Obiettivo del progetto

Il progetto è stato realizzato sulla piattaforma Google Colab che permette di scrivere ed eseguire in cloud codice python riducendo problemi di compatibilità tra ambienti di lavoro su diverse macchine. Le librerie usate per machine learning e elaborazione dei dati sono sklearn e numpy/pandas.

L'obiettivo del progetto è quello di realizzare un modello per la classificazione binaria di un dataset che punti a produrre una routine di addestramento con la migliore accuracy possibile.
Il dataset di training per addestrare il modello contiene valori sia numeric che categorical da gestire in fase di preprocessing.

Preprocessing

Il preprocessing è stato influenzato dalla presenza delle feature categoriche. Queste feature sono state convertite in numeric tramite *OneHotEncoding* che prende tutti i valori diversi che compaiono nella colonna e crea un uguale numero di colonne, ognuna rappresentante uno di questi valori. Le nuove colonne possono assumere per ogni riga solo i valori 0 e 1: 1 quando il sample assume il valore rappresentato da quella colonna, 0 altrimenti. Per questo motivo non tutte le fasi del preprocessing sono applicabili alle feature categoriche. Verrà analizzato in dettaglio in seguito.

Le fasi di preprocessing si articolano nel seguente modo:

Separazione training test set

Come prima operazione sul dataset sono stati individuati il training set, su cui effettuare l'addestramento, ed il test set su cui verificare l'accuratezza del modello. Lo split è stato eseguito utilizzando il metodo *train_test_split* che restituisce le colonne con i dati delle feature e la colonna finale contenente le label sia per il training che per il test set.
Lo split ha suddiviso train/test set size con rapporto 70/30, in modo stratificato per garantire che train e test set rispettino la proporzionalità sulle due classi rispetto al dataset originale.

Eliminazione valori nulli

E' l'unica fase di preprocessing eseguita prima di fare *OneHotEncoding*. Viene effettuato sia sul training che sul test set. Sul dataset a disposizione i valori nulli sono presenti solo sulle feature categoriche, per questo tipo l'unico imputer utilizzabile è il *SimpleImputer* con parametro strategy uguale a '*most_frequent'* che quindi sostituisce i valori nulli con il più frequente per la feature in considerazione. Questa tecnica è stata utilizzata solo sulle colonne categoriche.
Nonostante l'assenza di valori nulli sulle colonne numeriche, nell'eventualità siano invece presenti nel test set, è stata comunque inserita nel preprocessing l'imputazione dei valori nulli numerici con il *SimpleImputer* con strategia '*mean'*.

Feature scaling

Lo scaling serve per riportare il range dei valori delle varie feature sulla stessa scala di grandezza e viene effettuato sia sul training che sul test set. Per questo motivo si può applicare solo sulle colonne numeriche (le colonne categoriche trasformate con *OneHotEncoding* devono assumere solo valore 0 o 1). Durante le varie esecuzioni la scelta dello scaler non ha influenzato molto i risultati dell'addestramento per questo è stato scelto il *RobustScaler*, per garantire anche resistenza a possibili outlier.

Balancing & Anomaly detection

Il balancing ha lo scopo di equilibrare le due classi, in modo tale da rendere uguali la minoritaria e la maggioritaria, viene effettuato solo sul training set.
Ci sono due approcci possibili, oversampling, che aggiunge valori alla classe minoritaria per renderla uguale alla maggioritaria ed under sampling, che viceversa elimina valori dalla classe maggioritaria per renderla uguale alla minoritaria.
La fase di anomaly detection, invece, si occupa di eliminare righe del dataset che risultano anomale e che quindi potrebbero essere rumore del dataset e peggiorare le prestazioni.
Questi due aspetti sono collegati, facendo balancing infatti può verificarsi che:

- Nel caso si scelga di fare oversampling, potrebbero essere aggiunte delle righe che introducono del rumore ed eseguendo successivamente al balancing l'anomaly detection questo rumore introdotto viene eliminato
- Nel caso invece si scelga di fare undersampling non facendo anomaly detection c'è il rischio che vengano mantenute delle righe anomale ed eliminate righe utili. Quindi in questo caso l'anomaly detection può essere effettuata prima del balancing.

Nel dataset di partenza, le due classi erano suddivise in 76%/ 24%, facendo undersampling sarebbero stati eliminati troppi dati, infatti l'accuracy nelle esecuzioni con undersampling degrada notevolmente. Viene utilizzato il metodo di oversampling *SMOTE* (ha prestazioni migliori rispetto a *RandomOverSampling* che sceglie in modo randomico alcune righe e le duplica).
In ogni caso l'anomaly detection non ha apportato notevoli miglioramenti delle prestazioni, ma si è scelto comunque di effettuarla.

Classificatori

Per trovare il classificatore che desse accuratezza migliore è stata usata la grid search cross test. Inizialmente sono stati provati i classificatori: *SVC*, *linear_SVC* e *decision tree*.
Con i seguenti iperparametri:
SVC:

- *kernel*: rbf, sigmoid
- *C*: [0.01, 0.1, 1, 10]
- *gamma*: [0.01, 0.1, 1, 10], auto

LinearSVC:

- *C*: [0.01, 0.1, 1.0, 10]

Decision tree:

- *criterion*: gini, entropy
- *depth*: [3, 5, 10]

Da questa prima esecuzione il migliore è risultato essere *LinearSVC* con parametro C = 1.0 che dava un'accuracy sul training di 0.9813 e sul test di 0.8477 quindi con un evidente overfitting.
A questo punto sono stati provati i classificatori ensemble per cercare di migliorare ulteriormente l'accuracy ed allo stesso tempo diminuire l'overfitting.

AdaBoostingClassifier

- *base_estimator:* LinearSVC di iperparametro C = 1.0
- *learning_rate*: [0.1, 0.5, 1.0]

Per questo classificatore è stato come *base_estimator* il classificatore con i parametri che avevano dato migliori risultati nella fase precedente.
In questo modo diminuisce notevolmente l'overfitting, ma scende anche l'accuracy che sul training è di 0.8442 e sul test di 0.8200, anche diminuendo il learning rate si notano poche variazioni.

RandomForestClassifier

- *max_depth*: [5, 10, 25, 30]
- *n_estimators* [10, 100, 1000, 1500]

I parametri migliori sono risultati essere *max_depth* 25 e *n_estimators* 1000.
I risultati ottenuti hanno un'accuracy non troppo alta ed un overfitting contenuto. Sul training l'accuracy è di 0.8771, mentre sul test 0.7964.

GradientBoostingClassifier

- *n_estimators*: [10, 100, 1000, 1500]
- *depth*: [2, 4, 5]
- *learning_rate*: [0.1, 0.5, 1.0]

Risulta essere il miglior classificatore con i paramentri di *n_estimators* 1000 *depth* 2, *learning_rate* 0.5.
Tentativi con depth più alta andavano facilmente in overfitting anche aumentando il numero di stimatori e con learning rate massimo. In questo caso l'accuracy sul training è 0.9138 e sul testing 0.8556.

Conclusioni

In generale su questo dataset i classificatori migliori risultano essere gli ensamble. Con gli altri classificatori è risultato difficile non solo la gestione dell'overfitting, ma anche le prestazioni sull'accuracy risultavano essere più basse.
GradientBoosting da subito è risultato essere il migliore, consente infatti anche di abbassare il learning rate che negli altri casi, per conenere l'overfitting, doveva essere massimo degradando però le prestazioni.