Disposizione delle pattuglie - Chicago

Gruppo di lavoro

- Fabio Cirullo, Mat. 758296, f.cirullo@studenti.uniba.it

Repository GitHub: https://github.com/Fabb-24/DisposizionePattuglie_Icon

AA 2023-24

Indice

Introduzione

Il progetto ha come obiettivo principale la previsione del livello di criminalità nelle diverse aree della città di Chicago e l'ottimizzazione della disposizione delle pattuglie di polizia. Partendo da un'analisi dei dati sui crimini, il progetto mira a determinare la distribuzione più efficace delle risorse di polizia. Questo approccio consente di minimizzare il numero di pattuglie necessarie, garantendo al contempo la copertura e la sicurezza di tutte le zone.

Sommario

Il progetto si sviluppa attraverso diverse fasi. In primo luogo, sono stati raccolti e pre-processati i dati utilizzati nei diversi argomenti di interesse. Successivamente, sono stati utilizzati modelli di apprendimento supervisionato per predire il livello di criminalità nelle diverse zone della città. Questi risultati predittivi sono stati poi incorporati in una base di conoscenza sviluppata in Prolog, insieme alla definizione di altri fatti e regole utili alla ricerca della migliore disposizione. Infine, è stato implementato un sistema di ragionamento con vincoli (CSP) per ottimizzare la disposizione delle pattuglie di polizia in base a criteri definiti nella base di conoscenza.

Elenco argomenti di interesse

- Apprendimento supervisionato
- Rappresentazione e ragionamento relazionale
- Ragionamento con vincoli, ottimizzazione

Dati utilizzati

Scelta dei dataset

Per questo progetto, sono stati selezionati due dataset principali: il dataset dei crimini avvenuti a Chicago nel 2018 e il dataset contenente l'elenco delle community areas della città. Entrambi i dataset sono stati ricavati direttamente dal portale di open data di Chicago (Chicago Data Portal), un portale dove sono presenti numerosi dataset sui dati pubblici relativi alla città di Chicago.

Il dataset relativo alle aree di Chicago non è altro che una lista di tutte e 77 le aree di Chicago. Ogni community area viene descritta attraverso le seguenti feature:

- "the_geom": feature di tipo MULTIPOLYGON contenente l'elenco dei punti che danno forma al poligono che rappresenta la community area. I punti sono rappresentati come coppia di coordinate.
- "perimeter", "area", "comarea_", "comarea_id": 4 feature impostate a 0 per ogni riga del dataset.
- "area_numbe", "area_num_1": 2 feature che rappresentano la stessa informazione, ovvero il numero assegnato alla community area.
- "community": feature che rappresenta il nome della community area.
- "shape_area": indica, per ogni area, la sua estensione in metri quadri.
- "shape_len": indica, per ogni area, la lunghezza del poligono dell'area

Il secondo dataset, invece, contiene l'elenco completo dei crimini che si sono verificati a Chicago nel 2018. In particolare, questo dataset si lega con il precedente grazie alla feature che indica in quale area della città il crimine si è verificato. Ogni crimine è descritto dalle seguenti feature:

- "ID": identificatore univoco per il record.
- "Case Number": il numero del dipartimento di polizia, univoco per il crimine.
- "Date": data in cui si è verificato il crimine (vengono indicate sia la data che l'ora).
- "Block": indirizzo parzialmente oscurato in cui si è verificato il crimine.
- "IUCR": codice di segnalazione dei crimini (collegato al Primary Type e a Description).
- "Primary Type": descrizione principale del crimine.
- "Description": descrizione secondaria del crimine.
- "Location Description": descrizione del luogo in cui è avvenuto il crimine.
- "Arrest": indica se è stato effettuato un arresto.
- "Domestic": indica se il crimine è correlato all'ambiente domestico.
- "Beat": area di suddivisione del distretto di polizia.
- "District": distretto di polizia in cui il crimine si è verificato.
- "Ward": il quartiere dove il crimine è avvenuto.
- "Community Area": il numero della community area in cui il crimine è avvenuto.
- "FBI Code": classificazione del crimine secondo il sistema dell'FBI.
- "X Coordinate": coordinata x del luogo del crimine.
- "Y Coordinate": coordinata y del luogo del crimine.
- "Year": anno in cui è avvenuto il crimine.
- "Updated On": data e ora dell'ultimo aggiornamento del record del dataset.
- "Latitude": latitudine del luogo del crimine.
- "Longitude": longitudine del luogo del crimine.
- "Location": luogo del crimine.

Preprocessing dei dati

Per raggiungere lo scopo del progetto è stato necessario effettuare il preprocessing dei dati, fase che include una serie di passaggi per trattare i dati mancanti, modificare dati esistenti, aggregare e dividere colonne del dataset.

Queste operazioni vengono eseguite attraverso la libreria di Python "pandas". Questa libreria permette la creazione di un oggetto DataFrame attraverso la lettura dei dataset.

Prima si è effettuato il preprocessing sul dataset relativo alle aree di Chicago. Una volta creato l'oggetto DataFrame leggendo il dataset "chicagoAreas.csv":

1. Sono state rimosse le feature non utilizzate quali: "perimeter", "area", "comarea_", "comarea_id", "area_numbe" (in quanto è un duplicato di un'altra feature) e "shape_len".
2. Sono state rinominate le colonne rimanenti assegnando nomi più comprensibili (la feature "the_geom" ha assunto il nome di "Perimeter")
3. Si è trasformato il DataFrame in un GeoDataFrame per rendere possibile la lettura della feature "Perimeter" (di tipo MULTIPOLYGON)
4. Sono state ridotte le aree prese in considerazione in quanto, l'uso di tutte le aree di Chicago, avrebbe comportato significative complicazioni dal punto di vista computazionale

Le aree prese in considerazione sono quelle presenti nelle parti di Chicago: north side, northwest side, central, west side, far north side.

```
areas = [5, 6, 7, 21, 22, #north side
         15, 16, 17, 18, 19, 20, #northwest side
         8, 32, 33, #central
         23, 24, 25, 26, 27, 28, 29, 30, 31, #west side
         1, 2, 3, 4, 9, 10, 11, 12, 13, 14, 76, 77 #far north side
         ]
```

Dopo si è effettuato il preprocessing del dataset relativo ai crimini di Chicago. Come prima, è stato creato un DataFrame attraverso la lettura del dataset "chicagoCrimes2018.csv". Successivamente:

1. Sono state eliminate le feature inutilizzate: "ID", "Case Number", "Block", "IUCR", "Description", "Beat", "Location Description", "District", "Ward", "FBI Code", "X Coordinate", "Y Coordinate", "Updated On", "Latitude", "Longitude", "Location".
2. Sono stati ridotti i record del dataset mantenendo solo i crimini che si sono verificati nelle aree prese in considerazione prima.
3. Sono stati rimossi i record che presentavano valori nulli nelle feature rimaste. I valori nulli rappresentano informazioni mancanti che creano problematiche nella fase di addestramento, ma anche nelle fasi successive del preprocessing
4. La feature "Date" è stata sostituita da tre nuove feature: "Week" che indica il numero della settimana nell'anno, "Day" che indica il numero del giorno della settimana e "Time Slot" che indica la fascia oraria nel giorno (la fascia oraria dalle 9 alle 10 è indicata con 9, dalle 10 alle 11 con 10 e così via…)
5. Sono stati mappati tutti i tipi di crimine contenuti nella colonna "Primary Type" in 3 livelli di criminalità (0, 1, 2, dove il 2 è quello più alto): a ogni tipo di crimine è stato associato un livello di criminalità. Quindi la feature "Primary Type" è stata sostituita dalla nuova feature "Crime Severity"

6. Le feature “Domestic” e “Arrest” sono state convertite in intero, facendo corrispondere al false 0 e al true 2. In questo modo è stato possibile ottenere la nuova feature “Severity” che corrisponde alla media tra le feature “Crime Severity”, “Arrest” e “Domestic” in modo tale che al livello di criminalità partecipassero anche le feature “Domestic” e “Arrest”
7. Infine, sono state rimosse le feature ormai utilizzate “Crime Severity”, “Domestic” e “Arrest”.

La suddivisione dei tipi di crimine nei 3 livelli di criminalità è stata effettuata in questo modo:

```
high_severity = ['homicide', 'kidnapping', 'criminal sexual assault', 'arson', 'robbery', 'sex offense', 'human trafficking']
medium_severity = ['theft', 'assault', 'battery', 'burglary', 'weapons violation', 'narcotics', 'motor vehicle theft',
                   'criminal damage', 'offense involving children', 'prostitution', 'stalking', 'crim sexual assault']
low_severity = ['theft', 'other offense', 'deceptive practice', 'criminal trespass', 'interference with public officer',
                'public peace violation', 'gambling', 'intimidation', 'obscenity', 'non-criminal', 'liquor law violation',
                'public indecency', 'ritualism', 'other narcotic violation', 'concealed carry license violation']
```

dove “high_severity” corrisponde al livello di criminalità 2, “medium_severity” al livello 1 e “low_severity” al livello 0.

Apprendimento supervisionato

Fasi dell'apprendimento

L'apprendimento supervisionato è stato utilizzato per predire il livello di criminalità nelle diverse aree di Chicago. Si è scelto un task di classificazione piuttosto che di regressione per l'apprendimento in quanto la feature target, ovvero la feature di cui predire il valore (in questo caso il livello di criminalità dell'area), possiede un dominio di valori finito, quindi valori discreti.

Il task di apprendimento supervisionato viene completato attraverso l'esecuzione di più fasi:

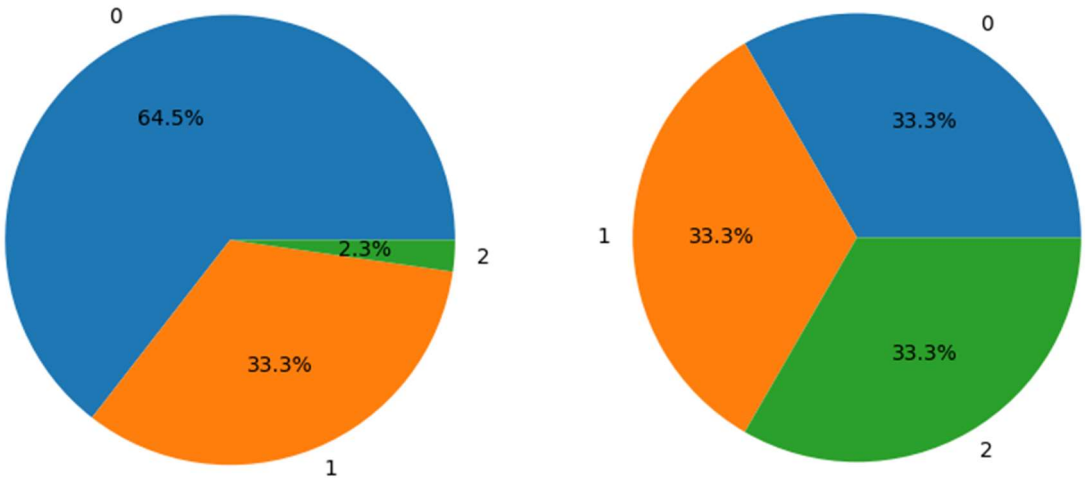
1. Oversampling
2. Ricerca dei migliori parametri
3. Addestramento
4. Valutazione

Oversampling

Il primo passo è stato l'oversampling delle classi minoritarie nel dataset. I dati sui crimini presentavano uno sbilanciamento significativo, con alcune categorie di crimini che si verificavano molto più frequentemente di altre. Per affrontare questo problema e migliorare la capacità del modello di predire correttamente le classi meno rappresentate, è stato applicato l'oversampling. Questa tecnica ha permesso di aumentare il numero di esempi nelle classi minoritarie, bilanciando il dataset e migliorando le performance del modello.

Per l'oversampling in Python è stata utilizzata la classe SMOTE della libreria "imblearn".

Di seguito sono mostrati i rapporti delle categorie di crimini nel dataset prima e dopo l'oversampling:



Ricerca dei migliori parametri

Prima di addestrare i modelli scelti occorre definire quali iperparametri essi debbano utilizzare. Gli iperparametri sono i parametri che ogni modello necessita per poter essere utilizzato e devono essere impostati prima della fase di addestramento.

Per ogni modello sono stati definiti quali iperparametri dovevano essere impostati per l'addestramento e per ogni iperparametro definito sono stati elencati i possibili valori che essi possono assumere. Lo scopo di questa ricerca è trovare la migliore combinazione dei valori per gli iperparametri sulla base dei dati utilizzati per poter creare un modello addestrato il più affidabile possibile.

Per ottenere la migliore combinazione di iperparametri per ogni modello, è stata utilizzata una ricerca su griglia. Questo tipo di ricerca consiste nella disposizione dei valori degli iperparametri su una griglia che permette di esaminare tutte le combinazioni possibili dei valori degli iperparametri definiti. La ricerca su griglia è stata implementata grazie alla classe "GridSearchCV" della libreria "scikit-learn" di Python.

```
def bestParams(self, X_train, y_train):
    """
    Funzione che restituisce i migliori parametri per i modelli.
    Utilizza la tecnica GridSearchCV per la ricerca dei migliori parametri

    Parametri:
        X_train (DataFrame): il dataset di training senza il target
        y_train (DataFrame): il target del dataset di training
    """

    values = {}
    for model_name, model in self.empty_models.items():
        print(f"\nSearching best params for {model_name} model...")
        grid_search = GridSearchCV(estimator=model, param_grid=self.param_grids[model_name],
                                   cv=5, n_jobs=-1, verbose=2, scoring='f1_macro', refit='f1_macro')
        pipeline = Pipeline([
            ('preprocessor', self.preprocessor),
            ('model', grid_search)
        ])
        pipeline.fit(X_train, y_train)
        values[model_name] = grid_search.best_params_
    return values
```

Anche per la scelta dei migliori iperparametri si può definire la cross validation da utilizzare (il parametro "cv") che in questo caso è stata impostata a 5 (quindi 5 fold e 5 iterazioni).

Si è scelto di salvare migliori parametri trovati in un file per dare la possibilità di eseguire nuovamente l'addestramento senza dover ricercare i migliori parametri, qualora i dati e i modelli utilizzati fossero gli stessi.

Addestramento

Nella fase di addestramento sono stati addestrati i modelli scelti con gli iperparametri trovati nella fase precedente sui dati di training.

Un problema ricorrente è quello di overfitting che si verifica quando il modello tende ad adattarsi eccessivamente ai dati di addestramento. Questo problema porta a una elevata performance sui dati di addestramento e a una scarsa performance sui dati di test e validazione.

Per ridurre il problema di overfitting è stata utilizzata la tecnica di k-fold cross-validation: questa tecnica suddivide il dataset in k parti (fold) di dimensioni simili. Il processo prevede poi una serie di iterazioni, dove in ognuna di esse, un fold viene usato come set di test e i restanti fold come set di addestramento. L'utilizzo di questa tecnica in Python è reso possibile dalla classe "RepeatedKFold" della libreria "scikit-learn".

```
cv = RepeatedKFold(n_splits=5, n_repeats=5)
```

Si è scelto di salvare il modello addestrato in un file .pkl per poter effettuare predizioni con il modello senza doverlo addestrare ogni volta.

Valutazione

Per valutare il modello sono state scelte più metriche di valutazione quali:

- **Accuracy**: l'accuratezza rappresenta la percentuale di predizioni corrette rispetto al totale delle predizioni effettuate dal modello.
- **Precision**: misura la proporzione di predizioni positive corrette tra tutte le predizioni positive fatte dal modello
- **Recall**: indica la proporzione di istanze positive correttamente identificate dal modello rispetto a tutte le istanze positive effettivamente presenti nel dataset.
- **F1**: il punteggio F1 è una media armonica di precision e recall

La valutazione dei modelli in Python avviene tramite la funzione "cross_val_score" della libreria "scikit-learn":

```
self.scoring = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
```

```
# Addestramento dei modelli: valutazione e salvataggio su file
for model_name, model in models.items():
    res[model_name] = {}

    print(f"\n\nTraining {model_name} model...")
    pipeline = ImbPipeline([
        ('preprocessor', self.preprocessor),
        ('model', model)
    ])

    # Addestramento e valutazione del modello tramite k-fold cross-validation
    for score in self.scoring:
        scores = cross_val_score(pipeline, X, y, cv=cv, scoring=score, n_jobs=-2)
        mean_score = np.mean(scores)
        res[model_name][score.split("_")[0]] = mean_score
    pipeline.fit(X_train, y_train)
```

Per valutare i modelli addestrati sono anche stati generati dei grafici che mostrano l'andamento dell'accuratezza del modello all'aumentare degli esempi utilizzati, sia per il set di training che per il set di test. La generazione e il salvataggio dei grafici è stata eseguita in Python con la libreria "matplotlib".

```python
def learningCurve(self, model, X, y, model_name, savePath):
    """
    Funzione che genera la learning curve per il modello passato come parametro.
    Salva il grafico su file

    Parametri:
        model (Model): il modello per cui generare la learning curve
        X (DataFrame): il dataset senza il target
        y (DataFrame): il target del dataset
        model_name (String): il nome del modello
    """

    # Generazione della learning curve
    train_sizes, train_scores, test_scores = learning_curve(
        estimator=model,
        X=X,
        y=y,
        cv=5,
        n_jobs=-1,
        train_sizes=np.linspace(0.1, 1.0, 10),
        scoring='accuracy'
    )

    # Calcolo delle medie e delle deviazioni standard dei punteggi di training e test
    train_scores_mean = np.mean(train_scores, axis=1)
    test_scores_mean = np.mean(test_scores, axis=1)

    # Creazione del grafico della learning curve
    plt.figure()
    plt.title(f"Learning Curve for {model_name}")
    plt.xlabel("Training examples")
    plt.ylabel("Score")
    plt.ylim((0.0, 1.1))
    plt.grid()
    plt.plot(train_sizes, train_scores_mean, 'o-', color="red", label="Training score")
    plt.plot(train_sizes, test_scores_mean, 'o-', color="green", label="Test score")
    plt.legend(loc="best")
```

Modelli utilizzati

Sono stati utilizzati e confrontati tre differenti modelli di apprendimento supervisionato. Questo approccio è stato adottato per valutare quale modello fosse più adatto in questo specifico contesto.

I modelli utilizzati sono:

- **Decision Tree**: è un modello di apprendimento supervisionato che crea una struttura ad albero dove i nodi interni sono nodi decisionali basati sulle feature di input e le foglie rappresentano le categorie della feature target di appartenenza.
- **Random Forest**: è un modello di apprendimento supervisionato che crea più alberi decisionali indipendenti, ognuno addestrato su una parte del dataset. La predizione finale viene formata combinando le previsioni dei singoli decision tree.
- **AdaBoost**: (Adaptive Boosting) è un modello di boosting (modello che impara dagli errori precedenti) che genera una sequenza di modelli deboli e combina le loro predizioni assegnando pesi a ognuna di esse in base alla loro performance.

Di seguito vengono indicati i risultati ottenuti durante l'apprendimento, partendo dai migliori parametri selezionati e specificando i valori delle metriche di valutazione.

Decision Tree

Gli iperparametri utilizzati per il modello Decision Tree sono i seguenti:

```
'Decision Tree': {'criterion': ['gini', 'entropy'],
                  'max_depth': [10, 20, 30, 40],
                  'min_samples_split': [2, 5, 10, 20],
                  'min_samples_leaf': [1, 2, 5, 10]},
```

- **criterion**: è la funzione utilizzata per misurare la qualità delle suddivisioni nei nodi interni dell'albero. Il valore "gini" indica l'utilizzo dell'indice di Gini che misura l'impurità di un nodo (un nodo è puro se contiene campioni di una sola classe). Il valore "entropy" indica l'utilizzo dell'entropia dell'informazione, che misura il grado di disordine o incertezza di un nodo (l'entropia è 0 se il nodo è puro)
- **max_depth**: indica la profondità massima dell'albero. Limitando la profondità dell'albero si può prevenire l'overfitting in quanto un albero troppo profondo può modellare troppo i dettagli del training set
- **min_samples_split**: indica il numero minimo di campioni richiesti per dividere un nodo. Un valore troppo alto renderebbe l'albero meno complesso e più generalizzabile
- **min_samples_leaf**: indica il numero minimo di campioni che devono essere presenti in una foglia dell'albero. Nodi foglia con pochi campioni potrebbero essere molto specifici ai dati di addestramento.

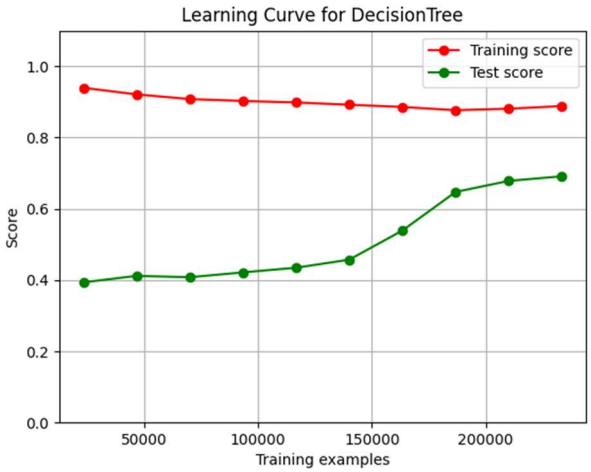
La ricerca dei migliori parametri per questo modello ha portato al seguente risultato:

```
"Decision Tree": {
    "criterion": "gini",
    "max_depth": 40,
    "min_samples_leaf": 1,
    "min_samples_split": 2
},
```

È possibile notare come siano stati scelti dei valori bassi per gli iperparametri “min_samples_split” e “min_samples_leaf”: questo potrebbe portare a situazioni di overfitting ai dati di addestramento.

Per quanto riguarda le metriche di valutazione del modello scelte, i risultati ottenuti sono:

```
Modello: DecisionTree
        accuracy: 0.7219906383549938
        precision: 0.7161840269548024
        recall: 0.7219492184139561
        f1: 0.7179189002473416
```



Random Forest

Gli iperparametri utilizzati per il modello Random Forest sono i seguenti:

```
'Random Forest': {'criterion': ['gini', 'entropy'],
                  'n_estimators': [100, 200],
                  'max_depth': [10, 20],
                  'min_samples_split': [2, 5, 10],
                  'min_samples_leaf': [1, 2, 5],
                  'bootstrap': [True, False]},
```

Gli iperparametri “criterion”, “max_depth”, “min_samples_split”, “min_samples_leaf” sono identici a quelli descritti per il modello Decision tree e si applicano ai singoli alberi del Random Forest.

- **n_estimators**: indica il numero di alberi nella random forest. Un numero alto di alberi può migliorare la performance, ma aumenta di molto il tempo di addestramento
- **bootstrap**: indica l’utilizzo del campionamento con ripetizione durante la creazione degli alberi. Se impostato a true, ogni albero viene addestrato su un campione bootstrap del dataset originale (alcuni esempi nel dataset possono essere selezionati più volte mentre altri possono essere esclusi). Se impostato a false, ogni albero viene addestrato sull’intero dataset senza campionamento.

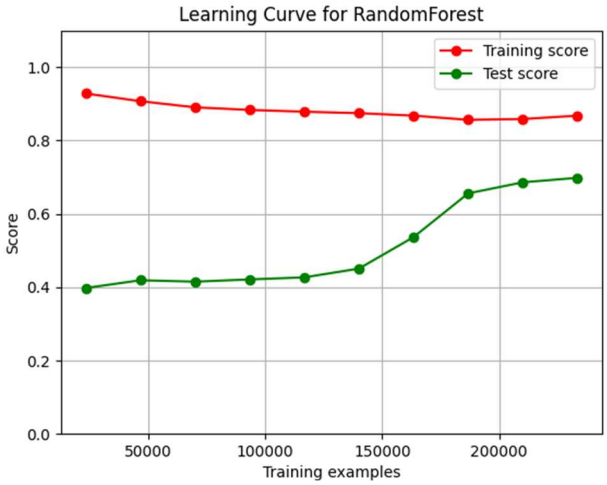
La ricerca dei migliori parametri per questo modello ha portato al seguente risultato:

```
"Random Forest": {
    "bootstrap": false,
    "criterion": "entropy",
    "max_depth": 20,
    "min_samples_leaf": 1,
    "min_samples_split": 5,
    "n_estimators": 200
},
```

Anche qui, come nel Decision Tree, “min_samples_leaf” ha assunto un valore basso.

Per quanto riguarda le metriche di valutazione del modello scelte, i risultati ottenuti sono:

```
Modello: RandomForest
        accuracy: 0.7360581782840073
        precision: 0.7304308870361207
        recall: 0.7359415303780775
        f1: 0.7322046096901202
```



Rispetto agli altri due modelli, il Random Forest è quello che ha impiegato più tempo per la fase di addestramento e valutazione.

AdaBoost

Gli iperparametri utilizzati per il modello AdaBoost sono i seguenti:

```
'AdaBoost': {'n_estimators': [50, 100, 200],
             'learning_rate': [0.01, 0.1, 1, 10],
             'algorithm': ['SAMME']}
```

- **n_estimators**: indica il numero massimo di stimatori da addestrare. Un valore alto aumenta la capacità del modello di correggere errori, ma può aumentare il rischio di overfitting e il tempo di addestramento
- **learning_rate**: indica una riduzione della considerazione di ciascuno stimatore. Questo iperparametro controlla il contributo di ciascun classificatore debole. Un learning rate più alto velocizza la fase di apprendimento ma può saltare buoni minimi locali, mentre un valore più basso rende l’apprendimento più graduale

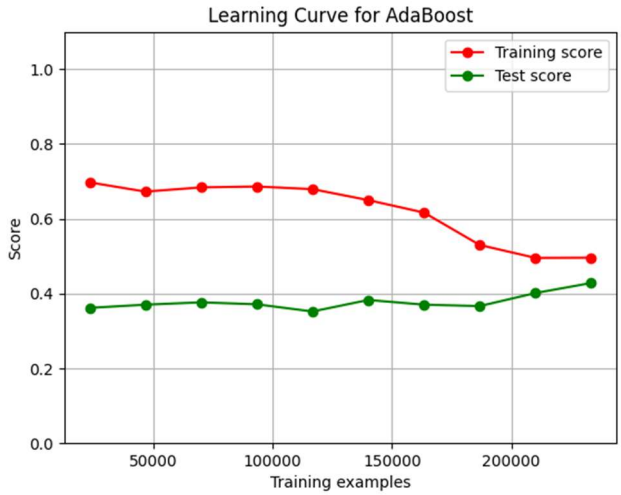
- **algorithm**: indica l'algoritmo di boosting da utilizzare. In questo caso viene utilizzato solo il SAMME che è una variante dell'AdaBoost.

La ricerca dei migliori parametri per questo modello ha portato al seguente risultato:

```
"AdaBoost": {
    "algorithm": "SAMME",
    "learning_rate": 1,
    "n_estimators": 200
}
```

Per quanto riguarda le metriche di valutazione del modello scelte, i risultati ottenuti sono:

```
Modello: AdaBoost
        accuracy: 0.4898734613292056
        precision: 0.48891339855817967
        recall: 0.48991311387175024
        f1: 0.4881815060612616
```



Confronto

I modelli Decision Tree e Random Forest presentano dei valori simili: il primo mostra delle metriche che si aggirano intorno al 72%, mentre il secondo, leggermente superiore, mostra valori intorno al 73%. Anche le curve di apprendimento sono molto simili ed entrambe mostrano una differenza finale di quasi 0.2 tra la curva relativa ai dati di addestramento e quella relativa ai dati di test: questo potrebbe indicare un leggero problema di overfitting in entrambi i casi, forse dato anche dagli iperparametri selezionati.

Il terzo modello, l'AdaBoost, invece mostra valori nettamente più bassi, intorno al 49% per le metriche di valutazione utilizzate. Nonostante le curve mostrino valori finali molti vicini, questo modello non è indicato per le predizioni nel contesto del progetto individuato.

In conclusione, dal confronto dei 3 modelli, emerge che il Random Forest ha ottenuto i migliori risultati: conviene quindi utilizzare questo modello per effettuare le predizioni dei livelli di criminalità nelle successive fasi di progetto.

Knowledge Base

Una knowledge base è un insieme di fatti e regole che descrivono le proprietà degli individui e le interazioni tra essi. Utilizzando regole logiche, una knowledge base consente il ragionamento deduttivo e può essere interrogata tramite un programma per ottenere informazioni e inferenze. La KB del progetto è stata implementata in Prolog, un linguaggio basato sul ragionamento logico.

La libreria Python utilizzata per la gestione della base di conoscenza in prolog è "pyswip". La KB è realizzata in un file .pl e aggiornata durante l'esecuzione del programma data la presenza di fatti dinamici.

Fatti e Regole

Come prima cosa si sono definiti fatti e regole della Knowledge Base nell'apposito file "kb.pl".

Nella KB è presente un solo individuo: esso rappresenta l'area di Chicago ed è nella forma *area(A)* dove A è la variabile numerica che rappresenta il numero assegnato all'area in questione. È stato dichiarato come dynamic nella KB in quanto tutte le aree vengono definite nel corso del programma.

```
% individui
:- dynamic(area/1).
```

Nell'apposita classe creata per la gestione e l'interrogazione della KB (PrologKB.py) è stata implementata la funzione per l'aggiunta delle aree.

```
def defineAreas(self):
    """
    Metodo che definisce i fatti area per le aree di Chicago, della forma area(AreaNumber)
    """

    for area in self.areasGdf.itertuples():
        self.prolog.assertz(f"area({area.AreaNumber})")
```

L'individuo area presenta 3 proprietà diverse quali: la gravità, le aree vicine e la dimensione. Tutte e 3 le proprietà sono definite come dynamic.

```
% proprietà individui
:- dynamic(severity/2).
:- dynamic(nearAreas/2).
:- dynamic(size/2).
```

- "severity" ha 2 attributi ed è nella forma *severity(area(A), Sev)*. Questa proprietà indica il livello di criminalità (Sev) dell'area A. Questa proprietà è dinamica poiché il livello di gravità viene predetto grazie al modello di apprendimento supervisionato generato nella fase precedente.

- “nearAreas” ha 2 attributi ed è nella forma *nearAreas(area(A), L)*, dove L è una lista di interi e ogni elemento della lista rappresenta il numero di un’area che tocca fisicamente l’area A. Anche questa proprietà è dinamica in quanto la vicinanza tra le aree è ricavata dal dataset relativo alle aree di Chicago: viene utilizzato il campo “Perimeter” del dataset per trovare tutte le aree adiacenti.
- “size” ha 2 attributi ed è nella forma *size(area(A), S)*, dove S è la grandezza dell’area A. Anche la grandezza delle aree viene ricavata dal dataset delle aree di Chicago.

```
def defineAreaSeverities(self):
    """
    Metodo che definisce i fatti areaSeverity per le aree di Chicago, della forma severity(area(A), Sev).
    La gravità dei crimini è prevista utilizzando il modello di machine learning addestrato
    """

    loadedModel = joblib.load(self.modelPath)
    for area in self.areasGdf.itertuples():
        data = {
            'Week': self.week,
            'Day': self.day,
            'Time Slot': self.timeslot,
            'Community Area': area.AreaNumber
        }
        prediction = loadedModel.predict(pd.DataFrame(data, index=[0]))
        self.prolog.assertz(f"severity(area({area.AreaNumber}), {int(prediction[0])})")
```

```
def defineNearAreas(self):
    """
    Metodo che definisce i fatti nearAreas per le aree vicine, della forma nearAreas(area(A), L).
    Due aree sono considerate vicine se hanno due punti del loro perimetro in comune
    """

    i = 0
    y = 0
    for area1 in self.areasGdf.itertuples():
        perimeter1 = MultiPolygon(area1.Perimeter)
        y = 0
        list = []
        for area2 in self.areasGdf.itertuples():
            if i != y:
                perimeter2 = MultiPolygon(area2.Perimeter)
                if perimeter1.touches(perimeter2):
                    list.append(area2.AreaNumber)
            y += 1
        self.prolog.assertz(f"nearAreas(area({area1.AreaNumber}), {list})")
        i += 1
```

```
def defineAreasSize(self):
    """
    Metodo che definisce i fatti size per le aree di Chicago, della forma size(area(A), S).
    Rappresentano la grandezza delle zone di Chicago
    """
    for area in self.areasGdf.itertuples():
        self.prolog.assertz(f"size(area({area.AreaNumber}), {area.AreaSize})")
```

L'ultimo fatto definito nella KB è quello relativo al pattugliamento delle aree. Il fatto *patrolArea(area(A), Val)* indica se l'area è pattugliata (Val assume valore true) o non è pattugliata (Val assume valore false).

```
% fatti
:- dynamic(patrolArea/2).
```

Anche questo fatto è dichiarato dynamic in quanto viene definito e rimosso durante l'esecuzione del ragionamento con vincoli nella parte successiva.

```
def setAreaPatrol(self, areaNum, patrol):
    """
    Metodo che imposta l'area come pattugliata.
    Aggiunge il fatto patrolArea per l'area specificata, della forma patrolArea(area(A))

    Parametri:
        areaNum (Int): il numero dell'area da impostare come pattugliata
        patrol (Bool): True se l'area è pattugliata, False altrimenti
    """

    # controllo se esiste già un fatto patrolArea per l'area specificata
    res = list(self.prolog.query(f"patrolArea(area({areaNum}), _)"))
    if res:
        self.prolog.retract(f"patrolArea(area({areaNum}), _)")
    self.prolog.assertz(f"patrolArea(area({areaNum}), {'true' if patrol else 'false'})")
```

```
def removeAreaPatrol(self, areaNum):
    """
    Metodo che rimuove l'area dalla lista delle aree pattugliate.
    Rimuove il fatto patrolArea per l'area specificata

    Parametri:
        areaNum (Int): il numero dell'area da rimuovere dalla lista delle aree pattugliate
    """
    if list(self.prolog.query(f"patrolArea(area({areaNum}), _)")):
        self.prolog.retract(f"patrolArea(area({areaNum}), _)")
```

Le regole definite, invece, sono le seguenti:

Regola che indica se due aree sono vicine

```
isNear(area(A1), area(A2)) :-
    nearAreas(area(A1), NearAreasList),
    member(A2, NearAreasList).
```

Regole che restituiscono la dimensione minore e maggiore tra tutte le aree

```
minSize(S) :-
    findall(X, size(_, X), L),
    min_list(L, S).

maxSize(S) :-
    findall(X, size(_, X), L),
    max_list(L, S).
```

Regole per la determinazione della gravità di un'area anche in base alla sua dimensione. Se l'area è superiore alla dimensione media viene aggiunto un grado di gravità

```
adjustedSeverity(area(A), Sev) :-
    severity(area(A), Sev1),
    Sev1 =< 1, size(area(A), Size),
    minSize(Min),
    maxSize(Max),
    Size > (Min + Max) / 2,
    Sev is Sev1 + 1.

adjustedSeverity(area(A), Sev) :-
    severity(area(A), Sev1),
    Sev1 is 2,
    Sev is Sev1.

adjustedSeverity(area(A), Sev) :-
    severity(area(A), Sev1),
    size(area(A), Size),
    minSize(Min),
    maxSize(Max),
    Size =< (Min + Max) / 2,
    Sev is Sev1.
```

Regole che determinano la distanza tra due aree (0, 1 o 2). 0 se le due aree rappresentano la stessa area, 1 se si toccano fisicamente, 2 se non sono vicine e tra loro c'è un'altra area

```
distance(area(A1), area(A2), 0) :-
    A1 = A2.

distance(area(A1), area(A2), 1) :-
    isNear(area(A1), area(A2)).

distance(area(A1), area(A2), 2) :-
    isNear(area(A1), area(X)),
    isNear(area(X), area(A2)),
    not(isNear(area(A1), area(A2))),
    not(distance(area(A1), area(A2), 0)).
```

Regola che determina tutte le aree entro una certa distanza da una determinata area. Per esempio, se D è 2, le aree A2 interessate saranno tutte le aree a distanza 0, 1 o 2

```
maxDistance(area(A1), area(A2), D) :-
    distance(area(A1), area(A2), D1),
    D1 =< D.
```

Regola che stabilisce se un'area è coperta da pattuglia o no secondo criteri sotto descritti

```
isSafe(area(A)) :-
    adjustedSeverity(area(A), S),
    maxDistance(area(A), area(AreaP), 2 - S),
    patrolArea(area(AreaP), true).
```

Regola che indica se un'area è considerabile, ovvero se si può determinare se è coperta da pattuglia o no

```
isConsiderable(area(A)) :-
    adjustedSeverity(area(A), S),
    forall(maxDistance(area(A), area(A1), 2 - S),
        patrolArea(area(A1), _)).
```

Nelle regole sono state utilizzate delle regole già presenti in prolog:

- **findall/3**: utilizzata per ottenere la lista di tutte le soluzioni di un predicato. Il primo argomento è la forma delle soluzioni da ricavare, il secondo argomento è il predicato o obiettivo per cui si stanno cercando le soluzioni e il terzo argomento è la lista in cui vengono memorizzate tutte le soluzioni trovate
- **member/2**: utilizzata per verificare se un elemento è presente in una lista. Il primo argomento è l'elemento di cui si vuole verificare la presenza, mentre il secondo argomento è la lista in cui deve essere cercato l'elemento
- **min_list/2**: utilizzata per trovare il valore minimo in una lista di numeri. Il primo argomento è la lista in cui cercare il valore minimo e il secondo argomento è la variabile che conterrà il valore minimo
- **max_list/2**: simile a min_list ma ricerca il valore maggiore invece che quello minore
- **forall/2**: utilizzata per verificare se una certa regola è vera per tutti gli elementi di una lista. Il primo argomento è la regola che deve essere vera per tutti gli elementi e il secondo argomento è l'azione da compiere per ciascun elemento

Per quanto riguarda le ultime due regole ("isConsiderable" e "isSafe"):

- un'area può essere considerata (ovvero, è possibile determinare se è coperta da pattuglia) quando per tutte le aree a distanza D da essa è definito il fatto patrolArea. La distanza D è pari a 2 – S, dove S è il grado di criminalità dell'area. In sostanza, un'area con severità 0 è considerabile se il fatto patrolArea è stato definito per tutte le aree a una distanza massima di 2 da essa, se l'area ha severità 1 la distanza massima è 1, altrimenti è 0.
- Un'area è coperta da pattuglia secondo un preciso criterio: se l'area ha severità 0, essa è coperta se è presente almeno una pattuglia in tutte le zone a una distanza massima di 2. Se l'area ha severità 1 è coperta se è presente almeno una pattuglia in tutte le aree adiacenti ad essa. Se l'area ha severità 2, è coperta se in essa è presente una pattuglia.

Query

Le interrogazioni in una KB permettono di ricavare informazioni da essa. Queste sono effettuate in Python attraverso la funzione "query" della libreria sopracitata. Ogni query viene effettuata in una funzione diversa che converte il risultato della query in una forma più comoda.

Sono state utilizzate sia query decisionali, sia query con variabili. In particolare, sono state formulate query per sapere se un'area è coperta da pattuglia, ottenere la lista di tutte le aree, ottenere la lista delle aree a una certa distanza da una specifica area, ottenere il grado di criminalità di un'area e ottenere la lista delle aree considerabili.

Ragionamento con vincoli

Per raggiungere l'obiettivo principale del progetto, ovvero trovare la miglior disposizione delle pattuglie nelle aree di Chicago, è stato utilizzato un problema di soddisfacimento di vincoli (CSP, Constraint Satisfaction Problem) di ottimizzazione. Lo scopo è quindi trovare la lista delle aree di Chicago a cui assegnare una pattuglia in modo che tutte le aree considerate siano coperte secondo i criteri definiti nella base di conoscenza. Sono state implementate anche funzioni di costo ed euristica per guidare la ricerca della soluzione ottimale.

Algoritmo di ottimizzazione

Un CSP è un problema che consiste in un insieme di variabili e vincoli. Il problema ha lo scopo di trovare una assegnazione totale (deve essere assegnato un valore a tutte le variabili utilizzate) che rispetti i vincoli definiti. Nel caso di un problema di ottimizzazione, l'obiettivo diventa quello di trovare la migliore assegnazione totale che soddisfi i vincoli definiti.

In Python è stato definito l'algoritmo di ricerca branch-and-bound per un problema di ottimizzazione

```
def cbsearch(self, CVs, CCs, context):
    """
    Metodo ricorsivo della ricerca branch-and-bound per risolvere il problema di CSP

    Attributes:
        CVs (list): Lista delle variabili rimanenti da assegnare
        CCs (list): Lista dei vincoli rimanenti da soddisfare
        context (dict): Il contesto corrente
    """

    can_eval = self.evalCs(CCs, context)
    rem_Cs = CCs.copy()
    rem_Cs = removeElements(rem_Cs, can_eval)
    cost_context = self.cost(context, can_eval)

    if cost_context + self.h(rem_Cs) < self.bound:
        if not CVs:
            self.best_asst = context
            self.bound = cost_context
        else:
            var = self.selectVariable(CVs, context)
            for val in self.Ds[var]:
                CVs2 = CVs.copy()
                CVs2.remove(var)
                context2 = context.copy()
                context2[var] = val
                self.cbsearch(CVs2, rem_Cs, context2)
```

Nell'algoritmo sono presenti funzioni che vanno definite:

- la funzione "cost" rappresenta il costo del contesto corrente (assegnazione corrente)

- la funzione “h” rappresenta la funzione euristica che stima il costo della parte di assegnazione rimanente
- la funzione “evalCs” trova tutti i vincoli non ancora soddisfatti che possono essere considerati (è possibile quindi capire se sono stati violati o soddisfatti)
- la funzione “selectVariable” restituisce la successiva variabile da assegnare e includere nel contesto

La funzione “selectVariable” può anche non essere definita e usata quella di default che si limita a restituire la prima variabile della lista di variabili ancora da assegnare.

La migliore assegnazione trovata viene quindi restituita come output dell’algoritmo.

```
def solve(self):
    """
    Funzione di risoluzione del problema di CSP.
    Restituisce la migliore assegnazione delle variabili del problema.

    Returns:
        Dict: La migliore assegnazione delle variabili del problema
    """

    print("Inizio risoluzione CSP...")
    self.cbsearch(self.Vs, self.Cs, {})

    return self.best_asst
```

Decisioni di Progetto

Variabili e Vincoli

Come detto precedentemente, un CSP è un problema che consiste in un insieme di variabili e un insieme di vincoli. Le variabili scelte in questo caso corrispondono alle aree di Chicago prese in considerazione. Il dominio di tutte le variabili è uguale e corrisponde a due valori: [true, false]. Se ad una variabile è assegnato il valore true vuol dire che all’area corrispondente alla variabile è stata assegnata una pattuglia, altrimenti se il valore assegnato è false, all’area in questione non viene assegnata la pattuglia.

I vincoli del CSP vengono indicati sempre attraverso l’area che li riguarda. In particolare, esiste un solo tipo di vincolo, definito precedentemente nella base di conoscenza, che viene soddisfatto se l’area è coperta da pattuglia o violato in caso contrario. Quindi, quando un vincolo può essere valutato, occorre verificare se l’area oggetto del vincolo è coperta da pattuglia o no.

```python
def findBestArrangement(self, bound=float('inf')):
    """
    Funzione che risolve il problema di ottimizzazione tramite CSP e restituisce la miglior disposizoine delle pattuglie

    Parametri:
        kb (KB): la knowledge base da utilizzare
        bound (Int): il bound iniziale

    Returns:
        Dict: La migliore disposizione delle pattuglie
    """

    areaList = self.kb.getAreasList()

    dm = {}
    for area in areaList:
        dm[area] = [False, True]

    ocsp = optimizationCsp(variables=areaList,
                           constraints=areaList,
                           domains=dm,
                           cost_function=self.cost,
                           heuristic_function=self.h,
                           evaluableConstraints_function=self.evaluableAreas,
                           selectVariable_function=self.selectVariable,
                           bound=bound
    )

    return ocsp.solve()
```

Vincoli valutabili

Ad ogni iterazione del CSP occorre trovare tutti i vincoli non ancora valutati e che sono valutabili nel contesto corrente. Per fare ciò è stata definita una funzione che, dopo aver aggiornato la base di conoscenza per adattarla al contesto corrente, interroga la KB per ottenere la lista delle aree valutabili attraverso l'apposita regola descritta in precedenza.

```python
def evaluableAreas(self, CCs, context):
    """
    Funzione che restituisce la lista delle aree valutabili (di cui si possono valutare i vincoli)

    Parametri:
        CCs (List): la lista dei vincoli
        context (Dict): il contesto corrente

    Returns:
        List: La lista delle aree valutabili
    """

    keys = list(context.keys())
    areas = self.kb.getAreasList()

    for area in keys:
        self.kb.setAreaPatrol(area, context[area])

    for area in areas:
        if area not in keys:
            self.kb.removeAreaPatrol(area)

    can_eval = list(dict.fromkeys(self.kb.evaluableAreas(CCs)))

    return can_eval
```

Costo ed Euristica

Durante le iterazioni vengono valutati il costo del contesto corrente e la sua euristica, utili a trovare la miglior soluzione del problema. La base del costo di una assegnazione è il numero di pattuglie assegnate: l'obiettivo è assegnare il minor numero di pattuglie e assegnare pattuglie nelle zone con un livello di criminalità più alto.

Se l'assegnazione corrente viola un vincolo valutabile allora il costo di quell'assegnazione sarà pari a infinito in quanto quella assegnazione non è ammissibile. Se non viola nessun vincolo, il costo corrisponde al numero di pattuglie disposte dove ogni pattuglia ha un peso: se la pattuglia è assegnata a una zona con livello di criminalità 2 ha peso 1, se il livello di criminalità della zona è 1 la pattuglia ha peso 2, altrimenti con un livello di criminalità pari a 0, il peso della pattuglia corrisponde a 3. Con questa funzione di costo viene assegnato un costo più basso a quelle disposizioni che presentano poche pattuglie in zone con criminalità alta, se non violano nessun vincolo.

```
def cost(self, context, Cs):
    """
    Metodo che calcola il costo del contesto specificato in base ai vincoli che possono essere valutati.
    Se un vincolo non è soddisfatto, il costo del contesto è pari a infinito.
    Il costo è dato dal numero di pattuglie assegnate dove ogni pattuglia ha un peso diverso in base alla gravità dell'area

    Parametri:
        context (Dict): il contesto corrente
        Cs (List): la lista dei vincoli valutabili

    Returns:
        Float: Il costo del contesto specificato
    """

    for c in Cs:
        if not self.kb.isAreaSafe(c):
            return float('inf')

    areas = context.keys()
    cost = 0
    for area in areas:
        if context[area]:
            if self.kb.getAreaSeverity(area) == 2:
                cost += 1
            elif self.kb.getAreaSeverity(area) == 1:
                cost += 2
            else:
                cost += 3
    return cost
```

La funzione euristica deve essere una stima del costo dall'assegnazione corrente all'assegnazione totale e deve essere sempre minore o uguale al costo effettivo di quella parte di assegnazione rimanente. Per questo motivo l'euristica in questo caso corrisponde al numero di aree con criminalità pari a 2 che non sono state ancora valutate. E' possibile definire una euristica del genere poiché, per non violare vincoli, ad un'area con livello 2 deve essere assegnata una pattuglia (che avrà peso 1).

```python
def h(self, Cs):
    """
    Funzione che restituisce l'euristica h per il contesto sp0ecificato sulla base dei vincoli non ancora soddisfatti

    Parametri:
        Cs (List): la lista dei vincoli rimanenti da soddisfare

    Returns:
        Int: L'euristica h per il contesto specificato
    """

    h = 0
    for c in Cs:
        if self.kb.getAreaSeverity(c) == 2:
            h += 1
    return h
```

Selezione della variabile

Un altro passo importante in un problema di ottimizzazione è la scelta della variabile successiva da assegnare. Un buon criterio di scelta della variabile può portare a trovare molto prima la migliore assegnazione totale. Per questo problema è stata definita una funzione che preferisce le aree vicine ad aree già assegnate e che abbiano un livello di criminalità più basso. In questo modo è possibile valutare il prima possibile i vincoli relativi alle aree già assegnate.

```python
def selectVariable(self, Vs, context):
    """
    Funzione che seleziona la variabile da assegnare in base al contesto specificato e alle variabili rimanenti.
    Vengono preferite le variabili che sono vicine ad aree già assegnate e che hanno gravità minore

    Parametri:
        Vs (List): la lista delle variabili rimanenti
        context (Dict): il contesto corrente

    Returns:
        Any: La variabile da assegnare
    """

    bestArea = Vs[0]
    keys = list(context.keys())
    for area in keys:
        if context[area]:
            nearAreas = self.kb.getAreasByDistance(area, 1)
            for nearArea in nearAreas:
                if nearArea in Vs:
                    if self.kb.getAreaSeverity(nearArea) == 0:
                        return nearArea
                    elif self.kb.getAreaSeverity(nearArea) == 1:
                        bestArea = nearArea
    return bestArea
```

In aggiunta, va specificata la decisione di dare un ordine specifico al dominio delle variabili: nel dizionario dei domini della variabili, ogni dominio presenta prima il valore false e poi il valore true. In questo modo verranno selezionate variabili con gravità minore a cui verrà assegnato dapprima il valore false (pattuglia non assegnata). Attraverso questa strategia, il problema di CSP cercherà la migliore soluzione valutando prima le soluzioni che assegnano meno pattuglie possibili alle aree con criminalità pari a 1.

Conclusioni

L'apprendimento supervisionato è stato fondamentale per predire il livello di criminalità nelle diverse aree. Le metriche di valutazione hanno permesso di confrontare l'efficacia dei modelli Decision Tree, Random Forest e AdaBoost, scegliendo quello più adatto. Tuttavia, i risultati ottenuti dall'apprendimento supervisionato non sono stati particolarmente elevati. Questo è dovuto in parte alle limitazioni della potenza computazionale disponibile, che ha costretto a limitare i valori degli iperparametri e il numero delle aree considerate, compromettendo la capacità di esplorare soluzioni più complesse e ottimizzate.

Il progetto presenta sicuramente possibili sviluppi futuri quali:

- miglioramento dell'apprendimento supervisionato attraverso altre tecniche più avanzate
- espansione della knowledge base e dei criteri di copertura delle aree con i dati relativi alle festività nella città di Chicago
- considerazione di ulteriori criteri per l'ottimizzazione della disposizione delle pattuglie come, per esempio, il tempo di risposta delle pattuglie
- realizzazione di una interfaccia utente per una migliore comprensione delle predizioni e disposizioni.

Riferimenti Bibliografici

[1] https://artint.info/3e/html/ArtInt3e.html

[2] https://data.cityofchicago.org/Public-Safety/Crimes-2018/3i3m-jwuy/about_data

[3] https://data.cityofchicago.org/Facilities-Geographic-Boundaries/Boundaries-Community-Areas-current-/cauq-8yn6