SVILUPPO APPLICAZIONI E ANALISI DATI CON PYTHON

Scikit-learn

Machine Learning Open Source in Python

Esempi Pratici & Esercizi

Docente: Marino Speciale

Indice della Lezione

- Introduzione: Architettura e API
- Preparazione dati: Dataset, Train-Test Split, Preprocessing
- Workflow Scikit-learn: Estimator, Fit, Predict, Score
- Esercizio 1: Classificazione Iris (Decision Tree → Random Forest)
- Pipeline e FeatureUnion: Automatizzare il workflow
- Esercizio 2: Titanic - Preprocessing + Classificazione
- Model Evaluation avanzata: ROC curves, Confusion Matrix, Metriche
- Esercizio 3: Feature Engineering + Hyperparameter Tuning

Cos'è Scikit-learn?

- Libreria Python per Machine Learning
- Built on: NumPy, SciPy, Matplotlib, Pandas
- API coerente: Tutti i modelli seguono lo stesso pattern
- Modelli predefiniti: 100+ algoritmi ML pronti all'uso
- Strumenti: preprocessing, feature selection, model selection
- Visualizzazione: confusion matrix, ROC curves, feature importance
- Installazione: `pip install scikit-learn`

Architettura e API coerente

```
# Importa i modelli
from sklearn import datasets, model_selection, preprocessing
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, confusion_matrix

# Tutti i modelli seguono lo STESSO pattern:
model = DecisionTreeClassifier()     # 1. Istanziare
model.fit(X_train, y_train)          # 2. Allenare
y_pred = model.predict(X_test)       # 3. Predire
score = model.score(X_test, y_test)  # 4. Valutare
```

Questa uniformità rende facile cambiare modello (es. da DecisionTreeClassifier a RandomForestClassifier) senza riscrivere tutto il codice.

Dataset Built-in di Scikit-learn

Scikit-learn fornisce già alcuni dataset di esempio pronti da usare, senza dover cercare file esterni. Sono perfetti per fare esercizi veloci su classificazione senza doversi preoccupare di reperire e “pulire” i dati.

```
# Importa il modulo che contiene i dataset di esempio.
from sklearn import datasets

# Iris - Classificazione (150 fiori, 3 specie)
iris = datasets.load_iris()
X_iris, y_iris = iris.data, iris.target     # le feature (misure) e le etichette (specie)

# Wine - Classificazione (178 vini, 3 classi)
wine = datasets.load_wine()

# Digits - Riconoscimento numeri (1797 immagini 8x8 di cifre scritte a mano)
digits = datasets.load_digits()
```

Train-Test Split con Scikit-learn

```
from sklearn.model_selection import train_test_split

# X = feature (input), y = etichette (target)
X_train, X_test, y_train, y_test = train_test_split(
    test_size=0.2,          # 80% train, 20% test
    random_state=42,        # reproducibilità, stessa divisione ogni volta
    stratify=y_iris         # mantiene le proporzioni delle classi
                            # se è classificazione, mantiene nel train/test la stessa
                            # distribuzione di classi (es. 30% classe 1, 70% classe 0)
)

# Stampa quanti campioni sono finiti nel train e quanti nel test
print(f'Train: {X_train.shape[0]}, Test: {X_test.shape[0]}')
```

Train: 120, Test: 30

Spiegazione: Train-Test Split con Scikit-learn

Il risultato "Train: 120, Test: 30" significa che il dataset originale di 150 campioni (fiori) è stato diviso in due parti:

- **120** campioni sono stati assegnati al set di addestramento (train), che serve per "insegnare" al modello
- **30** campioni sono stati messi nel set di test, usato per valutare la capacità del modello di generalizzare su dati nuovi.

In pratica, il modello impara dai 120 esempi e poi viene testato su 30 esempi mai visti prima per verificare quanto bene riesce a classificare correttamente le specie di iris.
Impostando `test_size=0.3`, il 30% dei dati viene usato per il test e il 70% per l'addestramento. Questo significa meno dati per addestrare il modello, ma una valutazione più robusta con più dati di test.
Cambiare `random_state` da 42 a 50 modifica la divisione casuale dei dati in train e test. Anche se la proporzione rimane la stessa, i campioni specifici assegnati a train e test cambiano. Questo può influenzare i risultati del modello se si esegue più volte la divisione con random_state diverso, perché il modello vede dati diversi in fase di addestramento e test.

Preprocessing: StandardScaler

Vediamo come scalare le feature con StandardScaler in modo corretto.

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()  # trasforma ogni feature per avere media 0 e deviazione standard 1
X_train_scaled = scaler.fit_transform(X_train)   # fit calcola media e deviazione standard
sul train; transform applica la formula di scaling ai dati di train.

X_test_scaled = scaler.transform(X_test)    # applica lo stesso scaling ai dati di test,
usando le statistiche calcolate sul train

IMPORTANTE: fit SOLO su train! Se fai fit su test → Data leakage. Significa che se calcoli
media/deviazione standard anche sui dati di test, il modello "vede" informazioni del test in
fase di training, e la valutazione non è più onesta.

# Media e std memorizzati:
print(f'Mean: {scaler.mean_}')
print(f'Std: {scaler.scale_}')
```

ESERCIZIO 1: Classificazione Iris (Parte 1)

📌 Obiettivo: Classificare i fiori iris in 3 categorie
📊 Dataset: 150 campioni, 4 features (lunghezza sepali, petali, etc)

```
# Step 1: Carica dati
from sklearn.datasets import load_iris
iris = load_iris()
X, y = iris.data, iris.target

# Step 2: Dividi train/test
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42)
```

ESERCIZIO 1: Classificazione Iris (Parte 2)

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

# Step 4: Allena Decision Tree
dt = DecisionTreeClassifier(max_depth=5, random_state=42)
dt.fit(X_train_scaled, y_train)
dt_score = dt.score(X_test_scaled, y_test)
print(f'Decision Tree Accuracy: {dt_score:.3f}')
```

💡 Prova anche RandomForestClassifier!

Pipeline: Automatizzare il Workflow

Utilizziamo la Pipeline per automatizzare il flusso: scaling + classificazione con Random Forest.

```
from sklearn.pipeline import Pipeline

# SENZA Pipeline dovremmo fare: fit_transform → fit → predict
# CON Pipeline: una riga sola!

pipe = Pipeline([
    ('scaler', StandardScaler()),
    ('model', RandomForestClassifier(n_estimators=100))
])

pipe.fit(X_train, y_train)            # fa fit di scaler + modello in sequenza
score = pipe.score(X_test, y_test)    # scala X_test e valuta
y_pred = pipe.predict(X_test)         # scala X_test e predice
```

Collega il preprocessing e il modello in un'unica "catena"; quando avvia fit, predict o score, la Pipeline esegue automaticamente tutti i passi nell'ordine giusto, evitando errori e data leakage.

FeatureUnion: Combinare Features

Combina più trasformazioni delle feature in parallelo per ottenere un unico matrice X_combined.
È utile quando si vuol dare al modello diverse rappresentazioni degli stessi dati (es. originali + PCA, originali + statistiche derivate) senza scrivere tutta la logica di concatenazione.

```
from sklearn.pipeline import FeatureUnion
from sklearn.decomposition import PCA

# Combina: scaled features + PCA
union = FeatureUnion([                  # mette insieme (concatena per colonne) questi output:
                                          hai sia le feature scalate originali sia le 2
                                          componenti PCA in un unico X_combined
    ('scaled', StandardScaler()),       # produce le feature scalate
    ('pca', PCA(n_components=2))        # produce 2 nuove feature "riassuntive"
])

X_combined = union.fit_transform(X_train)
print(f'Features combinate: {X_combined.shape[1]}')

# Utile per ensemble: combina preprocessing diversi
```

ESERCIZIO 2: Titanic - Data + Preprocessing

```
📌 Obiettivo: Predire chi è sopravvissuto al Titanic
📊 Features: Age, Sex, Fare, Pclass, Embarked, etc

# Step 1: Carica con Pandas
import pandas as pd
df = pd.read_csv('titanic.csv')

# Step 2: Gestisci missing values
df["Age"] = df["Age"].fillna(df["Age"].median())
df["Embarked"] = df["Embarked"].fillna(df["Embarked"].mode()[0])

# Step 3: Encoding categoriche (Sex, Embarked)
from sklearn.preprocessing import LabelEncoder
```

ESERCIZIO 2: Titanic - Classificazione

```
# Step 4: Prepara X, y
X = df[['Age', 'Fare', 'Pclass', 'Sex_encoded']]
y = df['Survived']

# Step 5: Train/Test Split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y)

# Standardizzazione
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Addestramento
model = RandomForestClassifier(n_estimators=100, random_state=42)
model.fit(X_train_scaled, y_train)
y_pred = model.predict(X_test_scaled)
```

Model Evaluation: Metriche Avanzate

Valutare le prestazioni di un modello di classificazione in Machine Learning, usando metriche avanzate di `sklearn.metrics`. Dopo aver addestrato un modello (es. RandomForest, LogisticRegression, ecc.), dobbiamo sapere quanto è efficace nel fare previsioni su dati mai visti (`X_test`).
Questo script calcola 5 metriche fondamentali per capire se il modello è buono, mediocre o da migliorare.

Model Evaluation: Metriche Avanzate

```
from sklearn.metrics import (accuracy_score, precision_score,recall_score, f1_score,
confusion_matrix, roc_auc_score)


# Predizioni
y_pred = model.predict(X_test)
y_pred_proba = model.predict_proba(X_test)[:, 1]


# Metriche
accuracy = accuracy_score(y_test, y_pred)   # Percentuale di previsioni corrette su tutti i casi
precision = precision_score(y_test, y_pred) # Quanti dei positivi previsti sono davvero positivi
(evita falsi allarmi)
recall = recall_score(y_test, y_pred)       # Quanti dei veri positivi sono stati trovati (evita
falsi negativi)
f1 = f1_score(y_test, y_pred)    # Media armonica tra precision e recall (bilancia i due)
auc = roc_auc_score(y_test, y_pred_proba)   # Area sotto la curva ROC: misura la capacità del
modello di distinguere le classi
```

Confusion Matrix: Visualizzazione

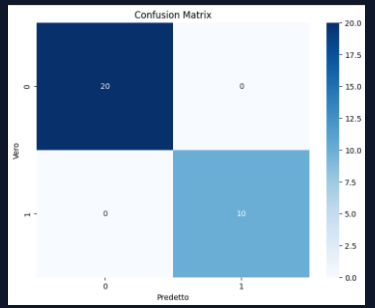
```
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns

cm = confusion_matrix(y_test, y_pred)

# Visualizza heatmap
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.xlabel('Predetto')
plt.ylabel('Vero')
plt.title('Confusion Matrix')
plt.show()
```

La confusion matrix mostra:
- True Positives (TP): predetti 1, erano 1
- True Negatives (TN): predetti 0, erano 0
- False Positives (FP): predetti 1, erano 0
- False Negatives (FN): predetti 0, erano 1

La heatmap rende visiva la qualità del modello:
- Numeri alti sulla diagonale → buone previsioni
- Numeri fuori diagonale → errori



ROC Curve & AUC Score

```
from sklearn.metrics import roc_curve, auc

# Calcola ROC curve
fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba)
auc_score = auc(fpr, tpr)

# Plot
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, label=f'ROC (AUC={auc_score:.3f})')
plt.plot([0, 1], [0, 1], 'k--', label='Random Classifier')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend()
plt.show()
```

- fpr: False Positive Rate
- tpr: True Positive Rate
- thresholds: soglie di decisione
- auc_score: area sotto la curva ROC

Interpretazione:
La ROC Curve mostra il compromesso tra **True Positive Rate** e **False Positive Rate.**

Una curva più vicina all'angolo in alto a sinistra indica un modello migliore.

L'AUC (Area Under Curve) misura la capacità del modello di distinguere le classi:

- AUC = 0.5 → modello casuale
- AUC = 1.0 → classificazione perfetta
- AUC > 0.8 → modello buono

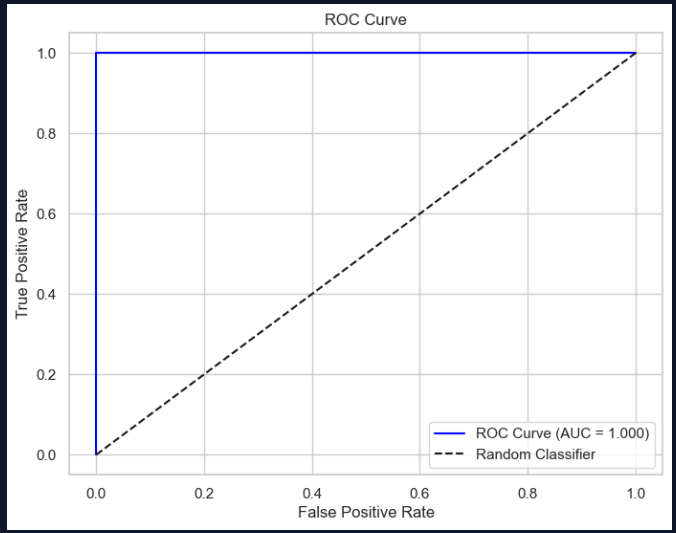
ROC Curve & AUC Score

Interpretazione:
La ROC Curve mostra il compromesso tra **True Positive Rate** e **False Positive Rate.**

Una curva più vicina all'angolo in alto a sinistra indica un modello migliore.

L'AUC (Area Under Curve) misura la capacità del modello di distinguere le classi:

- AUC = 0.5 → modello casuale
- AUC = 1.0 → classificazione perfetta
- AUC > 0.8 → modello buono



Feature Importance & Feature Selection

```
# Quando addestri un modello Random Forest, puoi chiedergli: "Quali feature ha usato di più per prendere decisioni?"
importances = model.feature_importances_

# Questa riga restituisce un array con l'importanza di ogni feature (valori tra 0 e 1).
for feat, imp in zip(feature_names, importances):
    print(f'{feat}: {imp:.4f}')

# Visualizza
plt.barh(feature_names, importances)
plt.xlabel('Importanza')
plt.show()

# Se vuoi selezionare solo le K feature più rilevanti, puoi usare:
from sklearn.feature_selection import SelectKBest, f_classif

# Questo crea un selettore che valuta ogni feature con un test statistico (f_classif = ANOVA F-test) e tiene solo le migliori. Esempio:
selector = SelectKBest(score_func=f_classif, k=3)X_new = selector.fit_transform(X, y)
```

Interpretazione
- Feature con importanza alta → molto usata dal modello
- Feature con importanza bassa → quasi ignorata
- Utile per capire quali variabili influenzano davvero la predizione

ESERCIZIO 3: Feature Engineering (Titanic)

In questo esercizio imparerai a:

- creare nuove feature significative
- trasformare variabili grezze in informazioni più utili
- migliorare la qualità del dataset per la classificazione.

Useremo il dataset Titanic e costruiremo:

```
# Feature 1: Age Groups → per creare fasce d'età
df['Age_group'] = pd.cut(df['Age'],
    bins=[0, 12, 18, 35, 60, 100])

# Feature 2: Family Size → per calcolare la dimensione familiare
df['Family_size'] = df['SibSp'] + df['Parch'] + 1

# Feature 3: Is_alone → indicatore binario di viaggio da soli
df['Is_alone'] = (df['Family_size'] == 1).astype(int)

# Risultato: Più features rilevanti → Modello migliore!
```

Cross-Validation: k-fold e stratified

La validazione incrociata è una tecnica per valutare la qualità di un modello su dati non visti, evitando che la valutazione dipenda da un singolo split train/test.
Queste tre tecniche di validazione con scikit-learn servono a stimare la performance del modello in modo più robusto e affidabile.

```
from sklearn.model_selection import (cross_val_score, StratifiedKFold, cross_validate)

# K-Fold semplice
scores = cross_val_score(model, X, y, cv=5)

# StratifiedFold (mantiene proporzioni classi)
skf = StratifiedKFold(n_splits=5, shuffle=True)
scores = cross_val_score(model, X, y, cv=skf)

# cross_validate per valutare più metriche contemporaneamente
scoring = {'accuracy': 'accuracy', 'auc': 'roc_auc'}
results = cross_validate(model, X, y, cv=5, scoring=scoring)
```

Cross-Validation: k-fold e stratified

K-Fold:

- Divide il dataset in 5 blocchi (fold)
- Per ogni fold: allena su 4, testa su 1
- Restituisce 5 punteggi (uno per ogni fold)
- Utile per stimare la variabilità del modello

StratifiedFold:

- Come K-Fold, ma mantiene le proporzioni delle classi in ogni fold
- Fondamentale per classificazione binaria o sbilanciata
- Evita che un fold abbia solo esempi di una classe

cross_validate:

- Permette di calcolare più metriche contemporaneamente
- Restituisce un dizionario con:
  - punteggi di accuratezza
  - punteggi AUC (area sotto la curva ROC)
- Utile per confrontare modelli su più criteri

Quando usare ciascuno?

- Usa cross_val_score per valutazioni rapide su una metrica
- Usa StratifiedKFold se hai classi sbilanciate
- Usa cross_validate se vuoi confrontare accuracy, AUC, F1, ecc. insieme

GridSearchCV: Hyperparameter Tuning

GridSearchCV viene usato per ottimizzare i iperparametri di un modello, in questo caso un `RandomForestClassifier`.

Cos'è GridSearchCV?
È una tecnica di ricerca esaustiva:

- prova tutte le combinazioni di iperparametri specificati
- valuta ogni modello con cross-validation
- restituisce la combinazione che dà le migliori prestazioni

Cosa ottieni:

- Un modello ottimizzato
- Una valutazione robusta
- Un set di iperparametri da usare per il modello finale

GridSearchCV: Hyperparameter Tuning

```
from sklearn.model_selection import GridSearchCV

# Dizionario dei parametri da testare. GridSearch proverà tutte le combinazioni di questi valori.
params = {
    'max_depth': [5, 10, 15, None],          # profondità massima degli alberi
    'min_samples_split': [2, 5, 10],         # minimo numero di campioni per dividere un nodo
    'n_estimators': [50, 100, 200] }         # numero di alberi nella foresta

# Creazione dell'oggetto GridSearchCV
grid = GridSearchCV(RandomForestClassifier(), params, cv=5)      # usa 5-fold cross-validation per
                                                                 valutare ogni combinazione

# Addestramento su dati di training
grid.fit(X_train, y_train)                  # Prova tutte le combinazioni. Per ciascuna: allena +
                                            valuta con cross-validation. Salva la migliore

# Risultati
print(f'Best params: {grid.best_params_}')     # combinazione ottimale
print(f'Best score: {grid.best_score_:.3f}')   # punteggio medio ottenuto con quella combinazione
```

RandomizedSearchCV: Tuning Veloce

È una tecnica di ricerca casuale: invece di provare tutte le combinazioni come GridSearchCV, ne prova solo un numero limitato, scelto a caso.
Questo lo rende molto più veloce, soprattutto quando hai tanti iperparametri.

```
from sklearn.model_selection import RandomizedSearchCV

params = {
    'max_depth': [5, 10, 15, 20, None],     # profondità massima degli alberi
    'min_samples_split': [2, 5, 10, 20],    # minimo numero di campioni per dividere un nodo
    'n_estimators': [50, 100, 200, 300]     # numero di alberi nella foresta
}

# In totale ci sarebbero 5 × 4 × 4 = 80 combinazioni. Ma con RandomizedSearchCV ne testiamo solo
20.
rand_search = RandomizedSearchCV(
    RandomForestClassifier(), params, # prova solo 20 combinazioni casuali
    n_iter=20, cv=5, random_state=42) # usa 5-fold cross-validation per valutare ogni combinazione
rand_search.fit(X_train, y_train)
```

Salvataggio & Caricamento Modello

Quando hai addestrato un modello che funziona bene, puoi:
- riutilizzarlo senza doverlo riaddestrare
- condividerlo con altri
- metterlo in produzione (deployment in un'app web, API, o script)

```
import joblib           # Meglio di pickle per sklearn. È ottimizzato per oggetti grandi come
                        modelli e pipeline.

# Salva il modello addestrato. Puoi scegliere qualsiasi nome e estensione (.pkl, .joblib,...)
joblib.dump(model, 'my_model.pkl')

# Carica il modello. Recupera il modello salvato.
model_loaded = joblib.load('my_model.pkl')

# - Usa il modello caricato per fare predizioni su nuovi dati X_new
y_new_pred = model_loaded.predict(X_new)

# Salva anche la pipeline
joblib.dump(pipe, 'my_pipeline.pkl')
```

Scikit-learn Ecosystem

🔗 Librerie che si integrano con Scikit-learn:

- XGBoost, LightGBM, CatBoost → Gradient Boosting avanzato
- Imbalanced-learn → Gestione classi sbilanciate (SMOTE)
- Hyperopt → Bayesian Optimization per tuning
- Optuna → Moderne alternative a GridSearch
- Skorch → Deep Learning (PyTorch) con API sklearn
- Imblearn → Oversampling/undersampling per dati sbilanciati

Best Practices Scikit-learn

- ✅ Usa Pipeline per evitare data leakage
- ✅ SEMPRE fit_transform su train, transform su test
- ✅ Usa stratify=y in train_test_split
- ✅ Cross-validation su train set (NON su test!)
- ✅ Hyperparameter tuning su train set con CV
- ✅ Valuta SEMPRE su test set separato
- ✅ Salva modelli con joblib, non pickle
- ✅ Usa scaling per distance-based algorithms (KNN, SVM, KMeans)
- ✅ Tree-based models (DT, RF) non necessitano scaling

Workflow Completo: Checklist

1. ✅ Load data (load_iris, load_wine, pandas.read_csv)
2. ✅ EDA (shape, describe, missing values, correlations)
3. ✅ Handle missing values (fillna, dropna)
4. ✅ Encoding (LabelEncoder, OneHotEncoder per categoriche)
5. ✅ Feature scaling (StandardScaler, MinMaxScaler)
6. ✅ Train/Test Split (stratify=y per classificazione)
7. ✅ Model Selection & Pipeline (Pipeline + preprocessor)
8. ✅ Hyperparameter Tuning (GridSearchCV, RandomizedSearchCV)
9. ✅ Model Evaluation (accuracy, precision, recall, F1, AUC)
10. ✅ Save Model (joblib.dump)

Risorse Ufficiali & Documentazione

📚 Documentazione Ufficiale: https://scikit-learn.org/stable/

📖 User Guide completo con esempi

🔗 Galleria di esempi (Gallery of examples)

💡 API Reference per ogni modulo

📊 Dataset online per esercitazione:

- Kaggle.com (Titanic, Housing, etc)
- UCI Machine Learning Repository
- sklearn.datasets built-in

Prossimi Passi

Progetti Pratici & Deep Dive

Progetto 1: Housing (Regressione) | Progetto 2: Heart Disease (Classificazione)