Progetto Gotcha

A.A. 2024-25

Dambra Antonio (Matricola 726459)

a.dambra6@studenti.uniba.it

`github.com/anto31ad/gotcha-system` (repository)

Indice

1 Introduzione

Gotcha! (formalmente *gotcha-system* o semplicemente *Gotcha*) è un *proof of concept* di un sistema esperto intelligente di supporto per la sicurezza dei sistemi informativi che operano attraverso Internet.

Il compito generale di Gotcha è di monitorare le interazioni tra il sistema informativo interno e l'utenza esterna (gli *eventi*) segnalare quelle *anomale* e permettere al controllore di specificare delle *azioni correttive*.

1.1 Terminologia

Prima di discutere i dettagli di Gotcha, occorre introdurre la terminologia essenziale.

1.1.1 Eventi

Come accennato, per *evento* si intende una transazione o interazione che avviene tra il sistema che si intende proteggere (es. un sito web) e l'esterno (es. un browser)

Gli eventi possono ricadere in categorie quali:

- accessi in lettura e scrittura alla base di dati (es. prodotti di un catalogo, personale, etc.)
- accessi ai servizi applicativi mediante browser Web (es. cloud storage, email, etc.)
- creazione di nuovi account (es. in siti e-commerce, siti ministeriali, etc.)
- e infiniti altri casi d'uso.

1.1.2 Eventi anomali

Un evento è *anomalo* nel momento in cui si discosta significativamente dagli altri eventi già verificati fino a quel momento sulla base di un qualche criterio;

Seguono alcuni esempi di anomalie:

- l'utente `Bob`, magari privilegiato, che normalmente effettua un accesso tra le 8 e le 18, improvvisamente effettua un accesso alle 2 di mattina;
- l'utente `Alice` esegue più tentativi di login, tutti fallimentari, magari anche in una finestra temporale molto piccola;
- l'utente privilegiato `admin`, esegue molte operazioni di `edit` in una finestra temporale piccola
- l'utente privilegiato `admin`, la cui sessione normalmente dura al più *60m*, rimane connesso per *120m* (magari eseguendo molte operazioni)
- e cosi via...

1.1.3 Azioni correttive

Nel momento in cui si riscontrano delle anomalie, seguirà sempre una **azione correttiva** da parte del controllore umano (o eventualmente automatico);

Una *azione correttiva* consiste nel decidere se l'anomalia

- è un sintomo di un attacco informatico che deve essere fermato (*minaccia*), oppure
- è, in realtà, un banale evento da considerare come un *falso positivo*. (es. "*Admin* si è ricordato di non aver inserito un collegamento ipertestuale importante, quindi accede al sistema dopo cena per correggere"),

1.2 Dominio applicativo

Considerate le ristrettezze in tempo e di manodopera, ho deciso di modellare Gotcha attorno ad un sistema fittizio chiamato "*il Manifesto*" (o solo *Manifesto*) che, in poche parole, rappresenta una singola pagina web che contiene un documento importante per una certa agenda politica.

Questa pagina web può essere modificata (operazione `edit`) solo da una ristretta cerchia di utenti (es. dal nome `admin`, `root`), mentre a tutti gli altri è consentito solo leggerlo.

Essendo *il Manifesto* parte di un mondo fittizio, tutti i dati che Gotcha utilizza sono a loro volta fittizi; Quindi, il **dataset** è generato in parte randomicamente e in parte tenendo conto delle regole di buon senso accennate più sopra, ovvero dettate dall'uso normale del sito Manifesto.

In altre parole, il dataset contiene eventi frequenti (la normalità) ed eventi rari (le anomalie);

Questa caratteristica del dataset è cruciale per poter costruire un rilevatore di anomalie, perchè se tutti gli eventi fossero troppo simili oppure completamente randomici, le tecniche di apprendimento automatico (approfondite nel corso dei capitoli successivi) sarebbero di poco aiuto.

2 Copertura degli argomenti del corso

Rispetto al programma del corso di Ingegneria della Conoscenza, Gotcha fa uso di metodi e tecniche relativi ai seguenti argomenti:

- **Rappresentazione proposizionale della conoscenza** (logica proposizionale, clausole definite, CWA, NAF, ...);
- **Rappresentazione e ragionamento relazionale** (Concettualizzazione, logica dei predicati, ...);
- **Apprendimento non supervisionato** (clustering, kMeans, regola del gomito, ...).

3 Concettualizzazione

Prima di poter realizzare qualunque sistema informatico, intelligente o meno, il primo passo è la *concettualizzazione*: chiarire chi sono le entità (dette **individui**) e le **relazioni** tra di esse, che sono rilevanti rispetto al compito da svolgere.

Il risultato della concettualizzazione sarà una descrizione formale di un *dominio/mondo*, adeguata per la computazione.

Come accennato nei paragrafi precedenti, il nostro dominio di interesse è quello del *Manifesto*: un sito Web che espone un semplice documento.

Tale dominio può essere formalizzato osservandolo da vari punti di vista: procediamo per gradi.

3.1 Utenti

Abbiamo anche detto che il Manifesto permette una operazione di modifica (`edit`) alla quale hanno accesso solo gli **utenti privilegiati** (`super_user`); Questo implica che esistono anche utenti **non privilegiati**, alla quale non è concesso fare `edit`.

Quindi possiamo definire un **utente** del Manifesto come una coppia `(Name, Super)`, dove

- `Name` è una sequenza di caratteri che identifica univocamente l'utente in Manifesto (es. `bob`)
- `Super` è una variabile booleana che specifica se l'utente è un utente privilegiato (i.e. *super user*) o meno.

3.2 Operazioni disponibili agli utenti

In Manifesto, le **operazioni disponibili** per gli utenti sono le seguenti

- `login` (tutti gli utenti) : avvia una sessione (es. per lettura) in Manifesto
- `logout` (tutti gli utenti) : chiudi la sessione corrente in Manifesto
- `edit` (solo super user) : aggiorna il contenuto del Manifesto

Quando l'utente esegue una di queste operazioni, viene creata una *istanza* di tale operazione, che può essere descritta in termini di una tupla

`(SessionID, Date, Time, User, Action)`

dove

- `SessionID` : l'identificativo della sessione (descritta nel dettaglio più sotto) a cui appartiene l'istanza;
- `Date` : il giorno in cui viene eseguita l'operazione;
- `Time` : l'ora del giorno in cui viene eseguita l'operazione;
- `User` : l'identificativo dell'utente che esegue l'operazione;
- `Action` : l'identificativo della operazione, cioè uno tra quelli descritti sopra (es. `login`).

La seguente tabella mostra un esempio di sequenza di istanze di operazioni:

SessionID	Date	Time	User	Action
abc123	2025-05-07	21:51	root	login
def456	2025-05-07	22:04	bob	login
abc123	2025-05-07	22:12	root	edit
ghi789	2025-05-07	22:19	alice	login
ghi789	2025-05-07	22:24	alice	logout
abc123	2025-05-07	22:34	root	logout
def456	2025-05-07	22:47	bob	logout

3.3 Sessione degli utenti

Con il termine *sessione*, menzionato prima, si intende una sequenza di istanze di operazioni che iniziano con un istanza di `login` e terminano con una di `logout` relative ad uno stesso utente.

Riprendendo la precendente tabella (con esempi di istanze), esempi di sessioni sono

- Utente `root`, sequenza `login (21:51) => edit (22.12) => logout (22.34)`
- Utente `bob`, sequenza `login (22:04) => logout (22:47)`

Questo concetto è importante per ragionare, ad esempio, su

- le differenze di tempo tra una operazione e l'altra;
- il numero di sessioni giornaliere;
- la distanza di tempo tra una sessione e un altra;

Queste misure sono utili per individuare ulteriori anomalie rispetto a quelle relative alle singole istanze di operazione.

> **N.B.** : per rispettare la scadenza di consegna, purtroppo sono riuscito a realizzare in Gotcha funzionalità che illustrino come queste misure si sarebbero potute usare per rilevare anomalie.

3.4 Vincoli per le operazioni e stato dell'utente

A questo punto, osserviamo che le suddette operazioni non possono essere eseguite in qualsiasi circostanza:

Ad esempio, per eseguire un `edit`, l'utente deve aver prima eseguito un `login` e solo nel caso in cui sia un super user, allora potrà procedere.

Più in generale, ogni operazione può essere eseguita solo quando l'utente si trova in specifici **stati**.

Nel caso del Manifesto, si individuano due stati per ogni utente:

- `online` : l'utente ha una sessione di Manifesto attiva (non termina con `logout`);
- `offline` : l'utente non è coinvolto in alcuna sessione attiva;

Segue che

- utente nello stato `offline` può eseguire solo `login`.
- utente nello stato `online` può eseguire solo `edit` (se super user) o `logout`.

Tutte le altre combinazioni non sono lecite; Se si dovessero verificare, sarebbero le prime a costituire una anomalia.

3.5 Eventi

Nel capitolo introduttivo abbiamo parlato degli eventi (i.e. interazioni tra il sistema da proteggere e l'esterno) come un concetto fondamentale per il rilevatore di anomalie.

In Manifesto, gli **eventi coincidono con le istanze delle operazioni** descritte poco sopra, ovvero con ogni riga della tabella di esempio.

4 Sommario

Questa sezione fornisce una panoramica del funzionamento di Gotcha e dei contenuti dei capitoli successivi.

4.1 Funzionamento generale di Gotcha

Il seguente diagramma illustra il funzionamento generale di Gotcha:

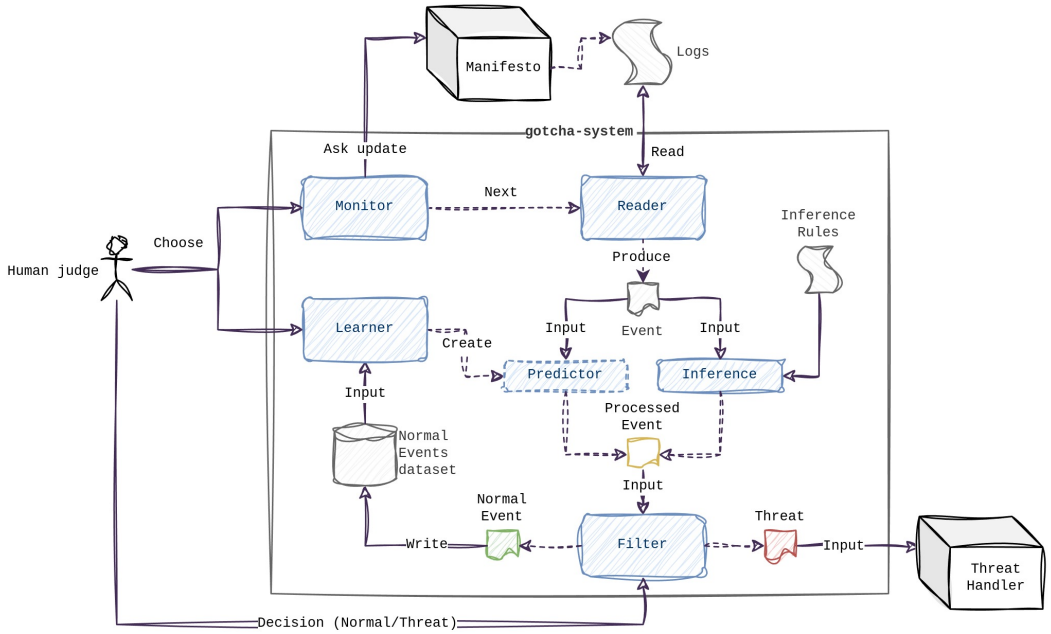

Figure 1: Architettura

Commentiamo:

Il giudice umano (*human judge*) è l'utente di Gotcha; egli può scegliere di iniziare a monitorare Manifesto oppure di addestrare, attraverso il componente *Learner*, un modello predittivo sui dati passati (cioè il *Normal Events Dataset*).

Supponendo di partire da zero, il giudice inizia a monitorare Manifesto;

il componente responsabile (*Monitor*) chiede a Manifesto di fornire degli aggiornamenti sul suo stato;

Manifesto risponde con una sequenza di eventi, nella forma di *Logs*;

I *Logs* sono processati dal *Reader*, che produce una lista di eventi (gli *Event*) che verrà sottoposta ai componenti *Inference* (per il ragionamento logico) e *Predictor* (il modello predittivo); Dato che siamo partiti da zero, in questa fase *Predictor* non è disponibile.

Quindi, *Inference* analizza gli eventi utilizzando delle regole di inferenza (i.e. *Inference Rules*) e produce una lista di eventi sospetti (i.e. lista di *Processed Event*).

Gli eventi sospetti sono sottoposti al filtraggio (componente *Filter*) che richiede l'intervento del giudice (l'arco *Decision* in figura) per stabilire se si tratta di un falso positivo (*Normal Event*) o di una reale minaccia (*Threat*):

- Le minacce sono gestite esternamente da un altro sistema (*Threat Handler*);
- Gli eventi normali popolano il dataset degli eventi normali (*Normal Events Dataset*).

A questo punto è stata completata una iterazione e pertanto si torna al primo passo.

Ora, se il dataset è abbastanza popolato, il giudice potrà scegliere di addestrare il modello predittivo (*Predictor*) a partire da essi; idealmente *Predictor* rende i rilevamenti più precisi, sopperendo alle mancanze di *Inference*.

4.2 Implementazione di Gotcha

Il diagramma che abbiamo descritto illustra una idea molto generale del funzionamento di Gotcha;

Nel concreto, sono stati realizzati i seguenti componenti:

- un **generatore del dataset** che simula il comportamento del sistema esterno *Manifesto* quando riceve richieste dal *Monitor*;
- una **unità di ragionamento logico** per realizzare *Inference* e le *Inference Rules*;
- una **unità di apprendimento e classificazione** (per realizzare *Learner* e *Predictor*) che si occupa di
  - studiare gli orari di attività di ciascun utente (*apprendimento*), in modo da...
  - costruire un classificatore di anomalie per ciascuno di tali utenti (*classificazione*);
- una **unità di controllo** (per realizzare *Monitor*, *Reader* e *Filter*) che consente all'utente (*human judge*) di utilizzare Gotcha;

I prossimi capitoli esamineranno, uno per volta, i suddetti componenti.

5 Generazione del dataset

Come accennato nella sezione introduttiva e precedente, il dataset contenente gli eventi di Manifesto è generato in modo *parzialmente* casuale.

L'obiettivo è di simulare delle sessioni realistiche, che esibiscano sia comportamenti frequenti che rari;

I comportamenti frequenti rappresentano l'uso normale di Manifesto, mentre quelli rari potrebbero essere indicatrici di minacce.

Segue una panoramica delle fasi di generazione, secondo una approccio bottom-up.

> Tutti gli snippet di codice di questo capitolo sono estratti da `app/manifesto.py`

5.1 Definizione degli utenti

I possibili utenti del nostro dataset fittizio sono `admin, root, alice, bob, eve, carol`, di cui `admin, root` sono super utenti.

Per ciascuno di questi utenti ho specificato anche una proprietà fondamentale per l'apprendimento non supervisionato (discusso nell'omonimo capitolo): le fascie orarie di attività (`time_ranges`).

Per esempio, ho stabilito che l'utente `alice` è tipicamente operativo nelle fasce orarie (approx.) tra le *6.40-13.00* e le *14.20-18.20*; Questa informazione, assieme alle altre relative ad `alice` è codificata nel seguente modo:

```
User(
    name='alice',
    is_super=False,
    time_ranges=[range(400, 780), range(860, 1100)]
)
```

Si noti che, per rappresentare l'ora del giorno, ho scelto di utilizzare i **minuti dopo la mezzanotte** poichè, essendo codificabili come interi, hanno facilitato la costruzione del modello predittivo che sarà discusso nell'apposito capitolo.

5.2 Generare la prossima azione di un utente

La prima operazione elementare è quella di specificare l'azione successiva di un utente;

come accennato, il tipo di operazione dipende dallo stato dell'utente:

- Se l'utente è offline, allora la prossima azione dovrebbe essere `login` (scelta deterministica)
- Se l'utente è online, allora la prossima azione può essere un `edit` (se super utente) o `logout`. (scelta non deterministica)

Poichè vogliamo rendere queste azioni più imprevedibili, ammettendo eventi rari, potremmo far si che, a volte

- L'utente è offline, ma effettua di nuovo `logout` o addirittura `edit`
- L'utente è online, ma effettua nuovamente un `login`
- L'utente non è super, ma effettua un `edit`

Questo significa realizzare una macchina a due stati (online, offline) non deterministica, dove da ogni stato partono tre archi con probabilità diverse di essere percorsi;

5.2.1 Azioni nello stato online

Concretamente, **per lo stato online**, ho realizzato una *funzione di ripartizione* delle probabilità (i.e. *Cumulative Distribution Function*) dell'azione successiva per ciascuna categoria di utenti.

Per gli utenti **non privilegiati** la distribuzione di probabiltà è la seguente

```
def _pick_next_unpriviledged_user_action():
    perc = random.random()
    if perc < 0.1:
        # probabilità del 10% di doppio login
        return UserAction.LOGIN
    elif perc < 0.15:
        # probabilità del 5% di edit (raro)
        return UserAction.EDIT
    elif perc < 0.9:
        # probabilità del 75% di logout
        return UserAction.LOGOUT
    else:
        # probabilità restante di inattività
        return UserAction.NONE
```

Invece, per gli utenti privilegiati (**super utenti**), la distribuzione è la seguente:

```
def _pick_next_super_user_action():
    perc = random.random()
    if perc < 0.05:
        # probabilità del 5% di doppio login
```

```
        return UserAction.LOGIN
    elif perc < 0.40:
        # probabilità del 35% di logout
        return UserAction.LOGOUT
    elif perc < 0.80:
        # probabilità del 40% di edit
        return UserAction.EDIT
    else:
        # probabilità restante (20%) di inattività
        return UserAction.NONE
```

Si tenga conto che:

- Le soglie sono state scelte in modo arbitrario.
- `UserAction.NONE` è utile per far scorrere il tempo, come sarà illustrato nel prossimo paragrafo.

5.2.2 Azioni nello stato offline

Per non complicare ulteriormente le cose, si è preferito fare si che nello stato offline, l'utente effettui sempre un `login`; Questa scelta sarà resa evidente dal prossimo paragrafo.

5.3 Generare una sessione

Ricordiamo che una sessione è una sequenza di eventi (o azioni, per semplificare).

Dato un utente, vogliamo specificare una sequenza di azioni utilizzando i generatori definiti poco sopra.

Per raggiungere tale obiettivo, ho realizzato la seguente funzione (opportunamente commentata):

```
def _generate_user_session(
# dati di input:
        # data-ora di partenza
        start_datetime: datetime,
        # nome utente
        user: str,
        # flag "é super user?"
        superuser: bool
# output: una lista di eventi e la durata (in minuti) della sessione
) -> tuple[list[Event], int]:

    # inizializza la sequenza di eventi della sessione
    events: list[Event] = []
    # genera un id di sessione casuale
    session_id = _random_session_id()
    # se l'utente è un super utente, usa il relativo generatore;
    # altrimenti usa quello per utenti non privilegiati
    next_action_func = _pick_next_super_user_action \
        if superuser else _pick_next_unpriviledged_user_action
    # inizializza minutaggio
    tot_min_elapsed = 0
    # la prima azione è il login
    cur_action = UserAction.LOGIN
    # flag per evitare certe operazioni durante la prima iterazione
    first_iteration = True
    # flag per uscire dal ciclo
    stop = False
    while not stop:
        # avanza di un certo numero di minuti tra 15 e 60
```

```
        tot_min_elapsed += random.randint(15, 60)

        if first_iteration:
            first_iteration = False
        else:
            # dalla 2a iterazione in poi, genera una azione casuale
            cur_action = next_action_func()
            if cur_action == UserAction.NONE:
                continue

        # calcola data-ora corrente attraverso i minuti trascorsi
        #   a partire dalla data-ora di partenza
        cur_datetime = start_datetime + timedelta(minutes=tot_min_elapsed)

        # crea ed aggiungi un evento alla sequenza
        events.append( Event(
            session_id=session_id,
            date=_datestr_from_datetime(cur_datetime),
            time=_minutes_from_datetime(cur_datetime),
            user=user,
            action=cur_action))

        # se l'azione generata è un `logout`,
        # nel 20% dei casi effettua continua con un'altra iterazione.
        # (anomalia)
        if cur_action == UserAction.LOGOUT:
            chance_of_logout_success = random.random()
            stop = True if chance_of_logout_success < 0.8 else False

    return (events, tot_min_elapsed)
```

Concentrandoci sulle azioni, i passi principali sono

1. genera un evento `login` e lo aggiunge alla sequenza; (i.e. l'utente passa allo stato online con 100% probabilità, come anticipato)
2. genera un azione casuale (*azione corrente*) tra `login, edit, logout` e `none` utilizzando la funzione di ripartizione (`next_action_func`);
3. crea un evento con l'azione corrente (diversa da `none`) e lo aggiunge alla sequenza.
4. se l'azione corrente **non è** `logout`, torna al passo 2;
5. se l'azione corrente è `logout`, si ha il 20% di possibilità che torni al passo 2; altrimenti stop.

5.4 Generare una sessione *per ogni* utente

Ora che abbiamo un generatore di sessione (`_generate_user_session`) per un utente qualsiasi, possiamo generare più sessioni per ciascun utente.

Nello specifico, i passi principali sono i seguenti

1. per ogni utente
    1. se l'ora corrente ricade nel suo time range, genera una sessione con una certa probabilità (e.g. 70%)
    2. se l'ora non ricade nel suo time range, genera una sessione con probabilità minore (e.g. 20%)
    3. avanza il tempo dei minuti restituiti da `_generate_user_session`
2. ordina gli eventi per data-ora
3. fornisci in output la sequenza di eventi e i minuti totali trascorsi

Questi passi possono essere ripetuti k volte per generare al più nk sessioni dove

- n è il numero di utenti;
- *al più* nk (e non esattamente) per via delle probabilità di inattività

Il codice che implementa questo algoritmo è il seguente:

```
def get_next_events(
    # data-ora di partenza
    start_datetime: datetime,
# output: una lista di eventi e i minuti trascorsi da start_datetime
) -> tuple[list[Event], int]:

    events: list[Event] = []
    minutes_past_midnight = _minutes_from_datetime(start_datetime)

    total_minutes_elapsed = 0
    # per ogni utente genera o meno una sessione
    #  a seconda delle probabilità
    for user in _users:

        # le probabilità di inattività dipendono dal time range
        # di questo utente
        is_their_timerange
            = any(minutes_past_midnight in timerange
                    for timerange in user.time_ranges)
        if is_their_timerange:
            chance_of_inactivity = 0.3
        else:
            chance_of_inactivity = 0.8

        # se il caso decide in favore di inattività, allora
        # non genera una sessione per questo utente
        random_perc = random.random()
        if random_perc < chance_of_inactivity:
            continue
        # altrimenti prosegue...

        # genera una sessione
        user_events, minutes_elapsed =
            _generate_user_session(
                start_datetime,
                user.name,
                user.is_super)

        # calcola i minuti trascorsi globali
        #  (rispetto a start_date)
        total_minutes_elapsed =
            max(total_minutes_elapsed, minutes_elapsed)
        # accoda gli eventi correnti alla lista globale
        events.extend(user_events)
    # fine ciclo.
    # Adesso ordina gli eventi per data e ora
    events.sort(key=lambda event: (event.date, event.time))

    return events, total_minutes_elapsed
```

5.5 Uso in Gotcha

Il generatore `get_next_events` viene usato in Gotcha per simulare l'interazione tra il componente *Monitor* e Manifesto;

Le modalità d'uso saranno approfondite nel capitolo relativo all'unità di controllo.

6 Ragionamento logico

Come primo approccio al rilevamento delle anomalie, ho scelto di utilizzare metodi e tecnologie proprie del *ragionamento logico*.

L'obiettivo di questa fase è di realizzare una *knowledge base (KB)* adeguata per il compito (descritta tra poco).

Segue una panoramica di alcuni aspetti importanti per lo sviluppo della KB.

6.1 Tecnologie utilizzate

In primo luogo occorre precisare che ho scelto di utilizzare due tecnologie:

- *Prolog* per definire la KB e sfruttare l'**inferenza logica**, essendo uno strumento pensato proprio per questo scopo;
- *Python* per il pre-processing del dataset, l'interazione con prolog e l'interpretazione dei risultati.

6.2 Knowledge Base

Una *knowledge base* (o *KB*) è un insieme di enunciati formali che descrivono le **caratteristiche** che sono **vere** nel mondo di interesse (es. *Manifesto*).

Nello specifico, questi enunciati, chiamati anche *assiomi*, si partizionano in **fatti** e **regole**:

- un *fatto* è un enunciato che è vero incondizionalmente (es. "`admin` è un superuser"); può essere anche chiamato **atomo**.
- una *regola* è un enunciato che è vero condizionalmente (es. "se `alice` è un superuser, allora può effettuare l'operazione `edit`")

Nel caso della KB di Manifesto, tra i **fatti** rientrano

```
:- dynamic unprocessed_event/4.
super_user(root).
super_user(admin).
```

Mentre tra le **regole** rientrano

```
night_time(TimeStr) :-
    sub_atom(TimeStr, 0, 2, _, HourAtom),
    atom_number(HourAtom, Hour),
    Hour < 6.

edit_night_time(Time) :-
    unprocessed_event(_, Time, _, edit),
    night_time(Time).

edit_from_unauthorized_user(User) :-
    unprocessed_event(_, _, User, edit),
    \+ super_user(User).

...
```

Il codice sorgente della KB di Manifesto è consultabile nel file localizzato in `/config/rules.pl`

6.3 Clausole definite

La definizione di "KB" è generica: non ci dice nulla su come gli assiomi devono essere specificati.

Poichè Prolog si basa sul linguaggio delle **clausole definite** (una forma specifica di logica proposizionale), in cui ogni assioma della KB segue una struttura specifica: quella della *clausola definita*.

Concretamente, ciò comporta che (in Prolog) le dichiarazioni degli assiomi presentano le seguenti caratteristiche:

1. Nel caso di **fatti**, le clausole hanno la forma

```
super_user(root).
```

oppure (come sarà approfondito poco più avanti)

```
:- dynamic unprocessed_event/4.
```

2. Nel caso di **regole**, invece, le clausole hanno la forma

```
edit_from_unauthorized_user(User) :-
    unprocessed_event(_, _, User, edit),
    \+ super_user(User).
```

dove

```
super_user(root)
unprocessed_event/4
edit_from_unauthorized_user(User)
```

sono dette *teste* della rispettive clausole (sono comunque atomi), mentre

```
:- unprocessed_event(_, _, User, edit),
\+ super_user(User).
```

è detto *corpo* (della relativa clausola), ed è formato da una congiunzione (operazione AND) di uno o più atomi.

Le clausole definite consentono di modellare un mondo senza ambiguità e in cui le relazioni causa-effetto sono certe, in quanto:

- la testa è formata da **una sola** affermazione elementare (es. "admin è un super user")
- la verità della testa dipende solo dalle verità degli atomi nel corpo; se il corpo non è specificato, la testa si assume vera a priori [1].

6.4 Assunzione di conoscenza completa (CWA) e Negation as Failure (NaF)

Osserviamo ancora le seguenti dichiarazioni di fatti in `config/rules.pl`

```
:- dynamic unprocessed_event/4.
:- dynamic blacklisted/1.

super_user(root).
super_user(admin).
```

notiamo che, da questi fatti, si può dedurre chi sono i superutenti, quali sono gli eventi non processati, etc.

Ma non sono esplicitamente riportati i relativi opposti (utenti non privilegiati, eventi processati, etc.).

[1] Questa affermazione è vera nel contesto delle convenzioni adottate da Prolog.

Nella *logica classica*, se non si riesce a provare che `super_user(alice)` è vera, non si può concludere automaticamente che è falsa, poichè si assume implicitamente che la **conoscenza contenuta nella KB non è completa** (*Open World Assumption, OWA*);

Tuttavia, nel caso di Prolog, viene implicitamente assunta la *completezza della conoscenza* (*Closed World Assumption, CWA*);

Questa assunzione comporta l'adozione della *Negation as Failure* (NaF): quando non si riesce a provare che `super_user(alice)=true`, allora si conclude che `super_user(alice)=false`.

Analogamente:

- Prolog non riesce a provare `blacklisted(eve)`? allora è `false`
- Prolog non riesce a provare `unprocessed_event(2025-01-01, 20.10, bob, edit)`? allora è `false`.

L'uso della NaF è evidente anche in regole come

```
edit_from_unauthorized_user(User) :-
    unprocessed_event(_, _, User, edit),
    \+ super_user(User).
```

Dove `\+ super_user(User)` significa "fornisci `true` se non riesci a provare `super_user(User)`", perchè chiaramente vogliamo rilevare un operazione illecita da parte di utenti non privilegiati.

> N.B. : è proprio la NaF a consentire, nel corpo delle clausole, l'uso dell'operatore di negazione `\+` (in letteratura ~) sugli atomi; Ricordiamo che, con la tipica assunzione di conoscenza incompleta (OWA), le clausole definite non ammettono l'uso di questo operatore.

6.5 Logica dei predicati

Gli assiomi della nostra KB Prolog non sono proposizioni proprie della logica proposizionale (classica), ma della *logica dei predicati*.

La logica classica si basa su una sintassi che non consente di ragionare in termini di individui e relazioni (come descritti nella sezione *concettualizzazione*) ad un qualsiasi livello di astrazione.

In altre parole, questa sintassi permette di specificare regole su **entità precise**, come le seguenti

> se *admin* è un super utente ed effettua un *login* notturno allora è una anomalia! se *Bob* non è un super utente ed esegue un `edit`, allora è un anomalia

Non sono ammesse, invece, espressioni del tipo

> "se *un qualsiasi* superutente effettua un `login` notturno allora è una anomalia!" "se *un qualsiasi* utente non è super utente ed effettua un `edit` notturno, allora è una anomalia!" "*tutti gli* `edit` notturni sono anomalie!"

La logica dei predicati è stata introdotta per sopperire a queste mancanze, introducendo il **ragionamento con variabili**;

Per esempio, una variabile come `User` rappresenta un insieme di individui concreti (quali `admin`, `bob`, `alice`, `eve`, `carol`, ...) per i quali è possibile specificare una regola comune come

```
edit_from_unauthorized_user(User) :-
    unprocessed_event(_, _, User, edit),
    \+ super_user(User).
```

la quale ci dice che una modifica non autorizzata (`edit_from_unauthorized_user` - il predicato) da parte di un utente `User` si verifica quando

- esiste un evento `edit` relativo a `User` non ancora processato (`unprocessed_event(_, _, User, edit)`), e
- `User` non è un super utente (`\+ super_user(User).`)

Con la logica proposizionale avremmo dovuto esplicitare una regola per ciascun utente, data, ora del giorno e tipo di evento.

> **N.B.** che individui *concreti* come `alice` (un utente), `edit` (una azione), `2024-02-01` (una data), ecc. sono detti anche *costanti*.

6.6 Fatti statici e dinamici

Riprendendo i fatti all'interno della KB, si notano due dichiarazioni diverse tra loro

```
:- dynamic unprocessed_event/4.

super_user(root).
```

La prima consente di dichiarare *dinamicamente* (cioè durante l'esecuzione) uno o più fatti di tipo `unprocessed_event` con 4 parametri (`Date, Time, User, Action`).

Questa scelta deriva dal fatto che il compito di Gotcha è di analizzare un **flusso di eventi non ancora processati** (i.e. *unprocessed event*).

Nel secondo caso, invece, si tratta di un *fatto statico*; questo riflette la decisione di non consentire l'aggiunta altri superuser durante l'esecuzione.

6.7 Binding

Prendiamo in esame la seguente regola:

```
edit_from_unauthorized_user(User) :-
    unprocessed_event(_, _, User, edit),
    \+ super_user(User).
```

Si potrebbe pensare che, trattandosi di una congiunzione di atomi, Prolog applichi la proprietà commutativa tipica dell'operatore `AND`, secondo cui l'ordine degli atomi è indifferente ai fini del risultato;

Tuttavia non è cosi che funziona, in quanto bisogna considerare il meccanismo di sostituzione delle variabili (*binding*).

Mostriamo il funzionamento con due esempi.

6.7.1 Regola "errata"

Partiamo dalla solita regola, in cui però l'ordine degli atomi nel corpo è invertito:

```
edit_from_unauthorized_user(User) :-
    \+ super_user(User),
    unprocessed_event(_, _, User, edit).
```

Supponiamo per semplicità di voler provare che `edit_from_unauthorized_user(bob)` è vera:

1. Per prima cosa, Prolog tenterà di provare il primo atomo del corpo: `\+ super_user(bob)`; quindi
2. effettua il **grounding**, ovvero genera tutte le possibili proposizioni che si ottengono sostituendo la variabile `User` in `super_user(User)` (i.e. le *istanze ground* della proposizione), ottenendo

```
super_user(root).
super_user(admin).
```

3. Poichè tra queste non esiste `super_user(bob)`, Prolog conclude (per la NaF) che `super_user(bob)` è **falso**
4. quindi `\+ super_user(bob)` è sempre **vero**
5. Ora Prolog prova il secondo atomo; questo sarà **vero** ogni qualvolta esistono, dopo il grounding, espressioni del tipo `unprocessed_event(_, _, bob, edit)`.

Per i passi (4) e (5), si ha che il risultato non rispetta la semantica intesa (i.e. edit da un utente non autorizzato); Pertanto questa regola è **errata**.

6.7.2 Regola corretta

Nel caso della regola originale, ovvero

```
edit_from_unauthorized_user(User) :-
    unprocessed_event(_, _, User, edit),
    \+ super_user(User).
```

e supponendo ancora di voler provare `edit_from_unauthorized_user(bob)`), avvengono le seguenti cose:

1. prima Prolog prova che esistono istanze di `unprocessed_event(_, _, bob, edit)`
2. se esistono, cerca di provare che `\+ super_user(Bob)`.
3. se entrambi gli atomi sono veri, allora si può concludere che, effettivamente, `bob` ha fatto un `edit` nonostante fosse un utente non privilegiato;

Quindi in questo caso la regola è coerente con la semantica desiderata.

6.8 Conseguenze logiche e query

Una volta realizzata una KB, uno dei compiti principali è stabilire quali sono le *conseguenze logiche* di tali KB, ovvero le verità non banali che seguono dagli assiomi.

Nel nostro caso, vogliamo conoscere le anomalie che sono conseguenze logiche dei fatti (gli eventi non processati, gli utenti online, ecc.).

A tale fine, la nostra KB contiene regole ausiliarie quali:

```
anomaly(edit_night_time, Time) :-
    edit_night_time(Time).

anomaly(edit_from_unauthorized_user, User) :-
    edit_from_unauthorized_user(User).

anomaly(edit_after_logout, User) :-
    edit_after_logout(User).

anomaly(blacklisted_user, User) :-
    blacklisted_user(User).

...
```

In queste regole, il predicato è lo stesso (i.e. `anomaly`), ma cambiano i parametri;

Ciò ci consente di interrogare la KB Prolog, tramite Python, con la semplice istruzione

```
prolog.query("anomaly(Type, Info)")
```

(vedi codice in `app/logic.py`).

Il risultato di questa interrogazione (chiamata *query*) sarà un **elenco**, anche vuoto, di *istanze di espressioni* di `anomaly(Type, Info)` che sono **vere**;

> le istanze di espressioni per `anomaly(Type, Info)` si ottengono sostituendo le variabili *Info* e *Type* con tutte le costanti ammesse.

Un esempio di risposta della query discussa è il seguente elenco:

```
anomaly(edit_after_logout, alice).
anomaly(blacklisted_user, eve).
anomaly(edit_from_unauthorized_user, bob).
```

6.9 Uso della KB in Gotcha

il ruolo della KB è quello di realizzare l'unità di ragionamento logico di Gotcha accennata nel sommario.

6.9.1 Clausole definite della KB di Gotcha

Segue l'elenco completo delle clausole definite della KB utilizzata per Gotcha e Manifesto.

```
% fatti
% ==============
:- dynamic unprocessed_event/4.
:- dynamic blacklisted/1.
:- dynamic online_user/1.
super_user(root).
super_user(admin).

% regole di supporto
% ==================
night_time(MinutesPastMidnight) :-
    MinutesPastMidnight >= 0,
    MinutesPastMidnight < 360.

% regole per modellare le anomalie
% ================================

double_login(User) :-
    unprocessed_event(_, _, User, login),
    online_user(User).

double_logout(User) :-
    unprocessed_event(_, _, User, logout),
    \+ online_user(User).

blacklisted_user(User) :-
    unprocessed_event(_, _, User, _),
    blacklisted(User).

edit_night_time(Time) :-
    unprocessed_event(_, Time, _, edit),
    night_time(Time).

edit_from_unauthorized_user(User) :-
    unprocessed_event(_, _, User, edit),
    \+ super_user(User).

edit_after_logout(User) :-
    unprocessed_event(_,_, User, edit),
    \+ online_user(User).

superuser_login_at_night_time(User, Time) :-
    unprocessed_event(_, Time, User, login),
    super_user(User),
    night_time(Time).

% regole ausiliare per semplificare la query
% ==========================================
```

```
anomaly(edit_night_time, Time) :-
    edit_night_time(Time).

anomaly(edit_from_unauthorized_user, User) :-
    edit_from_unauthorized_user(User).

anomaly(edit_after_logout, User) :-
    edit_after_logout(User).

anomaly(superuser_login_at_night_time, (User, Time)) :-
    superuser_login_at_night_time(User, Time).

anomaly(blacklisted_user, User) :-
    blacklisted_user(User).

anomaly(double_login, User) :-
    double_login(User).

anomaly(double_logout, User) :-
    double_logout(User).
```

6.9.2 Soppressione del SessionID

Si osserva che `unprocessed_event` ha 4 parametri anzichè 5, in quanto si è deciso di sopprimere il parametro *SessionID* (discusso in fase di concettualizzazione) in favore del fatto dinamico `online_user` per semplificare il ragionamento;

Usare il SessionID anzichè `online_user` per le regole coinvolte avrebbe comportato una loro implementazione simile alla seguente:

```
double_logout(Session, User) :-
    unprocessed_event(Session, _, Time1, User, logout),
    unprocessed_event(Session, _, Time2, User, logout),
    Time1 \= Time2.

edit_after_logout(Session, User) :-
    unprocessed_event(Session, _, TimeLogout, User, logout),
    unprocessed_event(Session, _, TimeEdit, User, edit),
    time_is_before_time(TimeLogout, TimeEdit).
```

Ho provato a percorrere questa strada, ma ho riscontrato alcune difficoltà che non sono riuscito a risolvere in tempo, pertanto ho ripiegato sulla soluzione iniziale.

6.9.3 Effettuare query

Nel modulo Python `app/logic.py`, è presente una funzione (`check_event_using_inference`) che svolge i seguenti passi, **un evento alla volta**:

1. asserisce nella KB il fatto `unprocessed_event(...)`, cioè che l'evento corrente non è stato ancora processato (ritorna su sezione *fatto dinamico*);
2. nel caso l'evento è di tipo `login` o `logout`, aggiorna lo stato del relativo utente, dichiarando o rimuovendo il fatto `online_user` rispettivamente.
3. chiede alla KB se questo evento presenta una o più anomalie, mediante

```
prolog.query("anomaly(Type, Info)")
```

3. rimuovi il fatto `unprocessed_event(...)`;
4. fornisci in output la lista di anomalie trovate;

La lista di anomalie sarà poi associata all'evento che l'ha causata e sottoposta al giudizio dell'utente di Gotcha, come indicato dal diagramma nel Sommario; Questo aspetto sarà approfondito nel capitolo "unità di controllo".

6.10 Considerazioni sulla complessità

La KB che ho realizzato è indubbiamente semplice, ma non per questo è necessariamente banale:

- i fatti dinamici agiscono come *sensori*:
  - `unprocessed_event` segnala la presenza di un evento da analizzare;
  - `online_user` e `blacklisted` permettono di tracciare gli utenti online e banditi rispettivamente;
- le regole si basano sui sensori per modellare le anomalie più rilevanti, ideate in fase di concettualizzazione;
- l'interrogazione della KB è facilitata dall'uso delle clausole ausiliarie (di testa `anomaly`), poichè consentono di utilizzare una singola istruzione (`anomaly(Type,Info)` per testare tutti i tipi di anomalie, eventualmente anche quelle aggiunte in futuro.

In altri termini, questo sistema, nonostante le sue dimensioni limitate, esibisce alcune delle proprietà fondamentali per la progettazione di KB più complesse.

7 Apprendimento non supervisionato

Nella seconda fase di sviluppo, ho deciso di realizzare un componente in grado di analizzare gli eventi passati etichettati come normali ed apprendere, nello specifico, gli orari di attività (accesso e modifica) tipici per ciascun utente;

La profilazione risultante è poi usata per decidere se l'ora di un nuovo evento ricade in un momento della giornata tipico per l'utente associato, oppure se si discosta significativamente (costituendo quindi una anomalia).

Come nel capitolo precedente, discutiamo passo passo gli aspetti importanti.

7.1 Tecnologie utilizzate

Le tecnologie utilizzate per questo sviluppo sono Python in combinazione con le librerie, tra le principali:

- *sklearn* (per il k-means),
- *kneed* (per l'ottimizzazione del k-means)
- *numpy* (per il preprocessing del dataset)
- *matplotlib* (per i grafici)

7.2 Apprendimento

L'apprendimento è la qualità di un sistema intelligente di migliorarsi passo dopo passo, sfruttando i dati ottenuti dall'esperienza passata.

Tra i principali, esistono due tipi di apprendimento: *supervisionato* e *non supervisionato*.

Nel caso di Gotcha, ho implementato un apprendimento di tipo **non supervisionato**; ma per comprendere meglio gli aspetti tecnici è opportuno parlare prima di apprendimento supervisionato.

7.3 Apprendimento supervisionato (e non supervisionato)

Nell'apprendimento supervisionato, il compito tipico consiste nell'analisi di un insieme di dati già classificati, detti *esempi di addestramento*, per costruire una funzione (detto *modello predittivo*) che è in grado di assegnare la classe appropriata a nuovi esempi.

A titolo esemplificativo, supponiamo di avere i seguenti dati sull'utente `bob`

Orario di attività (di `bob`)	Anomalia (Si/No)
10.00	No
13.00	No
14.00	Si
16.00	No
23.00	Si
...	...
14.30	Si
17.40	No

dove

- *Orario di attività* e *Anomalia* si dicono *feature* (i.e. caratteristiche)
- Ogni riga, esclusa l'intestazione, è un *esempio*;
- Ogni esempio si può vedere come una coppia (t, a) dove t, a sono i valori associati alle due feature *Orario di attività* e *Anomalia*;

Svolgere un compito di apprendimento supervisionato su questi dati significa capire quando un orario è anomalo e quando non lo è, in modo da poter classificare correttamente nuovi esempi.

Nel caso di Manifesto, però, gli esempi non sono espressi in tale forma, ma nella seguente:

Orario di attività (di `bob`)
11.15
12.30
13.00
16.10
17.20
...
9.30

cioè i dati relativi agli orari dell'esperienza passata (raccolti in `data/past-event.csv`) **non sono etichettati** con anomalo o non anomalo, ma descrivono semplicemente il comportamento tipico di `bob` (o di qualsiasi altro utente).

Sarebbe in teoria possibile svolgere un compito di apprendimento supervisionato su questi dati utilizzando tecniche molto semplici (come la media empirica), ma la loro efficacia è discutibile.

Un approccio più interessante, invece, si basa su due idee

- gli orari vicini tra loro formano un *gruppo*, ovvero la giornata tipica dell'utente può essere descritta in termini di uno o più *blocchi*;
- Quando un nuovo orario non rientra **in modo netto** in nessuno di questi blocchi, allora sarà classificato come anomalo.

L'idea di raggruppare i dati per formare blocchi (o *cluster*) è alla base del *clustering* che uno dei compiti principali di apprendimento *non* supervisionato.

7.4 Clustering

Il clustering è, quindi, il processo attraverso il quale gli esempi di addestramento sono analizzati con lo scopo di produrre un loro partizionamento (un insieme di *cluster*) basato su uno specifico criterio di somiglianza.

Per esempio, mediante il clustering sugli eventi passati, si potrebbe scoprire che l'utente `alice` è solito effettuare l'accesso in 3 distinti momenti della giornata (es. la mattina presto, a pranzo e a cena), mentre l'utente `bob` in 2 momenti (es. durante l'intera mattinata e l'intero pomeriggio).

L'algoritmo di clustering che ho scelto di utilizzare è il k-means, ma non prima di aver trasformato gli esempi in una forma più adeguata.

7.5 Preprocessing degli esempi

Nel capitolo "Generazione del dataset" avevo anticipato che per modellare l'ora del giorno ho utilizzato i "minuti dopo la mezzanotte", che banalmente è un intero compreso tra 0 e 1440;

Questa scelta consente di trattare le giornate in modo lineare ed agevole.

Tuttavia ciò non basta, perchè l'ora è una grandezza ciclica e deve essere modellata come tale per poter considerare vicine ore come le 23.00 e le 1.00.

Una soluzione semplice a questo problema è stata di codificare le ore come una coordinata su una circonferenza unitaria, utilizzando le funzioni trigonometriche sin e cos; Questo ha comportato che la singola feature "Orario" venisse rimpiazzata da due feature (cioè *seno* e *coseno*).

L'algoritmo di conversione da minuti (dopo mezzanotte) a coordinate sulla circonferenza è il seguente:

```
def create_time_column(minutes_list: list[int]):
    minutes_norm = np.array(minutes_list) / 1440
    minutes_sin = np.sin(2 * np.pi * minutes_norm)
    minutes_cos = np.cos(2 * np.pi * minutes_norm)
    return np.column_stack([minutes_sin, minutes_cos])
```

> N.B. `np` è un alias per la libreria *numpy*, accennata nella sezione "Tecnologie".

7.6 k-means

Il k-means è, in particolare, un algoritmo di *hard clustering* (contrapposto a *soft clustering*) in quanto assegna ogni esempio ad **un solo** cluster.

Si basa sulle seguenti idee:

- gli esempi sono punti collocati in uno spazio con tante dimensioni quante sono le feature;
- gli esempi *sufficientemente* vicini (in termini di distanza euclidea) formano un cluster;
- ogni cluster ha un *centroide*: il punto medio (aritmeticamente parlando) di ogni esempio nel cluster.
- il cluster di appartenenza di un nuovo esempio è quello associato al centroide più vicino.

  > Nel nostro caso, lo spazio è bidimensionale: seno e coseno (come discusso nella sezione precedente).

L'obiettivo del k-means è di creare k cluster per cui l'*inerzia* è *minimizzata**;

> L'*inerzia* è la somma delle distanze quadrate di ogni esempio rispetto al centroide del cluster a cui è stato assegnato.

Si dice *modello* di k-means il sistema di cluster risultante dalla sua esecuzione.

Queste nozioni sono importanti per comprendere le scelte implementative fatte per applicare k-means in Gotcha che saranno in seguito discusse.

7.6.1 Scelta di k (mediante regola del gomito)

L'efficacia del clustering dipende dal parametro k (il numero di cluster) che deve essere un buon compromesso tra:

- fedeltà rispetto agli esempi di addestramento (cioè l'*adattamento*), con l'obiettivo di minimizzare l'errore quadratico totale (cioè l'inerzia);
- capacità di generalizzare, in modo da collocare nuovi esempi nei cluster più appropriati;
- semplicità del modello (in concordanza con il principio del Rasoio di Occam).

Nel nostro caso vogliamo, **per ogni utente**, individuare un numero sufficiente di cluster (k) per descrivere in che momenti egli è tipicamente attivo in Manifesto;

Quindi, se gli utenti sono 5, dobbiamo addestrare 5 modelli k-means, ciascuno adattato alle peculiarità del relativo utente.

A tale fine, ho scelto di utilizzare la *regola del gomito* che stabilisce che il valore migliore per k è quello oltre il quale la riduzione dell'inerzia è trascurabile.

N.B. l'inerzia descresce all'aumentare di k

Un esempio di applicazione della regola del gomito (su circa 10 esempi) è sintetizzato dalla seguente figura:

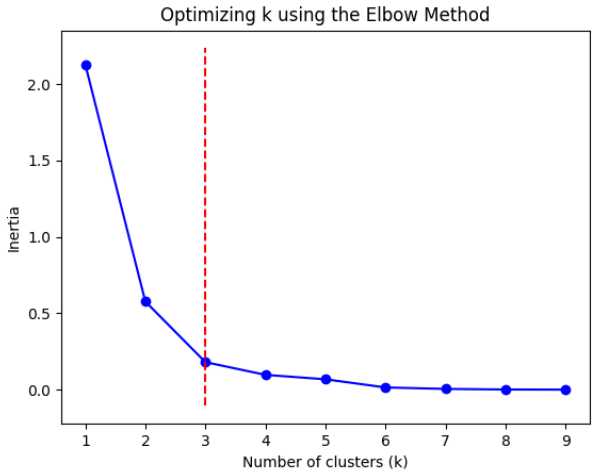

Figure 2: Esempio regola del gomito.

osserviamo che l'inerzia decresce molto velocemente nei primi due passi, per rallentare in $k = 3$ e stabilizzarsi da $k = 4$ in poi; $k = 3$ è la scelta ottimale poichè la riduzione di inerzia dal $k = 4$ non è abbastanza da giustificare una maggiore complessità del modello.

Segue la mia implementazione in Python della regola del gomito (consultabile in `app/learning.py`):

```
inertia_points = []

# esegui il k-means per ogni valore di k tra 1 e 9
for i in range(1, 10):
    # inizializza il modello
    model = KMeans(
        n_clusters=i,
        n_init=5,
        init='random',
    )
    # addestra il modello
    model.fit(data)
    # aggiungi l'inerzia di questo modello all'asse y
    inertia_points.append(model.inertia_)

# inizializza il piano cartesiano
kneeloc = KneeLocator(
    x=range_k,
    y=inertia_points,
    curve='convex',          # convessa, perchè cerchiamo il gomito
```

```
    direction='decreasing', # l'inerzia diminuisce all'aumentare di k
)
# determina la coordinata x in cui si trova il gomito
return kneloc.elbow
```

7.6.2 Minimizzare l'inerzia in assoluto

Il k-means non garantisce che la soluzione trovata sia quella perfetta, cioè quella che minimizza l'inerzia in assoluto (i.e. minimo globale);

Per mitigare questo problema, è possibile:

1. eseguire il k-means n volte inizializzando i centroidi in modo diverso (fare cioè un *random restart*), e poi...
2. selezionare la soluzione con l'inerzia minima.

Nonostante non sia comunque garantito un minimo globale, aver tentato inizializzazioni diverse consente di aumentare le probabilità di ottenere una soluzione che vi si avvicini molto.

Nella codebase, questa scelta si riconosce dalla seguente istruzione (in `app/learning.py`):

```
model = KMeans(
    # optimal_k è calcolato mediante la regola del gomito
    n_clusters=optimal_k,
    # esegui random restart 5 volte;
    # poi seleziona il cluster-set con la inertia minore
    n_init=5,
    init='random',
)
```

7.6.3 Esempio di clustering

La seguente figura mostra un possibile risultato del clustering degli orari di attività dell'utente `eve` (43 esempi):

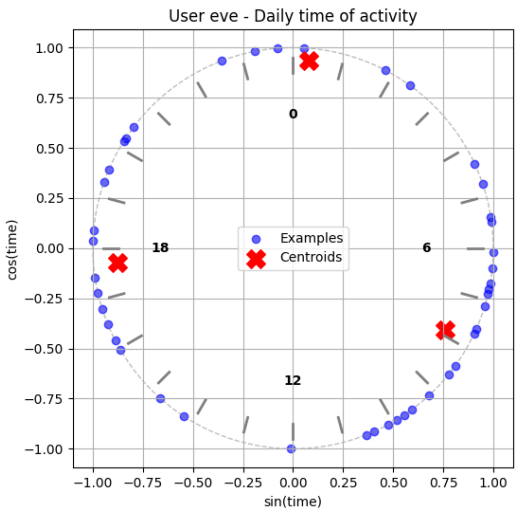

Figure 3: Esempio di clustering

Dalla figura si evince che `eve` è tipicamente attivo, approssimativamente, nelle fascie orarie

```
4.00 - 11.00
14.00 - 21.00
```

`22.00 - 1.30`

con un punto anomalo (tecnicamente chiamato *outlier*) intorno alle 12.00.

7.6.4 Limite: sensibilità agli outlier

Un altro problema di cui soffre il k-means è la sensibilità dei centroidi a valori estremi.

Questo problema viene mitigato in parte con l'aumentare del numero di esempi di addestramento:

Per esempio, quando ci sono 100 esempi concentrati nella stessa zona dello spazio e solo uno in una zona lontana, il centroide tende a sbilanciarsi verso la prima zona.

Questo si puo osservare nella figura relativa a `eve` fornita nella sezione precedente (basata su 43 esempi): il centroide posizionato in prossimità delle ore 6.00 è influenzato maggiormente dagli esempi collocati tra le 4.00 e le 11.00, rispetto al singolo esempio delle ore 12.00 circa (ciononostante, quest'ultimo ha avuto una influenza non banale).

7.7 Costruzione del classificatore

Ora che il k-means ha prodotto un profilo per ciascun utente, il prossimo passo è costruire un classificatore binario (per ognuno degli utenti) che decida se un nuovo esempio è una anomalia o meno.

L'idea è semplice: se l'esempio è *abbastanza* lontano dal centroide più vicino (cioè quello del cluster a cui appartiene) allora si considera un anomalia.

Per implementare questa idea ho scelto un approccio altrettanto semplice:

1. trasforma il centroide in minuti (dopo mezzanotte),
2. calcola la differenza tra l'esempio e il centroide (sempre in minuti),
3. se la differenza supera un certo numero di minuti (e.g. 120m), classifica l'esempio come anomalo.

Segue lo snippet di codice che implementa questo algoritmo (`app/learning.py`):

```
# recupera il modello relativo all'utente che compare nell'evento
user_model = user_predictors.get(event.user)
# estrae la feature orario (time) dall'evento e la converte in un esempio contenente seno e coseno
test_example = create_time_column([event.time])

# determina il centroide più vicino all'esempio
distances = user_model.transform(test_example)[0]
closest_cluster_id = np.argmin(distances)
closest_centroid = user_model.cluster_centers_[closest_cluster_id]

# estrai le feature seno e coseno dal centroide e convertile in minuti
sin_val, cos_val = closest_centroid[0], closest_centroid[1]
centroid_in_minutes = sincos_to_minutes(sin_val, cos_val)
# calcola la differenza, in minuti, tra il centroide e l'ora
diff = abs(centroid_in_minutes - event.time)
# correggi i casi estremi (es. la differenza tra 0m (mezzanotte) e 1380m (le 23.00) dovrebbe esser
diff = min(diff, 1440 - diff)

# se la differenza non eccede 120m, considera l'evento non anomalo.
if diff <= 120:
    return []
# altrimenti, segnala anomalia
return [f"time of activity is {diff} minutes off the"
        f" closest average time of activity for user '{event.user}'"
        f" ({minutes_to_hhmm(centroid_in_minutes)})"]
```

7.8 Uso in Gotcha

Ricordiamo che l'apprendimento discusso in questo capitolo è il mezzo attraverso cui realizzare l'**unità di apprendimento e classificazione** per Gotcha (vedi Sommario); questa si occupa di due compiti:

- *apprendimento* : costruire, per ciascun utente, un classificatore di anomalie basato sugli eventi passati, secondo l'approccio visto finora;
- *classificazione* : utilizzare i classificatori addestrati per classificare nuovi eventi come anomali o non anomali.

Questi due compiti non sono svolti sequenzialmente e contemporaneamente, ma in momenti separati, come sarà spiegato nel capitolo successivo ("unità di controllo");

Pertanto sarà lecito riferirsi ad apprendimento e classificazione come **unità separate**, d'ora in poi.

7.9 Considerazioni sulla valutazione

Nel contesto dell'apprendimento **supervisionato** (senza il "non"), la valutazione dei modelli predittivi è parte integrante del processo di progettazione e sviluppo.

In generale, la finalità della valutazione è quella di trovare una combinazione dei parametri per l'algoritmo scelto tale da soddisfare due necessità:

- adattarsi bene agli esempi di addestramento, in modo da comprendere le relazioni rilevanti del dominio di interesse;
- predire per nuovi esempi, la classe di appartenenza.

Soddisfare pienamente questi requisiti è impossibile, in quanto incompatibili:

- se il modello si adatta perfettamente agli esempi di addestramento, perde la capacità di generalizzare su esempi che non ha mai visto, poichè, nel processo decisionale, tiene conto di dettagli insignificanti (fenomeno di *overfitting*);
- se il modello non si adatta sufficientemente ai dati, trascura informazioni che servono ai fini di una classificazione accurata. (fenomeno di *underfitting*)

Inoltre, un altro criterio di valutazione importante è la *complessità* del modello: a parità di prestazioni è spesso preferibile un modello (o algoritmo) che utilizza meno risorse computazionali.

Nel mondo dell'apprendimento **non supervisionato** (quindi quello di nostro interesse), ci sono delle analogie, ma gli strumenti di valutazione utilizzati sono diversi;

Infatti, riprendendo le osservazioni fatte precedentemente, nell'apprendimento supervisionato gli esempi di addestramento **sono etichettati** (cioè già classificati) e questo consente di applicare tecniche come la *cross-validation*;

Nell'apprendimento non supervisionato gli esempi **non sono etichettati** e quindi, nel caso del clustering, non esiste una classe di riferimento con cui confrontare le classi calcolate;

ciononostante, esistono altri strumenti per misurare le prestazioni del modello: nel caso del k-means un possibile strumento è l'inerzia, come illustrato nella sezione in cui discuto della scelta di k.

8 Unità di controllo

L'unità di controllo orchestra i componenti visti finora e consente all'utente di interagire con Gotcha.

Il punto di partenza è il menù principale, che si presenta nel seguente modo:

```
=== Gotcha! (Main Menu) ===
Target system: Manifesto

Choose next operation by typing (what's inside) :
  (s)    Start monitoring events
```

```
(t)    Train an anomaly predictor for each user based on past events (reads from 'data/past-ever
(tdel) Delete predictors
(gen)  Test the generator of events (writes to 'data/test_log.csv')
(rst)  Delete past events data (deletes file 'data/past-events.csv')
(q)    quit
```

Tra le varie opzioni, ci interessano particolarmente monitoraggio (`s`) e addestramento (`t`).

8.1 Monitoraggio

Quando l'utente sceglie di monitorare Manifesto, Gotcha esegue i seguenti compiti:

1. genera degli eventi usando `get_next_events`; (cfr. capitolo *Generazione del Dataset*)
2. fornisce questi eventi all'unità di ragionamento logico;
3. fornisce questi eventi anche all'unità di classificazione;
4. gli eventi non anomali sono aggiunti al dataset `data/past-events.csv`
5. gli eventi anomali vengono aggregati in una unica lista; se non ci sono anomalie, **torna al passo 1**
6. consenti all'utente di decidere se le anomalie trovate sono veri positivi o falsi positivi.
7. i falsi positivi sono aggiunti al dataset `data/past-events.csv`
8. se l'utente di Gotcha intende effettuare una nuova iterazione, **torna al passo 1**; altrimenti torna al menù principale.

8.2 Addestramento

Quando l'utente sceglie dal menù la modalità di addestramento, Gotcha utilizza il dataset `data/past-events.csv` (prodotto durante il monitoraggio) per richiedere all'unità di apprendimento, ricordiamolo, di produrre un classificatore di anomalie per ciascun utente che compare in tale dataset.

Se gli esempi di addestramento sono troppo pochi, Gotcha segnala all'utente l'impossibilità di proseguire e lo invita ad eseguire ulteriori iterazioni di monitoraggio prima di ritentare l'addestramento.

9 Sviluppi futuri

Le possibilità di estensione sono innumerevoli.

Procedendo **verticalmente**, si potrebbe pensare di aggiungere altre regole di inferenza per Manifesto, oppure di costruire un predittore basato su altre feature;

Per esempio, si potrebbero calcolare il numero di sessioni giornaliere di ciascun utente e costruire un semplice modello di regressione costante (come la media empirica con pseudo-conteggi) per tale grandezza.

Procedendo **orizzontalmente**, invece, potremmo dotare Gotcha della capacità di monitorare e apprendere da altri sistemi oltre a Manifesto.

10 Trivia: scelta del nome del progetto

Nel pensare ad un nome, si palesò immediatamente nella mia testa l'interiezione "*Gotcha!*" (in italiano traducibile come "*Beccato!*"); Ho trovato questa parola molto calzante per il ruolo di questo sistema per il suo significato, oltre ad essere piuttosto simpatica e divertente da sentire e pronunciare.

Poichè questo nome risulterebbe informale e causa problemi in certi contesti (come per la denominazione del repository) per via del punto esclamativo ho deciso di utilizzare l'alias *gotcha-system*.