Introduzione a Cybersecurity e AI

Franco Arcieri

KNN

- È la domanda fondamentale per capire come "ragiona" il KNN.
  - K è il numero di "vicini" a cui l'algoritmo chiede consiglio prima di prendere una decisione.
  - Immagina che il KNN sia un giudice che deve decidere se un imputato (il nuovo dato) è colpevole o innocente.
- K=1: "Il Dittatore"
  - Quando imposti n_neighbors=1, l'algoritmo guarda solamente il punto più vicino in assoluto e ne copia l'etichetta.
  - "Chi è il tuo vicino più prossimo? È un cerchio rosso? Allora anche tu sei un cerchio rosso. Non mi interessa chi altro c'è intorno."
  - Overfitting: È estremamente sensibile al rumore.
  - Esempio Cyber: Immagina di avere un database di traffico di rete. Per sbaglio, un pacchetto normale è stato etichettato come "Attacco" nel training set (un errore umano).
    - Se arriva un nuovo pacchetto normale molto simile a quell'errore, con K=1 il sistema lo classificherà subito come "Attacco".
    - Vedrai quindi confini frastagliati, con piccole "isole" di colore sbagliato in mezzo ad aree di colore diverso
- K=15: "La Democrazia"
  - Quando imposti n_neighbors=15, l'algoritmo guarda i 15 punti più vicini e fa una votazione a maggioranza.
  - "Ok, guardiamo i 15 tizi intorno a te. 12 dicono che sei 'Spam' e 3 dicono che sei 'Legittimo'. Vince la maggioranza: sei Spam."
  - Generalizzazione: Ignora gli errori isolati.
  - Esempio Cyber: Se vicino c'è quel singolo pacchetto etichettato male (l'errore di prima), ma ci sono altri 14 pacchetti "Normali" intorno, il voto finirà 14 a 1 per "Normale". Il sistema corregge l'errore e non genera un falso allarme.
  - I confini tra le classi diventano linee curve e morbide. Le piccole anomalie vengono "assorbite" dalla maggioranza.

KNN

Caratteristica	K=1 (Basso)	K=15 (Alto)
Comportamento	Copia il singolo vicino più prossimo	Fa una votazione tra i vicini
Sensibilità al rumore	**Altissima.** Un solo dato sbagliato rovina la predizione	**Bassa.** La maggioranza corregge gli errori singoli
Confini decisionali	Complessi, frastagliati, a "macchia di leopardo"	Lisci, semplici, curve morbide
Rischio principale	**Overfitting** (Impara a memoria i dati, compresi gli errori)	**Underfitting** (Se K è troppo grande, diventa troppo generico e "sfocato")

KNN: la distanza?

- IRIS
  - Geometria pura
  - Nel dataset Iris, stiamo misurando lunghezze fisiche (petali e sepali in cm).
  - Funzione: Distanza Euclidea. È il classico "Teorema di Pitagora". Se un fiore ha un petalo lungo 2cm e l'altro 2.1cm, sono vicini. Lo spazio è continuo e le dimensioni hanno tutte lo stesso significato fisico
- MNIST
  - Alta Dimensionalità
  - Un'immagine è un vettore di 784 numeri (intensità di grigio).
  - Funzione Standard: Distanza Euclidea
    - Pro: Facile da capire (differenza pixel per pixel).
    - Contro: Soffre terribilmente se l'immagine è leggermente traslata o ruotata.
    - Se sposti un "1" di un pixel a destra, la distanza Euclidea esplode perché i pixel neri non si sovrappongono più, anche se è sempre un "1".
  - Funzione Avanzata: Cosine Similarity
    - .Invece di misurare quanto sono lontani i punti, misura l'angolo tra i due vettori
    - In alta dimensionalità (come le immagini o il testo), spesso funziona meglio perché ignora la "magnitudine" (es. se un'immagine è più scura dell'altra ma ha la stessa forma) e guarda la "direzione"
- CYBERSECURITY
  - Il caos dei dati misti (Distanza di Hamming / Manhattan)
  - Nei dati Cyber (es. log di rete) hai colonne con significati completamente diversi
    - Durata: 0.05 secondi (float)
    - Protocollo: TCP, UDP
    - Bytes: 1500 (intero)
  - Il problema: Se usi la Euclidea pura su dati grezzi, la colonna "Bytes" (valore 1500) dominerà completamente la colonna "Durata" (valore 0.05). La distanza sarà determinata solo dai Bytes.

KNN: la distanza?

- Normalizzazione (StandardScaler)
  - Prima di scegliere la distanza, devi portare tutto sulla stessa scala (media 0, varianza 1)
  - Senza questo, il KNN e NN è inutile.
  - Dopo la normalizzazione
    - Puoi usare la Euclidea.
- Per dati Categorici (es. Protocollo, Flag)
  - Distanza di Hamming o di Levenstein
    - Se stai confrontando stringhe o bit (es. TCP vs UDP, Flag SYN on/off)
    - Conta quanti caratteri/bit sono diversi
      - Esempio
        - Pacchetto A: [TCP, SYN, 80]
        - Pacchetto B: [UDP, SYN, 80]
        - Distanza: 1 (solo il primo elemento è diverso).
      - Per dati misti robusti
        - Distanza di Manhattan
          - Spesso si preferisce la Manhattan (somma dei valori assoluti delle differenze) alla Euclidea
          - Perché: La Manhattan è meno sensibile agli outlier estremi (es. un singolo pacchetto con dimensione enorme) rispetto alla Euclidea che eleva le differenze al quadrato (amplificando gli errori grandi).

KNN: la distanza?

- In scikit-learn

```
sklearn.neighbors import KNeighborsClassifier

# 1. Classica Euclidea (Default) - Va bene per Iris e dati normalizzati
knn_euclidean = KNeighborsClassifier(n_neighbors=3, metric='euclidean')

# 2. Manhattan (L1) - Spesso meglio per Cyber/Network ad alta dimensionalità
knn_manhattan = KNeighborsClassifier(n_neighbors=3, metric='manhattan')

# from 3. Hamming - Se avessero solo dati categorici (es. DNA o sequenze di Flag)
# Nota: richiede che i dati siano interi o binari
knn_hamming = KNeighborsClassifier(n_neighbors=3, metric='hamming')

# 4. Cosine - Ottimo per testo (Spam detection) o vettori sparsi
knn_cosine = KNeighborsClassifier(n_neighbors=3, metric='cosine')
```

KNN: Come si formano i cluster?

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn.neighbors import KNeighborsClassifier
from sklearn.datasets import make_blobs, make_moons

# 1. Creiamo un dataset sintetico 2D
# make_blobs crea cluster netti, make_moons crea forme complesse (ottimo per vedere la potenza del KNN)
X, y = make_moons(n_samples=200, noise=0.3, random_state=42)
# X, y = make_blobs(n_samples=200, centers=3, random_state=42)

cmap_light = ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF'])
cmap_bold = ListedColormap(['#FF0000', '#00FF00', '#0000FF'])

def plot_knn_boundary(k_value):
    clf = KNeighborsClassifier(n_neighbors=k_value)
    clf.fit(X, y)

    # Creiamo una "griglia" di punti che copre tutto il grafico
    h = .02  # passo della griglia
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))

    # Chiediamo al KNN di predire il colore per OGNI punto della griglia (lo sfondo)
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    # Disegniamo
    plt.figure(figsize=(8, 6))
    plt.pcolormesh(xx, yy, Z, cmap=cmap_light, shading='auto') # Sfondo colorato
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap_bold, edgecolor='k', s=20) # Punti reali
    plt.title(f"KNN Classificazione (K={k_value})\ni 'territori'")
    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())
    plt.show()

# ESEGUIAMO CON DIVERSI K
# K=1: "Overfitting". Il confine è frastagliato, insegue ogni singolo punto di rumore.
plot_knn_boundary(1)

# K=15: "Generalizzazione". Il confine è più morbido e ignora i punti anomali isolati.
plot_knn_boundary(15)
```

KNN: e cosa fa il training?

Prepara le strutture dati di ricerca successive. Non apprende nulla e non butta nulla

```python
import os
import pickle
import numpy as np
import tensorflow as tf
from sklearn.neighbors import KNeighborsClassifier
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

def get_file_size(filepath):
    """Restituisce la dimensione del file in Kilobyte (KB)"""
    return os.path.getsize(filepath) / 1024

# 1. SETUP DELL'ESPERIMENTO
# Creiamo un dataset "corposo": 10.000 campioni, 50 caratteristiche ciascuno
n_samples = 10000
n_features = 50
X = np.random.rand(n_samples, n_features).astype('float32')
y = np.random.randint(0, 2, n_samples)

print(f"--- ESPERIMENTO: {n_samples} Campioni con {n_features} caratteristiche ---")

# ==========================================
# 2. ADDESTRAMENTO E SALVATAGGIO KNN
# ==========================================
print("\nAddestramento KNN in corso...")
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X, y)

# Salviamo il modello su disco (simuliamo l'invio al cliente)
with open('modello_knn.pkl', 'wb') as f:
    pickle.dump(knn, f)

size_knn = get_file_size('modello_knn.pkl')
print(f"  Dimensione modello KNN su disco: {size_knn:.2f} KB")

# ==========================================
# 3. ADDESTRAMENTO E SALVATAGGIO RETE NEURALE
# ==========================================
print("\nAddestramento Rete Neurale in corso...")
model = Sequential([
    Dense(32, input_dim=n_features, activation='relu'), # Hidden Layer
    Dense(16, activation='relu'),                       # Hidden Layer
    Dense(1, activation='sigmoid')                      # Output Layer
])
model.compile(loss='binary_crossentropy', optimizer='adam')
model.fit(X, y, epochs=2, verbose=0, batch_size=32) # Poche epoche, basta che
inizializzi i pesi

# Salviamo il modello nel nuovo formato nativo di Keras
model.save('modello_nn.keras')

size_nn = get_file_size('modello_nn.keras')
print(f"  Dimensione modello Neural Network su disco: {size_nn:.2f} KB")

# ==========================================
# 4. IL RISULTATO FINALE (Il confronto)
# ==========================================
ratio = size_knn / size_nn
print(f"\n--- CONCLUSIONE ---")
print(f"Il modello KNN è {ratio:.1f} volte più pesante della Rete Neurale!")
print(f"Se dovessi spedire l'aggiornamento antivirus a 1 milione di clienti:")
print(f"- Traffico KNN: {size_knn * 1000000 / 1024 / 1024:.2f} GB")
print(f"- Traffico NN:  {size_nn * 1000000 / 1024 / 1024:.2f} GB")
```

KNN: MNIST

```python
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import mnist
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
import time

# 1. Caricamento dati
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# 2. FLATTENING: Da Matrice (28x28) a Vettore (784)
# Scikit-learn vuole vettori piatti, non matrici
x_train_flat = x_train.reshape(x_train.shape[0], -1)  # (60000, 784)
x_test_flat = x_test.reshape(x_test.shape[0], -1)    # (10000, 784)

# 3. RIDUZIONE DATASET
# Usiamo solo 5000 immagini per il training per non bloccare il PC
limit = 5000
x_train_small = x_train_flat[:limit]
y_train_small = y_train[:limit]

# Normalizzazione (0-255 -> 0-1) aiuta il calcolo della distanza euclidea
x_train_small = x_train_small / 255.0
x_test_flat = x_test_flat / 255.0

print(f"Training su {limit} immagini (vettori di dim 784)...")

# 4. Addestramento KNN
k = 3
knn = KNeighborsClassifier(n_neighbors=k)

start_time = time.time()
knn.fit(x_train_small, y_train_small) # "Fit" nel KNN è solo memorizzazione
print(f"Training completato in {time.time() - start_time:.2f} secondi.")

# 5. Predizione su un esempio
index = 0
sample_image = x_test_flat[index].reshape(1, -1)
prediction = knn.predict(sample_image)

# Visualizzazione risultato
plt.imshow(x_test[index], cmap='gray')
plt.title(f"Vero: {y_test[index]} - Predetto dal KNN: {prediction[0]}")
plt.show()

# 6. Accuracy su tutto il test set (o una parte)
print("Calcolo accuracy su 1000 test images...")
preds = knn.predict(x_test_flat[:1000])
acc = accuracy_score(y_test[:1000], preds)
print(f"Accuracy KNN (K={k}): {acc:.2%}")
```

KNN: MNIST

```
# Al posto di una sola immagine, mostra 50 immagini
# 1. Impostiamo la griglia: 5 righe, 10 colonne (Totale 50 immagini)
# figsize=(20, 10) rende la finestra bella grande
fig, axes = plt.subplots(5, 10, figsize=(20, 10))
fig.suptitle('Test KNN su 50 immagini casuali (Verde=Ok, Rosso=Errore)', fontsize=16)

# Appiattiamo l'array degli assi per poterci ciclare sopra facilmente
axes_flat = axes.flatten()

# 2. Ciclo su ogni "casella" della griglia
for i, ax in enumerate(axes_flat):

    # Prendiamo un indice a caso
    index = random.randint(0, x_test.shape[0] - 1)

    # Preparazione dato per il KNN (Reshape)
    sample_image_flat = x_test_flat[index].reshape(1, -1)

    # Predizione
    prediction = knn.predict(sample_image_flat)[0]
    true_label = y_test[index]

    # 3. Visualizzazione nell'asse corrente (ax)
    ax.imshow(x_test[index], cmap='gray')

    # Logica Colore: Verde se giusto, Rosso se sbagliato
    if prediction == true_label:
        color = 'green'
        title_text = f"{prediction}"
    else:
        color = 'red'
        title_text = f"V:{true_label} P:{prediction}" # Mostra Vero e Predetto se sbaglia

    ax.set_title(title_text, color=color, fontsize=10, fontweight='bold')
    ax.axis('off') # Nasconde i numerini degli assi (x, y) per pulizia

# 4. Aggiusta gli spazi automaticamente per non sovrapporre i testi
plt.tight_layout()
plt.subplots_adjust(top=0.92) # Lascia spazio per il titolo principale
plt.show()
```

KNN e NN

```
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
import matplotlib.pyplot as plt

# 1. Caricamento dati
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Normalizzazione (0-255 diventa 0-1) -> Fondamentale per le reti neurali
x_train, x_test = x_train / 255.0, x_test / 255.0

# L'immagine è una matrice
plt.imshow(x_train[0], cmap='gray')
plt.title(f"Label: {y_train[0]} - Shape originale: {x_train[0].shape}")
plt.show()

# 2. Creazione del Modello
model = Sequential([
    # Input layer che "schiaccia" la matrice 28x28 in un vettore lungo 784
    Flatten(input_shape=(28, 28)),

    # Hidden Layer (ricordate lo xor?)
    Dense(128, activation='relu'),

    # Output Layer (10 neuroni per le 10 cifre, softmax per la probabilità)
    Dense(10, activation='softmax')
])

# 3. Compile e Training
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5)
```

KNN e NN

```
# E ora una prova
# 1. Scegliamo un'immagine a caso dal test set (es. la numero 123)
idx = 123
immagine = x_test[idx]
etichetta_vera = y_test[idx]

# 2. PREDIZIONE
# Attenzione: Keras si aspetta un "batch" di immagini, anche se ne passiamo una sola.
# Dobbiamo aggiungere una dimensione: (28, 28) -> (1, 28, 28)
immagine_batch = np.expand_dims(immagine, axis=0)

# Otteniamo il vettore di 10 probabilità
predizioni = model.predict(immagine_batch)

# 3. INTERPRETAZIONE (Argmax)
# np.argmax ci dice l'indice del valore più alto nel vettore
numero_predetto = np.argmax(predizioni)
confidenza = np.max(predizioni) * 100 # Quanto è sicura?

# 4. VISUALIZZAZIONE
plt.imshow(immagine, cmap='gray')
plt.title(f"Vero: {etichetta_vera} | Predetto: {numero_predetto}\nSicurezza: {confidenza:.2f}%")
plt.axis('off')
plt.show()

print("Vettore Probabilità:", np.round(predizioni, 2))
```

KNN e NN

```
def visualizza_predizione_avanzata(index):
    img = x_test[index]
    true_label = y_test[index]

    # Predizione
    img_batch = np.expand_dims(img, axis=0)
    probs = model.predict(img_batch)[0] # Prendiamo il primo elemento del batch
    pred_label = np.argmax(probs)

    # Creiamo il grafico doppio
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 4))

    # Sinistra: Immagine
    ax1.imshow(img, cmap='gray')
    ax1.axis('off')
    color = 'green' if pred_label == true_label else 'red'
    ax1.set_title(f"Vero: {true_label} | Predetto: {pred_label}", color=color, fontsize=14)

    # Destra: Istogramma probabilità
    bars = ax2.bar(range(10), probs, color="#777777")
    ax2.set_ylim([0, 1])
    ax2.set_xticks(range(10))
    ax2.set_title("Cosa pensa la Rete (Softmax output)")

    # Evidenziamo la colonna della predizione (blu) e quella vera (verde)
    bars[pred_label].set_color('red') # Quello che ha scelto (se sbagliato è rosso)
    bars[true_label].set_color('green') # Quello che doveva essere

    plt.tight_layout()
    plt.show()

visualizza_predizione_avanzata(123)
visualizza_predizione_avanzata(321)
```

Se invece dei pixel avessimo le frequenze dei byte di un file .exe malevolo, cambierebbe qualcosa per la rete?"

RESHAPE

```
["google", "yahoo", "bing"]
non è una tabella

dati.reshape(-1, 1))
[
  ["google"],
  ["yahoo"],
  ["bing"]
]
Ora è una tabella
```

E la distanza di levenshtein?

```python
import numpy as np
from sklearn.neighbors import KNeighborsClassifier
def lev_dist_pure(s1, s2):
    if len(s1) < len(s2):
        return lev_dist_pure(s2, s1)
    if len(s2) == 0:
        return len(s1)
    previous_row = range(len(s2) + 1)
    for i, c1 in enumerate(s1):
        current_row = [i + 1]
        for j, c2 in enumerate(s2):
            insertions = previous_row[j + 1] + 1
            deletions = current_row[j] + 1
            substitutions = previous_row[j] + (c1 != c2)
            current_row.append(min(insertions, deletions, substitutions))
        previous_row = current_row
    return previous_row[-1]

# --- 1. DATI REALI (Le stringhe restano fuori dal KNN) ---
# Creiamo una lista "esterna" che funge da database
corpus_train = [
    "google.com",
    "facebook.com",
    "amazon.com",
    "g00gle.com",      # Phishing
    "faceb00k.login",  # Phishing
    "amaz0n-security"  # Phishing
]
y_train = np.array([0, 0, 0, 1, 1, 1])

# --- 2. DATI PER IL KNN (Solo gli indici!) ---
# Creiamo un array di numeri: [[0], [1], [2], [3], [4], [5]]
# Questo è ciò che vedrà Scikit-Learn. Per lui sono numeri, quindi è felice.
X_train_indices = np.arange(len(corpus_train)).reshape(-1, 1)

# --- 3. LA FUNZIONE METRICA TRUCCATA ---
def levenshtein_metric_proxy(x, y):
    # x e y qui dentro arrivano come numeri (float).
    # Esempio: x=[3.0], y=[0.0]

    # 1. Convertiamo in intero per avere l'indice
    idx1 = int(x[0])
    idx2 = int(y[0])

    # 2. Recuperiamo le stringhe vere dalla lista esterna
    # Nota: Dobbiamo gestire sia il training che il test.
    # Se l'indice è dentro il range del training, usiamo corpus_train.
    # Ma se stiamo testando, l'indice arriverà dal test set...
    # Qui stiamo assumendo che x e y siano entrambi indici che puntano
    # a una lista globale o che facciamo un "lookup".

   # Per far funzionare fit() e predict() insieme senza impazzire con gli indici,
    # passiamo direttamente le stringhe nel test set usando una Lambda,
    # ma siccome sklearn blocca le stringhe, la via più pulita è usare "precomputed".
    # Ma proviamo a finire questo approccio "proxy" con una lista unificata temporanea.

    str1 = data_source[idx1]
    str2 = data_source[idx2]

    return lev_dist_pure(str1, str2)

# ==============================================================================
# (Evita tutti i problemi di indici e tipi di dati)
# Questa è quella che ti consiglio di mostrare, funziona sempre.
# ==============================================================================

# 1. Calcoliamo noi la matrice delle distanze PRIMA
def compute_distance_matrix(list_a, list_b):
    matrix = np.zeros((len(list_a), len(list_b)))
    for i, str_a in enumerate(list_a):
        for j, str_b in enumerate(list_b):
            matrix[i, j] = lev_dist_pure(str_a, str_b)
    return matrix

print("1. Calcolo matrice distanze Training...")
# Distanza tra training e se stesso
X_train_dist_matrix = compute_distance_matrix(corpus_train, corpus_train)

# 2. Addestriamo il KNN dicendogli "Ho già calcolato tutto io" (metric='precomputed')
knn = KNeighborsClassifier(n_neighbors=1, metric='precomputed')
knn.fit(X_train_dist_matrix, y_train)

# 3. Test
test_domains = ["google.it", "googel.com", "microsoft.com"]

print("2. Calcolo matrice distanze Test...")
# Per predire, dobbiamo dare le distanze tra i NUOVI dati e i VECCHI dati (Training)
# Righe = Nuovi dati, Colonne = Dati Training
X_test_dist_matrix = compute_distance_matrix(test_domains, corpus_train)

print("\n--- RISULTATI ---")
preds = knn.predict(X_test_dist_matrix)

for domain, pred in zip(test_domains, preds):
    label = "Phishing" if pred == 1 else "Safe"
    # Troviamo il vicino più prossimo manualmente per mostrarlo
    dists = X_test_dist_matrix[test_domains.index(domain)]
    nearest_idx = np.argmin(dists)
    nearest_name = corpus_train[nearest_idx]
    nearest_dist = dists[nearest_idx]

    print(f"Dominio: '{domain}' -> {label}")
    print(f"  (Simile a: '{nearest_name}', Distanza: {int(nearest_dist)})")
```

E la stessa cosa con una NN?

```python
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from Levenshtein import distance as lev_dist

# 1. DEFINIAMO LE "ANCORE" (I domini legittimi di riferimento)
# Questi sono i punti fissi del nostro universo.
anchors = ["google.com", "facebook.com", "amazon.com", "apple.com", "microsoft.com"]

# 2. FUNZIONE DI VETTORIZZAZIONE (Feature Engineering)
# Trasforma una stringa in una lista di distanze dalle ancore
def string_to_vector(domain):
    # Calcola la distanza da OGNI ancora
    dists = [lev_dist(domain, anchor) for anchor in anchors]
    return np.array(dists)

# --- 3. CREIAMO IL DATASET ---
# Dati grezzi
raw_data = [
    "google.com",       # Legit (Dist da google: 0)
    "facebook.com",     # Legit
    "amazon.com",       # Legit
    "g0ogle.com",       # Phish (Dist da google: 1)
    "faceb00k.com",     # Phish (Dist da facebook: 2)
    "amaz0n.net",       # Phish
    "appple.com",       # Phish
    "microsft.com",     # Phish
    "yahoo.com"         # Legit (ma non è nelle ancore, vediamo che succede)
]

# Labels: 0 = Legit, 1 = Phish
labels = np.array([0, 0, 0, 1, 1, 1, 1, 1, 0])

# Trasformiamo le stringhe in vettori numerici
# X sarà una matrice (9 campioni, 5 ancore)
X = np.array([string_to_vector(d) for d in raw_data])

# Normalizziamo leggermente (opzionale ma utile per le NN)
X = X / 10.0  # Scaliamo le distanze (assumiamo max dist ~10-20)

print("Esempio di input trasformato per 'g0ogle.com':")
print(f"Ancore: {anchors}")
print(f"Vettore Distanze: {string_to_vector('g0ogle.com')}")
# Output atteso: [1, 7, 6, 7, 12] -> Nota l'1 all'inizio!

# --- 4. COSTRUIAMO LA RETE NEURALE ---
model = Sequential([
    # Input Layer: dimensione = numero di ancore (5)
    Dense(16, input_dim=len(anchors), activation='relu'),
    Dense(8, activation='relu'),
    Dense(1, activation='sigmoid') # Output: Probabilità Phishing
])

model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

# --- 5. ADDESTRAMENTO ---
print("\nTraining NN...")
model.fit(X, labels, epochs=100, verbose=0)
print("Training completato.")

# --- 6. TEST SU NUOVI DOMINI ---
test_domains = ["google.it", "googel.com", "amazon.com", "msoft.com"]
print("\n--- TEST ---")

for domain in test_domains:
    # 1. Preprocessing (uguale al training)
    vec = string_to_vector(domain)
    vec_norm = vec / 10.0

    # 2. Reshape per Keras (da (5,) a (1, 5))
    vec_input = vec_norm.reshape(1, -1)

    # 3. Predizione
    prob = model.predict(vec_input, verbose=0)[0][0]

    label = "PHISHING" if prob > 0.5 else "LEGIT"
    print(f"Dominio: '{domain}'")
    print(f" -> Vettore distanze: {vec}")
    print(f" -> Rete dice: {prob:.4f} ({label})")
```

E la stessa cosa con una NN?

- Il modello sembra un pochino debole
  - Aumentiamo le epoche e riproviamo
  - Ora come va?
- E se invece aumentassimo i dati errati?
  - Non è sufficiente, sarebbero tutti errati
- Vanno aumentati i dati “buoni” e quelli “errati”!!!!

E la stessa cosa con una NN?

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
import random
import string
from Levenshtein import distance as lev_dist

# --- 0. SETUP: SITI DA PROTEGGERE ---
ANCHORS = ["google.com", "facebook.com", "amazon.com",
"apple.com", "microsoft.com"]
print(f"Siti Protetti: {ANCHORS}")

def string_to_vector(domain):
    # Normalizziamo dividendo per 15.0 per avere numeri tra 0 e 1
    dists = [lev_dist(domain, anchor) for anchor in ANCHORS]
    return np.array(dists) / 15.0

def generate_typo_attack(domain):
    replacements = {'o':'0', 'l':'1', 'i':'1', 'a':'@', 'e':'3', 's':'5'}
    char_list = list(domain)
    num_changes = random.randint(1, 2)
    for _ in range(num_changes):
        idx = random.randint(0, len(char_list)-1)
        c = char_list[idx]
        if c in replacements and random.random() > 0.5:
            char_list[idx] = replacements[c]
        else:
            char_list[idx] = random.choice(string.ascii_lowercase)
    return "".join(char_list)

# --- 1. GENERAZIONE DATASET

X_data = []
y_data = []
samples_per_group = 150 # Abbondiamo coi dati

print("Generazione dataset in corso...")

# GRUPPO A: I SITI ORIGINALI (Insegnano: Distanza 0 = SICURO)
for _ in range(samples_per_group):
    # Prendiamo un'ancora a caso e la lasciamo INTATTA
    domain = random.choice(ANCHORS)
    X_data.append(string_to_vector(domain))
    y_data.append(0) # 0 = SICURO

# GRUPPO B: GLI ATTACCHI (Insegnano: Distanza Piccola =
PHISHING)
for _ in range(samples_per_group):
    target = random.choice(ANCHORS)
    fake_domain = generate_typo_attack(target)
    X_data.append(string_to_vector(fake_domain))
    y_data.append(1) # 1 = PHISHING

# GRUPPO C: I SITI RANDOM (Insegnano: Distanza Grande =
SICURO)
for _ in range(samples_per_group):
    rnd_name = ''.join(random.choices(string.ascii_lowercase, k=10)) +
".com"
    X_data.append(string_to_vector(rnd_name))
    y_data.append(0) # 0 = SICURO

# Setup array e shuffle
X_train = np.array(X_data)
y_train = np.array(y_data)
indices = np.arange(X_train.shape[0])
np.random.shuffle(indices)
X_train = X_train[indices]
y_train = y_train[indices]

print(f"Dataset pronto: {len(X_train)} campioni (1/3 Veri, 1/3 Attacchi, 1/3
Random)")

# --- 2. RETE NEURALE ---
# Aumentiamo un po' i neuroni per fargli capire la non-linearità
model = Sequential([
    Dense(64, input_dim=len(ANCHORS), activation='relu'),
    Dense(32, activation='relu'),
    Dense(1, activation='sigmoid')
])

model.compile(loss='binary_crossentropy', optimizer='adam',
metrics=['accuracy'])

print("Training... (Sto imparando che 0 è buono, 1 è male, 10 è buono)")
model.fit(X_train, y_train, epochs=200, verbose=0)

# --- 3. TEST FINALE ---
print("\n--- RISULTATI TEST ---")
test_cases = [
    "google.com",       # ORIGINALE (Dist 0) -> Deve essere SICURO
    "amazon.com",       # ORIGINALE (Dist 0) -> Deve essere SICURO
    "g0ogle.com",       # TYPO (Dist piccola) -> Deve essere PHISHING
    "faceb00k.com",     # TYPO (Dist piccola) -> Deve essere PHISHING
    "rnicrosoft.com",   # TYPO (Dist piccola) -> Deve essere PHISHING
    "repubblica.it",    # RANDOM (Dist grande) -> Deve essere SICURO
    "pizzapasta.com"    # RANDOM (Dist grande) -> Deve essere
SICURO
]

for domain in test_cases:
    vec = string_to_vector(domain)
    pred = model.predict(vec.reshape(1, -1), verbose=0)[0][0]

    # Ora la soglia la impostiamo severa
    is_phishing = pred > 0.5
    verdict = "⚠ PHISHING" if is_phishing else "    SICURO"

    print(f"{domain:<18} | Prob Phishing: {pred*100:6.2f}% | {verdict}")
```

Sequenza di esempi e esercizi

- words.py: correzione testo, individuazione parole “strane”
- wine.py: classificazione dei vini
- fruits.py: costruzione del dataset (finto) e analisi della classificazione fornita dal KNN
- nn_basic_classification.py: classificazione del traffico
- nn_autoencoder.py: identificazione di anomalie
- ml_spam_detection.py: analisi dello spam!!
  - buildspam_dataset.py che usa spamassassin_public_corpus
- Detection delle anomalie e esempio di ids
  - Vedi slide successive e poi ids.py
- pm.py: process mining e ricostruzione dei flussi

Reti neurali ricorrenti RNN

- LSTM (Long Short-Term Memory), per analizzare le serie temporali del traffico
- Gli attacchi moderni (come gli APT - Advanced Persistent Threats) sono lenti e "silenziosi".
  - Individuare l'attacco "Slowloris"
    - L'attacco Slowloris apre molte connessioni a un server web e invia i dati molto lentamente, tenendo le porte occupate per ore senza mai superare le soglie di allarme dei firewall/ids tradizionali.

- Come funziona?
  - Invece di analizzare una riga di dati alla volta, diamo alla rete una sequenza di pacchetti (una "finestra temporale").
  - Memoria: La LSTM ha delle "celle di memoria" che ricordano cosa è successo 10 o 20 pacchetti fa.
  - Riconoscimento del pattern: Se vede che un utente apre una connessione, invia un byte, aspetta 29 secondi, invia un altro byte, e ripete questo per 100 volte, la LSTM capisce che non è un comportamento umano ma un tentativo di esaurire le risorse del server.

- Codice Python

```
X_sequence = np.random.random((1000, 20, 10))
model_lstm = tf.keras.Sequential([
  # Lo strato LSTM "ricorda" l'andamento del traffico nel tempo
  tf.keras.layers.LSTM(64, input_shape=(20, 10), return_sequences=False),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(1, activation='sigmoid')
   ])
model_lstm.compile(optimizer='adam', loss='binary_crossentropy')
```

E per riconoscere attacchi zero-day?

- Invece di insegnare alla rete cos'è un attacco, le insegni solo cos'è il traffico normale.
- La rete impara a "comprimere" e "ricostruire" il traffico legittimo.
  - Quando arriva un attacco mai visto prima (un attacco Zero-Day), la rete non riuscirà a ricostruirlo correttamente.
  - L'errore di ricostruzione alto farà scattare l'allarme.

NN: un’applicazione

- Il dataset NSL-KDD è uno dei punti di riferimento più importanti a livello mondiale per chi si occupa di sicurezza informatica e intelligenza artificiale.
  - Deriva da KDD Cup 1999 Data (https://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html) che riporta i seguenti attacchi: back dos, buffer_overflow u2r, ftp_write r2l, guess_passwd r2l, imap r2l, ipsweep probe, land dos, loadmodule u2r, multihop r2l, neptune dos, nmap probe, perl u2r, phf r2l, pod dos, portsweep probe, rootkit u2r, satan probe, smurf dos, spy r2l, teardrop dos, warezclient r2l, warezmaster r2l

- Il dataset viene utilizzato dai ricercatori per addestrare algoritmi di Machine Learning a distinguere tra
  - Traffico normale: Connessioni di rete legittime.
  - Attacchi informatici: Tentativi di intrusione raggruppati in quattro categorie principali:
  - DoS (Denial of Service): Tentativi di sovraccaricare il sistema.
  - R2L (Remote to Local): Accesso non autorizzato da una macchina remota.
  - U2R (User to Root): Tentativi di ottenere privilegi di amministratore.
  - Probe: Scansione della rete alla ricerca di vulnerabilità.

NN: un’applicazione

- Il file è in formato CSV e ogni riga rappresenta una singola connessione di rete con 43 attributi totali:

N° Attributo	Esempio Contenuto	Descrizione
1-41	0, tcp, ftp_data, SF, 491...	Caratteristiche tecniche: durata, protocollo (TCP/UDP), servizio, byte inviati, ecc.
42	normal oppure neptune (syn flooding)	Label: indica se la connessione è sicura o il tipo di attacco.
43	20	Punteggio di difficoltà: indica quanto è stato difficile per i modelli precedenti classificare quella riga

NN: un’applicazione

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
import tensorflow as tf

# URL diretto al dataset (versione ridotta)
url = "https://raw.githubusercontent.com/defcom17/NSL_KDD/master/KDDTrain%2B_20Percent.txt"
# Colonne principali
col_names = ["duration","protocol_type","service","flag","src_bytes","dst_bytes","land","wrong_fragment","urgent","hot",
             "num_failed_logins","logged_in","num_compromised","root_shell","su_attempted","num_root","num_file_creations",
             "num_shells","num_access_files","num_outbound_cmds","is_host_login","is_guest_login","count","srv_count",
             "serror_rate","srv_serror_rate","rerror_rate","srv_rerror_rate","same_srv_rate","diff_srv_rate",
             "srv_diff_host_rate","dst_host_count","dst_host_srv_count","dst_host_same_srv_rate",
             "dst_host_diff_srv_rate","dst_host_same_src_port_rate","dst_host_srv_diff_host_rate",
             "dst_host_serror_rate","dst_host_srv_serror_rate","dst_host_rerror_rate","dst_host_srv_rerror_rate","label",
"difficulty"]

df = pd.read_csv(url, names=col_names)

# 1. PREPROCESSING
# Trasformiamo la label in Binario: "normal" -> 0, tutto il resto (attacco) -> 1
df['target'] = df['label'].apply(lambda x: 0 if x == 'normal' else 1)

# Selezioniamo alcune feature numeriche e categoriche
features_numeric = ['duration', 'src_bytes', 'dst_bytes', 'count', 'srv_count', 'dst_host_count']
features_cat = ['protocol_type', 'service', 'flag']

# Creiamo il set di dati X con le sole colonne scelte
X_numeric = df[features_numeric]
X_cat = pd.get_dummies(df[features_cat]) # Trasforma 'tcp' in colonne separate (0/1)

# Uniamo tutto
X = pd.concat([X_numeric, X_cat], axis=1)
y = df['target']

print(f"Nuovo numero di feature dopo l'encoding: {X.shape[1]}")

# 2. NORMALIZZAZIONE (Cruciale per KNN e NN)
# Le reti soffrono se 'src_bytes' è 1.000.000 e 'logged_in' è 1.
scaler = StandardScaler()
X = scaler.fit_transform(X)

# Split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

print(f"Dati pronti. Feature vector size: {X_train.shape[1]}")
```

```
# --- CONFRONTO ---

# A) KNN
print("Addestramento KNN...")
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train, y_train)
y_pred_knn = knn.predict(X_test)
print(f"KNN Accuracy: {accuracy_score(y_test, y_pred_knn):.4f}")

# B) Rete Neurale (TensorFlow)
print("\nAddestramento Neural Network...")
model = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(X_train.shape[1],)), # Input Layer: dimensione del vettore feature
    tf.keras.layers.Dense(16, activation='relu'),      # Hidden Layer
    tf.keras.layers.Dense(8, activation='relu'),       # Hidden Layer
    tf.keras.layers.Dense(1, activation='sigmoid')     # Output Layer (0 o 1)
])

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
model.fit(X_train, y_train, epochs=5, batch_size=32, verbose=1)

# Prendiamo i primi 10 campioni del test set
campioni = X_test[:10]
reali = y_test[:10].values

# Predizioni (per la NN usiamo una soglia di 0.5 perché l'output è una probabilità)
pred_knn = knn.predict(campioni)
pred_nn = (model.predict(campioni) > 0.5).astype(int).flatten()

print("Reale | KNN | NN")
for r, k, n in zip(reali, pred_knn, pred_nn):
    status = "   " if r == k == n else "x"
    print(f"  {r}   |  {k}  | {n}  {status}")

from sklearn.metrics import confusion_matrix, classification_report
print("\n--- Analisi Dettagliata KNN ---")
print(confusion_matrix(y_test, y_pred_knn))
print(classification_report(y_test, y_pred_knn, target_names=['Normal', 'Attack']))
```

NN: un’applicazione

```
print("\n--- TEST PRATICO ---")
 #
================================================================================
# METODO 1: INVENTIAMO DUE PACCHETTI (Compatibile con One-Hot Encoding)
================================================================================

# Definiamo i dati includendo anche le categorie (protocollo, servizio, flag)
dati_manuali = [
    # Caso A: Navigazione normale (TCP, HTTP, SF)
    {'duration': 0, 'src_bytes': 215, 'dst_bytes': 4500, 'count': 5, 'srv_count': 5, 'dst_host_count': 10,
     'protocol_type': 'tcp', 'service': 'http', 'flag': 'SF'},

    # Caso B: Attacco DoS (TCP, Private, S0)
    {'duration': 0, 'src_bytes': 0, 'dst_bytes': 0, 'count': 500, 'srv_count': 500, 'dst_host_count': 255,
     'protocol_type': 'tcp', 'service': 'private', 'flag': 'S0'}
]

# 1. Trasformiamo in DataFrame
df_manuale = pd.DataFrame(dati_manuali)

# 2. Applichiamo One-Hot Encoding (Dobbiamo allinearlo con le colonne di X!)
df_manuale_encoded = pd.get_dummies(df_manuale)

# 3. IL TRUCCO: Aggiungiamo le colonne mancanti che il modello si aspetta (mettendole a 0)
# e rimuoviamo quelle extra se presenti.
X_columns = pd.get_dummies(df[features_cat]).columns # Tutte le colonne generate prima
for col in X_columns:
    if col not in df_manuale_encoded.columns:
        df_manuale_encoded[col] = 0

# Ordiniamo le colonne esattamente come nel set di addestramento X
# Prima le numeriche, poi le categoriche codificate
final_cols = features_numeric + list(X_columns)
df_manuale_final = df_manuale_encoded[final_cols]

print(f"Dati manuali pronti. Formato: {df_manuale_final.shape}")

# --- ORA PUOI NORMALIZZARE E PREDIRE ---
nuovi_dati_scaled = scaler.transform(df_manuale_final)

print("\n--- RISULTATI KNN ---")
preds_knn = knn.predict(nuovi_dati_scaled)
for i, p in enumerate(preds_knn):
    print(f"Pacchetto {i+1}: {'ATTACCO' if p == 1 else 'NORMALE'}")

# ================================================================================
# METODO 2: PRENDIAMO UN DATO REALE DAL TEST SET (X_test)
# ================================================================================
print("\n--- VERIFICA SU UN DATO REALE (Dal Dataset) ---")

# Prendiamo un indice a caso, es. il numero 10
idx = 10
sample_data = X_test[idx].reshape(1, -1) # Reshape necessario per Keras (1 riga, N colonne)
true_label = y_test.iloc[idx] # La verità

# Predizione KNN
p_knn = knn.predict(sample_data)[0]

# Predizione NN
p_nn = model.predict(sample_data, verbose=0)[0][0]

print(f"Dato Scalato: {sample_data[0][:3]}... (mostrati primi 3 valori)")
print(f"Etichetta REALE: {'ATTACCO' if true_label==1 else 'NORMALE'}")
print(f"KNN dice:        {'ATTACCO' if p_knn==1 else 'NORMALE'}")
print(f"NN dice:         {p_nn:.4f} ({'ATTACCO' if p_nn>0.5 else 'NORMALE'})")
```

NN: un’applicazione

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
import tensorflow as tf

# 1. CARICAMENTO E MAPPATURA
url = "https://raw.githubusercontent.com/defcom17/NSL_KDD/master/KDDTrain%2B_20Percent.txt"
col_names = ["duration","protocol_type","service","flag","src_bytes","dst_bytes","land","wrong_fragment","urgent","hot",
             "num_failed_logins","logged_in","num_compromised","root_shell","su_attempted","num_root","num_file_creations",
             "num_shells","num_access_files","num_outbound_cmds","is_host_login","is_guest_login","count","srv_count",
             "serror_rate","srv_serror_rate","rerror_rate","srv_rerror_rate","same_srv_rate","diff_srv_rate",
             "srv_diff_host_rate","dst_host_count","dst_host_srv_count","dst_host_same_srv_rate",
             "dst_host_diff_srv_rate","dst_host_same_src_port_rate","dst_host_srv_diff_host_rate",
             "dst_host_serror_rate","dst_host_srv_serror_rate","dst_host_rerror_rate","dst_host_srv_rerror_rate","label",
"difficulty"]

df = pd.read_csv(url, names=col_names)

# Dizionario di mappatura (Standard per NSL-KDD)
attack_dict = {
    'normal': 'normal',
    'back': 'dos', 'land': 'dos', 'neptune': 'dos', 'pod': 'dos', 'smurf': 'dos', 'teardrop': 'dos', 'mailbomb': 'dos', 'apache2': 'dos',
'processtable': 'dos', 'udpstorm': 'dos',
    'ipsweep': 'probe', 'mscan': 'probe', 'nmap': 'probe', 'portsweep': 'probe', 'saint': 'probe', 'satan': 'probe',
    'ftp_write': 'r2l', 'guess_passwd': 'r2l', 'imap': 'r2l', 'multihop': 'r2l', 'phf': 'r2l', 'spy': 'r2l', 'warezclient': 'r2l', 'warezmaster': 'r2l',
'sendmail': 'r2l', 'named': 'r2l', 'snmpgetattack': 'r2l', 'snmpguess': 'r2l', 'xlock': 'r2l', 'xsnoop': 'r2l', 'httptunnel': 'r2l',
    'buffer_overflow': 'u2r', 'loadmodule': 'u2r', 'perl': 'u2r', 'rootkit': 'u2r', 'sqlattack': 'u2r', 'xterm': 'u2r', 'ps': 'u2r'
}

# Applichiamo la mappatura
df['attack_class'] = df['label'].map(attack_dict)

# 2. PREPROCESSING AVANZATO
# Trasformiamo le etichette di testo in numeri (0, 1, 2, 3, 4)
le = LabelEncoder()
df['target'] = le.fit_transform(df['attack_class'])
num_classes = len(le.classes_) # Saranno 5: normal, dos, probe, r2l, u2r

# One-Hot Encoding per le colonne categoriche (protocollo, servizio, flag)
X = pd.get_dummies(df.drop(['label', 'difficulty', 'attack_class', 'target'], axis=1))
y = df['target']

# Split e Scaling
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

```
# 3. RETE NEURALE MULTI-CLASSE
model = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(X_train.shape[1],)),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dropout(0.2), # Previene l'overfitting
    tf.keras.layers.Dense(32, activation='relu'),
    tf.keras.layers.Dense(num_classes, activation='softmax') # Softmax per probabilità multiple
])

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

print("Inizio addestramento multi-classe...")
model.fit(X_train, y_train, epochs=10, batch_size=32, verbose=1)

# 4. VERIFICA REALE
print("\n--- TEST DI CLASSIFICAZIONE ---")
# Prendiamo 100 esempi a caso dal test set
for i in range(10000):
                idx = np.random.randint(0, len(X_test))
                sample = X_test[idx:idx+1]
                real_label = le.inverse_transform([y_test.iloc[idx]])[0]
                # Predizione
                preds = model.predict(sample, verbose=0)
                predicted_class = le.inverse_transform([np.argmax(preds)])[0]
                confidence = np.max(preds) * 100
                print("\n**********************", i, "**********************")
                print(f"Classe Reale: {real_label}")
                print(f"Classe Predetta: {predicted_class} ({confidence:.2f}%)")
```

E per riconoscere attacchi zero-day?

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense
 # 1. CARICAMENTO DATI
url =
"https://raw.githubusercontent.com/defcom17/NSL_KDD/master/KDDTrain%2B_20Percent.txt"
col_names =
["duration","protocol_type","service","flag","src_bytes","dst_bytes","land","wrong_fragment","urge
nt","hot",

"num_failed_logins","logged_in","num_compromised","root_shell","su_attempted","num_root","n
um_file_creations",

"num_shells","num_access_files","num_outbound_cmds","is_host_login","is_guest_login","count
","srv_count",

"serror_rate","srv_serror_rate","rerror_rate","srv_rerror_rate","same_srv_rate","diff_srv_rate",
            "srv_diff_host_rate","dst_host_count","dst_host_srv_count","dst_host_same_srv_rate",
            "dst_host_diff_srv_rate","dst_host_same_src_port_rate","dst_host_srv_diff_host_rate",

"dst_host_serror_rate","dst_host_srv_serror_rate","dst_host_rerror_rate","dst_host_srv_rerror_ra
te","label", "difficulty"]
 df = pd.read_csv(url, names=col_names)
 # 2. PREPROCESSING
# Creiamo il target: 0 per normale, 1 per attacco
df['target'] = df['label'].apply(lambda x: 0 if x == 'normal' else 1)
 # Selezioniamo solo le colonne numeriche per semplicità in questo esempio
X = df.select_dtypes(include=[np.number]).drop(['target', 'difficulty'], axis=1)
y = df['target']
 # Split dei dati
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
 # Normalizzazione (Essenziale per Autoencoder)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
 # --- IL SEGRETO DELL'AUTOENCODER ---
# Addestriamo il modello SOLO sui dati NORMALI
X_train_normal = X_train_scaled[y_train == 0]
 # 3. COSTRUZIONE DEL MODELLO
input_dim = X_train_normal.shape[1]
 input_layer = Input(shape=(input_dim,))
# Encoder
encoder = Dense(20, activation="relu")(input_layer)
encoder = Dense(10, activation="relu")(encoder) # Bottleneck
# Decoder
decoder = Dense(20, activation="relu")(encoder)
decoder = Dense(input_dim, activation="linear")(decoder)
autoencoder = Model(inputs=input_layer, outputs=decoder)
autoencoder.compile(optimizer='adam', loss='mse')
# 4. ADDESTRAMENTO
print("Addestramento in corso (Solo traffico normale)...")
autoencoder.fit(X_train_normal, X_train_normal,
                epochs=20, batch_size=64,
                validation_split=0.1, verbose=0)
# 5. RILEVAMENTO ANOMALIE
# Calcoliamo l'errore di ricostruzione (MSE) sui dati di test
predictions = autoencoder.predict(X_test_scaled)
mse = np.mean(np.power(X_test_scaled - predictions, 2), axis=1)
# Calcoliamo una soglia di allarme (es. media + 3 volte la deviazione standard)
threshold = np.mean(mse[y_test == 0]) + 3 * np.std(mse[y_test == 0])
print(f"Soglia di allarme calcolata: {threshold:.4f}")
# Valutazione
y_pred = [1 if e > threshold else 0 for e in mse]
```

E per riconoscere attacchi zero-day?

```
# ============================================================
# APPLICAZIONE PRATICA: TEST SU UN GRUPPO DI INPUT
# ============================================================
# 1. Selezioniamo un mix di 10 campioni dal test set (alcuni normali, alcuni attacchi)
input_campioni = X_test_scaled[:10]
etichette_reali = y_test[:10].values
# 2. L'Autoencoder prova a ricostruire questi pacchetti
ricostruiti = autoencoder.predict(input_campioni, verbose=0)
# 3. Calcoliamo l'errore (MSE) per ogni singolo pacchetto
errori = np.mean(np.power(input_campioni - ricostruiti, 2), axis=1)
print(f"{'ID':<5} | {'Errore MSE':<15} | {'Soglia':<10} | {'Predizione':<12} |
{'Reale':<10}")
print("-" * 65)
for i in range(len(input_campioni)):
    errore = errori[i]
    predizione = "⚠ ATTACCO" if errore > threshold else "    NORMALE"
    reale = "ATTACCO" if etichette_reali[i] == 1 else "NORMALE"
    # Marcatori per evidenziare errori del modello
    mark = "" if (errore > threshold) == etichette_reali[i] else "✕ ERRATO"
    print(f"{i:<5} | {errore:<15.4f} | {threshold:<10.4f} | {predizione:<12} | {reale:<10}
{mark}")

#============================================================
# TEST SU DATI TOTALMENTE INVENTATI
#============================================================
# Creiamo un pacchetto "assurdo" (es. src_bytes altissimi e durata enorme)
pacchetto_fake = np.zeros((1, X_train_normal.shape[1]))
pacchetto_fake[0, 1] = 50.0  # Iniettiamo un valore fuori scala (dopo
standardizzazione)
pred_fake = autoencoder.predict(pacchetto_fake, verbose=0)
errore_fake = np.mean(np.power(pacchetto_fake - pred_fake, 2))
print(f"\nTest pacchetto inventato (Anomalo):")
print(f"Errore: {errore_fake:.4f} -> {'BLOCCATO' if errore_fake > threshold else
'PASSATO'}")
```

Dove cade KNN e vince NN?

- Immagina di analizzare due caratteristiche di un pacchetto:
    - Numero di istruzioni NOP (spesso usate negli exploit per far scivolare la CPU verso il codice malevolo).
    - Lunghezza del Pacchetto.
- Traffico Normale
    - Pochi NOP, lunghezza variabile (da piccola a grande).
- Attacco Standard
    - Tanti NOP, lunghezza piccola (l'exploit è compatto).
- L'Attacco Furbo (Obfuscated)
    - L'hacker prende l'attacco standard e aggiunge 1000 byte di zeri alla fine.
- Risultato: Il pacchetto ora è lunghissimo.

NN: un’applicazione

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neighbors import KNeighborsClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.preprocessing import StandardScaler

# 1. CREIAMO I DATI (Normale vs Exploit)
# Feature 0: NOP Count (0 = basso, 10 = alto)
# Feature 1: Packet Length (0 = breve, 100 = lungo)

# A) TRAFFICO NORMALE (Classe 0)
# Pochi NOP (0-2), Lunghezza molto variabile (0-100)
# Simuliamo 100 pacchetti normali sparsi
normal_nops = np.random.uniform(0, 2, 100)
normal_len = np.random.uniform(0, 100, 100)
X_normal = np.column_stack((normal_nops, normal_len))
y_normal = np.zeros(100)

# B) EXPLOIT NOTI (Classe 1)
# Tanti NOP (8-10), Lunghezza breve (0-20) perché sono shellcode compatti
exploit_nops = np.random.uniform(8, 10, 20)
exploit_len = np.random.uniform(0, 20, 20)
X_exploit = np.column_stack((exploit_nops, exploit_len))
y_exploit = np.ones(20)

# Uniamo tutto nel Training Set
X_train = np.vstack((X_normal, X_exploit))
y_train = np.hstack((y_normal, y_exploit))

# SCALING (Fondamentale!)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)

# 2. ADDESTRIAMO I MODELLI
# KNN (K=3)
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train_scaled, y_train)

# Rete Neurale (Semplice)
nn = MLPClassifier(hidden_layer_sizes=(16, 8), activation='relu', random_state=42, max_iter=1000)
nn.fit(X_train_scaled, y_train)

# 3. L'ATTACCO FURBO (Il Test)
# L'hacker prende l'exploit (NOP=9) ma aggiunge padding per renderlo LUNGO (Len=90)
# Geometricamente, questo punto è lontano dagli exploit (che sono corti)
# e vicino ai pacchetti normali (che sono lunghi).
attack_obfuscated = np.array([[9.0, 90.0]])
attack_scaled = scaler.transform(attack_obfuscated)

# 4. PREDIZIONI
pred_knn = knn.predict(attack_scaled)[0]
pred_nn = nn.predict(attack_scaled)[0]

# --- VISUALIZZAZIONE ---
plt.figure(figsize=(10, 6))

# Disegniamo i dati di training
plt.scatter(X_normal[:, 1], X_normal[:, 0], c='blue', label='Normale (Training)', alpha=0.6)
plt.scatter(X_exploit[:, 1], X_exploit[:, 0], c='red', label='Exploit Noti (Training)', marker='x', s=100)

# Disegniamo l'attacco furbo
plt.scatter(attack_obfuscated[:, 1], attack_obfuscated[:, 0], c='black', label='ATTACCO FURBO (Padding)',
marker='*', s=300, edgecolors='orange', linewidth=2)

# Titoli e assi
plt.xlabel('Lunghezza Pacchetto (Padding)')
plt.ylabel('Numero NOP (Pericolosità)')
plt.title(f"RISULTATO:\nKNN dice: {'SICURO (Errore)' if pred_knn==0 else 'ATTACCO'}\nNN dice: {'SICURO'
if pred_nn==0 else 'ATTACCO (Corretto)'}")
plt.legend()
plt.grid(True, linestyle='--', alpha=0.5)

# Disegniamo le linee di distanza per far capire perché il KNN sbaglia
# Troviamo il vicino più prossimo nel training set (senza scaling per il grafico)
dists = np.linalg.norm(X_train - attack_obfuscated, axis=1)
nearest_idx = np.argmin(dists)
nearest_point = X_train[nearest_idx]

plt.plot([attack_obfuscated[0][1], nearest_point[1]],
        [attack_obfuscated[0][0], nearest_point[0]],
        'k--', label='Distanza Euclidea (KNN)')

plt.show()
```

Tecniche di AI applicate alla sicurezza: process mining, machine learning, reti neurali

Capire il contesto della cybersecurity

Introdurre AI (ML, process mining, reti neurali) e loro ruoli nella sicurezza

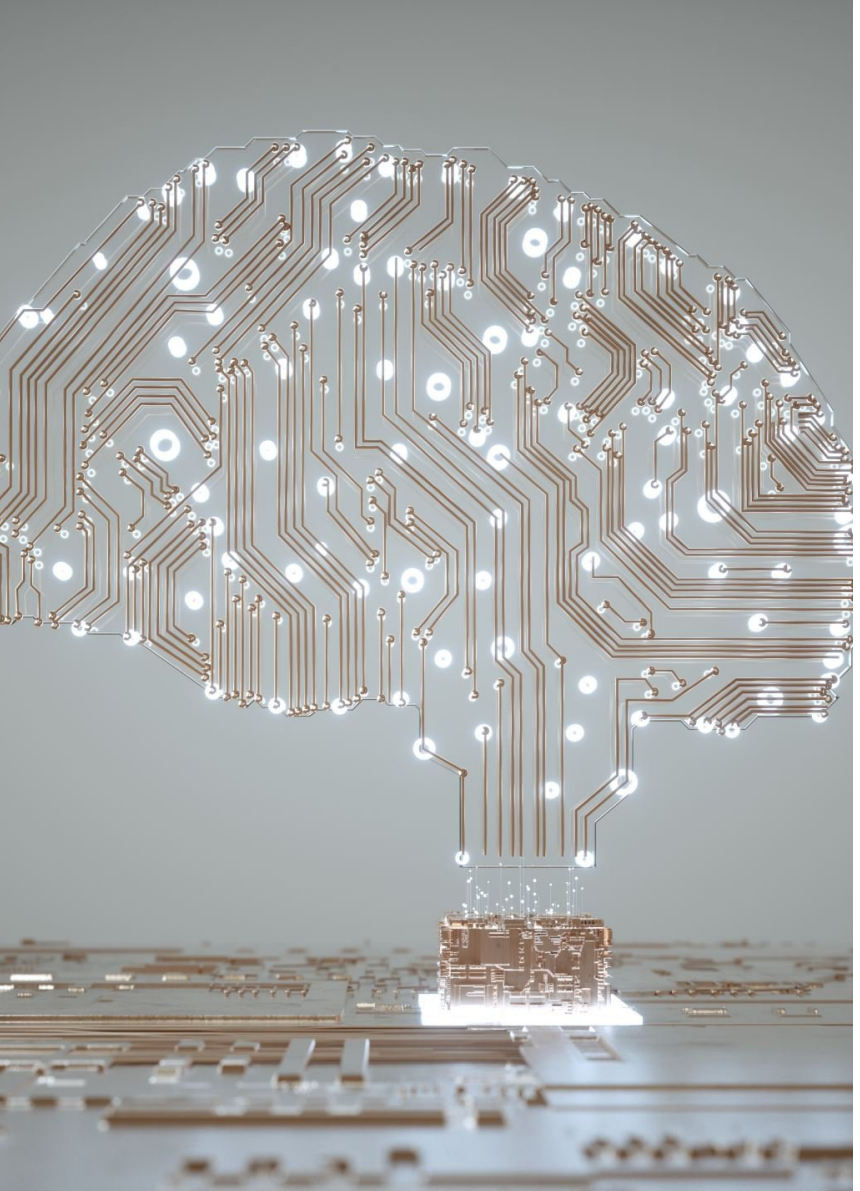
Minacce e monitoraggio

Principali minacce:

Malware, Phishing, DDoS, Insider Threat

Importanza dei log e del monitoraggio di rete:

Rilevamento tempestivo di anomalie
Tracciabilità delle azioni



Che cos’è l’AI?

- AI (Intelligenza Artificiale): insieme di tecniche che permettono ai computer di simulare l’intelligenza umana
- Machine Learning (ML): sottoinsieme dell’AI, basato su algoritmi che “apprendono” dai dati
- Deep Learning: utilizzo di reti neurali profonde per gestire problemi complessi

Dove studiare?

https://python-course.eu/machine-learning/

Operazione fondamentale nell'addestramento e nell'uso di reti neurali

- Prodotto matriciale algebrico (N*M x M*P => N*P)
  - a11 a12 a13 * b11 b12 = a11*b11+a12*b21+a13*b31 a11*b12+a12*b22+a13*b32
    a21 a22 a23 b21 b22 a21*b11+a22*b21+a23*b31 a21*b12+a22*b22+a23*b32
    b31 b32
- Stavolta su una macchina Single instruction Single data (x86, Arm, Risc, …) il numero di operazioni svolte per moltiplicare due matrici N*M x M*P => N*P*M operazioni
  - Il prodotto di due matrici 1000x1000 e 1000x1000 => 1000*1000*1000 => 1 miliardo di operazioni
- Su un'architettura "parallela" è meno di 1000x1000

Operazione fondamentale nell'addestramento e nell'uso di reti neurali

- Prodotto matriciale semplice
  - a11 a12 a13 * b11 b12 b13 = a11*b11 a12*b12 a13*b13
    a21 a22 a23 b21 b22 b23 a21*b21 a22*b22 a23*b23
- Esempio
  - 2 3 4 * 1 2 3 = 2 6 12

    4 5 6 2 2 1 8 10 6
- Costi di esecuzione su una macchina Intelx86 o Arm O Risc
  - Prodotto di due matrici NxM = sono n*m moltiplicazioni, quindi n*m cicli di clock
- Costi di esecuzione su una macchina "parallela" (SIMD), tipo scheda video con 4GB ram
  - Prodotto di due matrici NxM = sono n*m moltiplicazioni eseguite in parallelo, quindi 1 ciclo di clock

Riconoscere una IRIS

- In base a quali caratteristiche riconosciamo se una IRIS è setosa/virginica/versicolor?
  - 4 valori (larghezza e lunghezza dei petali e dei sepali)
- Quale è la dimensione spaziale del riconoscimento di una IRIS (mi serve un singolo valore, una coppia di valori, ecc?)
  - Mi serve un singolo valore
    - 0: setosa
    - 1: virginica
    - 2: versicolor

Riconoscere una IRIS

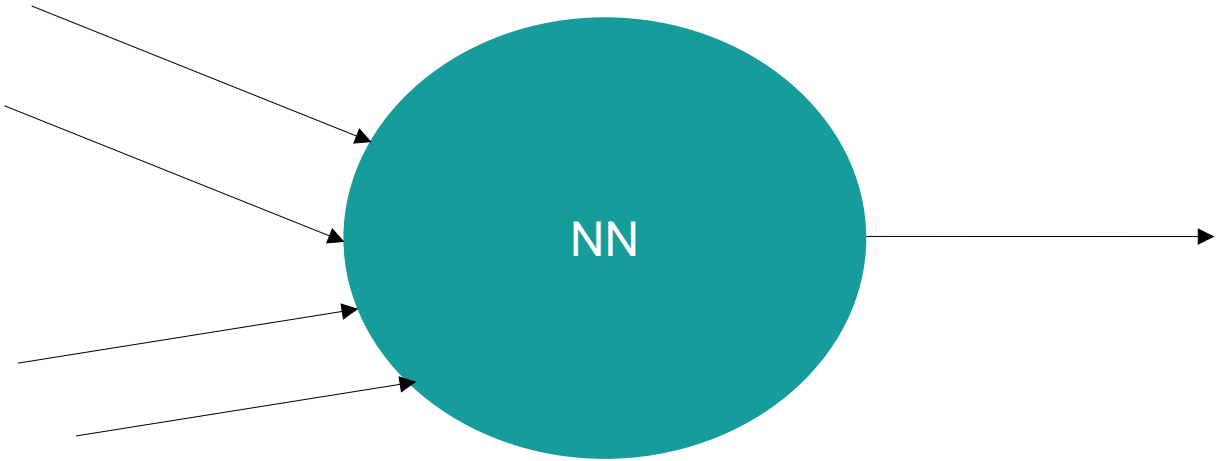
- Quindi, una rete neurale che dovesse essere in grado di riconoscere una IRIS dovrebbe avere

- Input: 4 valori
- Output: 1 valore

Confusion Matrix

	Actually Positive (1)	Actually Negative (0)
Predicted Positive (1)	True Positives (TPs)	False Positives (FPs)
Predicted Negative (0)	False Negatives (FNs)	True Negatives (TNs)

```
Stimate/Reali
            [setosa    versicolor virginica
 setosa     [12        0          0]
 versicolor[ 0         8          1]
 virginica [ 0         0          9]
]
```


NN

Cosa c’è in una rete neurale

- In sostanza c’è una funzione (complessa) che in ingresso richiede tanti parametri quante sono le frecce che entrano nella rete e in uscita ha tanti elementi quante sono le frecce uscenti dalla rete
- Nelle caso delle iris
  - F(x1,x2,x3,x4) => (y1)

Il modo più semplice per implementare la funzione interna di una NN è

X1
X2
X3
x4

W11	W12	W13	W14
W21	W22	W23	W24
W31	W32	W33	W34
W41	W42	W43	W44
W51	W52	W53	W54
W61	W62	W63	w64

Y11
Y12
Y13
Y14

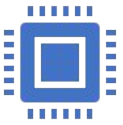
Z1	Z2	Z3	z4

0,1,2

Applicazioni AI in Cybersecurity

Rilevamento Malware: pattern di codice sospetti

Spam Filtering: classificazione automatica delle email indesiderate

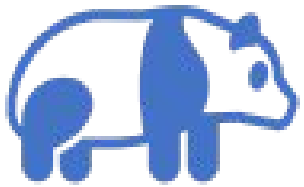


Anomaly Detection: individuazione di comportamenti anomali su reti e sistemi

Process Mining: analisi dei log per ricostruire e monitorare i processi (autenticazioni, flussi di transazione)

Reti Neurali: modelli ispirati al cervello umano, potenti per classificazione e predizione di eventi di sicurezza

Strumenti e Setup



Python e librerie

pandas, scikit-learn, TensorFlow/Keras, pm4py

Ambiente di sviluppo

Jupyter Notebook
Google Colab
VS Code

Installiamo jupyter

- `curl -fsSL https://pyenv.run | bash`
  - Per installare pyenv in modo da poter avere più di una versione di python installata
  - `export PYENV_ROOT="$HOME/.pyenv"`
  - `[[ -d $PYENV_ROOT/bin ]] && export PATH="$PYENV_ROOT/bin:$PATH"`
  - `eval "$(pyenv init - bash)"`
  - `sudo apt update`
  - `sudo apt upgrade`
  - `sudo apt install -y build-essential software-properties-common make gcc zlib1g-dev`
  - `pyenv install 3.10.16`
  - `sudo apt install -y make build-essential software-properties-common libssl-dev zlib1g-dev libbz2-dev libreadline-dev libsqlite3-dev wget curl llvm libncursesw5-dev xz-utils tk-dev libxml2-dev libxmlsec1-dev libffi-dev liblzma-dev python3-distutils graphviz`
- `python3.10 -mvenv .cyber`
- `. .cyber/bin/activate`
- `pip install notebook`
- `pip install jupyter`
- `pip install scikit-learn pm4py pandas`

Usiamo jupyter

- FILE: log_process_mining.csv
- case_id,activity,timestamp
- 1,Login,2022-01-01 08:00:00
- 1,ViewAccount,2022-01-01 08:01:05
- 1,Logout,2022-01-01 08:05:10
- 2,Login,2022-01-01 09:00:00
- 2,Error,2022-01-01 09:01:00
- 2,Logout,2022-01-01 09:02:00
- ==========
- 3,Login,2022-01-01 10:00:00
- 3,Error,2022-01-01 10:01:00
- 3,Register,2022-01-01 10:02:00
- 3,ViewAccount,2022-01-01 10:03:05

Process Mining per la Sicurezza

Cos’è il Process Mining?

Definizione: tecnica di analisi dei “log di eventi” per ricostruire i processi reali

Differenze con il Data Mining classico:

Si focalizza su “attività” e “sequenze temporali”
Utilissimo in ambito sicurezza per tracciare comportamenti e scovare deviazioni

Event Logs in ambito sicurezza

Case ID: identifica l’istanza di un processo (es. sessione utente)

Activity: l’azione compiuta (es. login, query, logout)

Timestamp: il momento in cui l’attività è stata eseguita

Eventuali attributi extra (IP, ruolo utente, ecc.)

Applicazioni in Cybersecurity

1

Analisi comportamenti utenti: rilevamento di pattern anomali nelle autenticazioni

2

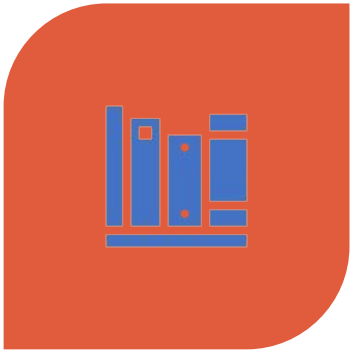
Monitoraggio transazioni: scovare transazioni non autorizzate o fraudolente

3

Analisi catena di approvvigionamento: individuare colli di bottiglia o punti vulnerabili

Strumenti di Process Mining

PROM (OPEN SOURCE)



PM4PY (LIBRERIA PYTHON)

ALTRE SOLUZIONI (DISCO, CELONIS)

Esempio di Workflow Process Mining

- Raccolta log in formato CSV, XES o database
- Import in pm4py
- Scoperta del modello di processo (alpha miner, heuristics miner, ecc.)
- Conformance checking: confronto tra modello “atteso” e log reali per identificare anomalie

Demo / Esercitazione

- Uso di un dataset di log fittizio (CSV)
- Caricamento in pm4py e generazione di un process map
- Ricerca di “deviazioni” (utenti che accedono in orari anomali, troppi tentativi di login errati)

Vantaggi e Limiti

Vantaggi: facile individuazione di “percorsi strani”, pattern insoliti

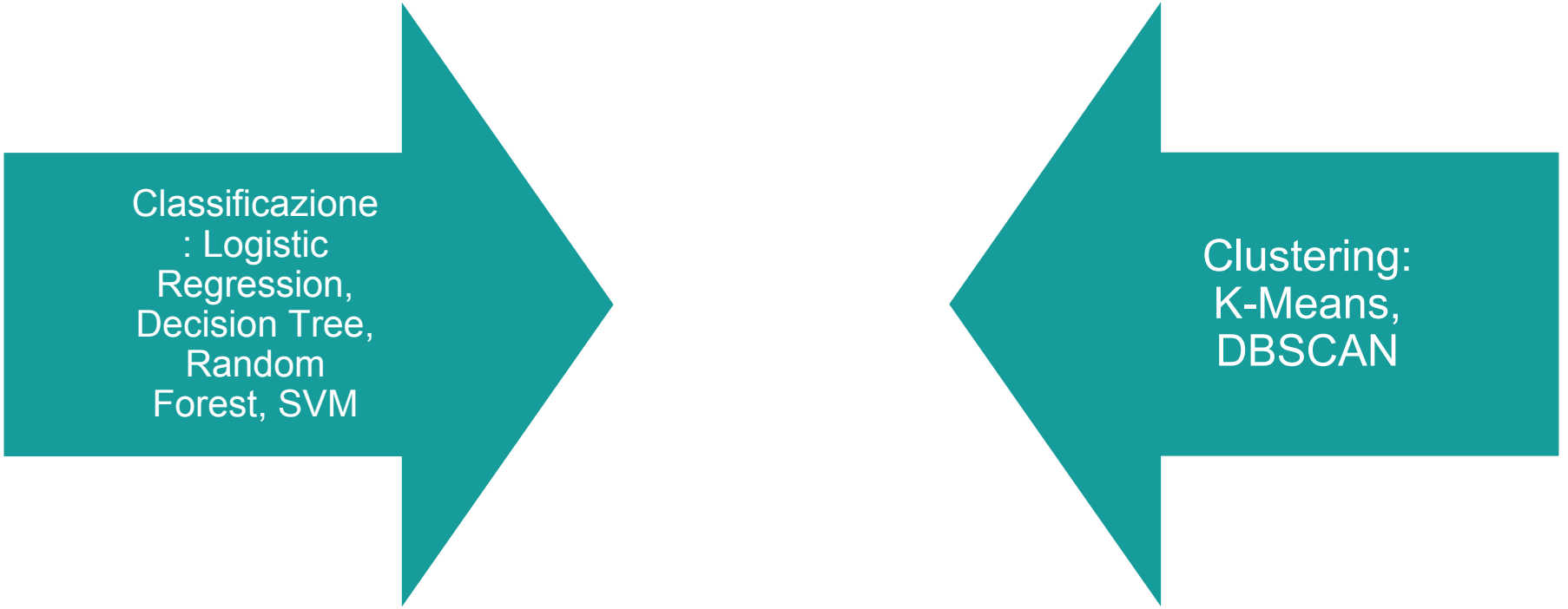
Limiti: qualità dei log; necessità di interpretazione corretta

Fondamenti di Machine Learning per la Cybersecurity

Tipi di apprendimento: Supervisionato (classificazione, regressione), Non supervisionato (clustering), Semi-supervisionato

Metriche: accuracy, precision, recall, F1, ROC AUC

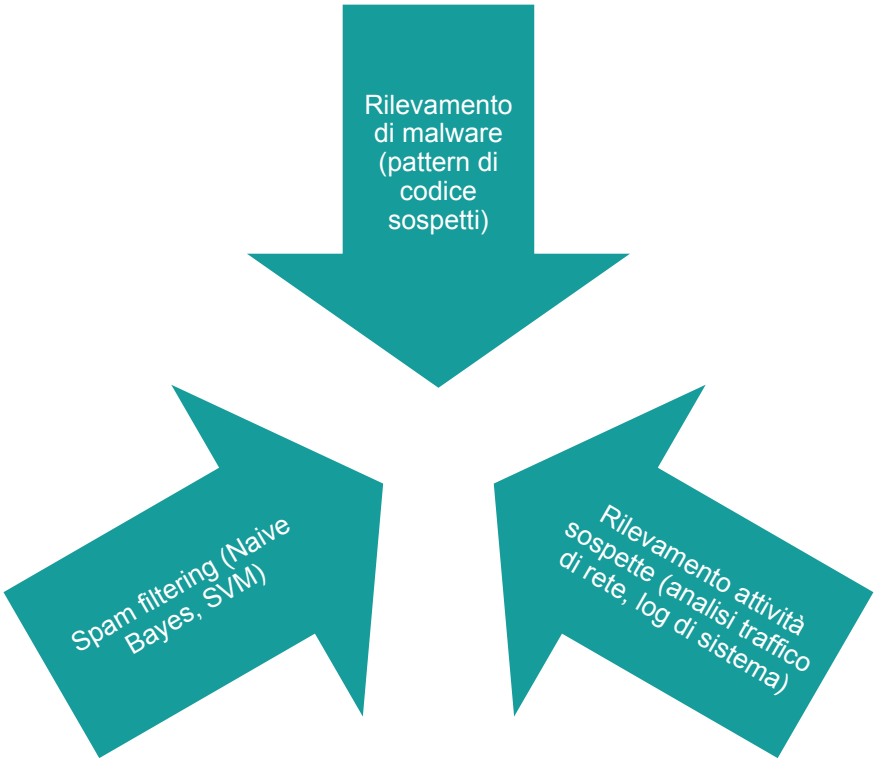
Algoritmi Principali



Dati e Feature Engineering

- Pulizia e gestione dei dati (missing values, encoding)
- Creazione di feature (es. statistiche di rete, tokenizzazione testo)
- Bilanciamento classi (sottocampionamento, sovracampionamento)

Uso in Cybersecurity



Esempio: Spam Detection

Panoramica dell’esercitazione: pipeline di classificazione con dataset Spam/Non-Spam (SpamAssassin)

Demo / Esercitazione

Dataset di esempio (SpamAssassin)

Pipeline:

Estrazione di feature (TF-IDF)

Addestramento modello (Naive Bayes / SVM)

Valutazione (confusion matrix, F1)

Best Practice ML

- Validazione incrociata (cross-validation)
- Hyperparameter tuning (Grid Search, Random Search)
- Salvare e ricaricare il modello (joblib, pickle)

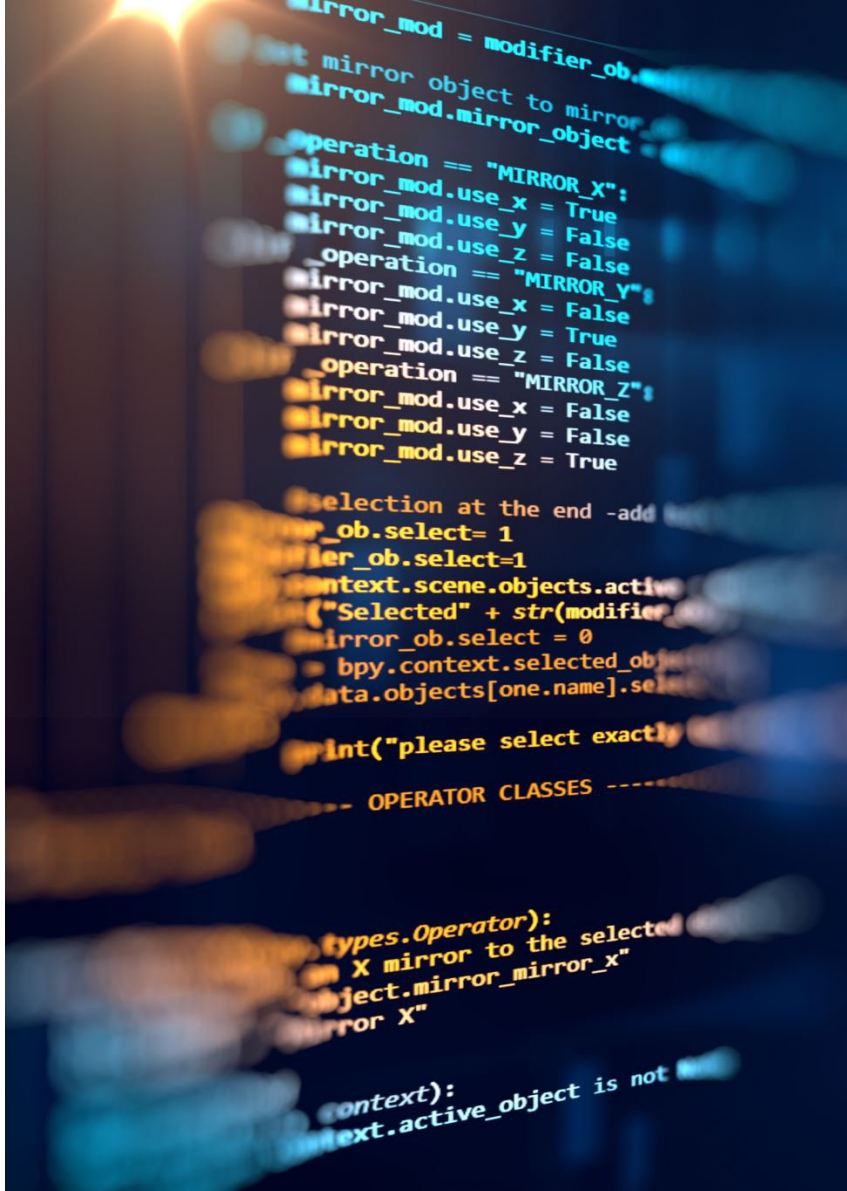
Anomaly Detection e Applicazioni IDS

Anomaly Detection

- Concetto: identificare pattern “rari” o “insoliti”
- Algoritmi: Isolation Forest, One-Class SVM, Local Outlier Factor, Autoencoder (cenni)

IDS (Intrusion Detection System)

- Basato su firme (signature-based) vs basato su anomalie
- Esempi di software: Snort, Zeek, Suricata
- Limiti: i sistemi basati su firme riconoscono solo attacchi già noti



Dataset in Cybersecurity

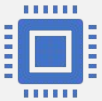
KDD Cup 99, NSL-KDD, CICIDS2017

Struttura tipica: protocollo, IP sorgente/destinazione, porte, flag, ecc.

Preprocessing necessario (label encoding, normalizzazione)

Pipeline di Anomaly Detection

Caricamento dataset



Preprocessing (scaling, encoding)

Addestramento Isolation Forest (ad es.)

Analisi dei risultati (matrice di confusione)

Demo / Esercitazione

Esempio con NSL-KDD e Isolation Forest

Confronto con un modello di classificazione supervisionata

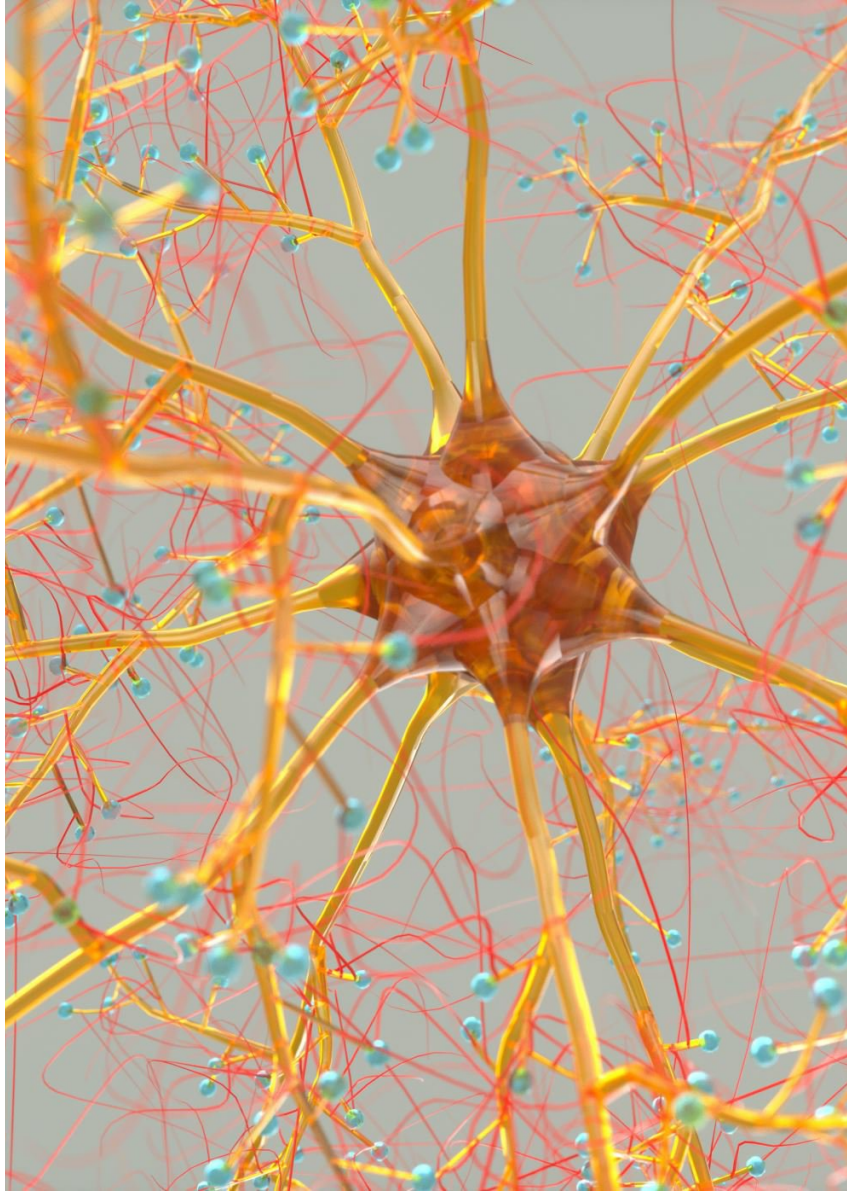
Conclusioni

IMPORTANZA DI COMBINARE METODI ANOMALY-BASED CON REGOLE/SISTEMI DI FIRMA

NECESSITÀ DI AGGIORNARE COSTANTEMENTE I MODELLI

Fondamenti di Reti Neurali (ANN)

- Neurone Artificiale
  - Ingresso (input), pesi, funzione di attivazione, uscita (output)
  - Confronto con neuroni biologici



Struttura di una MLP (Multi-Layer Perceptron)

Layer input, hidden e output

Funzioni di attivazione (ReLU, Sigmoid, Tanh)

Fasi di Addestramento

Forward pass e Backward pass (backpropagation)

Funzione di perdita (MSE, cross-entropy)

Ottimizzatori (SGD, Adam)

Strumenti Principali

- TensorFlow/Keras
- PyTorch
- Esempio di un semplice modello in Keras (codice Python)

Esempio Applicativo in Cybersecurity

Classificazione di pacchetti di rete (normale vs anomalo)

Riconoscimento pattern di attacco

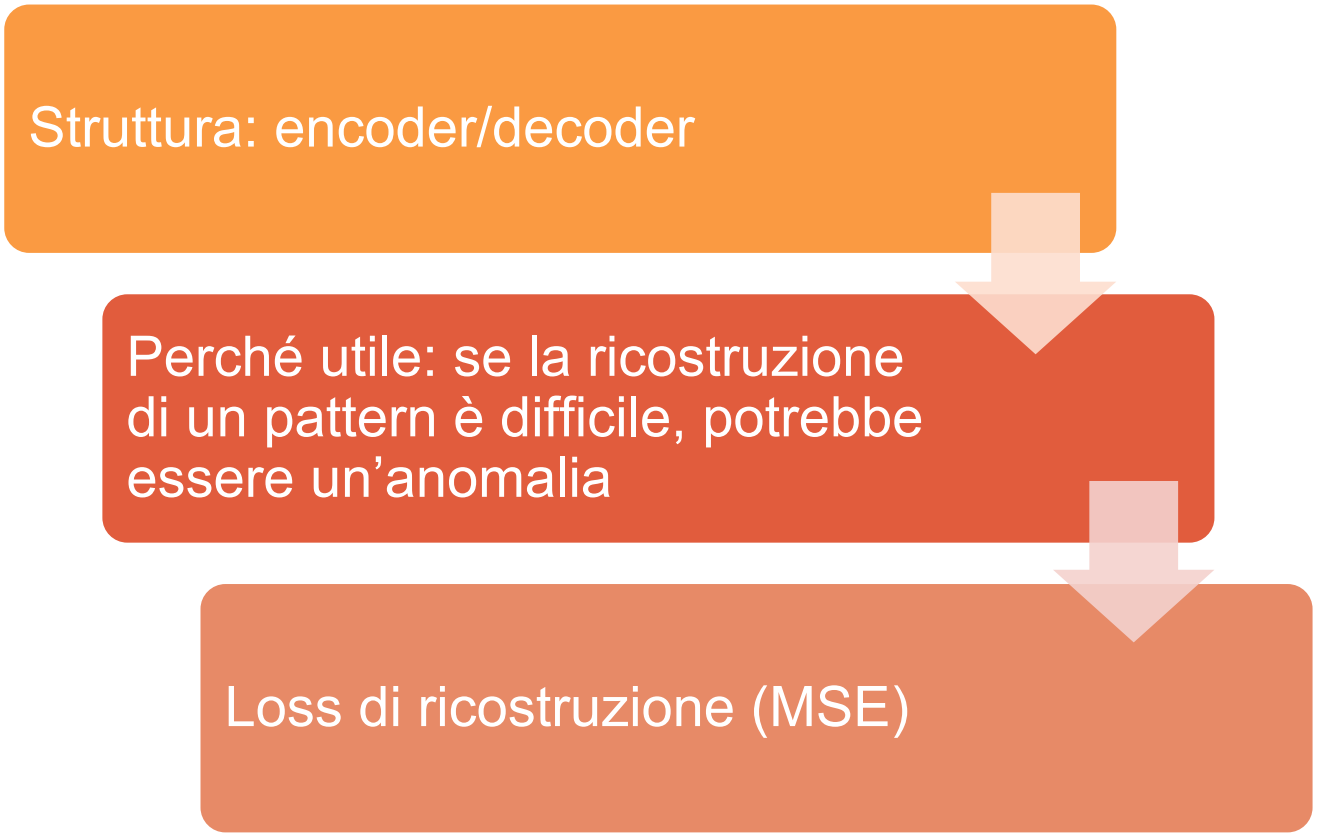
Demo / Esercitazione

Costruzione di una piccola rete neurale feed-forward

Dataset di traffico di rete (es. subset NSL-KDD)

Training e valutazione (accuracy, precision/recall)

Reti Neurali Avanzate (Autoencoder, CNN, RNN)



CNN (Cenni)


Convoluzioni e Pooling
Riconoscimento di pattern spaziali (o binari)
Possibili applicazioni su payload di rete (vista come “immagine” di byte)

RNN (Cenni)

Reti Ricorrenti: LSTM, GRU

Gestione dei dati sequenziali/temporali (log di sistema in serie temporale)

Esempi

Autoencoder su log di autenticazione considerati “normali”

Calcolo dell’errore di ricostruzione per identificare tentativi di login anomali

Demo / Esercitazione

Creazione di un autoencoder con Keras

Training su dati normali, test su dati anomali

Calcolo soglia di errore e interpretazione

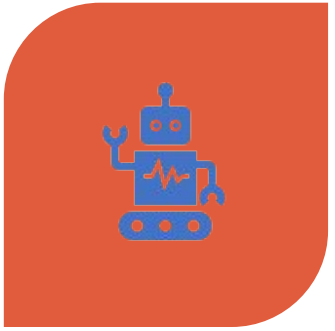
Integrazione Real-Time e Pipeline di Sicurezza

Reattività immediata agli attacchi

Necessità di elaborare flussi di dati continui (log, eventi)

Architettura di un SIEM (security information and event management)

RACCOLTA LOG (AGENT, SYSLOG, ETC.)



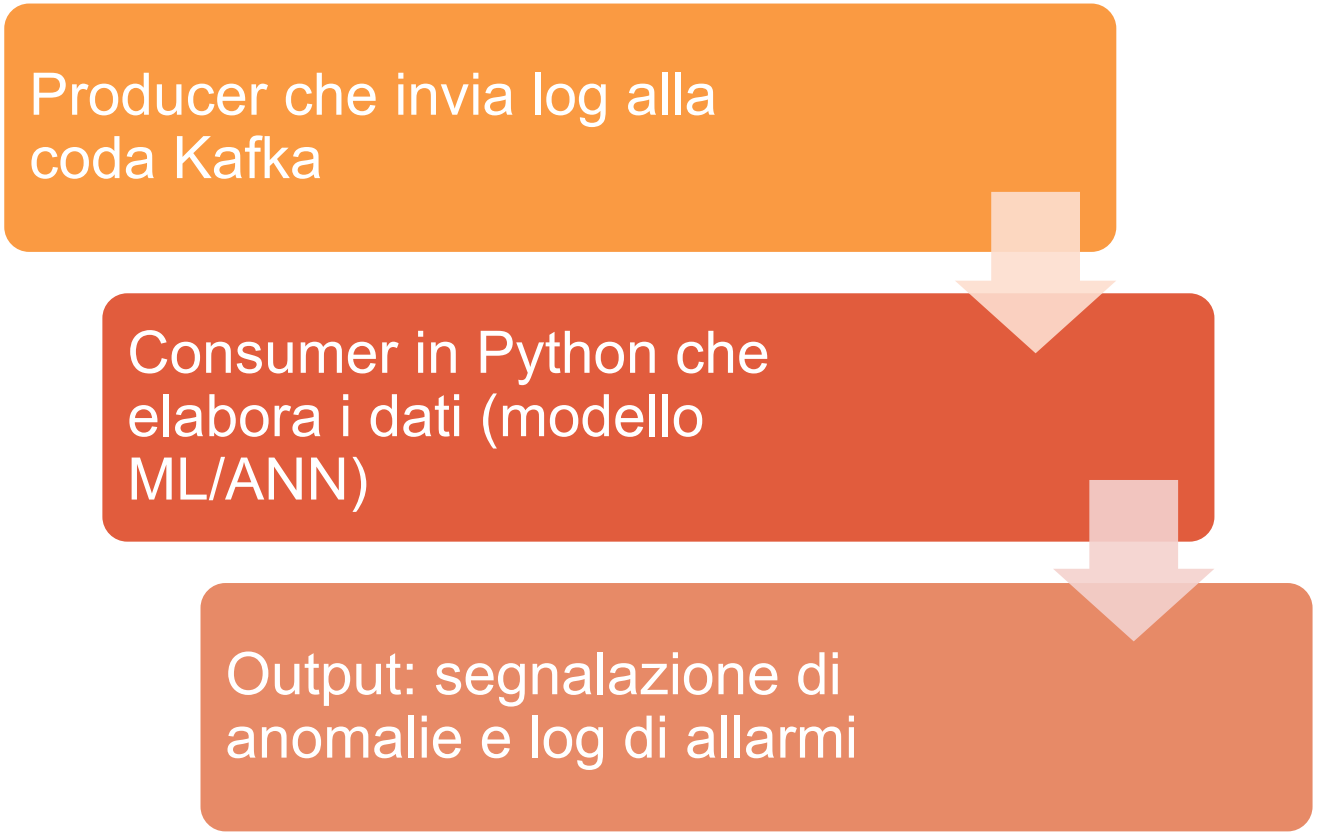
NORMALIZZAZIONE E ANALISI (AI/ML)

DASHBOARD E ALLARMI (EMAIL, SLACK, ECC.)

Strumenti di Streaming

- Apache Kafka, RabbitMQ
- Concetto di producer e consumer
- Micro-batch vs streaming continuo

Esempio di Pipeline con Python



Demo / Esercitazione

Simulazione di un flusso di log

Caricamento di un modello pre-addestrato (Isolation Forest)

Emissione di “alert” quando predice anomalia

Yolo - Ultralytics

- docker run --rm -it --net host --privileged --gpus="all" --volume "$PWD":/vapps -e "DISPLAY=$DISPLAY" --env XDG_RUNTIME_DIR=$XDG_RUNTIME_DIR --env QT_QPA_PLATFORM=xcb -e NVIDIA_DRIVER_CAPABILITIES=all -v /tmp/.X11-unix:/tmp/.X11-unix:rw --device /dev/dri --device /dev/bus/usb:/dev/bus/usb --volume="$HOME/.Xauthority:/root/.Xauthority:rw" ultralytics/ultralytics:latest /bin/bash -c "cd /vapps; /bin/bash -i"
- Se non avete la GPU allora eliminare
  - --gpus="all"
  - e NVIDIA_DRIVER_CAPABILITIES=all

Yolo – you only look once

```
import cv2
from ultralytics import YOLO
# 1. Carichiamo il modello YOLOv8 Nano (il più veloce su CPU)
model = YOLO('yolov8n.pt')
# 2. Apriamo il file video (o la webcam mettendo 0)
video_path = "video.mp4" # Sostituisci con il tuo file
cap = cv2.VideoCapture(video_path)
print("Analisi in corso... Premi 'q' per uscire.")
while cap.isOpened():
    success, frame = cap.read()
    if not success:
        break
    # 3. Eseguiamo il rilevamento solo sulle PERSONE (classe 0 in YOLO)
    # device='cpu' forza l'uso del processore
    results = model.predict(frame, classes=[0], conf=0.4, device='cpu', verbose=False)
    # 4. Conteggio delle persone rilevate
    person_count = len(results[0].boxes)
    # 5. Visualizzazione dei risultati sul frame
    annotated_frame = results[0].plot() # Disegna i rettangoli attorno alle persone
    # Logica di allarme: se superiore a 5 persone
    color = (0, 0, 255) if person_count > 5 else (0, 255, 0)
    label = f"PERSONE: {person_count} - ALLARME ASSEMBRAMENTO!" if person_count > 5 else f"PERSONE: {person_count}"
    cv2.putText(annotated_frame, label, (50, 50), cv2.FONT_HERSHEY_SIMPLEX,
                1, color, 3, cv2.LINE_AA)
    # 6. Mostra il video
    cv2.imshow("Monitoraggio Assembramenti", annotated_frame)
    # Esci premendo 'q'
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break
cap.release()
cv2.destroyAllWindows()
```

Uso delle immagini per rappresentare attacchi di rete

- Il metodo "Matrice di Pixel" (Grayscale)
  - Ogni pacchetto di rete o ogni sessione viene trasformata in un'immagine dove ogni pixel rappresenta un valore (byte).
  - Prendi i primi 784 byte di un flusso di rete.
  - Li disponi in una griglia 28x28.
  - Ogni valore di byte (da 0 a 255) diventa l'intensità di grigio di un pixel.
  - Risultato: Un attacco DDoS apparirà come una trama molto regolare e ripetitiva, mentre una navigazione normale sembrerà "rumore" casuale.
- Immagini da Metadati (Feature Maps)
  - Invece di usare i byte grezzi, usi le feature che abbiamo visto nel dataset NSL-KDD (durata, byte inviati, numero di errori).
  - Se hai 36 feature, le disponi in una griglia 6x6.
  - Le colori in base alla loro importanza.
  - Ultralytics/YOLO può quindi scansionare queste "mappe" per cercare oggetti (anomalie) che corrispondono a firme di attacco.
- Grafi di Relazione (Network Topology)
  - Si trasforma il traffico in un grafo dove i nodi sono gli IP e le linee sono le connessioni.
  - Un attacco Port Scan apparirà visivamente come un "ventaglio" (un unico punto che si connette a tantissimi altri).
  - Un'infezione Botnet sembrerà una stella o una rete a maglie fitte.

Ne vale la pena?

Metodo	Difficoltà Addestramento	Potenza Richiesta	Capacità di Analisi
Autoencoder (Numerico)	Bassa	Ottima su CPU	Trova anomalie matematiche
YOLO (Immagini)	Alta	✕ Serve GPU	Trova pattern visivi e temporali
KNN / RF (Classico)	Molto Bassa	Leggera	Ottima per attacchi noti