

Università degli studi di Bari facoltà di
scienze MM.FF.NN

Progetto ingegneria della conoscenza

TrainDelay-project

by

Vito Proscia mat. 735975

Email: v.proscia3@studenti.uniba.it



**UNIVERSITÀ
DEGLI STUDI DI BARI
ALDO MORO**

Repository github: [TrainDelay-project](https://github.com/TrainDelay-project)

Anno accademico 2022-2023

Contents

1	Introduzione	3
1.1	Definizione obiettivo principale	3
1.2	Tool utilizzati	3
2	Rappresentazione formale della conoscenza	3
2.1	Origine dei dati	4
2.2	Descrizione dei dati	4
2.3	Grafo	5
2.4	Query Knowledge base	7
3	Machine Learning	10
3.1	Origine dataset	10
3.2	Analisi del dataset	10
3.3	Preparazione dati	11
3.4	Apprendimento automatico	12
3.5	Random forest	13
3.6	Valutazione dei modelli	14
3.7	Risultati	15
3.8	Considerazioni	16
4	Interfaccia Grafica	16
5	Sviluppi futuri	17

1 Introduzione

1.1 Definizione obiettivo principale

L'obiettivo principale del progetto è la creazione di un motore di ricerca che trova i migliori itinerari di viaggio in treno sulla base della stazione di partenza e di arrivo e che, per ogni viaggio, mostra una predizione del probabile ritardo.

Questo sistema non solo potrà far risparmiare del tempo a chi organizza dei viaggi valutando ogni singola tratta in termini di stazioni e orari di partenza e di arrivo, ma garantirà un risparmio economico ai viaggiatori garantendo che la tratta scelta dal sistema sia la minima e necessaria per arrivare alla destinazione.

1.2 Tool utilizzati

Per la sperimentazione sono stati usati diversi strumenti/librerie, i principali sono:

- **PySWIP**, libreria Python che fornisce un'interfaccia per utilizzare SWI-Prolog, usato per la rappresentazione formale della schedul dei treni,
- **NetworkX**, libreria Python utilizzata per la creazione, l'analisi e la manipolazione di reti complesse. Questa libreria fornisce un insieme di strumenti per la rappresentazione di reti e grafi, oltre a un'ampia gamma di algoritmi e funzioni per eseguire diverse operazioni su di essi,
- **scikit-learn**, libreria open-source che fornisce un vasto insieme di strumenti nell'ambito del machine learning e dell'apprendimento automatico.

2 Rappresentazione formale della conoscenza

La rappresentazione formale della conoscenza è importante per consentire l'espressione della conoscenza in modo preciso, organizzato e interpretabile da parte di sistemi informatici.

Per gestire formalmente la conoscenza, facilitando la ricerca e l'accesso a informazioni specifiche, si costruisce una **knowledge base** (base di conoscenza), una raccolta strutturata di informazioni o dati che rappresenta la conoscenza su un determinato dominio o argomento, utilizzate per immagazzinare e organizzare la conoscenza in modo che sia accessibile e utilizzabile da sistemi informatici.

2.1 Origine dei dati

Tutte le informazioni relative allo schedul dei treni sono state reperite per mezzo dell'interrogazione alle API messe a disposizione dal sito [viaggiatreno](#), mentre le informazioni relative alle stazioni sono state recuperate dal repository "trenitalia: scraping di viaggiatreno".[1]

2.2 Descrizione dei dati

La parte iniziale del progetto si è concentrata sulla rappresentazione formale attraverso fatti e regole Prolog (linguaggio di programmazione logica utilizzato per definire relazioni tra fatti e regole attraverso la logica dei predicati) dello schedule dei treni e delle stazioni, in particolare ogni treno si è ritenuto opportuno caratterizzarlo da:

1. *ID treno*, identificatore univoco del treno
2. *Tipo di treno*, regionale o nazionale
3. *ID stazione di partenza*
4. *ID stazione di arrivo*
5. *Orario di partenza* (HH:MM)
6. *Orario di arrivo* (HH:MM)
7. *Lista delle fermate*.

Di seguito è riportato un esempio:

```
1 train(320, nazionale, s01700, s01301, '15:10', '15:58', [s01700, s01307, s01301]).
2 train(321, nazionale, s01301, s01700, '18:02', '18:50', [s01301, s01307, s01700]).
3 train(322, nazionale, s01700, s01301, '17:10', '17:58', [s01700, s01307, s01301]).
4 train(323, nazionale, s01301, s01700, '20:02', '20:50', [s01301, s01307, s01700]).
5 train(324, nazionale, s01700, s01301, '19:10', '19:58', [s01700, s01307, s01301]).
6 train(325, nazionale, s01301, s01700, '22:02', '22:50', [s01301, s01307, s01700]).
```

Figure 1: Esempio di rappresentazione formale dello shedule dei treni

Mentre ogni stazione si è pensato caratterizzarla da:

1. *ID stazione*, identificatore univoco delle stazioni
2. *Nome stazione*
3. *Regione stazione*

Di seguito è riportato un esempio:

```
1 station(s06950, "BINARIO S.MARCO VECCHIO", "Toscana").
2 station(s11113, "BISCEGLIE", "Puglia").
3 station(s00870, "BISTAGNO", "Piemonte").
4 station(s01202, "BISUSCHIO VIGGIU", "Lombardia").
5 station(s11501, "BITETTO PALO DEL COLLE", "Puglia").
6 station(s03313, "BIVIO D`AURISINA", "Friuli Venezia Giulia").
```

Figure 2: Esempio di rappresentazione formale delle stazioni

Questa rappresentazione dei treni e delle stazioni nel linguaggio di programmazione logica ha fornito una base solida per la gestione e l'analisi dei dati ferroviari.

Con questa struttura dati, è stato possibile effettuare interrogazioni, trovare treni tra stazioni, e condurre altre operazioni analitiche necessarie per il progetto.

2.3 Grafo

2.3.1 Costruzione grafo

Per la possibilità di ricercare l'itinerario di viaggio migliore, cioè con il numero minimo di stazioni, si è pensato di costruire un grafo delle stazioni, dove ogni nodo rappresenta una stazione diversa (*idStazione*) e la presenza di un arco tra due nodi si traduce in un collegamento ferroviario tra le due stazioni.

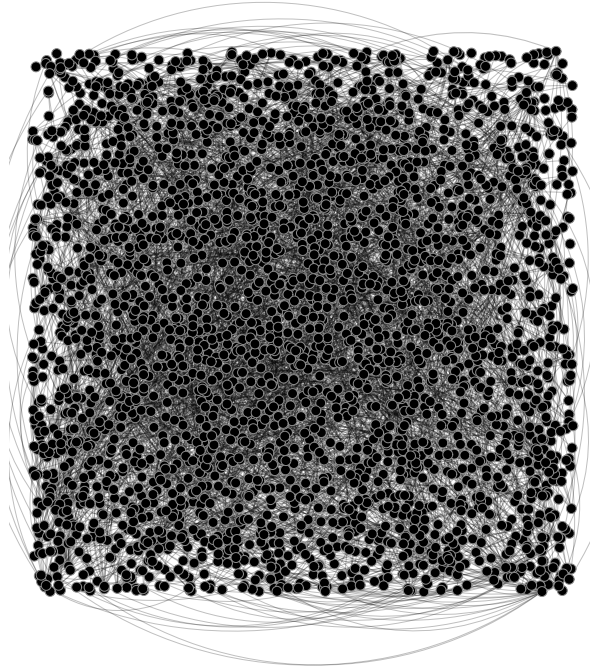


Figure 3: Grafo delle stazioni

2.3.2 Ricerca grafo

Per la ricerca del percorso più breve tra due stazioni si è utilizzato **l'algoritmo di Dijkstra**, un algoritmo di ricerca del cammino più breve in un grafo pesato con pesi non negativi, in questo caso specifico il peso di ogni arco è 1, quindi l'algoritmo cercherà il percorso con il più breve numero di nodi.

Inizia da un nodo sorgente e calcola le distanze minime da esso a tutti gli altri nodi, mantenendo una coda di priorità, durante l'esecuzione, visita i nodi adiacenti al nodo corrente e aggiorna le distanze minime se trova un cammino più breve. Implementato dalla libreria **NetworkX** che restituisce il percorso più breve sotto forma di lista di nodi.

Anche se il grafo prodotto non è propriamente pesato, si è pensato di usare l'algoritmo di Dijkstra, per una possibile estensione e miglioramento del sistema convertendo il grafo attuale in uno pesato, magari andando ad impostare come pesi degli archi le distanze che intercorrono tra le stazioni collegate.

Algorithm 1 Algoritmo di Dijkstra

```
1: procedure DIJKSTRA( $G, s$ )
2:    $dist \leftarrow$  array di distanze inizializzato a  $\infty$  per tutti i nodi
3:    $dist[s] \leftarrow 0$ 
4:    $S \leftarrow$  insieme vuoto dei nodi visitati
5:   while  $S$  non contiene tutti i nodi do
6:      $u \leftarrow$  nodo non visitato con la minima distanza in  $dist$ 
7:     Aggiungi  $u$  a  $S$ 
8:     for all nodi adiacenti  $v$  di  $u$  do
9:        $alt \leftarrow dist[u] +$  peso dell'arco tra  $u$  e  $v$ 
10:      if  $alt < dist[v]$  then
11:         $dist[v] \leftarrow alt$ 
12:      end if
13:    end for
14:  end while
15:  return  $dist$ 
16: end procedure
```

Operazione	Complessità
Inizializzazione	$O(V)$
Inserimento di nodi nella coda	$O(V \log V)$
Estrazione del nodo con distanza minima	$O(\log V)$
Aggiornamento delle distanze	$O(E \log V)$
Totale	$O((V + E) \log V)$

Table 1: Complessità dell'algoritmo di Dijkstra con heap binario

2.4 Query Knowledge base

Per andare ad interagire un Knowledge base vengono eseguite delle **query**, interrogazioni alla KB, che permettono di estrapolare informazioni specifiche. In questo caso sono state messe a disposizione delle query predefinite per poter recuperare le informazioni relative ai treni secondo le esigenze dell'utente. In particolare sono state pensate quattro query principali, che verranno eseguite in base alle esigenze dell'utente correlate alla ricerca specifica che andrà a fare.

Query messe a disposizione per andare ad interagire, in maniera efficiente, con la base di conoscenza da parte del sistema:

1. Restituisce la lista di tutti i treni (**Trains**) che partono da una determinata stazione (**StationName**)

```
1 % Rule for finding all trains departing from a station
2 trains_departure_from_station_name(StationName, Trains) :-
3     findall(TrainID, (station(DepartureStationID, StationName, _),
4         train(TrainID, _, DepartureStationID, _, _, _)), Trains).
```

Figure 4: Regola n.1

2. Restituisce la lista di treni (**Trains**) che partono da una stazione specifica (**StationName**) a un determinato orario di partenza (**Departure**).

```
1 % Rule for finding all trains leaving a station after a specific t
ime (HH:MM)
2 trains_departure_from_station_name_at_time(StationName, Departure,
3     Trains) :-
4     findall(TrainID, (station(DepartureStationID, StationName, _),
5         train(TrainID, _, DepartureStationID, _, DepartureTime, _, _), equ
6         al_major_time(DepartureTime, Departure)), Trains).
```

Figure 5: Regola n.2

3. Restituisce la lista di treni (**Trains**) che collegano due stazioni specifiche, partendo dalla stazione **DepartureStationName** e arrivando **ArrivalStationName** dopo un orario specifico **Time** (nella forma HH:MM)

```
1 % Rule for finding all trains between two stations after a specific time (HH:MM)
2 trains_departure_between_stations_name_at_time(DepartureStationName, ArrivalStationName, Time, Trains) :-
3     findall(TrainID, (station(DepartureStationID, DepartureStationName, _),
4         station(ArrivalStationID, ArrivalStationName, _),
5         train(TrainID, _, DepartureStationID, ArrivalStationID, DepartureTime, _, _),
6         equal_major_time(DepartureTime, Time))),
7     Trains).
```

Figure 6: Regola n.3

4. Restituisce la lista di treni (**Trains**) che collegano due stazioni specifiche, partendo dalla stazione **DepartureStationName** e arrivando **ArrivalStationName**

```
1 % Rule for finding all trains between two stations
2 trains_departure_between_stations_name(DepartureStationName, ArrivalStationName, Trains) :-
3     findall(TrainID, (station(DepartureStationID, DepartureStationName, _),
4         station(ArrivalStationID, ArrivalStationName, _),
5         train(TrainID, _, DepartureStationID, ArrivalStationID, _, _, _))),
6     Trains).
```

Figure 7: Regola n.4

Oltre queste query sono state prodotte altre regole, di supporto, per il funzionamento interno di queste principali, come quella per convertire le stringhe che rappresentano gli orari dal formato "HH:MM" in minuti, quella per reperire tutte le info dei treni dall'id, etc...

3 Machine Learning

Una delle funzionalità cardine del sistema è quella di predire se un treno sarà in ritardo, per implementarla è stato necessario addestrare un modello di Machine learning che in base a certe caratteristiche dello specifico treno farà una previsione più o meno corretta sul suo andamento.

In particolare il task sarà di classificazione binaria, cioè verrà predetto se un treno farà ritardo o meno basandosi su un insieme di caratteristiche e dati.

3.1 Origine dataset

Il dataset di addestramento e test è stato recuperato per mezzo dell'interrogazione tramite API al servizio viaggiotreno.it, in particolare si sono recuperate le informazioni giornaliere circa i treni (ID, tipo di treno, ...) ed in più il ritardo effettuato della corsa specifica fornendo così un insieme di dati essenziali per l'addestramento e la valutazione del modello di classificazione.

3.2 Analisi del dataset

Il dataset ottenuto è composto da circa 101169 osservazioni per otto features che, come già detto sopra, vanno a descrivere una serie di caratteristiche legate all'andamento giornaliero dei treni, abbiamo:

1. *train_id*[numeric]: identificatore univoco del treno,
2. *origin*[string]: nome della stazione di partenza,
3. *arrival*[string]: nome della stazione di arrivo,
4. *departure_time*[string]: orario di partenza (HH:MM),
5. *arrival_time*[string]: orario di arrivo (HH:MM),
6. *delay*[numeric]: ritardo registrato,
7. *train_type*[string]: tipo di treno, regionale o nazionale,
8. *detection_date*[date]: data della corsa.

	train_ID	origin	arrival	departure_time	arrival_time	delay	train_type	detection_date
0	13	M N CADORNA	LAVENO	06:39	08:23	0	regionale	2023-08-13
1	20	LAVENO	M N CADORNA	06:38	08:09	0	regionale	2023-08-13
2	25	M N CADORNA	LAVENO	08:52	10:23	0	regionale	2023-08-13
3	26	LAVENO	M N CADORNA	07:38	09:09	0	regionale	2023-08-13
4	29	M N CADORNA	LAVENO	09:39	11:23	0	regionale	2023-08-13

Figure 8: Esempio delle prime istanze del dataset

3.3 Preparazione dati

Prendendo il dataset così descritto ci sono una serie di problematiche da risolvere per poter usare i dati, in particolare andando a considerare le osservazioni notiamo che per i valori nominali, in questo caso quelli che esprimono il nome della stazione di partenza e di arrivo, ci sono dei caratteri che andrebbero modificati per formattare meglio il dataset, in particolare si sono andati a sostituire i punti, le virgole, gli accenti e le doppie virgolette con degli underscore per poter meglio gestire il dataset.

3.3.1 Analisi input features

Considerando le feature di input, cioè quelle sulle quali il modello andrà ad imparare, non tutte sono importanti per per il raggiungimento del nostro scopo, in particolare si sono escluse:

- *origin* (il modello scelto non accetta dati di tipo string),
- *arrival* (il modello scelto non accetta dati di tipo string),
- *detection_date* (nessuna correlazione sulle feature su cui fare predizione).

Per quanto riguarda le feature rimanenti si è optato per una trasformazione dei valori per adattarli al modello, specificatamente i valori di *departure_time* e *arrival_time* da stringhe nel formato HH:MM si è passati a valori numerici che rappresentano i minuti ($hour * 60 + minutes$), inoltre si è operato anche su *train_type* eseguendo una binarizzazione:

$$\begin{cases} 1 & \text{per } train_type(i) = regionale \\ 0 & \text{per } train_type(i) = nazionale \end{cases}$$

3.3.2 Analisi target feature

La target feature rappresenta l'obiettivo della nostra predizione, in questo caso *delay* che rappresenta il ritardo di una determinata corsa, essendo un task di classificazione, cioè prevedere se un determinato treno farà ritardo o meno, anche in questo caso si è eseguita una binarizzazione:

$$\begin{cases} 1 & \text{per } \textit{delay}(i) > 0 \\ 0 & \text{otherwise} \end{cases}$$

Nel dataset abbiamo una distribuzione abbastanza bilanciata dei valori per la feature *delay*.

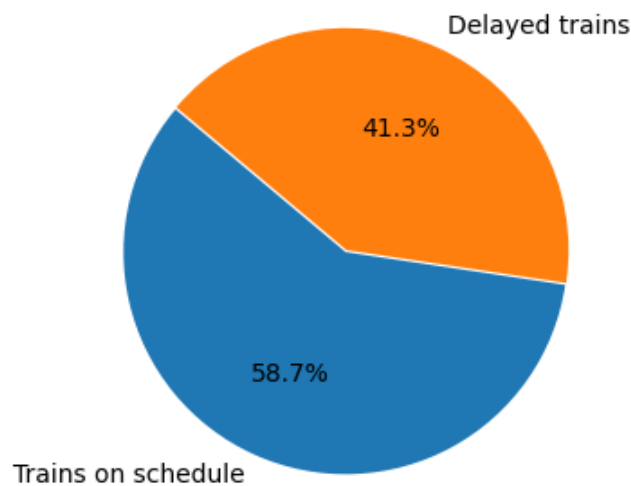


Figure 9: Distribuzione dei treni con e senza ritardo

3.4 Apprendimento automatico

Per produrre un sistema che va predire se un treno farà ritardo o meno di ricorre all'apprendimento automatico o *machine learning* che consiste

nell'addestrare un modello ad imparare dai dati e a migliorare le proprie prestazioni nei compiti specifici senza essere esplicitamente programmanti.

3.5 Random forest

Per questo caso specifico si è usata la **Random forest**, modello di apprendimento automatico che combina molteplici alberi decisionali (*decision trees*), entrando così nella categoria di modelli *ensemble*, per migliorare la previsione e la generalizzazione.

La random forest fa parte della gamma dei modelli di **apprendimento supervisionato**, cioè viene addestrato su un insieme di dati di addestramento che includono sia le caratteristiche (input features) che le risposte corrette (output features). Il modello impara a fare previsioni basate su questi esempi etichettati.

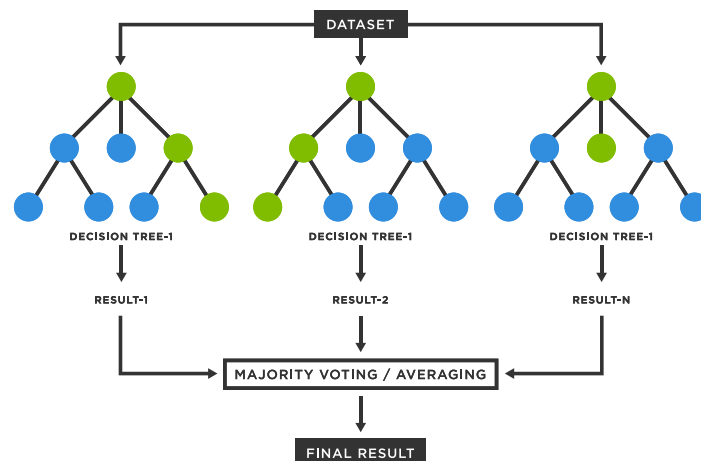


Figure 10: Esempio di random forest www.spotfire.com

3.5.1 Funzionamento

In particolare la Random Forest funziona creando un insieme di alberi decisionali, ognuno addestrato su un subset casuale dei dati di addestramento e su un subset casuale delle caratteristiche.

Questo processo introduce variabilità e riduce il rischio di sovradattamento (*overfitting*). Quando si effettua una previsione, ciascun albero fornisce una previsione, e la Random Forest combina queste previsioni per ottenere un risultato finale più accurato e stabile.

3.5.2 Configurazione scelta

Dopo varie sperimentazioni si riporta la migliore configurazione del modello, in particolare attraverso la libreria `scikit-learn` [2] si è definita una random forest le cui caratteristiche peculiari sono:

1. `'n_estimators' = 50`, che rappresenta il numero di alberi su cui lavorare
2. `criterion = "gini"`, che rappresenta il criterio di *split* degli alberi, in particolare il criterio punta a minimizzare l'impurità dei nodi, quindi per scegliere l'attributo che meglio divide i dati in base alla target feature per ogni nodo si calcola:

$$GINI(v) = 1 - \sum_{i=1}^{|c|} p_i^2$$

dove:

- v è il nodo in esame,
- c rappresenta l'insieme delle classi,
- p_i è la probabilità che un campione nel nodo v appartenga alla classe i .

infine viene scelto il nodo con minor impurità.

L'indice `gini` è stato scelto per la sua efficienza, infatti la complessità per il calcolo dell'indice per uno nodo è dell'ordine di $O(c)$, dove c è il numero di classi.

3.6 Valutazione dei modelli

Per andare a valutare i modelli prodotti dopo l'addestramento si è usata la k-Fold-Cross-Validation, che esegue la ripartizione del dataset D in k sottoinsiemi (folds), D_1, D_2, \dots, D_k prevede k iterazioni ed all' i -esima iterazione il sottoinsieme D_i sarà usato come dataset di test, mentre l'unione degli altri sarà usata per allenare il modello.

Alla fine delle k iterazioni, vengono raccolti i risultati delle valutazioni e calcolate le metriche di performance, come l'accuratezza o l'F1-score, per valutare quanto bene il modello si comporta in media su dati di test diversi.

3.6.1 Metriche scelte

Ai fini della valutazione si hanno a disposizione moltissime metriche adatte per moltissime situazioni, le più comuni sono:

$$Precision = \frac{TP}{TP + FP} \quad Recall = \frac{TP}{TP + FN}$$

Inoltre per avere un quadro della situazione più dettagliato si presentano altre metriche che mettono in correlazione le due precedenti, come:

- $f\beta$ - score, combinazione delle metriche classiche di *precision* e *recall* bilanciate da un parametro β nella media armonica.

$$f\beta - score = \frac{(1 + \beta^2) \cdot Precision \cdot Recall}{\beta^2 \cdot Precision + Recall}$$

- $AUC-ROC$ (Area Under the Receiver Operating Characteristic Curve), area sotto la curva che rappresenta la relazione tra il tasso di veri positivi (True Positive Rate, TPR) e il tasso di falsi positivi (False Positive Rate, FPR) al variare della soglia di classificazione.

$$TPR = \frac{TP}{TP + FN} \quad FPR = \frac{FP}{FP + TN}$$

3.7 Risultati

In questa sezione verranno riportati i risultati di tutte le metriche discusse precedentemente:

- **Precision:** 0.9441
- **Recall:** 0.8664
- **Matrice di confusione:**

Actual_negative	Actual_positive	
11522	425	Predict_negative
1107	7180	Predict_positive

- **F2-score:** 0.9242
- **AUC-ROC:** 0.9154

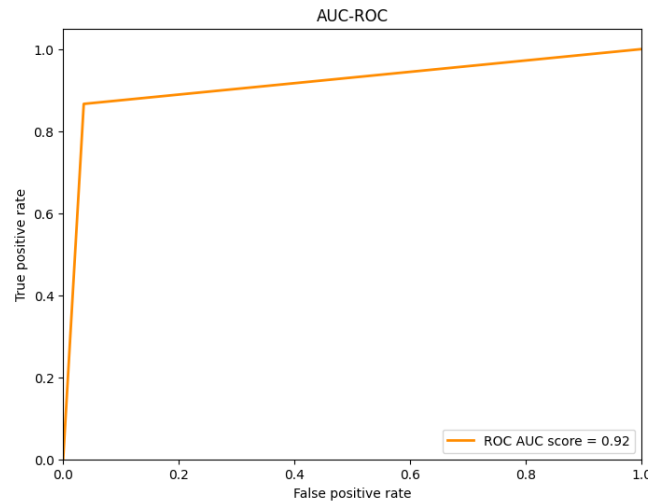


Figure 11: Grafico AUC-ROC

3.8 Considerazioni

In generale il modello di classificazione mostra prestazioni solide, con un'accuratezza elevata, alta precisione e una buona capacità discriminativa. Tuttavia, il richiamo è leggermente inferiore al 90%, suggerendo possibili aree di miglioramento. L'F2 score bilanciato fornisce un buon compromesso tra precisione e richiamo, indicando un equilibrio nel modello.

4 Interfaccia Grafica

Per interagire andare ad interagire con il sistema, all'avvio viene mostrato un menù, navigabile con i tasti direzionali \uparrow e \downarrow , con tre opzioni:

1. **Cerca treno tra due stazioni**, per rintracciare tutti i treni tra la stazione di partenza e la stazione di arrivo dopo un determinato orario,
2. **Cerca itinerario**, fornisce un itinerario, cioè tutte le informazioni relative ai treni da prendere (orario, tipo, etc...) tra due stazioni,
3. **Uscire**, per terminare l'applicazione.

Di seguito verranno riportate alcune immagini relative alla UI del sistema, includendo il menù principale e le diverse funzionalità:

relative a treni e stazioni ed integrazione con altri servizi ferroviari (Italo, Frecciarossa, ...)

2. Implementazione di una vera e propria interfaccia grafica
3. Miglioramento della ricerca andando a suggerire all'utente, sulla base dei caratteri inseriti, le stazioni che iniziano con quei caratteri
4. Integrazione di dati in tempo reale andando a fornire all'utente informazioni in *real time*

References

- [1] @sebas. *trenitalia: scraping di viaggiatreno*. 2012. URL: <https://github.com/sabas/trenitalia>.
- [2] G. Varoquaux A. Gramfort D. Cournapeau O. Grisel **and** A. Mueller. *Scikit-learn Machine Learning in Python*. **version** 1.3.1. URL: <https://scikit-learn.org/stable/index.html>.