rice_variety_classification

September 12, 2020

1 Classificazione delle varietà di riso in base alle caratteristiche morfologiche

Programmazione di Applicazioni Data Intensive A.A. 19/20
Laurea in Ingegneria e Scienze Informatiche
DISI - Università di Bologna, Cesena

Anthony Guglielmi

Citazioni: CINAR, I. and KOKLU, M., (2019). "Classification of Rice Varieties Using Artificial Intelligence Methods." International Journal of Intelligent Systems and Applications in Engineering, 7(3), 188-194. DOI: https://doi.org/10.18201/ijisae.2019355381

1.1 Parte 1a) Descrizione dell'area di studio

Tra i diveri tipi di riso certificato coltivato in TURCHIA, sono state selezionate per lo studio la specie Osmancik, che ha una vasta superficie di piantagione dal 1997 e la specie Cammeo coltivata dal 2014.

Per le due specie sono state scattate un totale di 3810 immagini del chicco di riso, e dopo una sofisticata rielaborazione sono state fatte deduzioni sulle loro caratteristiche. Per ogni chicco di riso sono state ottenute 7 caratteristiche morfologiche.

Lo scopo del seguente studio è quello di individuare una correlazione tra le caratteristiche morfologiche e la classe di appartenenza del chicco di riso, e sulla base di quella correlazione individuare il miglior modello di learning che sia in grado di classificare correttamente le due varietà.

Vengono importate le librerie necessarie per scaricare i file, organizzare le strutture dati e disegnare i grafici.

[1]:
```
%matplotlib inline
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import sklearn
import scipy
```

1.1.1 Caricamento dei dati e preprocessing

- Per l'analisi del problema viene utilizzato il set di dati pubblicato nell'archivio di CiteData.
- Per sicurezza ne è stata caricata una copia su gitHub da cui accingeremo per proseguire.

[2]:
```
import os.path
if not os.path.exists("Dataset.zip"):
    from urllib.request import urlretrieve
    urlretrieve("https://git.io/JUsiU", "Dataset.zip")
    from zipfile import ZipFile
    with ZipFile("Dataset.zip") as f:
        f.extractall()
```

Utilizziamo la funzione di pandas per caricare in un dataframe direttamente i dati dal file excel.

[3]:
```
rice = pd.read_excel("Rice_Osmancik_Cammeo_Dataset.xlsx")
```

Di seguito sono riportate le dimensioni in memoria, il numero di istanze non nulle, il nome e il tipo delle feature che compongono i dati raccolti nel dataset.

[4]:
```
rice.info(memory_usage="deep")
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3810 entries, 0 to 3809
Data columns (total 8 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   AREA          3810 non-null   int64
 1   PERIMETER     3810 non-null   float64
 2   MAJORAXIS     3810 non-null   float64
 3   MINORAXIS     3810 non-null   float64
 4   ECCENTRICITY  3810 non-null   float64
 5   CONVEX_AREA   3810 non-null   int64
 6   EXTENT        3810 non-null   float64
 7   CLASS         3810 non-null   object
dtypes: float64(5), int64(2), object(1)
memory usage: 447.1 KB
```

Notiamo subito che la feature “CLASS” è di tipo generico object, dal nome della feature possiamo già dedurre che si tratta di una varibile di tipo categorica. Utilizzando il metodo unique possiamo verificarlo. Questo metodo ci restituisce un array di valori distinti.

[5]:
```
rice["CLASS"].unique()
```

[5]:
```
array(['Cammeo', 'Osmancik'], dtype=object)
```

Avendo confermato che si tratta di una variabile categorica, è bene procedere alla conversione di tipo passando da object a category. Ciò comporterà un minore utilizzo in memoria dei dati caricati.

Possiamo prendere due strade, o convertire direttamente la colonna interessata, o ricaricare i dati specificando che la colonna “CLASS” è di tipo category. Si procede con la seconda opzione.

[6]:
```
rice = pd.read_excel("Rice_Osmancik_Cammeo_Dataset.xlsx", dtype={"CLASS" :␣
↪"category"})
```

[7]:
```
rice.info(memory_usage="deep")
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3810 entries, 0 to 3809
Data columns (total 8 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   AREA          3810 non-null   int64
 1   PERIMETER     3810 non-null   float64
 2   MAJORAXIS     3810 non-null   float64
 3   MINORAXIS     3810 non-null   float64
 4   ECCENTRICITY  3810 non-null   float64
 5   CONVEX_AREA   3810 non-null   int64
 6   EXTENT        3810 non-null   float64
 7   CLASS         3810 non-null   category
dtypes: category(1), float64(5), int64(2)
memory usage: 212.3 KB
```

Notiamo subito che la quantità di memoria utilizzata si è dimezzata, è passata da circa 447 KB a 212 KB.

Visualizziamo un anteprima dei dati caricati.

[8]:
```
rice.head(5)
```

[8]:

	AREA	PERIMETER	MAJORAXIS	MINORAXIS	ECCENTRICITY	CONVEX_AREA	EXTENT	CLASS
0	15231	525.578979	229.749878	85.093788	0.928882	15617	0.572896	Cammeo
1	14656	494.311005	206.020065	91.730972	0.895405	15072	0.615436	Cammeo
2	14634	501.122009	214.106781	87.768288	0.912118	14954	0.693259	Cammeo
3	13176	458.342987	193.337387	87.448395	0.891861	13368	0.640669	Cammeo
4	14688	507.166992	211.743378	89.312454	0.906691	15262	0.646024	Cammeo

Si rinominano le feature come da convenzione.

[9]:
```
rice.columns = ["area", "perimeter", "major_axis", "minor_axis", "eccentricy",␣
↪"convex_area", "extent", "class"]
```

[10]:
```
rice.head(5)
```

[10]:

	area	perimeter	major_axis	minor_axis	eccentricy	convex_area	extent	class
0	15231	525.578979	229.749878	85.093788	0.928882	15617	0.572896	Cammeo
1	14656	494.311005	206.020065	91.730972	0.895405	15072	0.615436	Cammeo

```
2  14634  501.122009  214.106781   87.768288    0.912118        14954
3  13176  458.342987  193.337387   87.448395    0.891861        13368
4  14688  507.166992  211.743378   89.312454    0.906691        15262

     extent   class
0  0.572896  Cammeo
1  0.615436  Cammeo
2  0.693259  Cammeo
3  0.640669  Cammeo
4  0.646024  Cammeo
```

1.1.2 Significato delle Features

Le feautures sono state ricavate attraverso la rielaborazione delle fotografie scattate ai chicchi di riso, quindi l'unità di misura di riferimento sono i pixel.

Ogni osservazione è descritta da 7 variabili numeriche più un'ottava variabile che ne identifica la classe di appartenenza.

Si riportano le descrizioni delle features:

1. **Area**: restituisce il numero di pixel entro i confini del chicco di riso.
2. **Perimetro**: calcola la circonferenza calcolando la distanza tra i pixel attorno ai bordi del chicco di riso.
3. **Lunghezza dell'asse maggiore**: rappresenta la linea più lunga che può essere tracciata sul chicco di riso, cioè la distanza dell'asse principale.
4. **Lunghezza dell'asse minore**: rappresenta la linea più corta che può essere tracciata sul chicco di riso, cioè la piccola distanza dell'asse.
5. **Eccentricità**: misura quanto è rotonda l'ellisse, che ha gli stessi momenti del chicco di riso.
6. **Area convessa**: restituisce il numero di pixel del guscio convesso più piccolo della regione formata dal chicco di riso.
7. **Estensione**: restituisce il rapporto tra la regione formata dal chicco di riso e i pixel del riquadro di delimitazione.
8. **Classe**: Cammeo e Osmancik (le due varietà di riso)

Parte 1b) Analisi esplorativa

Per le features numeriche, con il metodo `describe` è possibile avere una rappresentazione statistica ottenendo per ciascuna di esse: - Numero di istanze/osservazioni - Media - Deviazione standard - Valore minimo - Percentili (25°, 50° e 75° percentile) - Valore massimo

[11]:
```
rice.describe()
```

[11]:
```
                area    perimeter   major_axis   minor_axis   eccentricy  \
count    3810.000000  3810.000000  3810.000000  3810.000000  3810.000000
mean    12667.727559   454.239180   188.776222    86.313750     0.886871
std      1732.367706    35.597081    17.448679     5.729817     0.020818
min      7551.000000   359.100006   145.264465    59.532406     0.777233
25%     11370.500000   426.144753   174.353855    82.731695     0.872402
50%     12421.500000   448.852493   185.810059    86.434647     0.889050
```

```
75%    13950.000000   483.683746   203.550438    90.143677     0.902588
max    18913.000000   548.445984   239.010498   107.542450     0.948007

        convex_area       extent
count   3810.000000  3810.000000
mean   12952.496850     0.661934
std     1776.972042     0.077239
min     7723.000000     0.497413
25%    11626.250000     0.598862
50%    12706.500000     0.645361
75%    14284.000000     0.726562
max    19099.000000     0.861050
```

- Notiamo, anche da questa panoramica, l'assenza di valori nulli; lo possiamo confermare anche controllando la matrice dei valori.

[12]:
```
rice.isnull().values.any()
```

[12]:
```
False
```

- A prima vista, si potrebbe azzardare, senza conoscere la distribuzione effettiva delle classi, che i chicchi di riso presi in esame abbiano alcune delle caratteristiche morfologiche molto simili tra loro, in quanto il 50% delle instanze ricadono in un range di valori molto concentrati che si avvicinano molto al valore medio.
- Osservando la deviazione standard notiamo che non è molto alta. Notare che le features hanno ordini di grandezza diversi.

Per uniformare la creazione dei grafici che andremo ad analizzare, si sceglie di fissare arbitrariamente i colori alle classi:

[13]:
```
osmancik_color = '#1f77b4' # blu
cammeo_color = '#ff7f0e' # arancione
```

[14]:
```
# Dizionario che associa a ogni classi il suo colore
rice_color_map = {"Cammeo": cammeo_color, "Osmancik": osmancik_color}
```

[15]:
```
# Mappiamo i colori alle classi delle istanze presenti nel dataset
rice_colors = rice["class"].map(rice_color_map)
```

[16]:
```
rice_colors.head(5)
```

[16]:
```
0    #ff7f0e
1    #ff7f0e
2    #ff7f0e
3    #ff7f0e
4    #ff7f0e
Name: class, dtype: category
Categories (2, object): ['#ff7f0e', '#1f77b4']
```

Per le colonne categoriche possiamo verificare la distribuzione dei valori mediante il metodo `value_counts`.

Avendo solamente la feature "class" come categorica possiamo verificare il numero delle istanze suddivise in base alla loro classe di apparentenza.

[17]:
```
rice["class"].value_counts()
```

[17]:
```
Osmancik    2180
Cammeo      1630
Name: class, dtype: int64
```

Visualizziamo lo stesso dato ma in forma percentuale

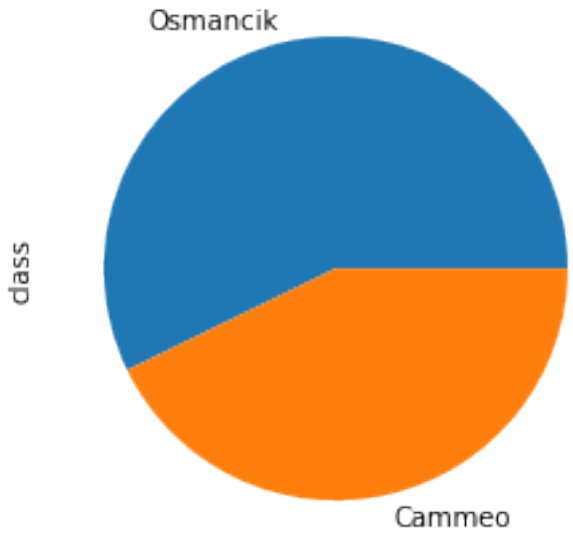
[18]:
```
rice["class"].value_counts(normalize = True) *100
```

[18]:
```
Osmancik    57.217848
Cammeo      42.782152
Name: class, dtype: float64
```

Possiamo notare che la distrubuzione delle classi è abbastanza bilanciata, questo è un bene in quanto in fase di modellazione e valutazione i risultati dovrebbero essere leggermente più affidabili e non molto alterati dallo sbilanciamento delle classi.

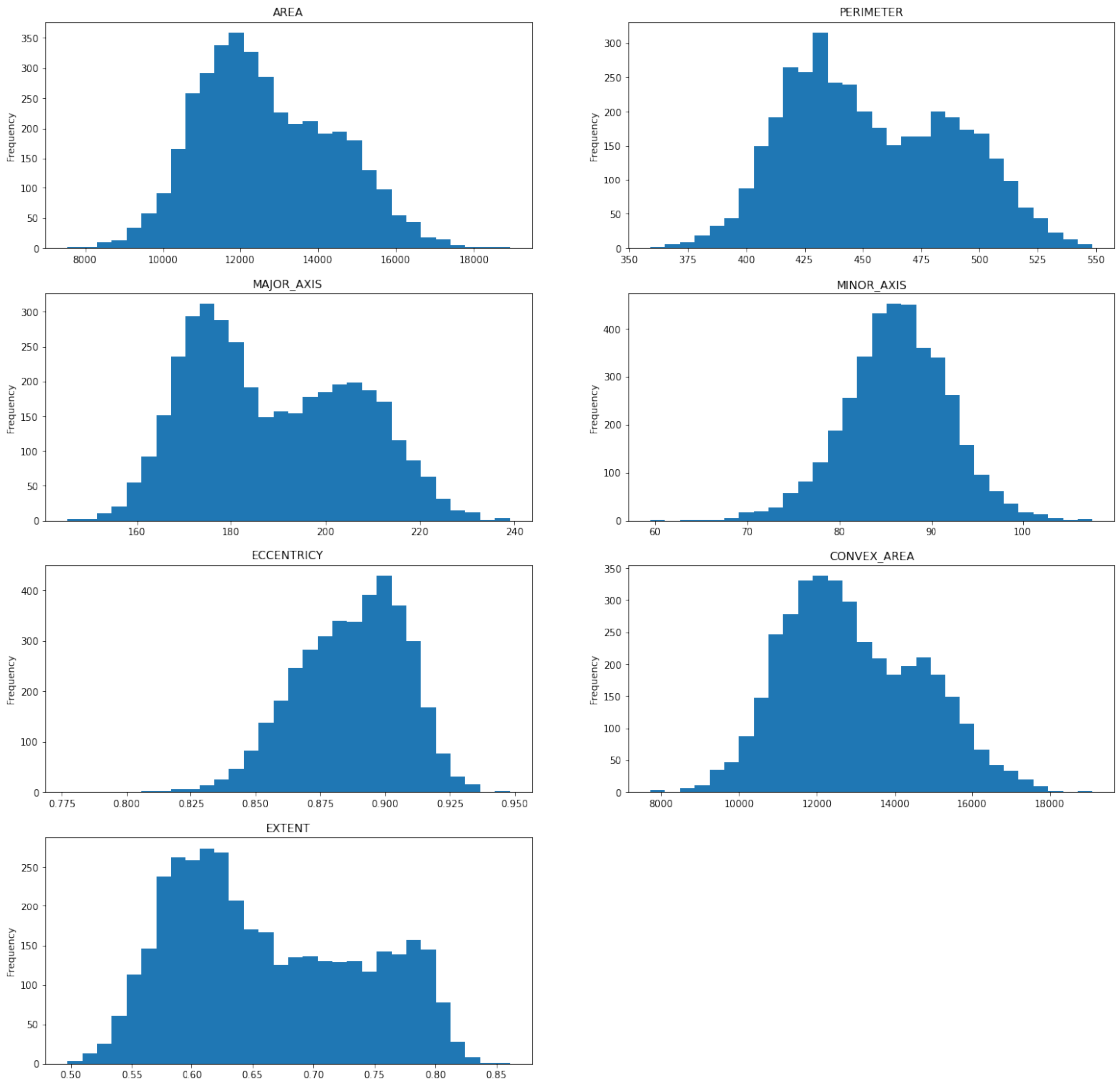
Il seguente grafico conferma quanto detto sopra.

[19]:
```
rice["class"].value_counts().plot.pie(colors=[osmancik_color, cammeo_color]);
```



Rappresentiamo ora la distribuzione dei valori delle singole features al fine di graficare la loro distribuzione rispetto alla loro frequenza.

[20]:
```
plt.figure(figsize=(20, 20))
for n, col in enumerate(["area", "perimeter", "major_axis", "minor_axis",␣
 ↪"eccentricy", "convex_area", "extent"], start=1):
    rice[col].plot.hist(ax=plt.subplot(4, 2, n),title=col.upper(), bins=30)
```

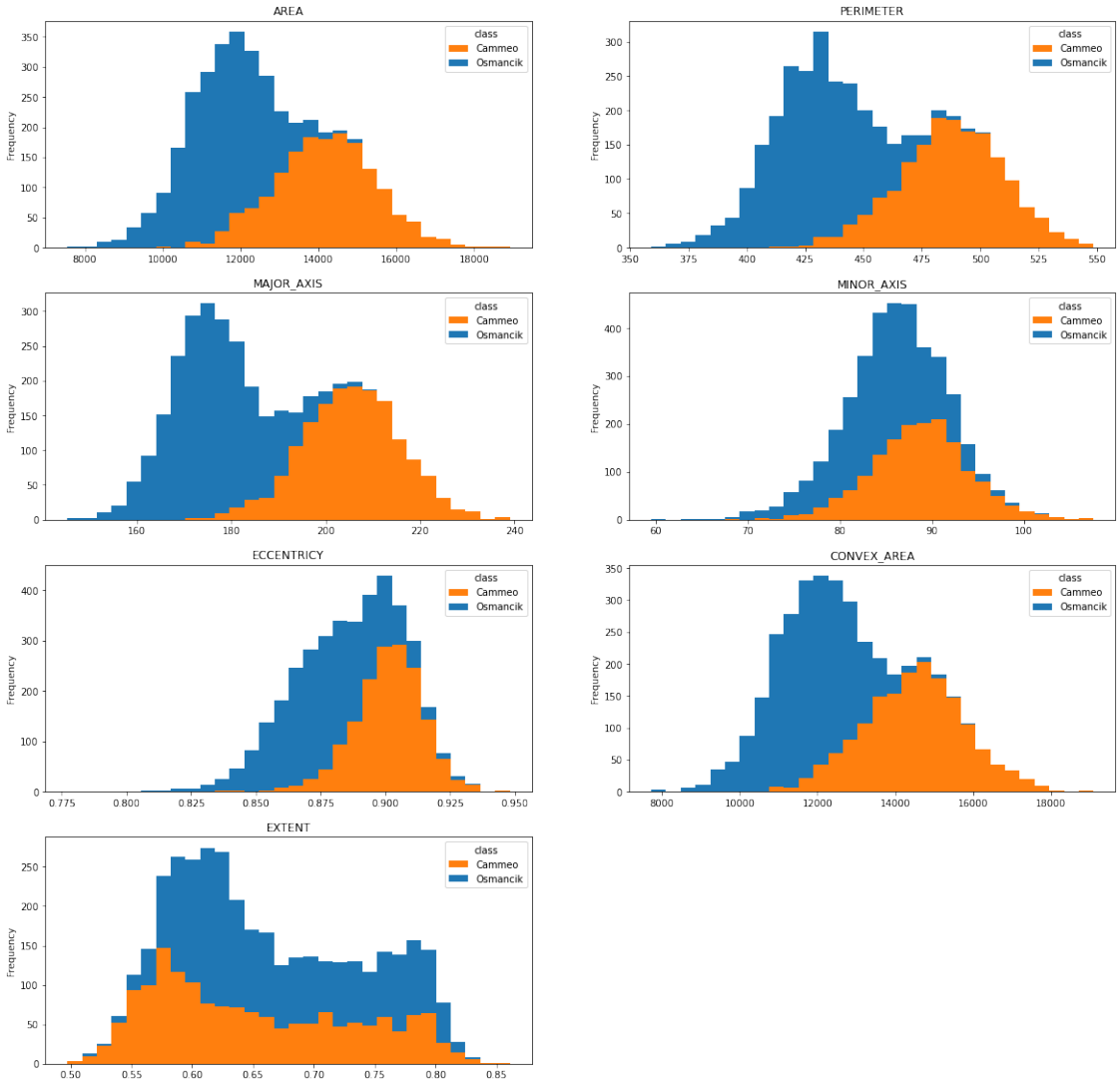


- Da questi grafici osserviamo che alcune delle features (“area”, “minor_axis”, “eccentricy”) hanno una distribuzione concentrata tendente a una distrubizione gaussiana, il che potrebbe far pensare che per quelle caratteristiche le due varietà di riso siano simili;
- Mentre si osserva che le altre feature hanno una distribuzione meno concentrata creando quasi due “campane” o meglio picchi distinti, il che porta a pensare che queste siano le feature che potrebbero caratterizzare di più le due varietà di riso.

Per avere una visione più chiara dell’analisi precedente si ripropongono gli stessi grafici ma con le classi messe in evidenza. Le due classi sono “impilate”, possiamo vedere la distribuzione di

entrambe senza perdere informazione.

[21]:
```
plt.figure(figsize=(20, 20))
for n, col in enumerate(["area", "perimeter", "major_axis", "minor_axis",␣
 ↪"eccentricy", "convex_area", "extent"], start=1):
    rice.pivot(columns="class")[col] \
    .plot.hist(bins=30, stacked=True, ax=plt.subplot(4, 2, n), title=col.
 ↪upper(), color=[cammeo_color, osmancik_color]);
```



Come ipotizzato sopra, in questi grafici possiamo notare con più chiarezza che la maggior parte delle feautures che presentano una distribuzione meno concentrata, creando due "picchi", caratterizzano in maniera abbastanza netta le due varietà di riso.

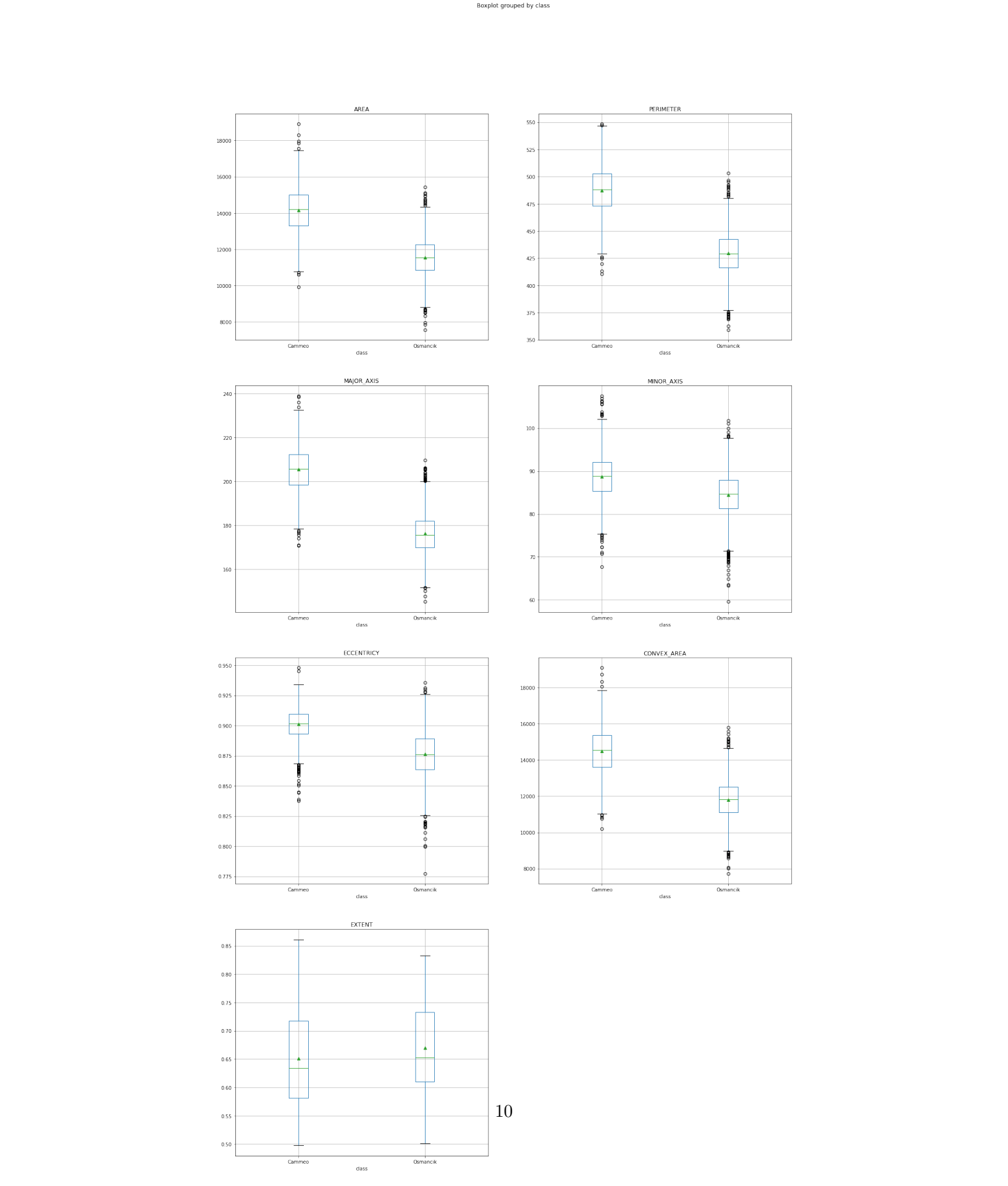
Possiamo notare che le concentrazioni dei valori più alti appartengono alla varietà Cammeo, mentre le concentrazioni di valori più bassi appartengono alla varietà Osmancik.

Diversamente da quanto si era ipotizzato inzialmente, notiamo invece che la distribuzione delle classi per quanto riguarda la feature “extent” è abbastanza omogenea anche se a prima vista si notava la presenza dei due “picchi”.

Stessa cosa per quanto riguarda la distribuzione delle features “eccentricy” e “area” che nonostante abbiano una distribuzione tendente a una gaussiana presentano una differenza abbastanza marcata delle due classi, più si va verso i valori alti più la feature caratterizza la varietà di riso Cammeo dalla varietà Osmancik.

Possiamo confermarlo andando a vedere nel dettaglio come sono distribuite le statistiche di base suddivendole per classi.

[22]:
```
plt.figure(figsize=(20, 40))
for n, col in enumerate(["area", "perimeter", "major_axis", "minor_axis",␣
↪"eccentricy", "convex_area", "extent"], start=1):
    rice.boxplot(column=col, by="class", showmeans=True, ax=plt.subplot(4, 2,␣
↪n))
    plt.title(col.upper())
```


Boxplot grouped by class
AREA
PERIMETER
MAJOR_AXIS
MINOR_AXIS
ECCENTRICY
CONVEX_AREA
EXTENT
Cammeo
Osmancik
class

1.1.3 Esplorazione relazioni fra feature

Utilizzando il metodo `corr()` possiamo visualizzare una tabella con le loro correlazioni. Nella diagonale avremo le correlazioni delle features con se stesse, mentre nelle altre celle avremo le correllazioni di features diverse tra loro.

Si crea un nuovo dataframe copiando "rice" ma sostituendo i valori delle classi con 0 e 1 in modo da poter calcolare anche la correlazione tra le classi e le singole features.

[23]:
```
rice_binary = rice.replace(to_replace=["Osmancik", 'Cammeo'], value=[0, 1])
```

[24]:
```
correlation = rice_binary.corr()
correlation.style.background_gradient(cmap='coolwarm').set_precision(2)
```

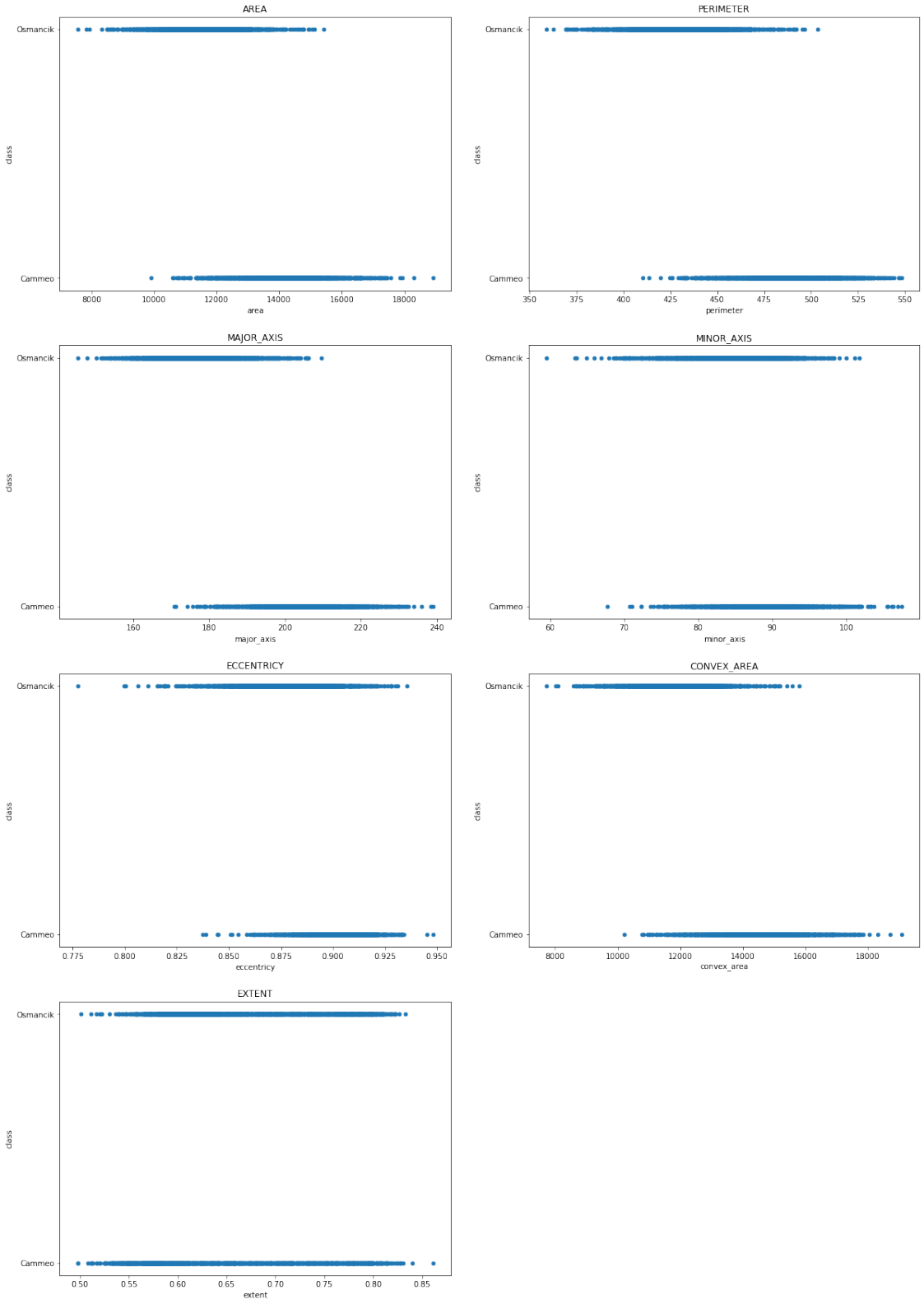
[24]:
```
<pandas.io.formats.style.Styler at 0x1b9241a7ac8>
```

Dalla tabella sopra riportata notiamo che ci sono diverse correlazioni alte tra coppie di features diverse. Ad esempio c'è una forte correlazione tra l'area e il perimetro del chicco di riso, tra l'area e la lunghezza dell'asse maggiore, mentre non c'è una forte correlazione tra l'area e l'estensione, tra l'estensione e l'eccentricità, ecc.

Nella tabella si possono visualizzare che ci sono delle forti correlazioni tra alcune features e la classe di appartenenza.

Visualizziamo graficamente la correlazione tra le feautures e le classi.

[25]:
```
plt.figure(figsize=(20, 30))
for n, col in enumerate(["area", "perimeter", "major_axis", "minor_axis",␣
 ↪"eccentricy", "convex_area", "extent"], start=1):
    rice.plot.scatter(col, "class", ax=plt.subplot(4, 2, n), title=col.upper());
```

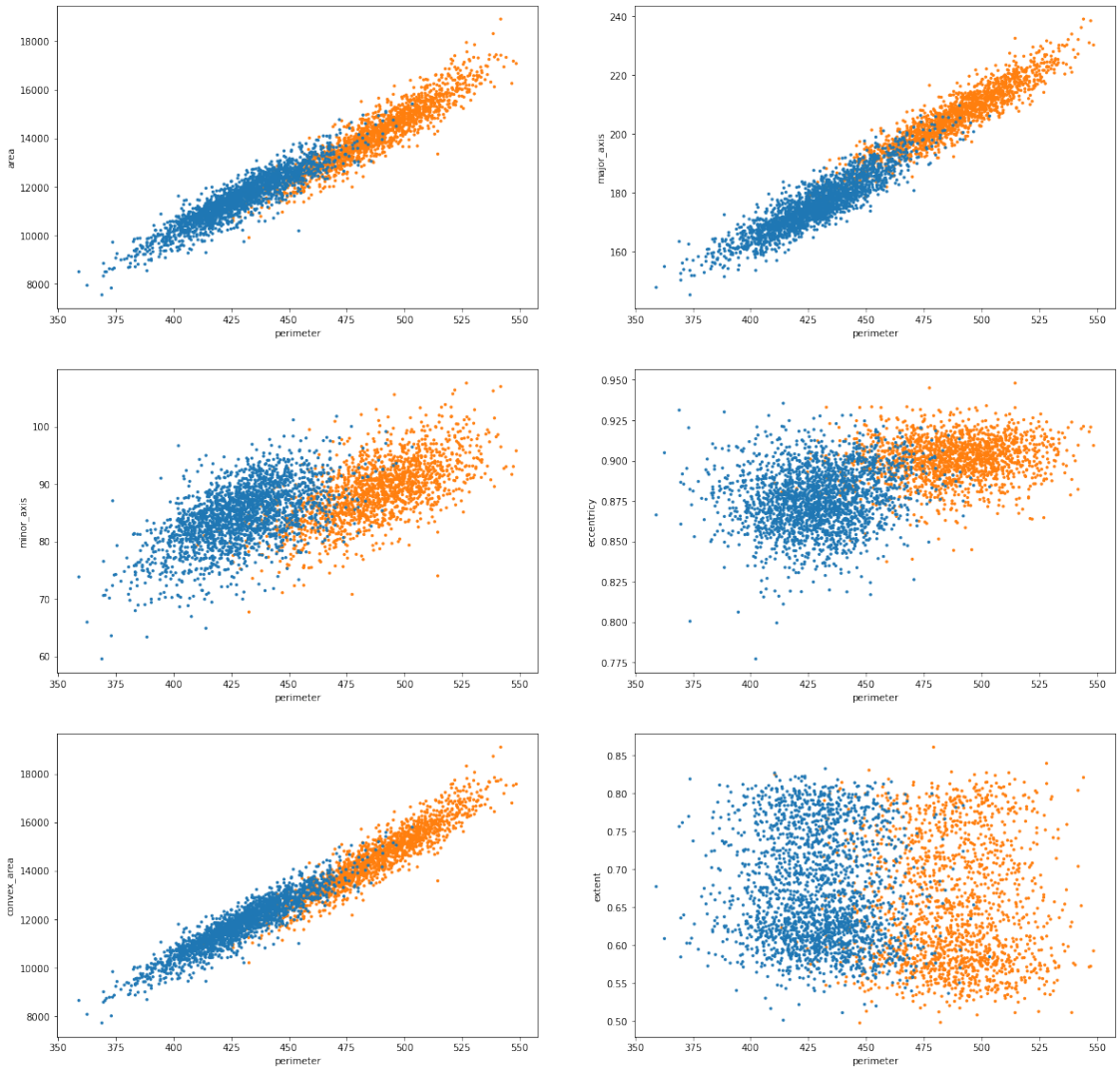

AREA
Osmancik
Cammeo
class
8000
10000
12000
14000
16000
18000
area
PERIMETER
Osmancik
Cammeo
class
350
375
400
425
450
475
500
525
550
perimeter
MAJOR_AXIS
Osmancik
Cammeo
class
160
180
200
220
240
major_axis
MINOR_AXIS
Osmancik
Cammeo
class
60
70
80
90
100
minor_axis
ECCENTRICY
Osmancik
Cammeo
class
0.775
0.800
0.825
0.850
0.875
0.900
0.925
0.950
eccentricy
CONVEX_AREA
Osmancik
Cammeo
class
8000
10000
12000
14000
16000
18000
convex_area
EXTENT
Osmancik
Cammeo
class
0.50
0.55
0.60
0.65
0.70
0.75
0.80
0.85
extent

Come osservato anche durante la fase iniziale di analisi dei dati, alcuni grafici mostrano chiaramente che all'aumentare del valore di riferimento si va più verso una classe (Cammeo) e vicersa (Osmancik), mentre per le features "extent", "minor_axis" la correlazione è quasi nulla e probabilmente influirà poco sul tipo di classe di appartenenza.

Visualizziamo ora la distribuzione nello spazio delle due classi prendendo coppie di features. Si sceglie di prendere arbitrariamente come riferimento per l'asse delle x il perimetro, mentre per l'asse y le restanti features. Definiamo una funzione che poi sarà utilizzata anche successivamente.

[26]:
```
def plot_features_realtionship (X, y=None):
    plt.figure(figsize=(20, 20))
    for n, col in enumerate(["area", "major_axis", "minor_axis", "eccentricy",␣
↪"convex_area", "extent"], start=1):
        if y is None:
            X.plot.scatter("perimeter", col, c=rice_colors, s=5, ax=plt.
↪subplot(3, 2, n))
        else:
            X.plot.scatter("perimeter", col, c=y.map(rice_color_map), s=5,␣
↪ax=plt.subplot(3, 2, n))
```

[27]:
```
plot_features_realtionship(rice)
```



Come già abbiamo notato, anche qui si può osservare che tra il perimetro e alcune features esiste una forte correlazione lineare. Quello che si nota maggiormente in questi grafici è l'esistenza di una distinzione abbastanza evidente tra le due classi.

Possiamo già dire che con una semplice classificazione lineare non riusciremmo in alcun caso a distinguere nettamente le due classi, ma potremmo ottenere un buon risultato.

1.2 Parte 2 - Preprocessing

La parte di preprocessing si occupa di elaborare i dati delle feautures in modo da renderle interpretabili / utilizzabili ai modelli di learning.

Nel nostro caso di studio abbiamo tutte le features numeriche reali, quindi non si ha la necessità di utilizzare la codifica one-hot-encoding, utilizzata per rappresentare in forma vettoriale le variabili categoriche.

Avendo variabili numeriche abbiamo la possibilità di standardizzarle, questa procedura ci permette di mettere sullo stesso piano tutte le nostre features andando a risolvere i problemi di scalabilità dei dati.

Prima di andare a effettuare le trasformazioni delle features, importiamo le librerie necessarie e selezioniamo i dati su cui lavorare: - la variabile y da predire è la classe: Osmancik o Cammeo; - le variabili X le restanti features, le caratteristiche morfologiche.

[28]:
```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import Perceptron
from sklearn.linear_model import LogisticRegression
from sklearn.neural_network import MLPClassifier
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import PolynomialFeatures
import warnings #Per nascondere i convergeWarning dei filtri
warnings.filterwarnings("ignore")
```

[29]:
```
y = rice["class"]
X = rice.drop(columns="class")
```

1.2.1 Hold-Out

Avendo il dataset per intero, dobbiamo ricorrere al metodo `Hold-Out` per suddividere i dati in train set e validation set.

Sul train set andremo ad addestrare i nostri modelli, sul validation set a validarli. Suddividiamo i dati in training set e in validation set con la funzione `train_test_split`

[30]:
```
X_train, X_val, y_train, y_val = train_test_split(
    X, y,             # dati da suddividere
    test_size=1/3,    # proporzione: 2/3 training, 1/3 validation
    random_state=42   # seed per la riproducibilità
)
```

Andiamo a verificare come sono stati suddivisi i dati

[31]:
```
X_train.shape, X_val.shape
```

[31]:
```
((2540, 7), (1270, 7))
```

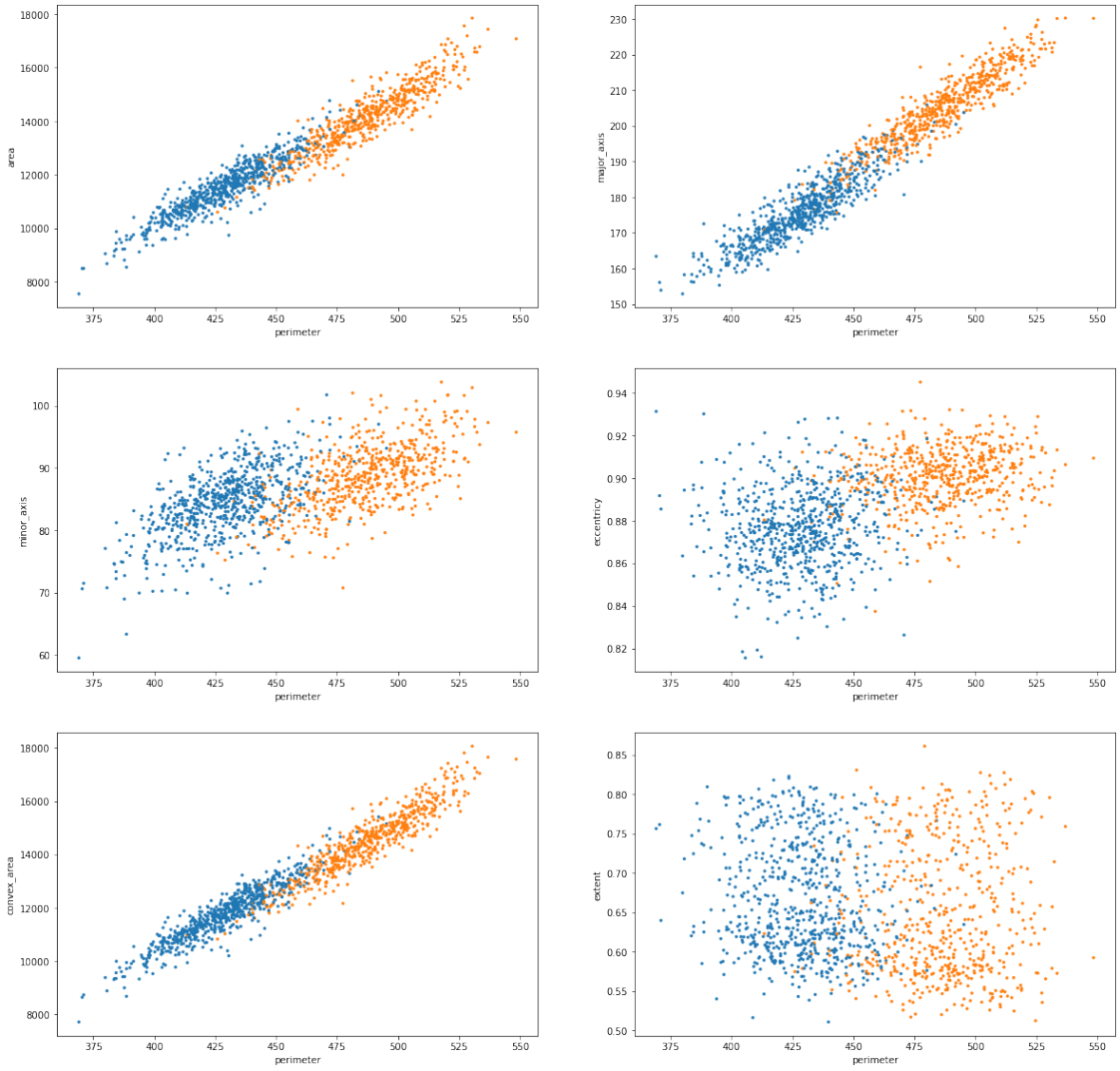
[32]:
```
y_train.shape, y_val.shape
```

[32]:
```
((2540,), (1270,))
```

Notiamo che sono state riservate 2540 osservazioni per il train e 1270 osservazioni per il validation.

Visualizziamo nuovamente la distribuzione delle classi in base alle loro features ma questa volta riferita solamente ai dati presenti validation set.

[33]:
```
plot_features_realtionship(X_val, y_val)
```



Notiamo che le distribuzioni sono rimaste simili a prima ma con i grafici meno popolati data la minor presenza di osservazioni.

1.2.2 Perceptron, Logistic Regression, Multi-layer Perceptron Classifier

Per sondare il terreno, proviamo a classificare le due classi adottando modelli di classificazione lineare in forma semplice.

Il `perceptron` è un algoritmo di apprendimento molto semplice e concettualmente simile alla discesa gradiente i parametri sono inizializzati casualmente si itera il training set: per ogni istanza mal classificata, i parametri vengono aggiornati proporzionalmente ai valori di x e ad un learning rate preimpostato.

La `Logistic Regression` è un modello di classificazione binaria basato sulla regressione lineare

Il `Multi-layer Perceptron Classifier` è un modello di rete neurale semplice il cui output è un iperpiano determinato dalla combinazione lineare delle variabili in input, ognuna con un peso diverso.

Perceptron

- Proviamo un primo approcio generando un modello Perceptron semplice senza la standardizzazione delle variabili. Impostiamo il seed per garantire la riproducibilità degli esperimenti e lo addestriamo sui dati di training.

[34]:
```
model_perceptron = Perceptron(random_state=123)
model_perceptron.fit(X_train, y_train)
```

[34]:
```
Perceptron(random_state=123)
```

Utilizziamo il metodo score per validare l'accuratezza del modello sui dati di validation

[35]:
```
model_perceptron.score(X_val, y_val)
```

[35]:
```
0.6763779527559055
```

Notiamo che con un modello base e senza standardizzazione delle features abbiamo ottenuto uno score del 67%. Un modello per essere accettabile dovrebbe avere uno score di almeno il 70/75%, il punteggio da noi ottenuto è basso.

Logistic Regression

- Facciamo la stessa cosa per il modello Logistic Regression, anche qui impostiamo il seed.

[36]:
```
model_log_reg = LogisticRegression(solver="saga", random_state=123)
model_log_reg.fit(X_train, y_train)
model_log_reg.score(X_val, y_val)
```

[36]:
```
0.7393700787401575
```

Notiamo che con la Logistic Regression abbiamo ottenuto abbastanza al limite, il 74%

MLPClassifier

- Stesso approcio per il modello MLPClassifier

[37]:
```
model_mlp = MLPClassifier(random_state=123)
model_mlp.fit(X_train, y_train)
model_mlp.score(X_val, y_val)
```

[37]:
```
0.5881889763779528
```

In questo caso abbiamo ottenuto un punteggio molto basso.

1.2.3 StandardScaler

Utilizziamo il filtro `StandardScaler` di sklearn per la standardizzazione dei dati. Proviamo a standardizzare le features, ricreiamo i modelli sopra, li riaddestriamo e li valutiamo col metodo score.

[38]:
```
scaler = StandardScaler()
Xn_train = scaler.fit_transform(X_train)
Xn_val = scaler.transform(X_val)
```

Perceptron

- Perceptron semplice con dati stardardizzati.

[39]:
```
model = Perceptron(random_state=123)
model.fit(Xn_train, y_train)
model.score(Xn_val, y_val)
```

[39]:
```
0.9188976377952756
```

Notiamo che applicando la standardizzazione delle feature, abbiamo migliorato di molto lo score del modello rispetto a prima.

Logistic Regression

- Logistic Regression semplice con dati stardardizzati.

[40]:
```
model_prova = LogisticRegression(solver="saga", random_state=123)
model_prova.fit(Xn_train, y_train)
model_prova.score(Xn_val, y_val)
```

[40]:
```
0.9307086614173228
```

Notiamo che applicando la standardizzazione delle feature abbiamo migliorato di molto lo score del modello

MLPClassifier

- MLPClassifier semplice con dati stardardizzati.

[41]:
```
model_prova = MLPClassifier(random_state=123)
model_prova.fit(Xn_train, y_train)
model_prova.score(Xn_val, y_val)
```

[41]:
```
0.931496062992126
```

Notiamo che applicando la standardizzazione delle feature, abbiamo migliorato, anche in questo caso, di molto lo score del modello. Con l'accuratezza del 93% abbiamo ottenuto un modello migliore rispetto a prima.

Osservazioni Abbiamo notato in tutti e tre i modelli che applicando la standardizzazione delle feature abbiamo avuto degli score superiori al 90%. Abbiamo ottenuto dei modelli migliori rispetto a prima, ma che ancora non separano del tutto le due classi.

Con questi passaggi abbiamo dimostrato che in fase di preprocessing conviene standardizzare le feature.

1.2.4 Bilanciamento delle Classi

Proviamo a vedere graficamente come si comporta il Perceptron e se è influenzato dal bilanciamento delle classi. Per poterlo fare prendiamo in considerazione solamente le features "perimeter" e "extent".

Vengono definite le funzioni per rappresentare il modello 2D con le features standardizzate.

[42]:
```
def separator_2d(model, x1):
    w1 = model.coef_[0, 0]
    w2 = model.coef_[0, 1]
    b = model.intercept_[0]
    return (w1/w2) * x1 - (b/w2)
```

[43]:
```
def plot_separator_on_data(X, y, model=None):
    X = np.array(X)
    colors = pd.Series(y).map(rice_color_map)
    plt.scatter(X[:, 0], X[:, 1], c=colors, s=3)
    if model is not None:
        xlim, ylim = plt.xlim(), plt.ylim()
        sep_x = np.linspace(*xlim, 2)
        sep_y = separator_2d(model, sep_x)
        plt.plot(sep_x, sep_y, c="red", linewidth=2)
        plt.xlim(xlim); plt.ylim(ylim)
```

Si crea il dataframe con le due colonne delle features scelte e la serie delle classi di appartenenza

[44]:
```
y = rice["class"]
X2d = rice[["perimeter", "extent"]]
```

Suddividiamo i dati appena creati in train e validation set

[45]:
```
X2d_train, X2d_val, y_train, y_val = train_test_split(
    X2d, y,           # dati da suddividere
    test_size=1/3,    # proporzione: 2/3 training, 1/3 validation
    random_state=42   # seed per la riproducibilità
)
```

Prendiamo i dati e li trasformiamo utilizzando il filtro `StandardScaler()`.

[46]:
```
scaler = StandardScaler()
X2dn_train = scaler.fit_transform(X2d_train)
X2dn_val = scaler.transform(X2d_val)
```

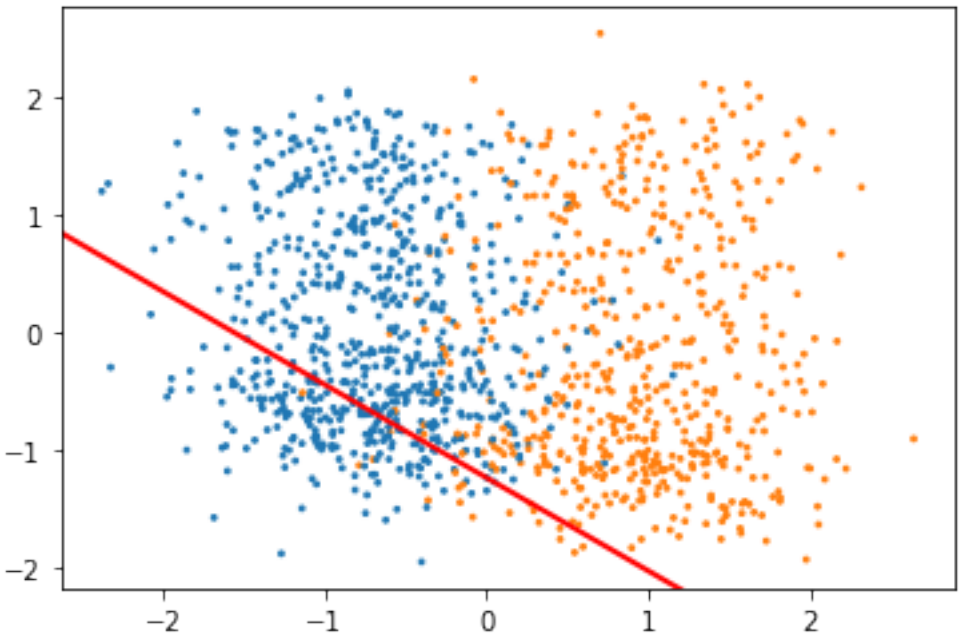
Perceptron

- Dichiariamo un modello Perceptron con il seed impostato per la riprocibilità dei tentativi, senza standardizzazione e lo addestriamo e lo validiamo.

[47]:
```
model = Perceptron(random_state=123)
model.fit(X2d_train, y_train)
model.score(X2d_val, y_val)
```

[47]:
```
0.44803149606299214
```

[48]:
```
# nota: per poter rappresentare graficamente il modello ottenuto è stato␣
 ↪volutamente
#       utilizzato il set di validation standardizzato.
plot_separator_on_data(X2dn_val, y_val, model)
```



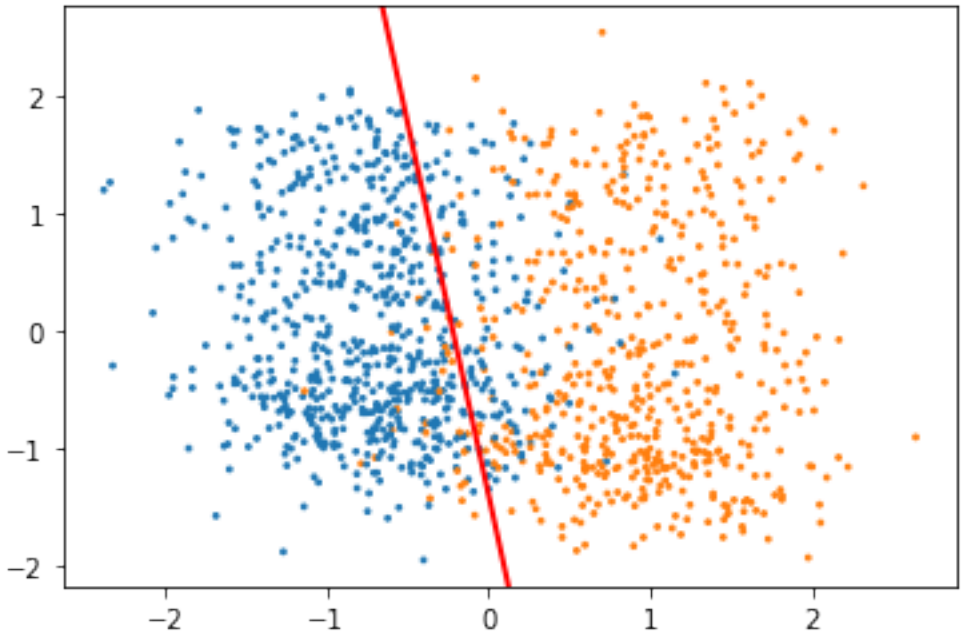
Nel grafico sopra, si nota chiaramente che l'iperpiano di separazione generato dal modello Perceptron semplice, non standardizzato e non bilanciato non separa correttamente le due classi.

Addestriamo il modello sui dati standardizzati.

[49]:
```
model = Perceptron(random_state=123)
model.fit(X2dn_train, y_train)
model.score(X2dn_val, y_val)
```

[49]:
```
0.9125984251968504
```

[50]:
```
plot_separator_on_data(X2dn_val, y_val, model)
```



Nel grafico, ora si nota che con l'accuratezza del 91% abbiamo ottenuto un modello migliore ma che ancora non separa correttamente le due classi. Notiamo che alcune istanze blu sono a destra dell'iperpiano di separazione quando la maggioranza di loro si trovano a sinistra, mentre le istanze arancioni sono separate quasi del tutto con qualche istanza che viene classificata erroneamente come di classe blu.

Il modello in esame ha separato meglio la classe arancione da quella blu in quanto ha più peso nella classificazione delle istanze. Nel caso del modello sopra addestrato, l'accuratezza è stata condizionata anche dalla maggior presenza delle istanze arancioni rispetto a quelle blu.

Proviamo ora a bilanciare il peso delle classi aumentando il numero delle istanze della classe di minoranza attraverso la libreria imblearn, utilizzando `RandomOverSampler`.

RandomOverSampler permette di impostare la strategia di bilanciamento e il seed andando a creare randomicamente altre osservazioni simili a quelle presenti nella classe da bilanciare. Nel nostro caso vogliamo che le classi abbiano lo stesso numero di istanze.

[51]:
```
import imblearn
from imblearn.over_sampling import RandomOverSampler
```

Creiamo l'oggetto RandomOverSampler e impostiamo i parametri

[52]:
```
ros = RandomOverSampler(sampling_strategy='minority', random_state=42)
```

[53]:
```
X2d_resampled, y_resampled = ros.fit_resample(X2d, y)
```

Verifichiamo il numero delle istanze. Notiamo che ora le due classi hanno lo stesso numero.

[54]:
```
y_resampled.value_counts()
```

[54]:
```
Osmancik    2180
Cammeo      2180
Name: class, dtype: int64
```

Dividiamo nuovamente in training e validation set.

[55]:
```
X2d_resampled_train, X2d_resampled_val, y_resampled_train, y_resampled_val =␣
↪train_test_split(
    X2d_resampled, y_resampled,  # dati da suddividere
    test_size=1/3,               # proporzione: 2/3 training, 1/3 validation
    random_state=42              # seed per la riproducibilità
)
```

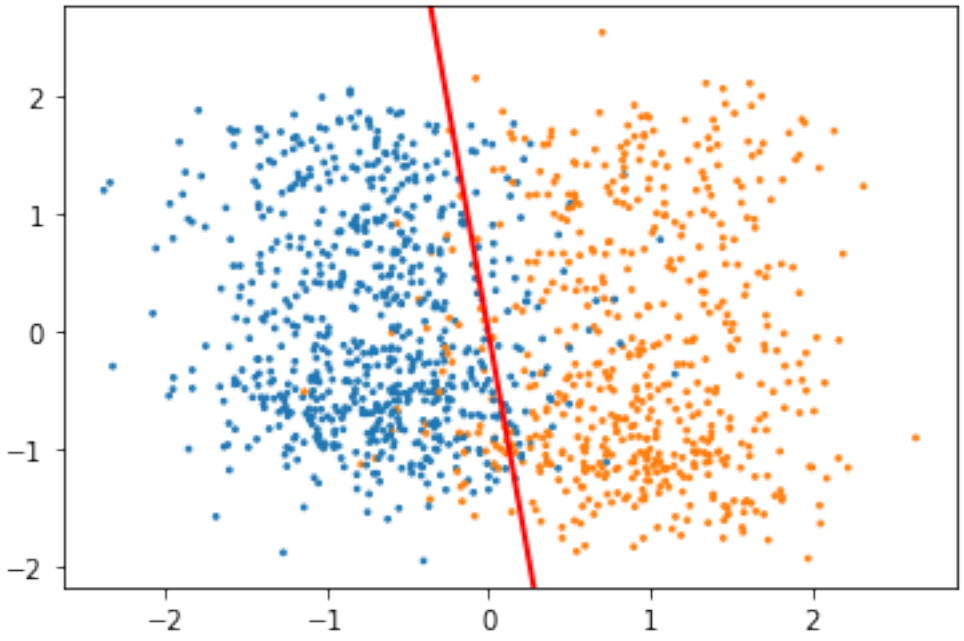
[56]:
```
scaler = StandardScaler()
X2dn_resampled_train = scaler.fit_transform(X2d_resampled_train)
X2dn_resampled_val = scaler.transform(X2d_resampled_val)
```

[57]:
```
model = Perceptron(random_state=123)
model.fit(X2dn_resampled_train, y_resampled_train)
model.score(X2dn_resampled_val, y_resampled_val)
```

[57]:
```
0.9002751031636864
```

Abbiamo ottenuto un risultato leggermente inferiore, siamo passati dall'91 % al 90%. Vediamolo graficamente.

[58]:
```
plot_separator_on_data(X2dn_val, y_val, model)
```



Anche se abbiamo ottenuto uno score leggermente più basso, si nota che l'iperpiano di separazione si è spostato leggermente più destra andando a separare visivamente meglio le due classi.

Questo perché il modello Perceptron nella sua semplicità implementativa, durante l'addestramento, è più sensibile al bilanciamento delle classi, ciò non dovrebbe accadere con gli altri due modelli di learning che abbiamo considerato.

Logistic Regression

- Per correttezza, proviamo ad addestrare il modello Logistic Regression con i training e validation set bilanciati.

[59]:
```
# Modello senza bilanciamento
model = LogisticRegression(solver="saga", random_state=123)
model.fit(X2dn_train, y_train)
model.score(X2dn_val, y_val)
```

[59]: 0.9196850393700787

[60]:
```
# Modello con bilanciamento
model = LogisticRegression(random_state=123)
model.fit(X2dn_resampled_train, y_resampled_train)
model.score(X2dn_resampled_val, y_resampled_val)
```

[60]: 0.8961485557083907

MLPClassifier

- Idem con il MLPClassifier, proviamo ad addestrare il modello con i training e validation set bilanciati.

[61]:
```
# Modello senza bilanciamento
model = MLPClassifier(random_state=123)
model.fit(X2dn_train, y_train)
model.score(X2dn_val, y_val)
```

[61]: 0.9204724409448819

[62]:
```
# Modello con bilanciamento
model = MLPClassifier(random_state=123)
model.fit(X2dn_resampled_train, y_resampled_train)
model.score(X2dn_resampled_val, y_resampled_val)
```

[62]: 0.8968363136176066

Notiamo che in questo caso abbiamo peggiorato le accuratezze dei modelli, siamo passati dal 92% al 90%.

Osservazioni Abbiamo notato che il bilanciamento delle classi, in questo caso, è tollerato meglio con il Perceptron, mentre con gli altri due modelli si ha una perdita di accuratezza.

Essendo i dati, per bilanciare le due classi, generati causualmente si sceglie di non bilanciare il dataset originale. In fase di modellazione, effetturemo comunque, in aggiunta, una ricerca degli iperparametri migliori con le classi bilanciate solamente al modello Perceptron.

1.2.5 Pipeline

Per prima cosa, si vuole utilizzare l'oggetto `Pipeline` importato da sklearn. Pipeline ci consentirà di sfruttare l'interfaccia di sklearn incapuslando le trasformazioni (preprocessing) delle variabili e il modello da applicare. Finora abbiamo visto che è necessaria la standardizzazione.

1.2.6 Regolarizzazione l1 (Lasso)

Proviamo ora ad applicare al modello sopra sia la standardizzazione che la regolarizzazione `l1`. Con la regolarizzazione "l1" saremo in grado di vedere se esistono feature poco rilevanti per l'addrestamento del modello.

Perceptron

- In questo caso riprendiamo il modello Perceptron, addrestriamo due modelli, entrambi con standardizzazione dei dati, ma uno senza regolarizzazione e l'altro con la regolarizzazione "l1".

[63]:
```
model = Pipeline([
    ("scale", StandardScaler()),
    ("linreg", Perceptron(random_state=123))
])
model.fit(X_train, y_train)
model.score(X_val, y_val)
```

[63]:
```
0.9188976377952756
```

Vediamo i pesi delle features senza la regolarizzazione "l1"

[64]:
```
pd.DataFrame(model.named_steps["linreg"].coef_, index=["weight"],␣
 ↪columns=X_train.columns)
```

[64]:
```
            area  perimeter  major_axis  minor_axis  eccentricy  convex_area  \
weight  8.308346  -1.632718   -2.791846    1.475367   -0.145859   -16.951401

          extent
weight -0.928883
```

Con regolarizzazone "l1".

[65]:
```
model = Pipeline([
    ("scale", StandardScaler()),
    ("linreg", Perceptron(random_state=123, penalty="l1", alpha=0.01))
])
```

```
model.fit(X_train, y_train)
model.score(X_val, y_val)
```

[65]: 0.926771653543307

Vediamo i pesi delle features con la regolarizzazione "l1"

[66]:
```
pd.DataFrame(model.named_steps["linreg"].coef_, index=["weight"],␣
↪columns=X_train.columns)
```

[66]:

	area	perimeter	major_axis	minor_axis	eccentricy	convex_area	extent
weight	0.0	-0.626875	-2.303003	0.0	0.0	0.0	0.0

Osservando i parametri appresi dai due modelli osserviamo che effettuando la regolarizzazione "l1" ci sono alcune features che hanno un peso pari a zero.

Osservando invece gli score notiamo che il modello con regolarizzazione ottiene un punteggio inferiore.

Logistic Regression

- Stesso procedimento con la Logistic Regression

[67]:
```
model = Pipeline([
    ("scale", StandardScaler()),
    ("linreg", LogisticRegression(solver="saga", random_state=123))
])
model.fit(X_train, y_train)
model.score(X_val, y_val)
```

[67]: 0.9307086614173228

Vediamo i pesi delle features senza la regolarizzazione "l1"

[68]:
```
pd.DataFrame(model.named_steps["linreg"].coef_, index=["weight"],␣
↪columns=X_train.columns)
```

[68]:

	area	perimeter	major_axis	minor_axis	eccentricy	convex_area	extent
weight	-0.136174	-0.981657	-0.849949	0.361758	-1.27975	-2.098372	-0.106927

Con regolarizzazone "l1".

[69]:
```
model = Pipeline([
    ("scale", StandardScaler()),
    ("linreg", LogisticRegression(solver="saga", random_state=123,␣
 ↪penalty="l1", C=0.01))
])
model.fit(X_train, y_train)
model.score(X_val, y_val)
```

[69]:
```
0.9291338582677166
```

Vediamo i pesi delle features con la regolarizzazione "l1"

[70]:
```
pd.DataFrame(model.named_steps["linreg"].coef_, index=["weight"],␣
 ↪columns=X_train.columns)
```

[70]:
```
         area  perimeter  major_axis  minor_axis  eccentricy  convex_area  \
weight    0.0        0.0   -2.719398         0.0         0.0          0.0

         extent
weight      0.0
```

Osservando i parametri appresi dai due modelli osserviamo che effettuando la regolarizzazione "l1" ci sono alcune features che hanno un peso pari a zero.

Osservando invece gli score notiamo che il modello con regolarizzazione ottiene un punteggio leggermente inferiore, ma è stato addestrato utilizzando meno features.

MLPClassifier

- Per il MLPClassifier non è implementara la regolarizzazione "l1", è implementata di default la regolarizzazione "l2" attraverso il parametro alpha impostato a 0.0001. Anche se concettualmente sono diverse, vogliamo osservare ugualmente i pesi generati dal modello.

[71]:
```
model = Pipeline([
    ("scale", StandardScaler()),
    ("linreg", MLPClassifier(random_state=123))
])
model.fit(X_train, y_train)
model.score(X_val, y_val)
```

[71]:
```
0.931496062992126
```

Vediamo i pesi delle features con la regolarizzazione "l2".

[72]:
```
pd.set_option('display.max_columns', None) # per visualizzare tutte le colonne
pd.DataFrame(model.named_steps["linreg"].coefs_[0], index=X_train.columns,␣
 ↪columns=["Node_" + str(x) for x in range(0,100)])
```

[72]:

	Node_0	Node_1	Node_2	Node_3	Node_4	Node_5
area	0.074635	-0.188826	-0.135493	-0.011772	0.072730	-0.035183
perimeter	0.037504	0.034290	-0.117179	-0.276144	-0.161366	0.167127
major_axis	0.056571	-0.242416	0.116932	0.154921	0.081980	-0.112009
minor_axis	-0.256016	0.050052	-0.205462	-0.239205	-0.203951	0.480292
eccentricy	0.043181	-0.129630	0.255115	-0.161122	0.177046	-0.569053
convex_area	-0.073843	-0.196530	0.256315	-0.267658	0.103531	-0.023161
extent	0.052500	0.307819	0.184012	0.250167	-0.043150	0.112627

	Node_6	Node_7	Node_8	Node_9	Node_10	Node_11
area	0.226563	0.080003	0.125399	-0.072865	-0.162196	0.274387
perimeter	0.237228	-0.020361	0.234651	-0.177575	-0.169645	0.011336
major_axis	-0.125629	0.044881	0.155827	-0.303431	0.075959	0.110673
minor_axis	-0.090668	0.044940	0.187302	-0.085350	-0.063392	0.229973
eccentricy	0.178123	-0.083230	-0.020505	-0.342620	-0.265211	-0.041930
convex_area	0.060038	-0.161734	0.396365	-0.063657	0.003050	0.405853
extent	-0.153831	-0.068696	-0.010421	-0.366724	0.131351	-0.043503

	Node_12	Node_13	Node_14	Node_15	Node_16	Node_17
area	-0.006576	-0.259754	0.016321	0.296888	-0.111492	-0.082007
perimeter	-0.052001	0.070771	-0.004987	0.100618	0.079582	0.073520
major_axis	0.043729	0.034730	0.016166	0.072997	-0.103690	-0.004269
minor_axis	-0.047405	-0.068050	0.088440	0.374582	0.017035	0.229188
eccentricy	0.282519	-0.061465	-0.042429	0.038702	-0.079661	-0.054761
convex_area	0.318506	-0.191753	0.017615	0.318471	0.010805	0.333711
extent	-0.111723	0.203725	-0.235118	-0.063083	-0.041842	0.128765

	Node_18	Node_19	Node_20	Node_21	Node_22	Node_23
area	-0.092285	-0.044934	0.012219	0.191911	0.070963	0.010055
perimeter	-0.305793	0.338249	0.212695	-0.131734	-0.154297	-0.038962
major_axis	0.042799	-0.022143	-0.006893	0.232359	-0.170480	0.127371
minor_axis	0.021927	-0.131777	-0.309778	0.045808	0.123908	-0.186646
eccentricy	-0.187749	0.375251	0.097765	-0.107390	0.180286	0.120469
convex_area	-0.198194	0.044378	0.098962	0.055727	0.134795	-0.259870
extent	0.315086	-0.251008	0.155362	0.264050	0.018908	0.195111

	Node_24	Node_25	Node_26	Node_27	Node_28	Node_29
area	0.097523	-0.055061	-0.161784	-0.110297	-0.063305	-0.001180
perimeter	0.212631	0.203651	-0.100612	0.013323	0.321375	-0.097541
major_axis	0.214896	-0.245138	-0.035420	-0.013327	0.100211	-0.308646
minor_axis	-0.192027	0.024527	-0.103680	0.175789	-0.235747	0.362051
eccentricy	-0.073105	0.100296	0.252912	-0.078929	0.259217	-0.199147
convex_area	-0.048416	-0.138852	-0.198433	-0.167353	0.319078	-0.071610
extent	0.185027	-0.231209	0.036848	0.071257	-0.145397	-0.266517

	Node_30	Node_31	Node_32	Node_33	Node_34	Node_35
area	-0.159304	-0.091775	-0.109359	0.158712	-0.059554	-0.099160

```
perimeter    0.135911 -0.184862 -0.128604  0.272368  0.212263 -0.160342
major_axis  -0.089348 -0.273207 -0.279947  0.281935  0.148704 -0.179442
minor_axis   0.110791  0.264921  0.267714  0.250229 -0.079214 -0.132982
eccentricy  -0.068950 -0.344127 -0.385792 -0.021739 -0.095879 -0.066580
convex_area -0.183576  0.149286 -0.105366  0.020668  0.029590 -0.026580
extent       0.021065 -0.154640 -0.201785  0.079872  0.112538 -0.014620

               Node_36   Node_37   Node_38   Node_39   Node_40   Node_41  \
area        -0.001075  0.325776  0.218981 -0.027581  0.036654 -0.274169
perimeter    0.057897 -0.019474 -0.046902  0.217039 -0.298878 -0.205635
major_axis  -0.022172  0.360723  0.009352  0.235123 -0.299314 -0.054707
minor_axis   0.004014  0.185535 -0.030657 -0.198481  0.186105  0.172691
eccentricy   0.264396 -0.078332  0.013752  0.190239 -0.119932 -0.112627
convex_area -0.078325  0.221924  0.086359  0.251278 -0.216434 -0.181418
extent      -0.215666  0.060742  0.221648 -0.090972 -0.320116  0.235217

               Node_42   Node_43   Node_44   Node_45   Node_46   Node_47  \
area        -0.153646 -0.117018  0.161556 -0.153249 -0.025795  0.311646
perimeter    0.150104 -0.062688 -0.079638  0.068713  0.152290 -0.124629
major_axis   0.175363 -0.134332 -0.230677 -0.066685  0.240163  0.006120
minor_axis  -0.114174 -0.004722 -0.124621 -0.167379 -0.063379  0.005898
eccentricy   0.101345 -0.182557 -0.261262  0.144395  0.193563 -0.035962
convex_area -0.003998 -0.051901 -0.143525  0.093555  0.136790 -0.094457
extent       0.318083  0.078779 -0.339205 -0.096318  0.104654  0.280122

               Node_48   Node_49   Node_50   Node_51   Node_52   Node_53  \
area        -0.002683  0.214722 -0.172144  0.118361  0.012068  0.030220
perimeter    0.135686  0.228595 -0.130704  0.112323 -0.306840 -0.039338
major_axis   0.185604  0.229260 -0.116696 -0.082763 -0.016372  0.023439
minor_axis   0.120018  0.300349  0.090376  0.215909 -0.059443 -0.122509
eccentricy   0.077312 -0.165846 -0.170298  0.083178 -0.145575  0.154664
convex_area  0.094848  0.408301  0.044161 -0.215382 -0.136907  0.218940
extent      -0.157218 -0.070930 -0.023257 -0.186947 -0.290514  0.141140

               Node_54   Node_55   Node_56   Node_57   Node_58   Node_59  \
area        -0.087927 -0.126159 -0.144523  0.055657  0.128787  0.044595
perimeter    0.003091 -0.263997 -0.105155  0.185871  0.014702 -0.047474
major_axis  -0.232337  0.173311 -0.117843  0.190954 -0.297044 -0.124094
minor_axis  -0.163533 -0.116890 -0.145075 -0.261952  0.037144  0.011537
eccentricy   0.081102 -0.162466 -0.465424 -0.195896 -0.006434  0.127701
convex_area -0.254882 -0.009973 -0.290512  0.099668 -0.193547 -0.150021
extent       0.237752 -0.327897 -0.142607  0.113166  0.160053  0.271333

               Node_60   Node_61   Node_62   Node_63   Node_64   Node_65  \
area         0.033999  0.020210  0.017242  0.096815  0.145557 -0.288861
perimeter   -0.144507  0.050670 -0.018393 -0.248983  0.113480  0.061473
major_axis  -0.069404 -0.277697  0.181492 -0.172316  0.019426 -0.206860
```

```
minor_axis  -0.078694 -0.091984  0.066361 -0.039117 -0.188111  0.018594
eccentricy  -0.032892 -0.184779 -0.036704  0.077551  0.189405 -0.181552
convex_area -0.072156 -0.296403  0.061540 -0.065556  0.254531 -0.135029
extent      -0.255604  0.273432 -0.116647  0.026967 -0.084511  0.311370

              Node_66   Node_67   Node_68   Node_69   Node_70   Node_71  \
area         0.089757 -0.095814 -0.169485 -0.035996 -0.157662  0.132889
perimeter   -0.207344  0.268019  0.160969 -0.037287  0.176867 -0.444715
major_axis  -0.248705 -0.052314 -0.162009 -0.017287 -0.051493 -0.209925
minor_axis  -0.047918 -0.115810  0.259067  0.048278  0.207956  0.028613
eccentricy   0.067835 -0.070509 -0.327982 -0.035343 -0.059176 -0.033007
convex_area -0.316940 -0.080640  0.096311 -0.102931  0.035936 -0.258934
extent       0.129613  0.010272  0.074374  0.167309 -0.299721 -0.405239

              Node_72   Node_73   Node_74   Node_75   Node_76   Node_77  \
area        -0.038595  0.218454 -0.209808  0.100367 -0.056664 -0.084785
perimeter    0.079141  0.216905  0.277359 -0.117553 -0.352221 -0.099472
major_axis  -0.376249  0.027924  0.100222 -0.114778 -0.055532 -0.042543
minor_axis  -0.071350  0.212421  0.132615 -0.085870  0.092725  0.080848
eccentricy  -0.228924 -0.041306  0.344191 -0.185673 -0.029909 -0.191662
convex_area -0.301892  0.383641  0.064205  0.243929 -0.118355  0.251529
extent       0.395713 -0.022196 -0.151838 -0.071907  0.319812 -0.090153

              Node_78   Node_79   Node_80   Node_81   Node_82   Node_83  \
area        -0.149093  0.058582 -0.114249  0.021989  0.028811  0.211495
perimeter   -0.037881 -0.090136 -0.021380 -0.297639 -0.128041  0.283163
major_axis   0.061237 -0.281009  0.126964 -0.306172 -0.160303  0.302247
minor_axis   0.193204 -0.153535 -0.093375  0.093143  0.074179  0.344889
eccentricy   0.090569 -0.196526 -0.256289  0.046652  0.099751  0.112249
convex_area  0.270769  0.119144 -0.012351 -0.004651  0.108006  0.410168
extent       0.180018 -0.017977  0.128767 -0.328521 -0.006995 -0.054566

              Node_84   Node_85   Node_86   Node_87   Node_88   Node_89  \
area         0.089729  0.102970 -0.047729 -0.230606 -0.009783 -0.050606
perimeter   -0.014529  0.089480 -0.292758  0.092833  0.264173 -0.043419
major_axis  -0.351632  0.039048  0.196141  0.243027  0.191290  0.151131
minor_axis   0.213513 -0.192085  0.068872  0.054434  0.163426 -0.211350
eccentricy  -0.425474 -0.158088 -0.097943  0.004155  0.205195 -0.265326
convex_area -0.027832  0.018621 -0.275955  0.094416  0.290582 -0.219385
extent      -0.216325  0.270050  0.240299 -0.098862 -0.190842 -0.349967

              Node_90   Node_91   Node_92   Node_93   Node_94   Node_95  \
area         0.103239  0.179177 -0.083820  0.238734  0.137383  0.100474
perimeter   -0.243939  0.161041  0.011811  0.151408 -0.132211  0.132809
major_axis  -0.115600  0.260114 -0.009930  0.293641 -0.078133  0.306306
minor_axis  -0.181110 -0.160706  0.174137  0.098273 -0.040046 -0.109611
eccentricy  -0.076326  0.296437  0.283054 -0.083047 -0.151858  0.338943
```

```
convex_area -0.116868  0.018460  0.037332 -0.015957  0.105977  0.054603
extent      -0.347086  0.049183 -0.119626  0.042136 -0.035913  0.054645

              Node_96   Node_97   Node_98   Node_99
area        -0.226210 -0.078282 -0.181584 -0.071362
perimeter   -0.151595  0.046299  0.010330 -0.144037
major_axis   0.170416  0.098815 -0.178368  0.092233
minor_axis  -0.161974 -0.048094 -0.182888  0.134732
eccentricy   0.233651  0.129676 -0.000044  0.189949
convex_area  0.119286 -0.035480 -0.238223 -0.173066
extent       0.178161  0.027950 -0.029819  0.198942
```

Osservazioni Abbiamo notato che la regolarizzazione "l1" in alcuni casi tende a peggiorare leggermente lo score, in altri lo migliora considerando che viene addestrato un modello sulla base di meno features.

In fase di modellazione si terrà conto di testare la ricerca degli iperparametri migliori anche attraverso la regolarizzazione "l1".

1.2.7 PolynomialFeatures

Ora proviamo a introdurre il filtro `PolynomialFeatures`. Questo filtro permette di generare una nuova matrice di features costituita da tutte le combinazioni polinomiali delle stesse con grado inferiore o uguale al grado specificato.

Perpeptron Con filtro PolynomialFeatures e grado 2.

[73]:
```
model = Pipeline([
    ("scale", StandardScaler()),
    ("poly", PolynomialFeatures(degree=2, include_bias=False)),
    ("linreg", Perceptron(random_state=123))
])
model.fit(X_train, y_train)
model.score(X_val, y_val)
```

[73]:
```
0.9110236220472441
```

Logistic Regression Con filtro PolynomialFeatures e grado 2.

[74]:
```
model = Pipeline([
    ("scale", StandardScaler()),
    ("poly", PolynomialFeatures(degree=3, include_bias=False)),
    ("linreg", LogisticRegression(solver="saga", random_state=123))
])
model.fit(X_train, y_train)
model.score(X_val, y_val)
```

[74]: 0.9307086614173228

MLPClassifier Con filtro PolynomialFeatures e grado 2.

[75]:
```
model = Pipeline([
    ("scale", StandardScaler()),
    ("poly", PolynomialFeatures(degree=2, include_bias=False)),
    ("linreg", MLPClassifier(random_state=123))
])
model.fit(X_train, y_train)
model.score(X_val, y_val)
```

[75]: 0.9244094488188976

Osservazioni Notiamo che, rispetto ai modelli senza il filtro PolynomialFeatures, non abbiamo guadagnato in termini di score, ma siamo riusciti a mantenere la stessa accuratezza.

In fase di modellazione si terrà conto di testare la ricerca degli iperparametri migliori anche attraverso l'uso PolynomialFeatures.

1.3 Parte 3 - Modellazione

Importiamo le librerie per la modellazione.

[76]:
```
from sklearn.model_selection import KFold
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import confusion_matrix
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import  f1_score
```

Procederemo creando diversi modelli di classificazione tenendo conto quanto osservato in fase di preprocessing, ovvero lavorando con le features standardizzate.

Per poter garantire la riproducibilità dei risultati e confrontare sullo stesso piano i modelli che genereremo, fissiamo i seed arbitrariamente.

1.3.1 Grid Search e Strified K-fold Cross Validation, Nested Cross-Validation

Tramite **Grid Search** e **Strified K-fold cross validation** verranno generati modelli differenti per individuare gli iperparametri migliori. Durante la fase di studio sono state effettuate e testate diverse combinazioni di iperparametri. Per ridurre i tempi di addestramento verranno riportate solamente quelle combinazioni che hanno generato i `mean_test_score` migliori.

Dovendo addestrare un modello a riconoscere delle classi, è opportuno che le proporzioni di ciascuna classe nei fold siano il più possibile equamente distribuite. Si decide di utilizzare `StratifiedKFold` (variante di `KFold`) che garantisce uguale distribuzione delle classi tra un fold e l'altro.

Creiamo due oggetti Strified K-fold, uno per la cross fold interna e uno per quella esterna.

[77]:
```
outer_skf_cv = StratifiedKFold(3, shuffle=True, random_state=888) #Esterna
inner_skf_cv = StratifiedKFold(5, shuffle=True, random_state=888) #Interna
```

Definiamo una funzione per utilizzare la nested cross-validation per generare k fold "esterni" su tutti i dati disponibili e che per ciascuno si esegua il tuning degli iperparametri con una cross validation "interna" usando le parti di training dei fold esterni.

[78]:
```
def nested_cv(model, grid, X=X, y=y):
    results = []
    grid_searchs = []
    for train_indices, val_indices in outer_skf_cv.split(X, y):
        gs = GridSearchCV(model, grid, cv=inner_skf_cv)
        gs.fit(X.iloc[train_indices], y.iloc[train_indices])
        score = gs.score(X.iloc[val_indices], y.iloc[val_indices])
        results.append(score)
        grid_searchs.append(gs)
    return results, grid_searchs
```

1.3.2 Perceptron

Viene riproposto nuovamente il Perceptron, questa volta cercando gli iperparametri migliori attraverso la nested stratified cross-fold validation e la grid search.

Creiamo il modello e la grid.

[79]:
```
perceptron_model = Pipeline([
    ("scaler", StandardScaler()),
    ("poly", None),
    ("lr", Perceptron(random_state=123))
])
```

[80]:
```
perceptron_grid = [
    {
        "poly" : [None],
        "lr__penalty" : ["none"]
    },
    {
        "poly" : [PolynomialFeatures(include_bias=False)],
        "poly__degree" : [2, 3],
        "lr__penalty" : ["none"]
    },
    {
        "poly" : [None],
        "lr__penalty": ["l2", "l1", "elasticnet"],
        "lr__alpha": np.logspace(-4, 3, 8)
    },
```

```
    {
        "poly" : [PolynomialFeatures(include_bias=False)],
        "poly__degree" : [2, 3],
        "lr__penalty": ["l2", "l1", "elasticnet"],
        "lr__alpha": np.logspace(-4, 3, 8),
    }
]
```

[81]:
```
%%time
perceptron_score, perceptron_gs = nested_cv(perceptron_model, perceptron_grid)
```

```
Wall time: 43.8 s
```

Visualizziamo la configurazione (gli iperparametri migliori) del modello che ha ottenuto lo score migliore.

[82]:
```
perceptron_best_gs = perceptron_gs[np.argmax(perceptron_score)]
perceptron_best_param = perceptron_best_gs.best_params_
perceptron_best_param
```

[82]:
```
{'lr__alpha': 0.01, 'lr__penalty': 'l1', 'poly': None}
```

Si riporta il ranking score del modello che ha ottenuto il punteggio migliore.

[83]:
```
pd.DataFrame(perceptron_best_gs.cv_results_).sort_values("rank_test_score").
↪head(5)
```

[83]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_lr__penalty	param_poly	param_poly__degree	param_lr__alpha
10	0.014599	0.001018	0.005601	0.001357	l1	None	NaN	0.01
1	0.020199	0.002785	0.007000	0.000894	none	PolynomialFeatures(include_bias=False)	2	NaN
3	0.016799	0.001470	0.006402	0.000491	l2	None	NaN	0.0001
5	0.012996	0.000893	0.005403	0.000799	elasticnet	None	NaN	0.0001
35	0.029600	0.002061	0.007002	0.002099	l1	PolynomialFeatures(include_bias=False)	2	0.001

```
                                                 params  split0_test_score  \
10  {'lr__alpha': 0.01, 'lr__penalty': 'l1', 'poly…           0.921260
1   {'lr__penalty': 'none', 'poly': PolynomialFeat…           0.911417
3   {'lr__alpha': 0.0001, 'lr__penalty': 'l2', 'po…           0.903543
5   {'lr__alpha': 0.0001, 'lr__penalty': 'elasticn…           0.903543
35  {'lr__alpha': 0.001, 'lr__penalty': 'l1', 'pol…           0.907480

    split1_test_score  split2_test_score  split3_test_score  \
10           0.891732           0.907480           0.942913
1            0.893701           0.923228           0.925197
3            0.874016           0.915354           0.931102
5            0.874016           0.915354           0.931102
35           0.897638           0.901575           0.937008

    split4_test_score  mean_test_score  std_test_score  rank_test_score
10           0.937008         0.920079        0.018824                1
1            0.921260         0.914961        0.011639                2
3            0.940945         0.912992        0.023338                3
5            0.940945         0.912992        0.023338                3
35           0.893701         0.907480        0.015450                5
```

Accuratezza del modello Perceptron Si riportano gli score calcolati dalla funzione nested_cv

[84]:
```
perceptron_score
```

[84]:
```
[0.9173228346456693, 0.9133858267716536, 0.8700787401574803]
```

Notiamo che con il modello che ha ottenuto lo score migliore ha una percentuale di accuratezza del 92%. La media di tutti e tre i risultati è pari al 90%

[85]:
```
np.mean(perceptron_score)
```

[85]:
```
0.9002624671916011
```

Analizziamo il modello che ha ottenuto lo score migliore con le altre metriche di accuratezza.

Estraiamo le predizioni e visualizziamo la matrice di confusione.

[86]:
```
perceptron_preds = perceptron_best_gs.predict(X_val)
perceptron_conf_matrix = confusion_matrix(y_val, perceptron_preds)
```

[87]:
```
perceptron_cm_df = pd.DataFrame(perceptron_conf_matrix, columns =␣
↪["predicted_"+ x for x in perceptron_best_gs.classes_],
             index = ["real_"+x for x in perceptron_best_gs.classes_])
perceptron_cm_df
```

[87]:
```
              predicted_Cammeo  predicted_Osmancik
real_Cammeo                537                  32
```

```
real_Osmancik                58                  643
```

Notiamo che non tutte le istanze della classe Cammeo sono state idendificate correttamente, e quindi qualche chicco di riso della varietà Cammeo è stato erroneamente classificato come facente parte della classe Osmancik. Stessa cosa il contrario, alcune istanze della classe Osmancik sono state classificate erroneamente come facenti parte della classe Cammeo.

La `Precision` indica la percentuale di istanze classificate correttamente come facenti parte di una classe e che sono realmente tali.

[88]:
```
perceptron_precision = pd.DataFrame(precision_score(y_val, perceptron_preds,␣
↪average = None),
             columns=["Precision_score"], index=perceptron_best_gs.classes_)
perceptron_precision
```

[88]:
```
          Precision_score
Cammeo           0.902521
Osmancik         0.952593
```

Possiamo vedere che la classe Cammeo è quella che ne ha risentito di più 90%, mentre la classe Osmancik ha ottenuto una precision alta, il 95%.

La `Recall` misura la sensibilità del modello, ci dà l'indicazione di quante istanze reali di una classe sono state rilevate essere tali dal modello

[89]:
```
perceptron_recall = pd.DataFrame(recall_score(y_val, perceptron_preds, average␣
↪= None),
                                 columns=["Recall_score"],␣
↪index=perceptron_best_gs.classes_)
perceptron_recall
```

[89]:
```
          Recall_score
Cammeo        0.943761
Osmancik      0.917261
```

Notiamo in questo caso che entrambe la classe Cammeo ha ottenuto una recall del 94%, mentre la classe Osmancik del 92%.

Ora misuriamo l'`F1_Score` che è la misura di accuratezza che combina insieme la precision e la recall, ovvero la media armonica tra precision e recall, se la differenza tra le due misure è molto forte la media armonica sarà più vicino a quella bassa.

[90]:
```
perceptron_f1_measure = pd.DataFrame(f1_score(y_val, perceptron_preds, average␣
↪= None),
                                     columns=["F1_Measure_score"],␣
↪index=perceptron_best_gs.classes_)
perceptron_f1_measure
```

[90]:
```
           F1_Measure_score
Cammeo             0.922680
Osmancik           0.934593
```

Notiamo che abbiamo ottenuto con il Perceptron una buona accuratezza. La media tra le due F1_measure si aggira intorno al 93%

[91]:
```
perceptron_f1_mean_measure = f1_score(y_val, perceptron_preds, average =␣
↪"macro")
perceptron_f1_mean_measure
```

[91]:
```
0.928636717813474
```

1.3.3 Perceptron pesato

Per verificare meglio quanto e se influisce il bilanciamento delle classi nel Perceptron, viene riproposto nuovamente il modello e la grid precedenti.

Prendiamo i dataset X e y per effettuare il bilanciamento delle classi.

[92]:
```
ros = RandomOverSampler(sampling_strategy='minority', random_state=42)
```

[93]:
```
X_resampled, y_resampled = ros.fit_resample(X, y)
```

Verifichiamo il numero delle istanze. Notiamo che ora le due classi hanno lo stesso numero.

[94]:
```
y_resampled.value_counts()
```

[94]:
```
Osmancik    2180
Cammeo      2180
Name: class, dtype: int64
```

Creiamo il modello e la grid. La grid la copiamo dal modello Perceptron non pesato.

[95]:
```
balanced_perceptron_model = Pipeline([
    ("scaler", StandardScaler()),
    ("poly", None),
    ("lr", Perceptron(random_state=123))
])
```

[96]:
```
balanced_perceptron_grid = perceptron_grid
```

[97]:
```
%%time
balanced_perceptron_score, balanced_perceptron_gs =␣
↪nested_cv(balanced_perceptron_model, balanced_perceptron_grid, X_resampled,␣
↪y_resampled)
```

```
Wall time: 39.1 s
```

Visualizziamo la configurazione (gli iperparametri migliori) del modello che ha ottenuto lo score migliore.

[98]:
```
bal_perceptron_best_gs = balanced_perceptron_gs[np.
↪argmax(balanced_perceptron_score)]
bal_perceptron_best_param = bal_perceptron_best_gs.best_params_
bal_perceptron_best_param
```

[98]:
```
{'lr__alpha': 0.1, 'lr__penalty': 'l1', 'poly': None}
```

Si riporta il ranking score del modello che ha ottenuto il punteggio migliore.

[99]:
```
pd.DataFrame(bal_perceptron_best_gs.cv_results_).sort_values("rank_test_score").
↪head(5)
```

[99]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	\
47	0.019199	0.004534	0.005197	0.001158	
13	0.010398	0.000489	0.003802	0.000400	
0	0.013200	0.001326	0.005605	0.001028	
7	0.010403	0.000494	0.003999	0.000003	
42	0.040802	0.006884	0.006602	0.001627	

	param_lr__penalty	param_poly	\
47	l1	PolynomialFeatures(include_bias=False)	
13	l1	None	
0	none	None	
7	l1	None	
42	l1	PolynomialFeatures(include_bias=False)	

	param_poly__degree	param_lr__alpha	\
47	2	0.1	
13	NaN	0.1	
0	NaN	NaN	
7	NaN	0.001	
42	3	0.01	

	params	split0_test_score	\
47	{'lr__alpha': 0.1, 'lr__penalty': 'l1', 'poly'…	0.927835	
13	{'lr__alpha': 0.1, 'lr__penalty': 'l1', 'poly'…	0.927835	
0	{'lr__penalty': 'none', 'poly': None}	0.934708	
7	{'lr__alpha': 0.001, 'lr__penalty': 'l1', 'pol…	0.912371	
42	{'lr__alpha': 0.01, 'lr__penalty': 'l1', 'poly…	0.893471	

	split1_test_score	split2_test_score	split3_test_score	\
47	0.924399	0.912220	0.936317	
13	0.924399	0.912220	0.936317	
0	0.860825	0.898451	0.920826	
7	0.893471	0.907057	0.939759	

```
42          0.924399          0.846816          0.924269

    split4_test_score  mean_test_score  std_test_score  rank_test_score
47           0.908778         0.921910        0.010151                1
13           0.908778         0.921910        0.010151                1
0            0.903614         0.903685        0.024983                3
7            0.865749         0.903681        0.024214                4
42           0.905336         0.898858        0.028557                5
```

Accuratezza del modello Perceptron Pesato Si riportano gli score calcolati dalla funzione nested_cv

[100]:
```
balanced_perceptron_score
```

[100]:
```
[0.9181568088033012, 0.9291121816930489, 0.9311768754301445]
```

Notiamo che con il modello che ha ottenuto lo score migliore ha una percentuale di accuratezza del 93%. La media di tutti e tre i risultati e pari al 92,6%

[101]:
```
np.mean(balanced_perceptron_score)
```

[101]:
```
0.9261486219754982
```

Analizziamo il modello che ha ottenuto lo score migliore ottenuto con le altre metriche di accuratezza.

Estraiamo le predizioni e visualizziamo la matrice di confusione.

[102]:
```
balanced_perceptron_preds = bal_perceptron_best_gs.predict(X_val)
balanced_perceptron_conf_matrix = confusion_matrix(y_val,␣
↪balanced_perceptron_preds)
```

[103]:
```
bal_perceptron_cm_df = pd.DataFrame(balanced_perceptron_conf_matrix, columns =␣
↪["predicted_"+ x for x in bal_perceptron_best_gs.classes_],
             index = ["real_"+x for x in bal_perceptron_best_gs.classes_])
bal_perceptron_cm_df
```

[103]:
```
               predicted_Cammeo  predicted_Osmancik
real_Cammeo                 525                  44
real_Osmancik                41                 660
```

Notiamo che in questo caso abbiamo ottenuto un leggero miglioramento nella confusion matrix.

Visualizziamo la precision.

[104]:
```
bal_perceptron_precision = pd.DataFrame(precision_score(y_val,␣
↪balanced_perceptron_preds, average = None),
                                        columns=["Precision_score"],␣
↪index=bal_perceptron_best_gs.classes_)
```

```
bal_perceptron_precision
```

[104]:

	Precision_score
Cammeo	0.927562
Osmancik	0.937500

In questo caso, abbiamo ottenuto un miglioramento della precision per la classe Cammeo, ma abbiamo peggiorato leggermente la precision della classe Osmancik.

Osserviamo ora la recall.

[105]:
```
bal_perceptron_recall = pd.DataFrame(recall_score(y_val,␣
↪balanced_perceptron_preds, average = None),
                                          columns=["Recall_score"],␣
↪index=bal_perceptron_best_gs.classes_)
bal_perceptron_recall
```

[105]:

	Recall_score
Cammeo	0.922671
Osmancik	0.941512

Ora notiamo che la recall, rispetto al modello non pesato, è aumentata a favore della classe Osmancik e peggiorata leggermente per la classe Cammeo.

Ora misuriamo l'`F1_Score` per capire se il peso abbia influito o meno sulla precisione complessiva del modello.

[106]:
```
bal_perceptron_f1_measure = pd.DataFrame(f1_score(y_val,␣
↪balanced_perceptron_preds, average = None),
                                             columns=["F1_Measure_score"],␣
↪index=bal_perceptron_best_gs.classes_)
bal_perceptron_f1_measure
```

[106]:

	F1_Measure_score
Cammeo	0.925110
Osmancik	0.939502

Notiamo che il modello pesato, avendo bilanciato le misure di precision e recall, migliora la f1_measure media di circa 1% rispetto al modello non pesato

[107]:
```
bal_perceptron_f1_mean_measure = f1_score(y_val, balanced_perceptron_preds,␣
↪average = "macro")
bal_perceptron_f1_mean_measure
```

[107]: 0.9323059557590104

1.3.4 Logistic Regression

Viene riproposta nuovamente la Logistic Regression, questa volta cercando gli iperparametri migliori attraverso la nested stratified cross-fold validation e la grid search.

Creiamo il modello e la grid.

[108]:
```
log_reg_model = Pipeline([
    ("scaler", StandardScaler()),
    ("poly", None),
    ("lr", LogisticRegression(solver="saga", random_state=123))
])
```

[109]:
```
log_reg_grid = [
    {
        "poly" : [None],
        "lr__penalty" : ["none"]
    },
    {
        "scaler" : [None, StandardScaler()],
        "poly" : [None],
        "lr__penalty" : ["l2", "l1"],
        "lr__C" : np.logspace(-2, 2, 5)
    },
    {
        "poly" : [None],
        "lr__penalty" : ["elasticnet"],
        "lr__C" : np.logspace(-2, 2, 5),
        "lr__l1_ratio": [ 0.2, 0.5]
    },
    {
        "poly" : [PolynomialFeatures(include_bias=False)],
        "poly__degree" : [2, 3],
        "lr__penalty" : ["none"]
    },
    {
        "poly" : [PolynomialFeatures(include_bias=False)],
        "poly__degree" : [2, 3],
        "lr__penalty" : ["l2", "l1"],
        "lr__C" : np.logspace(-2, 2, 5)
    },
    {
        "poly" : [PolynomialFeatures(include_bias=False)],
        "poly__degree" : [2, 3],
        "lr__penalty" : ["elasticnet"],
        "lr__C" : np.logspace(-2, 2, 5),
        "lr__l1_ratio": [ 0.2, 0.5]
    }
```

```
]
```

[110]:
```
%%time
log_reg_score, log_reg_gs = nested_cv(log_reg_model, log_reg_grid)
```

```
Wall time: 7min 32s
```

Visualizziamo la configurazione (gli iperparametri migliori) del modello che ha ottenuto lo score migliore.

[111]:
```
log_reg_best_gs = log_reg_gs[np.argmax(log_reg_score)]
log_reg_best_param = log_reg_best_gs.best_params_
log_reg_best_param
```

[111]:
```
{'lr__C': 1.0,
 'lr__penalty': 'l2',
 'poly': PolynomialFeatures(include_bias=False),
 'poly__degree': 2}
```

Si riporta il ranking score del modello che ha ottenuto il punteggio migliore.

[112]:
```
pd.DataFrame(log_reg_best_gs.cv_results_).sort_values("rank_test_score").head(5)
```

[112]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_lr__penalty	param_poly	param_lr__C	param_scaler	param_lr__l1_ratio	param_poly__degree	params	split0_test_score
41	0.210999	0.009379	0.004797	0.000410	l2	PolynomialFeatures(include_bias=False)	1	NaN	NaN	2	{'lr__C': 1.0, 'lr__penalty': 'l2', 'poly': Po…	0.917323
63	0.338202	0.050637	0.005797	0.000754	elasticnet	PolynomialFeatures(include_bias=False)	1	NaN	0.5	2	{'lr__C': 1.0, 'lr__l1_ratio': 0.5, 'lr__penal…	0.917323
61	0.265214	0.011821	0.004386	0.000482	elasticnet	PolynomialFeatures(include_bias=False)	1	NaN	0.2	2	{'lr__C': 1.0, 'lr__l1_ratio': 0.2, 'lr__penal…	0.917323
51	0.386200	0.029970	0.007201	0.001601	l1	PolynomialFeatures(include_bias=False)	100	NaN	NaN	2		
31	0.165993	0.014036	0.006401	0.001743	none	PolynomialFeatures(include_bias=False)	NaN	NaN	NaN	2		

```
51  {'lr__C': 100.0, 'lr__penalty': 'l1', 'poly': …          0.917323
31  {'lr__penalty': 'none', 'poly': PolynomialFeat…          0.917323

    split1_test_score  split2_test_score  split3_test_score  \
41           0.903543           0.925197           0.944882
63           0.903543           0.925197           0.944882
61           0.903543           0.925197           0.944882
51           0.903543           0.925197           0.942913
31           0.903543           0.925197           0.942913

    split4_test_score  mean_test_score  std_test_score  rank_test_score
41           0.942913         0.926772        0.015620                1
63           0.942913         0.926772        0.015620                1
61           0.942913         0.926772        0.015620                1
51           0.942913         0.926378        0.015177                4
31           0.942913         0.926378        0.015177                4
```

Accuratezza del modello Logistic Regression Si riportano gli score calcolati dalla funzione nested_cv

[113]:
```
log_reg_score
```

[113]:
```
[0.9393700787401574, 0.9165354330708662, 0.9283464566929134]
```

Notiamo che con il modello che ha ottenuto lo score migliore ha una percentuale di accuratezza del 94%. La media di tutti e tre i risultati e pari al 92,8%

[114]:
```
np.mean(log_reg_score)
```

[114]:
```
0.9280839895013123
```

Analizziamo il modello che ha ottenuto lo score migliore ottenuto con le altre metriche di accuratezza.

Estraiamo le predizioni e visualizziamo la matrice di confusione.

[115]:
```
log_reg_preds = log_reg_best_gs.predict(X_val)
log_reg_conf_matrix = confusion_matrix(y_val, log_reg_preds)
```

[116]:
```
log_reg_cm_df = pd.DataFrame(log_reg_conf_matrix, columns =["predicted_"+x for␣
↪x in log_reg_best_gs.classes_],
             index = ["real_"+x for x in log_reg_best_gs.classes_])
log_reg_cm_df
```

[116]:
```
               predicted_Cammeo  predicted_Osmancik
real_Cammeo                520                  49
real_Osmancik               38                 663
```

Visualizziamo la precision.

[117]:
```
log_reg_precision = pd.DataFrame(precision_score(y_val, log_reg_preds, average␣
↪= None),
                                 columns=["Precision_score"],␣
↪index=log_reg_best_gs.classes_)
log_reg_precision
```

[117]:

	Precision_score
Cammeo	0.93190
Osmancik	0.93118

Abbiamo ottenuto una precision alta per entrambe le classi e molto bilanciata.

[118]:
```
log_reg_recall = pd.DataFrame(recall_score(y_val, log_reg_preds, average =␣
↪None),
                              columns=["Recall_score"], index=log_reg_best_gs.
↪classes_)
log_reg_recall
```

[118]:

	Recall_score
Cammeo	0.913884
Osmancik	0.945792

Questa volta la classe Cammeo riporta una recall del 91%, mentre per la varietà Osmacink abbiamo ottenuto una recall più alta, il 95%. Rispetto alla recall precedente è più sbilanciata verso la classe Osmancik.

Ora misuriamo l'F1_Score del modello

[119]:
```
log_reg_f1_measure = pd.DataFrame(f1_score(y_val, log_reg_preds, average =␣
↪None),
                                  columns=["F1_Measure_score"],␣
↪index=log_reg_best_gs.classes_)
log_reg_f1_measure
```

[119]:

	F1_Measure_score
Cammeo	0.922804
Osmancik	0.938429

Notiamo che abbiamo ottenuto una buona accuratezza. La media tra le due F1_measure è del 93%.

[120]:
```
log_reg_f1_mean_measure = f1_score(y_val, log_reg_preds, average = "macro")
log_reg_f1_mean_measure
```

[120]: 0.9306163894524855

1.3.5 Multi-layer Perceptron Classifier

Viene riproposto nuovamente il Multi-Layer Perceptron Classifier, questa volta cercando gli iperparametri migliori attraverso la nested stratified cross-fold validation e la grid search.

Creiamo il modello e la grid.

[121]:
```
mlp_model = Pipeline([
    ("scaler", StandardScaler()),
    ("poly", None),
    ("mlp", MLPClassifier(random_state=123))
])
```

[122]:
```
mlp_grid = [
    {
        "poly" : [None],
        "mlp__activation" : ["identity", "relu"]
    },
    {
        "poly" : [None],
        "mlp__activation" : ["identity", "relu"],
        "mlp__hidden_layer_sizes" : [(200), (100)]
    },
    {
        "poly" : [PolynomialFeatures(include_bias=False)],
        "poly__degree" : [2, 3],
        "mlp__activation" : ["identity", "relu"],
        "mlp__hidden_layer_sizes" : [(200, 50), (100)]
    }
]
```

[123]:
```
%%time
mlp_score, mlp_gs = nested_cv(mlp_model, mlp_grid)
```

```
Wall time: 14min 15s
```

Visualizziamo la configurazione (gli iperparametri migliori) del modello che ha ottenuto lo score migliore.

[124]:
```
mlp_best_gs = mlp_gs[np.argmax(mlp_score)]
mlp_best_param = mlp_best_gs.best_params_
mlp_best_param
```

[124]:
```
{'mlp__activation': 'identity', 'poly': None}
```

Si riporta il ranking score del modello che ha ottenuto il punteggio migliore.

[125]:
```
pd.DataFrame(mlp_best_gs.cv_results_).sort_values("rank_test_score").head(5)
```

[125]:
```
   mean_fit_time  std_fit_time  mean_score_time  std_score_time  \
0       0.411604      0.058323         0.007395        0.001019
3       0.563603      0.136538         0.008597        0.001017
7       1.910205      0.302964         0.010996        0.001416
2       0.501999      0.097162         0.007800        0.000749
8       0.648206      0.070658         0.007396        0.001021

  param_mlp__activation                              param_poly  \
0              identity                                    None
3              identity                                    None
7              identity  PolynomialFeatures(include_bias=False)
2              identity                                    None
8              identity  PolynomialFeatures(include_bias=False)

  param_mlp__hidden_layer_sizes param_poly__degree  \
0                           NaN                NaN
3                           100                NaN
7                     (200, 50)                  3
2                           200                NaN
8                           100                  2

                                              params  split0_test_score  \
0      {'mlp__activation': 'identity', 'poly': None}           0.921260
3  {'mlp__activation': 'identity', 'mlp__hidden_l…           0.921260
7  {'mlp__activation': 'identity', 'mlp__hidden_l…           0.919291
2  {'mlp__activation': 'identity', 'mlp__hidden_l…           0.921260
8  {'mlp__activation': 'identity', 'mlp__hidden_l…           0.919291

   split1_test_score  split2_test_score  split3_test_score  split4_test_score  \
0           0.901575           0.921260           0.938976           0.944882
3           0.901575           0.921260           0.938976           0.944882
7           0.905512           0.927165           0.935039           0.940945
2           0.901575           0.919291           0.938976           0.944882
8           0.895669           0.925197           0.940945           0.944882

   mean_test_score  std_test_score  rank_test_score
0         0.925591        0.015268                1
3         0.925591        0.015268                1
7         0.925591        0.012413                1
2         0.925197        0.015400                4
8         0.925197        0.017563                4
```

Accuratezza del modello Multi-layer perceptron Si riportano gli score calcolati dalla funzione nested_cv

[126]:
```
mlp_score
```

[126]:
```
[0.9362204724409449, 0.9251968503937008, 0.925984251968504]
```

Notiamo che anche con questo modello l'accuratezza è alta. Il modello che ha ottenuto lo score migliore ha una percentuale di accuratezza del 94%. La media di tutti e tre i risultati è del 93% circa.

[127]:
```
np.mean(mlp_score)
```

[127]:
```
0.9291338582677166
```

Analizziamo il modello che ha ottenuto lo score migliore con le altre metriche di accuratezza.

Estraiamo le predizioni e visualizziamo la matrice di confusione.

[128]:
```
mlp_preds = mlp_best_gs.predict(X_val)
mlp_conf_matrix = confusion_matrix(y_val, mlp_preds)
```

[129]:
```
mlp_cm_df = pd.DataFrame(mlp_conf_matrix, columns =["predicted_"+x for x in␣
↪mlp_best_gs.classes_],
             index = ["real_"+x for x in mlp_best_gs.classes_])
mlp_cm_df
```

[129]:

	predicted_Cammeo	predicted_Osmancik
real_Cammeo	520	49
real_Osmancik	39	662

Visualizziamo la precision.

[130]:
```
mlp_precision = pd.DataFrame(precision_score(y_val, mlp_preds, average = None),
                             columns=["Precision_score"], index=mlp_best_gs.
↪classes_)
mlp_precision
```

[130]:

	Precision_score
Cammeo	0.930233
Osmancik	0.931083

Abbiamo ottenuto una precision alta per entrambe le classi e molto bilanciata.

[131]:
```
mlp_recall = pd.DataFrame(recall_score(y_val, mlp_preds, average = None),␣
↪columns=["Recall_score"], index=mlp_best_gs.classes_)
mlp_recall
```

[131]:

	Recall_score
Cammeo	0.913884
Osmancik	0.944365

Anche per la recall di questo modello abbiamo ottenuto valori alti, la classe Cammeo riporta una recall del 91%, la varietà Osmancik una recall del 94%.

Ora misuriamo l'F1_Score del modello

[132]:
```
mlp_f1_measure = pd.DataFrame(f1_score(y_val, mlp_preds, average = None),
                              columns=["F1_Measure_score"], index=mlp_best_gs.
 ↪classes_)
mlp_f1_measure
```

[132]:

	F1_Measure_score
Cammeo	0.921986
Osmancik	0.937677

Notiamo che abbiamo ottenuto una buona accuratezza. La media tra le due F1_measure è quasi del 93%.

[133]:
```
mlp_f1_mean_measure = f1_score(y_val, mlp_preds, average = "macro")
mlp_f1_mean_measure
```

[133]:
```
0.9298314347135997
```

1.3.6 Random Model (Modello Casuale)

Ora vogliamo misurare metriche viste sopra generando un modello casuale.

Importiamo la libreria necessaria per generare il modello. `DummyClassifier` permette di impostare diversi parametri per poter generale modelli casuali im base alle necessità. Nel nostro caso vogliamo che il modello generi casualmente previsioni in modo uniforme. Per farlo impostiamo il parametro `strategy="uniform"`. Il seed è per la riproducibilità dell'esperimento.

[134]:
```
from sklearn.dummy import DummyClassifier
```

[135]:
```
random_model = DummyClassifier(strategy="uniform", random_state=123)
random_model.fit(X_train, y_train)
```

[135]:
```
DummyClassifier(random_state=123, strategy='uniform')
```

Accuratezza del modello Casuale Ora passiamo a misurare l'accuratezza del modello

[136]:
```
random_model.score(X_val, y_val)
```

[136]:
```
0.49921259842519683
```

Notiamo che con questo modello l'accuratezza è intorno al 50%

Ora otteniamo le predizioni col modello generato e visualizziamo la matrice di confusione.

[137]:
```
random_preds = random_model.predict(X_val)
random_conf_matrix = confusion_matrix(y_val, random_preds)
```

[138]:
```
random_cm_df = pd.DataFrame(random_conf_matrix, columns =["predicted_"+x for x␣
↪in random_model.classes_],
             index = ["real_"+x for x in random_model.classes_])
random_cm_df
```

[138]:

	predicted_Cammeo	predicted_Osmancik
real_Cammeo	265	304
real_Osmancik	332	369

Notiamo che molte istanze reali della classe Osmancik sono state idendificate erroneamente come Cammeo, la stessa cosa è successa con le istanze della classe Cammeo identificate come Osmancik.

Possiamo infatti vedere che entrambe le classi hanno ottenuto una precision molto bassa.

[139]:
```
random_precision = pd.DataFrame(precision_score(y_val, random_preds, average =␣
↪None),
                                columns=["Precision_score"], index=random_model.
↪classes_)
random_precision
```

[139]:

	Precision_score
Cammeo	0.443886
Osmancik	0.548291

Stessa cosa per la recall.

[140]:
```
random_recall = pd.DataFrame(recall_score(y_val, random_preds, average = None),
                             columns=["Recall_score"], index=random_model.
↪classes_)
random_recall
```

[140]:

	Recall_score
Cammeo	0.465729
Osmancik	0.526391

Ora misuriamo l'F1_Score del modello.

[141]:
```
random_f1_measure = pd.DataFrame(f1_score(y_val, random_preds, average = None),
                                 columns=["F1_Measure_score"],␣
↪index=random_model.classes_)
random_f1_measure
```

[141]:

	F1_Measure_score
Cammeo	0.454545
Osmancik	0.537118

Rispetto agli altri modelli, abbiamo ottenuto una f1_measure media bassa, il 50%

[142]:
```
random_f1_mean_measure = f1_score(y_val, random_preds, average="macro")
random_f1_mean_measure
```

[142]:
```
0.4958316792377928
```

1.3.7 Logistic regression (no nested_skf_cv)

Ora vogliamo misurare metriche viste sopra su un modello creato impostando manualmente gli iperparametri. Riportiamo uno dei modelli creati all'inizio.

[143]:
```
simply_model_log_reg = LogisticRegression(solver="saga", random_state=123)
simply_model_log_reg.fit(X_train, y_train)
simply_model_log_reg.score(X_val, y_val)
```

[143]:
```
0.7393700787401575
```

Notiamo che con questo modello l'accuratezza è intorno al 74%

Ora otteniamo le predizioni col modello generato e visualizziamo la matrice di confusione.

[144]:
```
simply_mlr_preds = simply_model_log_reg.predict(X_val)
simply_mlr_conf_matrix = confusion_matrix(y_val, simply_mlr_preds)
```

[145]:
```
simply_mlr_cm_df = pd.DataFrame(simply_mlr_conf_matrix, columns␣
↪=["predicted_"+x for x in simply_model_log_reg.classes_],
             index = ["real_"+x for x in simply_model_log_reg.classes_])
simply_mlr_cm_df
```

[145]:

	predicted_Cammeo	predicted_Osmancik
real_Cammeo	271	298
real_Osmancik	33	668

Notiamo che alcune istanze reali della classe Osmancik sono state idendificate erroneamente come Cammeo, la stessa cosa, in maniera più grave, è successa con le istanze della classe Cammeo identificate come Osmancik.

Possiamo infatti vedere che la classe Osmancik ha ottenuto una precision molto bassa rispetto a quella ottenuta dalla classe Cammeo.

[146]:
```
simply_mlr_precision = pd.DataFrame(precision_score(y_val, simply_mlr_preds,␣
↪average = None),
                                    columns=["Precision_score"],␣
↪index=simply_model_log_reg.classes_)
simply_mlr_precision
```

[146]:

	Precision_score
Cammeo	0.891447
Osmancik	0.691511

Viceversa per la recall, la classe Cammeo in questo caso è la più penalizzata.

[147]:
```
simply_mlr_recall = pd.DataFrame(recall_score(y_val, simply_mlr_preds, average␣
↪= None),
                                 columns=["Recall_score"],␣
↪index=simply_model_log_reg.classes_)
simply_mlr_recall
```

[147]:

	Recall_score
Cammeo	0.476274
Osmancik	0.952924

Ora misuriamo l'`F1_Score` del modello.

[148]:
```
simply_mlr_measure = pd.DataFrame(f1_score(y_val, simply_mlr_preds, average =␣
↪None),
                                  columns=["F1_Measure_score"],␣
↪index=simply_model_log_reg.classes_)
simply_mlr_measure
```

[148]:

	F1_Measure_score
Cammeo	0.620848
Osmancik	0.801440

Rispetto agli altri modelli, abbiamo ottenuto una f1_measure media del 71%

[149]:
```
simply_mlr_mean_measure = f1_score(y_val, simply_mlr_preds, average="macro")
simply_mlr_mean_measure
```

[149]: 0.7111436819165378

1.4 Parte 4 - Valutazione di modelli con Intervallo di confidenza

Importiamo da SciPy l'oggetto `norm` che rappresenta la distribuzione normale standard

[150]:
```
from scipy.stats import norm
```

L'accuratezza ci da un'indicazione di quanto il modello sia efficace nel prevedere le classi corrette delle istanze. Tale valore è comunque una stima che dipende anche dal validation set usato, un test su molte osservazioni è in generale più significativo di uno su poche osservazioni.

Fissiamo un livello di confidenza, ovvero una percentuale di certezza che vogliamo avere.

Vogliamo individuare l'intervallo di confidenza, cioè l'intervallo di valori in cui l'accuratezza "reale" del modello si trova col `95%` di probabilità.

Definiamo una funzione che calcoli l'intervallo di confidenza e restituisca una tupla con i due estremi (poniamo Z=1.96 come default), e una funzione che restituisca l'intervallo di confidenza desiderato dell'accuratezza del modello (poniamo level=0.95 come default)

[151]:
```
def conf_interval(a, N, Z=1.96):
    c = (2 * N * a + Z**2) / (2 * (N + Z**2))
    d = Z * np.sqrt(Z**2 + 4*N*a - 4*N*a**2) / (2 * (N + Z**2))
    return c - d, c + d
```

[152]:
```
def model_conf_interval(model, X, y, level=0.95):
    a = model.score(X, y)
#     N = len(X)
    N = X.shape[0]
    Z = norm.ppf((1 + level) / 2)
    return conf_interval(a, N, Z)
```

Misuriamo ora l'intervallo di confidenza al 95% (quello di riferimento) dei modelli generati sopra con lo score migliore. Verrà effettuata anche la misurazione con l'intervallo al 99%.

1.4.1 Intervallo confidenza modello Perceptron

Intervallo di confidenza al 95%

[153]:
```
perceptron_conf_interval = model_conf_interval(perceptron_best_gs, X_val, y_val)
perceptron_conf_interval
```

[153]:
```
(0.9136892099119382, 0.9419902723755265)
```

Intervallo di confidenza al 99%

[154]:
```
perceptron_conf_interval = model_conf_interval(perceptron_best_gs, X_val,␣
↪y_val, 0.99)
perceptron_conf_interval
```

[154]:
```
(0.9082708681065598, 0.9455362798118253)
```

1.4.2 Intervallo confidenza modello Perceptron pesato

Intervallo di confidenza al 95%

[155]:
```
bal_perceptron_conf_interval = model_conf_interval(bal_perceptron_best_gs,␣
↪X_val, y_val)
bal_perceptron_conf_interval
```

[155]:
```
(0.917979675514842, 0.9455500772523241)
```

Intervallo di confidenza al 99%

[156]:
```
bal_perceptron_conf_interval = model_conf_interval(bal_perceptron_best_gs,␣
↪X_val, y_val, 0.99)
bal_perceptron_conf_interval
```

[156]:
```
(0.9126644722136481, 0.9489757688049056)
```

1.4.3 Intervallo confidenza modello Logistic Regression

Intervallo di confidenza al 95%

[157]:
```
log_reg_conf_interval = model_conf_interval(log_reg_best_gs, X_val, y_val)
log_reg_conf_interval
```

[157]:
```
(0.9162619383127274, 0.9441277062625583)
```

Intervallo di confidenza al 99%

[158]:
```
log_reg_conf_interval = model_conf_interval(log_reg_best_gs, X_val, y_val, 0.99)
log_reg_conf_interval
```

[158]:
```
(0.9109050179455579, 0.9476019858329282)
```

1.4.4 Intervallo confidenza modello Multi-layer perceptron

Intervallo di confidenza al 95%

[159]:
```
mlp_conf_interval = model_conf_interval(mlp_best_gs, X_val, y_val)
mlp_conf_interval
```

[159]:
```
(0.9154038532831834, 0.9434157371961618)
```

Intervallo di confidenza al 99%

[160]:
```
mlp_conf_interval = model_conf_interval(mlp_best_gs, X_val, y_val, 0.99)
mlp_conf_interval
```

[160]:
```
(0.910026307535589, 0.9469140776228635)
```

1.4.5 Intervallo confidenza modello Casuale

Intervallo di confidenza al 95%

[161]:
```
random_conf_interval = model_conf_interval(random_model, X_val, y_val)
random_conf_interval
```

[161]:
```
(0.4717575439862779, 0.5266724019177818)
```

Intervallo di confidenza al 99%

[162]:
```
random_conf_interval = model_conf_interval(random_model, X_val, y_val, 0.99)
random_conf_interval
```

[162]:
```
(0.46317101322186377, 0.5352623681581026)
```

1.4.6 Intervallo confidenza modello generato manualmente.

Intervallo di confidenza al 95%

[163]:
```
simply_mlr_conf_interval = model_conf_interval(simply_model_log_reg, X_val,␣
↪y_val)
simply_mlr_conf_interval
```

[163]:
```
(0.7145309553124983, 0.7627654898533534)
```

Intervallo di confidenza al 99%

[164]:
```
simply_mlr_conf_interval = model_conf_interval(simply_model_log_reg, X_val,␣
↪y_val, 0.99)
simply_mlr_conf_interval
```

[164]:
```
(0.7064550065785189, 0.769797053911736)
```

1.4.7 Confrontro tra modelli

Dati due modelli diversi, vogliamo poter valutare se l'accuratezza misurata su uno sia significativamente migliore dell'accuratezza misurata sull'altro.

Con questa valutazione, andremo a calcolare l'intervallo di confidenza della differenza confrontando i tre modelli che hanno ottenuto uno score migliore attraverso la GridSearch con il modello casuale e il modello generato manualmente. Se l'intervallo di confidenza conterrà lo 0 (zero) vorrà dire che per quel intervallo la differenza tra i due modelli non è statisticamente significativa.

Sotto le funzioni per calcolare l'intervallo di confidenza della differenza.

[165]:
```
def diff_interval(a1, a2, N1, N2, Z):
    d = abs(a1 - a2)
    sd = np.sqrt(a1 * (1-a1) / N1 + a2 * (1-a2) / N2)
    return d - Z * sd, d + Z * sd
```

[166]:
```
def model_diff_interval(m1, m2, X, y, level=0.95):
    a1 = m1.score(X, y)
    a2 = m2.score(X, y)
    N = len(X)
    Z = norm.ppf((1 + level) / 2)
    return diff_interval(a1, a2, N, N, Z)
```

I tre modelli (su quattro) che tramite la GridSearch hanno ottenuto gli score migliori sono: Perceptron Pesato, Logistic Regression e MLPClassifier

1.4.8 Perceptron Pesato Vs. Others

-

[167]:
```
model_diff_interval(bal_perceptron_best_gs, random_model, X_val, y_val)
```

[167]:
```
(0.4031160068520978, 0.4646005285809731)
```

-

[168]:
```
model_diff_interval(bal_perceptron_best_gs, random_model, X_val, y_val, 0.99)
```

[168]:
```
(0.39345608838233415, 0.4742604470507368)
```

-

[169]:
```
model_diff_interval(bal_perceptron_best_gs, simply_model_log_reg, X_val, y_val)
```

[169]:
```
(0.16591994758750458, 0.22148162721564504)
```

-

[170]:
```
model_diff_interval(bal_perceptron_best_gs, simply_model_log_reg, X_val, y_val,␣
↪0.99)
```

[170]:
```
(0.1571905750534266, 0.23021099974972303)
```

1.4.9 Logistic Regression Vs. Others

-

[171]:
```
model_diff_interval(log_reg_best_gs, random_model, X_val, y_val)
```

[171]:
```
(0.40147429552375036, 0.4630926336101079)
```

-

[172]:
```
model_diff_interval(log_reg_best_gs, random_model, X_val, y_val, 0.99)
```

[172]:
```
(0.3917933529804987, 0.4727735761533595)
```

-

[173]:
```
model_diff_interval(log_reg_best_gs, simply_model_log_reg, X_val, y_val)
```

[173]:
```
(0.16427112193227536, 0.2199808465716616)
```

-

[174]:
```
model_diff_interval(log_reg_best_gs, simply_model_log_reg, X_val, y_val, 0.99)
```

[174]:
```
(0.15551848984118566, 0.2287334786627513)
```

1.4.10 MLPClassifier Vs. Others

-

[175]:
```
model_diff_interval(mlp_best_gs, random_model, X_val, y_val)
```

[175]:
```
(0.4006535854962538, 0.4623385404879981)
```

-

[176]:
```
model_diff_interval(mlp_best_gs, random_model, X_val, y_val, 0.99)
```

[176]:
```
(0.3909621766786188, 0.4720299493056331)
```

-

[177]:
```
model_diff_interval(mlp_best_gs, simply_model_log_reg, X_val, y_val)
```

[177]:
```
(0.16344688362885929, 0.21923028172547132)
```

-

[178]:
```
model_diff_interval(mlp_best_gs, simply_model_log_reg, X_val, y_val, 0.99)
```

[178]:
```
(0.15468267659874146, 0.22799448875558914)
```

Dalle misurazioni effettuate sopra, possiamo dire con una probabilità del 95% e del 99% che la differenza tra i modelli è statisticamente significativa.

1.4.11 Osservazioni

Dalle misurazioni effettuate sopra possiamo dire con una probabilità del 95% e del 99% che la differenza tra i modelli ottenuti con la GridSearch e i modelli utilizzati per il confronto è statisticamente significativa.

1.5 Parte 5 - Scelta del modello migliore

Per la scelta del modello migliore dobbiamo fare alcune precisazioni. In fase di studio, si è notato che, cambiando il seed al modello o alla stratifield Kfold, il Perceptron si comportava in modo anomalo. A ogni cambio del seed, i modelli che utilizzavano il Perceptron, cambiavano di molto gli score. Dopo diversi test, si è arrivati alla conclusione che il perceptron dipendeva molto dalla suvvivisione dei dati.

Nel punto 4, il Perceptron Pesato, abbiamo voluto confrontarlo ugualmente in quanto abbiamo effettuato gli addestramenti di tutti modelli con gli stessi seed. Inoltre, a differenza degli altri Perceptron, è quello che sembra aver dimostrato meno dipendenza dai dati, anche se un po' ne soffriva anche lui.

Avendo notato ciò con i Perceptron, ci siamo sentiti di escluderli entrambi dalla candidatura come modello migliore.

Rimangono solamente il Logistic Regression e il MLPClassifier. Entrambi i modelli hanno ottenuto un F1_measure medio molto alto.

[179]:
```
metrix_matrix = [[np.mean(log_reg_score), np.mean(mlp_score)],
                 [np.max(log_reg_score), np.max(mlp_score)],
                 [log_reg_f1_mean_measure, mlp_f1_mean_measure],
                 [log_reg_precision.mean().values[0], mlp_precision.mean().
 ↪values[0]],
                [log_reg_recall.mean().values[0], mlp_recall.mean().values[0]]]
metrix_matrix = np.array(metrix_matrix)
metrix_matrix *= 100 # ottengo la percentuale
```

[180]:
```
index_matrix = ["mean_score", "max_score","mean_F1_measure", "mean_precision",␣
 ↪"mean_recall"]
```

[181]:
```
pd.set_option("display.precision", 2) # impostiamo la precisione per␣
 ↪confrontare le misure con quelle del DOI
measure_df = pd.DataFrame(metrix_matrix, index=index_matrix,␣
 ↪columns=["log_reg", "mlp_classifier"])
measure_df
```

[181]:

	log_reg	mlp_classifier
mean_score	92.81	92.91
max_score	93.94	93.62
mean_F1_measure	93.06	92.98
mean_precision	93.15	93.07

```
mean_recall        92.98           92.91
```

Facendo riferimento al DOI indicato nelle citazioni, abbiamo la possibilità di confrontare i nostri risultati con quelli pubblicati. Notiamo che l'accuratezza media del modello Logistic Regression è leggermente inferiore, mentre quella del modello MLPClassifier è leggermente più alto. Per quanto riguarda le altre misure di accuratezza sopra indicate abbiamo ottenuto un punteggio migliore rispetto al DOI.

Visualizziamo la media di tutte le misure prese in considerazione

[182]:
```
pd.DataFrame(measure_df.mean(0), columns=["mean_measure"])
```

[182]:
```
                mean_measure
log_reg                93.19
mlp_classifier         93.10
```

1.5.1 Scelta del modello migliore

Dovendo scegliere tra i due modelli quale sia il migliore, sia per semplicità implementativa, ma soprattutto per semplicità intepretativa, si decide di prendere come modello migliore quello ottenuto con la ***Logistic Regression***.

Il modello migliore della Logistic Regression è stato ottenuto con i seguenti iperparametri e la seguente configurazione:

[183]:
```
log_reg_best_gs.best_params_
```

[183]:
```
{'lr__C': 1.0,
 'lr__penalty': 'l2',
 'poly': PolynomialFeatures(include_bias=False),
 'poly__degree': 2}
```

Visualizziamo ora i parametri trovati dal modello migliore dalla GridSearch

[184]:
```
pd.set_option('display.float_format', lambda x: '%.3f' % x) # per rimuovere la␣
↪notazione scientifica


index_poly_features_names =  log_reg_best_gs.best_estimator_.
↪named_steps["poly"] \
    .get_feature_names(["area", "perimeter", "major_axis", "minor_axis",␣
↪"eccentricy", "convex_area", "extent", "class"])


data_to_be_displayed = log_reg_best_gs.best_estimator_.named_steps["lr"].coef_.
↪ravel()
```

```
weight_df = pd.DataFrame(index=index_poly_features_names,␣
↪data=data_to_be_displayed, columns=["weight"]).T
weight_df
```

[184]:
```
         area  perimeter  major_axis  minor_axis  eccentricy  convex_area  \
weight -0.791     -0.993      -1.255       0.074      -1.309       -0.972

        extent  area^2  area perimeter  area major_axis  area minor_axis  \
weight  -0.126  -0.086           0.019            0.036           -0.115

        area eccentricy  area convex_area  area extent  perimeter^2  \
weight            0.148            -0.076       -0.067        0.132

        perimeter major_axis  perimeter minor_axis  perimeter eccentricy  \
weight                 0.003                 0.054                 0.001

        perimeter convex_area  perimeter extent  major_axis^2  \
weight                 0.020             0.147        -0.016

        major_axis minor_axis  major_axis eccentricy  major_axis convex_area  \
weight                 0.028                 0.018                -0.002

        major_axis extent  minor_axis^2  minor_axis eccentricy  \
weight             0.012         0.035                -0.026

        minor_axis convex_area  minor_axis extent  eccentricy^2  \
weight                 -0.050             0.010         0.114

        eccentricy convex_area  eccentricy extent  convex_area^2  \
weight                  0.045             0.054         -0.063

        convex_area extent  extent^2
weight              0.001     0.172
```

Si nota che abbiamo dei parametri in più, questi sono stati generati dal grado 2 del polinomio

[185]:
```
weight_df.T.sort_values("weight", ascending=True).head(5)
```

[185]:
```
             weight
eccentricy   -1.309
major_axis   -1.255
perimeter    -0.993
convex_area  -0.972
area         -0.791
```

[186]:
```
weight_df.T.sort_values("weight", ascending=True).tail(5)
```

[186]:
```
                   weight
eccentricy^2        0.114
perimeter^2         0.132
perimeter extent    0.147
area eccentricy     0.148
extent^2            0.172
```

Infine, notiamo che i primi 5 pesi che caratterizzano maggiormente il modello sono dati dalle seguenti features: - eccentricity - minor_axis - perimeter - convex_area - area

1.6 Extra - Reti neurali con Keras

Si vuole effettuare un confronto con un modello di classificazione basato sulle reti neurali. Importiamo le librerie necessarie.

[187]:
```
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)

from tensorflow import keras
from keras.models import Sequential
from keras.layers import Dense
import tensorflow as tf
```

```
Using TensorFlow backend.
```

Risuddividiamo nuovamente la matrice dei dati dalla colonna delle classi utilizzando questa volta il dataframe con la feature "class" binarizzata.

[188]:
```
y_rice = rice_binary["class"]
X_rice = rice_binary.drop(columns="class")
```

Ricorriamo nuovamente al metodo `Hold-Out` per suddividere i dati in train set e validation set.

[189]:
```
X_train, X_val, y_train, y_val = train_test_split(
    X_rice, y_rice,             # dati da suddividere
    test_size=1/3,    # proporzione: 2/3 training, 1/3 validation
    random_state=42   # seed per la riproducibilità
)
```

Standardizziamo le features numeriche.

[190]:
```
scale = StandardScaler()
```

[191]:
```
Xn_train = scale.fit_transform(X_train)
```

[192]:
```
Xn_val = scale.transform(X_val)
```

Convertiamo la feature “class” di train e validation col metodo `to_categorical` di keras

[193]:
```
from keras.utils import to_categorical
yt_train = to_categorical(y_train)
yt_val = to_categorical(y_val)
```

Impostiamo il modello della rete neurale.

[194]:
```
from keras.regularizers import l1
from keras.layers import Dropout
keras_model = Sequential([
    Dense(32, activation="relu", kernel_regularizer=l1(0.001), input_dim=7),
    Dense(16, activation="relu"),
    Dropout(0.1),
    Dense(8, activation="relu"),
    Dropout(0.3),
    Dense(2, activation="softmax")
])
```

Compiliamo il modello indicando il tipo di ottimizzatore, la loss e la metrica di accuratezza.

[195]:
```
keras_model.compile(
    optimizer="adam",
    loss="categorical_crossentropy",
    metrics=["cosine_proximity"]
)
```

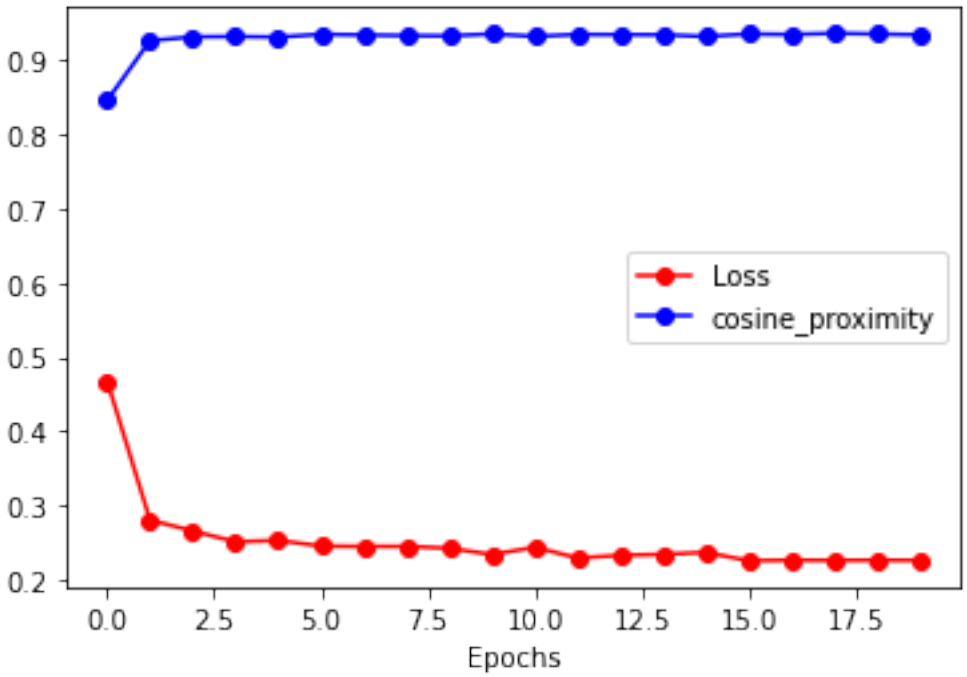
[196]:
```
fit_history = keras_model.fit(Xn_train, yt_train, batch_size=10, epochs=20)
```

```
WARNING:tensorflow:From C:\Users\Home\anaconda3\lib\site-
packages\keras\backend\tensorflow_backend.py:422: The name tf.global_variables
is deprecated. Please use tf.compat.v1.global_variables instead.

Epoch 1/20
2540/2540 [==============================] - 1s 367us/step - loss: 0.4677 -
cosine_proximity: 0.8454
Epoch 2/20
2540/2540 [==============================] - 0s 191us/step - loss: 0.2808 -
cosine_proximity: 0.9258
Epoch 3/20
2540/2540 [==============================] - 1s 229us/step - loss: 0.2660 -
cosine_proximity: 0.9309
Epoch 4/20
2540/2540 [==============================] - 1s 233us/step - loss: 0.2514 -
cosine_proximity: 0.9314
Epoch 5/20
2540/2540 [==============================] - 1s 220us/step - loss: 0.2533 -
```

```
cosine_proximity: 0.9306
Epoch 6/20
2540/2540 [==============================] - 1s 209us/step - loss: 0.2457 -
cosine_proximity: 0.9344
Epoch 7/20
2540/2540 [==============================] - 0s 185us/step - loss: 0.2452 -
cosine_proximity: 0.9335
Epoch 8/20
2540/2540 [==============================] - 0s 181us/step - loss: 0.2453 -
cosine_proximity: 0.9326
Epoch 9/20
2540/2540 [==============================] - 1s 230us/step - loss: 0.2427 -
cosine_proximity: 0.9323
Epoch 10/20
2540/2540 [==============================] - 1s 220us/step - loss: 0.2342 -
cosine_proximity: 0.9349
Epoch 11/20
2540/2540 [==============================] - 1s 208us/step - loss: 0.2441 -
cosine_proximity: 0.9315
Epoch 12/20
2540/2540 [==============================] - 1s 226us/step - loss: 0.2296 -
cosine_proximity: 0.9342
Epoch 13/20
2540/2540 [==============================] - 1s 215us/step - loss: 0.2331 -
cosine_proximity: 0.9339
Epoch 14/20
2540/2540 [==============================] - 1s 242us/step - loss: 0.2345 -
cosine_proximity: 0.9338
Epoch 15/20
2540/2540 [==============================] - 1s 224us/step - loss: 0.2372 -
cosine_proximity: 0.9314
Epoch 16/20
2540/2540 [==============================] - 1s 266us/step - loss: 0.2260 -
cosine_proximity: 0.9349
Epoch 17/20
2540/2540 [==============================] - 1s 225us/step - loss: 0.2266 -
cosine_proximity: 0.9341
Epoch 18/20
2540/2540 [==============================] - 0s 181us/step - loss: 0.2264 -
cosine_proximity: 0.9357
Epoch 19/20
2540/2540 [==============================] - 0s 188us/step - loss: 0.2266 -
cosine_proximity: 0.9348
Epoch 20/20
2540/2540 [==============================] - 0s 194us/step - loss: 0.2264 -
cosine_proximity: 0.9337
```

[197]:
```
plt.plot(fit_history.history["loss"], "ro-")
plt.plot(fit_history.history["cosine_proximity"], "bo-")
plt.legend(["Loss", "cosine_proximity"])
plt.xlabel("Epochs");
```



Visualizziamo le probabilità di appartenenza alla classe

[198]:
```
keras_model.predict(Xn_val[:5])
```

[198]:
```
array([[0.7970022 , 0.20299786],
       [0.9842144 , 0.01578563],
       [0.96039385, 0.03960611],
       [0.08907217, 0.91092783],
       [0.99475926, 0.00524068]], dtype=float32)
```

Visualizziamo la stessa di sopra ma con le classi predette.

[199]:
```
keras_model.predict_classes(Xn_val[:5])
```

[199]:
```
array([0, 0, 0, 1, 0], dtype=int64)
```

Valutiamo il modello sul validation set

[200]:
```
keras_model.evaluate(Xn_val, yt_val)
```

```
1270/1270 [==============================] - 0s 96us/step
```

[200]: `[0.19731695914831687, 0.9428719878196716]`

Sul modello di rete neurale con Keras abbiamo ottenuto, sul validation set, un'accuratezza del 94% e una loss del 19%.