UNIVERSITÀ
DEGLI STUDI DI BARI
ALDO MORO

Fraud-Detection

Documentazione
Caso di Studio

AA 2023-24

Gruppo di lavoro

Monopoli Vincenzo [MAT. 760451]
v.monopoli8@studenti.uniba.it

Repository GitHub:

https://github.com/Vincy02/Fraud-Detection

Sommario

Introduzione

Nell'era digitale, la rapida crescita delle transazioni on-line ha portato a un aumento delle frodi. Per i commercianti e le piattaforme di vendita, la capacità di rilevare e prevenire transazioni fraudolente è diventata una priorità critica. Tali transazioni non solo rappresentano una perdita economica significativa, ma danneggiano soprattutto la fiducia dei clienti e la reputazione della piattaforma.

Questo progetto si propone di sviluppare un sistema di rilevamento delle frodi basato su tecniche di machine learning, in grado di analizzare le transazioni.
L'obiettivo sarebbe quello di integrare il sistema direttamente nei processi di pagamento delle piattaforme di vendita online, per identificare e bloccare in modo proattivo transazioni sospette prima che possano essere processate.

Per raggiungere questo scopo, il sistema sarà sviluppato usando diverse tecniche di apprendimento automatico, tra cui modelli di apprendimento supervisionato e non supervisionato, tecniche di riduzione della dimensionalità come il PCA (Principal Component Analysis) per migliorare l'efficienza e il tempo di risposta (usato nell'apprendimento non supervisionato).
Inoltre, verrà fatto uso anche di un approccio basato sul ragionamento logico che permetterebbe al sistema di fare inferenze più avanzate e quindi in futuro anche di adattarsi a nuove tipologie di frode emergenti.

Elenco argomenti di interesse

- **Rappresentazione e ragionamento logico**: utilizzo di Prolog per il ragionamento su una base di conoscenza ingegnerizzata, partendo dai dati contenuti nel dataset, con lo scopo di generare nuove conoscenze.
- **Apprendimento non supervisionato**: utilizzo del K-means per la creazione di cluster partendo dal dataset fornito.
- **Apprendimento supervisionato**: ensemble di alberi decisionali (Random Forest, Gradient Boosting) e modello lineare per stimare probabilità (Regressione Logistica) per portare a termine un task di classificazione binaria.
- **Rete Neurale:** sperimentata la creazione e l'addestramento di una rete neurale "*semplice*" per il task di classificazione binaria.

Descrizione del dominio e dati utilizzati

Il dominio è rappresentato dal dataset in formato *csv* scaricato da Kaggle, i cui dati sono stati generati utilizzando la libreria *Faker* di Python e logiche personalizzate per simulare schemi di transazioni realistiche e scenari di frode, infatti esso non rappresenta individui o transazioni reali.
Contiene una varietà di caratteristiche comunemente presenti nei dati transazionali, con attributi aggiuntivi specificamente creati per supportare lo sviluppo e il test di algoritmi di rilevamento delle frodi.

Le feature presenti nel dataset originale sono:

- **Transaction ID**: identificatore univoco per ogni transazione;
- **Customer ID**: identificatore univoco per ogni cliente;
- **Transaction Amount**: ammontare in dollari di ogni transazione;
- **Transaction Date**: data in cui è avvenuta la transazione;
- **Payment Method**: metodo usato per completare la transazione (credit card, PayPal, ecc.);
- **Product Category**: categoria del prodotto coinvolto nella transazione;
- **Quantity**: numero di unità del prodotto coinvolto nella transazione;
- **Customer Age**: età del cliente coinvolto nella transazione;
- **Customer Location**: locazione geografica del cliente;
- **Device Used**: tipologia di dispositivo usato per effettuare la transazione (mobile, desktop, ecc.);
- **IP Address**: indirizzo IP del dispositivo usato per la transazione;
- **Shipping Address**: indirizzo a cui la merce è stata spedita;
- **Billing Address**: indirizzo associato al metodo di pagamento;
- **Is Fraudulent**: variabile booleana che indica se la transazione è fraudolenta o no (1 se fraudolenta, 0 se non lo è);
- **Account Age Days**: numeri di giorni trascorsi dalla data di creazione dell'account del cliente al momento della transazione;
- **Transaction Hour**: ora del giorno in cui è avvenuta la transazione.

Preprocessing dei dati

Caricamento del dataset

Il primo passo è stato il caricamento del dataset, scaricato da Kaggle, in formato CSV. Questo mi ha permesso di iniziare subito con l'analisi e la preparazione dei dati per gli step successivi.

```
import pandas as pd

dataset_name = "Fraudulent_E-Commerce_Transaction_Data.csv"
data = pd.read_csv("../Data/"+dataset_name)
```

Data cleaning

Successivamente, è stato avviato il processo di pulizia dei dati, avvenuto in due fasi. La prima “pulizia” dei dati è avvenuta prima dell’analisi di questi ultimi e quindi ancora prima della creazione della KB con Prolog, la seconda dopo questa fase.

```
# Rimozione della feature non importante
data.drop(['IP Address'], axis=1, inplace=True)
```

Inoltre, per seguire le convenzioni comunemente adottate, si è deciso di rinominare le colonne utilizzando la notazione **snake_case**, sostituendo gli spazi con underscore per garantire uniformità e facilità di utilizzo nel codice.

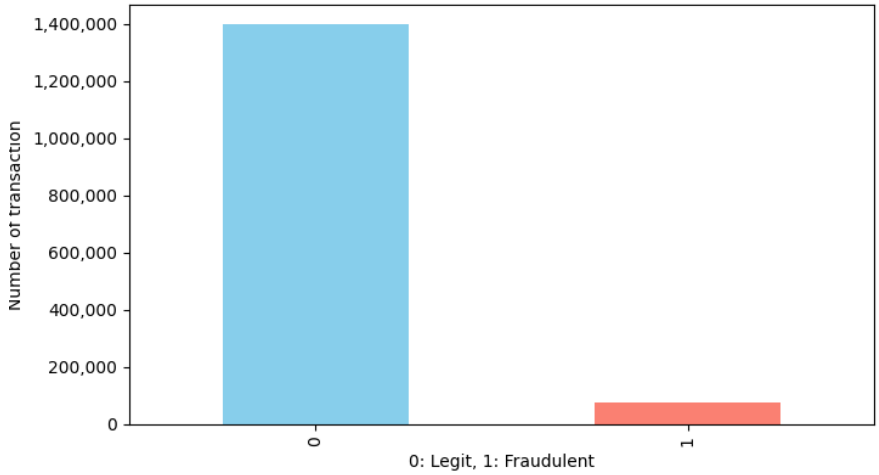
```
# Rinominazione delle colonne
cols_old = data.columns.tolist()
snakecase = lambda x: inflection.parameterize(x, separator='_')
cols_new = list(map(snakecase, cols_old))
data.columns = cols_new
```

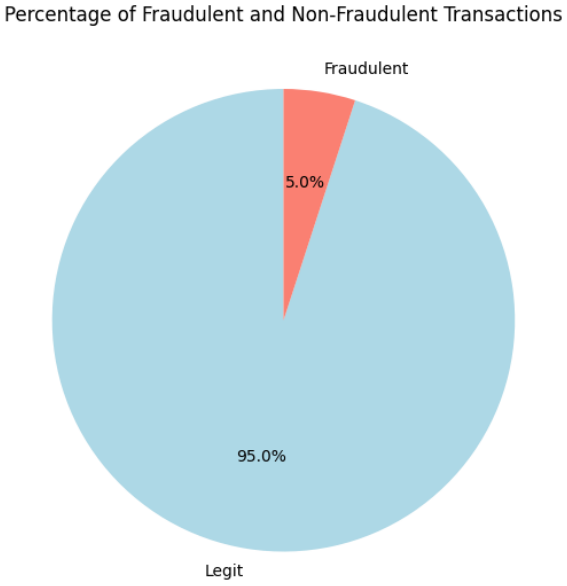
Non sono stati riscontrati valori nulli; quindi, non è stato necessario applicare ulteriori operazioni di **data imputation**.

```
RangeIndex: 1472952 entries, 0 to 1472951
Data columns (total 15 columns):
 #   Column              Non-Null Count    Dtype
---  ------              --------------    -----
 0   transaction_id      1472952 non-null  object
 1   customer_id         1472952 non-null  object
 2   transaction_amount  1472952 non-null  float64
 3   transaction_date    1472952 non-null  object
 4   payment_method      1472952 non-null  object
 5   product_category    1472952 non-null  object
 6   quantity            1472952 non-null  int64
 7   customer_age        1472952 non-null  int64
 8   customer_location   1472952 non-null  object
 9   device_used         1472952 non-null  object
 10  shipping_address    1472952 non-null  object
 11  billing_address     1472952 non-null  object
 12  is_fraudulent       1472952 non-null  int64
 13  account_age_days    1472952 non-null  int64
 14  transaction_hour    1472952 non-null  int64
dtypes: float64(1), int64(5), object(9)
memory usage: 168.6+ MB
```

Data analysis

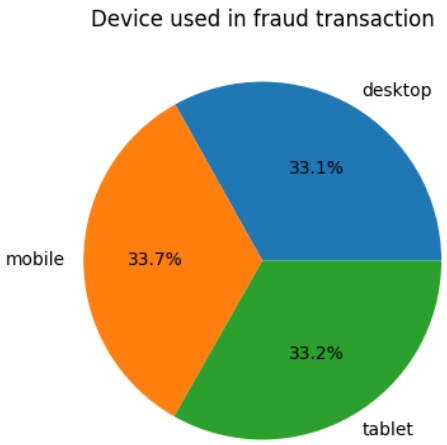
Dopo aver eseguito la prima fase di pulizia dei dati, mi sono dedicato dunque all'analisi esplorativa dei dati per comprendere meglio il dataset e identificarne le caratteristiche principali.
Ho creato diversi grafici per visualizzare la distribuzione delle variabili e individuare eventuali anomalie o pattern nascosti.
Questo processo mi ha aiutato a formulare ipotesi che saranno utili nello sviluppo dei modelli di machine learning e ha supportato il processo di selezione delle feature (“seconda fase di pulizia”), al fine di migliorare le performance dei modelli.
Inizialmente, ho deciso di analizzare la distribuzione delle transazioni fraudolente e non fraudolente.

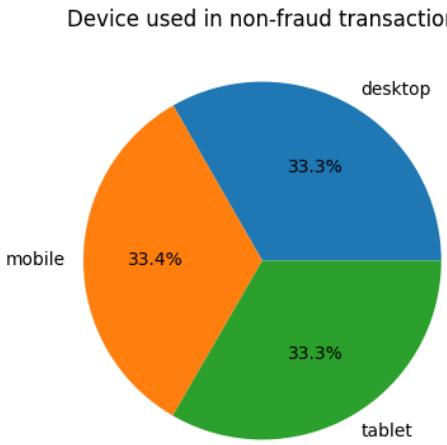


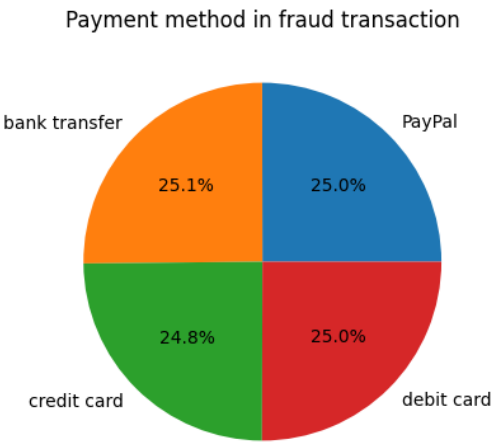


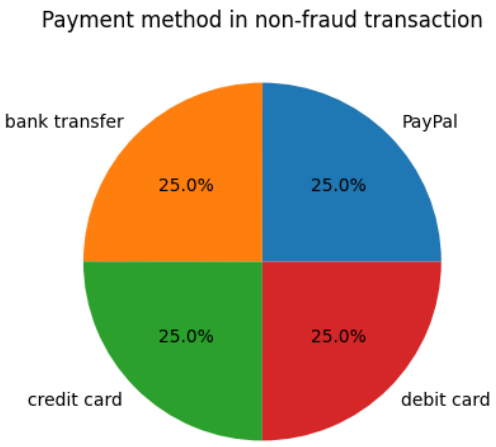
Dai grafici generati, è emerso chiaramente un problema di sbilanciamento dei dati. Questo squilibrio tra le classi potrebbe influenzare negativamente l'addestramento dei modelli, portando a prestazioni subottimali.
Per affrontare questo problema prevedo di applicare tecniche di **oversampling** (durante la fase di apprendimento supervisionato) per bilanciare le categorie e migliorare la capacità dei modelli di rilevare le transazioni fraudolente.

Successivamente, ho esaminato la distribuzione delle categorie di alcune variabili categoriche, come "*device_used*" e "*payment_method*".





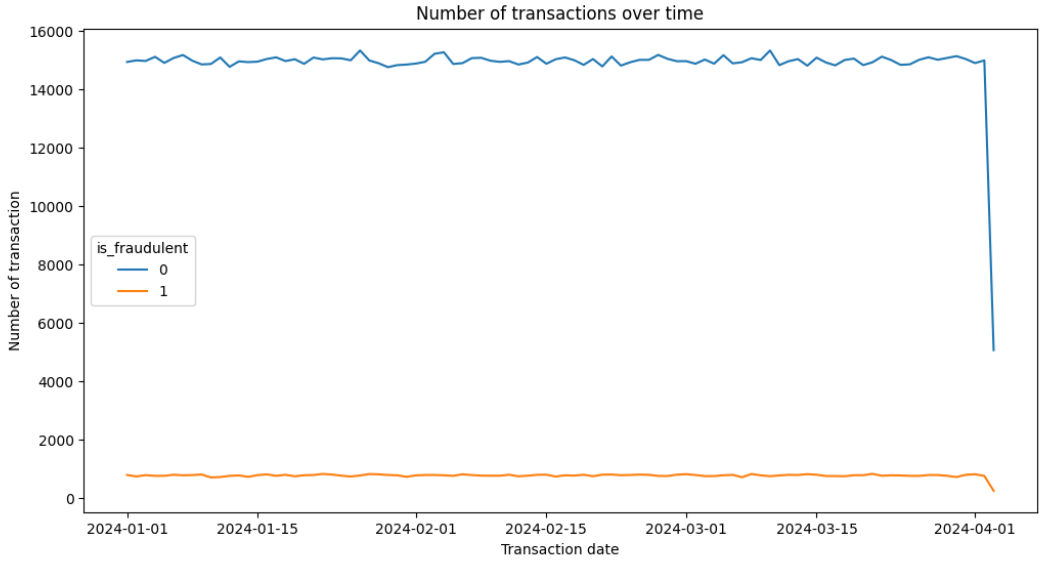




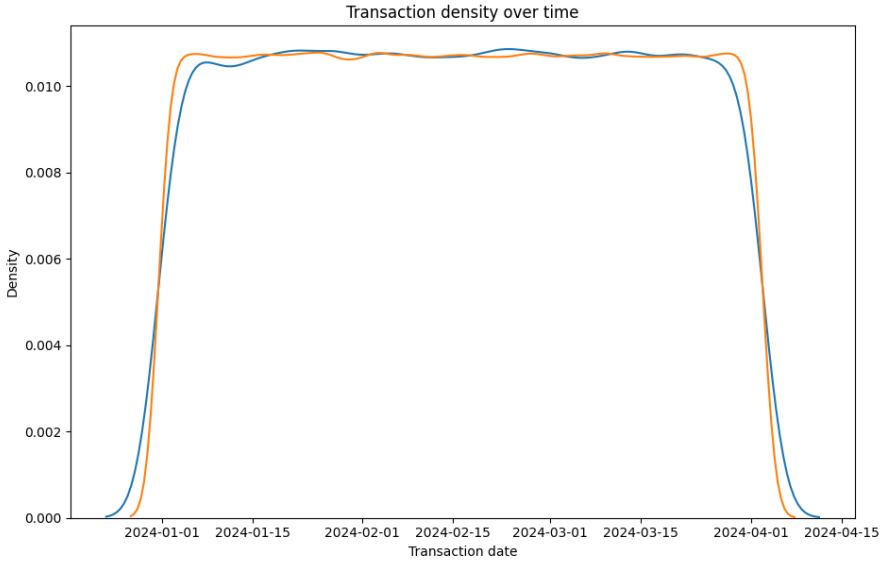
L'analisi della distribuzione dei metodi di pagamento nelle transazioni fraudolente e non fraudolente mostra una ripartizione pressoché uniforme tra le diverse opzioni (bonifico bancario, PayPal, carta di credito e carta di debito).

Sebbene questo possa indicare una bassa importanza di queste feature nell'addestramento del modello, ho deciso di mantenerle.
Sarebbe possibile fare ulteriori analisi multivariate per determinare se queste variabili, in combinazione con altre, possano effettivamente fornire informazioni utili per migliorare la performance del modello.

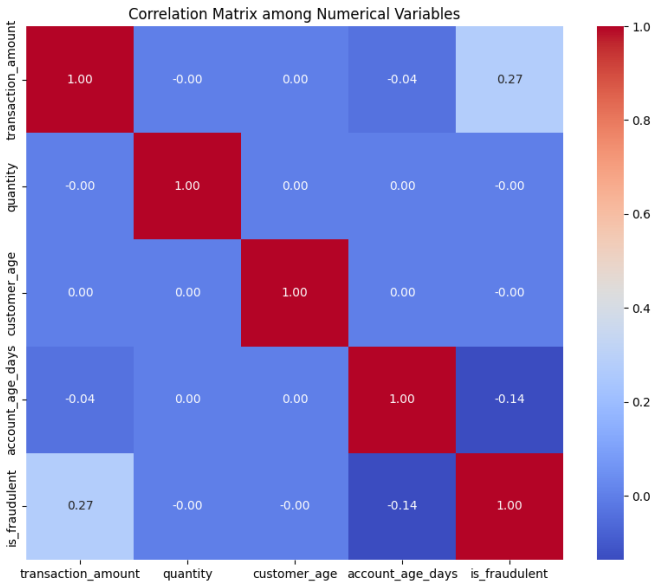
Un'altra analisi è stata effettuata sulla quantità di transazioni effettuate nel tempo, suddivise tra fraudolente e non fraudolente.
Si nota una sostanziale stabilità nel numero delle transazioni legittime e no, con un calo ponderale verso la fine del 2024-04-01; poiché come possiamo ricordare, i dati sono stati generati con la libreria Faker, e il calo è stato interpretato come un artefatto della generazione dei dati.



Ulteriore analisi per quanto riguarda la distribuzione delle transazioni nel tempo è stata effettuata con il Kernel Density Plot (KDE), che serve per stimare la distribuzione di una variabile continua in modo non parametrico.
Questo mi ha portato a ipotizzare che la data, presa singolarmente, non sia un forte indicatore per distinguere il tipo di transazione, dato che le loro distribuzioni sono molto simili tra loro; per questo motivo ho pensato che non fosse un indicatore per capire se una transazione potesse essere fraudolenta, infatti durante la “seconda pulizia” del dataset è stata etichettata come feature non importante ed eliminata.



Infine, è stata costruita una Heatmap (matrice di correlazione) per analizzare la correlazione tra le variabili numeriche selezionate, fornendo ulteriori indicazioni sulle relazioni tra le feature.



L'analisi della Heatmap rivela che l'importo della transazione presenta una correlazione positiva moderata con la probabilità che la transazione sia fraudolenta. Questo suggerisce che transazioni di importo elevato potrebbero essere associate a un rischio di frode maggiore.
Inoltre, si nota una leggera correlazione negativa tra l'età dell'account e la probabilità di frode, indicando che gli account più vecchi potrebbero essere meno a rischio.

Ragionamento logico

Il ragionamento logico si basa sulla logica matematica, tramite la quale viene costruita una Knowledge Base (KB), cioè una base di conoscenza che contiene assiomi. Gli assiomi rappresentano affermazioni assolute e si suddividono in **fatti** e **regole**.

- **Fatti**: sono verità immutabili, considerate "dati" certi e inalterabili.
- **Regole**: sono affermazioni condizionali, che stabiliscono che qualcosa è vero solo se vengono soddisfatte determinate condizioni.

Questa struttura permette di eseguire inferenze logiche, deducendo nuove informazioni a partire dagli assiomi presenti nella KB.

Nonostante le diverse features presenti nel dataset originale, per i task di classificazione che si vuole effettuare sui dati ho trovato essenziale introdurre altre features più significative.

Le features introdotte sono le seguenti:

- *is_suspicious_transaction*: se la transazione viene riconosciuta come possibile transazione fraudolenta in base al tipo di metodo di pagamento e il suo importo.
- *high_risk_category_and_amount*: se la transazione viene etichettata come rischiosa in base alla categoria e alla quantità del prodotto che si vuole acquistare.
- *suspicious_customer_location*: se il luogo da cui l'utente sta acquistando risulta più sospetto per quanto riguarda transazioni di tipo fraudolente.

Queste nuove conoscenze generate mi permetteranno di migliorare l'analisi e la classificazione delle transazioni. Tutte le features introdotte sono di tipo booleano, quindi 1 per *True* e 0 per *False*, e saranno salvate nelle corrispettive colonne insieme al dataset, il risultato finale è stato salvato in un file *csv*: "new_data".

Per creare dunque queste nuove conoscenze, per ogni dato del dataset è stata creata un fatto nella base di conoscenza in Prolog.
I fatti rappresentano informazioni concrete sui dati delle transazioni (presenti nel dataset), mentre le regole ci permetteranno di inferire le nuove informazioni in base ai dati presenti nei fatti.
Quindi il primo passo è stato generare i fatti da ogni singolo dato presente nel dataset; facendo ciò ho potuto applicare le regole discusse precedentemente.

Sia i fatti che le regole create sono stati salvati nei rispettivi file: *kb.pl* e *rules.pl*.

```
suspicious_threshold('clothing', 4).
suspicious_threshold('home & garden', 4).
suspicious_threshold('toys & games', 4).
suspicious_threshold('health & beauty', 4).
suspicious_threshold('electronics', 4).
is_suspicious_transaction(TransactionID) :- transaction(TransactionID, _, Amount, _, PaymentMethod, _, _, _, _, _, _, _, _, _, _),
                                            (Amount >= 2000, PaymentMethod == 'bank transfer').
high_risk_category_and_amount(TransactionID) :- transaction(TransactionID, _, _, _, _, Category, Quantity, _, _, _, _, _, _, _, _),
                                                suspicious_threshold(Category, Threshold), Quantity > Threshold.
suspicious_customer_location(TransactionID, CustomerLocation) :-
        transaction(TransactionID, _, _, Date, _, _, _, _, CustomerLocation, _, _, _, _, _, _),
        findall(_Date, (
            transaction(_TransactionID, _, _, _Date, _, _, _, _, CustomerLocation, _, _, _, _, _, _),
            _TransactionID \= TransactionID,
            Date1 is Date,
            Date2 is _Date,
            Diff is abs(Date1 - Date2),
            Diff =< 2
        ), Days),
        length(Days, Count),
        Count >= 50.
```

rules.pl

Le regole e i rispettivi threshold sono stati individuati a partire dai dati presenti nel dataset e assunzioni/ragionamenti puramente personali.

- is_suspicious_transaction: ho pensato che se la tipologia di pagamento fosse stata un *trasferimento bancario* ci fosse stata più probabilità che quest'ultimo fosse stato di tipo fraudolento, questo perché una volta effettuato un bonifico, è più difficile annullarlo rispetto a una transazione con carta di credito o PayPal dove spesso esistono procedure per contestare un pagamento; e anche perché trasferimenti bancari possono essere meno tracciabili, rendendo più difficile individuare la destinazione dei fondi.
  Questo a mio parere però non era abbastanza a rendere una transazione sospetta; per questo motivo ho pensato di calcolare la distribuzione dei pagamenti presenti nel dataset secondo alcuni thresholds e analizzarli.

```
226.76830923886183
12701.75
10.0
{0: 1329724, 500: 121515, 1000: 17436, 2000: 4081, 5000: 195, 10000: 1}
```

  I primi 3 dati sono rispettivamente la media degli importi delle transazioni, la transazione più grande e la più piccola; infine troviamo il vettore con i thresholds, da cui si può notare una notevole diminuzione di transazioni sorpassati i 2000 dollari.
  Per questo ho scelto questo ammontare per etichettare, in aggiunta al tipo di transazione, se quest'ultima potesse risultare sospetta.
- high_risk_category_and_amount: ho pensato di analizzare ogni categoria di prodotti presenti nel dataset ed etichettare come possibili transazioni sospette se la quantità di prodotti acquistata in una singola transazione superasse la media di quella categoria.

```
{'clothing': np.float64(2.9991641722077067),
 'home & garden': np.float64(2.9982227125084266),
 'toys & games': np.float64(2.9998300857047706),
 'health & beauty': np.float64(2.999617425101653),
 'electronics': np.float64(3.0043007985262147)}

{'clothing': np.int64(5),
 'home & garden': np.int64(5),
 'toys & games': np.int64(5),
 'health & beauty': np.int64(5),
 'electronics': np.int64(5)}

{'clothing': np.int64(1),
 'home & garden': np.int64(1),
 'toys & games': np.int64(1),
 'health & beauty': np.int64(1),
 'electronics': np.int64(1)}
```

Il primo vettore rappresenta la media dei prodotti acquistati per transazione per ogni categoria, il secondo vettore il massimo di prodotti acquistati per transazione per ogni categoria e infine il minimo di prodotti acquistati per transazione per ogni categoria.
È stato scelto di usare come thresholds per ogni categoria un numero di prodotti superiori 4, ovvero tutti quelle transazioni che stavano superando la media + 1.

- suspicious_customer_location: inizialmente l'idea era quella di vedere se lo stesso cliente (quindi stesso ID utente) avesse effettuato in un intorno di X giorni l'acquisto di uno stesso prodotto, perché molte frodi online avvengono perché si lascia la spunta su *acquisto ripetuto*. Però visto che non esistevano identificativi univoci e non ripetuti all'interno del dataset si è optato per analizzare le diverse locazioni da dove gli utenti stessero effettuando l'acquisto dei diversi prodotti e osservare se più di X=50 transazioni in un periodo di Y=4 giorni si fossero effettuate, identificando così una transazione sospetta.
  Perché 50 transazioni e 4 giorni di intervallo?
  Effettuando diverse prove randomiche sui dati ho notato che mediamente più di 40-50 transazioni in un periodo di tempo di 4 giorni non veniva superato.

```
Città: South Robert
    Data: 2024-02-10 ± 4gg, Transazioni: 30
    Data: 2024-01-23 ± 4gg, Transazioni: 32
    Data: 2024-02-25 ± 4gg, Transazioni: 36
    Data: 2024-03-13 ± 4gg, Transazioni: 45
    Data: 2024-01-01 ± 4gg, Transazioni: 12
    Data: 2024-02-21 ± 4gg, Transazioni: 39
```

```
Città: Lake Sherri
    Data: 2024-02-10 ± 4gg, Transazioni: 1
    Data: 2024-01-23 ± 4gg, Transazioni: 1
    Data: 2024-02-25 ± 4gg, Transazioni: 2
    Data: 2024-03-13 ± 4gg, Transazioni: 4
    Data: 2024-01-01 ± 4gg, Transazioni: 1
    Data: 2024-02-21 ± 4gg, Transazioni: 0
```

Esempi delle prove effettuate

Processing dei dati

A seguito dell'analisi esplorativa e dell'arricchimento del dataset con la fase di ragionamento logico, ho proceduto nuovamente alla selezione delle features utili per le fasi successive di apprendimento supervisionato e non supervisionato (come già accennato nella fase di preprocessing dei dati).

Le colonne ritenute non informative sono state eliminate.

Successivamente ho diviso le features rimanenti in due gruppi: numeriche e categoriche.
Le features numeriche sono come dice la parola stessa composte solo da valori numerici; invece le features categoriche sono attributi o caratteristiche del dataset che possono assumere un numero limitato di valori distinti, solitamente espressi come categorie o etichette.

Infine, ho creato due nuovi dataset: X, contenente le features selezionate (sia numeriche che categoriche), e y, contenente la variabile target 'is_fraudulent', che sarà utilizzata per l'addestramento dei modelli di classificazione.

```
# Rimozione della feature non importanti (v2)
data.drop(['transaction_id', 'customer_id',
           'shipping_address', 'billing_address',
           'transaction_date', 'customer_location'], axis=1, inplace=True)

# Definizione delle colonne numeriche e categoriche
numeric_columns = ['transaction_amount',
                   'quantity',
                   'customer_age',
                   'account_age_days',
                   'transaction_hour',
                   'is_suspicious_transaction',
                   'high_risk_category_and_amount',
                   'suspicious_customer_location']

categorical_columns = ['payment_method',
                       'product_category',
                       'device_used']

X = data[categorical_columns + numeric_columns].copy()
y = data['is_fraudulent'].copy()
```

Apprendimento non supervisionato

L'apprendimento non supervisionato è una tecnica di apprendimento automatico che consiste nel fornire al sistema informatico una serie di input (esperienza del sistema) che egli riclassificherà ed organizzerà sulla base di caratteristiche comuni per cercare di effettuare ragionamenti e previsioni sugli input successivi.

Durante la fase di apprendimento vengono forniti solo esempi non etichettati, in quanto le classi non sono note a priori ma devono essere apprese automaticamente dal sistema.

Esistono due tipo principali di algoritmi per l'apprendimento non supervisionato:

- **Clustering:** questi algoritmi raggruppano i dati in cluster, ovvero in gruppi di osservazioni simili.
  L'obiettivo è identificare sottoinsiemi omogenei all'interno del dataset.
- **Reduction of dimensionality**: questi algoritmi riducono la dimensionalità dei dati, proiettandoli in uno spazio a minore dimensione preservando la maggior parte della varianza.
  L'obiettivo è semplificare i dati e renderli più facili da visualizzare e analizzare.

Nel contesto di studio ho voluto verificare se l'apprendimento non supervisionato potesse fornire ulteriori spunti di analisi per identificare pattern latenti nel dataset delle transazioni fraudolente.
A tal fine ho pensato di condurre una analisi delle componenti principali (PCA), ovvero una tecnica statistica della riduzione della dimensionalità dei dati, mantenendo il più possibile l'informazione originale; per poi passare alla fase di clustering, scegliendo il numero di cluster ottimale attraverso la regola del gomito.

Analisi delle Componenti Principali

Il PCA riduce la complessità del dataset proiettandolo su uno spazio a dimensione ridotta, pur mantenendo la maggior parte delle informazioni contenute nei dati originali. Le variabili originali del dataset vengono trasformate in un nuovo insieme di variabili, chiamate componenti principali, ortogonali tra loro e ordinate in base alla quantità di varianza che spiegano nei dati.

Il PCA cerca quindi di trovare una nuova serie di componenti principali che rappresentano le direzioni lungo cui i dati variano di più.
Nel contesto di studio ho pensato di identificare 2 componenti principali (una per l'asse X e l'altra per l'asse Y).
La prima componente principale è quella lungo cui i dati mostrano la maggiore

dispersione; la seconda componente principale è ortogonale alla prima e spiega la massima varianza residua.
Sempre parlando del contesto di studio ho usato il PCA per ridurre la dimensionalità del dataset puntando ad identificare un numero di componenti principali dell'80% della varianza totale.
Questo valore di soglia è stato scelto per garantire che la maggior parte dell'informazione utile fosse preservata, permettendo al contempo una significativa semplificazione dei dati.

Il dataset è stato inizialmente standardizzato, in modo che ciascuna feature numerica avesse media pari a zero e deviazione standard pari a uno; e che ogni feature categorica fosse trasformata in variabili dummy, per consentire di lavorare sempre con variabili di tipo numerico e non tipo testuale, garantendo che il PCA non fosse influenzato da diverse scale di misura o dati di tipo testuale.

```
# Apprendimento non supervisionato
_X = X.copy()
scaler = StandardScaler()
_X[numeric_columns] = scaler.fit_transform(_X[numeric_columns])

_X = pd.get_dummies(_X, columns=categorical_columns)
_X.columns = _X.columns.str.replace(' ', '_', regex=False)

from unSupervisedLearning import clusterDataAndVisualize
clusterDataAndVisualize(_X)
```

Successivamente, il PCA è stato eseguito sul dataset standardizzato.

```
# Riduzione della dimensionalità mediante PCA
pca = PCA(n_components=0.8)
_dataSet = pca.fit_transform(dataSet)
```

Clustering con K-Means

Il **clustering** è una tecnica di apprendimento non supervisionato utilizzata per suddividere un insieme di dati in gruppi omogenei, detti **cluster**, in modo che gli elementi all'interno di ciascun cluster siano simili tra loro e dissimili dagli elementi di altri cluster.

Esistono due approcci principali per effettuare clustering: l'**hard clustering** e il **soft clustering**.
Nel **hard clustering**, ogni esempio viene assegnato in modo definitivo a un solo cluster, senza sovrapposizioni.
Nel **soft clustering**, invece, a ogni esempio viene associata una probabilità di

appartenenza a ciascun cluster, riflettendo così un grado di appartenenza che può essere distribuito tra più cluster.

Nel mio caso, ho scelto di utilizzare l'algoritmo di **hard clustering K-means**.
Uno dei problemi principali di questo algoritmo è determinare il numero ideale di cluster da fornire in input. Per fare ciò, ho utilizzato una strategia nota come **metodo del gomito** (*elbow method*). Questa tecnica prevede di analizzare la variazione dell'inertia al variare del numero di cluster.
L'**inertia** rappresenta la somma delle distanze quadrate tra ogni punto dei dati e il centro del cluster a cui è assegnato.

In pratica, l'algoritmo prevede di eseguire KMeans più volte, variando il numero di cluster e registrando l'inertia per ciascuna esecuzione. Successivamente, si rappresentano i risultati su un grafico che mostra il numero di cluster sull'asse delle ascisse e l'inertia sull'asse delle ordinate. Il grafico produrrà una curva in cui si identifica il cosiddetto "gomito": il punto in cui la riduzione dell'inertia inizia a essere meno significativa. Questo punto rappresenta un buon compromesso tra il numero di cluster e la compattezza dei cluster stessi, suggerendo il numero ideale di cluster per il dataset.

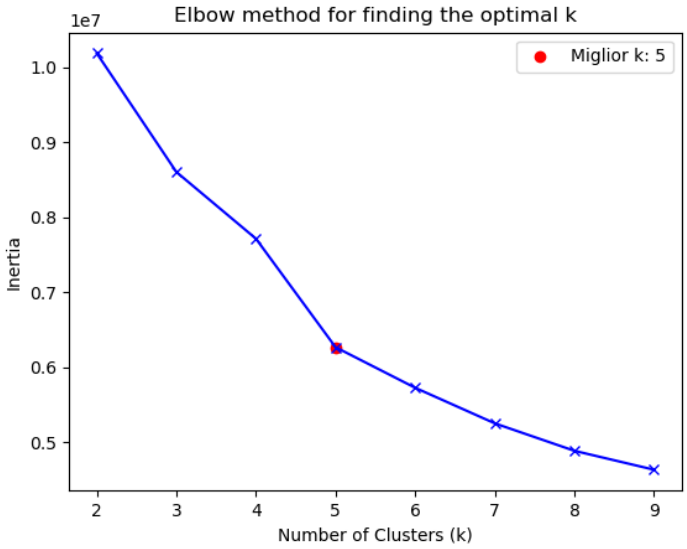
```python
# Funzione che calcola il numero di cluster ottimale per il dataset mediante il metodo del gomito
def elbowMethod(dataSet):
    inertia = []
    maxK = 10
    for i in range(2, maxK):
        kmeans = KMeans(n_clusters=i, n_init=10, init='k-means++', random_state=42)
        kmeans.fit(dataSet)
        inertia.append(kmeans.inertia_)
    kl = KneeLocator(range(2, maxK), inertia, curve="convex", direction="decreasing")

    # Visualizzo il grafico
    plt.plot(range(2, maxK), inertia, 'bx-')
    plt.scatter(kl.elbow, inertia[kl.elbow - 2], c='red', label=f'Miglior k: {kl.elbow}')
    plt.xlabel('Numero di Cluster (k)')
    plt.ylabel('Inertia')
    plt.title('Metodo del gomito per trovare il k ottimale')
    plt.legend()
    plt.show()
    return kl.elbow
```

Per quanto riguarda le scelte progettuali, ho deciso di eseguire l'algoritmo **KMeans** con valori di cluster che variano da 1 a 10, al fine di identificare il numero ottimale di cluster.

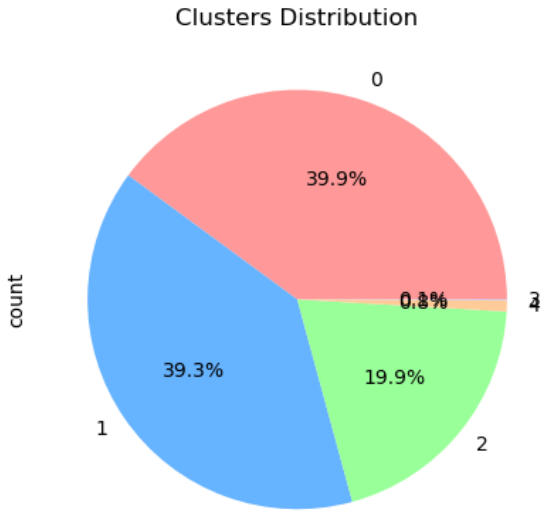
Il parametro **n_init=10** rappresenta il numero di "random restart": per ogni valore di cluster i, l'algoritmo eseguirà 10 inizializzazioni diverse e restituirà il clustering migliore in termini di **inertia** (ovvero la soluzione con la somma delle distanze quadrate più bassa tra i punti e i rispettivi centroidi).
Questa configurazione aiuta a ottenere un risultato più stabile e minimizza la probabilità di convergere a un minimo locale.

Per quanto riguarda il parametro **init='k-means++'**, questo metodo di inizializzazione seleziona i centroidi iniziali in modo da distribuire meglio i punti nello spazio, riducendo la possibilità di ottenere un clustering di bassa qualità.

Infine, ho utilizzato **random_state=42** per garantire la riproducibilità dei risultati: impostando questo valore fisso, l'algoritmo produrrà sempre lo stesso risultato ogni volta che viene eseguito, facilitando il confronto e l'analisi dei risultati.



Nel mio caso possiamo vedere come il numero ottimale di cluster è k = 5, dunque successivamente è stato eseguito l'algoritmo KMeans con 3 cluster.
Il risultato del clustering è il seguente:




```
Number of Clusters: 5
Silhouette score avg: 0.19214805658648978
```

Il risultato ha prodotto un Silhouette Score è 0.192, un valore molto vicino allo zero, che indica una bassa coesione interna ai cluster e una scarsa separazione tra essi.

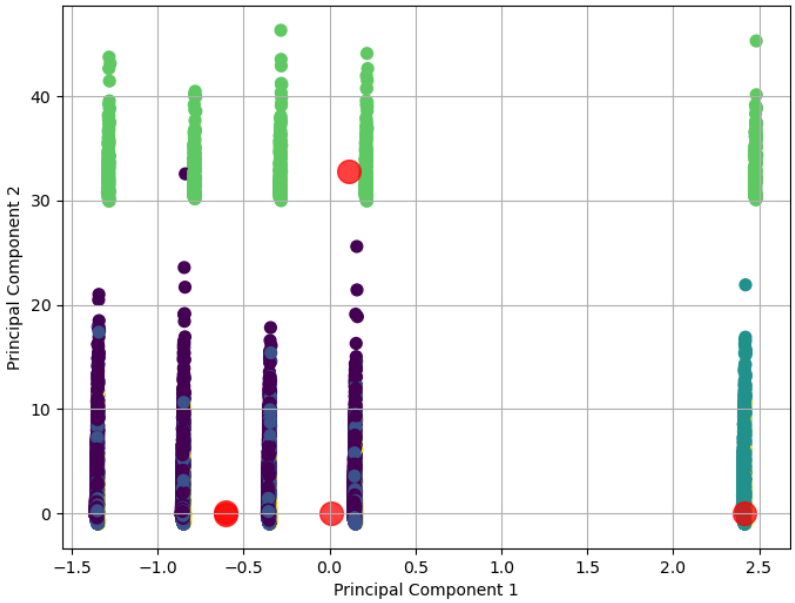
Questo punteggio suggerisce che i cluster individuati non sono ben separati né coerenti.
Il **Silhouette Score** è una metrica utilizzata per valutare la qualità di un clustering. In altre parole, ci dice quanto bene i dati sono stati divisi in cluster.

$$Silhouette\ Score\ =\ (b\ -\ a)\ /\ max(a, b)$$

- **a:** Distanza media dal proprio cluster.
- **b:** Distanza media dal cluster più vicino.

Interpretazione del punteggio:

- **Valori vicini a 1:** Il punto è ben all'interno del suo cluster e lontano dagli altri. Indica un clustering di alta qualità.
- **Valori vicini a 0:** Il punto è vicino al confine tra due cluster, suggerendo una sovrapposizione tra i gruppi.
- **Valori vicini a -1:** Il punto è probabilmente assegnato al cluster sbagliato.



I risultati ottenuti dall'algoritmo K-means evidenziano (come si può vedere in figura) una significativa sovrapposizione tra i cluster, suggerendo che i dati non sono ben separabili in gruppi distinti.
Inoltre, la presenza di centroidi sovrapposti indica che i cluster non sono stabili e potrebbero non rappresentare in modo accurato la struttura intrinseca dei dati.

Pertanto, si conclude che l'apprendimento non supervisionato non è stata la scelta più adatta per analizzare questo dataset e che quindi non ha fornito ulteriori informazioni utili da aggiungere ai dati.

Apprendimento Supervisionato

L'**apprendimento supervisionato** è una categoria dell'apprendimento automatico in cui un modello viene addestrato su un dataset etichettato, cioè un insieme di dati in cui ogni esempio è associato a una **variabile target** o **label**.
L'obiettivo dell'apprendimento supervisionato è che il modello apprenda la relazione tra i dati di input e i valori di output per poter fare previsioni accurate su nuovi dati non etichettati.

L'apprendimento supervisionato si divide in due categorie principali, in base alla natura del valore target:

- **Classificazione**: In un problema di classificazione, l'output è una **variabile categorica**. L'obiettivo è prevedere la categoria o classe a cui appartiene ciascun esempio, sulla base delle caratteristiche fornite.
- **Regressione**: Nei problemi di regressione, l'output è una **variabile continua**. In questo caso, il modello cerca di prevedere un valore numerico basato sui dati di input.

Durante l'addestramento, l'algoritmo supervisionato analizza i dati di input e impara a mappare ciascun input alla corrispondente etichetta, minimizzando l'errore tra le previsioni del modello e i valori effettivi. Una volta addestrato, il modello può fare previsioni su nuovi dati mai visti prima, cercando di generalizzare la conoscenza appresa dal set di addestramento.

L'implementazione di un modello di apprendimento automatico richiede il passaggio attraverso diverse fasi fondamentali:

1. **Scelta del modello**: La prima fase consiste nel selezionare il modello più adatto al problema da risolvere, valutando i diversi algoritmi disponibili in base alla natura dei dati e agli obiettivi.
2. **Selezione degli iperparametri**: Una volta scelto il modello, è necessario definire i suoi iperparametri, ovvero i parametri che non vengono appresi dal modello stesso, ma che devono essere impostati manualmente per ottimizzare le sue prestazioni.
3. **Addestramento**: In questa fase, il modello viene addestrato utilizzando un insieme di dati di addestramento, attraverso cui apprende la relazione tra input e output per poter fare previsioni su nuovi dati.

4. **Test**: Dopo l'addestramento, il modello viene testato su un insieme di dati separato, noto come *test set*, per verificare la sua capacità di generalizzare su dati non visti in precedenza.
5. **Valutazione delle prestazioni**: Infine, si valutano le prestazioni del modello attraverso metriche specifiche, per determinare quanto accuratamente riesce a svolgere il compito assegnato e se sono necessari miglioramenti.
6. **Confronto con altri modelli**: Infine, è possibile confrontare le prestazioni del modello con quelle di altri modelli creati (se possibile).
   Questo confronto aiuta a identificare la soluzione migliore generata per il problema, garantendo di identificare e scegliere il modello più appropriato e ottimizzato per i dati specifici.

Nel contesto del presente caso di studio, è stato adottato l'apprendimento supervisionato per affrontare un problema di classificazione.
L'obiettivo è cercare di predire se una transazione sia o meno di tipo fraudolenta per poterla identificare e quindi bloccare in modo proattivo transazioni sospette prima che possano essere processate; utilizzando feature di input selezionate durante la fase di preprocessing dei dati.
La feature target, oggetto della valutazione, è la variabile booleana "**Is Fraudulent**", che indica per l'appunto se una transazione è fraudolenta o meno.
Pertanto, si tratta di un task di classificazione booleana.

Oversampling

Come già accennato nella fase di data analysis vi era un grosso sbilanciamento tra le due classi (*Fraudulent*:5% - *Legit*:95%), e per affrontare questo problema ho deciso di applicare tecniche di *oversampling* per bilanciare le categorie e migliorare la capacità dei modelli di rilevare le transazioni fraudolente.
Ovviamente prima di poter passare alla fase di *oversampling*, ho deciso di dividere il dataset in train e test set, standardizzare i valori numerici e applicare il one-hot encoder sui dati categorici.

```
# Apprendimento supervisionato - preparazione dei dati
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Standardizzazione valori numerici
scaler = StandardScaler()
scaler.fit(X_train[numeric_columns])
X_train[numeric_columns] = scaler.transform(X_train[numeric_columns])
X_test[numeric_columns] = scaler.transform(X_test[numeric_columns])

# One-hot encoder su dati categorici
X_train = pd.get_dummies(X_train, columns=categorical_columns)
X_test = pd.get_dummies(X_test, columns=categorical_columns)
X_train.columns = X_train.columns.str.replace(' ', '_', regex=False)
X_test.columns = X_test.columns.str.replace(' ', '_', regex=False)
```

Una volta effettuata questa procedura son passato alla fase di *oversampling*, che non è stata fatta sull'intero dataset, ma bensì solo sui dati del train set; così da evitare di "*inquinare*" i dati del test set.

```
# Oversampling
from imblearn.over_sampling import SMOTE
smote = SMOTE(sampling_strategy=0.5, random_state=42)
X_resampled, y_resampled = smote.fit_resample(X_train, y_train)

# Riduzione test-set
sample_size = 1000000
X_reduced = X_resampled.sample(n=sample_size, random_state=42)
y_reduced = y_resampled.loc[X_reduced.index]

# Riassegno index train-set
X_reduced = X_reduced.reset_index(drop=True)
y_reduced = y_reduced.reset_index(drop=True)

# Riassegno valori variabili a X_train, y_train
del X_train, y_train
X_train = X_reduced
y_train = y_reduced
```

Inizio applicando la tecnica di oversampling **SMOTE (Synthetic Minority Over-sampling Technique)** al dataset di training.
Questa tecnica è particolarmente utile quando si ha a che fare con dataset sbilanciati, ovvero quando una classe è significativamente più rappresentata rispetto all'altra.

Come funziona *SMOTE*?

1. **Identificazione dei vicini:** Per ogni esempio della classe minoritaria, vengono identificati i suoi vicini più "*vicini*".
2. **Generazione di nuovi esempi:** Per ogni vicino, viene generato un nuovo esempio sintetico lungo il segmento di linea che collega il punto originale al suo vicino.
3. **Bilanciamento del dataset:** Questo processo viene ripetuto fino a raggiungere il rapporto desiderato tra le classi.

Nel codice, come si può notare, il parametro *sampling_strategy* impostato a 0.5, questo parametro corrisponde al rapporto desiderato tra il numero di campioni della classe di minoranza e il numero di campioni della classe di maggioranza dopo il ricampionamento.
Ho deciso di non equiparare il numero di dati per le due categoriche poiché mi sembrava eccessivo passare da un rapporto 95% - 5% a un rapporto 50% - 50%.

Dopo l'oversampling, il dataset di training era diventato troppo grande per essere analizzato, circa 2mln di elementi, come si può vedere dall'immagine.

```
Number of elements:
 is_fraudulent
0    1119285
1     559642
Name: count, dtype: int64
Percentages:
 is_fraudulent
0    0.666667
1    0.333333
Name: count, dtype: float64
```

Per migliorare l'efficienza computazionale, si ho deciso di ridurre il dataset mantenendo una buona rappresentazione delle classi.
In questo caso, viene campionato un sottoinsieme di 1mln di esempi dal *dataset oversampled*.

```
Number of elements (after reduction):
 is_fraudulent
0    666712
1    333288
Name: count, dtype: int64
Percentages (after reduction):
 is_fraudulent
0    0.666712
1    0.333288
Name: count, dtype: float64
```

Infine, gli indici del dataset ridotto vengono riassegnati per garantire una sequenza numerica consecutiva.

Scelta e addestramento dei modelli

Per questo progetto i modelli di apprendimento supervisionato che sono stati selezionati per il confronto nel task di classificazione booleana sono tre, ognuno con caratteristiche ben definite, e sono elencati di seguito.

Inoltre, è stato scelto per ogni modello presentato un insieme di iperparametri diversi da valutare per trovare le combinazioni migliori.

Random Forest

Il **Random Forest** è un algoritmo di apprendimento supervisionato che appartiene alla famiglia degli algoritmi di *ensemble*. Esso combina la potenza predittiva di numerosi alberi decisionali, ciascuno costruito su un sottoinsieme casuale dei dati e delle features.

Come funziona?

1. **Creazione di alberi decisionali multipli:** Ogni albero viene costruito su un sottoinsieme casuale dei dati e su un sottoinsieme casuale delle features.
2. **Predizione:** Quando si presenta un nuovo dato, ogni albero fornisce una predizione. La predizione finale del Random Forest è determinata dalla maggioranza dei voti degli alberi.

Quali sono gli iperparametri scelti?

```
params = {
    'n_estimators': randint(low=10, high=300),
    'max_depth': randint(low=4, high=12),
    'min_samples_split': randint(low=2, high=10),
    'min_samples_leaf': randint(low=1, high=8)
}
```

- *n_estimators*: con valori da un minimo di 10 ad un massimo di 300. Rappresenta il numero di alberi decisionali che compongono la *foresta.*
- *max_depth*: con valori da un minimo di 4 ad un massimo di 12. Definisce la profondità massima di ogni albero decisionale.
- *min_samples_split*: con valori da un minimo di 2 ad un massimo di 10. Indica il numero minimo di campioni (osservazioni) richiesti per dividere un nodo interno durante la costruzione dell'albero.
- *min_samples_split*: con valori da un minimo di 1 ad un massimo di 8. *Definisce il numero minimo di campioni richiesti in ogni foglia dell'albero.*

Gradient Boosting

Il **Gradient Boosting** è un altro algoritmo di apprendimento supervisionato di tipo *ensemble*. A differenza del Random Forest, gli alberi decisionali nel Gradient Boosting vengono costruiti in modo sequenziale, con ogni nuovo albero che cerca di correggere gli errori commessi dagli alberi precedenti.

Come funziona?

1. **Inizializzazione:** Viene creato un modello base (ad esempio, un albero decisionale con una profondità minima).
2. **Iterazioni successive:** Ad ogni iterazione, viene costruito un nuovo albero decisionale che cerca di minimizzare l'errore residuo del modello precedente. Gli alberi successivi vengono aggiunti al modello in modo ponderato.
3. **Predizione:** La predizione finale è la somma ponderata delle predizioni di tutti gli alberi.

Quali sono gli iperparametri scelti?

```
params = {
    'n_estimators': randint(low=100, high=300),
    'learning_rate': [0.1, 0.15, 0.2],
    'max_depth': randint(low=4, high=12),
    'subsample': [0.5, 0.7, 1.0]
}
```

- *n_estimators*: con valori da un minimo di 100 ad un massimo di 300.
  Indica il numero di stimatori (alberi decisionali) che verranno aggiunti al modello in modo sequenziale.
- *learning_rate*: con valori fissi = [0.1, 0.15, 0.2].
  Controlla il contributo di ogni stimatore successivo al modello.
  Un valore basso significa che ogni stimatore apprende lentamente e corregge gradualmente gli errori del modello precedente.
- *max_depth*: con valori da un minimo di 4 ad un massimo di 12.
  Definisce la profondità massima di ogni albero decisionale.
- *subsample*: con valori fissi = [0.5, 0.7, 1.0].
  Indica la frazione di osservazioni utilizzate per addestrare ogni stimatore.
  Un valore inferiore a 1 introduce una forma di bagging, riducendo la varianza e prevenendo l'overfitting.

Logistic Regression

La **Regressione Logistica** è un algoritmo di apprendimento supervisionato utilizzato principalmente per problemi di classificazione binaria. Sebbene il nome suggerisca una regressione, in realtà modella la probabilità che un'osservazione appartenga a una particolare classe.

Spesso molto utilizzato per problemi di classificazione binaria, particolarmente efficace quando la variabile dipendente è booleana

Come funziona?

- **Funzione logistica:** Utilizza la funzione logistica (o sigmoid) per mappare i valori previsti a una probabilità compresa tra 0 e 1.
- **Soglia di decisione:** Viene fissata una soglia per classificare le osservazioni.

Quali sono gli iperparametri scelti?

```
params = {
    'penalty' :['l2'],
    'C': uniform(0.01, 10),
    'max_iter': [1000, 10000, 15000]
}
```

- penalty: regolarizzazione L2, è una tecnica utilizzata per prevenire l'overfitting. Aggiunge una penalità alla funzione di costo del modello, basata sulla somma dei quadrati dei coefficienti dei pesi. Questo aiuta a ridurre la complessità del modello e a migliorare la sua capacità di generalizzare a nuovi dati.
- C: con valori da un minimo di 0.01 ad un massimo di 10.
  Questo parametro controlla la forza della regolarizzazione L2.
  Un valore più basso di C indica una maggiore regolarizzazione, mentre un valore più alto indica una minore regolarizzazione.
- max_inter: con valori fissi = [1000, 10000, 15000].
  Questo parametro specifica il numero massimo di iterazioni che l'algoritmo di ottimizzazione può eseguire durante l'addestramento del modello.

Regolazione degli iperparametri

La regolazione degli iperparametri svolge un ruolo cruciale nell'ottimizzazione dei modelli di apprendimento automatico per ottenere prestazioni migliori.
Si tratta di selezionare la migliore combinazione di valori degli iperparametri che producono la massima accuratezza (o anche l'errore più basso).
La *GridSearch* e la *RandomSearch* sono due tecniche popolari utilizzate per la regolazione degli iperparametri.

In questo progetto ho deciso di usare la **RandomSearchCV:**
come suggerisce il nome, esplora lo spazio degli iperparametri campionando casualmente i valori da distribuzioni o intervalli predefiniti; combinandola con la tecnica di *K-Fold Cross Validation* (CV).

A differenza della *GridSearch*, la *RandomSearch* non copre sistematicamente l'intero spazio dei parametri.
Si concentra invece su combinazioni di valori di iperparametri selezionati in modo casuale. La *RandomSearch* offre maggiore flessibilità ed efficienza quando si ha a che fare con un gran numero di iperparametri o quando i valori ottimali dei parametri sono meno intuitivi o noti.
Può coprire una gamma più ampia di valori e può convergere a risultati migliori più rapidamente della *GridSearch*, soprattutto nei casi in cui pochi iperparametri hanno un impatto significativo sulle prestazioni.

Nella *K-Fold CV* il dataset viene diviso in *k fold* (insiemi disgiunti) e il modello viene addestrato *k* volte.
Per ogni iterazione un solo fold viene usato per il testing mentre gli altri *k-1* fold vengono utilizzati per il training.
In questo modo è possibile testare e addestrare il modello su dati diversi per comprendere la robustezza della valutazione delle prestazioni del modello nei diversi casi che si sviluppano.

Nel mio caso ogni modello è stato addestrato con un valore di *k* = 4 (4 fold) e n_iter = 50, ovvero indica che il numero di iterazioni che l'algoritmo *RandomSearch* eseguirà sarà esattamente 50.

```
crf = dcv.RandomizedSearchCV(estimator=cuml.ensemble.RandomForestClassifier(),
                             param_distributions=params,
                             n_iter=50,
                             cv=4,
                             return_train_score=True,
                             random_state=42)
```

Esempio di come i parametri del RandomizedSearchCV sono stati impostati (cv=4 e n_iter=50)

Durante la validazione incrociata, è stata scelta l'accuratezza, o *accuracy*, come metrica principale per valutare le performance dei modelli.
L'accuratezza, misura la percentuale di classificazioni corrette rispetto al totale delle classificazioni effettuate, e viene massimizzata per identificare il miglior modello.
Per ciascun modello, il valore medio dell'accuratezza è stato calcolato come la media delle accuratezze ottenute in tutte le iterazioni della K-Fold Cross Validation.

Come accennato in precedenza, ho inizialmente diviso il dataset in un insieme di addestramento e uno di test.
Successivamente, ho applicato la validazione incrociata **esclusivamente** all'insieme di addestramento.
L'insieme di test, invece, è stato utilizzato come un **set di valutazione finale** e indipendente, per ottenere una stima più affidabile delle prestazioni del modello migliore.
In questo modo ho potuto ottenere una stima più affidabile di come il modello si comporta su dati mai visti.
Le materiche usate in questa ulteriore fase di valutazione solo le seguenti:

- *accuracy (A)*: rappresenta la proporzione di classificazioni corrette rispetto al numero totale di esempi.
$$accuracy = (TP + TN) / (TP + TN + FP + FN)$$
- *precision (B)*: misura la proporzione di esempi classificati come positivi che sono effettivamente positivi.
$$precision = TP / (TP + FP)$$
- *recall (R): misura la proporzione di esempi positivi che sono stati correttamente identificati dal modello.*
$$recall = TP / (TP + FN)$$
- *F1-Score (F1)*: è la media armonica di precisione e richiamo, fornendo una misura bilanciata delle prestazioni del modello.
$$F1 - Score = 2 * (Precision * Recall) / (Precision + Recall)$$

TP (True Positive): Esempi classificati correttamente come positivi.
TN (True Negative): Esempi classificati correttamente come negativi.
FP (False Positive): Esempi classificati erroneamente come positivi.
FN (False Negative): Esempi classificati erroneamente come negativi.

Inoltre, è stata calcolata la *deviazione standard* dell'accuratezza per ciascun modello per valutare la variabilità dei risultati. In questo modo è stato possibile ottenere non solo una misura della performance media, ma anche una valutazione della consistenza e della stabilità del modello attraverso le diverse suddivisioni e ripetizioni.

Analisi dei risultati

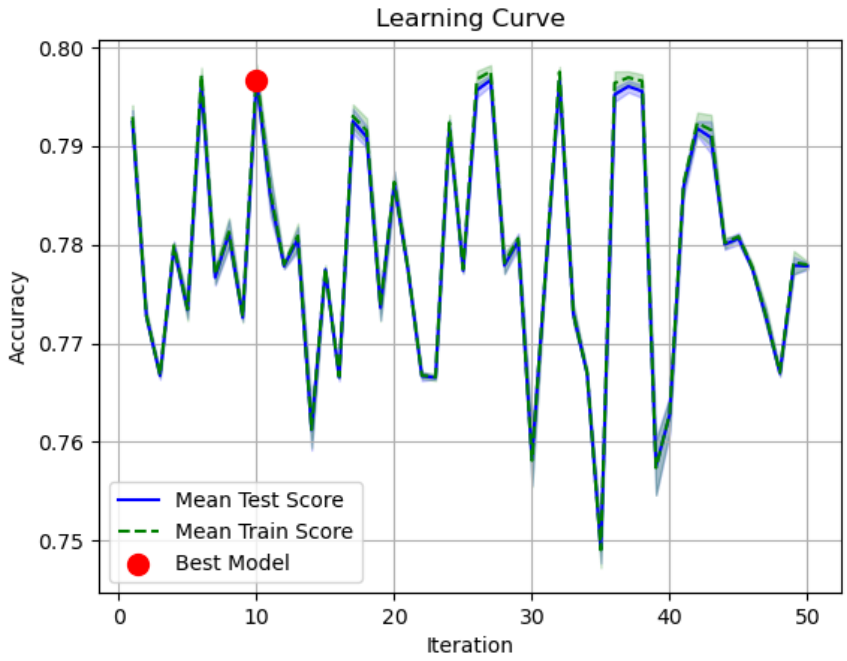
Una volta completata la fase di addestramento e ottimizzazione dei modelli, il passo successivo è stata l'analisi dei risultati ottenuti per poter quindi selezionare il miglior modello tra quelli testati.
Per fare ciò, bisogna interpretare le metriche di performance raccolte durante la fase di addestramento e test.

Tutti i risultati ottenuti è possibile ritrovarli nella directory */Results*.

Modello	Configurazione migliore	$\bar{x}$ e ρ della accuratezza	Differenza Mean Score Test CV Train	Metriche EVAL test set *"aggiuntivo"*
Random Forest	'max_depth': 11, 'min_samples_leaf': 4 'min_samples_split': 5 'n_estimators': 280	Train set CV $\bar{x}$ = 0.7975 ρ = 0.0006 Test set CV $\bar{x}$ = 0.7967 ρ = 0.0005	-0.0008	A = 0.9070 P = 0.2428 R = 0.4037 F1 = 0.3032
Gradient Boosting	'learning_rate': 0.2 'max_depth': 11 'n_estimators': 288 'subsample': 0.5	Train set CV $\bar{x}$ = 0.9705 ρ = 0.0003 Test set CV $\bar{x}$ = 0.9502 ρ = 0.0005	-0.0202	A = 0.9507 P = 0.5204 R = 0.2195 F1 = 0.3088
Logistic Regression	'C': 2.92 'max_iter': 15000 'penalty': 'l2'	Train set CV $\bar{x}$ = 0.7691 ρ = 0.0001 Test set CV $\bar{x}$ = 0.7691 ρ = 0.0005	2.3245e-06	A = 0.8911 P = 0.2197 R = 0.4598 F1 = 0.2974

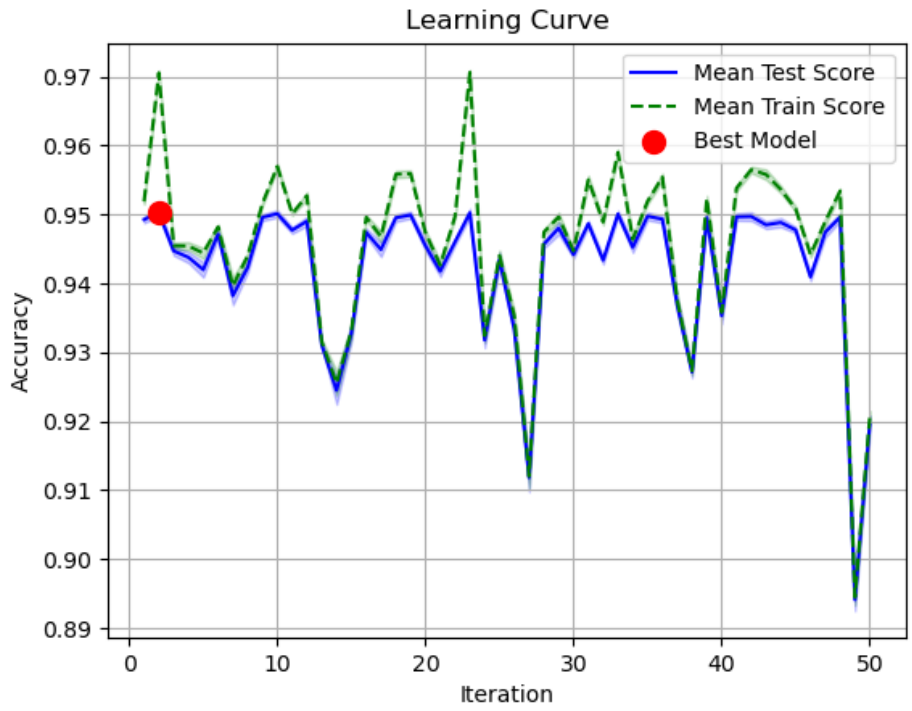
Valutiamo ora singolarmente i risultati ottenuti per comprendere quale modello è migliore tra quelli analizzati.

Analisi risultati del modello Random Forest



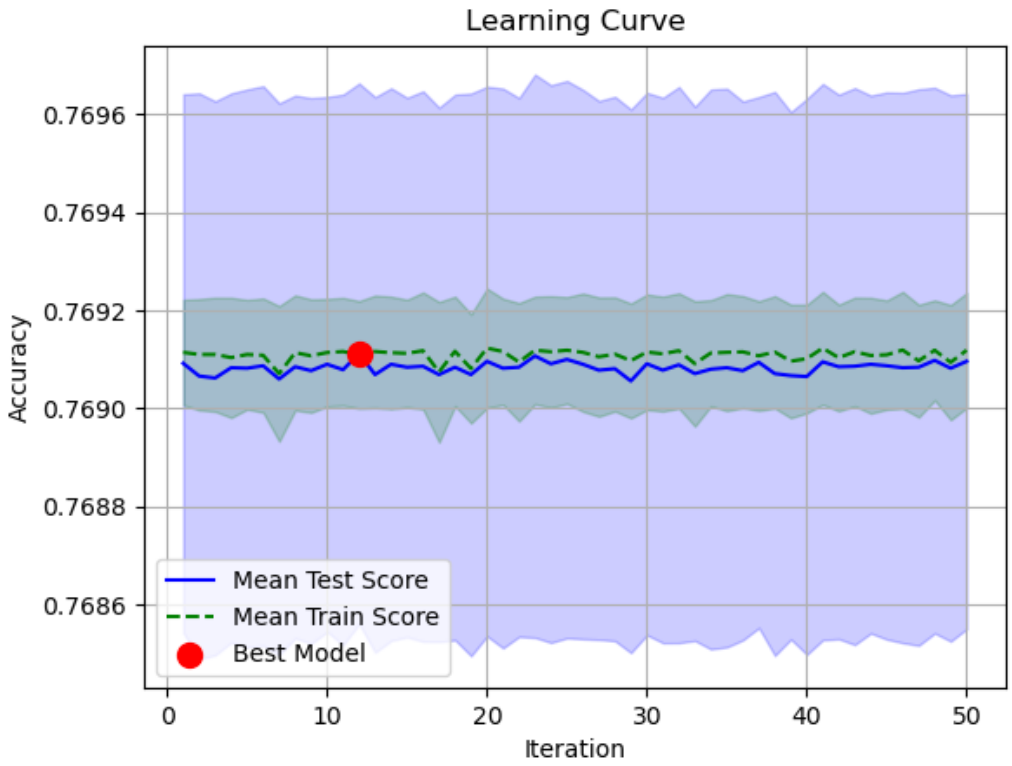
- **Accuratezza sul Test Set**: 0.9070, buon risultato per essere su dati mai visti.
- **Precisione**: 0.2428, un valore basso, il che indica che il modello ha una tendenza ad avere molti falsi positivi.
- **Recall**: 0.4037, valore basso, indica quindi che il modello non riesce a catturare tutti i veri positivi (rileva meno della metà dei casi positivi).
- **F1 Score**: 0.3032, valore basso e richiama a conferma il fatto che il modello ha delle difficoltà a bilanciare precisione e recall.
- **Accuratezza CV**: l'accuratezza del modello in fase di training è stata del 0.7975, mentre l'accuratezza media sul Test set CV è stata di 0.7967.
  Questa somiglianza tra le performance sui due set è un buon segno, poiché suggerisce che il modello non soffre di overfitting e generalizza bene sui dati di test.
- **Varianza e Deviazione Standard**: le basse varianze e deviazioni standard indicano che le prestazioni del modello sono molto consistenti tra i vari fold della cross-validation, mostrando quindi una stabilità nelle previsioni.
- La **differenza media tra le performance di training e test**: è molto piccola (circa -0.0008), suggerendo una buona generalizzazione.
  Il modello *non* sta sovradattando i dati di addestramento.

Analisi risultati del modello Gradient Boosting



- **Accuratezza sul Test Set**: 0.9507, questo è un miglioramento rispetto al modello *Random Forest* (0.9070), indicando una buona capacità di classificare correttamente la maggior parte dei campioni di test.
- **Precisione**: 0.5204, valore piu' alto rispetto al modello *Random Forest* (0.2428), indicando una riduzione dei falsi positivi.
  Il modello è quindi più affidabile nelle previsioni positive.
- **Recall**: 0.2195, il recall è inferiore rispetto al modello precedentemente analizzato (0.4037), quindi questo modello non riesce a identificare bene molti veri positivi.
- **F1 Score**: 0.3088, l'F1 score è simile al *Random Forest* (0.3032), suggerendo che, nonostante il miglioramento in precisione, la capacità di bilanciare tra precisione e richiamo non è molto migliorata.
- **Accuratezza CV**: l'accuratezza del modello in fase di training è stata del 0.9705, che è leggermente più alta rispetto all'accuratezza di Test CV 0.950, mostrando una leggera tendenza al sovradattamento; ma comunque un buon equilibrio.
- **Varianza e Deviazione Standard**: le basse varianze e deviazioni standard indicano che le prestazioni del modello sono molto consistenti tra i vari fold della cross-validation, mostrando quindi una stabilità nelle previsioni.
- La **differenza media tra le performance di training e test**: -0.0202, il che indica che l'accuratezza sul test set è solo di poco inferiore a quella sul training set, il che è positivo per la generalizzazione.

Analisi risultati del modello Logistic Regression



- **Accuratezza sul Test Set**: 0.8911, inferiore rispetto al modello precedentemente analizzato (*Gradient Boosting*: 0.951), ma comunque piuttosto buona.
- **Precisione**: 0.2197, piuttosto bassa, suggerendo che il modello classifica in modo errato molti esempi come positivi.
- **Recall**: 0.4598, più alto rispetto ai modelli precedentemente analizzati. Questo indica che il modello è in grado di catturare una parte maggiore dei veri positivi.
- **F1 Score**: 0.2974, simile agli altri modelli analizzati, confermando che anche in questo caso la capacità di bilanciare tra *precisione* e *recall* non è particolarmente elevata
- **Accuratezza CV**: 0.7691 molto simile, praticamente uguale, all'accuratezza media del Test-set CV (0.7691), suggerendo che il modello è ben bilanciato tra training e test e non tende a sovradattarsi.
- **Varianza e Deviazione Standard**: Molto basse per entrambi i set, quindi le prestazioni sono consistenti tra i vari fold della cross-validation.
- La **differenza media tra le performance di training e test**: praticamente nulla (2.32e-06), il che indica una generalizzazione eccellente, quasi senza differenze di accuratezza tra training e test.

Analisi migliore modello individuato

Riassunto breve delle considerazioni per ogni modello sviluppato:

- *Random Forest*: questo modello presenta una buona accuratezza e un richiamo accettabile, ma una precisione bassa, risultando in molti falsi positivi.
- *Gradient Boosting*: questo modello ha l'accuratezza più alta e una precisione migliorata rispetto agli altri modelli, ma il richiamo è piuttosto basso, quindi perde molti veri positivi.
- *Logistic Regression*: questo modello ha un buon bilanciamento tra accuratezza e richiamo, con il valore di richiamo più alto tra i tre modelli.
  Ma una precisiome molto bassa, il che porta a molti falsi negativi.

Pertanto, ho pensato di "etichettare" come **migliore modello** sviluppato il modello **Gradient Boosting**, poiché si è dimostrato il migliore in termini di accuratezza e precisione, suggerendo un'elevata capacità di ridurre i falsi positivi.
Questo è molto importante nel contesto delle frodi, poiché una bassa precisione può comportare un numero eccessivo di avvisi falsi, il che aumenta i costi operativi e potrebbero far perdere fiducia agli utenti.
Il modello evidenziato, inoltre, ha ottenuto il punteggio di F1 score migliore tra tutti, ovvero una buona mediazione tra precisione e il recall.

Rete Neurale

Una rete neurale è un programma di machine learning, o modello, che prende decisioni in modo simile al cervello umano, utilizzando processi che imitano il modo in cui i neuroni del cervello umano lavorano insieme per identificare fenomeni o arrivare a conclusioni.

Ogni rete neurale è costituita da nodi e archi: un livello di input, uno o più livelli nascosti e un livello di output.
Ogni nodo si connette ad altri nodi e ha il suo peso e una sua soglia associati.
Se l'output di qualsiasi singolo nodo è al di sopra del valore di soglia specificato, tale nodo viene attivato, inviando i dati al livello successivo della rete.
In caso contrario, non viene passato alcun dato al livello successivo della rete.

Le reti neurali si basano su dati di addestramento per imparare e migliorare la loro precisione nel tempo.
Una volta ottimizzato, sono strumenti potenti nel campo dell'informatica e dell'intelligenza artificiale, che consentono di classificare e raggruppare i dati ad alta velocità.

Quando si parla di reti neurali, due concetti fondamentali sono la **classificazione** e la **regressione**.

- **Classificazione:** la rete neurale deve assegnare un'etichetta (o una classe) a un dato input.
- **Regressione:** la rete neurale deve prevedere un valore numerico continuo.

Nel contesto di studio parliamo ovviamente di una rete neurale dedicata alla classificazione.

Apprendimento:

- **Backpropagation:** l'algoritmo più comunemente utilizzato per addestrare le reti neurali (usato anche nel caso di studio).
  Esso calcola l'errore tra la predizione della rete e l'etichetta corretta (fraudolenta o legittima) e aggiorna i pesi dei nodi in modo da ridurre tale errore.
- **Ottimizzatori:** come, ad esempio, Adam (usato nel caso di studio), vengono utilizzati per guidare il processo di ottimizzazione e trovare i pesi ottimali della rete.

Creazione del modello

```python
# Definizione modello
class BinaryClassifier(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(BinaryClassifier, self).__init__()
        self.linearLayer1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(0.2)
        self.linearLayer2 = nn.Linear(hidden_size, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        out = self.linearLayer1(x)
        out = self.relu(out)
        out = self.dropout(out)
        out = self.linearLayer2(out)
        out = self.sigmoid(out)
        return out
```

Come si può osservare dal codice esso è composto da due parti principali:
una dichiarativa (*init*) dove possiamo trovare i componenti principali del modello; e una sezione dove viene definita la ***forward propagation***, durante il quale i dati di input attraversano la rete (e quindi i suoi strati) per poi generare un output finale.

Ho pensato di creare una rete neurale "*semplice*" e di dimensioni ridotte per poter catturare al meglio le informazioni e non essere troppo dispersiva, e non far andare il modello quindi in *overfitting*.

Iniziamo dal primo layer di tipo *Linear*, questo livello applica una trasformazione lineare agli input, calcolando una combinazione lineare degli input, seguito successivamente da una funzione di attivazione ***reLU***.

Successivamente, una parte dei neuroni viene disattivata attraverso il **dropout** (per evitare *overfitting*), una tecnica che disattiva casualmente una percentuale (in questo caso il 20%) dei neuroni durante ogni passo dell'addestramento.
I dati rimanenti passano al livello successivo (di tipo lineare) che produce un singolo valore finale.
Questo valore passa infine attraverso la funzione ***sigmoid***, che restituisce una probabilità, la quale viene utilizzata per classificare l'input in una delle due classi previste.

Ciascuna di queste scelte contribuisce a rendere il modello efficace e robusto, senza eccedere in complessità o richiedere risorse computazionali molto elevate.

Layer di tipo Linear

Un livello lineare, noto anche come livello pienamente connesso (**fully connected**), rappresenta il cuore delle operazioni di base di una rete neurale.
In un layer lineare (generalmente), ogni neurone è collegato a tutti i neuroni del livello precedente.
La trasformazione lineare applicata da questo layer è descritta dall'equazione:

$$output = input \times W^T + b$$

dove:

- **input** è un vettore di dati in ingresso (in questo caso le features identificate)
- **W** è una matrice di **pesi** appresi dalla rete durante l'addestramento, che rappresentano l'importanza di ciascun input.
- **b** è un vettore di **bias** che consente di spostare la funzione di attivazione lungo l'asse delle ordinate, migliorando la capacità di apprendimento.

Il livello lineare è fondamentale perché permette alla rete di imparare relazioni tra i dati: modifica e ottimizza i pesi e i bias per minimizzare l'errore.

Funzione di Attivazione ReLU

Dopo un livello lineare, è comune applicare una **funzione di attivazione** per introdurre non linearità, e **ReLU (Rectified Linear Unit)** è una delle più popolari.
La funzione ReLU è definita come:

$$ReLU(x) = max(0, x)$$

Questa funzione restituisce zero per tutti i valori negativi di *x* e mantiene i valori positivi.

ReLU riduce anche il rischio di **vanishing gradient**, ossia un problema in cui i gradienti dei livelli precedenti diventano molto piccoli durante la backpropagation, rallentando o addirittura bloccando l'apprendimento.
Con ReLU, i gradienti tendono a rimanere più stabili, accelerando l'addestramento e migliorando la capacità della rete di apprendere pattern complessi.

Dropout

Dropout è una tecnica di regolarizzazione utilizzata per ridurre il rischio di **overfitting**. Durante l'addestramento di una rete neurale, il dropout disattiva casualmente una percentuale dei neuroni del livello, rendendoli inattivi per quel ciclo di addestramento.

L'idea alla base del dropout è quella di impedire alla rete di dipendere troppo da una specifica combinazione di neuroni. Rimuovendo casualmente i neuroni, si forza la rete a trovare più modi per rappresentare l'informazione, aumentando la capacità di generalizzazione del modello.
Durante il test, il dropout viene disattivato e la rete utilizza tutti i neuroni, ma i pesi sono scalati per compensare la fase di training.

Funzione Sigmoid

La **funzione sigmoid** è un'altra funzione di attivazione, spesso usata nell'ultimo livello delle reti neurali per compiti di classificazione binaria. La sigmoid mappa i valori in input su un intervallo compreso tra 0 e 1, secondo la formula:

$$sigmoid(x) = \frac{1}{e + x^{-1}}$$

Questo rende la funzione *sigmoid* particolarmente utile quando si vuole ottenere una probabilità come output: se la sigmoid produce un valore vicino a 1, significa che la rete è propensa a classificare l'input come appartenente alla classe positiva (1); se invece il valore è vicino a 0, la rete propende per la classe negativa (0).

In problemi di classificazione binaria, la funzione sigmoid è dunque ideale per convertire i valori in probabilità, rendendo semplice interpretare la previsione della rete.

Iperparametri

Gli iperparametri sono parametri che non vengono appresi dal modello durante il processo di addestramento, ma devono essere impostati prima dell'inizio dell'addestramento della rete.
La scelta appropriata degli iperparametri è molto importante per il buon funzionamento della rete neurale.

```
# Iperparametri
input_size = X_train.shape[1]
hidden_size = 64
num_epochs = 5
learning_rate = 0.01
```

1. input_size: questo parametro rappresenta il numero di caratteristiche (o features) nei dati di input.
In questo caso, è impostato su *X_train.shape[1],* ovvero il numero di colonne (features) nel DataFrame.

2. hidden_size: parametro che rappresenta il numero di neuroni nel primo layer nascosto della rete (nel mio caso 64).

3. num_epochs: parametro che indica il numero di epoche, ovvero quante volte il modello passerà attraverso l'intero dataset durante il processo di addestramento. In questo caso, ho impostato a 5.

4. learning_rate: questo parametro definisce la velocità con cui il modello impara durante l'ottimizzazione. In questo caso, ho impostato a 0.01.

Addestramento

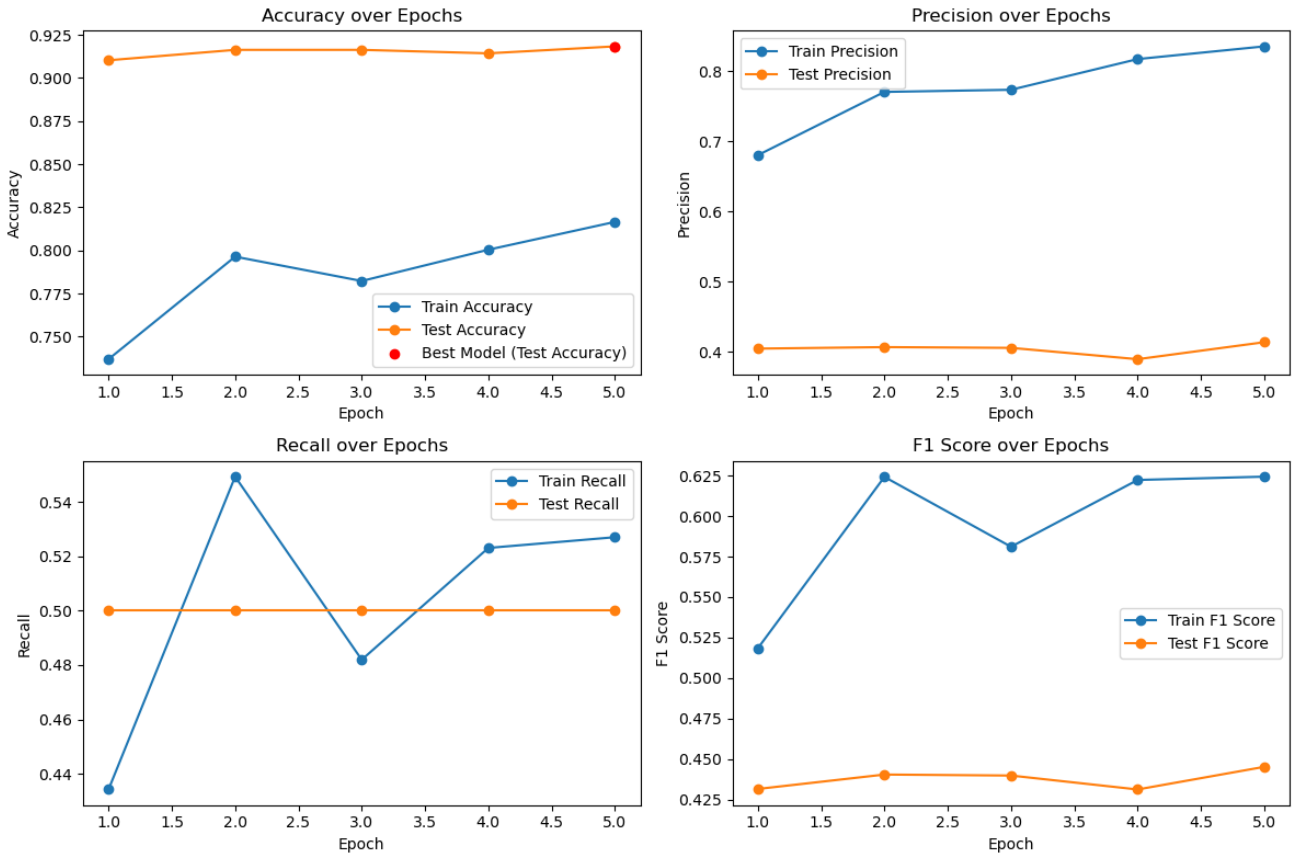
Il ciclo principale di addestramento si sviluppa su più epoche, che rappresentano il numero totale di passaggi attraverso il dataset di addestramento. In ogni epoca, vengono eseguiti i seguenti passaggi:

1. **Inizializzazione delle Metriche**: per ogni epoca, si inizializzano liste vuote per raccogliere le metriche usate per analizzare in passi successivi i dati di train e test.
   Queste metriche includono *accuratezza*, *precisione*, *recall* e *F1-score*.
2. **Caricamento dei Dati**: viene utilizzato un *DataLoader* per iterare sui dati di addestramento, che fornisce in batch gli input e le etichette corrispondenti.
3. **Ottimizzazione e Aggiornamento dei Pesi**:
   - **Azzeramento dei Gradienti**: prima di ogni aggiornamento, si azzerano i gradienti accumulati dal passo precedente.
   - **Forward Propagation**: gli input vengono passati attraverso il modello per ottenere le previsioni.
   - **Calcolo del Loss**: viene calcolata il loss confrontando le previsioni con le etichette reali utilizzando una funzione di perdita appropriata (in questo caso: *BCELoss*).
   - **Backward Propagation**: viene eseguita la *propagazione inversa* per calcolare i gradienti.
   - **Aggiornamento dei Pesi**: i pesi del modello vengono aggiornati utilizzando l'ottimizzatore (Adam in questo caso).
4. **Valutazione Intermedia**: ogni mille batch (in questo caso), il modello viene messo in modalità *"valutazione"* per poter calcolare le metriche di prestazione sui dati di addestramento e test:

- Gli output vengono confrontati con le etichette reali per calcolare le metriche: *accuratezza*, *precisione, recall* e *F1-score* (già ampiamente discusse precedentemente).

5. **Aggiornamento delle Medie**: alla fine di ogni epoca, vengono calcolate le medie delle metriche raccolte e stampate a video per entrambi i set di dati (train e test).
6. **Salvataggio del Miglior Modello**: dopo ogni epoca, inoltre, la media dell'accuratezza sul *test set* viene confrontata con la migliore accuratezza registrata.
   Se la nuova accuratezza è migliore, viene salvato lo stato corrente del modello come miglior modello e i suoi relativi parametri.

Analisi risultati e considerazioni



Per avere una metrica di confronto anche con i modelli creati nelle fasi precedenti del progetto, anche qui ho optato per la metrica della accuratezza come “principale” per la scelta del modello finale.

Anche in questo caso, come discusso precedentemente, ho calcolato l’accuracy, la precisione, recall e F1-score (anche del train set, in questo caso).

Dati modello migliore individuato durante la fase di training:

	Accuracy	Precision	Recall	F1 Score
Train	0.8165	0.8350	0.5271	0.6244
Test	0.9183	0.4140	0.5000	0.4452

Risultati ottenuti: un buon modello, accuracy molto più alta rispetto al train set (+10%), una precisione poco sotto al 50% e un recall pari al 50%.

- **Accuratezza (Test)**: 0.9183 abbastanza alta, suggerendo che il modello è ben bilanciato tra training e test e non tende a sovradattarsi.
- **Precisione**: 0.4140, pochino sotto la media, suggerendo che il modello classifica in modo errato alcuni esempi come positivi.
- **Recall**: 0.5, nella media. Questo indica che il modello è in grado di catturare una parte maggiore dei veri positivi.
- **F1 Score**: 0.4452, pochino sotto la media, indicando quindi una leggera capacità da parte del sistema di bilanciare tra *precisione* e *recall*.

Mettiamolo a confronto con il modello del ***Gradient Boosting***, sviluppato nella fase precedente:

	Accuracy	Precision	Recall	F1 Score
Train	0.9705	/	/	/
Test - EVAL	0.9507	0.5204	0.2195	0.3088

Possiamo notare un netto distacco nel *recall* da parte del modello neurale, circa 21% (*Gradient Boosting*) contro il 50% del modello neurale, il che lo porta ad avere anche un punteggio superiore nella metrica F1.
Per quanto riguarda invece l'accuracy e precisione nei sistemi, il *Gradient Boosting* vince la competizione.

Quindi il modello sviluppato, sicuramente è possibile ritenerlo valido, rispetto anche agli altri modelli proposti, e può essere consigliato al posto del *Gradient Boosting* quando si preferisce avere un recall superiore a discapito di una possibile minore precisione e accuratezza da parte del sistema.

In conclusione, se dovessi scegliere quale modello "premiare al primo posto", opterei per la rete neurale.
Questo perché rappresenta una soluzione più flessibile e facilmente manutenibile nel tempo, rispetto al Gradient Boosting.

Le reti neurali, infatti, offrono la possibilità di adattarsi a una vasta gamma di problemi e possono essere aggiornate o modificate con relativa semplicità, rendendole particolarmente vantaggiose in scenari in continua evoluzione, come nel contesto in questione (le frodi digitali).

Conclusioni e sviluppi futuri

Il progetto da un punto di vista funzionale, potrebbe essere molto utile quindi a identificare un problema nel sistema di e-commerce (ad esempio) prima ancora che si presenti e che quindi una transazione fraudolenta possa andare in porto.
Il sistema potrebbe relazionarsi anche con una intelligenza artificiale (LLM) che potrebbe trovare e individuare nuovi pattern nascosti, migliorando ulteriormente le prestazioni del modello.
Sarebbe possibile migliorare ulteriormente il modello tramite l'ingegnerizzazione delle caratteristiche, come l'inserimento di variabili derivate che tengano conto di pattern temporali, frequenza delle transazioni, e comportamenti specifici dei clienti nel sistema.
Sarebbe auspicabile provare il progetto lavorando con un dataset contenenti dati reali, provenienti ad esempio da una azienda di e-commerce, così da poter effettivamente confermare i risultati, o migliorare i modelli creati se necessario.