frac{N}{2}\log 2\pi$$

A deterministic value of θ can be obtained from its distribution by using an *estimator*. When distributions are Gaussian, all these estimators give same value:

- **Maximum likelihood estimator** (MLE) $\longrightarrow \hat{\theta}_{MLE} = \underset{\theta}{\operatorname{argmax}}\, p(x|\theta)$, used when no prior information on θ is given.
- **Maximum a posteriori** (MAP) $\longrightarrow \hat{\theta}_{MAP} = \underset{\theta}{\operatorname{argmax}}\, p(\theta|x)$
- **Minimum mean square error** (MMSE) $\longrightarrow \hat{\theta}_{MSE} = E\{\theta|x\}$
- **Median absolute deviation** (MAD) $\longrightarrow \hat{\theta}_{MAD} = \text{median}\{p(\theta|x)\}$

Gaussian processes

Gaussian processes (GP) propose a probabilistic model of $f(x)$ directly from x, with no weights involved (non-parametric regression). Model output is a multivariate Gaussian distribution (purely statistical based on available training samples), defined by *mean* and *covariance functions* (one distribution for each sample x_i). Test data is predicted according to these statistics. Output of training and test data (f and f_t) are jointly distributed. If model accounts for noise in training samples, then y is used instead of f.

$$f(x_i) \sim \mathcal{GP}\left(\mu(x_i), \kappa(x_i,x_j)\right)$$
$$y = f(x) + \epsilon \qquad \epsilon \sim \mathcal{N}(0,\sigma_n^2)$$

$$\text{No noise:} \quad p\left(\begin{bmatrix} f \\ f_t \end{bmatrix}\right) \sim \mathcal{N}\left(\begin{bmatrix} \mu(x) \\ \mu(x_t) \end{bmatrix}, \begin{bmatrix} K & K_t \\ K_t & K_{tt} \end{bmatrix}\right)$$
$$\text{With noise:} \quad p\left(\begin{bmatrix} y \\ f_t \end{bmatrix}\right) \sim \mathcal{N}\left(\begin{bmatrix} \mu(x) \\ \mu(x_t) \end{bmatrix}, \begin{bmatrix} K+\sigma_n^2 I & K_t \\ K_t & K_{tt} \end{bmatrix}\right)$$

Mean function $\mu(x)$ is usually set to 0 and covariance function $\kappa(x_i,x_j)$ is a kernel, defines influence of each sample on all others. Kernel can be computed between training (x) and test (x_t) datasets: $K = \kappa(x,x)$, $K_t = \kappa(x,x_t)$ and $K_{tt} = \kappa(x_t,x_t)$.

- **Prior distribution**: initial knowledge of $\mu(x)$ and $\kappa(x_i,x_j)$, can be used to generate samples.
- **Predictive distribution**: distribution of f_t given f (without noise) or y (with noise), prediction of test data.

$$\text{No noise:} \quad p(f_t|x_t,x,f) \sim \mathcal{N}\big(\mu(x_t) + K_t^\top K^{-1}\big(f-\mu(x)\big), K_{tt} - K_t^\top K^{-1} K_t\big)$$
$$\text{With noise:} \quad p(f_t|x_t,x,y) \sim \mathcal{N}\big(\mu(x_t) + K_t^\top (K+\sigma_n^2 I)^{-1}\big(y-\mu(x)\big), K_{tt} - K_t^\top (K+\sigma_n^2 I)^{-1} K_t\big)$$

When noise is considered, model can also return noisy predictive distribution: $p(y_t|x_t,x,y) \sim \mathcal{N}(\dots, \dots + \sigma_n^2)$. Similar to Bayesian regression, hyperparameters θ (σ_n^2 and γ) can be estimated by testing different values and comparing the confidence interval (CI) with true observations (95% samples should lie within 2σ), or by maximizing evidence $p(y|x,\theta)$. A deterministic value of θ can be obtained from its distribution by using an estimator.

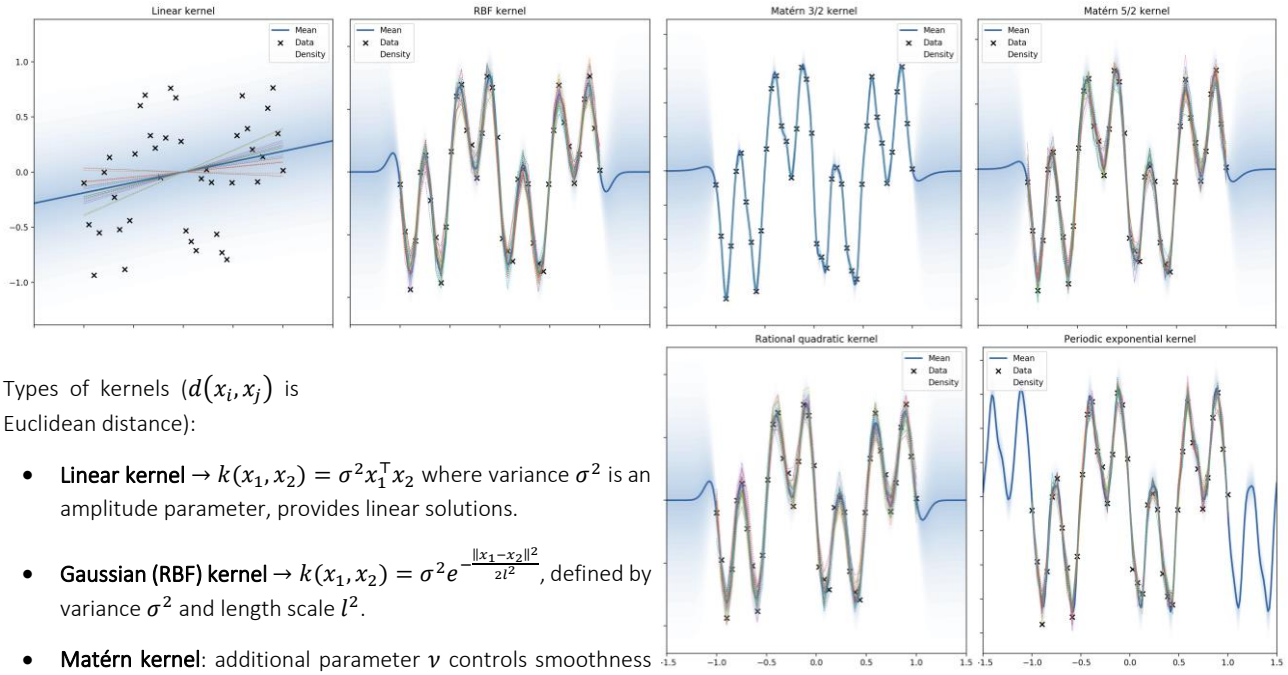
A GP regressor is modeled with its covariance function (kernel or combination of kernels, defined by a default parameter value and a bounded range to explore). In GP, *length scale* parameter l is used instead of γ ($\gamma = \frac{1}{2l^2}$), equivalent to Gaussian variance: $\uparrow l \rightarrow$ wider Gaussian shape $\rightarrow$ smoother output. Output of the model are the statistics of the predictive distribution of y_t.

```
# Implement Gaussian process regressor
from sklearn.gaussian_process import GaussianProcessRegressor as GP
from sklearn.gaussian_process.kernels import RBF, WhiteKernel, ConstantKernel
K_const = ConstantKernel(1, (1e-3,1e3))             # constant kernel
K_rbf   = RBF(10, (1e-2,1e2))                       # Gaussian kernel (defined by length scale)
K_noise = WhiteKernel(1, (1e-10,1e1))               # noise kernel (defined by variance)
K       = K_const*K_rbf + K_noise                   # global kernel (prior covariance function)
reg = GP(kernel=K, n_restarts_optimizer=10)         # GP regressor
reg.fit(X_trn, Y_trn)
y_mean, y_cov = reg.predict(X_tst, return_cov=True) # predict mean vector and covariance matrix
y_mean, y_std = reg.predict(X_tst, return_std=True) # predict mean vector and standard deviation
```

GPy↗ is a library specific for GP. Kernels already include constant and white noise.

```
# Implement Gaussian process regressor with GPy
!pip install GPy
from GPy import kern, models
K = kern.RBF(input_dim=1, variance=1, lengthscale=1) # Gaussian kernel
K.plot()                                             # plot kernel
reg = GPy.models.GPRegression(X_trn, Y_trn, K)       # GP regressor
reg.optimize_restarts(num_restarts=10)               # train model
display(reg)                                         # display model parameters
```

```
reg.plot(plot_density=True)                    # plot predictive distribution and observations
```



Types of kernels ($d(x_i, x_j)$ is Euclidean distance):

- **Linear kernel** → $k(x_1, x_2) = \sigma^2 x_1^\top x_2$ where variance σ^2 is an amplitude parameter, provides linear solutions.
- **Gaussian (RBF) kernel** → $k(x_1, x_2) = \sigma^2 e^{-\frac{\|x_1 - x_2\|^2}{2l^2}}$, defined by variance σ^2 and length scale l^2.
- **Matérn kernel**: additional parameter ν controls smoothness ($\nu \to \infty$ becomes Gaussian, $\nu \to \frac{1}{2}$ becomes exponential).
  - **Matérn 3/2 kernel** ($\nu = \frac{3}{2}$) → $k(x_i, x_j) = \sigma^2 \left(1 + \frac{\sqrt{3}}{l} d(x_i, x_j)\right) e^{-\frac{\sqrt{3}}{l} d(x_i, x_j)}$
  - **Matérn 5/2 kernel** ($\nu = \frac{5}{2}$) → $k(x_i, x_j) = \sigma^2 \left(1 + \frac{\sqrt{5}}{l} d(x_i, x_j) + \frac{5}{3l} d(x_i, x_j)^2\right) e^{-\frac{\sqrt{5}}{l} d(x_i, x_j)}$
- **Rational quadratic kernel** → $k(x_i, x_j) = \sigma^2 \left(1 + \frac{d(x_i, x_j)^2}{2\alpha l^2}\right)^{-\alpha}$ with scale mixing parameter α (power), is a mixture of RBF kernels of different length scales.
- **Periodic exponential kernel** → $k(x_i, x_j) = \sigma^2 e^{-\frac{2\sin^2(\pi d(x_i, x_j)/p)}{l^2}}$ with periodic parameter p, repeats signal beyond observations instead of tending to 0.
- **Combination of kernels**: kernels can be linearly combined (summed and multiplied). Combined kernels can operate on different domains (e.g.: exponential kernels with periodicity in different dimensions) by setting active domains.
- **Automatic relevance determination** (ARD): kernel formula is adapted to include independent parameters (length scale) for each input feature. Since model is non-parametric, complex kernels can be defined with many hyperparameters without increasing computational needs a lot. Model is trained to learn importance of each feature (↓ l → higher relevance).

$$k(x_1, x_2) = \sigma^2 e^{-\sum_{j=1}^{M} \frac{(x_{1,j} - x_{1,j})^2}{2l_j^2}}$$

```
# Kernels in GPy
K = GPy.kern.Bias(input_dim=1)                              # bias (constant) kernel
K = GPy.kern.Linear(input_dim=1)                            # linear kernel
K = GPy.kern.RBF(1, variance=1, lengthscale=0.1)            # Gaussian kernel
K = GPy.kern.Exponential(1, 1, 0.1)                         # exponential kernel
K = GPy.kern.Matern32(1, 1, 0.1)                            # Matern 32 kernel
K = GPy.kern.Matern52(1, 1, 0.1)                            # Matern 52 kernel
K = GPy.kern.RatQuad(1, 1, 0.1, power=0.5)                  # rational quadratic kernel
K = GPy.kern.PeriodicExponential(1, 1, 0.1, period=2*np.pi) # periodic exponential kernel
K = K1*K2 + K3                                              # combination of kernels
K = GPy.kern.Exponential(1, active_dims=[0])                # exponential kernel acting only on dimension 0
K = GPy.kern.RBF(2, ARD=True)                               # kerneld defined for ARD
```

Computational cost of GP lies on inverting training kernel in each iteration, grows with size of training set $\mathcal{O}(N^3)$. To reduce it, *sparse GP* make predictions based on a subset of data.

- **Inducing points**: subset of samples created from the input feature space (they do not correspond to real training samples) used to provide a sparse GP solution. To be effective, the number of inducing points and their location in the feature space must be optimized. Prediction will be more accurate near inducing points.

```
# Implement sparse GP
```

```
Z = np.random.rand(12,1)  # initial location of inducing points (12 points)
reg = GPy.models.SparseGPRegression(X_trn, Y_trn, kernel=K, Z=Z)
reg.inducing_inputs.fix() # fix location of inducing points
reg.Z.unconstrain()       # allow optimization of inducing points location
reg.optimize()            # optimize
```

Deep Learning

Deep learning (DL) is type of ML that uses multilayer perceptron (MLP) or artificial neural networks. The structural unit works similar to biological neuron: inputs are (linearly) weighted and summed, then (non-linearly) transformed into an output value. Network is trained by tuning those weights in an attempt to optimize some loss function. Neural networks especially excel at automatically learning and extracting relevant features from images. A trained network can be reused or partially retrained in new problems. DL networks can be implemented with PyTorch↗ library (*torch*, *torchvision*…).

```
# Define transform to normalize data
from torchvision import datasets, transforms
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])

# Download data
trainset = datasets.MNIST('~/.pytorch/MNIST_data/', download=True, train=True,  transform=transform)
testset  = datasets.MNIST('~/.pytorch/MNIST_data/', download=True, train=False, transform=transform)

X_trn, Y_trn = trainset.data, trainset.targets # train dataset
X_tst, Y_tst = testset.data, testset.targets   # test dataset
```

Dense neural networks

Dense neural networks (DNN) are hierarchical compositions of simple mathematical functions organized in layers, used to correlate some input features x with a target value or label y. Complexity of network determined by number of layers and parameters. Easy to reach high accuracy but difficult to further improve it. Two types of layers (dense and activation) are alternated. First layer in the DNN (receives input) is dense, while last layer (returns output) is activation (softmax function in classification). Internal layers called *hidden* layers.

- **Dense layers**: parameters (weights and bias) are introduced to project (by linear combination) input values into a number of *logits* ($z = Wx + w_0$). Parameters are randomly initialized at beginning and optimized during training. More logits introduced and more dense layers → more parameters.

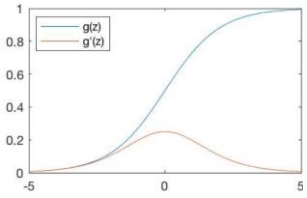
  ```
  # Dense (linear) layer
  from torch import nn
  joutput = nn.Linear(dim_x, dim_z) # z = W*x + w0
  ```

- **Activation layers**: an activation function (sigmoid, tanh, softmax, ReLU) transforms logits into *units* (values mapped to some integer representing a class), introducing non-linearities that allow for complex relations between input and output. No additional parameters (operation is elementwise, simplifies optimization).

  ```
  # Activation (non-linear) layer
  sigmoid    = nn.Sigmoid()          # g = 1/(1+exp(-z))
  tanh       = nn.Tanh()             # g = (exp(z) - exp(-z))/(exp(z) + exp(-z))
  logsoftmax = nn.LogSoftmax(dim=1) # g_i = log(exp(z_i)/sum(exp(z)))
  relu       = nn.ReLU()             # g = max(0, z)
  ```

Sigmoid function

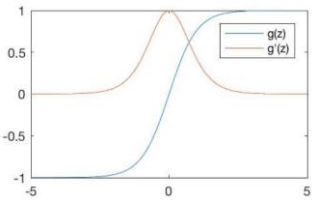
$$g = \frac{1}{1 + e^{-z}}$$



Maps to a (0, 1) range
(binary)

Hyperbolic tangent

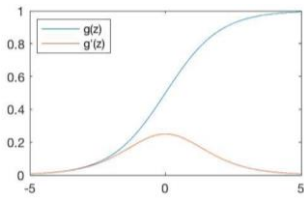
$$g = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$



Maps to a (–1, 1) range
(binary)

Softmax function

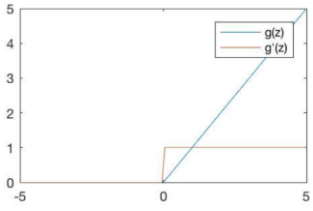
$$g_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$



Maps to a (0, 1) range
(normalized multiclass output)

Rectified linear unit (ReLU)

$$g = \max(0, z)$$



Maps to a (0, K) range
(multiclass output)

Network construction

Python implementation of neural networks follows object-oriented programming (OOP) manner: network is defined as a class, specifying all methods and attributes. Input to the network is a matrix x where each column corresponds to a feature and each row is a sample. All features are considered informative by themselves and contribute to output. When input samples are 2D (images), they

are flattened into vectors before feeding `forward` method (`x.view(x.shape[0], -1)`). Each pixel of the image will then be presented as a feature and spatial correlation is lost.

```
# Implement a dense neural network classifier
class DNN(nn.Module): # nn.Module provides useful methods and attributes for MLPs

  # Initialization of layer functions
  def __init__(self, dim_x, hidden, dim_y):
    super().__init__()                      # inherit methods from nn.Module
    self.output1 = nn.Linear(dim_x, hidden) # linear function (dense layer 1)
    self.output2 = nn.Linear(hidden, dim_y) # linear function (dense layer 2, hidden)
    self.relu = nn.ReLU()                   # activation function (hidden layer)
    self.logsoftmax = nn.LogSoftmax(dim=1)  # activation function (last layer)

  # Pass input tensor through all layers
  def forward(self, x):
    x = self.output1(x)     # layer 1 (dense, first)
    x = self.relu(x)        # layer 2 (activation)
    x = self.output2(x)     # layer 3 (dense)
    x = self.logsoftmax(x) # layer 4 (activation, last)
    return x

net = DNN(dim_x, hidden, dim_y) # DNN classifier
f = net.forward(X)              # predict output
```

Network training

Output predictions f are compared with test values y according to some loss function, which is optimized to find the weights that best correlate x and y. Optimization is done in iterative process: in each iteration, parameters are adjusted closer to optimum solution. Since loss function measures error, its evolution during training is used to assess performance across iterations (therefore loss value is stored).

- **Loss function** $\mathcal{L}$: defined between predictions f and target values y, depends on the parameters of the model. Typically non-convex (many local optima). E.g.: MSE (`.MSELoss`), negative log-likelihood (`.NLLLoss`), binary cross entropy (`.BCELoss`)…
- **Optimization method**: gradients of $\mathcal{L}$ w.r.t. weights are computed and used to update those weights. Usually involves a *learning rate* parameter that adjusts speed of optimization: ↑ learning rate → algorithm may skip optimum and diverge. ↓ Learning rate → long time to converge. E.g.: stochastic gradient descent (`.SDG`), Adam algorithm (`.Adam`)…

  ```
  # Train DNN with loss function optimization
  from torch import optim
  iterations = 20                           # number of iterations
  lr = 0.001                                # learning rate
  criterion = nn.NLLLoss()                  # loss function
  optim = optim.Adam(net.parameters(), lr) # optimization method
  trn_loss = []                             # evolution of training loss

  for i in range(iterations):
    f = net.forward(X_trn)                  # predict output
    loss = criterion(f, Y_trn)              # compute loss
    loss.backward()                         # compute gradients w.r.t. parameters
    optim.step()                            # update parameters
    trn_loss.append(loss.item())            # store loss value
  ```

- **Backpropagation**: gradients are evaluated from *top* (output layer) to *bottom* (input layer) to reuse computations since bottom gradients depend on top ones: by the chain rule, gradient δ_j in layer $l-1$ (bottom, closer to input) depends on sum of gradients in layer l (immediately top, closer to output, d_l dimensions) multiplied by weights w_{kj} and by derivative of activation function g' evaluated at logits z_j. Facilitated by elementwise activation functions.

$$\delta_j^{(l-1)} = g'\left(z_j^{(l-1)}\right)\sum_{k=1}^{d_l} w_{kj}\delta_k^{(l)}$$

- **Vanishing gradients**: because weights are typically < 1, gradients will become smaller as backpropagation moves to bottom layers. Therefore, training (and network learning) is not uniform across layers: parameters in bottom layers will not be trained as much or fast as in top layers, or not trained at all.
  - **Residual layer**: short paths are introduced to skip some layers when computing gradients. Used to combat vanishing gradients and train weights in bottom layers.
  - **Auxiliary classifier**: additional loss functions are evaluated at intermediate layers and then linearly combined and optimized. Used in very deep networks to train weights in layers previous to auxiliary loss function.
- **Mini-batch**: training dataset is divided in random batches (subsets) which are trained separately and their results averaged. Improves predictions by increasing diversity (same as in ensembles) and efficiency for large databases (faster convergence, used

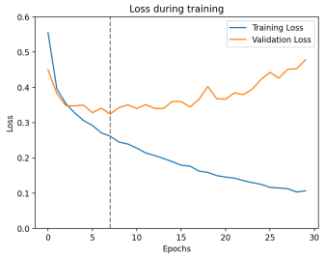
in SGD). An iterable object (loader) containing all batches is created and an optimization loop is used to iterate through it. Gradients are reset for each batch and loss value in one iteration is average loss of all batches.

- **Epoch**: when all training samples have been used at least once to update parameters, it is called an epoch (used as alternative to iterations). To guarantee that all samples are evenly used, one epoch uses batches of the training dataset without replacement (each sample only appears in one batch). In next epoch (when loader is iterated again), samples are shuffled and mini-batches are created again.

```
# Train DNN with mini-batches and epochs
from torch.utils.data import DataLoader
epochs = 20                          # number of epochs
batch_size = 64                      # batch size
trn_loss = []                        # evolution of training loss
loader = DataLoader(trainset, batch_size=batch_size, shuffle=True) # loader (iterable) object with batches

for e in range(epochs):              # for each epoch
  epoch_loss = []                    # epoch loss (vector of batch losses)
  for X, Y in trainloader:           # for each batch
    optim.zero_grad()                # reset gradients
    f = net.forward(X)               # predict output
    loss = criterion(f, Y)           # compute loss
    loss.backward()                  # compute gradients
    optim.step()                     # update parameters
    epoch_loss.append(loss.item())   # store batch loss

  trn_loss.append(np.mean(epoch_loss) # store epoch loss
```

Network regularization

A DNN with a few layers and dimensions will probably overfit due to *memorization*: network has enough degrees of freedom (DOF) to perfectly adapt to training data (low training loss), resulting in bad *generalization* to new data (test loss starts to increase). Memorization usually happens at top layers (close to output, weights are trained faster) and means the network is complex enough. Regularization tries to control overfitting by using a validation set.



- **Early stopping**: training is finished before overfitting occurs by setting a low number of epochs. This number is determined according to validation loss (choose number of epochs that yields lowest validation loss, or where validation loss is stabilized).
- **Dropout layer**: hidden units are randomly dropped out (set to 0) to reduce DOF of model, delaying or preventing memorization (network cannot rely on any node). More epochs needed to converge. Dropout layers are defined after activation layers, typically near top. Dropout is forced during training according to some probability (of being dropped out), but is not applied during validation or testing.

```
prob = 0.2                           # probability of dropout (20%)
self.dropout = nn.Dropout(p=prob)    # define dropout layer
x = self.dropout(x)                  # apply dropout layer
self.train()                         # turn on dropout (training)
self.eval()                          # turn off dropout (validation, testing)
```

During training, weights of each layer are updated and the statistics (mean, variance) of data will change (*internal covariate shift*), leading to non-normalized inputs to hidden layers. Mini-batches also present a shift in their distribution, compared to the whole dataset. This affects convergence of the network.

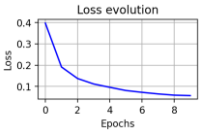
- **Batch normalization**: data is normalized to a fixed distribution (predefined mean γ and variance β) before each layer.
- **Layer normalization**:

Network evaluation

- **Accuracy**: predicted labels are compared with real ones and the accuracy of the model is obtained. If model output is soft (probability of each class), it is converted to hard (most probable class). Gradient tracking is turned off to save computation costs. Boolean values need to be transformed into numerical tensors (or "detached" and converted to *NumPy* arrays).

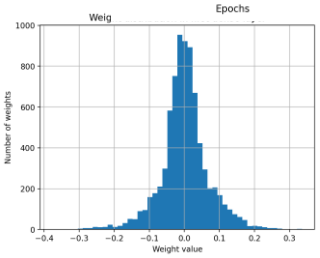
```
with torch.no_grad():        # turn off gradient computation
  o = net.forward(X)         # prediction as soft output (probability of each label)
  _, f = o.topk(1, dim=1)    # prediction as hard output (label)
  accuracy = np.mean(f==Y)   # accuracy of DNN classifier
```

- **Loss evolution**: loss function measures error between prediction and target values so its evolution during training shows the performance of optimization along several epochs.



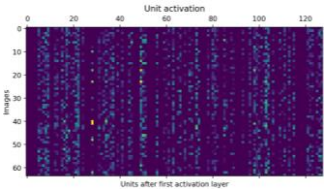

```
plt.plot(epochs, trn_loss) # plot loss along epochs
```

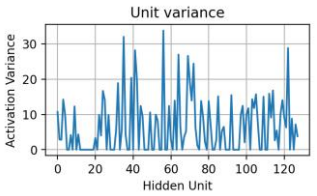
- **Weight analysis**: weights of each dense layer and their gradients can be extracted from the model and analyzed to find overfitting or bias problems. Weight histogram should be centered at 0, bell-shaped and symmetric.




```
W1 = net.output1.weight        # get weight matrix of dense layer 1
grad = net.output1.weight.grad # get gradient of weights
plt.hist(W1.reshape(-1))
plt.hist(grad.reshape(-1))
```

- **Unused units**: units of hidden layers that are always active/inactive are unused and could be removed from network to reduce computations (reducing logits dimension). Variance of unit activation along all images can be used to detect unused units (they have variance close to 0).






```
X_batch, Y_batch = next(iter(testloader)) # get one mini-batch
z = net.output1(X_batch)                  # compute logits
units = net.relu(z)                       # compute units
plt.matshow(units.detach().numpy())       # show activation
plt.plot(np.var(units, axis=0))           # show activation variance
```

- **Layer correlation**: unit activation map should be sparse across samples. Layers that show correlation (repeat same activation pattern than neighboring layers) are not contributing and should be removed.

- **Feature explainability**: having a trained DNN, *activation maximization* is used to find which features of input are relevant to output (e.g., which genetic biomarkers influence the most to some disease). Network is used to analyze features rather than to classify. DNN parameters and soft output are fixed, then input is randomly initiated and optimized to match the output. Only relevant feature dimensions will be adjusted.
    - **INVASE**: method to select relevant features, uses two DNN: *actor* trained with all features, and *critic* trained with a random subset of features provided by a feature selector. Output of both are compared for different feature subsamples in order to find most relevant features (feature selector is trained).

Network saving and loading

Network parameters (weight matrix and bias of each dense layer) are accessed through the `state_dict` attribute. They can be saved, loaded and assigned to a network.

```
net.state_dict()                              # dictionary with parameters
torch.save(net.state_dict(), 'file_name.pth') # save network parameters
state_dict = torch.load('file_name.pth')      # load network parameters
net.load_state_dict(state_dict)               # assign parameters to network
```

Network implementation

Training loop, regularization and accuracy evaluation can be introduced in the model class as new methods.

```
# Implement a DNN classifier with all methods
class DNN(nn.Module):

  # Initialization of parameters and methods
  def __init__(self, dim_x, hidden1, hidden2, dim_y, prob, epochs=100, lr=0.001):
    super().__init__()
    self.output1 = nn.Linear(   dimx, hidden1)          # dense layer 1
    self.output2 = nn.Linear(hidden1, hidden2)          # dense layer 2
    self.output3 = nn.Linear(hidden2, dim_y)            # dense layer 3
    self.relu = nn.ReLU()                               # activation layer
    self.dropout = nn.Dropout(p=prob)                   # dropout layer
    self.logsoftmax = nn.LogSoftmax(dim=1)              # activation layer (last)

    self.epochs = epochs                                # number of epochs
    self.lr = lr                                        # learning rate
    self.criterion = nn.NLLLoss()                       # loss function
    self.optim = optim.Adam(self.parameters(), self.lr) # optimization method
    self.trn_loss = []                                  # evolution of training loss
    self.val_loss = []                                  # evolution of validation loss

    self.device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    self.to(self.device)                                # use GPU if available
```

```
  # Pass input tensor through all layers
  def forward(self, x):
    x = self.output1(x)     # layer 1: dense (first)
    x = self.relu(x)        #          activation
    x = self.dropout(x)     #          dropout
    x = self.output2(x)     # layer 2: dense
    x = self.relu(x)        #          activation
    x = self.dropout(x)     #          dropout
    x = self.output3(x)     # layer 3: dense
    x = self.logsoftmax(x) #          activation (last)
    return x

  # Network is trained on data
  def fit(self, trainloader, validloader):
    for e in range(self.epochs):
      epoch_loss = []

      # Training loss
      for X, Y in trainloader:
        X = X.to(self.device)
        self.optim.zero_grad()
        f = self.forward(X)
        loss = self.criterion(f, Y)
        loss.backward()
        self.optim.step()
        epoch_loss.append(loss.item())

      self.trn_loss.append(np.mean(epoch_loss))

      # Validation loss
      self.eval()
      with torch.no_grad():
        epoch_loss = []
        for X, Y in validloader:
          X = X.to(self.device)
          f = self.forward(X)
          loss = self.criterion(f, Y)
          epoch_loss.append(loss.item())

        self.val_loss.append(np.mean(epoch_loss))
      self.train()

  # Accuracy evaluation
  def score(self, X, Y):
    with torch.no_grad():
      X = X.to(self.device)
      o = self.forward(X)
      _, f = o.topk(1, dim=1)
      accuracy = np.mean(f==Y)

    return accuracy

net = DNN(dim_x, hidden1, hidden2, dim_y, prob, epochs, lr) # DNN classifier
net.fit(trainloader, validloader)        # train DNN
net.forward(X_tst)                       # predict output
accuracy = net.score(X_tst, Y_tst)       # predict output and compute accuracy
torch.save(net.state_dict(), 'file.pth') # save network parameters
```

CONVOLUTIONAL NEURAL NETWORKS

Style transfer

1D CNN

Graph CNN

Region CNN

- Faster region CNN:

RECURRENT NEURAL NETWORKS

Natural Language Processing

Natural language processing (NLP) is the process of extracting information from human text. NLP models trained on a *corpus* (collection of text documents). Data corpora can be downloaded from NLTK↗ library.

```
# Download corpus
import nltk
from nltk.corpus import movie_reviews as corpus
nltk.download('movie_reviews') # corpora from NLTK need to be imported and downloaded
id = corpus.fileids()          # document IDs
corpus.categories()            # document labels
corpus.raw(id)                 # document raw text
```

TEXT NORMALIZATION

Input text is preprocessed and standardized to facilitate its later vectorization.

Normalization with NLTK

The same NLTK library contains functions for each step in normalization.

- **Tokenization**: text string of a document is split into *tokens* (words, numbers, punctuation marks, symbols…).

```
# Tokenize a text document
nltk.download('punkt')                 # download model to identify punctuation marks
text = corpus.raw(corpus.fileids()[0]) # extract text from first document in corpus
tokens = nltk.sent_tokenize(text)      # divide text into sentences
tokens = nltk.word_tokenize(text)      # divide text into words
```

- **Cleaning**: non-alphanumeric characters (punctuation), *stop words* (words without semantic content: determinants, prepositions, pronouns…) and very rare or common terms are removed to reduce dimensionality and focus on meaning. Typos or spelling errors can be corrected.

```
# Clean text
from nltk.corpus import stopwords
nltk.download('stopwords')
stopwords_en = stopwords.words('english')              # download stop words
tokens = [t for t in tokens if t not in stopwords_en] # remove stop words from text
tokens = [t for t in tokens if t.isalnum()]           # remove non-alphanumeric tokens
```

- **Homogenization**: tokens with same meaning but different style are unified (same verb in different tenses, singular and plural, lowercase and uppercase) and words are reduced to their root by *stemming* (removes grammatical inflections, affixes) or *lemmatization* (derives the lexical base form). Reduces dictionary size and computational costs.

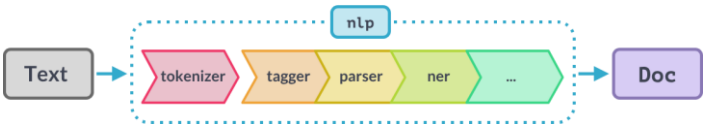
```
# Homogenize tokens
from nltk.stem.wordnet import WordNetLemmatizer
nltk.download('wordnet')
lemmatizer = WordNetLemmatizer()
tokens = [lemmatizer.lemmatize(t) for t in tokens] # lemmatize tokens
tokens = [t.lower() for t in tokens]               # convert to lowercase
```

- **N-gram detection**: groups of words that often appear together (e.g., *machine learning*) are identified and considered a single token, based on statistical co-occurrence.

Normalization with spaCy

A pre-trained language model (with normalization features and vocabulary dictionary) can be loaded from spaCy↗ to process text into `doc` objects containing `token` objects with several attributes.

- **Tokenizer**: splits text into tokens.
- **Tagger**: assigns a part-of-speech (POS) tag to each word (verb, noun, adjective…).



- **Parser**: identifies syntactic relationships (subject, object, complements…).
- **Named entity recognition** (NER): identifies named entities (people, places, organizations…), similar to N-gram detection.

```
import spacy
spacy.explain('VBP')                # explain meaning of an attribute tag
nlp = spacy.load('en_core_web_sm') # download small English NLP model
list(nlp.vocab.strings)             # vocabulary of the model
list(nlp.Defaults.stop_words)       # stop words in the model

doc = nlp(text)                     # process text
token = doc[0]                      # access token in document
span = doc[0:10]                    # access span of tokens
token.text                          # original text content
token.lemma_                        # base form (lemma) of token
token.pos_                          # POS tag
token.tag_                          # finer POS tag
```

```
token.is_alpha                    # whether token is alphabetic
token.is_digit                    # whether token is numeric
token.is_stop                     # whether token is stop word
token.is_punct                    # whether token is punctuation mark
doc.ents                          # named entities in doc
```

Normalization pipeline

All these processes can be combined into a normalization pipeline, which is applied to the whole corpus:

```
# Define text normalization functions
def normalize_NLTK(text):
    tokens = [t.lower() for t in nltk.word_tokenize(text) if t.isalnum()]
    tokens = [lemmatizer.lemmatize(t) for t in tokens if (t not in stopwords_en)]
    return tokens

def normalize_spaCy(text):
    tokens = nlp(text)
    tokens = [t.lemma_.lower() for t in tokens if (t.is_alpha or t.is_digit) and not t.is_stop]
    return tokens

# Create corpus with normalized text
corpus_norm = []
for id in corpus.fileids():
    text = corpus.raw(id)
    tokens = normalize_spaCy(text)
    corpus_norm.append(tokens)
```

TEXT VECTORIZATION

Normalized text is converted into numerical format: each token, sentence or document is represented as a vector. Necessary for further applications (text classification, summarization…).

Document embeddings

Each doc is represented by a vector based on the importance (presence) of the words contained in it. Similar documents are represented with similar vectors, and vice versa.

- One-hot encoding:
- Bag-of-words (BoW): vocabulary dictionary size d. Information about word order, grammar and syntax is lost.
- TF-IDF: constructed from BoW.
- More advanced vectorization techniques in topic modeling.

Word embeddings

Tokens are projected into a latent space that captures semantic relationship between words. Each token is then represented as a vector. Words that are semantically similar are projected into similar vector representations, and unrelated words to points that are far apart.

- Non-contextual: use nearby words to learn a word's embedding. word2vect, glove, fasttext
- Contextual: use whole document at once to obtain embedding. transformers
- Check semantic coherence: t-SNE, most_similar, semantic operations

TEXT CLASSIFICATION

TOPIC MODELING

Unsupervised method to identify and characterize main topics in a collection of documents, and to represent (vectorize) a doc according to its topics. Based on *co-occurrence* of words (BoW or TF-IDF). Topics are characterized by specific words that share a similar context. Documents are vectorized semantically by the contribution of each topic to them. Implemented with Gensim↗ or MALLET↗.

Topic modeling with Gensim

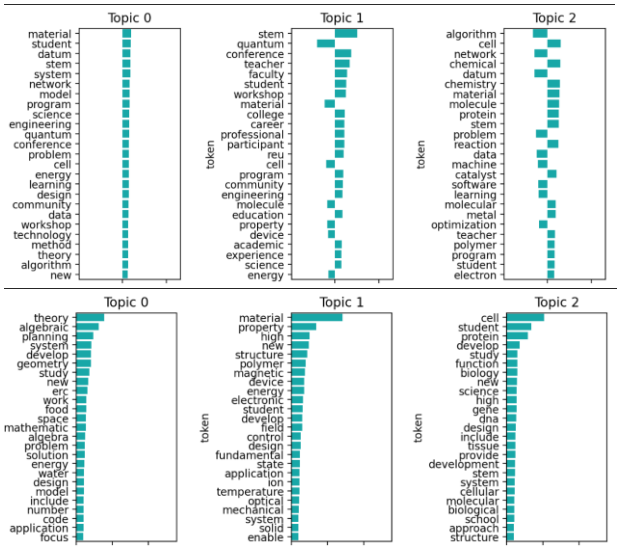
- **Latent semantic indexing** (LSI): documents are projected into a latent space of dimensionality k (number of topics) through singular value decomposition. Docs given as a BoW or (preferably) TF-IDF matrix A of size $n \times d$ (number of docs × dictionary size), which is decomposed into three matrices (the first k singular values are used to approximate A):

$$A \approx DST^\top$$

- $D_{n\times k} \rightarrow$ unitary *document-topic* matrix, provides weight of each topic for a particular document. Topic contribution can be positive (relevant) or negative (uncorrelated). Obtained by applying LSI transformation to corpus.
- $S_{k\times k} \rightarrow$ diagonal matrix of rank k, where only most relevant singular values (topics) are kept.
- $T^\top_{k\times d} \rightarrow$ unitary *topic-token* matrix, provides weight that each token has for every topic. Token contribution can be positive (relevant) or negative (uncorrelated). Visualized through `show_topics` and `show_topic` functions.

```
# Implement LSI model
from gensim.models import LsiModel
k = 20                  # number of topics
D = Dictionary(corpus) # vocabulary dictionary
lsi = LsiModel(corpus_tfidf, num_topics=k, id2word=D) # LSI model
corpus_lsi = lsi[corpus_tfidf]              # transform corpus (obtain document-topic matrix)
lsi.show_topics(num_topics=k, num_words=10) # print 10 most relevant tokens of each topic
lsi.show_topic(i, topn=25)                  # get 25 most relevant tokens of topic i and their weight
```

Topic 0 provides an average of the whole corpus (contains most common tokens in the collection) and can be used to center data. All other topics are characterized by positive and negative weights, which may encompass two contrast topics (one for each sign) or just one topic (one sign associated to a topic, the other one containing common words without particular significance).



- **Latent Dirichlet allocation** (LDA): probabilistic, generative model where topics are characterized by probability distributions over tokens ($\beta_{1:k}$), and documents are characterized by probability distributions over topics ($\theta_{1:n}$). Distributions are Dirichlet: all weights positive, sum up to 1, sparse representation. Docs given as BoW only. Two parameters control sparsity:
  - $\eta \rightarrow$ controls sparsity of topic characterization: $\uparrow \eta \rightarrow$ flatter density $\rightarrow$ more tokens describe each topic.
  - $\alpha \rightarrow$ controls sparsity of document characterization: $\uparrow \alpha \rightarrow$ more topics define each document.

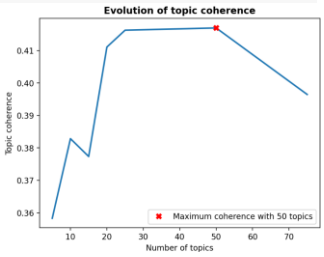

```
# Implement LDA model
from gensim.models.ldamodel import LdaModel
lda = LdaModel(corpus_bow, num_topics=k, id2word=D) # LDA model
corpus_lda = lda[mycorpus_bow]              # transform corpus
lda.show_topics(num_topics=k, num_words=10) # print 10 most relevant tokens of each topic
lda.show_topic(i, topn=25)                  # get 25 most relevant tokens of topic i and their weight
```

Unlike LSI, topic 0 is just another topic, like all others. Some topics may be more informative than others. Some persistent tokens with no significant contribution appear in most of the topics.

- **Topic visualization**: visualization is crucial to understand results of topic modeling. Module *pyLDAvis* used to visualize and interact with LDA model. Also, see semantic graphs for more advanced visualization techniques.

```
# Visualize LDA model
!pip install pyLDAvis==3.4.1
import pyLDAvis
vis_data = pyLDAvis.gensim.prepare(lda, corpus_bow, D)
pyLDAvis.display(vis_data)
```

- **Coherence models**: provide a metric to analyze impact of number of topics, can be used to select appropriate number.




```
# Visualize LDA model
from gensim.models.coherencemodel import CoherenceModel
cm = CoherenceModel(lda, texts=corpus, dictionary=D) # coherence model
cm.get_coherence()                                   # get coherence metric
```

Topic modeling with MALLET

Java-based LDA implementation with Monte Carlo optimization that allows parallelization.

```
# Install Java
import os
```

```
def install_java():
    !apt-get install -y openjdk-8-jdk-headless -qq > /dev/null    # install openjdk
    os.environ['JAVA_HOME'] = '/usr/lib/jvm/java-8-openjdk-amd64' # set environment variable
    !java -version                                                # check Java version
install_java()

# Install MALLET
!wget http://mallet.cs.umass.edu/dist/mallet-2.0.8.zip
!unzip mallet-2.0.8.zip
os.environ['MALLET_HOME'] = 'mallet-2.0.8'
mallet_path = 'mallet-2.0.8/bin/mallet'

# Transform corpus into binary format and import
with open('corpus.txt', 'w') as f:
    for d,doc in enumerate(corpus):
        f.write(f'{d} 0 {doc}\n')
!mallet-2.0.8/bin/mallet import-file --input corpus.txt --output corpus.mallet --keep-sequence --remove-stopwords

# Train MALLET model on corpus
!mallet-2.0.8/bin/mallet train-topics --input corpus.mallet --num-topics 20 --num-iterations 100

# Additional arguments to export results
    --output-doc-topics doc_topics.txt                    # Export topic composition per doc
    --topic-word-weights-file topic_weights.txt           # Export token composition per topic
    --word-topic-counts-file wtc_counts.txt               # Export sparse representation of topic-token
    --output-topic-keys topic_keys.txt --num-top-words 20 # Export 20 most relevant tokens per topic
```

SEMANTIC GRAPHS

Semantic graphs are visual representations of a corpus based on the semantic relationships between the different documents.

Similarity matrix

Once documents in a corpus have been vectorized with a topic modeling algorithm (such as LDA), *similarity measures* can be computed between each document pair (x_1, x_2):

- **Bhattacharyya coefficient** (BC): used with probabilistic embeddings, bounded to $[0,1]$ range → $s_{BC}(x_1, x_2) = \sqrt{x_1}^{\top}\sqrt{x_2}$
- **Cosine similarity**: bounded to $[-1,1]$ range → $s_{CS}(x_1, x_2) = \frac{x_1^{\top} x_2}{\|x_1\|\|x_2\|}$

BC can be computed at once between one doc and whole corpus, or of the corpus with itself. This last returns the similarity matrix $S = \sqrt{X}\sqrt{X}^{\top}$, needed to create semantic graph. S can be used to represent the corpus as a connected, weighted graph where each *node* is a document, *edges* link similar documents, and edge *weights* are the similarity measures. Some additional steps below.

```
# Construct sparse similarity matrix
import scipy.sparse as scsp
X = scsp.csr_matrix(corpus_lda)    # corpus given as sparse, LDA representation
S = np.sqrt(X) * np.sqrt(X).T      # compute Bhattacharyya coefficients
S = scsp.triu(S, k=1)              # exploit sparsity and symmetry to save memory
S.data[S.data < thr] = 0           # remove edges under threshold `thr`
S.eliminate_zeros()

# Obtain semantic graph from similarity matrix
import networkx as nx
G = nx.from_scipy_sparse_array(S) # semantic graph
lcc = max(nx.connected_components(G), key=len)
G = G.subgraph(lcc)               # largest connected component subgraph
```

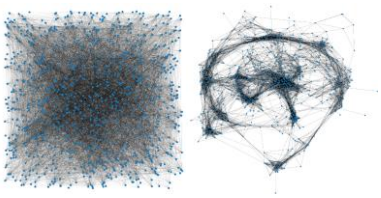
- S is *symmetric* and *sparse* if doc vectorization was sparse, and diagonal is all ones (allows to save memory).
- Removing edges with similarity value under a *threshold* helps reduce computations, clean the graph and focus on relevant connections (and lower bound of similarity can be restored from remaining graph paths).
- After thresholding, nodes or groups with low similarity may be isolated from the rest of the graph. We can focus on the *largest connected component*.

Layout algorithms

Semantic graph created with NetworkX↗ library. Layout algorithms produce a useful visualization of the graph where similar documents are projected together and disconnected or weakly connected nodes are far apart. *Force-directed algorithms* move nodes iteratively following repulsive and attractive forces defined by proximity to other nodes.

- **Random layout**: not useful, connections not distinguished at all.

```
positions = nx.random_layout(G)
nx.draw(G, positions) # show G with random layout
```



- **Fruchterman-Reingold**: used with small- or medium-size networks.

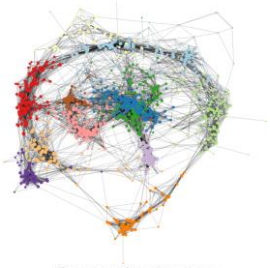
```
positions = nx.spring_layout(G)
nx.draw(G, positions) # show G with Fruchterman-Reingold algorithm
```

- **Force Atlas**: alternative for large networks, finds trade-off between convergence speed and graph quality.

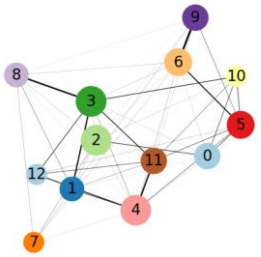
Community detection

Community detection algorithms identify clusters in the graph by searching for a partition of the nodes so that nodes in same subset (community) are strongly connected while nodes from different subsets are weakly related. Some quality metrics and methods:

- **Modularity**: sum of edge weights that fall within a community minus sum of weight edges that would fall within the community if edges were distributed at random, used to measure accuracy of community structure. Modularity goes from $-\frac{1}{2}$ (nodes within community are disconnected) to 1 (each community is an independent, fully connected subgraph).
- **Louvain algorithm**: greedy approach to estimate partition with best modularity, starts with single-node communities and iteratively groups them in bigger communities so that modularity is maximally increased.



Community structure




```
# Find communities and compute quality metrics
import networkx.algorithms.community as nx_comm
C = nx_comm.louvain_communities(G)
modularity = nx_comm.modularity(G, C)
coverage, performance = nx_comm.partition_quality(G, C)
```

- **Coverage**: ratio of the number of intra-community edges to the total number of edges in the graph (from 0 to 1).
- **Performance**: number of intra-community edges plus inter-community non-edges divided by the total number of potential edges (from 0 to 1).
- **Community graph**: can be made from a matrix of inter-community similarities, analogous to semantic graph.

Centrality measures

Evaluate nodes according to their position in the graph.

- **Degree centrality**: sum of edge weights of a node (or number of nodes in unweighted graph).

```
centrality = nx.degree_centrality(G) # compute degree centrality
```

- **Closeness centrality**: average length of the shortest path from a node to all other nodes in the graph. Based on distance between nodes, which can be computed from similarities: in the case of BC similarity, *Hellinger distance* is $d_H = 1 - s_{BC}$

```
S_bc = nx.get_edge_attributes(G, 'weight')                    # get BC similarity (edge weights)
D_h = {edge:{'distance':1-s} for edge,s in S_bc.items()}      # compute Hellinger distance
nx.set_edge_attributes(G, distance)                           # add Hellinger distance to graph
centrality = nx.closeness_centrality(G, distance='distance') # compute closeness centrality
```

- **Betweenness centrality**: number of shortest paths between any pair of nodes in the graph that pass through a specific node. Can be used to find documents that share two topics entirely.

```
centrality = nx.betweenness_centrality(G, weight='distance') # compute betweenness centrality
```

- **PageRank**: starting from node n, we perform a random walk through the graph, jumping to any of the neighbour nodes with probabilities proportional to graph weights. With some probability d (hyperparameter) we jump back to original node. PageRank accounts for the expected frequencies of visiting a given node through this walk.

```
centrality = nx.pagerank(G) # compute PageRank
```

Computer Vision

DENSE MOTION ESTIMATION

- **Motion field**: actual 3D displacement of objects in a scene, projected into 2D when filmed in video.

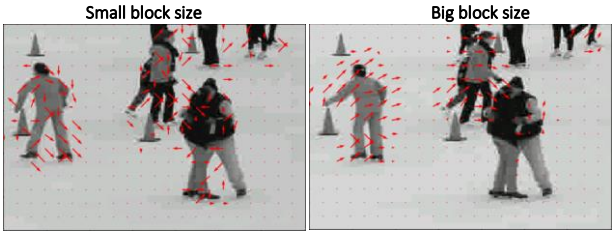
- **Optical flow**: approximation of motion field in a video based on change of pixel intensities between frames. Motion vector $\vec{u} = (u, v)$ used to define direction and magnitude of a pixel's or region's displacement.

In dense motion estimation (DME), image is divided into a grid and motion vector is estimated at each point of the grid. Number of parameters: $\text{grid height} \times \text{grid width} \times \#\text{ of frames} \times 2$. Finer grid → increased *granularity* → more parameters (and computations). Used in video compression.

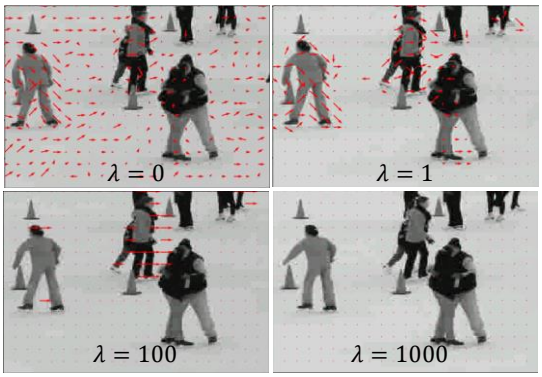
Block-based motion estimation

A region of interest (reference block) is cropped from image and matched to its corresponding region in the following frame (target block). Motion assumed to be constant for all block, without rotation (only panning, tilting) and bounded by a maximum velocity. Depends on video contents. Bigger block size and search area increase computational cost.

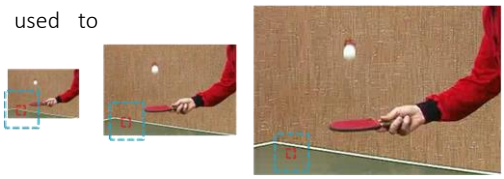
- **Block size**: size of reference block, adapted to object size.
  - Small → no discriminative information, greatly affected by noise, motion vectors point in different directions.
  - Big → uses more surrounding pixels so arrows are smoother, but may include other moving objects and transmit displacements to surroundings.



- **Search area**: region centered at reference pixel where target block is searched for, limits maximum velocity.
  - Small → if object moves fast, it may lie out of search area.
  - Big → may include other objects similar to target and mismatch them.
- Similarity or error metrics: compare reference (I_0) and target (I_1) blocks to assess matching quality; error metrics optimized at minimum, similarity metrics at maximum (or $1 - \text{metric}$).
  - **Sum of squared differences** (error): sensitive to illumination changes → $SSD = \sum |I_1(x_i + u) - I_0(x_i)|^2$
  - **Sum of absolute differences** (error): not differentiable in origin → $SAD = \sum |I_1(x_i + u) - I_0(x_i)|$
  - **Bias and gain** (error): adapts I_0 to intensity in I_1, need to optimize α and β → $\sum [I_1(x_i + u) - (1 + \alpha)I_0 - \beta]^2$
  - **Normalized cross-correlation** (similarity): assumes linear relation → $NCC = \sum \frac{(I_0(x_i) - \mu_{I_0}) \cdot (I_1(x_i + u) - \mu_{I_1})}{\sigma_{I_0} \cdot \sigma_{I_1}}$
- **Regularization**: a term is introduced to enforce small motion vectors, depends on λ → $C(u) = D(u) + \lambda \|u\|_2$, where $D(u)$ is an error metric.
  - Small → weak regularization, motion vectors not bounded in size so almost all points show an arrow (even static regions).
  - Big → strong regularization, small vectors are favored so few arrows (or none) appear, all pointing in direction of overall motion.



- **Search pattern**: reach optimal location in lowest number of steps.
  - Full search: all possible solutions within search area are tested (brute force algorithm, expensive).
  - Gradient-based methods: move in direction of error gradient (requires convex cost function).
- **Hierarchical motion estimation**: image pyramid of different scales used to progressively improve motion estimation from low to high scales by centering search area on previous scale's solution; improves robustness and precision.



Fourier-based motion estimation

Differential motion estimation

Applied to whole image, based on gradient restriction equation $\nabla I(x,t)u(x,t) + I_t(x,t) = 0$ where $\nabla I(x,t)$ is image gradient and $I_t(x,t)$ is derivative w.r.t. time.

- **Horn & Schunk**: regularization over vector field so that contiguous pixels present similar motion.
- **Lucas & Kanade**: assumes constant motion in neighborhood around pixel.

- **Farneback**: not differential method. Similar to block-based but instead of matching blocks, it uses polynomial models to estimate surfaces from pixel values, and compares those functions.