Università degli Studi di Salerno

Progetto del corso di

Intelligenza artificiale: Metodi ed Applicazioni

Anno: 2024/2025

GRAN PREMIO MIVIA

Gruppo 03 - Membri
Claudia Montefusco: 0612707404
Rossella Pale: 0612707284
Alessio Leo: 0612707279
Emanuele Tocci 0612707488

Progetto del corso di Intelligenza artificiale

Gruppo 03

2024/2025

Contents

1 Introduzione

Il progetto prevede lo sviluppo di un sistema di guida autonoma all'interno dell'ambiente **TORCS** (The Open Racing Car Simulator), un simulatore open source in 3D dedicato alle corse automobilistiche. Nel nostro caso, il veicolo non affronterà altri concorrenti in pista: l'obiettivo sarà esclusivamente quello di completare un giro di pista nel minor tempo possibile, senza incidenti e mantenendo sempre il controllo del mezzo.

1.1 Approccio adottato (Behavioral Cloning)

Tenendo conto di quanto detto, per la realizzazione del progetto, in particolare per l'addestramento del modello AI, si è ritenuto ottimale utilizzare un approccio di **behavioral cloning** (BC). Questa tecnica di apprendimento supervisionato mira a replicare il comportamento umano osservando le sue azioni in risposta a determinati input.

Nell'ambito della guida autonoma, BC consente di **addestrare un classificatore** imitando le decisioni di guida di un pilota umano (come la direzione dello sterzo) a partire dai dati sensoriali. In questo modo, l'intero processo decisionale viene *appreso direttamente dai dati.*

Tuttavia, il BC presenta alcune *limitazioni* che possono compromettere l'efficacia in scenari più complessi. Uno degli aspetti critici è la dipendenza dai dati di addestramento: se il dataset non copre una **varietà sufficiente** di situazioni, il modello potrebbe incontrare difficoltà nell'adattarsi a condizioni impreviste. Inoltre, vi è il rischio che piccoli errori durante la guida autonoma si amplifichino progressivamente, portando il veicolo fuori traiettoria.

2 Scelte progettuali

Per raggiungere gli obiettivi sopra descritti, abbiamo deciso di strutturare il progetto in diverse fasi:

- **Raccolta dati**: Un pilota umano guida il veicolo all'interno del simulatore, mentre il sistema registra in tempo reale sia i dati sensoriali provenienti dal veicolo sia i comandi di guida impartiti. Questa fase permette di acquisire esempi concreti del comportamento umano in diverse situazioni di guida.
- **Creazione del dataset**: I dati raccolti vengono filtrati e normalizzati per eliminare eventuali anomalie e garantire l'omogeneità delle informazioni. Successivamente, i dati vengono salvati in un file CSV che costituisce la base per la fase di classificazione automatica.
- **Implementazione dell'agente autonomo**: Il classificatore KNN, utilizzando il dataset raccolto, viene integrato nel simulatore per controllare direttamente il veicolo. L'agente autonomo riceve i dati sensoriali in tempo reale, li confronta con gli esempi memorizzati e determina il comando di guida più appropriato in base ai k vicini più simili.
- **Validazione e Test**: Il comportamento dell'auto viene valutato su pista senza intervento umano, per verificare la capacità del sistema di completare i giri in autonomia. La validazione si concentra sia sulla correttezza delle azioni eseguite che sulla rapidità e fluidità della guida.

2.1 Scelta del classificatore

Nel processo di sviluppo del sistema di guida autonoma, la scelta del classificatore riveste un ruolo fondamentale poiché determina il metodo con cui il veicolo autonomo interpreterà i dati sensoriali e prenderà decisioni in tempo reale. Il classificatore utilizzato è un **K-Nearest Neighbors** (KNN), implementato nella classe `NearestNeighbor`.

Il classificatore si basa su un approccio supervisionato in cui, dato un nuovo campione (`Sample`) da classificare, si cercano i k campioni più vicini presenti nel dataset di addestramento, utilizzando una struttura *KD-Tree* per ottimizzare la ricerca. La classe del campione in input viene determinata tramite voto di maggioranza tra le classi dei k vicini trovati.

Tuttavia, un aspetto negativo di tale classificatore è la sua scarsa scalabilità con dataset grandi o con molte feature, poiché la ricerca dei vicini può diventare **costosa**. Per mitigare questo problema, abbiamo deciso di realizzare dataset più leggeri e mantenere il piú basso possibile il valore di k.

2.1.1 Scelta del parametro K

L'obiettivo principale è trovare un **valore ottimale** per il parametro K del classificatore, ossia il **numero dei vicini** che considerare per determinare l'azione da intraprendere. Un valore adeguato di K permette al sistema di avere una scelta più ampia di record vicini, migliorando così la **precisione** e l'affidabilità delle risposte del sistema. Il processo di ottimizzazione di tale parametro, ha comportato il testing di **diversi valori** per osservare come influenzano le prestazioni del sistema in termini di tempo di completamento del giro e numero di collisioni.

È stato fin da subito evidente un **peggioramento** delle prestazioni all'aumentare del valore di K. In linea generale abbiamo ottenuto **prestazioni accettabili** fino ad un valore di $K = 3$. Nel nostro caso abbiamo deciso di utilizzare $K = 1$ per entrambi i classificatori poiché consente di ottenere le migliori prestazioni in termini di tempo e di risposta in curva.

Figure 1: Risultati simulati su 10 giri

2.2 Scelta delle feature

La selezione delle feature rappresenta una fase cruciale nella progettazione di un classificatore di *machine learning*, in quanto incide direttamente sulle prestazioni, sulla generalizzazione e sull'efficienza del modello. Nel nostro progetto, abbiamo scelto di limitare il numero di feature utilizzate, privilegiando solo quelle considerate realmente indispensabili per la guida autonoma. Questa scelta è motivata dal fatto che l'inclusione di troppe feature, oltre ad aumentare la complessità computazionale, può introdurre rumore e ridurre l'efficacia del classificatore KNN, particolarmente sensibile ad attributi irrilevanti o ridondanti.

- **Sensori di distanza dalla pista (track)**: misurano la distanza tra la macchina e i bordi della pista in direzioni predefinite, aiutando il classificatore a capire la posizione del veicolo rispetto ai bordi. Ne vengono selezionati solo 7:
  - `track4`: -30 °
  - `track6`: -15 °
  - `track8`: -5 °
  - `track9`: 0 °
  - `track10`: 5 °
  - `track12`: 15 °
  - `track14`: 30 °
- **Posizione trasversale sulla pista** (`trackPos`): indica la posizione laterale del veicolo rispetto al centro pista, utile per capire quando quest'ultimo si sposta troppo verso un lato.
- **Angolo rispetto all'asse della pista** (`angle`): indica quanto il veicolo è orientato rispetto alla direzione della pista, utile per capire se si è perso l'allineamento con la pista.
- **Velocità longitudinale** (`speedX`): misura la velocità del veicolo lungo l'asse longitudinale.
- **Velocità laterale** (`speedY`): misura la velocità del veicolo lungo l'asse trasversale.

2.3 Scelta delle azioni

Per mantenere il codice e la classificazione il piú semplice possibile, abbiamo deciso di mappare solamente 6 azioni basilari.

Classe	Azione
0	Gira a sinistra
1	Accelera
2	Gira a destra
3	Frena
4	Retromarcia
5	Decelerazione

Table 1: Mapping tra etichetta numerica e azione simbolica

3 Architettura dell'applicazione

3.1 Organizzazione codice sorgente

Il progetto è organizzato in modo modulare per facilitare la gestione, la manutenzione e l'estendibilità del codice. In particolar modo abbiamo deciso di riorganizzare il tutto in diversi package per suddividere al meglio le varie responsabilitá e mantenere l'ambiente di lavoro il più pulito possibile.

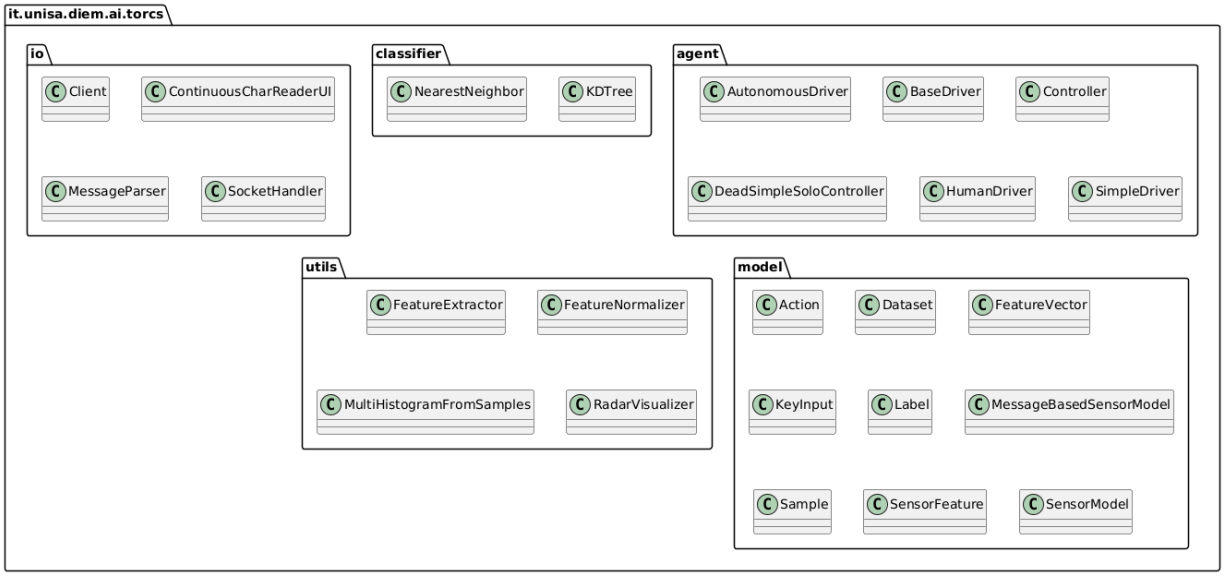


Figure 2: Diagramma dei package

3.2 Panoramica sui modelli di dati

Per comprende al meglio come è stato strutturato il progetto é necessario spendere due parole sui modelli di dati implementati, inseriti all'interno dell'omonimo package `model`.

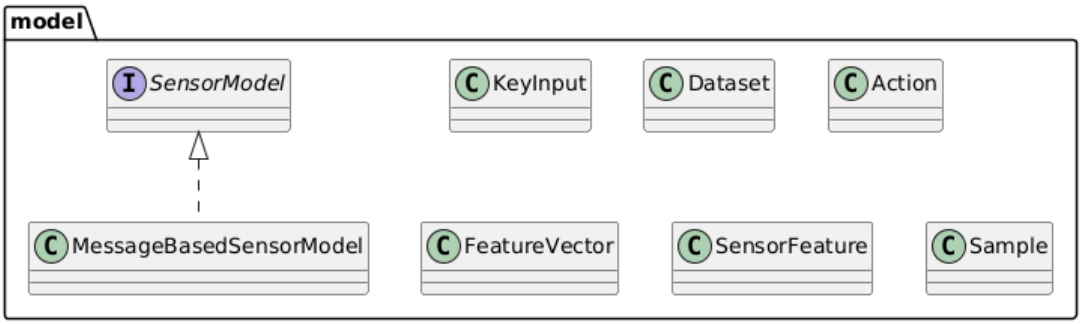


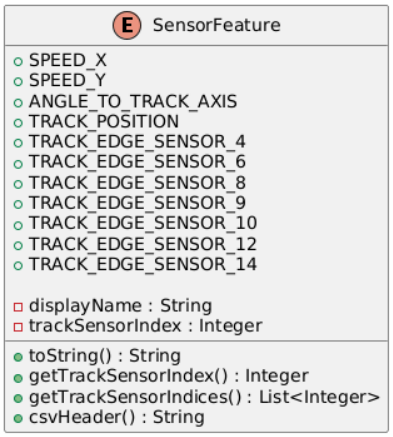
Figure 3: Diagramma del package Model

3.2.1 KeyInput

Classe statica utilizzata per tracciare in tempo reale lo stato dei principali **tasti** di controllo durante la guida manuale in TORCS. Ogni **variabile booleana** rappresenta se un determinato tasto (accelerazione, retromarcia, sterzata sinistra/destra, freno) è attualmente **premuto**.

La classe viene aggiornata dinamicamente dall'interfaccia grafica (`ContinuousCharReaderUI`) e fornisce un'interfaccia semplice e centralizzata per i controller manuali come `HumanDriver`, consentendo di tradurre gli input dell'utente in **comandi di guida**.

3.2.2 SensorFeature

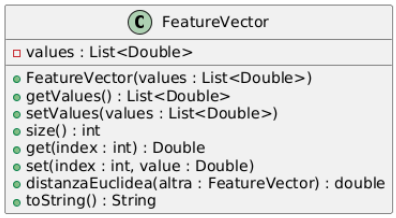


Enumerazione che definisce l'**insieme delle features** (sensori) utilizzate per la creazione del relativo vettore (`FeatureVector`). Ad ogni feature è associata un nome leggibile (`displayName`) e un **indice numerico** identificativo (`trackSensorIndex`).

Serve come **riferimento centralizzato** per tutte le feature utilizzate nel processo di classificazione... in questo modo si limita il numero di modifiche necessarie qualora si vogliano cambiare le features scelte.

Figure 4: Diagramma della classe SensorFeature

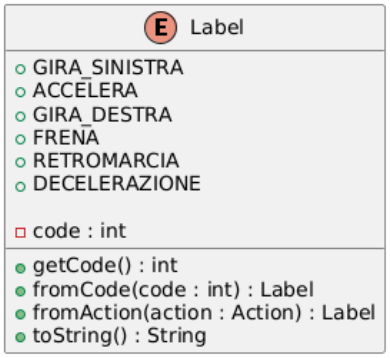
3.2.3 FeatureVector



FeatureVector rappresenta un **vettore di feature**, ovvero una lista ordinata di valori Double corrispondenti ai sensori specificati nell'enumerazione `SensorFeature`. Ogni istanza descrive lo stato del veicolo in un dato istante, ed è parte integrante di un `Sample`.

Figure 5: Diagramma della classe FeatureVector

3.2.4 Label



Il modello appresenta l'**azione simbolica** che un agente puó compiere nel simulatore TORCS. Si tratta di un'**enumerazione** che definisce un insieme finito di **etichette discrete**, ognuna corrispondente a un **comportamento** del veicolo all'interno del simulatore. Ogni label é associata ad un **codice** intero univoco, utilizzato per semplificare la fase di classificazione.

Figure 6: Diagramma della classe Label

Le label vengono derivate a partire da un'**azione continua** (`Action`), tramite una funzione di **discretizzazione** che assegna priorità alle manovre più rilevanti (es. retromarcia o frenata). Queste etichette sono utilizzate per annotare le features e costruire il (`Sample`) nel dataset, rendendo possibile l'apprendimento supervisionato del comportamento di guida. Abbiamo identificato le seguenti azioni: vedi **??**sec:azioni).

In sintesi, Label fornisce un **vocabolario** simbolico ridotto per rappresentare le decisioni di guida.

3.2.5 Sample

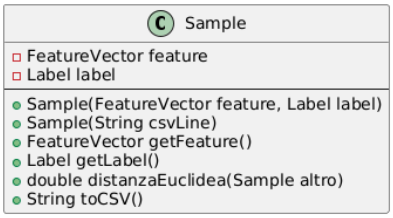


Figure 7: Diagramma della classe Sample

Modello rappresentativo di un singolo **campione etichettato**, composto da un vettore di feature numeriche (`FeatureVector`) e da una label (`Label`) associata.

3.2.6 Dataset

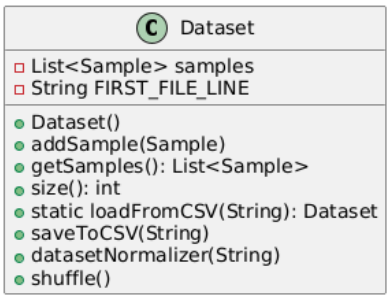


Figure 8: Diagramma della classe Dataset

Il modello rappresenta un insieme supervisionato di dati raccolti durante le sessioni di guida nel simulatore TORCS. Ogni dataset è costituito da una **lista di campioni** (`Sample`).

La struttura del dataset riflette l'approccio di apprendimento supervisionato adottato nel progetto: i dati raccolti vengono serializzati in **formato CSV** per essere successivamente elaborati e utilizzati in fase di addestramento.

Il dataset può essere mescolato e normalizzato per rendere più efficace il processo di training del classificatore.

3.3 Panoramica sugli Agenti

All'interno del package `agent` sono presenti 3 diversi client di guida:

- SimpleDriver
- HumanDriver
- AutonomousDriver

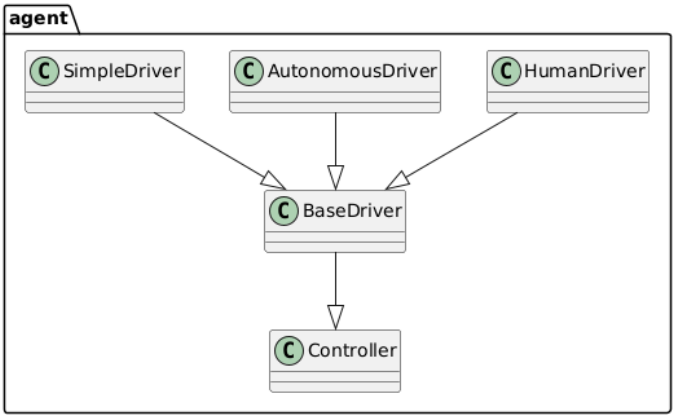


Figure 9: Diagramma del package Agent

3.3.1 BaseDriver

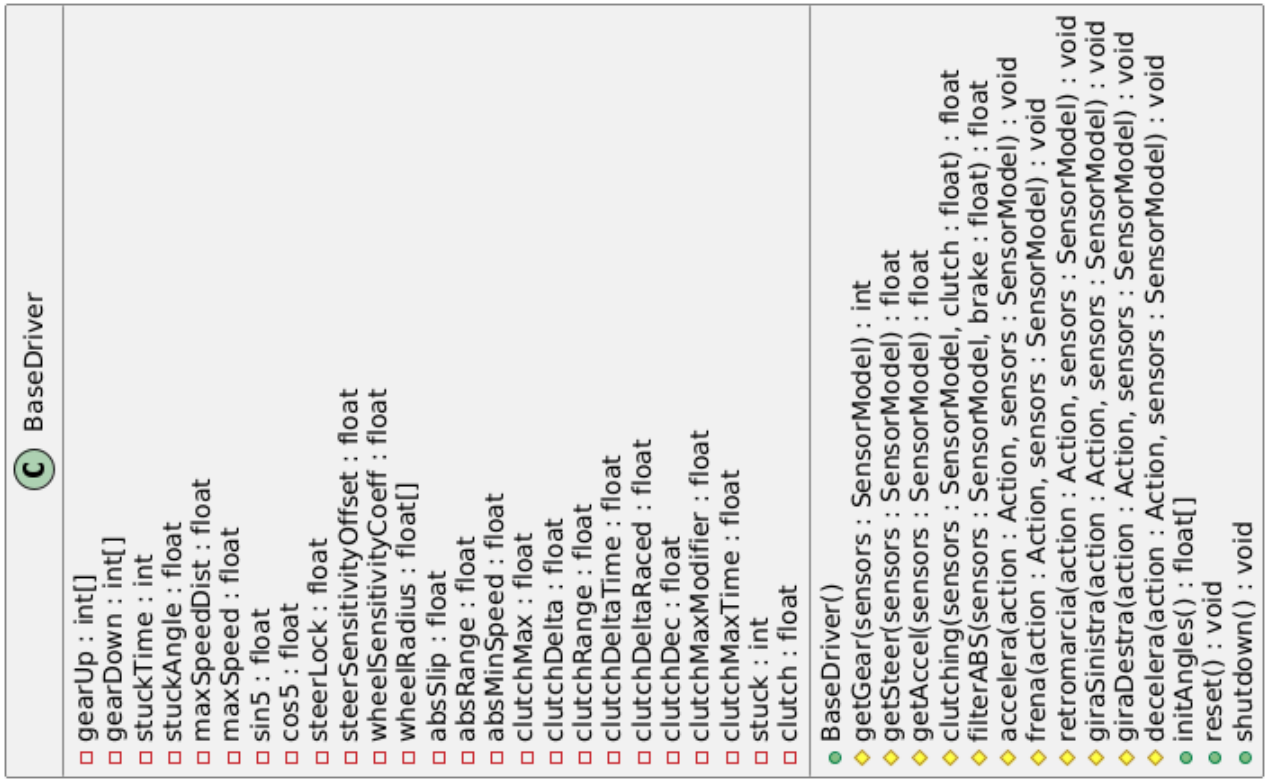


Figure 10: Diagramma della classe BaseDriver

Classe astratta che fornisce l'infrastruttura comune per tutti i driver di guida in TORCS. Implementa metodi e utility essenziali per:

- **Gestione cambio marcia**: Algoritmi automatici per cambio marcia basati su soglie RPM configurabili
- **Sistema sterzata**: Calcolo dell'angolo di sterzo considerando posizione sulla pista, velocità e angolo rispetto all'asse
- **Controllo accelerazione**: Logica per determinare la velocità target in base ai sensori di bordo pista
- **Sistema ABS**: Simulazione di un sistema antibloccaggio ruote durante la frenata

- **Gestione frizione**: Controllo automatico della frizione per partenze e cambi marcia
- **Azioni base di guida**: Metodi predefiniti per accelerare, frenare, sterzare e manovre di recupero

La classe definisce costanti di configurazione per tutti i parametri fisici del veicolo e, in particolare, le **funzioni di guida**:

```
void accelera(Action action, SensorModel sensors) {
    action.accelerate = 1.0;
    action.brake = 0.0;
    action.steering = 0f;
    action.clutch = clutching(sensors, (float) action.clutch);
    action.gear = getGear(sensors);
}

void frena(Action action, SensorModel sensors) {
    action.brake = filterABS(sensors, 1f);
    action.clutch = clutching(sensors, (float) action.clutch);
    action.gear = getGear(sensors);
    action.accelerate = 0.0f;
}

void retromarcia(Action action, SensorModel sensors) {
    action.gear = -1;
    action.accelerate = 1.0f;
    action.brake = 0.0;
    action.clutch = clutchMax;
}

void giraSinistra(Action action, SensorModel sensors) {
    if (sensors.getSpeed() < 15)
        action.accelerate = 0.5;
    action.steering = 0.3f;
    action.brake = 0.0f;
    action.gear = sensors.getGear();
    action.clutch = 0.0f;
}

void giraDestra(Action action, SensorModel sensors) {
    if (sensors.getSpeed() < 15)
        action.accelerate = 0.5;
    action.steering = -0.3f;
    action.brake = 0.0f;
    action.gear = sensors.getGear();
    action.clutch = 0.0f;
}

void decelera(Action action, SensorModel sensors) {
    action.steering = 0.0f;
    action.accelerate = 0.2f;
    action.brake = 0.0f;
    action.gear = sensors.getGear();
    action.clutch = 0.0f;
}
```

3.3.1.1 SimpleDriver Driver di guida autonoma basato su regole. Il suo comportamento si articola in due modalità principali:

- **Modalitá Normale:**
  - Calcolo automatico di accelerazione/frenata basato sui sensori di distanza
  - Gestione cambio marcia e sterzo adattivi alla velocità
  - Applicazione di ABS e controllo frizione
- **Modalitá Recovery:**
  - Rilevamento automatico di situazioni di blocco (angolo > 30 per oltre 100 cicli)
  - Manovre di recupero con retromarcia e correzione sterzo
  - Logica per riallineamento con l'asse della pista

3.3.1.2 HumanDriver Driver di guida manuale la guida tramite input da tastiera. Le sue caratteristiche principali includono:

- **Controllo manuale:** lettura in tempo reale dei comandi da tastiera;
- **Raccolta dati:** Sistema di dataset collection che registra:
  - Dataset grezzo e normalizzato di tutte le azioni;
  - Dataset separati per guida normale e manovre di recupero;
- **Salvataggio automatico:** Export dei dataset in formato CSV alla chiusura della simulazione

La classe é stata utilizzata per la creazione dei vari dataset forniti.

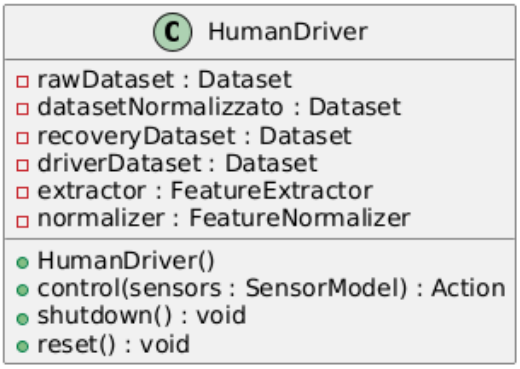


Figure 11: Diagramma HumanDriver

3.3.1.3 AutonomousDriver Agente di guida autonoma che combina **machine learning** con strategie di recupero tradizionali. La classificazione viene effettuata mediante classificatore KNN (vedi: 5).

- **Architettura dual-KNN**
  - **DriverKNN:** Classificatore per la guida normale, addestrato su dataset di guida umana
  - **RecoveryKNN:** Classificatore specializzato per situazioni critiche e manovre di recupero

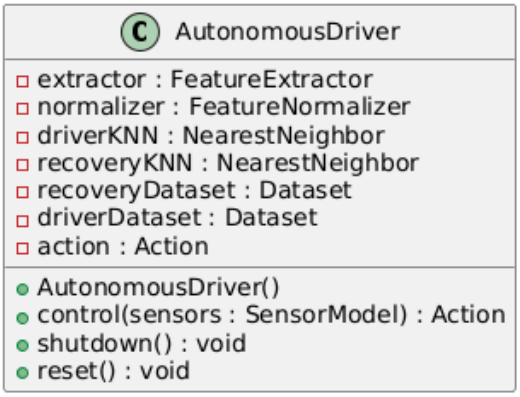


Figure 12: Diagramma AutonomousDriver

- **Preprocessing:** Estrazione e normalizzazione automatica delle features dai sensori

- **Strategia di controllo**
  - **Modalitá di guida standard:** Utilizza classificazione K-Nearest Neighbor (`DriverKNN` 5) per predire l'azione ottimale basandosi sui pattern appresi dal dataset di training
  - **Modalitá Recovery:** Utilizza il secondo classificatore (`RecoveryKNN`) per predire l'azione ottimale basandosi sui pattern appresi dal dataset di training

3.4 Panoramica sulle classi utilitarie

Il package `utils` contiene alcuni classi di utilità che possono essere utilizzate in vari contesti. Dal momento che, durante la progettazione abbiamo riscontrato diverse difficoltà (vedi 7.1), abbiamo predisposto un sotto-package apposito `debugging` all'interno del quale sono presenti alcune classi che ci hanno aiutato molto a rilevare eventuali anomalie.

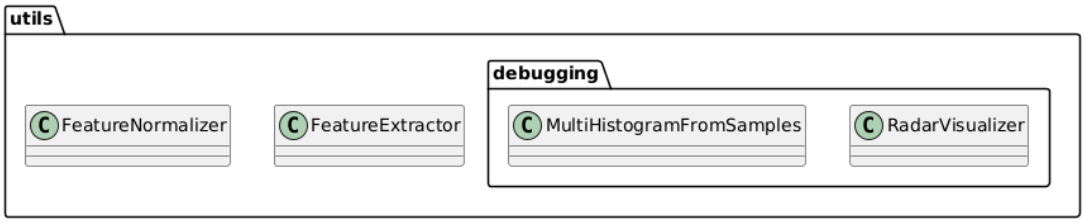


Figure 13: Diagramma del package Utils

3.4.1 FeatureExtractor

La classe FeatureExtractor è responsabile dell'**estrazione delle caratteristiche** (feature) rilevanti da un oggetto `SensorModel`, che rappresenta lo stato attuale dei sensori del veicolo. A partire da un sottoinsieme predefinito di variabili selezionate tramite l'enum `SensorFeature` costruisce un oggetto `FeatureVector` contenente i valori numerici ordinati.

3.4.2 FeatureNormalizer

La classe FeatureNormalizer si occupa della **normalizzazione** dei vettori di feature (`FeatureVector`) estratti dallo stato sensoriale del veicolo mediante l'utility `FeatureExtractor`.

Per ciascuna feature (come velocità, posizione, angolo, o distanza dai bordi pista), la classe utilizza un **range predefinito** di valori minimi e massimi attesi. I valori fuori range vengono **clippati** per evitare distorsioni. Il risultato é un nuovo `FeatureVector` normalizzato, compatibile con il classificatore implementato.

3.4.2.1 Implementazione del normalizzatore

Per la normalizzazione abbiamo deciso di utilizzare un **normalizzatore min-max** che trasforma il valore in input (x) in un range $[0; 1]$:

$$x_{norm} = \frac{x - min}{max - min}$$

```
1 private static double normalizzatoreMinMax(double data, double min, double
      max) {
2     double normalized = (data - min) / (max - min);
3     return Math.max(0.0, Math.min(1.0, normalized));  // clipping
4 }
```

Il mantenimento di tale range é garantito dalla funzione di **clipping**.

3.4.3 RadarVisualizer

La classe RadarVisualizer fornisce una **rappresentazione grafica dei sensori di bordo** pista (*track edge sensors*). La classe disegna i raggi corrispondenti ai sensori scelti su un pannello 2D, mostrando la distanza rilevata da ciascun sensore tramite linee e le rispettive etichette.

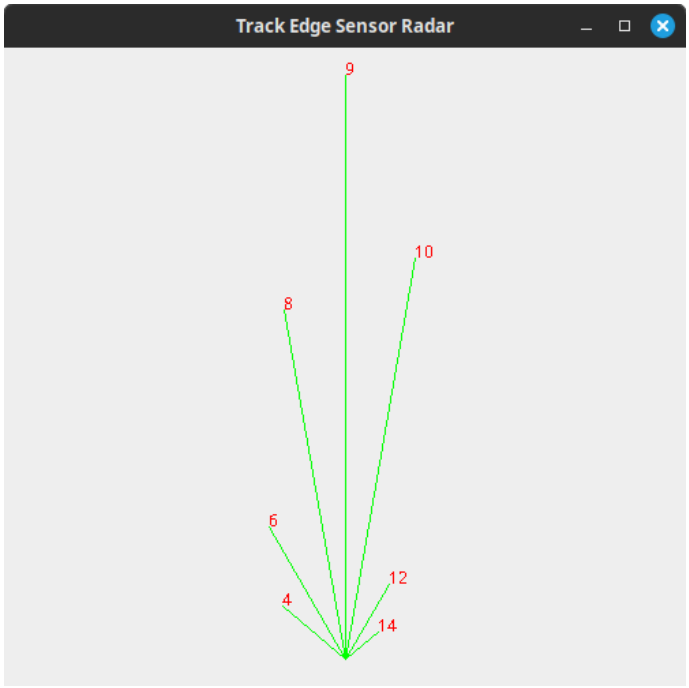
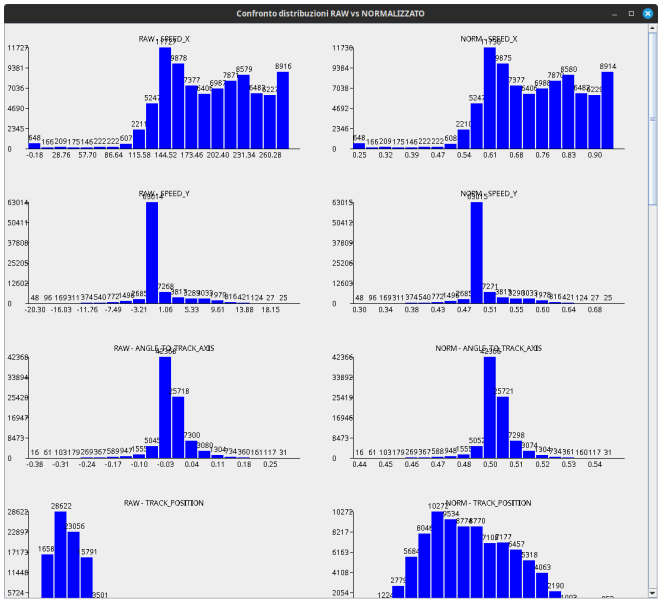
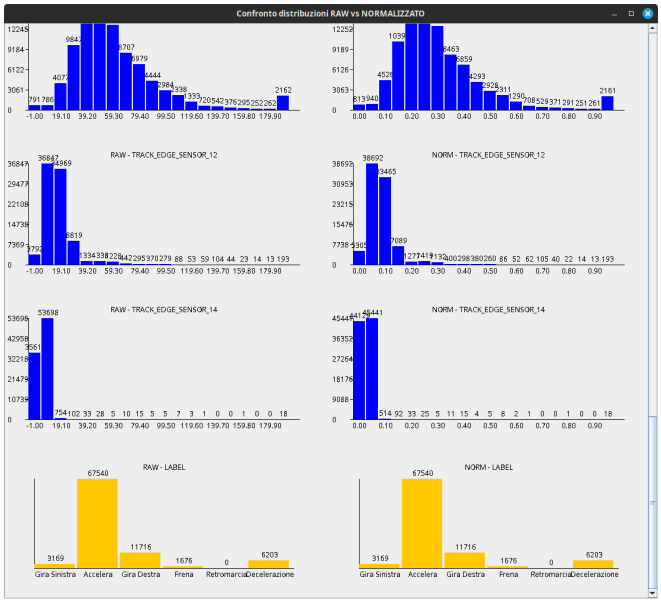


Figure 14: Radar Visualizer

3.4.4 MultiHistogramFromSamples

La classe MultiHistogramFromSamples è uno strumento grafico per l'**analisi esplorativa** dei dati raccolti durante la guida in TORCS. La sua funzione principale è il confronto tra le **distribuzioni dei valori** delle features in dataset differenti.





3.5 Panoramica sugli script forniti

All'interno del package principale sono stati inseriti alcuni script che semplificano le fasi di compilazione ed esecuzione:

- `build.sh`: compila tutti i file presenti nella cartella `src` e posiziona i compilati in `classes`
- `autonomousRun.sh`: avvia l'agente di guida autonoma (`AutonomousDriver`)
- `humanRun.sh`: avvia l'agente di guida manuale (`HumanDriver`)
- `simpleRun.sh`: avvia l'agente di guida autonoma mediante regole (`SimpleDriver`)
- `histogram.sh`: avvia il tool di visualizzazione delle distribuzioni delle features

Di default sono impostati:

- **host:** localhost
- **port:** 3001

4 Raccolta dati e creazione dataset

La prima fase del *behavioral cloning* consiste nella raccolta dei dati necessari per costruire il dataset che verrà poi utilizzato dall'agente autonomo.

La raccolta avviene tramite la guida manuale del veicolo all'interno del simulatore TORCS. Il processo inizia avviando la classe `HumanDriver`, che permette al pilota umano di controllare l'auto tramite l'interfaccia grafica `ContinuousCharReaderUI`.

Il controllo del veicolo si basa su un sistema che ascolta gli eventi della tastiera: quando vengono premuti o rilasciati i tasti `W`, `A`, `S`, `D` e `spazio`, la classe `KeyInput` aggiorna lo stato dei comandi di guida (accelerazione, sterzo, freno). In questo modo, ad ogni istante, il sistema registra sia i dati sensoriali del veicolo sia i comandi impartiti dal pilota, creando così un dataset di esempi utili per l'addestramento dell'agente.

```
@Override
public void keyPressed(KeyEvent e) {
    switch (Character.toLowerCase(e.getKeyChar())) {
        case 'w': KeyInput.up = true; break;
        case 'a': KeyInput.left = true; break;
        case 's': KeyInput.down = true; break;
        case 'd': KeyInput.right = true; break;
    }
    if (e.getKeyCode() == KeyEvent.VK_SPACE) {
         KeyInput.brake = true;
    }
}
```

`HumanDriver` legge lo stato dei comandi tramite `ContinuousCharReaderUI` e decide quali azioni (rappresentate dall'oggetto `Action`) eseguire.

```
// Leggo i comandi dalla tastiera (KeyInput)
Action action = new Action();
double speedX = sensors.getSpeed();

// Accelerazione, freno, retromarcia
if (KeyInput.brake) {
    frena(action, sensors);
} else if (KeyInput.down && speedX < 5.0) {
    retromarcia(action, sensors);
} else if (KeyInput.up) {
    accelera(action, sensors);
```

```
} else {
    action.accelerate = 0.0;
    action.brake = 0.0;
}

// Sterzo
if (KeyInput.left && !KeyInput.right) {
    giraSinistra(action, sensors);
} else if (KeyInput.right && !KeyInput.left) {
    giraDestra(action, sensors);
} else {
    action.steering = 0.0;
}
```

Una volta impostati i parametri dell'oggetto `Action`, si ricava la corrispettiva `Label` tramite il metodo statico `Label.fromAction()`. Si estraggono e si normalizzano le features (`FeatureVector`) e si crea l'oggetto `Sample` che verrá poi inserito all'interno del relativo oggetto `Dataset`.

Tale oggetto sarà poi "convertito" in formato `CSV` e salvato nella relativa directory (`data`) solamente durante l'esecuzione del metodo `shutdown`.

```
@Override
public void shutdown() {
    rawDataset.saveToCSV("data/raw_dataset.csv");
    datasetNormalizzato.saveToCSV("data/dataset_normalizzato.csv");
    driverDataset.saveToCSV("data/driver_dataset.csv");
    recoveryDataset.saveToCSV("data/recovery_dataset.csv");
}
```

4.1 Dataset utilizzati

Durante la fase di addestramento, servendoci della classe `HumanDriver`, abbiamo utilizzato l'ambiente di simulazione TORCS per eseguire diverse gare. L'obiettivo era quello di creare un dataset che rappresentasse al meglio le diverse condizioni di guida, registrando un'ampia gamma di situazioni. In particolar modo abbiamo deciso di salvare i dati su 4 dataset differenti, in modo tale da semplificare le successive fasi di debug:

- **rawDataset**: dataset contenente l'insieme completo dei dati grezzi;
- **datasetNormalizzato**: dataset contenente l'insieme completo dei dati normalizzati;
- **driverDataset**: dataset contenente un sottoinsieme di dati utilizzati per la guida all'interno della pista;
- **recoveryDataset**: dataset contenente un sottoinsieme di dati utilizzati per la gestione della modalitá *Recovery*;

Si tiene comunuque presente che (effettivamente) gli unici dataset utilizzati dall'agente di guida autonoma sono gli ultimi 2... l'insieme completo dei dati ci è servito solamente per semplificare la fase di debug.

5 Implementazione del classificatore KNN

Il classificatore **K-Nearest Neighbors** (KNN) rappresenta il nucleo del processo decisionale del sistema di guida autonoma, permettendo di *analizzare gli stati del veicolo e prevedere l'azione più ap-*

propriata in base ai dati raccolti. La classe che si occupa della sua realizzazione è `NearestNeighbor`, che sfrutta internamente una sttruttura KDTree per ottimizzare la ricerca dei vicini.

1. All'avvio, il classificatore riceve in input un dataset (`Dataset`) e costruisce un KD-Tree con i relativi campioni (`Sample`). Ogni nodo dell'albero rappresenta un punto nello spazio delle feature (`FeatureVector`)

2. Quando arriva un nuovo stato del veicolo (campione di test), viene calcolata la distanza euclidea rispetto ai punti memorizzati, e viene recuperata la lista dei k **vicini più simili** tramite la funzione `kNearestNeighbors`.

3. Una volta trovati i k vicini, il metodo `classify` analizza i campioni trovati e assegna al nuovo stato la classe (mediante un numero intero) piú frequenti tra essi. Tale codice sará poi converito in un oggetto `Label`.

```
public int classify(Sample testPoint, int k) {
    List<Sample> kNearestNeighbors = findKNearestNeighbors(testPoint, k);

    // Azzeramento dei conteggi
    Arrays.fill(classCounts, 0);

    // Conta le occorrenze di ciascuna classe tra i vicini
    for (Sample neighbor : kNearestNeighbors) {
        int classCode = neighbor.getLabel().getCode();
        classCounts[classCode]++;
    }

    // Seleziona la classe con il maggior numero di occorrenze
    int maxCount = -1;
    int predictedClass = -1;
    for (int i = 0; i < classCounts.length; i++) {
        if (classCounts[i] > maxCount) {
            maxCount = classCounts[i];
            predictedClass = i;
        }
    }
    return predictedClass;
}
```

Sia `NearestNeighbor` che `KD-Tree` sono stati opportunamente modificati al fine da poter lavorare direttamente con gli oggetti definiti, in particolar modo `Dataset` e `Sample`.

6 Risultati ottenuti

7 Problematiche Riscontrate

Durante la realizzazione del progetto sono emerse diverse **problematiche** che hanno rallentato lo sviluppo del driver.

7.1 Selezione delle features

Le prime difficoltà si sono presentate già nella fase di **selezione delle feature** sensoriali da utilizzare: quanti sensori considerare? Quali scegliere? Quale angolazione assegnare a ciascun sensore?

Abbiamo adottato un approccio **graduale**, riducendo progressivamente il numero di sensori impiegati. Il miglior compromesso è stato raggiunto utilizzando i **7 sensori di bordo pista** descritti in Sezione 2.2 (2.2), ai quali è stato aggiunto il sensore di **velocità laterale**. Quest'ultimo ha contribuito in modo significativo a migliorare la **classificazione in curva**, riducendo le frequenti sbandate osservate durante i test iniziali.

7.2 Letture incoerenti dei sensori

Tuttavia, durante i test sulla pista *Forza* è emersa una **criticità importante**: i sensori di bordo pista forniscono **letture incoerenti**. Come si può osservare in 15, nonostante il veicolo si trovi al centro della carreggiata, alcuni sensori (es. il sensore 14 a sinistra) rilevano erroneamente un **ostacolo vicino**, come se il veicolo fosse sul bordo. Questo comportamento ha influenzato **negativamente** la qualità dei dati raccolti, rendendo più difficile l'apprendimento del comportamento corretto. Durante i primi test, il veicolo attivava erroneamente la modalità di **recovery** non appena transitava in questa zona della carreggiata, sbandando visibilmente nonostante si trovasse ancora al centro della pista.

Figure 15: Letture incoerenti dei sensori

7.3 Utilizzo di un unico dataset

Un'ulteriore problematica è emersa dall'utilizzo di un **unico dataset** contenente sia dati validi di guida che situazioni di fuori pista. Per affrontare questo problema, si è rivelata più efficace la strategia di **separare i dati** in due dataset distinti, ciascuno associato a un classificatore KNN dedicato: uno per la guida normale e uno per la modalità di recupero.

7.4 Dimensione dei dataset e sbilanciamento

Sono state effettuate numerose prove con dataset di **dimensioni differenti**, osservando che oltre una certa soglia le performance peggioravano sensibilmente. Per questo motivo è stato necessario ricostruire un dataset più **compatto**, ottimizzato in termini di dimensione e bilanciamento (16). Abbiamo infine tentato un **riequilibrio** delle classi, data la prevalenza dell'azione "Accelera", ma gli esperimenti condotti non hanno portato a miglioramenti apprezzabili, anzi tutt'altro!

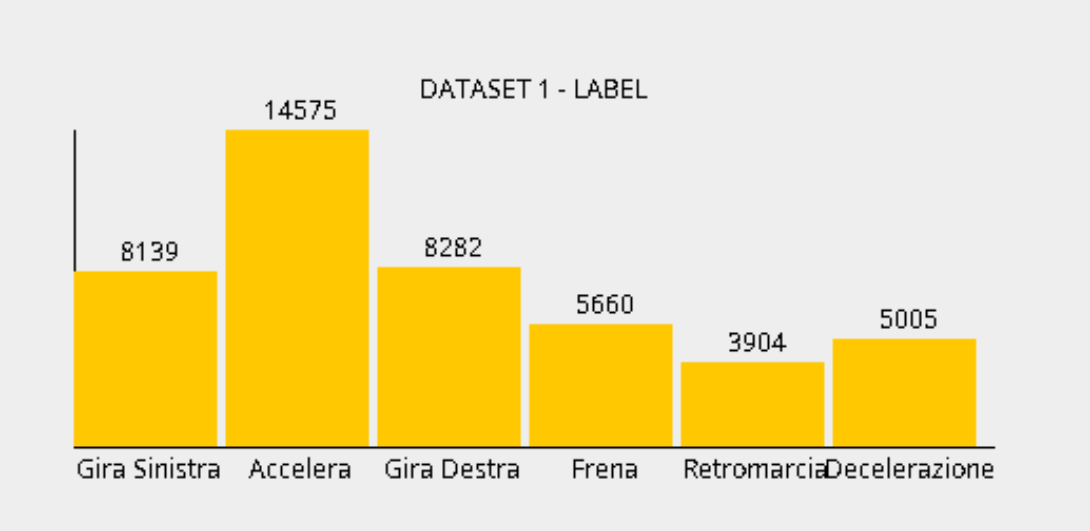


Figure 16: Sbilanciamento del dataset

8 Documentazione tecnica

La documentazione tecnica del codice é stata generata mediante **Javadoc** ed é disponibile presso questo indirizzo: https://emanueletocci.github.io/uni-ai-project/.