SVILUPPO APPLICAZIONI E ANALISI DATI CON PYTHON

Classificazione e Regressione

Algoritmi e Tecniche con Scikit-learn

Supervised Learning: Algoritmi, Metriche & Progetti

Docente: Marino Speciale

Indice della Lezione

- Classificazione vs Regressione: Differenze chiave
- Algoritmi di Classificazione: Logistic, KNN, SVM, Naive Bayes
- Esercizio 1/2: Spam Detection (Naive Bayes + TF-IDF) / (Logistic Regression)
- Metriche di Classificazione: Precision-Recall, Classification Report
- Algoritmi di Regressione: Linear, Ridge, Lasso, Polynomial
- Esercizio 2: House Prices Prediction (Regressione)
- Metriche di Regressione: MSE, RMSE, MAE, R^2
- Esercizio 3: Wine Quality (Multi-class Classification)
- Gradient Boosting: XGBoost & LightGBM

Classificazione vs Regressione

Servono a **prevedere** qualcosa a partire dai dati, ma il tipo di “qualcosa” è diverso: categorie nel primo caso, numeri nel secondo.

🏷️ **CLASSIFICAZIONE** (è usata quando devi decidere “in che gruppo metto questo elemento?”):

- Output: Categoria discreta (0/1, Sì/No, classe A/B/C, Spam/Ham, gatto/cane, sano/malato)
- Esempi: Email spam, Diagnosi malattia, Riconoscimento immagini, Sentiment di una recensione (positivo/negativo/neutro)
- Metriche: Accuracy, Precision, Recall, F1, AUC

📊 **REGRESSIONE** (è usata quando la domanda è “quanto sarà questo valore?”):

- Output: Valore numerico continuo (prezzo, quantità, temperatura, età, probabilità)
- Esempi: Prezzo di una casa in base a metri quadri, zona, stato, ecc.; Vendite future di un prodotto (forecast) partendo da serie storiche; Premio di un’assicurazione o rischio di default di un cliente; Domanda di taxi / corse in una certa zona e ora; Temperatura
- Metriche: MSE, RMSE, MAE, R^2

Classificazione

Esempi:

- Titanic: prevedere se il passeggero sopravvive (0/1) da età, sesso, classe, tariffa.
- Spam detection: TF-IDF su testo email + Naive Bayes o Logistic Regression.
- Marketing: predire se un utente cliccherà una campagna (CTR) da device, ora, canale.
- Wine dataset di sklearn: classificare il tipo di vino da feature chimiche.
- Sanità: rischio di diabete da feature cliniche (dataset Pima Indians).

Workflow tipico in sklearn:

1. Carichi i dati, dividi in train/test.
2. Preprocessi (scaling/encoding).
3. Alleni un classificatore.
4. Valuti con accuracy, precision, recall, F1, confusion matrix, ROC/PR curve.

Algoritmi di Classificazione: Logistic Regression

La Logistic Regression è un algoritmo di *classificazione supervisionata* che serve a stimare la probabilità che un'osservazione appartenga a una certa classe (es. 0/1, sì/no, spam/non spam).

Idea di base:

- Parte da un modello lineare (come la regressione lineare) che combina le feature in una singola quantità.
- Passa questo valore attraverso una funzione sigmoide (logistic o S-shaped) che lo "schiaccia" tra 0 e 1 e lo interpreta come probabilità di classe positiva.
- Se la probabilità è ≥ soglia (tipicamente 0.5) predice classe 1, altrimenti 0 → quindi è un algoritmo di classificazione, nonostante il nome "regression".

Logistic Regression: schema minimale

```
from sklearn.linear_model import LogisticRegression

# Classificazione binaria (0/1)
model = LogisticRegression(max_iter=1000)
model.fit(X_train, y_train)

# Probabilità predette
y_proba = model.predict_proba(X_test)
# [[0.8, 0.2], [0.3, 0.7], ...]
# Prima colonna: P(classe 0)
# Seconda colonna: P(classe 1)

# Classe finale (threshold 0.5)
y_pred = model.predict(X_test)
```

Logistic Regression: Risultato

	precision	recall	f1-score	support
0	0.83	0.86	0.84	110
1	0.77	0.71	0.74	69
accuracy			0.80	179
macro avg	0.80	0.79	0.79	179
weighted avg	0.80	0.80	0.80	179

Classe 0 = passeggeri non sopravvissuti.
Classe 1 = passeggeri sopravvissuti.

Per ogni classe hai:

- precision: tra quelli che il modello ha predetto come quella classe, quanti sono davvero corretti.
  - 0: 0.83 → l'83% dei "non sopravvissuti" predetti lo sono davvero.
  - 1: 0.77 → il 77% dei "sopravvissuti" predetti lo sono davvero.
- recall: tra tutti i veri elementi di quella classe, quanti il modello li ha trovati.
  - 0: 0.86 → il modello riconosce l'86% dei veri non sopravvissuti.
  - 1: 0.71 → riconosce il 71% dei veri sopravvissuti.
- f1-score: media armonica di precision e recall (più alto → migliore compromesso).
  - 0: 0.84
  - 1: 0.74
- support: numero di esempi di quella classe nel set di test.
  - 0: 110 passeggeri
  - 1: 69 passeggeri

Algoritmi di Classificazione: K-Nearest Neighbors (KNN)

KNN è un algoritmo di apprendimento supervisionato che classifica (o fa regressione) un nuovo punto guardando le etichette dei k punti più vicini nel training set e “votando” tra loro.

Idea di base

- Ogni campione è un punto in uno spazio delle feature (età, prezzo, altezza, ecc.).
- Per un nuovo punto, KNN calcola la distanza da tutti i punti noti (tipicamente distanza euclidea) e prende i k vicini più prossimi.
- Classificazione: assegna la classe più frequente tra i k vicini (maggioranza).
- Regressione: restituisce la media (o mediana) dei valori target dei k vicini.

Un esempio classico: hai punti etichettati come “mela” o “banana” in base a peso e colore; per un nuovo frutto, guardi i 3 o 5 frutti più vicini e lo etichetti con la classe prevalente.

Ruolo del parametro k
k è il numero di vicini considerati: è un iperparametro scelto dall’utente.
k piccolo (es. 1) → modello molto flessibile ma rumoroso, rischia overfitting.
k grande → decisioni più stabili ma più “smussate”, rischia underfitting se troppo grande.
Tipicamente si sceglie k provando più valori e valutando l’accuracy (o f1) in validation/cross-validation.

K-Nearest Neighbors: schema minimale

```
from sklearn.neighbors import KNeighborsClassifier

# Classificazione basata su 'vicinanza'
# Guarda i K vicini più prossimi, vota maggioranza

knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)

# ⚠️  KNN necessita SCALING!
# Distanze dipendono dalle scale delle features

# Tuning: prova K = 3, 5, 7, 11
# K troppo piccolo → Overfitting
# K troppo grande → Underfitting
```

KNN: Risultato per k = 5

	precision	recall	f1-score	support
0	0.84	0.88	0.86	110
1	0.80	0.74	0.77	69
accuracy			0.83	179
macro avg	0.82	0.81	0.81	179
weighted avg	0.83	0.82	0.83	179

Classe 0 = passeggeri non sopravvissuti.
Classe 1 = passeggeri sopravvissuti.

KNeighborsClassifier funziona esattamente come gli altri stimatori di sklearn (fit, predict, predict_proba), solo che la predizione avviene guardando i 5 vicini più vicini nello spazio delle feature.

Per ogni classe hai:

- precision: tra quelli che il modello ha predetto come quella classe, quanti sono davvero corretti.
- recall: tra tutti i veri elementi di quella classe, quanti il modello li ha trovati.
- f1-score: media armonica di precision e recall (più alto → migliore compromesso).
- support: numero di esempi di quella classe nel set di test.

Algoritmi di Classificazione: Support Vector Machines (SVM)

SVM è una famiglia di algoritmi di apprendimento supervisionato che classificano (o fanno regressione) trovando l'iperpiano che separa le classi con il margine massimo nello spazio delle feature.

Idea intuitiva

- Immagina i punti di due classi in 2D: esistono molte rette che li separano; SVM sceglie quella che massimizza la distanza dai punti più vicini di ciascuna classe (il margine).
- I punti più vicini all'iperpiano si chiamano support vectors e sono gli unici che determinano la posizione della frontiera: se li sposti, la frontiera cambia; se rimuovi altri punti lontani, la frontiera quasi non cambia.

Questa logica di "massimo margine" rende SVM abbastanza robusto al rumore e spesso con buona capacità di generalizzazione.

Lineare vs non lineare e kernel trick

- Se i dati sono linearmente separabili, SVM cerca un iperpiano lineare (retta in 2D, piano in 3D, ecc.).
- Per dati non linearmente separabili, SVM usa il kernel trick: applica implicitamente una trasformazione in uno spazio di dimensione più alta dove i dati diventano (più) separabili linearmente, senza calcolare esplicitamente le nuove coordinate.
- Kernel comuni: lineare, polinomiale, RBF (radial basis function).

Didatticamente: è come se curvassi il piano o aggiungessi una terza dimensione, in modo che una separazione complessa in 2D diventi una separazione lineare in 3D.

Support Vector Machines: Proprietà pratiche

- È usato soprattutto per binary classification, ma esistono estensioni one-vs-rest / one-vs-one per multi-classe e la variante SVR per regressione.
- Funziona bene con feature space ad alta dimensione e quando il numero di feature è grande rispetto al numero di campioni.
- Lo iperparametro chiave è C (controlla il compromesso tra margine ampio e errori di classificazione, soft margin), più i parametri del kernel (es. gamma per RBF).

Support Vector Machines: schema minimale

```
from sklearn.svm import SVC

# Trova l'iperpiano che separa le classi
# Kernel: linear, rbf (Gaussian), poly
svm = SVC(kernel='rbf', C=1.0, gamma='scale')
svm.fit(X_train, y_train)
```

```
# Parametri chiave:
# C → Regolarizzazione (↑ C = meno errori train)
# gamma → Influenza kernel RBF (↑ gamma = più complesso)
# ⚠️  SVM necessita SCALING!
```

Variante con probabilità tipo Logistic/KNN:

In SVC le probabilità sono ottenute con una calibrazione (Platt scaling) a partire dalla distanza dall'iperpiano di separazione, quindi è un buon aggancio teorico per collegare decision function e probabilità.

```
svm_clf = SVC(kernel="rbf", C=1.0, gamma="scale", probability=True)
svm_clf.fit(X_train_scaled, y_train)

# Probabilità di appartenenza alla classe 1 (sopravvissuto) per ogni campione di test
y_proba = svm_clf.predict_proba(X_test_scaled)[:, 1]
print("Prime 10 probabilità di sopravvivenza:", y_proba[:10])
```

SVM: Risultato

	precision	recall	f1-score	support
0	0.83	0.91	0.87	110
1	0.83	0.71	0.77	69
accuracy			0.83	179
macro avg	0.83	0.81	0.82	179
weighted avg	0.83	0.83	0.83	179

Classe 0 = passeggeri non sopravvissuti.
Classe 1 = passeggeri sopravvissuti.

SVC implementa la Support Vector Classification e segue la stessa API degli altri modelli di scikit-learn (fit, predict, opcionalmente predict_proba con probability=True).

Algoritmi di Classificazione: Naive Bayes (Probabilistico)

Naive Bayes è un classificatore probabilistico supervisionato che usa il teorema di Bayes e l'assunzione "ingenua" che le feature siano indipendenti tra loro a parità di classe.

Idea probabilistica

- Per ogni classe C_k ,il modello stima la probabilità $P(C_k \mid feature)$ usando il teorema di Bayes: "probabilità della classe dato le feature = (quanto sono probabili le feature in quella classe) × (quanto è probabile la classe in generale)".
- "Naive" significa che si assume che le feature siano condizionalmente indipendenti dato la classe, cioè la probabilità congiunta è approssimata come prodotto delle probabilità di ogni singola feature: $P(x_1, \ldots, x_n \mid C_k) \approx \prod_i P(x_i \mid C_k)$.

Questo rende i calcoli molto semplici e veloci, anche con tante feature (soprattutto in NLP).

Naive Bayes: Vantaggi

Perché è utile

- È estremamente semplice, veloce e scalabile, con pochi parametri da stimare e complessità lineare nel numero di feature e classi.
- Funziona sorprendentemente bene anche se l'assunzione di indipendenza è violata, motivo per cui viene spesso usato come baseline nei problemi di classificazione.

Un esempio classico: classificatore di spam che guarda le parole presenti nell'email; per ogni parola stima $P(\textit{parola} \mid \textit{spam})$ e $P(\textit{parola} \mid \textit{ham})$,poi combina queste probabilità per decidere se l'email è spam o no.

Varianti principali

- Gaussian Naive Bayes: per feature continue, assume che $P(x_i \mid C_k)$ segua una distribuzione normale.
- Multinomial Naive Bayes: molto usato in text classification (bag-of-words, conteggi di parole).
- Bernoulli Naive Bayes: per feature binarie (presenza/assenza di parole, flag).

Naive Bayes: schema minimale

```
from sklearn.naive_bayes import MultinomialNB, GaussianNB

# MultinomialNB → Dati discreti (word counts, TF-IDF)
# GaussianNB → Dati continui

# Esempio: Spam detection
nb = MultinomialNB()
nb.fit(X_train_tfidf, y_train)  # X = word frequencies
```

Vantaggi:

- Veloce, semplice, funziona bene su testo
- Richiede pochi dati di training
- Assume indipendenza features (naive)

Naive Bayes: Risultato

	precision	recall	f1-score	support
0	0.82	0.85	0.83	110
1	0.74	0.70	0.72	69
accuracy			0.79	179
macro avg	0.78	0.77	0.77	179
weighted avg	0.79	0.79	0.79	179

Classe 0 = passeggeri non sopravvissuti.
Classe 1 = passeggeri sopravvissuti.

GaussianNB assume che, dato la classe, ogni feature numerica segua una distribuzione normale e che le feature siano indipendenti tra loro; l'API è la stessa degli altri modelli scikit-learn (fit, predict, predict_proba).

Probabilità di sopravvivenza (come negli altri modelli)
Per mostrare la natura probabilistica del modello:

```
# Probabilità di appartenenza alla classe 1 (sopravvissuto) per ogni campione di test
y_proba = gnb.predict_proba(X_test_scaled)[:, 1]

print("Prime 10 probabilità di sopravvivenza:", y_proba[:10])
```

`predict_proba` restituisce una matrice (n_samples, n_classes) con le probabilità per ciascuna classe

ESERCIZIO 1: Spam Detection (Parte 1)

📌 Obiettivo: Classificare email spam/ham
📊 Dataset: Email text → TF-IDF features

```
# Step 1: Carica dati
emails = ['Buy now!', 'Meeting tomorrow', 'Win prize!', ...]
labels = [1, 0, 1, ...]  # 1=spam, 0=ham

# Step 2: TF-IDF Vectorization
from sklearn.feature_extraction.text import TfidfVectorizer

vectorizer = TfidfVectorizer(max_features=1000)
X = vectorizer.fit_transform(emails)  # Sparse matrix
```

ESERCIZIO 1: Spam Detection (Parte 2)

```
# Step 3: Train/Test Split
X_train, X_test, y_train, y_test = train_test_split(
    X, labels, test_size=0.2, random_state=42)

# Step 4: Naive Bayes
nb = MultinomialNB()
nb.fit(X_train, y_train)

# Step 5: Valuta
y_pred = nb.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy:.3f}')
```

Prova anche con Logistic Regression (Esercizio 2)

Metriche di Classificazione: Classification Report

Il classification report è una tabella che riassume le principali metriche di valutazione di un modello di classificazione: *precision*, *recall*, *f1-score* e *support* per ogni classe, più alcune medie e l'accuracy complessiva. È molto più informativo della sola accuracy, soprattutto quando le classi sono sbilanciate o interessa capire su quale classe il modello sbaglia di più.

```
from sklearn.metrics import classification_report

print(classification_report(y_test, y_pred))

# Output:
#               precision  recall  f1-score  support
#
#     Ham(0)       0.95      0.98      0.96      100
#    Spam(1)       0.97      0.92      0.95       80
#
# accuracy                           0.96      180
```

Classification Report: cosa contiene

Per ogni classe (es. 0 = non spam, 1 = spam) trovi:

- precision: tra tutti i campioni che il modello ha predetto come quella classe, quanti sono davvero corretti (pochi falsi positivi → precision alta).
- recall: tra tutti i campioni che appartengono davvero a quella classe, quanti il modello li ha trovati (pochi falsi negativi → recall alta).
- f1-score: media armonica di precision e recall, bilancia le due (vale 1 se precision = recall = 1, 0 se entrambe scarse).
- support: numero di esempi reali di quella classe presenti nel set di test.

Sotto le righe delle singole classi vedi:

- accuracy: percentuale di predizioni corrette sul totale dei campioni.
- macro avg: media aritmetica di precision/recall/f1 sulle classi, tutte pesate allo stesso modo (utile se le classi sono sbilanciate).
- weighted avg: media pesata per il support di ciascuna classe (tiene conto di quante istanze ha ogni classe).

Metriche di Classificazione: Precision-Recall Curve

La Precision-Recall Curve è un grafico che mostra come cambiano precision e recall della classe positiva al variare della soglia usata per trasformare le probabilità in predizioni 0/1.

Cosa rappresenta

- Sull’asse x hai la recall (sensibilità).
- Sull’asse y hai la precision.
- Ogni punto della curva corrisponde a una soglia diversa applicata alle probabilità (o ai punteggi) del modello: soglia bassa → più positivi predetti (recall alta, precision tipicamente più bassa), soglia alta → meno positivi (recall bassa, precision più alta).

È molto usata quando le classi sono sbilanciate (es. fraud detection, spam detection), perché in questi casi la ROC curve tende a essere troppo ottimista, mentre la Precision-Recall mette a fuoco il trade-off sui positivi.

Precision-Recall Curve: Vantaggi e schema minimale

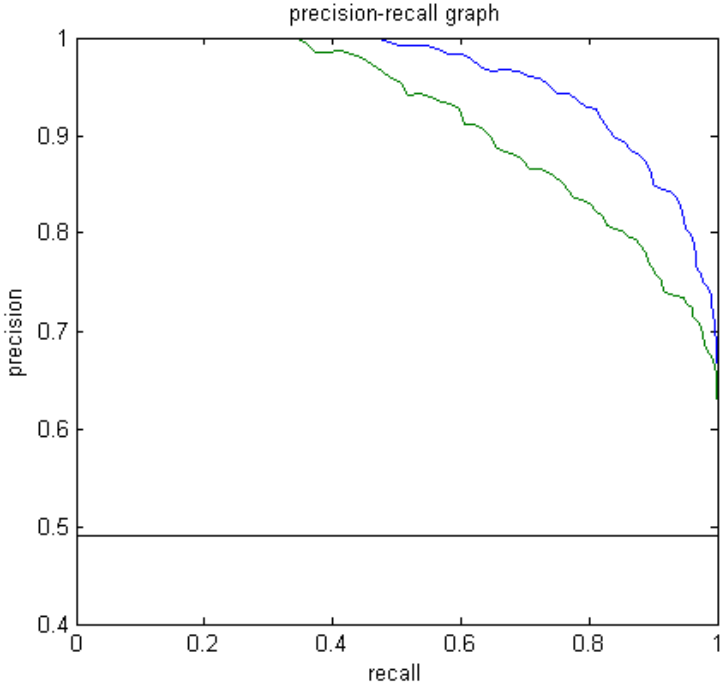
- Ti aiuta a scegliere una soglia operativa in base alle priorità del problema: ad esempio, vuoi più recall (trovare quasi tutti gli spam) o più precision (pochi falsi allarmi)?
- L'area sotto la curva (Average Precision / AUPRC) è un riassunto numerico della qualità del modello per la classe positiva, analogo all'AUC-ROC ma più informativo su dataset fortemente sbilanciati.

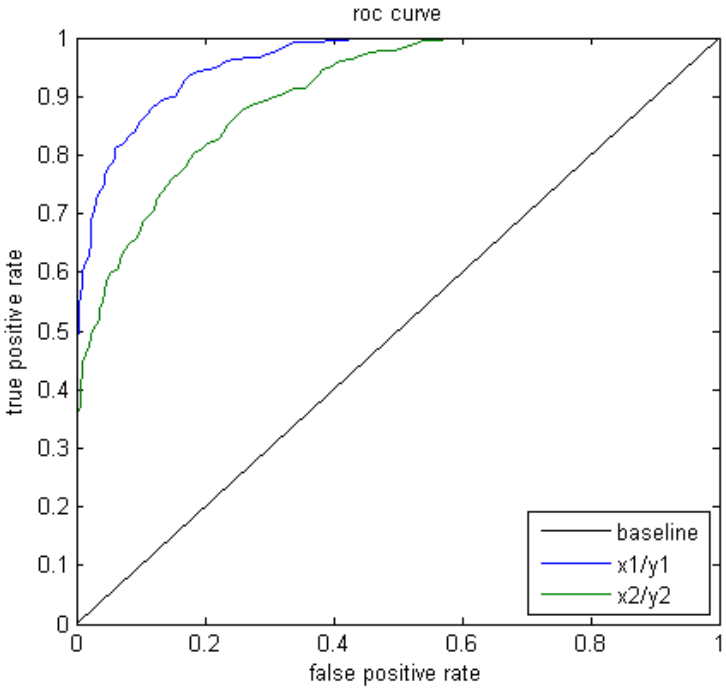
```
from sklearn.metrics import precision_recall_curve
import matplotlib.pyplot as plt

# Utile per classi sbilanciate
precision, recall, thresholds = precision_recall_curve(y_test, y_proba[:, 1])

plt.plot(recall, precision)
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curve')
plt.show()
```

dove y_scores sono le probabilità o i punteggi di appartenenza alla classe positiva (es. `predict_proba()[:, 1]` o `decision_function()`).


precision-recall graph
precision
recall
1
0.9
0.8
0.7
0.6
0.5
0.4
0
0.2
0.4
0.6
0.8
1


roc curve
true positive rate
false positive rate
baseline
x1/y1
x2/y2
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0.2
0.4
0.6
0.8
1

Regressione

Esempi:

- House prices: dataset sintetico o reale, predizione del prezzo casa.
- Calorie bruciate in base ad età, peso, durata e intensità dell'esercizio.
- Demand forecasting: numero di corse taxi o prodotti venduti domani.

Workflow tipico:

1. Carichi i dati e fai EDA veloce (scatter, heatmap).
2. Alleni Linear/Ridge/Lasso o un modello tree-based.
3. Valuti con MSE, RMSE, MAE, R^2.
4. Fai un residual plot per mostrare limiti del modello.

Algoritmi di regressione: Linear Regression

La linear regression è un modello di apprendimento supervisionato che stima una relazione lineare tra una variabile target continua y e una o più feature x. Spesso il primo modello da introdurre perché collega direttamente concetti di retta, errore e ottimizzazione con la pratica del machine learning.

Idea di base

- Si assume che il target sia approssimabile da una retta (o iperpiano): $y \approx \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p + \varepsilon$ dove β_i sono i coefficienti da imparare e ε è l'errore.
- L'algoritmo sceglie i coefficienti che minimizzano la somma dei quadrati degli errori tra valori osservati e predetti (metodo dei minimi quadrati, OLS).

Esempio classico: prevedere il prezzo di una casa in funzione di metri quadri, numero di stanze, zona, ecc.; l'uscita è un numero reale, non una classe.

Cosa produce e come si usa

- In fase di training il modello "impara" i coefficienti β, che si interpretano come "quanto cambia y se x_j aumenta di 1 unità, tenendo fisse le altre variabili".
- Una volta addestrato, puoi inserire nuove feature x nell'equazione lineare e ottenere una predizione numerica del target.

Linear Regression: schema minimale

```
from sklearn.linear_model import LinearRegression

model = LinearRegression()
model.fit(X_train, y_train)

# Coefficienti (pesi features)
print(f'Coefficienti: {model.coef_}')
print(f'Intercetta: {model.intercept_}')

# Predici
y_pred = model.predict(X_test)

# Valutazione (MSE e R^2)
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
```

- LinearRegression implementa la regressione lineare OLS e si usa con fit/predict come gli altri estimator di scikit-learn.
- mean_squared_error misura quanto, in media, le predizioni si discostano dal valore reale (quadraticamente), mentre r2_score indica quanta parte della variabilità del target è spiegata dal modello.

```
Coefficienti del modello:
  feature        coef
0     age   37.904021
1     sex -241.964362
2     bmi  542.428759
3      bp  347.703844
4      s1 -931.488846
5      s2  518.062277
6      s3  163.419983
7      s4  275.317902
8      s5  736.198859
9      s6   48.670657
```

Un esempio classico è usare la linear regression sul dataset diabetes di scikit-learn per predire una variabile continua (progressione della malattia).

Algoritmi di regressione: Ridge & Lasso Regression (Regolarizzazione)

Ridge e Lasso sono varianti della regressione lineare con regolarizzazione, usate per ridurre l'overfitting "punendo" i coefficienti troppo grandi nel modello.

Idea di regolarizzazione

- Si parte dalla regressione lineare classica e si aggiunge un termine di penalità alla funzione di perdita, controllato da un parametro λ (o α in scikit-learn).
- Questo termine spinge i coefficienti verso zero, riducendo la complessità del modello: si accetta un po' più di bias in cambio di meno varianza e migliore generalizzazione su dati nuovi.

Ridge & Lasso Regression: Differenza

Ridge Regression (L2)

- Aggiunge una penalità L2: somma dei quadrati dei coefficienti, $\lambda \sum \beta_j^2$.
- Effetto: i coefficienti vengono rimpiccioliti ma raramente esattamente a zero; tutte le feature restano nel modello, solo con peso ridotto.
- È utile quando hai molte feature correlate (multicollinearità) e vuoi stabilizzare i coefficienti senza fare selezione automatica di variabili.

Lasso Regression (L1)

- Aggiunge una penalità L1: somma dei valori assoluti dei coefficienti, $\lambda \sum | \beta_j |$.
- Effetto: oltre a ridurre la magnitudine, può portare alcuni coefficienti esattamente a zero, facendo di fatto feature selection automatica.
- È ideale quando sospetti che solo un sottoinsieme di feature sia davvero importante e vuoi un modello più sparso e interpretabile.

In pratica, Ridge “liscia” il modello mantenendo tutte le variabili, mentre Lasso “taglia” quelle meno utili; entrambi aiutano a tenere sotto controllo la complessità e a prevenire l’overfitting.

Ridge & Lasso Regression: schema minimale

```
from sklearn.linear_model import Ridge, Lasso

# Ridge (L2): penalizza coefficienti grandi
ridge = Ridge(alpha=1.0)  # ↑ alpha = ↑ regolarizzazione
ridge.fit(X_train, y_train)

# Lasso (L1): può azzerare coefficienti (feature selection)
lasso = Lasso(alpha=0.1)
lasso.fit(X_train, y_train)
```

Dall'esempio sul diabetes, con questi parametri, Lasso si adatta un po' meglio ai dati (senza sembrare in overfitting). Questo illustra bene la differenza: Ridge "tiene tutto ma riduce", Lasso "sceglie" un sottoinsieme di variabili. La cross-validation ha provato i valori di alpha inseriti e ha trovato che 0.1 è quello che dà le migliori prestazioni medie sul validation set per entrambi i modelli.
Interpretazione: con questo dataset, una regolarizzazione moderata (non troppo forte, non nulla) è il compromesso migliore tra bias e varianza, sia per Ridge sia per Lasso.

Ridge MSE: 3077.41593882723
Ridge R2: 0.419152926359865 45
Lasso MSE: 2798.1934851697188
Lasso R2: 0.4718547867276227

Coefficienti Ridge vs Lasso:

	feature	ridge_coef	lasso_coef
0	age	45.367377	0.000000
1	sex	-76.666086	-152.664779
2	bmi	291.338832	552.697775
3	bp	198.995817	303.365158
4	s1	-0.530310	-81.365007
5	s2	-28.577050	-0.000000
6	s3	-144.511905	-229.255776
7	s4	119.260066	0.000000
8	s5	230.221608	447.919525
9	s6	112.149830	29.642617

Miglior alpha Ridge: 0.1

Miglior alpha Lasso: 0.1

Algoritmi di regressione: Polynomial Regression

La Polynomial Regression è una regressione lineare applicata a feature trasformate con potenze (x, x^2, x^3, …), usata per modellare relazioni non lineari tra input e target.

Idea di base

- Invece di usare solo x, il modello usa $x, x^2, x^3, \dots, x^d$,e la funzione diventa: $y \approx \beta_0 + \beta_1 x + \beta_2 x^2 + \cdots + \beta_d x^d$.
- La curva risultante può seguire andamenti curvi dei dati, ma il modello resta "lineare nei parametri" (si stima sempre una regressione lineare, solo con più feature).

Esempio classico: dati che crescono prima lentamente, poi più velocemente e poi si appiattiscono; una retta sottostima/soverstima, mentre una parabola (grado 2) o un polinomio di grado 3 li segue molto meglio.

Quando usarla e attenzioni

- Utile quando il legame tra variabile indipendente e target non è ben rappresentato da una linea retta, ma mostra curvatura.
- Se il grado del polinomio è troppo alto, rischi overfitting: la curva passa quasi esattamente per tutti i punti di training ma generalizza male; per questo si sceglie il grado con validation/cross-validation.

In scikit-learn, si usa tipicamente PolynomialFeatures per generare le potenze e poi LinearRegression o un altro modello lineare sulle feature estese.

Polynomial Regression: schema minimale

```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression

# Trasforma features: x → x, x²
poly = PolynomialFeatures(degree=2)    # crea tutte le combinazioni fino al grado 2: per ogni
                                       # coppia di feature ottieni termini tipo x_i^2 , x_i x_j
X_poly = poly.fit_transform(X_train)

# Poi usa Linear Regression normale
model = LinearRegression()        # lavora su queste feature estese, quindi il modello può
                                  # rappresentare relazioni non lineari pur restando lineare
                                  # nei coefficienti.
model.fit(X_poly, y_train)

# Test: trasforma anche X_test
X_test_poly = poly.transform(X_test)
y_pred = model.predict(X_test_poly)
```

Metriche di Regressione: MSE, RMSE, MAE, R^2

Le metriche di regressione servono a misurare quanto bene un modello predice valori numerici rispetto ai valori reali. Le più usate sono MSE, RMSE, MAE e R^2.

```
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
mse = mean_squared_error(y_test, y_pred)
rmse = mse ** 0.5
```

MSE – Mean Squared Error

- È la media dei quadrati degli errori: MSE $= \frac{1}{n} \sum (y_{true} - y_{pred})^2$.
- Penalizza molto gli errori grandi (perché li eleva al quadrato) ed è utile per confrontare modelli: più piccolo è, meglio è.

RMSE – Root Mean Squared Error

- È la radice quadrata dell'MSE: RMSE $= \sqrt{MSE}$.
- Ha la stessa unità di misura del target (es. euro, kg, ecc.), quindi è più interpretabile: "errore medio" tipico in unità del problema.

Metriche di Regressione: MSE, RMSE, MAE, R^2

```
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
mae = mean_absolute_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
```

MAE – Mean Absolute Error

- È la media degli errori assoluti: MAE $= \frac{1}{n}\sum |y_{true} - y_{pred}|$.
- È meno sensibile agli outlier rispetto a MSE/RMSE, perché l'errore non viene elevato al quadrato; rappresenta l'errore medio "in valore assoluto".

R^2 – Coefficiente di determinazione

- Misura quanta parte della variabilità del target è spiegata dal modello rispetto a un modello banale che predice sempre la media.
- Valori tipici:
  - 1 → modello perfetto (nessun errore); 0 → non spiega nulla meglio della media.
  - Negativo → peggio di una previsione costante uguale alla media.

In pratica: MSE/RMSE/MAE ti dicono "di quanto sbaglio in media", R^2 ti dice "quanto bene il modello spiega il fenomeno complessivamente".

Interpretazione R^2 Score

R^2 = 1 - (SS_res / SS_tot)

R^2 = 1.0 → Modello perfetto ✅
R^2 = 0.9 → Ottimo
R^2 = 0.7 → Buono
R^2 = 0.5 → Accettabile
R^2 < 0.3 → Scarso ❌
R^2 < 0 → Peggio della media!

R^2 indica quanta variabilità di y è spiegata dal modello
Esempio: R^2 = 0.85 → 85% varianza spiegata

ESERCIZIO 3: House Prices Prediction (Parte 1)

```
📌 Obiettivo: Predire prezzo casa
📊 Features: Area (sqft), # Camere, Età casa, Distanza centro

# Step 1: Carica dati (esempio sintetico)
import pandas as pd
df = pd.DataFrame({
    'Area': [1500, 2000, 1200, 1800, 2500],
    'Bedrooms': [3, 4, 2, 3, 5],
    'Age': [10, 5, 20, 8, 2],
    'Price': [300000, 400000, 250000, 350000, 500000]
})
```

ESERCIZIO 3: House Prices Prediction (Parte 2)

```
# Step 2: Separa X, y
X = df[['Area', 'Bedrooms', 'Age', 'Distance']]
y = df['Price']

# Step 3: Train/Test Split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.4, random_state=42)

# Step 4: Linear Regression
model = LinearRegression()
model.fit(X_train, y_train)

# Step 5: Valuta con MSE, R²
y_pred = model.predict(X_test)
```

```
   Area  Bedrooms  Age  Distance   Price
0  1500         3   10         5  300000
1  2000         4    5         3  400000
2  1200         2   20         8  250000
3  1800         3    8         4  350000
4  2500         5    2         2  500000

Valori reali: [400000, 500000]
Valori predetti: [np.float64(383333.19645778975), np.float64(466666.2238340256)]

MSE : 694461486.90
RMSE: 26352.64
MAE : 25000.29
R²  : 0.722
```

Le metriche riassumono quanto il modello si avvicina ai prezzi reali:

- MSE penalizza maggiormente gli errori grandi;
- RMSE ha la stessa unità dei prezzi (euro);
- MAE è l'errore assoluto medio;
- R^2 indica la quota di variabilità del prezzo spiegata dal modello (1 = perfetto).

Algoritmi di regressione: Support Vector Regression (SVR)

Support Vector Regression (SVR) è la versione per regressione delle Support Vector Machines: predice valori continui cercando una funzione che stia "più piatta possibile" e tolleri piccoli errori entro un certo margine.

Idea intuitiva

- SVR cerca una funzione (lineare o non lineare) tale che la maggior parte dei punti cada dentro un tubo di ampiezza 2ε attorno alla curva.
- Gli errori dentro questo tubo non vengono penalizzati (loss = 0); si penalizzano solo i punti che stanno fuori oltre ε → *epsilon-insensitive loss*.
- Come nelle SVM di classificazione, solo alcuni punti "di bordo" determinano il modello: sono i support vectors, il che rende SVR relativamente efficiente e robusto.

In pratica, invece di minimizzare la somma dei quadrati degli errori come nella regressione lineare, SVR minimizza una combinazione tra:

- la piattezza della funzione (norma dei pesi piccola)
- e le deviazioni maggiori di ε, controllate dal parametro *C*.

Lineare vs non lineare

- Con kernel lineare, SVR è una regressione quasi lineare ma con margine ε.
- Con kernel RBF, polinomiale, sigmoid, usa il kernel trick per modellare relazioni non lineari nello spazio delle feature.

Support Vector Regression (SVR): schema minimale

```
from sklearn.svm import SVR
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline

# SVM per regressione
regr = make_pipeline(StandardScaler(), SVR(kernel="rbf", C=1.0, epsilon=0.2))
regr.fit(X_train, y_train)
y_pred = regr.predict(X_test)
```

epsilon → tolleranza errore, è la larghezza del “tubo” in cui gli errori non vengono penalizzati
C → regolarizzazione, controlla quanto penalizzi gli errori fuori dal margine epsilon

⚠️ SVR necessita SCALING perché SVR è sensibile alla scala delle feature!

Si usa quando:
- Relazioni non lineari
- Dati con outlier

ESERCIZIO 3: Wine Quality (Multi-class)

```
📌 Obiettivo: Classificare qualità vino (3-9)
📊 Features: acidità, zucchero, alcol, pH, etc

# Step 1: Carica dataset
from sklearn.datasets import load_wine
wine = load_wine()
X, y = wine.data, wine.target

# Step 2: Train/Test Split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, stratify=y, random_state=42)

# Step 3: Scaling
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

ESERCIZIO 3: Wine Quality (Parte 2)

```python
# Step 4: SVM Multi-class
svm = SVC(kernel='rbf', C=10, gamma='scale')
svm.fit(X_train_scaled, y_train)


# Step 5: Valuta
y_pred = svm.predict(X_test_scaled)
accuracy = accuracy_score(y_test, y_pred)
print(classification_report(y_test, y_pred))
```

```
Feature names: ['alcohol', 'malic_acid', 'ash', 'alcalinity_of_ash', 'magnesium', 'total_phenols', 'flavanoids', 'nonflavanoid_phenols', 'proanthocyanins', 'color_intensity', 'hue', 'od280/od315_of_diluted_wines', 'proline']
Classi: ['class_0' 'class_1' 'class_2']
   alcohol  malic_acid   ash  alcalinity_of_ash  magnesium  total_phenols  flavanoids  nonflavanoid_phenols  proanthocyanins  color_intensity   hue  od280/od315_of_diluted_wines  proline  target
0    14.23        1.71  2.43               15.6      127.0           2.80        3.06                  0.28             2.29             5.64  1.04                          3.92   1065.0       0
1    13.20        1.78  2.14               11.2      100.0           2.65        2.76                  0.26             1.28             4.38  1.05                          3.40   1050.0       0
2    13.16        2.36  2.67               18.6      101.0           2.80        3.24                  0.30             2.81             5.68  1.03                          3.17   1185.0       0
3    14.37        1.95  2.50               16.8      113.0           3.85        3.49                  0.24             2.18             7.80  0.86                          3.45   1480.0       0
4    13.24        2.59  2.87               21.0      118.0           2.80        2.69                  0.39             1.82             4.32  1.04                          2.93    735.0       0

Accuracy SVM (wine): 0.944

Classification report:
              precision    recall  f1-score   support

     class_0       1.00      1.00      1.00        12
     class_1       0.88      1.00      0.93        14
     class_2       1.00      0.80      0.89        10

    accuracy                           0.94        36
   macro avg       0.96      0.93      0.94        36
weighted avg       0.95      0.94      0.94        36
```

Gradient Boosting: XGBoost

XGBoost è una libreria/algoritmo di gradient boosting su alberi decisionali particolarmente ottimizzata e potente; il nome significa eXtreme Gradient Boosting.

Idea di base (Gradient Boosting)

- Usa tanti alberi decisionali deboli (poco profondi) costruiti in sequenza.
- Ogni nuovo albero viene addestrato per correggere gli errori (residui) del modello costruito fino a quel momento: si muove lungo il gradiente della funzione di perdita, da cui "gradient boosting".
- Le predizioni finali sono la somma (o media pesata) dei contributi di tutti gli alberi, spesso con un learning rate che riduce l'impatto di ciascun albero per evitare overfitting.

Cosa rende XGBoost "extreme"

Rispetto a un gradient boosting "classico", XGBoost introduce varie ottimizzazioni:

- Implementazione molto veloce e scalabile (CPU/GPU, distribuito, out-of-core).
- Forte regolarizzazione sui alberi (controllo della complessità con penalità su foglie, profondità, numero di nodi) per ridurre overfitting.
- Gestione efficiente di dati mancanti e feature sparse, con ricerca ottimizzata degli split (weighted quantile sketch, split gain, ecc.).
- Supporto per diversi obiettivi: regressione, classificazione binaria/multi-classe, ranking, ecc.

Gradient Boosting: XGBoost

```
# pip install xgboost
from xgboost import XGBClassifier, XGBRegressor

# Classificazione
xgb_clf = XGBClassifier(n_estimators=100, max_depth=5)
xgb_clf.fit(X_train, y_train)

# Regressione
xgb_reg = XGBRegressor(n_estimators=100, learning_rate=0.1)
xgb_reg.fit(X_train, y_train)
```

```
XGBoost MSE : 3166.55
XGBoost RMSE: 56.27
XGBoost R²  : 0.402
```

Gradient Boosting: LightGBM

Light Gradient Boosting Machine è un framework di gradient boosting su alberi decisionali progettato per essere molto veloce e poco costoso in memoria. È spesso la scelta ideale quando hai dataset grandi e molte feature, e ti serve training rapido con buone prestazioni, soprattutto su problemi di competizione o produzione su dati tabellari.

Idea di base

- Come XGBoost, è un metodo di boosted trees: costruisce tanti alberi deboli in sequenza, dove ogni albero successivo corregge gli errori dei precedenti seguendo il gradiente della funzione di perdita.
- È pensato per compiti di classificazione, regressione e ranking su dati tabellari/strutturati.

Cosa lo distingue dagli altri boosting (es. XGBoost)

- Usa una strategia di crescita leaf-wise (best-first): a ogni passo sceglie la foglia che porta alla maggiore riduzione della loss e la divide, invece di crescere l'albero livello per livello. Questo permette di raggiungere la stessa accuratezza con meno nodi e quindi più velocemente, ma richiede un po' più attenzione al rischio di overfitting.
- Introduce tecniche specifiche per velocità ed efficienza:
  - GOSS (Gradient-based One-Side Sampling): tiene tutti i campioni con gradiente grande (difficili) e campiona solo parte di quelli con gradiente piccolo, riducendo il costo di calcolo.
  - EFB (Exclusive Feature Bundling): raggruppa feature mutuamente esclusive per ridurre la dimensionalità effettiva e la memoria, utile con molte variabili sparse.

Gradient Boosting: LightGBM

```
# pip install lightgbm
from lightgbm import LGBMClassifier, LGBMRegressor

# Classificazione
lgbm_clf = LGBMClassifier(n_estimators=100, max_depth=5)
lgbm_clf.fit(X_train, y_train)

# Regressione
lgbm_reg = LGBMRegressor(n_estimators=100, learning_rate=0.1)
lgbm_reg.fit(X_train, y_train)
```

```
LightGBM MSE : 3250.82
LightGBM RMSE: 57.02
LightGBM R²  : 0.386

Importanza feature LightGBM:
bmi    909
s5     875
bp     800
s2     783
s1     697
age    641
s6     629
s3     618
s4     229
sex    113
dtype: int32
```

Quando usare quale Algoritmo (Classificazione)

LOGISTIC REGRESSION → Baseline, interpretabile, veloce

KNN → Pochi dati, non parametrico, decision boundary complessa

SVM (RBF) → Non lineare, dataset medio, pochi outlier

NAIVE BAYES → Testo, dati categorici, veloce

RANDOM FOREST → Robusto, feature importance, no scaling

GRADIENT BOOSTING → Massima accuracy, competizioni

⚠️ Scaling necessario: KNN, SVM, Logistic Regression

✅ No scaling: Tree-based (RF, XGBoost, LightGBM)

Quando usare quale Algoritmo (Regressione)

LINEAR REGRESSION → Baseline, interpretabile, relazioni lineari

RIDGE → Multicollinearità tra features

LASSO → Feature selection automatica, molte features

POLYNOMIAL → Relazioni non lineari (curvilinee)

SVR → Non lineare, outlier

RANDOM FOREST → Robusto, no scaling, feature importance

GRADIENT BOOSTING → Massima accuracy, competizioni

Metriche chiave: R^2, RMSE (stessa unità y), MAE (outlier)

Best Practices Classificazione & Regressione

- ✅ SEMPRE train/test split con stratify (classificazione)
- ✅ Scaling per: KNN, SVM, Logistic, Ridge/Lasso
- ✅ Cross-validation per scegliere modello
- ✅ Classificazione: Usa classification_report, ROC, PR curve
- ✅ Regressione: Usa MSE, R^2, visualizza residui
- ✅ Classi sbilanciate: Precision-Recall curve, SMOTE
- ✅ Feature Engineering: crea features rilevanti
- ✅ Hyperparameter Tuning: GridSearchCV, RandomizedSearchCV
- ✅ Ensemble: combina modelli (VotingClassifier/Regressor)

Residual Plot (Diagnostica Regressione)

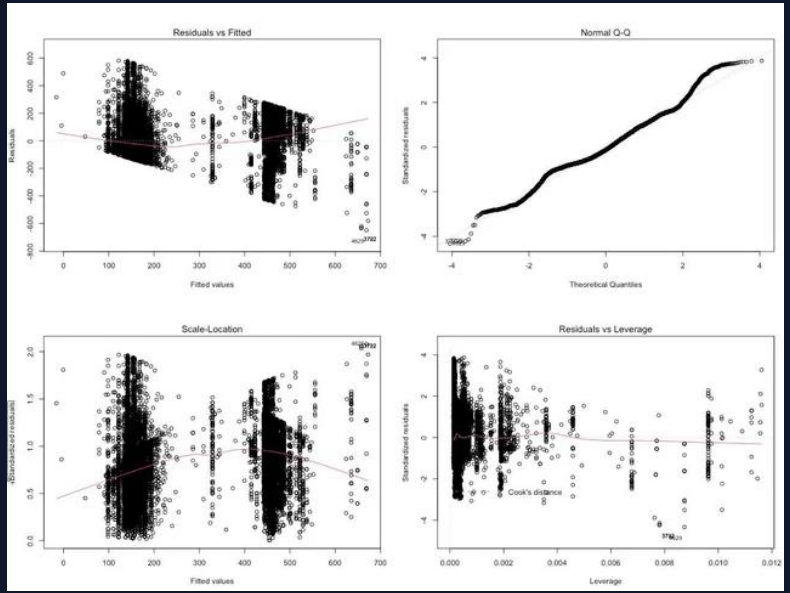
Un residual plot è un grafico che mostra gli errori di regressione (residui) sull'asse y in funzione dei valori previsti (o di una feature) sull'asse x, ed è uno strumento di diagnostica per valutare se il modello di regressione è adeguato.

Cosa rappresenta

- Un residuo è $y_{vero} - y_{predetto}$:quanto il modello sbaglia per ciascun punto (positivo se sottostima, negativo se sovrastima).
- Nel residual plot, ogni punto ha coordinate (valore previsto, residuo) oppure (feature, residuo).

Se il modello lineare è ragionevole, i residui dovrebbero:

- essere sparsi casualmente sopra e sotto lo zero;
- formare una "banda orizzontale" senza pattern evidenti;
- non mostrare pochi punti molto lontani (outlier influenti).



Residual Plot (Diagnostica Regressione)

Cosa ci puoi diagnosticare
Guardando il residual plot puoi individuare:

- Non linearità: pattern a U, a S o curve → il modello lineare non cattura bene la relazione, servono termini polinomiali o un modello non lineare.
- Heteroscedasticity: forma “a cono” (varianza dei residui che cresce o decresce con il fitted) → violazione dell’ipotesi di varianza costante.
- Outlier / punti influenti: residui molto lontani dalla banda centrale → possibili errori di misura o casi speciali da analizzare.

In sintesi, il residual plot è un check visivo: se i punti sembrano rumore bianco attorno allo zero, il modello di regressione è coerente con le sue assunzioni; se emergono pattern, c’è qualcosa da rivedere nel modello o nelle trasformazioni delle variabili.

Residual Plot (Diagnostica Regressione)

```
import matplotlib.pyplot as plt

# Calcola residui
residuals = y_test - y_pred

# Plot residui vs predizioni
plt.scatter(y_pred, residuals)
plt.axhline(y=0, color='r', linestyle='--')
plt.xlabel('Predicted Values')
plt.ylabel('Residuals')
plt.title('Residual Plot')
plt.show()

# Residui casuali → Modello OK ✅
# Pattern nei residui → Modello non cattura relazione ❌
```

Prossimi Passi

Progetti Avanzati & Deep Learning

Progetto 1: Sentiment Analysis | Progetto 2: Time Series Forecasting