Data leakage: analisi causale, implicazioni statistiche e strategie di mitigazione

Eduardo De Filippis
www.eduardodefilippis.com

Nel ciclo di vita di un progetto di data science, poche insidie sono tanto perniciose quanto il data leakage. A differenza dell'overfitting, che è spesso sintomo di un'eccessiva complessità del modello rispetto ai dati, il data leakage è un errore metodologico fondamentale nella costruzione del dataset o nella pipeline di validazione. Esso crea un'illusione di efficace modellazione a fronte di modelli che, pur mostrando performance eccezionali in fase di training e validazione, falliscono rovinosamente una volta messi in produzione.

Questo articolo esplora la natura statistica del leakage, le sue manifestazioni più comuni (target leakage e train-test contamination) e le architetture software necessarie per prevenirlo.

1 Definizione formale e probabilistica

Il data leakage si verifica quando il dataset di addestramento include informazioni che non sarebbero disponibili al momento dell'inferenza nel mondo reale.

Formalmente, l'obiettivo in ambito supervised learning è stimare la distribuzione condizionata della variabile target Y dato un vettore di feature X, ovvero $P(Y|X)$. Un modello robusto apprende una funzione $f(X)$ che approssima questa probabilità minimizzando una funzione di costo.

Tuttavia, il leakage introduce una variabile latente Z (informazione "proibita" o futura) nel set di addestramento. Di conseguenza, il modello non apprende $P(Y|X)$, bensì:

$$P(Y|X, Z)$$

Poiché Z contiene spesso una correlazione spuria o causale diretta con Y (ad esempio, è una conseguenza di Y e non una causa), il modello sposta il suo peso decisionale quasi interamente su Z. Quando il modello viene portato in produzione (deploy), la variabile Z non è disponibile ($Z = \emptyset$), facendo crollare la capacità predittiva del modello e collassare il modello stesso. Ciò poiché la distribuzione appresa $P(Y|X, Z)$ differisce radicalmente dalla distribuzione reale $P(Y|X)$.

2 Tassonomia del leakage

Possiamo categorizzare il fenomeno del data leakage in due macro-famiglie principali, distinte per origine e impatto.

2.1 Target leakage (causalità inversa)

Il target leakage è l'errore più grave e concettuale. Avviene quando nel vettore delle feature vengono incluse variabili che sono, cronologicamente o causalmente, a valle della variabile target.

Consideriamo il classico esempio della previsione del "churn" (abbandono del cliente). L'obiettivo è prevedere se un cliente abbandonerà il servizio nel mese successivo (t_{+1}). Se il dataset include una feature come `ha_ricevuto_email_conferma_disdetta`, stiamo introducendo un proxy perfetto del target. Questa email viene generata dopo che l'evento target (la decisione di disdire) si è verificato.

In questo scenario, il modello non impara a riconoscere i segnali precoci di insoddisfazione (che è il vero obiettivo di business), ma impara semplicemente a leggere l'avvenuta disdetta. In produzione, dove dobbiamo agire prima che la disdetta avvenga, questa informazione sarà assente.

2.2 Train-test contamination (bias statistico)

Questa tipologia è più sottile ed è legata a un'implementazione errata della pipeline di preprocessing. La contaminazione ("fuga di dati") avviene quando le statistiche globali dell'intero dataset influenzano la trasformazione dei dati di training.

Il caso più immediato è la standardizzazione (Z-score normalization). La formula per trasformare una feature x è:

$$z = \frac{x - \mu}{\sigma}$$

dove μ è la media e σ la deviazione standard. In librerie come scikit-learn (StandardScaler), σ è calcolata sulla popolazione (N al denominatore):

$$\sigma = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \mu)^2}{N}}$$

L'errore comune consiste nel calcolare μ e σ sull'intero dataset X (prima dello split tra training e test). Facendo ciò, il training set "vede" indirettamente ("sbircia") la distribuzione del test set attraverso i parametri globali.

L'impatto può sembrare marginale, ma in realtà implica una sorta di riduzione artificiale dell'errore di generalizzazione stimato.

Questa problematica può emergere in pressoché tutti i modelli ma può essere specialmente evidente ed impattante quando si trattano dataset ad alta dimensionalità o se si lavora con algoritmi sensibili alla scala (come SVM, KNN o Reti Neurali).

3 Applicazione pratica: l'impatto della contaminazione

Per quantificare l'effetto della contaminazione train-test, analizziamo uno scenario controllato in Python. Utilizzeremo un dataset sintetico con molto rumore per evidenziare come una pipeline errata possa portare a stime ottimistiche.

```python
import numpy as np
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
from sklearn.metrics import accuracy_score

############################

# Generazione del dataset sintetico
X, y = make_classification(
    n_samples=1000,
    n_features=20,
    n_informative=2,
    n_redundant=2,
    random_state=2025
)

############################

# SCENARIO A: APPROCCIO ERRATO (leakage)
# Errore: Normalizzazione applicata all'intero dataset prima dello split
# Il modello "sbircia" la distribuzione del test set tramite media e dev. std globali
scaler_leak = StandardScaler()
X_leaked = scaler_leak.fit_transform(X)

X_train_leak, X_test_leak, y_train_leak, y_test_leak = train_test_split(
    X_leaked, y, test_size=0.2, random_state=42
)

```

```
model_leak = LogisticRegression()
model_leak.fit(X_train_leak, y_train_leak)
y_pred_leak = model_leak.predict(X_test_leak)
score_leak = accuracy_score(y_test_leak, y_pred_leak)

print(f"Accuratezza con leakage (stima distorta): {score_leak:.4f}")

##########################

# SCENARIO B: APPROCCIO CORRETTO (pipeline)
# Split dei dati grezzi. I parametri di scaling sono appresi solo sul train set
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=2025
)

# Utilizziamo una pipeline di Scikit-Learn
# La pipeline garantisce che durante il 'fit', lo scaler veda solo X_train
# Durante il predict, il modello user  le statistiche apprese dal train per
    trasformare X_test.
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('model', LogisticRegression())
])

pipeline.fit(X_train, y_train)
y_pred_correct = pipeline.predict(X_test)
score_correct = accuracy_score(y_test, y_pred_correct)

print(f"Accuratezza corretta (stima corretta e realistica): {score_correct:.4f}")

##########################

# Analisi della distorsione
delta = score_leak - score_correct
print(f"Sovrastima della performance dovuta al leakage: {delta:.4f}")
```

Listing 1: Confronto tra approccio errato (leakage) e corretto (pipeline)

L'utilizzo della classe Pipeline di scikit-learn non è solo una questione di stile, ma una necessità architetturale per incapsulare le trasformazioni e garantire l'isolamento dei fold durante la cross-validation.

4 Il caso delle serie temporali: il look-ahead bias

Quando si trattano dati temporali (time series), il concetto di leakage si estende al look-ahead bias.

Utilizzare una funzione standard come `train_test_split` con l'opzione `shuffle=True` (spesso di default) su dati temporali è metodologicamente errato. Infatti, mescolando casualmente i dati, si addestra il modello su dati del futuro (es. dati di Dicembre) per predire dati del passato (es. dati di Gennaio dello stesso anno). In tal caso, poiché le serie temporali presentano spesso autocorrelazione e trend, il modello interpolerà i valori mancanti ottenendo performance irreali.

Per non incorrere nel look-ahead bias, la validazione deve essere rigorosamente cronologica. Ovvero, il training set deve sempre temporalmente antecedere il validation set, simulando così la reale disponibilità del dato.

Strumenti come `TimeSeriesSplit` sono essenziali in questo contesto.

5 Rilevamento e strategie di mitigazione

Identificare il leakage richiede un approccio investigativo che combina analisi statistica e conoscenza del dominio.

Il campanello d'allarme più evidente è una performance "troppo bella per essere vera". Accuratezze superiori al 90-95% o AUC $>$ 0.90-0.95% in problemi complessi su dati reali dovrebbero sollevare immediati sospetti.

In questi casi, un'analisi dell'importanza delle feature (es. tramite `feature_importances_` in modelli tree-based o dei coefficienti in modelli lineari) spesso rivela una singola feature che domina completamente il modello. Quella feature è il probabile vettore del leakage.

Per mitigare il rischio, è fondamentale adottare un approccio difensivo e preventivo:

- **Separazione precoce:** Il `train_test_split` deve essere la prima operazione effettuata, prima di qualsiasi imputazione di valori mancanti, encoding o scaling.
- **Pipeline automatizzate:** Utilizzare sempre wrapper come le Pipeline per garantire che ogni trasformazione sia "fittata" solo sul train e applicata al test in modalità inferenza.
- **Validazione temporale:** Se esiste una colonna con dati temporali (es. data/ora), verificare sempre se il target dipende dal tempo e, in tal caso, utilizzare split cronologici.
- **Collaborazione con gli SME (Subject Matter Experts):** Spesso il leakage è nascosto nella semantica dei dati, piuttosto che nella statistica. Discutere con gli esperti di dominio per capire l'esatto momento in cui ogni feature viene registrata nei sistemi aziendali è l'unico modo per escludere il target leakage.

In conclusione, il data leakage va ben oltre il bug tecnico, ma rappresenta un fallimento nella modellazione della realtà.

Nella data science resta prioritario l'obiettivo di garantire che il processo di validazione sia una simulazione fedele dell'ignoto che il modello dovrà affrontare in produzione.