Indice

1 Introduzione

1.1 Creatori

Massimo Tubito 717440 m.tubito5@studenti.uniba.it

Ferrulli Nunzio 719508 n.ferrulli5@studenti.uniba.it

Link di GitHub per scaricare il progetto: https://github.com/FerrulliNunzio/Progetto_Icon_TF.git

1.2 Strumenti software

1.2.1 IDE

E' stato utilizzato l'IDE **PyCharm** che è un linguaggio di programmazione **Python** sviluppato da **JetBrains** ed è un **ambiente di sviluppo integrato** (IDE), ovvero un software che, in fase di programmazione, supporta i programmatori nello sviluppo e debugging del codice sorgente di un programma.

1.2.2 Linguaggio utilizzato

Il software è sviluppato interamente in python che è un linguaggio di programmazione di "alto livello", orientato a oggetti, adatto, tra gli altri usi, a sviluppare applicazioni distribuite, scripting, computazione numerica e system testing. **La versione di python utilizzata è la 3.8**

1.2.3 Librerie utilizzate

Le librerie utilizzate nel progetto sono **Sklearn** e **PySwip** (per poter utilizzare la libreria <pyswip> è necessario aver installato [swi-prolog](https://www.swi-prolog.org/Download.html))

1.3 Il Progetto

Il software simula una situazione in cui si verifica un incidente ed è necessario l'intervento da parte di una **stazione dei vigili del fuoco.** Verrà generato un evento casuale il quale verrà classificato e gli verrà associato un valore che rappresenta il grado di emergenza. Successivamente si interrogherà una base di conoscenza (knowledge base) per determinare quale stazione dei pompieri ha le risorse per intervenire. Si determinerà, infine, per ogni stazione dei vigili del fuoco, che soddisfa i requisiti per intervenire, il percorso per arrivare sul luogo dell'evento, tenendo conto del tempo e distanza al luogo dell'evento.

2 Algoritmi di classificazione

Per capire di che tipo di intervento bisognerà eseguire bisognerà capire la gravità dell'incidente infatti utilizziamo un classificatore che prenderà in input un evento generato casualmente secondo 5 features, verrà classificato e darà in output il grado di emergenza associato all'evento.

		Boolean			
Input features	Numero dei feriti	Esplosioni	Incendi	Calamità naturali	Incidente stradale
Dominio	{0,1,2,3}	{0,1,2}	{0,1,2}	{0,1,2,3}	{0,1,2,3,4}

Numero dei feriti : ciascun valore del dominio indica un range del numero dei feriti

- 0 : 0
- 1 : 1->3
- 2 : 4->7
- 3 : 8+

Nel progetto è utilizzata la libreria Sklearn per la costruzione del Decision Tree e classificazione degli input

2.1 Alberi decisionali

Gli alberi decisionali (DT) sono un metodo di apprendimento supervisionato non parametrico utilizzato per la classificazione. L'obiettivo è creare un modello che preveda il valore di una variabile target apprendendo semplici regole decisionali dedotte dalle caratteristiche dei dati.

Vantaggi:

Possono essere visualizzati e interpretati facilmente. Richiede poca preparazione dei dati e sono in grado di gestire dati sia numerici che categoriali e problemi con più output. E'

possibile validare un modello utilizzando test statici in questo modo si tiene conto dell'affidabilità del modello.

Svantaggi:

Si possono creare alberi troppo complessi che non generalizzano bene i dati (overfitting). Per evitare questo problema sono necessari meccanismi come la potatura. Gli alberi decisionali possono essere instabili perché piccole variazioni nei dati potrebbero comportare la generazione di un albero completamente diverso. Le previsioni degli alberi decisionali non sono né lisce né continue, ma approssimazioni costanti a tratti. Il problema dell'apprendimento di un albero decisionale ottimo è noto per essere NP-completo e gli algoritmi pratici di apprendimento dell'albero decisionale si basano su algoritmi euristici.

2.2 Classificazione

La classe **DecisionTreeClassifier** della libreria **sklearn** è in grado di eseguire la classificazione multiclasse su un set di dati. Il **DecisionTreeClassifier** prende come input due array: un array X, sparso o denso, di forma che contiene i campioni di addestramento e un array Y di valori interi, shape , che contiene le etichette di classe per i campioni di addestramento:(n_samples, n_features)(n_samples,)

```
def set_training(balance_data, test):
    X = balance_data.values[:, 0:6]
    Y = balance_data.values[:, 6]

    X_train, X_test, y_train, y_test = train_test_split(
        X, Y, test_size=0.000001, random_state=0)

    X_test = test

    return X, Y, X_train, X_test, y_train, y_test

# addestro il classificatore usando come parametro di split
l'entropia
def train_using_entropy(X_train, y_train):
    clf_entropy = DecisionTreeClassifier(
        criterion="entropy", random_state=100,
        max_depth=3, min_samples_leaf=5)
    clf_entropy.fit(X_train, y_train)
    return clf_entropy
```

La funzione DecisionTreeClassifier può prendere in input diversi parametri tra cui quelli utilizzati da noi sono:

- criterion: può assumere i deguenti valori {"gini", "entropia", "log_loss"}, di default assume il valore "gini"
- random_state: Controlla la casualità dello stimatore. È di tipo intero e non assume valori di default
- max_depth: La profondità massima dell'albero. È di tipo intero e non assume nessun valore come default
- min_samples_leaf: Il numero minimo di campioni richiesto per essere in un nodo foglia. È di tipo intero o float e di default assume valore 1

Dopo essere stato adattato, il modello può quindi essere utilizzato per prevedere la classe dei campioni.

```
def prediction(X_test, clf_object):
    y_pred = clf_object.predict(X_test)
    return y_pred
```

Nel caso in cui ci siano più classi con la stessa e la più alta probabilità, il classificatore predicherà la classe con l'indice più basso tra quelle classi. Il **DecisionTreeClassifier** è in grado di effettuare sia la classificazione binaria (dove le etichette sono [-1, 1]) sia la classificazione multiclasse (dove le etichette sono [0, …, K-1]).

2.3 Complessità

In generale, il costo del tempo di esecuzione per costruire un albero binario bilanciato è $O(n_{samples}n_{features}log(n_{samples}))$ e tempo di interrogazione $O(log(n_{samples}))$. Sebbene l'algoritmo di costruzione dell'albero tenti di generare alberi bilanciati, non saranno sempre bilanciati. Supponendo che i sottoalberi rimangano approssimativamente bilanciati, il costo di ciascun nodo consiste nella ricerca $O(n_{features})$ per trovare la caratteristica che offre la maggiore riduzione del criterio dell'impurità. Questo ha un costo di $O(n_{features}n_{samples}log(n_{samples}))$ ad ogni nodo, portando ad un costo totale su tutti gli alberi (sommando il costo ad ogni nodo) di $O(n_{features}n^2_{samples}log(n_{samples}))$.

1. Knowledge base

La knowledge base è stata utilizzata per rappresentare le varie caserme e le varie truppe disponibili per ognuna. Ogni caserma dispone di tre tipi di truppe

- Agenti: numeri di agenti del fuoco disponibili in una caserma
- Veicoli: numero dei veicoli disponibili in una caserma (si intende con veicoli l'autopompa e il fuoristrada)
- Veicoli speciali: numero dei veicoli speciali disponibili in una caserma (si intende con veicoli speciali i veicoli di supporto come l'autobotte che viene utilizzata solo in casi speciali)

Una volta classificato l'incidente, in base al grado di pericolosità verrà generato un intervento che andrà a definire quali sono le truppe e i mezzi necessari per eseguire l'intervento. Questo servirà per interrogare la base di conoscenza per determinare quali sono le caserme in grado di intervenire secondo le necessità di ciascun grado di emergenza descritte di seguito.

grado 1

-numero agenti richiesti 2

-numero veicoli 2

-numero veicoli speciali 0

-tempo 40

grado 2

-numero agenti richiesti 7

-numero veicoli 5

-numero veicoli speciali 0

-tempo 30

grado 3

-numero agenti richiesti 10

-numero veicoli 7

-numero veicoli speciali 1

-tempo 25

grado 4

-numero agenti richiesti 15

-numero veicoli 10

-numero veicoli speciali 5

-tempo 20

per l'implementazione della knowledge base abbiamo utilizzato la libreria prolog.

3.1 Sintassi

Nel Prolog, la logica del programma è espressa sotto forma di relazioni e le attività di calcolo vengono attivate da un'interrogazione relativa a tali relazioni.

3.1.1 Tipi di dati

L'elemento generico del Prolog si chiama *termine*. I termini possono essere *costanti* (*atomi* o *numeri*), *variabili* o *termini composti*.

- Un **atomo** è un nome generico senza significato intrinseco,
- Un **numero** può essere intero o decimale.
- Una **variabile** è indicata per mezzo di una stringa di lettere, numeri e underscore (_) che comincia con una maiuscola o un trattino basso.
- Un **termine composto** è formato da un atomo detto "funtore" e da uno o più argomenti - anch'essi termini - scritti tra parentesi e separati da virgole, p.es.

Casi speciali di termini composti:

- Una **lista** è una collezione ordinata di termini, separati da virgole; viene indicata per mezzo di parentesi quadre; è ammessa la lista vuota `[]`.
- Una **stringa** è una sequenza di caratteri delimitata da doppi apici ("),

3.1.2 Regole e fatti

Una **regola** ha la forma:

```
Testa :- Corpo.
```

che si legge: "Testa è vera se Corpo è vero." (Si noti che la regola termina con un punto.)

Un singolo termine (anche composto), senza il segno `:-`, viene chiamato **fatto**. I fatti equivalgono a regole senza corpo, che sono considerate automaticamente vere.

3.2 Database

SWI-Prolog offre diversi modi per memorizzare i dati in una memoria accessibile a livello globale, cioè al di fuori degli *stack* Prolog . I dati memorizzati in questo modo, in particolare, non cambiano durante il *backtracking*. Di seguito sono elencate le principali opzioni per la memorizzazione dei dati:

Utilizzo di predicati dinamici

I predicati dinamici sono predicati per i quali l'elenco di clausole viene modificato in fase di esecuzione utilizzando asserta/1 , assertz/1. I predicati modificati in questo modo devono essere dichiarati utilizzando la direttiva dynamic/1. Le prestazioni dipendono dall'implementazione di Prolog. In SWI-Prolog, l'esecuzione di query sui predicati dinamici ha le stesse prestazioni di quelli statici. I predicati di manipolazione sono veloci. È possibile eseguire il wrapping dei predicati dinamici utilizzando la libreria library(persistency) per mantenere un backup dei dati su disco. I predicati dinamici sono disponibili in due versioni, condivisi tra i thread e locali a ciascun thread. Quest'ultima versione viene creata utilizzando la direttiva thread_local/1 .

3.2.1 asserts

Affermare una clausola (fatto o regola) nel database. Il predicato assertz/1 asserisce la clausola come ultima clausola. Il deprecato assert/1 equivale ad assertz/1 . Se lo spazio del programma per il modulo di destinazione è limitato (vedi set_module/1), asserta/1 può sollevare resource_error(program_space)un'eccezione.

Creazione del database con assertz nel programma

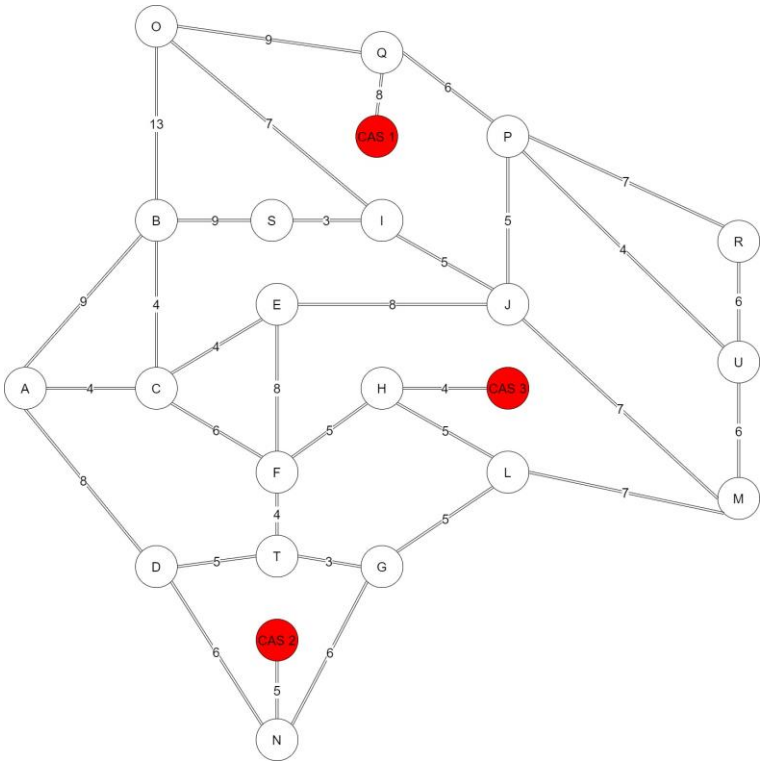
```
def createKB():
    kb = Prolog()
    for i in range(3):
        kb.assertz("caserma(caserma_" + str(i+1) + ")")

    kb.assertz("agenti(caserma_1, 5)")
    kb.assertz("agenti(caserma_2, 10)")
    kb.assertz("agenti(caserma_3, 20)")

    kb.assertz("veicoli(caserma_1, 3)")
    kb.assertz("veicoli(caserma_2, 7)")
    kb.assertz("veicoli(caserma_3, 10)")

    kb.assertz("veicoli_speciali(caserma_1, 0)")
    kb.assertz("veicoli_speciali(caserma_2, 2)")
    kb.assertz("veicoli_speciali(caserma_3, 5)")

    return kb
```

4. Ricerca del percorso

Abbiamo pensato di rappresentare la città dove avvengono gli incidenti tramite un grafo pesato.

Per trovare il percorso l'algoritmo A* determinerà per ogni caserma il tempo richiesto per arrivare sul luogo dell'evento e inserirà tali valori in una **knowledge base** rappresentata in **Prolog**

La ricerca del percorso è effettuata sulla base del seguente grafo:



I pesi dell'arco<X,Y> rappresentano i minuti necessari per arrivare dal nodo X al nodo Y.

4.1 Algoritmo a*

```
'''

Comportamento:
    La funzione "a_strar" calcola il percorso migliore per arrivare
    da un nodo iniziale (start) a un nodo obiettivo (goal)

Input:
    self: rappresenta l'istanza dell'oggetto a cui si fa riferimento
    start: nodo iniziale da cui raggiungere l'obiettivo (goal)
    goal: nodo obiettivo da raggiungere partendo da un nodo iniziale (start)

Output:
    percorso minimo tra il nodo iniziale (start) e il nodo obiettivo (goal)

'''
def a_star(self, start, goal):

    if start not in self.connections or goal not in self.connections:
```

```
        raise Exception("Non sono stati forniti nodi validi.")

    open_list = list()
    start.set_real_distance_value(0)
    start.set_heuristic_distance_value(self.euristic(start, goal))
    closed_list = list()
    open_list.append(start)

    while len(open_list) != 0:
        current = self.min_search(open_list)
        open_list.remove(current)
        if current.is_same(goal):
            break
        successor: Node
        for successor in self.connection(current):
            successor_current_cost = current.realDistanceValue +
self.path_weight(current, successor)
            if successor in open_list:
                if successor.realDistanceValue <= successor_current_cost:
                    break
            elif successor in closed_list:
                if successor.realDistanceValue <= successor_current_cost:
                    continue
            else:
                open_list.append(successor)

successor.set_heuristic_distance_value(self.euristic(successor, goal))
            successor.set_real_distance_value(successor_current_cost)
            successor.set_parent(current)
        closed_list.append(current)
'''

Comportamento:
    La funzione "min_search" data una lista restituisce il nodo
    avente il parametro totalDistanceValue minimo.
    (euristica + distanza stimata)

Input:
    self: rappresenta l'istanza dell'oggetto a cui si fa riferimento
    list: lista dalla quale trovare il nodo avente il parametro
          totalDistanceValue minimo

Output:
    Valore minimo nei nodi del parametro totalDistanceValue

'''
def min_search(self, list: list()):
    min = Node("", "", 100, 100)
    min.set_real_distance_value(9999)
    min.set_heuristic_distance_value(9999)

    for item in list:
        item: Node()
        if item.totalDistanceValue <= min.totalDistanceValue:
            min = item

    return min
```