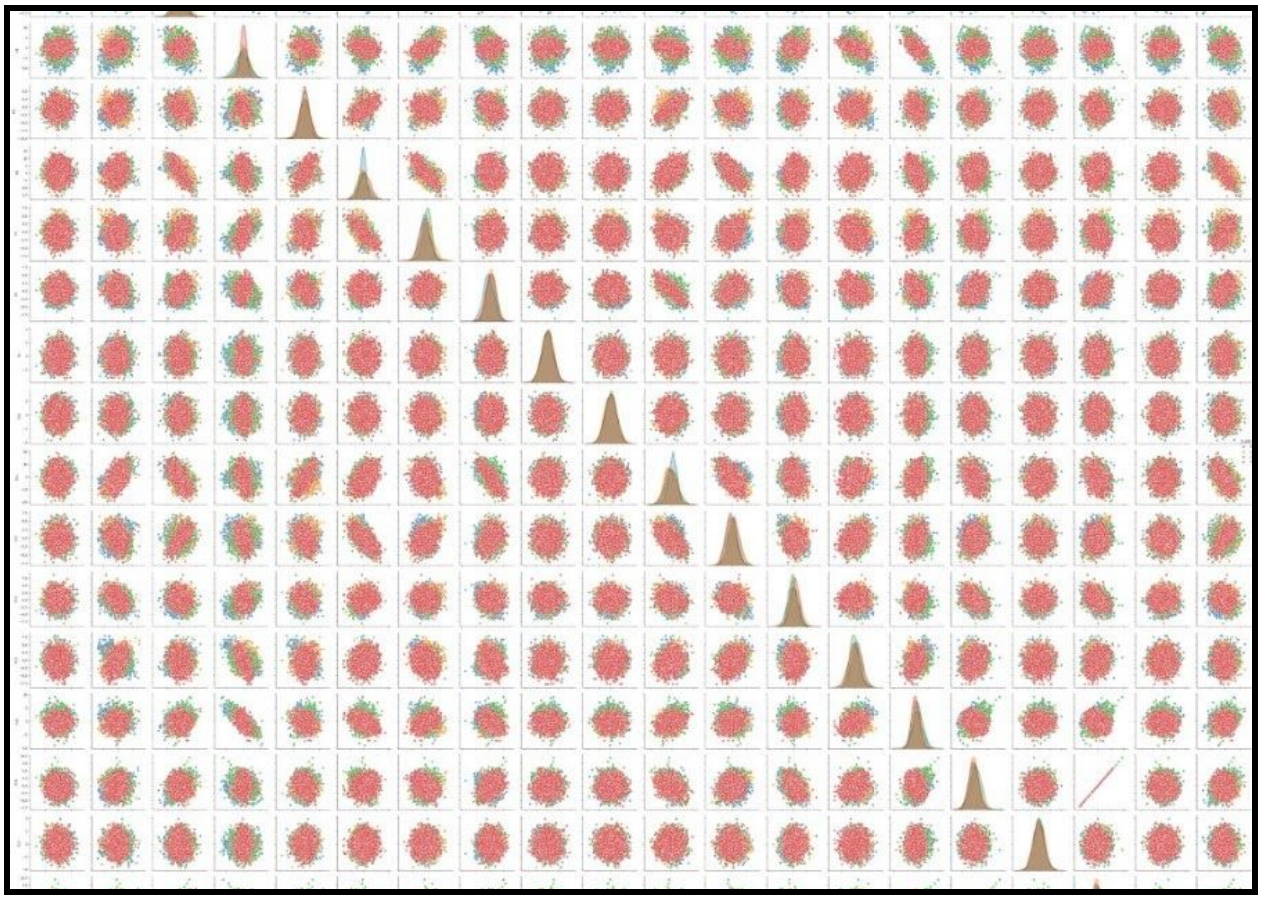
METODI DI OTTIMIZZAZIONE PER BIG DATA

Relazione sul progetto finale



Alessandro Tenaglia - Roberto Masocco

20/06/2020

A.A. 2019/2020

INTRODUZIONE

Presentiamo in questa relazione un quadro completo del lavoro svolto, descrivendone le varie fasi, discutendo gli strumenti utilizzati e illustrando le soluzioni adottate e i risultati ottenuti. Dove necessario, abbiamo corredato i dati numerici con grafici che meglio rappresentassero quanto da noi riscontrato.

Per eventuali dettagli relativi all'implementazione software, ulteriori rispetto a quanto discusso nel presente documento, si fa riferimento ai file di codice allegati ed ai commenti in essi contenuti. L'ambiente di sviluppo impiegato è stato PyCharm della JetBrains, con ambienti Python configurati mediante Anaconda.

Ad una breve presentazione del contenuto della repository seguono paragrafi che spiegano singolarmente le varie operazioni, nello stesso ordine in cui queste sono state, svolte seguendo il modello CRISP DM:

1. Data Understanding.
2. Data Preparation.
3. Modeling.

CONTENUTO DELLA REPOSITORY

I file *evaluation.py* e *aux_lib.py* contenuti nella stessa cartella del presente documento sono relativi alla valutazione del progetto. Per i dettagli su come eseguirla si rimanda al file *README.txt*.

Il file *best_pipeline.sav* contiene la miglior pipeline da noi selezionata, tarata e addestrata sul training set in nostro possesso e poi serializzata. Si tratta di un oggetto *Pipeline* definito nel modulo *sklearn.pipeline* di Scikit-learn, che ha bisogno della classe *KNNReplacerIQR()* definita in *aux_lib.py* per funzionare. I dettagli di questa scelta saranno forniti nel seguito.

La cartella *Bootcamp* contiene il codice di tutti gli script che abbiamo usato nelle varie fasi del lavoro, in particolare per quanto riguarda le grid search e la cross-validation dei vari modelli testati, nonché funzioni ausiliarie per descrivere i dati e produrre grafici, il tutto adeguatamente commentato. La sottocartella *Results* contiene poi i risultati generati automaticamente dalle nostre grid search. Abbiamo voluto includere tutto ciò a corredo di questa relazione per rendere il più possibile chiare le scelte fatte e le soluzioni adottate anche dal punto di vista software, ma precisiamo che tali script non sono forniti con l'intento principale di essere eseguiti (ad esempio: alcune importazioni di moduli o file potrebbero non funzionare correttamente), anche perché i test che implementano richiedono molto tempo d'esecuzione.

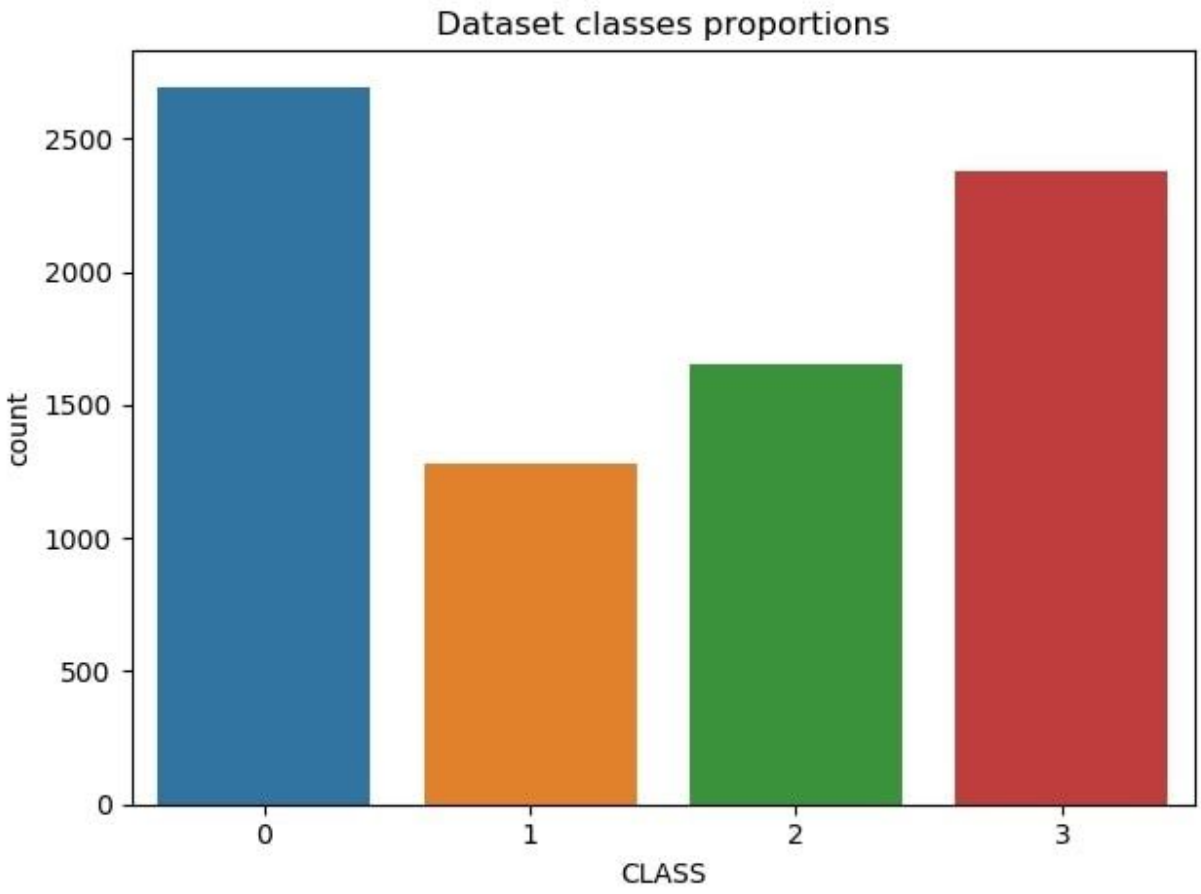
DATA UNDERSTANDING

Capire quale potesse essere il miglior classificatore da testare e successivamente applicare, ha richiesto uno studio preliminare attento del dataset ricevuto. Questo, racchiuso in un file CSV, è composto da 8000 punti, descritti da 20 *feature* ciascuno e assegnati a una tra 4 possibili classi, indicate con numeri da 0 a 3.

```
Dataset shape: (8000, 21)
                F1           F2  ...          F20         CLASS
count  7994.000000  7994.000000  ...  7997.000000  8000.000000
mean     -0.013077    -0.261413  ...    -0.355555     1.463375
std       1.006235     1.852793  ...     1.794666     1.231198
min      -4.181155    -6.980290  ...    -7.563245     0.000000
25%      -0.698506    -1.441144  ...    -1.563262     0.000000
50%      -0.028194    -0.261095  ...    -0.373514     2.000000
75%       0.666096     0.944857  ...     0.825741     3.000000
max       3.774161     7.155359  ...     6.774458     3.000000
```

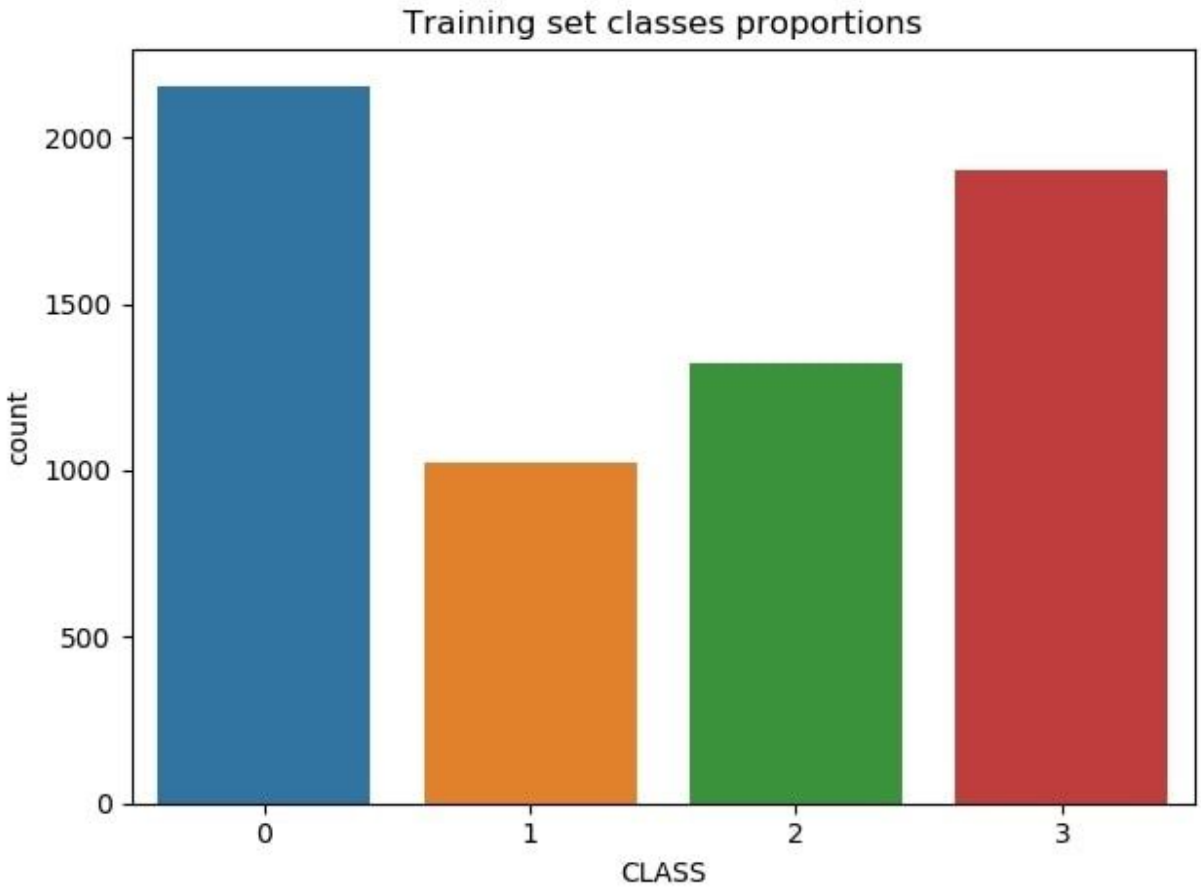
Dalla descrizione del dataset abbiamo riscontrato che tutte le *feature* sono di natura quantitativa, con valori sia positivi sia negativi, e con diversi range di appartenenza. Inoltre abbiamo rilevato la presenza di qualche valore mancante all'interno del dataset e di alcuni outlier.

Infine un istogramma delle classi target ha manifestato una predominanza di punti assegnati alle classi 0 e 3, e una minoranza alla classe 1 e 2, come mostrato nella figura sottostante.

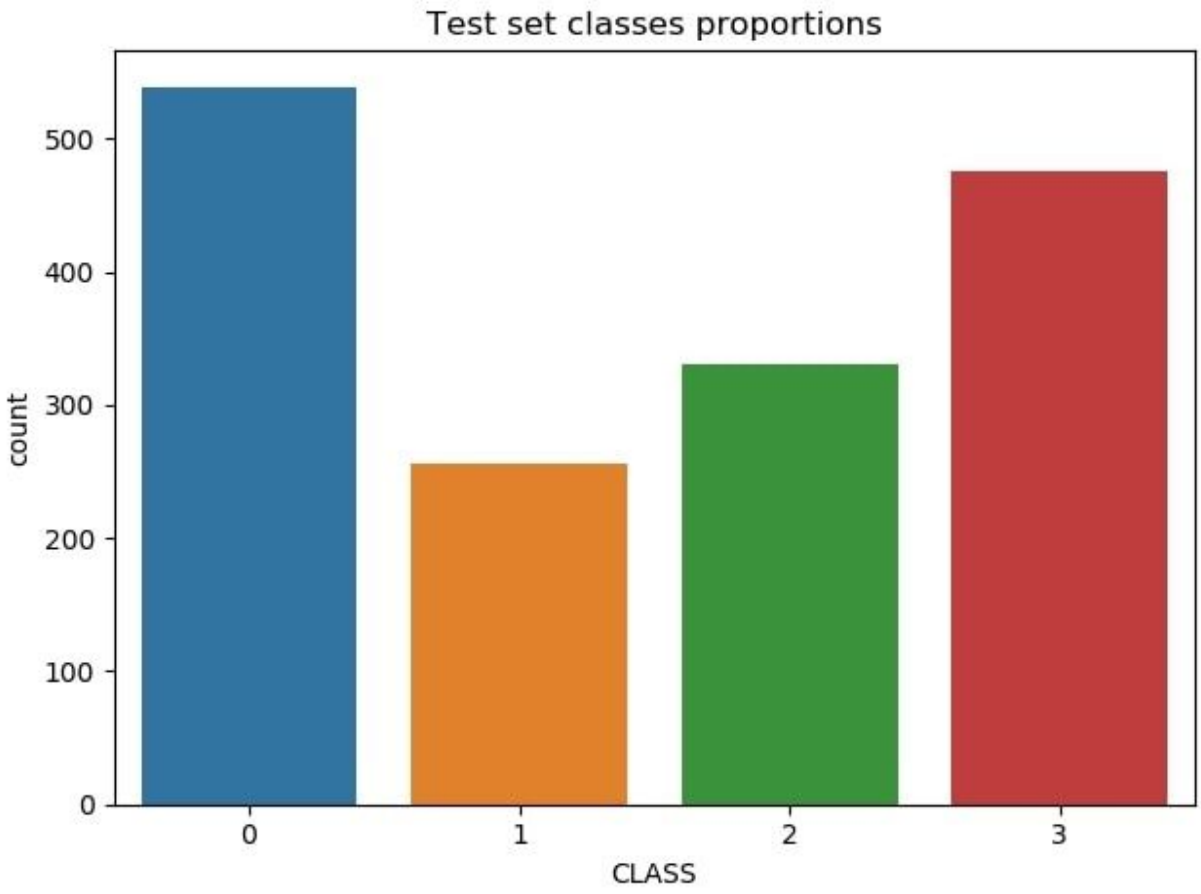

Dataset classes proportions
count
2500
2000
1500
1000
500
0
0
1
2
3
CLASS

DATA PREPARATION

Per addestrare e testare i classificatori abbiamo diviso il dataset in *training set* (80%) e *test set* (20%) usando il metodo *train_test_split,* del modulo *sklearn.model_select*. Quest'ultimo ci ha permesso sia di scegliere "casualmente" la suddivisione dei punti (garantendo però la riproducibilità tra le varie run impostando il parametro *random_state* a 42[1]) sia di rispettare le stesse proporzioni delle classi del dataset originale in entrambe le parti attraverso l'attributo *stratify*, come abbiamo verificato coi due seguenti istogrammi.

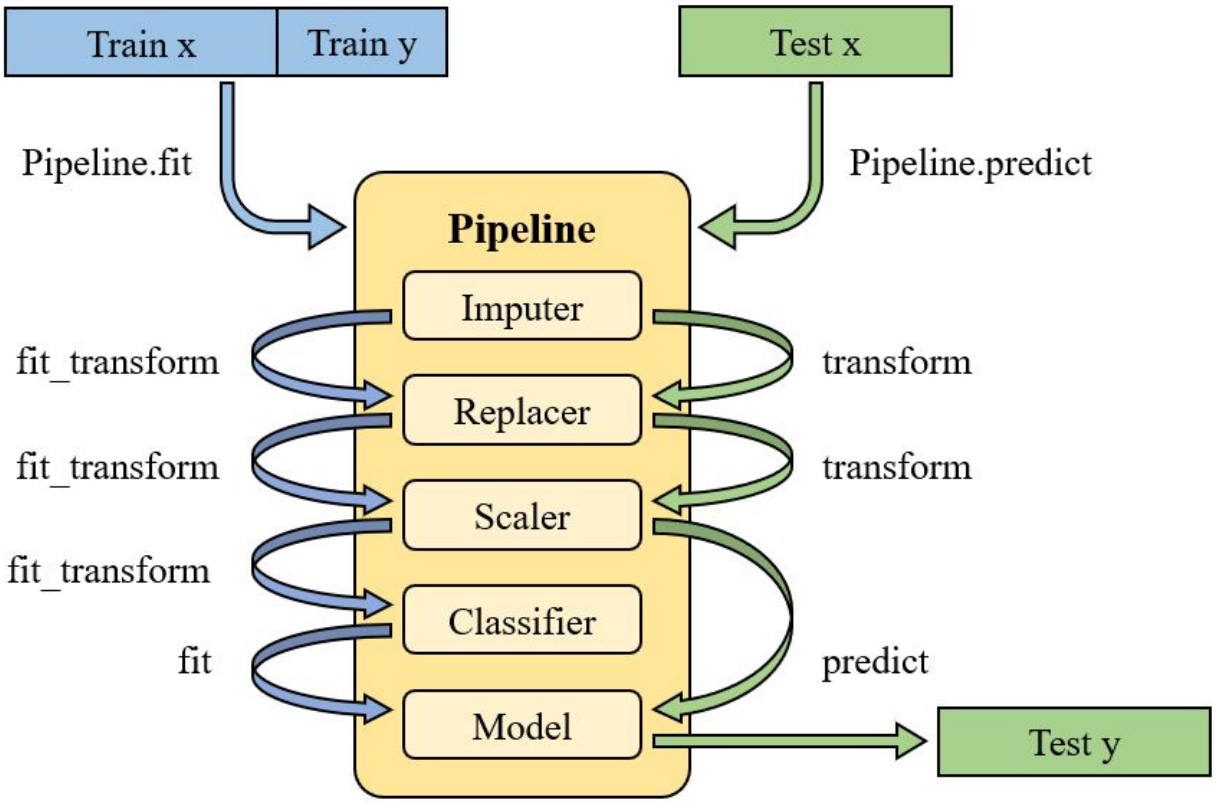


[1] cfr. Adams D., Guida galattica per gli autostoppisti - Il ciclo completo, Mondadori, Milano, 2015, p. 141.



A seguito dello split in *training set* e *test set* ci siamo rapidamente resi conto di come diverse metodologie di preprocessing potessero influire sulle capacità del classificatore finale, anche e soprattutto a parità di questo. E' risultato naturale pertanto includere la scelta della miglior *pipeline* di preprocessing nelle decisioni da prendere, e quindi nei test da eseguire. Un grande aiuto nel fare questo ci è stato dato dalla classe *Pipeline*[2] offerta nel modulo *sklearn.pipeline* della libreria Scikit-learn: con essa è possibile specificare più possibili combinazioni di metodi di preprocessing e classificatori, impostare le griglie di parametri per ciascuno dei vari step e lanciare delle grid search/cross-validation con gli stessi oggetti *GridSearchCV,* che restituirà la miglior combinazione di metodi di preprocessing, classificatore e parametri per ciascuno.

[2] https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html



Le istanze delle classi costituenti i vari stadi della pipeline vanno dapprima "addestrate" sui dati del *training set*, chiamando un metodo *fit*, e poi possono processare i dati eseguendo il metodo *transform*, in modo che ogni step lavori sull'output corretto dello step precedente. In fase di *predict* il *test set* viene processato esclusivamente con i vari metodi *transform*, utilizzando le istanze precedentemente configurate e tarate **sul solo training set**, in accordo alle norme di progettazione.

Sostituzione dei valori mancanti

Il primo passo nella fase di preparazione dei dati è stato quello di gestire i valori mancanti, in quanto la loro presenza non permette l'esecuzione dei classificatori.

Il primo approccio che abbiamo seguito è stato quello di sostituire i valori mancanti con la media della feature di appartenenza, essendo tutte quantitative, attraverso la classe *SimpleImputer* del modulo *sklearn.impute* e specificando la strategia *mean*.

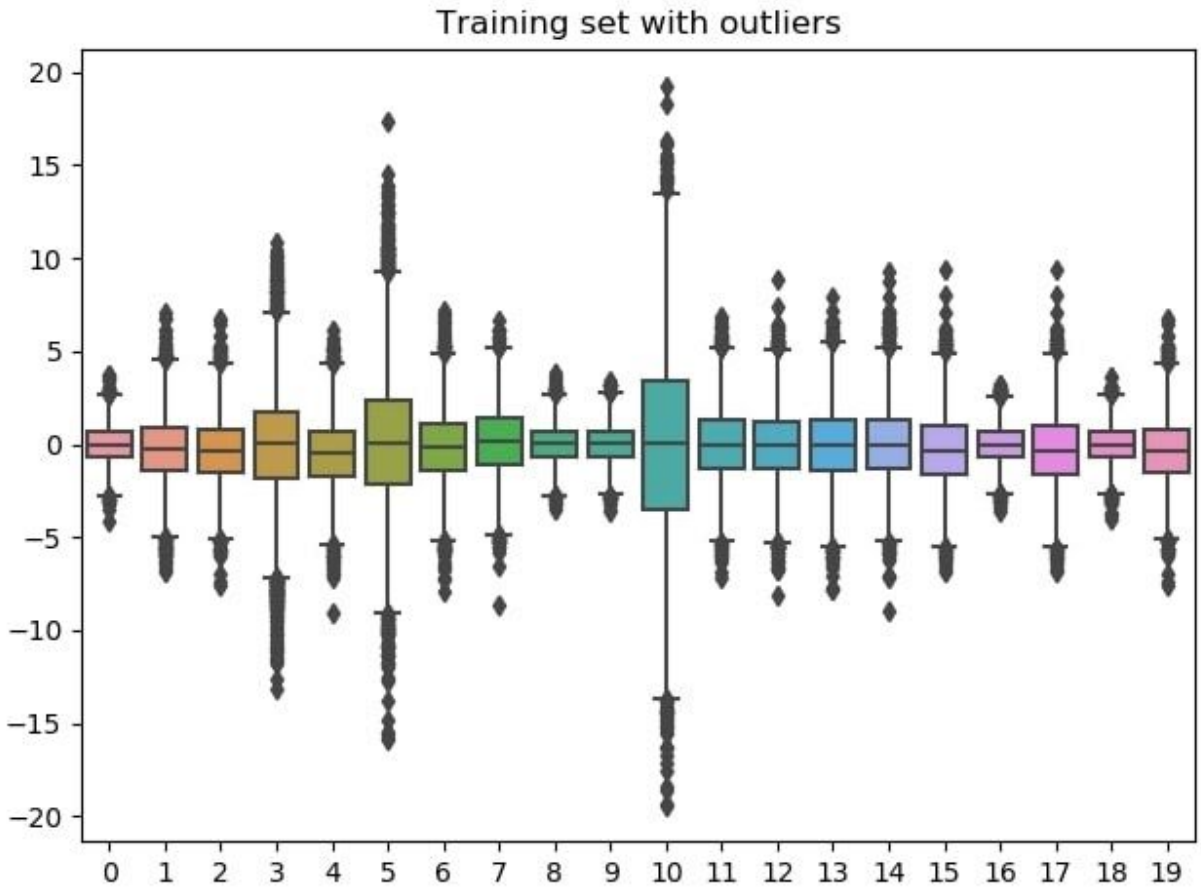
Successivamente abbiamo raffinato questo metodo implementando la sostituzione tramite l'algoritmo *K-Nearest-Neighbors*, in cui il valore da inserire viene calcolato a partire da quelli dei soli vicini del punto interessato, determinati in base alle altre feature non nulle. Questo approccio è stato implementato attraverso la classe *KNNImputer* del modulo *sklearn.*impute, il cui parametro di riferimento è *n_neighbors*, che specifica il numero di vicini da considerare per calcolare il nuovo valore da sostituire. La scelta di questo parametro è stata effettuata in cross validation, includendolo nella miglior pipeline ottenuta, quindi determinato anche in relazione al classificatore utilizzato negli step successivi.

Sostituzione degli outliers

Il secondo problema che abbiamo affrontato è stato quello di ripulire il training set dagli *outlier*. Per risolverlo è stato necessario affrontare due questioni distinte: capire quando considerare i punti "*outlier*" e come sostituirli in maniera opportuna.

Il primo approccio utilizzato per rilevare gli *outlier* è stato quello dello *scarto interquartile*, che fornisce una misura della variabilità dei dati. In base al quale gli outlier sono definiti come i punti che si trovano al di sotto di Q1 - 1,5 IQR o al di sopra di Q3 + 1,5 IQR.

Segue il boxplot delle features del training set prima di effettuare la sostituzione, il quale fornisce una rappresentazione grafica molto semplice e permette di rilevare immediatamente i valori anomali.



Un approccio alternativo che abbiamo sperimentato per rilevare gli outlier è stato quello di utilizzare lo *Z-Score*, ossia la misura di quante deviazioni standard ogni valore si discosta dalla media. In particolare un punto è considerato outlier se il suo *Z-Score* è in modulo maggiore di 3.

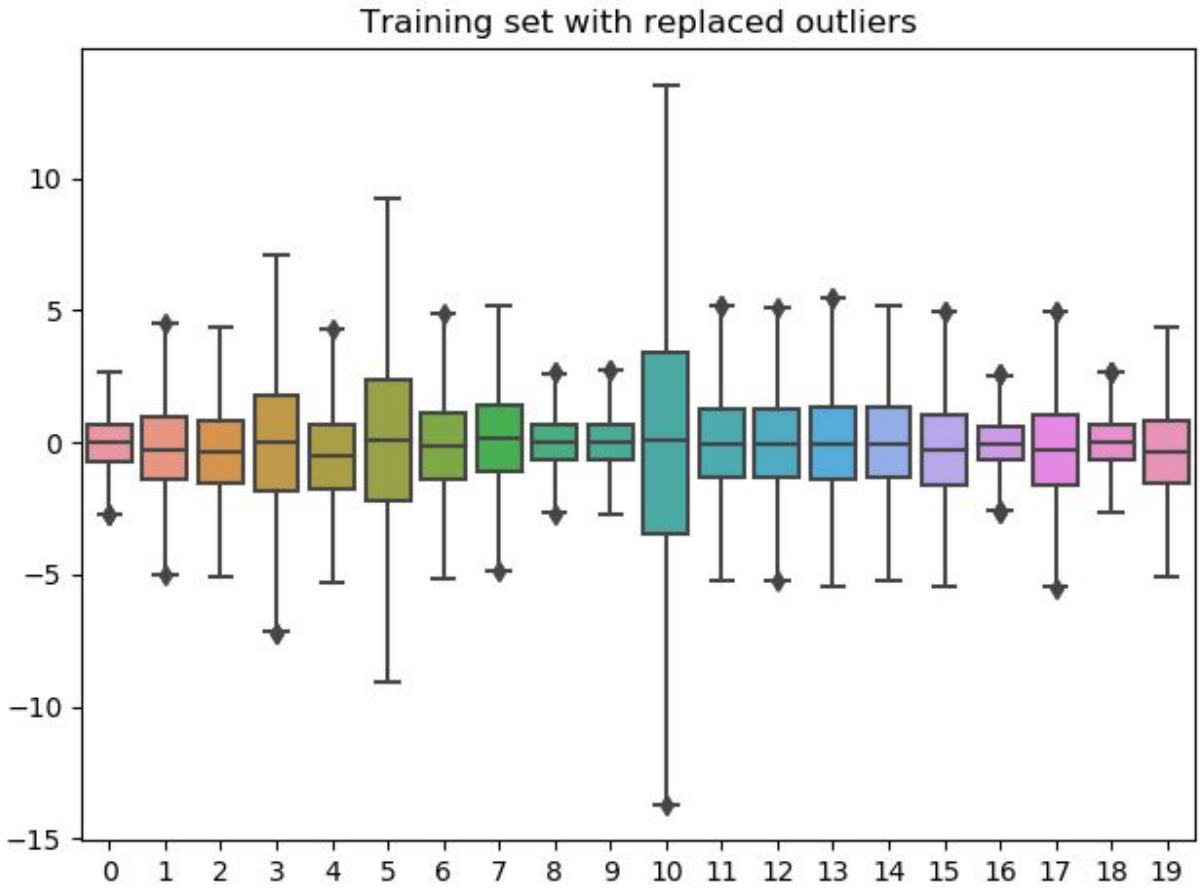
Una volta che gli outlier sono stati determinati abbiamo affrontato la questione del come questi potessero essere sostituiti. Le soluzioni da noi adottate sono state le stesse proposte precedentemente per la gestione dei valori mancanti, quindi o sostituirli con la media della feature di appartenenza oppure utilizzare il più sofisticato KNN con un'opportuna scelta del parametro *n_neighbors*.

Per poter includere uno step all'interno di una *Pipeline* è richiesto l'utilizzo di classi che implementano metodi di *fit* e *transform*, quindi è stato necessario creare delle nuove classi ad hoc che eseguissero il rilevamento e la sostituzione

degli outlier. Nello specifico abbiamo realizzato 4 classi differenti per considerare tutte le possibili combinazioni tra i metodi di rilevamento e sostituzione sopra descritti, rispettivamente *MeanReplacerIQR*, *MeanReplacerZS*, *KNNReplacerIQR* e *KNNReplacerZS*.

Ancora una volta, nel modello finale il metodo di rilevamento e l'algoritmo di sostituzione, con i relativi parametri, sono stati scelti in base alla miglior pipeline ottenuta in cross validation.

Nello specifico il modello migliore si è dimostrato essere quello che utilizza l'IQR per rilevare gli outlier e l'algoritmo KNN per sostituirli con un numero di vicini pari a 2. Segue il boxplot delle features del training set dopo aver sostituito opportunamente gli outlier.

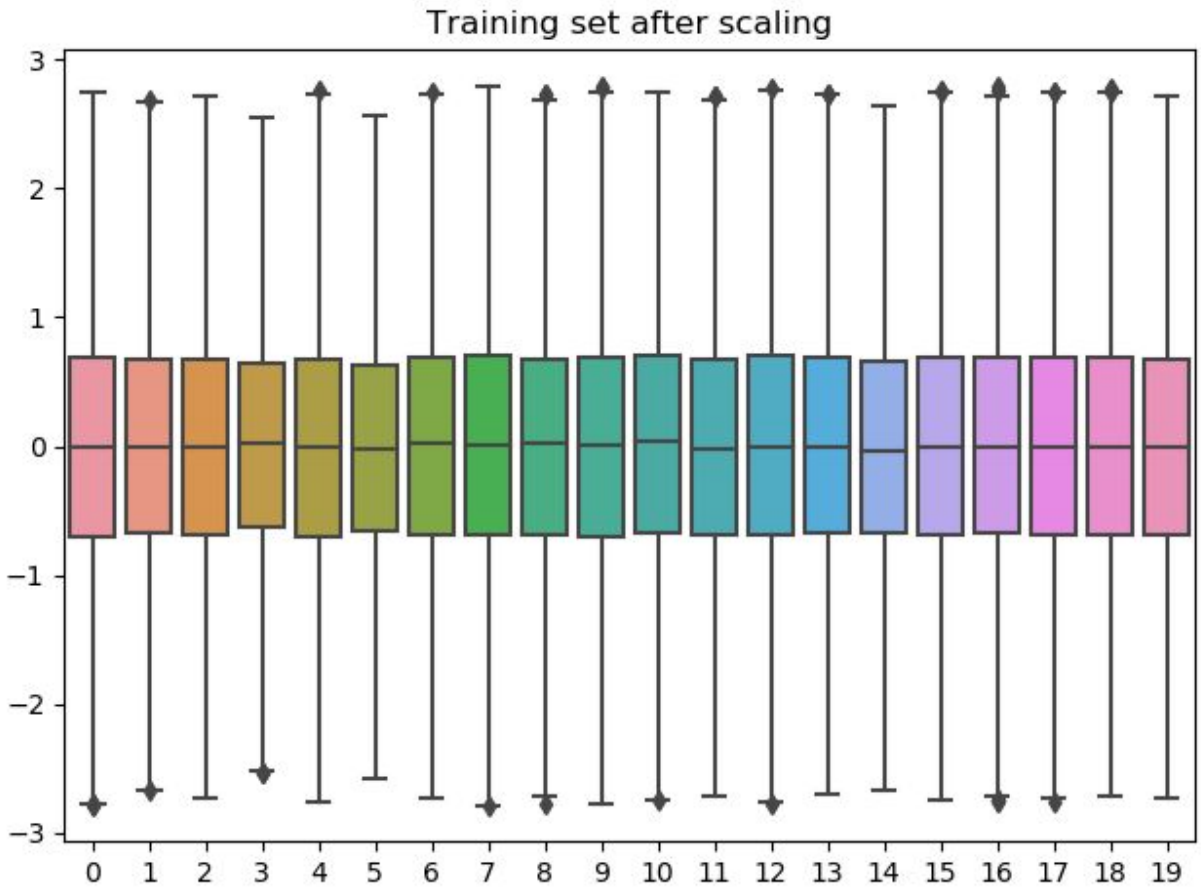


Scaling dei dati

Il passo successivo nella fase di preparazione del dataset è stato quello di riscalare le feature, aspetto cruciale del preprocessing in quanto gli algoritmi di classificazione risentono pesantemente di eventuali differenze tra le scale dei valori.

La scelta è ricaduta sulla classe *StandardScaler* del modulo *sklearn.preprocessing*, il quale ha permesso di rimappare i valori delle feature in un range unico che fosse comunque abbastanza ampio da racchiudere tutta l'informazione data da esse, ma anche simmetrico facendo sì che le stesse avessero media nulla.

Segue il boxplot del training set dopo aver riscalato le feature, con il relativo *describe*.



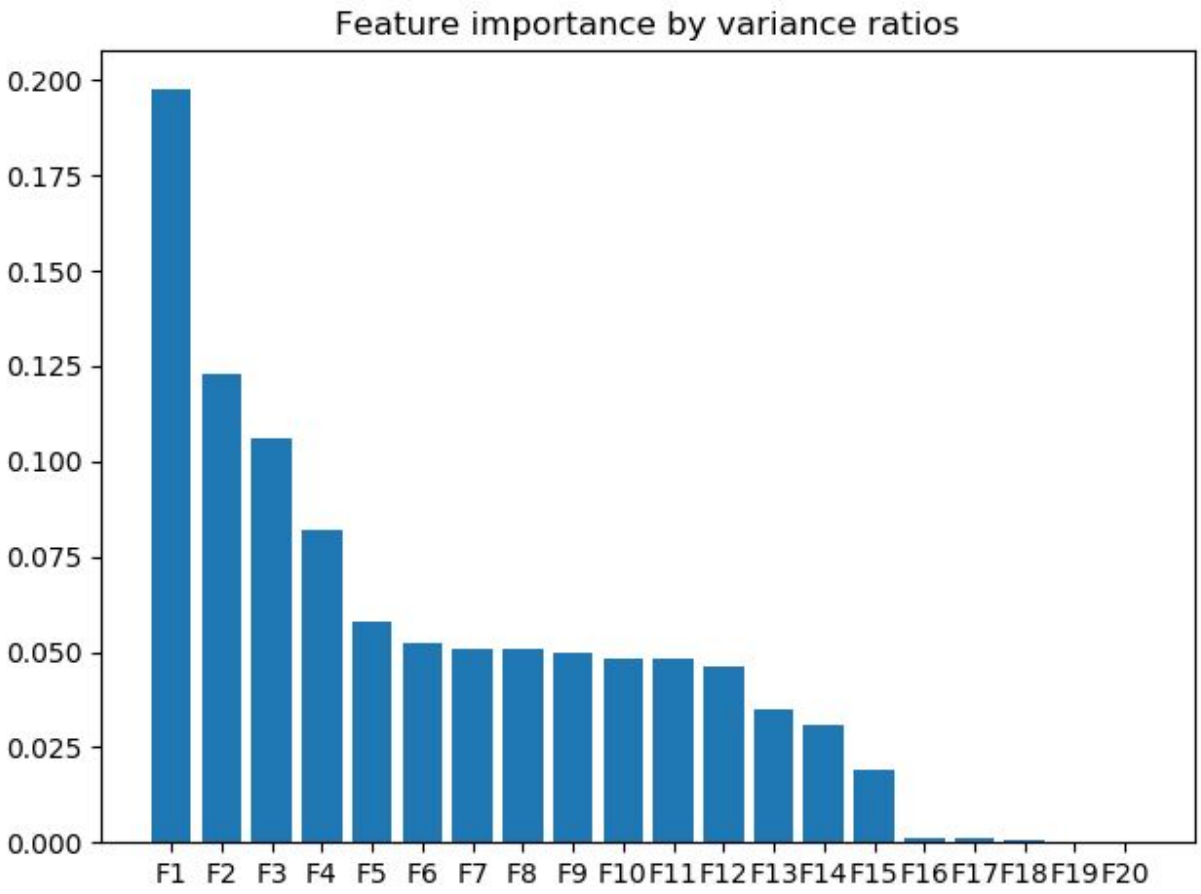
```
Training set features properties after scaling:
                 0             1   ...            18            19
count  6.400000e+03  6.400000e+03  ...  6.400000e+03  6.400000e+03
mean  -3.108624e-17  1.387779e-17  ...  1.554312e-17  6.661338e-18
std    1.000078e+00  1.000078e+00  ...  1.000078e+00  1.000078e+00
min   -2.785831e+00 -2.676752e+00  ... -2.719299e+00 -2.736254e+00
25%   -6.991649e-01 -6.663306e-01  ... -6.854187e-01 -6.907344e-01
50%   -6.259116e-03 -7.557603e-03  ... -7.060451e-03 -5.871797e-03
75%    6.839522e-01  6.730324e-01  ...  6.887719e-01  6.740943e-01
max    2.747062e+00  2.685713e+00  ...  2.768942e+00  2.714771e+00
```

Ulteriori considerazioni

Questi sono gli step che costituiscono la pipeline finale.

Prima di entrare nell'ultima fase della cross-validation abbiamo tentato anche altre strade, per cercare di mitigare alcuni problemi di cui il dataset sembra soffrire e che pensavamo potessero influenzare la precisione dei classificatori.

Avendo molte feature, ben 20 per ciascun punto, abbiamo pensato che non tutte potessero essere ugualmente importanti. Per avere un'idea di come ciascuna influenzasse la classificazione abbiamo eseguito una *Principal Component Analysis* utilizzando la classe *PCA* del modulo *sklearn.decomposition*. Un grafico dell'apporto delle features sulla varianza delle classi ha mostrato che molte sono ugualmente importanti, e solo le ultime cinque potrebbero dirsi irrilevanti.



Test preliminari sui modelli più promettenti non hanno però mostrato miglioramenti significativi eliminando le feature "meno importanti", e concentrandosi su quelle "più importanti" la situazione è peggiorata a causa di un'eccessiva perdita d'informazione.

Infine abbiamo provato a ridurre gli effetti dello sbilanciamento del dataset a favore delle classi 0 e 3 effettuando un *resampling* dei dati. Nonostante le varie metodologie testate fornite dalla libreria *imbalanced-learn*[3], né l'*over* né l'*under-sampling* si sono dimostrati utili: nel primo caso ogni classificatore risentiva pesantemente di overfitting, non mostrando buone capacità di generalizzazione sui nuovi dati dopo l'addestramento, mentre nel secondo la perdità d'informazione risultava sempre eccessiva ed impediva ad ogni modello di raggiungere una precisione soddisfacente.

[3] https://imbalanced-learn.readthedocs.io/en/stable/index.html

MODELING

La ricerca del miglior modello, sulla base di tutte le idee sviluppate fino a questo punto, è stata una ricerca combinata della miglior pipeline di preprocessing e del classificatore. Dopo aver selezionato alcuni classificatori promettenti tra quelli più comuni, abbiamo eseguito delle *grid search* per tarare i loro migliori iperparametri e selezionare una sequenza di step di preprocessing adeguata, nonché un set di parametri di configurazione, tra quelle proposte in precedenza. La metrica di valutazione dei risultati sul training set è stata sempre l'F1 macro score.

Riguardo gli step di preprocessing, tra tutte le simulazioni effettuate la pipeline migliore è risultata sempre quella basata sull'algoritmo KNN sia per il riempimento dei valori mancanti che per la sostituzione degli outlier, rilevati tramite IQR. La griglia scelta per il numero di vicini in KNN è pari a [2, 5, 10], e i valori migliori non erano ovviamente sempre gli stessi tra le varie cross-validation dei classificatori.

Il primo tipo di classificatore che abbiamo provato ad applicare sono state le *Support Vector Machines*, configurando gli oggetti *SVC* del modulo *sklearn.svm* per operare in modalità multiclasse *one-vs-one*. Abbiamo testato vari kernel e relativi iperparametri, più il parametro di configurazione *class_weight* specificando "*balanced*" oppure "None":

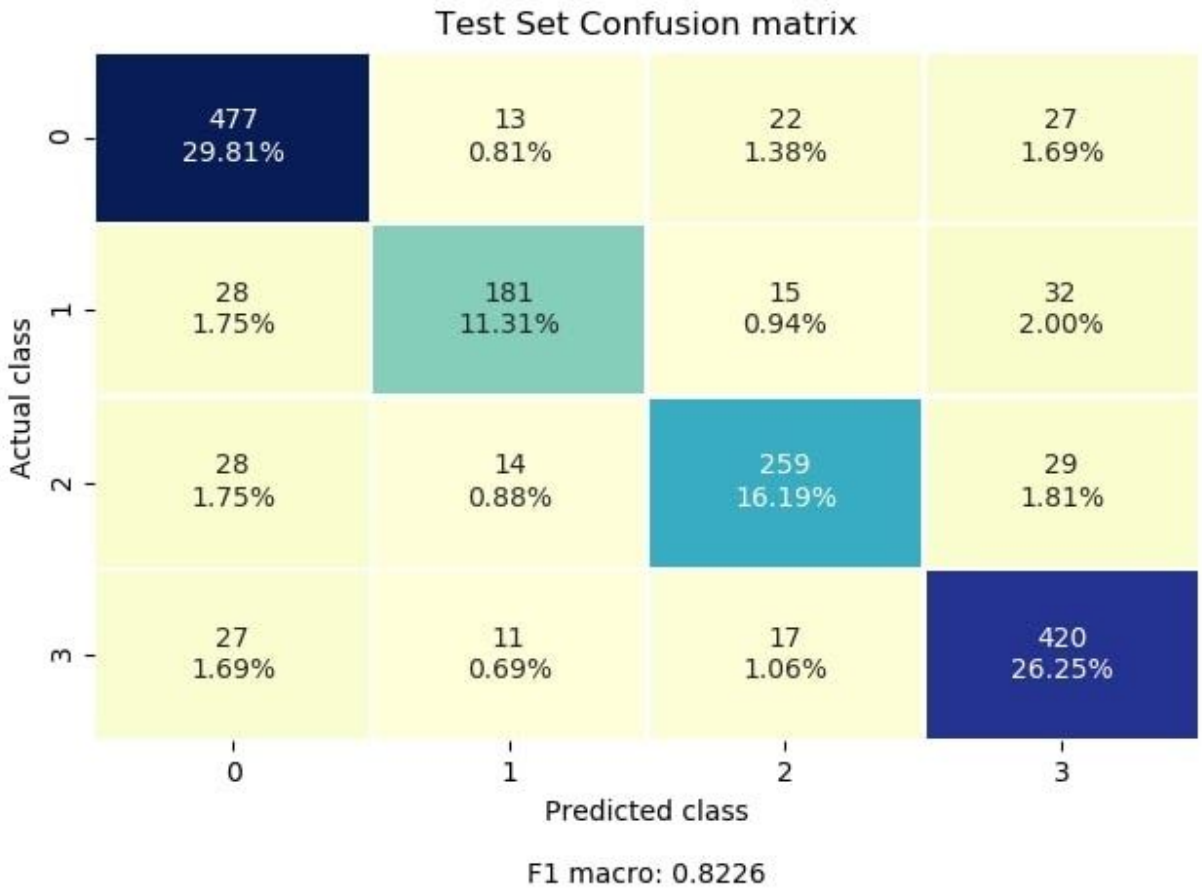
- **Kernel lineare:** La grid search è stata eseguita su una griglia logaritmica in base 10 per l'iperparametro *C*. Il miglior score sul training set è stato 0.5512 con C = 100.
- **Kernel polinomiale:** Facendo variare il grado del polinomio tra 1 e 4 e cercando *C* in griglie logaritmiche in base 10 e 2, il miglior score è stato 0.8070 con grado 2 e C = 16.
- **Kernel RBF:** Su questo abbiamo eseguito molteplici test, scegliendo *C* e γ entro griglie logaritmiche sia in base 10 che 2. Essendo i risultati promettenti abbiamo poi infittito la ricerca su range più specifici, ottenendo come miglior score sul training set 0.8205 con C = 50 e γ = 0.01.

Nonostante le perfomance della SVM con kernel gaussiano fossero soddisfacenti abbiamo provato a sperimentare altri due classificatori:

- **KNN:** Il numero di vicini è stato selezionato entro una griglia semplice pari a [2, 5, 10], provando anche due norme diverse con $p = 1$ e $p = 2$. Il miglior risultato è stato 0.7448 con $k = 10$ e $p = 1$.
- **Random Forests:** L'implementazione di questo classificatore in Scikit-learn richiede molti iperparametri. Ne abbiamo testati alcuni:
    - *n_estimators*: Numero di alberi nella foresta, scelto in [100, 500, 1000].
    - *max_depth*: Profondità massima degli alberi, scelta in [10, 50, 100, *None*].
    - *min_samples_leaf*: Minimo numero di punti che devono essere assegnati ad una "foglia", scelto in [1, 2, 4].
    - *min_samples_split*: Minimo numero di punti che vanno considerati per eseguire uno split in un nodo, scelto in [2, 5, 10].

  Il miglior risultato è stato 0.7767 con 1000 alberi, massima profondità 50, almeno 1 punto per ogni foglia e 2 per ogni nodo di split.

A fronte di tutte le prove il miglior classificatore si è dimostrata essere una **SVM con kernel gaussiano con C = 50, γ = 0.01 e *class_weight* = *None***, unita a una pipeline di preprocessing in cui entrambi gli algoritmi KNN nei due step considerano 2 vicini. Riaddestrando tale pipeline completa sul *training set* ed eseguendola per intero sul test set isolato all'inizio, abbiamo ottenuto la seguente matrice di confusione con un F1 macro score pari a **0.8226**.



Soddisfatti del risultato ottenuto abbiamo rigenerato la stessa pipeline riaddestrandola sull'intero dataset a nostra disposizione, serializzando e salvando poi l'oggetto ottenuto per poterlo ricaricare agevolmente durante la procedura di valutazione utilizzando il modulo standard *pickle* di Python.