**Supervised and unsupervised learning:**

* L’unsupervised learning crea delle strutture per dividere i dati secondo i pattern che riesce a riconoscere, non ha quindi in input ne il numero di classi ne le classi, tenta invece di “trovarle” sulla base delle differenze nei dati.
* Il supervised learning invece avendo a disposizione le label (quindi le classi), “è in grado di predire” a quale classe appartengono gli elementi, osservando i pattern nei dati e classificandoli di conseguenza.

**Steps:**

* Pre-processing
* Training of the models
* Evaluation of the models

**Pre-processing:**

* Imputers (to deal with missing data)
* Encoders for categorical data
* Scalers for numerical data
* Dimensionality reduction
* Sampling

La fase di pre-processing serve a rendere il dataset “pulito” e pronto per essere dato in pasto ai classifiers.

**Missing data:** il nostro dataset deve essere “pulito” dato che i classifier non sono in grado di gestire dati mancanti; per gestire i missing data si può agire in 3 modi, eliminando la colonna con missing values (quindi eliminando tutta la feature), eliminando le righe con determinati missing values (si possono eliminare tutte le righe con almeno un missing value oppure si può decidere di eliminare solo quelle che hanno un missing value per una colonna critica per poi agire con gli imputer per le colonne più gestibili); l’ultima alternativa sono gli **imputers.**

Gli imputers sono degli algoritmi che hanno l’obiettivo di “fillare” i dati mancanti, possono essere di più tipi:

* KNN
* Simple
* Others, based on probabilistic or linear algorithms

**KNN imputer:** si basa sull’algoritmo “k-nearest neighbors” che può anche essere usato come classifier, in sostanza cerca gli elementi del dataset con più somiglianza a quello con il missing data e ne copia il valore (si può usare solo per i numerical data).

**Simple imputer:** non fa altro che riempire i missing data con strategie che richiedono poco calcolo ma che possono risultare poco rappresentative (media, moda e mediana sono le tre strategie utilizzabili).

Chiaramente il KNN imputer risulterà più preciso, ma dovendo runnare ogni volta un algoritmo per trovare i “neighbors” per ogni row con dato mancante, richiederà molta capacità computazionale, quindi molto più tempo per essere eseguito, mentre il simple imputer fa un unico calcolo per colonna (media, mediana o moda) e ne riporta il valore in ogni occorrenza mancante.

**Encoders:** gli encoders servono per trasformare le categorical feature in numerical, dato che le stringhe di testo per un computer non sono rappresentative (un computer non sa le somiglianze e le differenze tra un “cane” e un “gatto”, ma sa perfettamente che 1 è diverso da 2, quindi diremo cane = 1 e gatto = 2 per permettere al pc di capirne le differenze).

Lista encoders:

* Ordinal encoder
* One hot encoder
* Others like label, binary, base n, target and so on

**Ordinal encoder:** trasforma le possibili occorrenze di una feature in numeri interi, quindi mappa ogni possibile valore categorico ad un valore numerico intero; è molto utile quando si tratta con valori con un ordine intrinseco (tipo la taglia delle magliette, S, M, L…).

**One hot encoder:** per ogni possibile occorrenza di una feature crea una nuova colonna booleana che metterà true (1) in caso quella determinata riga ci fosse quel valore categorico e false (0) altrimenti.

Un parametro molto utile è quello del “drop”, che permette di eliminare.

Il one hot encoder creando una colonna per ogni possibile occorrenza rischia di far aumentare di troppo il numero di colonne del dataset, d’altra parte però l’ordinal encoder va in errore quando in fase di test incontra nuove occorrenze che non sa come mappare (se per esempio non incontra mai il valore “cane” nella fase di train, quando andrà a testare non saprà come mapparlo ed andrà in errore).

PRO: altri encoders possono essere il binary (utilizzabile quando i valori categorici sono solo 2, per esempio può trasformare maschio e femmina in 0 e 1 mantenendo la colonna iniziale), oppure il base n (quando i possibili valori sono n), target che sostituisce la categoria con la media dell’outcome di quell’occorrenza.

**Scalers:** l’obiettivo degli scaler è rendere i dati numerici più “leggibili” per il computer e in scala uniforme fra le varie feature (per esempio nel KNN o nelle neural networks è molto importante che le varie feature abbiano lo stesso range), altri classifier hanno invece bisogno di dati che siano normalizzati (generalmente media=0 e deviazione standard=1).

Lista scalers:

* Standard
* Min max
* Others like Max abs, quantile, normalizer and so on

**Standard:** lo standard scaler standardizza i dati secondo una media e una deviazione standard che di default sono quelli della normale (media=0, std=1, si tratta quindi di uno scaler lineare).

**Min max:** comprime tutti i dati in un range preimpostato che di default è [0, 1], ma che può essere personalizzato (si tratta quindi di uno scaler lineare).

PRO: altri scaler possono essere per esempio il max abs che divide i dati per il massimo assoluto (quindi per trovarlo guarda anche i dati negativi e gli applica il valore assoluto), ha la caratteristica di mantenere il segno del valore iniziale (impossibile con il min max se il valore massimo negativo e quello positivo non coincidono)

**Dimensionality reduction:** l’obiettivo di questa fase è diminuire il più possibile le dimensioni del dataset mantenendo però le informazioni più rilevanti (dato che sicuramente ci sono feature con più o meno peso).

Lista tecniche:

* PCA (unsupervised)
* LDA (supervised)
* Sequential feature selection (supervised)
* Sequential backward selection (supervised)

**PCA:** crea delle combinazioni lineari partendo dalle feature, creando i “principal components” (PC1, PC2…).

Questa tecnica fa parte delle “unsupervised”, infatti la PCA non conosce le classi, ma solo le feature degli elementi.

Prima di cercare le combinazioni lineari, serve normalizzare i dati.

Immaginando un grafico a 3 dimensioni, si trova la retta che permetta ai dati di avere maggior varianza, questo sarà il nostro PC1, poi il PC2 una retta perpendicolare a PC1 (), e insieme formeranno un grafico a 2 dimensioni.

**LDA:** crea delle combinazioni lineari come la PCA, ma essendo un “supervised algorithm”, e conoscendo le classi, fa in modo di massimizzare la distanza tra il punto medio di ogni classe e minimizzare la varianza all’interno delle classi.

**Sequential feature selection:** cerca di trovare il miglior compromesso tra numero di feature e performance, per fare ciò parte cercando la miglior performance con 1 sola feature (quindi le tenta tutte singolarmente), trovata quella che singolarmente si comporta meglio, viene cercata la seconda (sempre tentandole tutte in combinazione con la prima che ormai è fissa), va avanti cosi finchè non raggiunge il numero massimo di feature comunicato.

Selezionerà poi il subset di feature con performance più alta.

Viene definita soluzione sub-ottimale, dato che non tenta tutte le combinazioni possibili ma comunque riesce a raggiungere un buon compromesso.

**Sequential backward selection:** molto simile alla precedente, solo che al posto che aggiungere una feature alla volta, parte con tutto il dataset e rimuove le feature con performance più bassa.

Esiste poi il cosiddetto metodo “**brute force**”, che prova tutte le combinazioni possibili, ma chiaramente richiede un’elevata potenza di calcolo e/o una marea di tempo per essere eseguito, sopratutto con i dataset con molte colonne (l’incremento di complessità dovrebbe essere fattoriale, quindi una tragedy)

**Imbalanced data:** un dataset viene definito imbalanced quando la presenza delle classi non è uniforme, quindi quando c’è la predominanza di un determinato output.

Questo può essere un problema dato che i nostri classifier potrebbero non avere abbastanza dati per performare al meglio la classificazione e/o potrebbero dare troppo peso alla classe maggioritaria ignorando quella minoritaria.

Per gestire questo problema si usano i samplers che hanno l’obiettivo o di creare nuove istanze delle classi minoritarie oppure di cancellarne alcune dalle classi maggioritarie.

Lista samplers

* Random over sampler
* Random under sampler
* SMOTE

**Random over sampler:** non fa altro che prendere un’istanza a caso dalla classe minoritaria e duplicarla.

**Random under sampler:** prende un’istanza a caso dalla classe maggioritaria e la cancella.

**SMOTE:** prende a caso un’instanza della classe minoritaria, ne cerca i knn, calcola i delta delle feature con uno dei knn, poi li moltiplica per un numero random tra 0 e 1 e ne somma il risultato alle feature dell’istanza iniziale creandone una nuova.

**Classifiers:** i classifiers sono la ✨linfa vitale✨ del machine learning, in sostanza sono gli algoritmi che permettono la classificazione degli elementi (o regressione).

Lista classifiers:

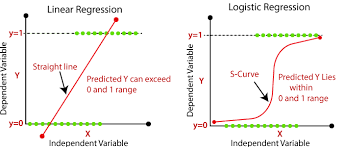
* Linear regression (unsupervised)
* Logistic regression (unsupervised)
* Perceptron (supervised)
* KNN (supervised)
* Decision tree (supervised)
* Random forest
* XGBoost
* ✨LGBTQ+✨

**Linear regression:** questa strategia ha come output un valore continuo, in sostanza ad ogni feature calcola un valore (peso) che indica la dipendenza dell’output con quella determinata feature; quindi con somma di pesi\*valore\_feature si ottiene un valore che rappresenta l’expected value per l’outcome.

Massimizza le prestazioni minimizzando la sommatoria delle distanze tra la retta e tutti gli elementi.

Essendo lineare funziona bene quando le features hanno una linear relationship.

**Logistic regression:** la logistic regression è simile alla linear regression ma al posto di un pattern lineare, crea un pattern a forma di S, quindi al posto di avere una retta per predire i valori si avrà una funzione a forma di S, che quindi separa in modo netto gli estremi dei valori.



**Perceptron:** funziona come la linear regression, però per problemi di classificazione, quindi al posto di un output continuo avrà un output binario.

Per trasformare l’output in binario viene definito un threshold che rappresenta il confine tra lo 0 e l’1.

L’errore può essere calcolato in vari modi, uno è quello dell’entropy (quindi usando i logaritmi).

PRO: hyperparameters of perceptron

| alpha | controls the learning rate of the algorithm, which specifies how fast the model should update the weights during training |
| --- | --- |
| max\_iter (epochs) | determines the maximum number of iterations the algorithm should run before it stops |
| tol | specifies the tolerance for the stopping criteria. If the change in the weights between iterations is less than the specified tolerance, the algorithm will stop iterating |
| random\_state | sets the random seed used by the algorithm for reproducibility |
| class\_weight | allows the user to assign weights to the classes in the training data, can be useful when dealing with imbalanced datasets |
| penalty | determines the type of regularization to use during training it’s a technique used to prevent overfitting by adding a penalty term to the loss function |
| eta0 | it’s the initial ​​learning rate (also called step size) to control the size of the weight updates at each iteration |

**KNN:** questo classifier cerca di prevedere la classe di appartenenza di un’istanza osservando la classe dell’istanza (o delle istanze) più simili a quella in questione.

Per calcolare la distanza tra le istanze vengono usate funzioni tipo euclidean distance, la cosine similarity ecc. ecc.

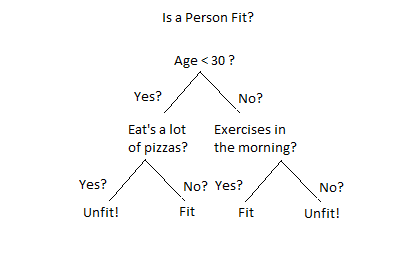
KNN può essere sensibile alla distribuzione dei dati, è quindi consigliabile normalizzarli prima.

Richiede molta memoria dato che deve sempre avere tutto il dataset, quindi può essere impegnativo con dataset grossi.

PRO: hyperparameters of KNN

| n\_neighbors | number of neighbors to use for classification, it is usually an odd number to avoid ties |
| --- | --- |
| weights | determines how to weight the contributions of the neighbors, it can be set to 'uniform' for equal weights, or 'distance' for inverse proportional weights based on the distance |
| algorithm | the algorithm to use for nearest neighbors search, it can be set to 'ball\_tree', 'kd\_tree', 'brute', or 'auto' |
| leaf\_size | sets the random seed used by the algorithm for reproducibility |
| p | allows the user to assign weights to the classes in the training data, can be useful when dealing with imbalanced datasets |
| metric | distance metric to use, it can be any of the metrics available, such as 'euclidean', 'manhattan', 'chebyshev', 'minkowski', etc. |

**Decision tree:** è sostanzialmente una ramificazione di if a forma di tree.



Ad ogni nodo del tree, si procede verso sinistra in caso la risposta all’if del nodo sia true, oppure a destra in caso sia false.

Per costruire l’albero si cercano i boundaries tra le classi sfruttando cose strane tipo l’entropia.

Il decision tree è molto sensibile al valore dei dati, infatti spesso va in overfitting, tendenzialmente la sua profondità (il numero di livelli del tree) va limitato alla radice quadrata del numero di features.

PRO: hyperparameters of decision tree

| max\_depth | the maximum depth of the tree, it stops growing when this depth is reached |
| --- | --- |
| criterion | the function used to measure the quality of a split, the two most commonly used criteria are "gini" and "entropy" |
| max\_leaf\_nodes | the maximum number of leaf nodes that can be in the tree |

**Random forest:** la random forest cerca di risolvere i problemi del decision tree, quindi sensibilità al dataset e overfitting.

Per agire sulla sensibilità crea più subset del dataset (selezionando un numero limitato di feature per ogni subset in modo casuale) e riempiendolo con righe prese a caso (con ripetizione, come in prob).

Per evitare l’overfitting crea un decision tree per ogni subset del dataset (ogni subset avrà un numero limitato di feature, diverso per ogni subset) e al momento di decidere se un elemento appartiene ad una classe o meno, performa il calcolo su tutti i decision tree e una volta ottenuti gli outcome assegna quello più ricorrente (quindi se 3 decision tree dicono true e 2 dicono false allora vince il true).

PRO: hyperparameters of random forest

| max\_depth | the maximum depth of the tree, it stops growing when this depth is reached |
| --- | --- |
| criterion | the function used to measure the quality of a split, the two most commonly used criteria are "gini" and "entropy" |
| max\_leaf\_nodes | the maximum number of leaf nodes that can be in the tree |

**Ada boost:** è un decision tree dove il numero massimo di livelli è 2 (aka stumps), avendo solo 2 livelli significa che ogni tree ha solo un unico if ed è il root node; nella stra grande maggioranza dei casi viene implementato questo algoritmo creando una random forest di stumps, a differenza della random forest però in questo caso non tutti gli stumps hanno lo stesso peso nella votazione e gli stumps sono costruiti sull’errore degli stumps precedenti (sono quindi dipendenti fra di loro).

Il peso che ha ogni stump nel voto viene calcolato con una funzione logaritmica che permette di mappare il valore “peso” degli stump in relazione alla loro precisione.

Con precisione da 0.5 a 1 (quindi migliori di una scelta random) viene assegnato un peso iniziale tra 0 e 1 (dando quindi un valore positivo) mentre con una precisione dello stump da 0 a 0.5 viene assegnato un peso iniziale tra -1 e 0 (dando quindi un peso negativo), questo perchè con una performance sotto lo 0.5 è meglio prendere l’opposto della previsione.

Dopo aver fatto questo calcolo bisogna moltiplicare il peso iniziale per una funzione esponenziale che prende in considerazione la precisione degli stump precedenti (chiaramente per il primo stump questo non vale).

**Gradient boost:** come ada boost usa dei tree con lunghezza fissa e che dipendono dall’errore dei trees precedenti (sta volta però si può decidere arbitrariamente la lunghezza dei trees).

~~A differenza dei decision trees il gradient boost non cerca di prevedere il valore della target feature, ma costruisce i trees in modo da trovare la differenza con il valore medio della colonna target.~~

**XGBoost:** come il gradient boost si basa sulla previsione dei residual (cioè la differenza con l’average value, e non il value in se); ci sono più modi di costruire un XGBoost

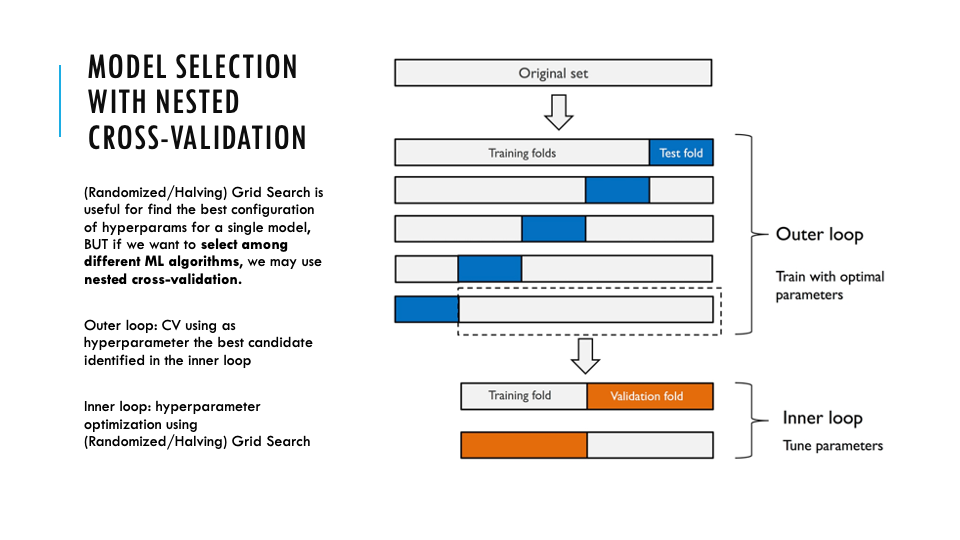
**Nested cross validation:** per trovare il modello migliore, proviamo a combinare differenti tecniche di sampling, dimensionality reduction e classifying ognuna con più di un’alternativa per ogni hyperparameter, in modo da trovare il miglior compromesso.

Questa fase si divide in due:

* **outer loop** (valuta le performance con gli hyperparameter dell’inner loop)
* **inner loop** (ricerca dei migliori hyperparameter per ogni modello)

La cross validation prevede il parametro “cv”, che rappresenta il numero di subsets nel quale verrà diviso il dataset per poter valutare le performance con un validation set.

Per ogni fold dell’outer loop (una per ogni subset), viene eseguito un inner loop che anch’esso ha un parametro “cv” per poter testare l’efficacia degli hyperparameter (l’inner loop prende in input i training subsets, quindi ad ogni fold cambierà leggermente l’input e di conseguenza anche l’output).



**Grid Search**: fa parte dell’inner loop, ed esplora tutte le possibili combinazioni di hyperparmeters per massimizzare la metrica di performance scelta (f1-score o accuracy\_score).

Trova sempre quella ottimale ma ha un tempo di computazione piú lungo della Randomized => perfetto per poche configurazioni.

**Randomized search**: a differenza della grid esplora un sottoinsieme casuale delle possibili combinazioni di hyperparameters, potrebbe quindi non trovare la combinazione ottimale infatti si usa quando abbiamo molte configurazioni.

**Cross validation:** una volta trovati i migliori parametri, viene utilizzata per valutare le prestazioni del modello ottimizzato sulla base di questi parametri.

**Learning curve:** descrive il variare delle performance (asse y) al variare della dimensione del dataset (asse x), è normale che nella parte iniziale si abbia un po’ di rumore (confusione) e tendenzialmente se training e validation convergono vuol dire che si è raggiunto un buon compromesso (infatti quando nell’f1 score cerchi il delta minore possibile è la stessa cosa di plottare e cercare il punto in cui validation e test sono più vicini).

Il fatto che debbano convergere è importante perché indica che da un determinato punto in poi le performance sono stabili.

**Validation curve:** indica la performance al variare di un hyperparameter del tuo algoritmo che gli dici tu, per esempio può essere la max depth della random forest oppure il numero di neighbours nel knn, e li non deve necessariamente convergere, ma basta che prendi il punto nel quale training e validation sono più vicini.

**Roc auc:** rappresenta il rapporto dei true positive rispetto ai false positive

**Confusion matrix:**

