UNIVERSITA' DEGLI STUDI DI BRESCIA
DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE
Corso di Laurea Magistrale in Ingegneria Informatica

Secondary Mushrooms Dataset

Elaborato del corso di

Machine Learning e Data Mining

aa. 2021-22

Edoardo Fratus
Lorenzo Bargnani
Mattia Andreocchi

INDICE

1. INTRODUZIONE AL DATASET

In questo documento viene presentato il lavoro svolto da Fratus Edoardo, Lorenzo Bargnani e Mattia Andreocchi nell'ambito del progetto del corso di Machine Learning e Data Mining.
Il dataset utilizzato per questa relazione è stato oggetto di un lavoro di tesi alla Philipps-University di Marburg, Bioinformatics Division ed è stato reperito nel repository di Machine Learning UCI al seguente indirizzo: https://archive.ics.uci.edu/ml/datasets/Secondary+Mushroom+Dataset. Ogni istanza rappresenta un fungo, descritto attraverso le sue caratteristiche fisiche e nominali.
Tale dataset è stato poi rivisto dai membri del gruppo al fine di ottenere una versione più autoesplicativa dello stesso; in particolare le sigle presenti nel dataset sono state sostituite con parole complete al fine di comprendere meglio i dati in esso contenuti. Quest'ultima versione è disponibile in allegato a questa relazione nel file *Third_Mushrooms_dataset.cvs*.
L'obiettivo dell'elaborato è stato quello di analizzare il dataset al fine di proporre e addestrare dei modelli di Machine Learning in grado di classificare con efficacia un fungo come velenoso o come commestibile sulla base delle sue caratteristiche.

1.1 Analisi preliminare del dataset

Il dataset è composto da 61069 istanze, ognuna descritta da 20 attributi che definiscono le caratteristiche fisiche di un ipotetico fungo appartenente ad una delle 173 specie prese in esame.
Dopo aver caricato il dataset rivisto all'interno del Colab Notebook presente in allegato, attraverso il comando `data_set.info()`, abbiamo ottenuto l'elenco dei 21 attributi:

```
Data columns (total 21 columns):
 #   Column                 Non-Null Count  Dtype
---  ------                 --------------  -----
 0   class                  61069 non-null  object
 1   cap-diameter           61069 non-null  float64
 2   cap-shape              61069 non-null  object
 3   cap-surface            46949 non-null  object
 4   cap-color              61069 non-null  object
 5   does-bruise-or-bleed   61069 non-null  object
 6   gill-attachment        51185 non-null  object
 7   gill-spacing           36006 non-null  object
 8   gill-color             61069 non-null  object
 9   stem-height            61069 non-null  float64
 10  stem-width             61069 non-null  float64
 11  stem-root              9531 non-null   object
 12  stem-surface           22945 non-null  object
 13  stem-color             61069 non-null  object
 14  veil-type              3177 non-null   object
 15  veil-color             7413 non-null   object
 16  has-ring               61069 non-null  object
 17  ring-type              58598 non-null  object
 18  spore-print-color      6354 non-null   object
 19  habitat                61069 non-null  object
 20  season                 61069 non-null  object
dtypes: float64(3), object(18)
memory usage: 10.3+ MB
```

Figura 1.1: descrizione generale degli attributi del dataset

A partire da tale output abbiamo potuto subito ottenere diverse informazioni sul dataset:

- Presenza di 3 attributi di tipo numerico, in particolare float
- Presenza di 17 attributi nominali, rappresentati inizialmente come tipo object
- Presenza di missing values in alcune delle features, anche in quantità elevate

È ora riportata una tabella contenente una descrizione delle variabili e dei loro possibili valori, utile per comprendere meglio le features del dataset:

Feature	Descrizione	Possibili valori
Class	Indica la classe a cui appartiene il fungo	Poisonous Edibile
Cap-Diameter	Indica il diametro della cappella del fungo	Numero reale in cm
Cap-Shape	Indica la forma della cappella del fungo	Bell Conical Convex Flat
Cap_surface	Indica la superficie della cappella del fungo	Fibrous Grooves Scaly Smooth Dry Shiny Leathery Silky Sticky
Cap-Color	Indica il colore della cappella del fungo	Brown Buff Gray Green Pink Purple Red White Yellow Orange Black Blue

Feature	Descrizione	Possibili valori
Does Bruise or bleed	Indica se rilasciano sostanze liquide quando vengono tagliati	Valore booleano
Gill-Atttachment	Indica il tipo di connessione fra lo stelo e la cappella del fungo	Adnate Adnexed Decurrent Pores Sinuate Free None Unknown
Gill-Spacing	Indica la distanza fra la cappella e lo stelo del fungo	Close Distant None
Gill-Color	Indica il colore della porzione del fungo che connette stelo e cappella del fungo	Brown Buff Gray Green Pink Purple Red White Yellow Orange Black Blue None
Stem-Width	Indica la larghezza dello stelo del fungo	Numero reale in cm
Stem-Height	Indica l'altezza dello stelo del fungo	Numero reale in cm
Stem-Root	Indica il tipo di radice del fungo	Bulbous Swollen Club Cup Equal Rhizomorphs Rooted

Feature	Descrizione	Possibili valori
Stem-Surface	Indica la superficie dello stelo del fungo	Fibrous Grooves Scaly Smooth Dry Shiny Leathery Silky Sticky None
Stem-Color	Indica il colore dello stelo del fungo	Brown Buff Gray Green Pink Purple Red White Yellow Orange Black Blue None
Veil-type	Indica il tipo di velo del fungo	Partial Universal
Veil-Color	Indica il colore del velo del fungo	Brown Buff Gray Green Pink Purple Red White Yellow Orange Black Blue None
Has-ring	Indica se il fungo è dotato di anelli	Valore booleano

Feature	Descrizione	Possibili valori
Ring-Type	Indica il tipo di anelli del fungo	Cobwebby Evanescent Flaring Grooved Large Pendant Sheathing Zone Scaly Movable None Unknown
Spore Print Color	Colore delle spore	Brown Buff Gray Green Pink Purple Red White Yellow Orange Black Blue
Habitat	Habitat in cui è possibile trovare il fungo	Grasses Leaves Meadows Paths Heaths Urban Waste Woods
Season	Stagione in cui è possibile trovare il fungo	Spring Summer Autumn Winter

Tabella 1.1: descrizione e possibili valori degli attributi

È importante notare come alcune delle features abbiano come possibile valore “None” o “Unknown”, ad indicare che quella caratteristica non è presente in alcune istanze oppure ha un valore sconosciuto. Questi dati non sono stati considerati come missing values, ma semplicemente come un possibile valore attribuibile alla feature in questione.

Una volta ottenute queste informazioni generiche, abbiamo utilizzato il comando `dataset.describe()` per ottenere una descrizione statistica delle features numeriche, ottenendo l'output mostrato nella figura 1.2.

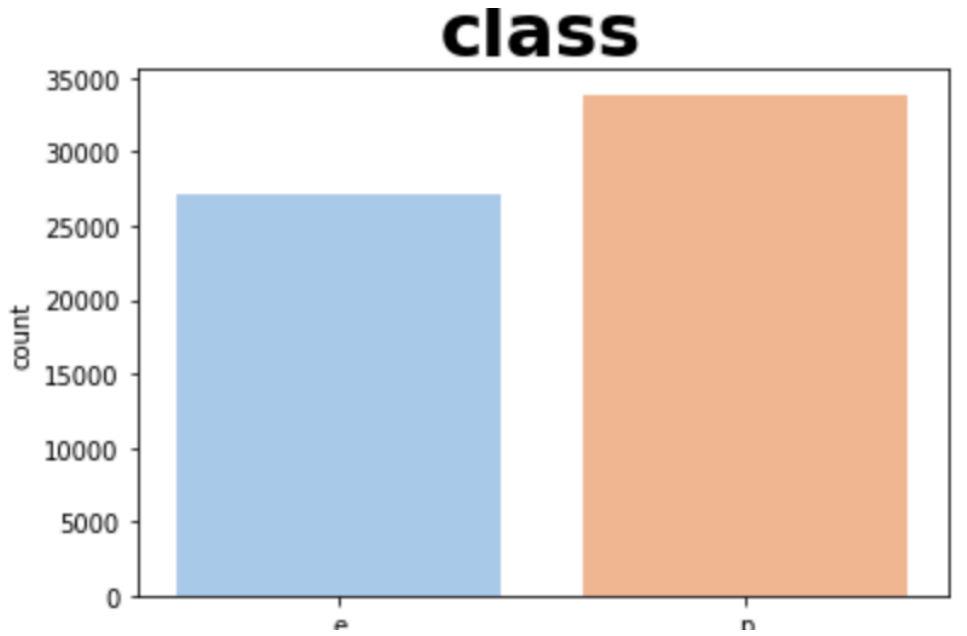
	cap-diameter	stem-height	stem-width
count	61069.000000	61069.000000	61069.000000
mean	6.733854	6.581538	12.149410
std	5.264845	3.370017	10.035955
min	0.380000	0.000000	0.000000
25%	3.480000	4.640000	5.210000
50%	5.860000	5.950000	10.190000
75%	8.540000	7.740000	16.570000
max	62.340000	33.920000	103.910000

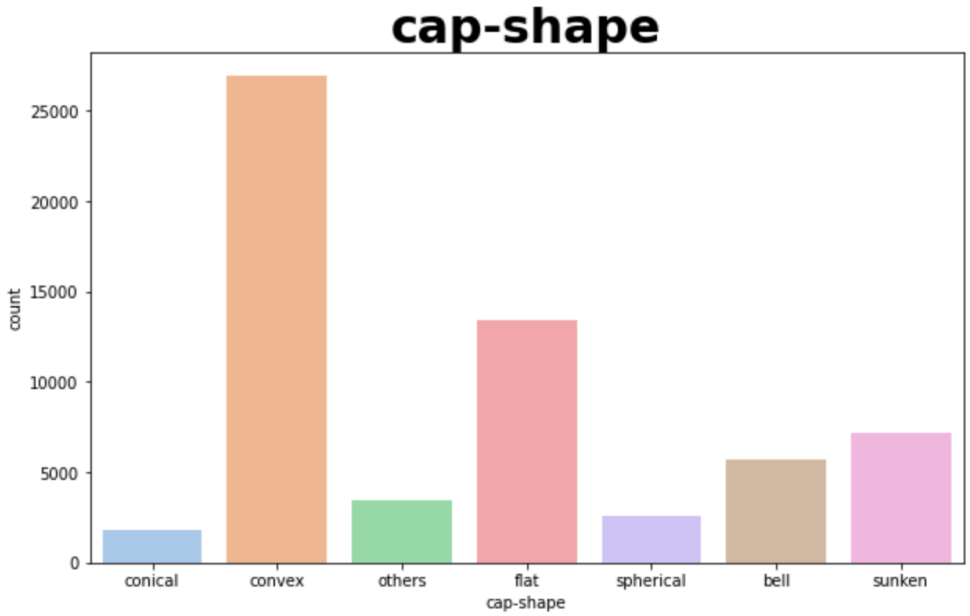
Figura 1.2: analisi statistica delle features numeriche

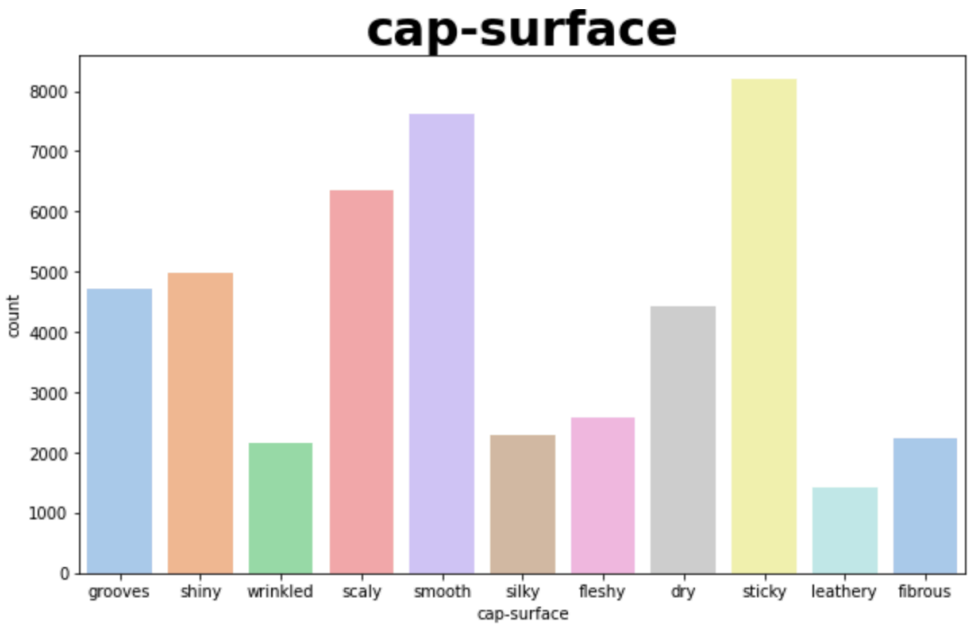
Dal risultato ottenuto è subito visibile la presenza di valori di outlier all'interno del dataset, in quanto per tutte e 3 le features analizzate si ha che il valore massimo è ben più grande della media sommata alla deviazione standard.

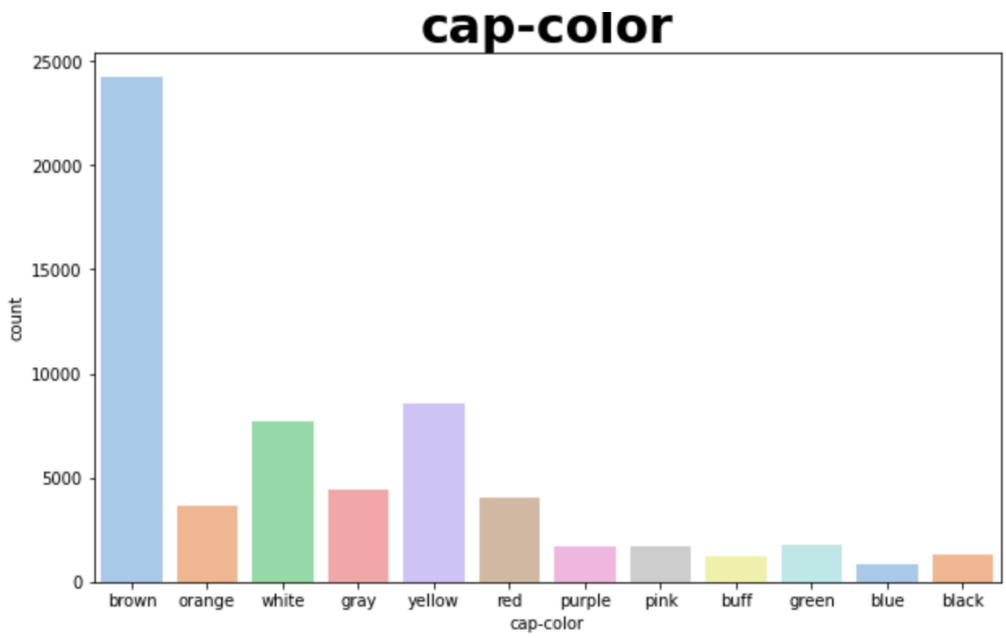
1.2 Analisi grafica del dataset

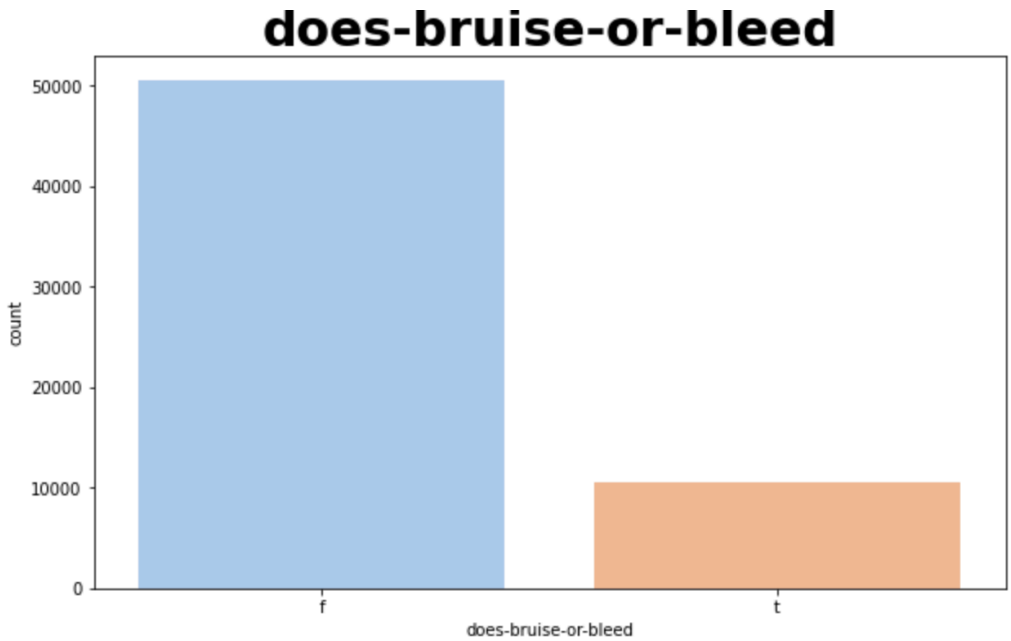
Dopo aver eseguito un'analisi preliminare atta ad ottenere le informazioni di base sul dataset e le sue features, abbiamo deciso di svolgere anche un'analisi di tipo grafico, in modo da ottenere informazioni riguardanti la distribuzione dei valori all'interno di ogni attributo.
Abbiamo quindi utilizzato dei barchart, ottenuti sfruttando le librerie `matplotlib e seaborn.` I risultati sono presentati di seguito:

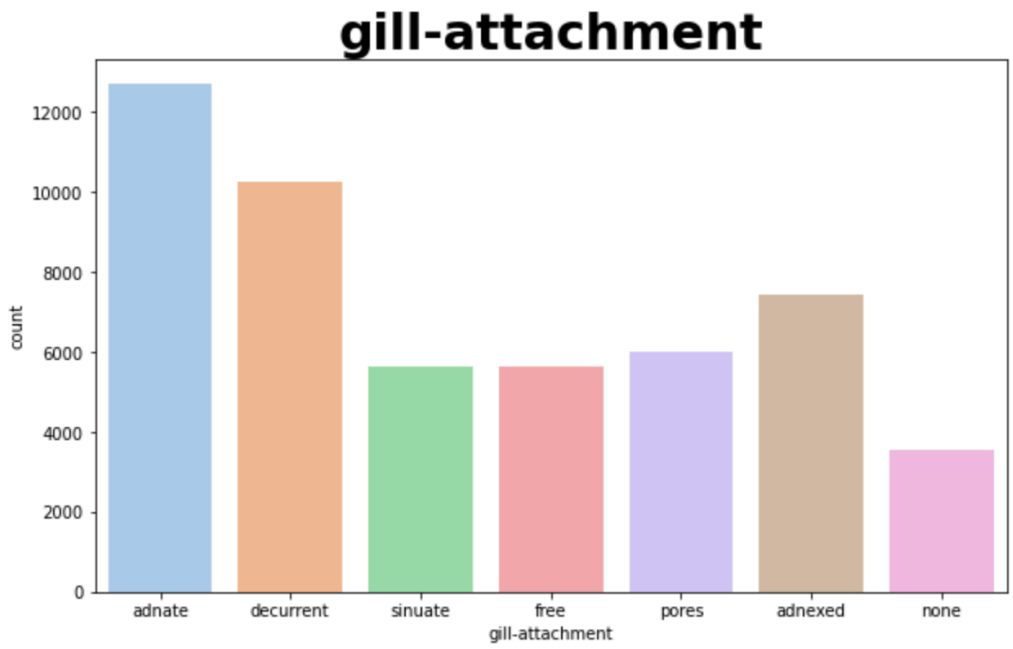


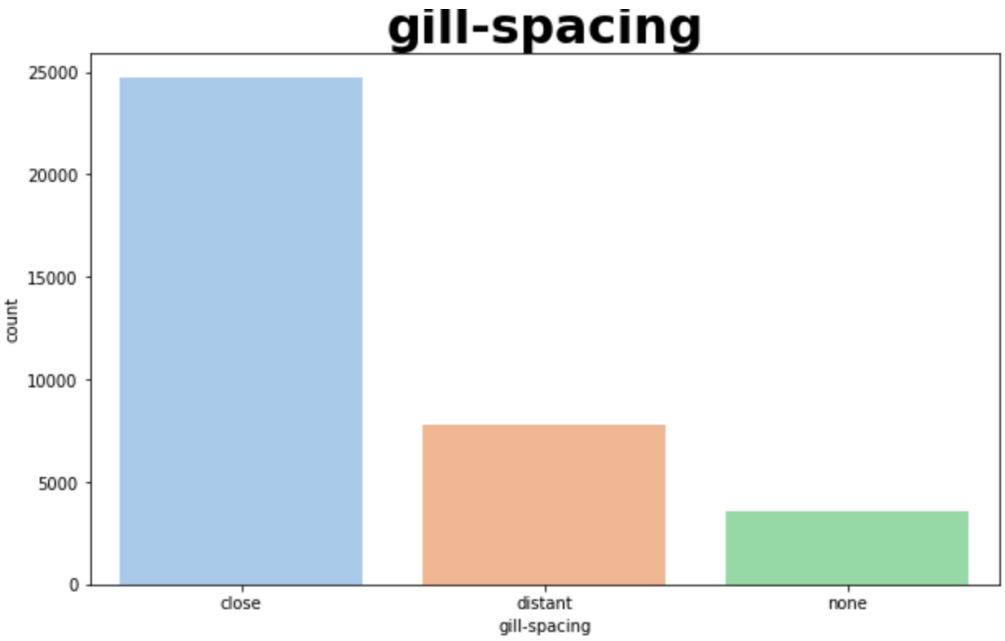


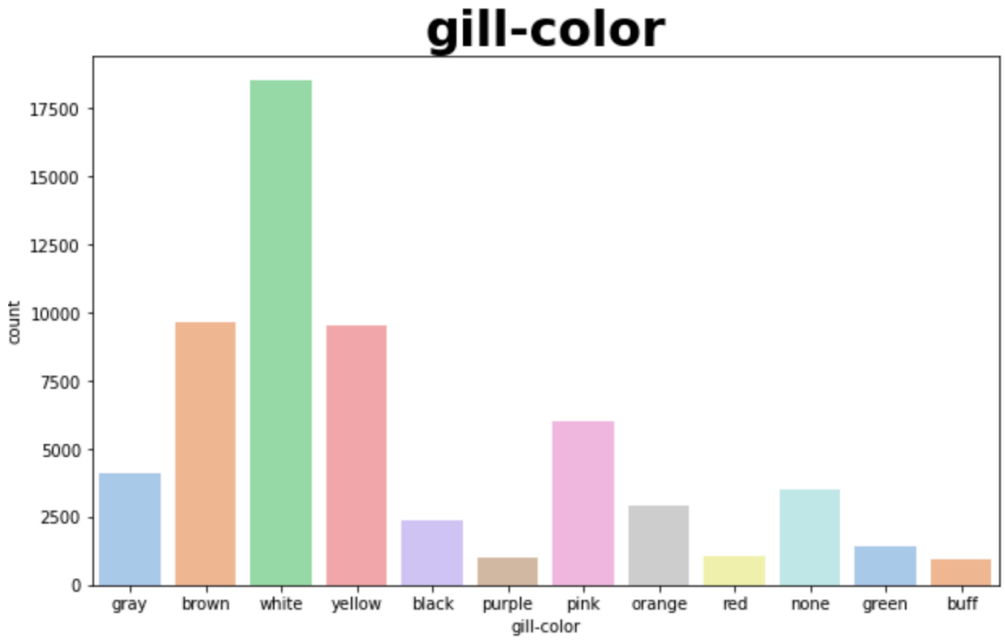

cap-surface
8000
7000
6000
5000
4000
3000
2000
1000
0
count
grooves
shiny
wrinkled
scaly
smooth
silky
fleshy
dry
sticky
leathery
fibrous
cap-surface

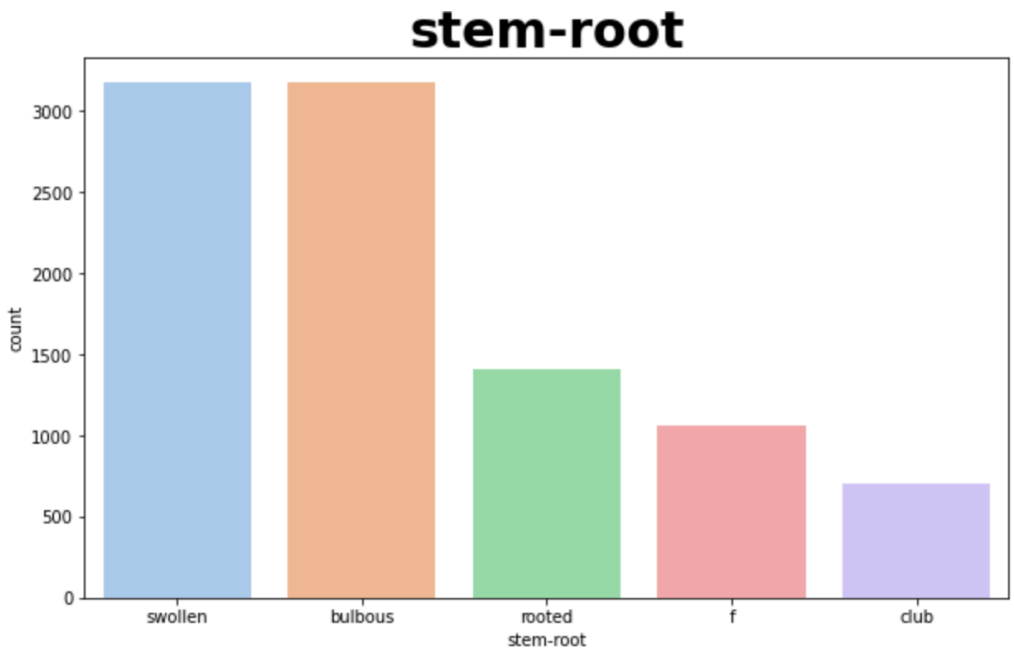

cap-color
25000
20000
15000
10000
5000
0
count
brown
orange
white
gray
yellow
red
purple
pink
buff
green
blue
black
cap-color

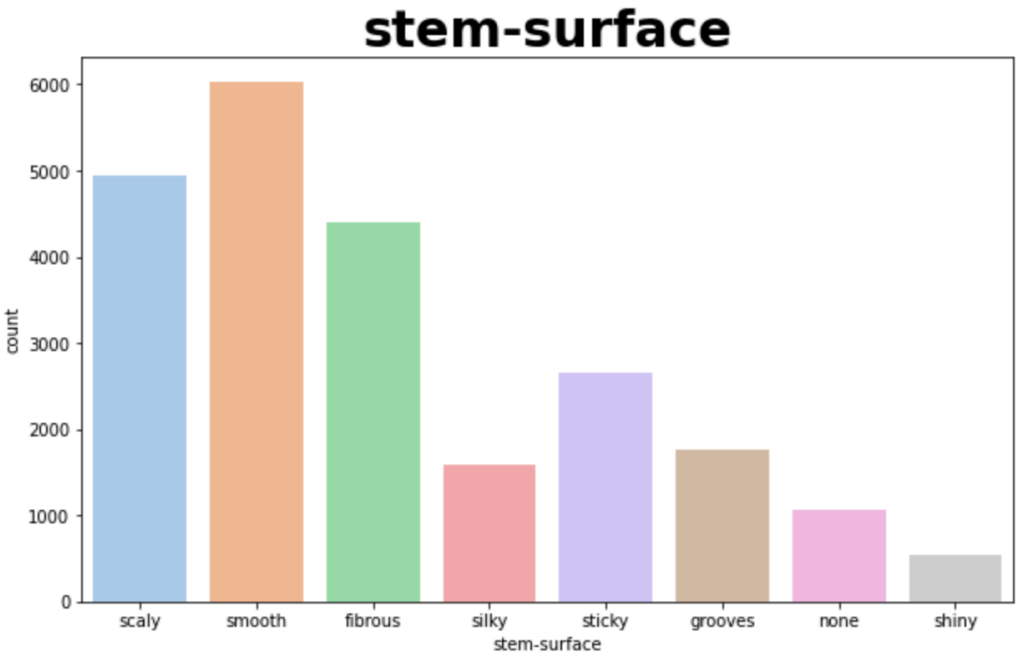

does-bruise-or-bleed
50000
40000
30000
20000
10000
0
count
f
t
does-bruise-or-bleed

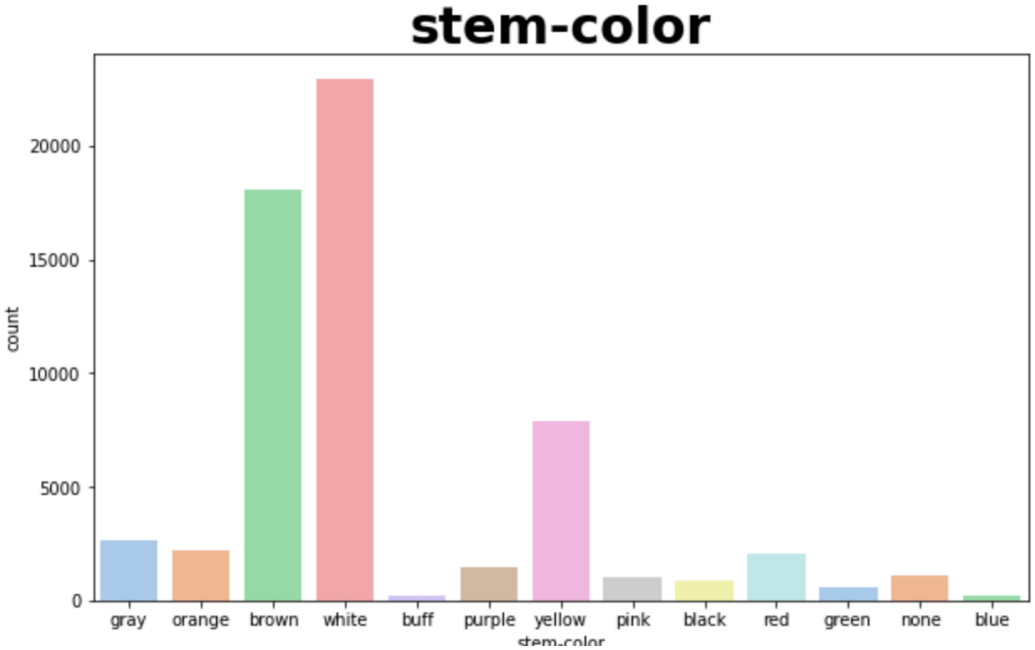

gill-attachment
count
12000
10000
8000
6000
4000
2000
0
adnate
decurrent
sinuate
free
pores
adnexed
none
gill-attachment

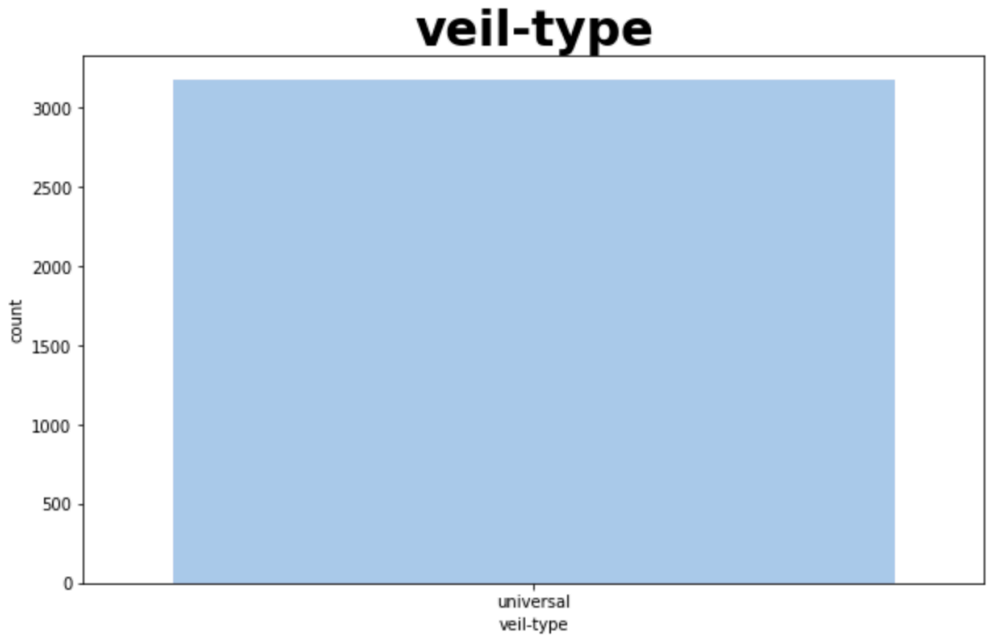

gill-spacing
count
25000
20000
15000
10000
5000
0
close
distant
none
gill-spacing

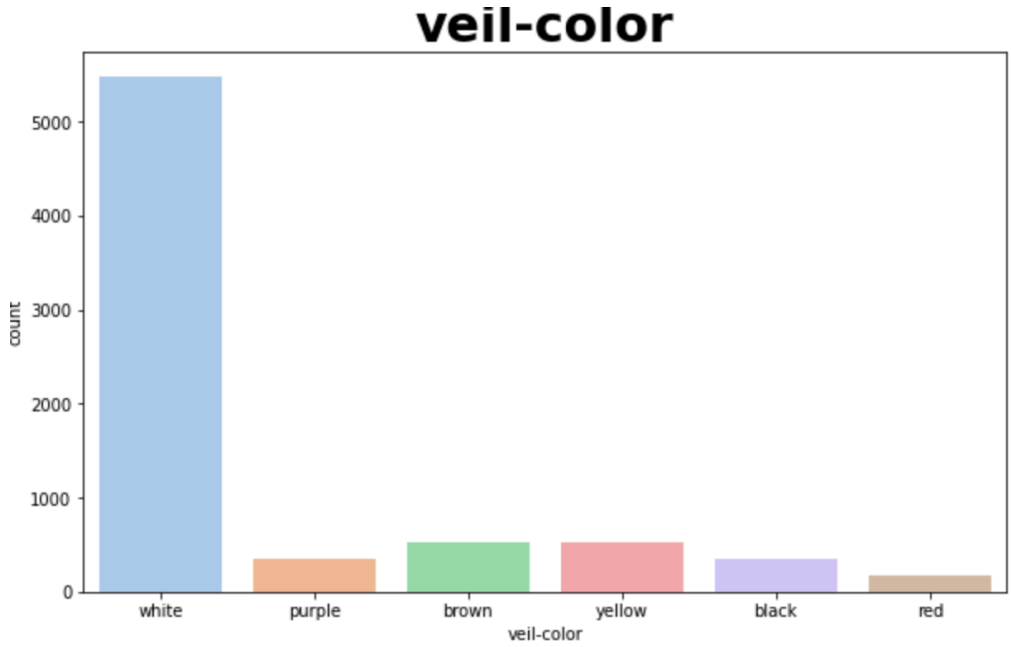

gill-color
count
17500
15000
12500
10000
7500
5000
2500
0
gray
brown
white
yellow
black
purple
pink
orange
red
none
green
buff
gill-color

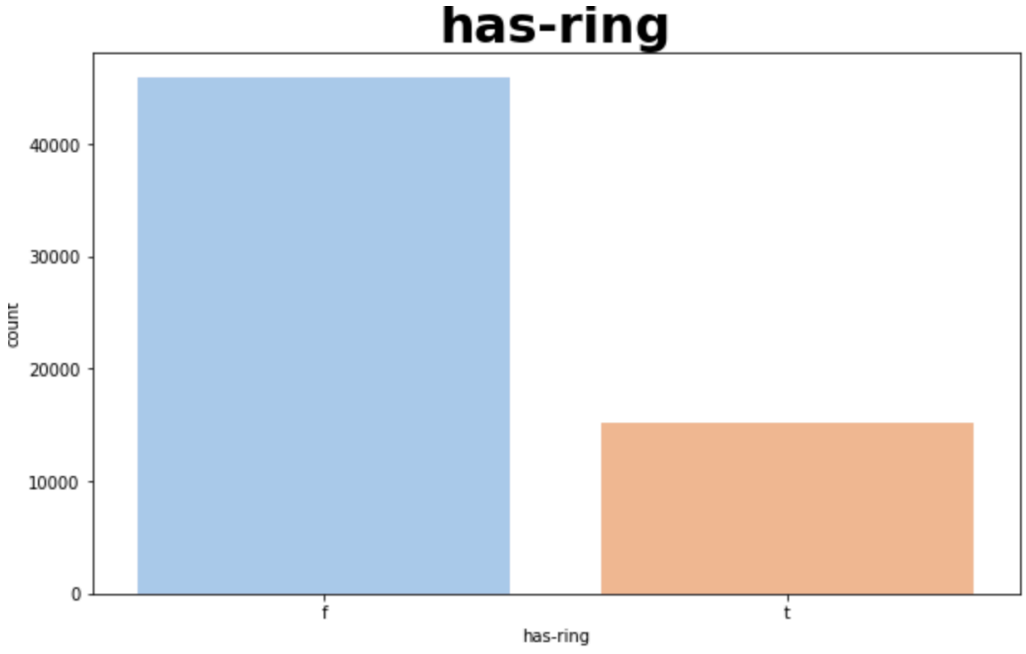

stem-root
3000
2500
2000
1500
1000
500
0
count
swollen
bulbous
rooted
f
club
stem-root

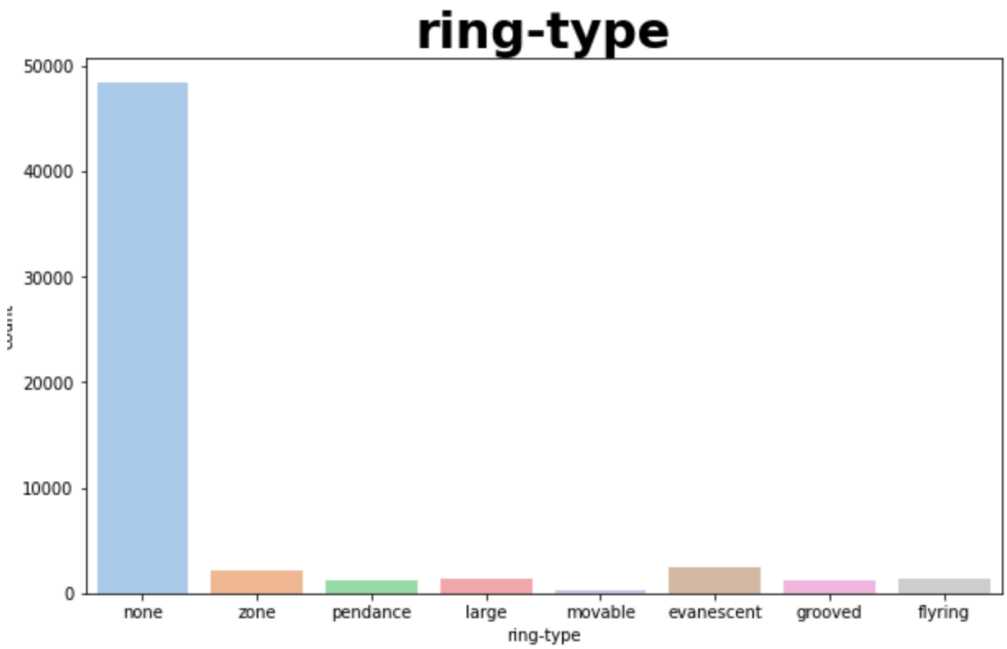

stem-surface
6000
5000
4000
3000
2000
1000
0
count
scaly
smooth
fibrous
silky
sticky
grooves
none
shiny
stem-surface

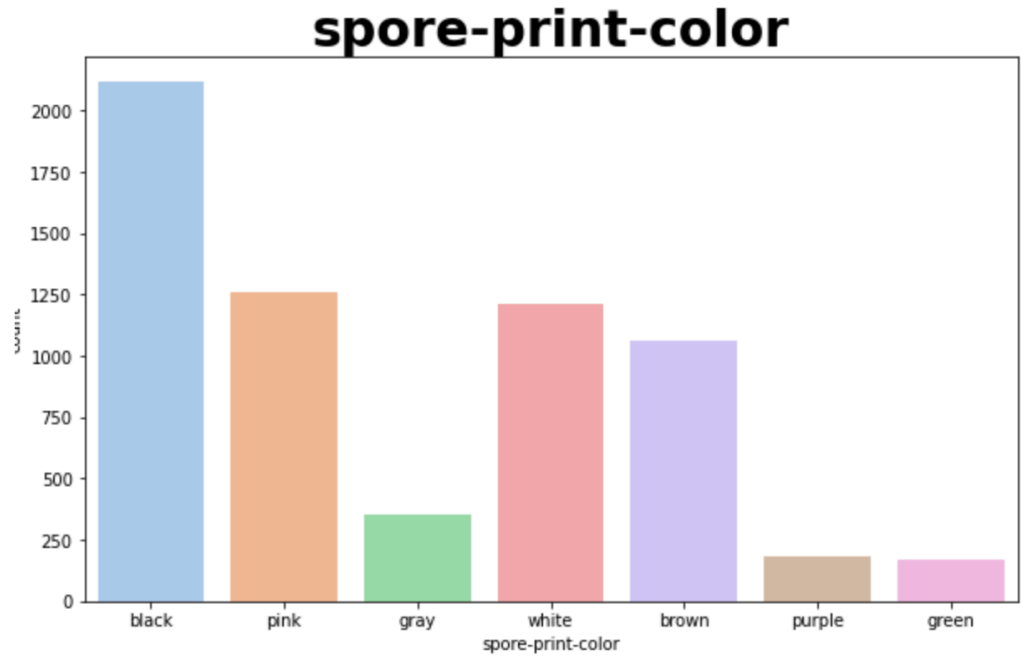

stem-color
20000
15000
10000
5000
0
count
gray
orange
brown
white
buff
purple
yellow
pink
black
red
green
none
blue

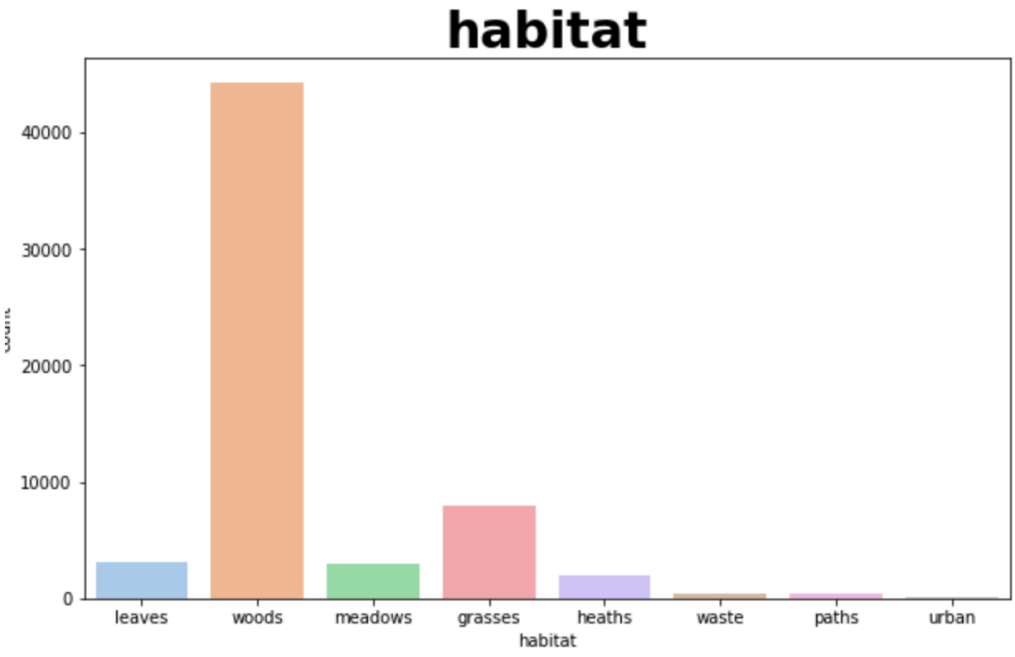

veil-type
3000
2500
2000
1500
1000
500
0
count
universal
veil-type

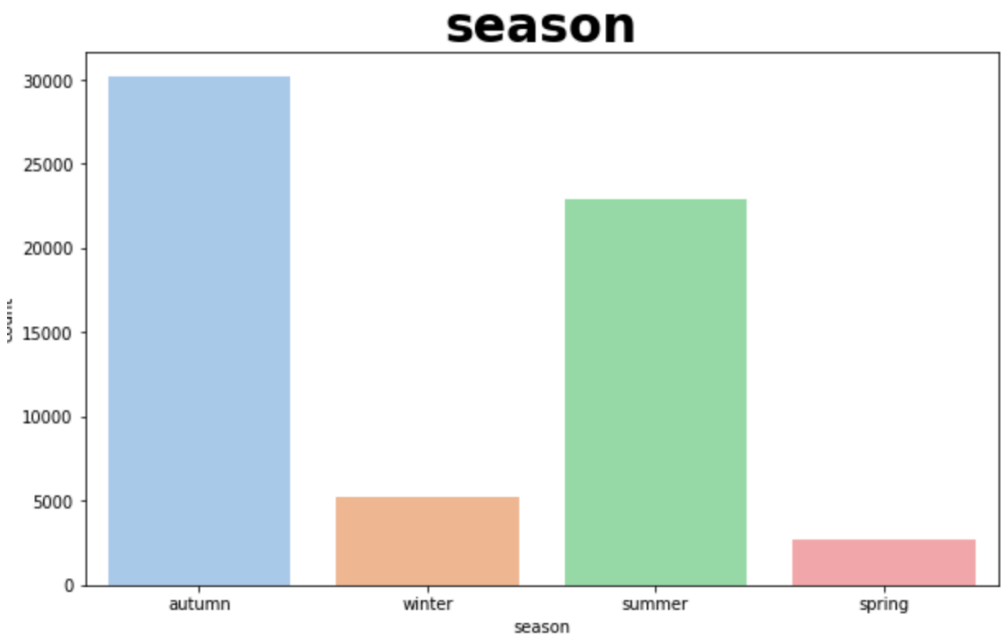

veil-color
5000
4000
3000
2000
1000
0
count
white
purple
brown
yellow
black
red
veil-color


has-ring
40000
30000
20000
10000
0
count
f
t
has-ring


ring-type
50000
40000
30000
20000
10000
0
none
zone
pendance
large
movable
evanescent
grooved
flyring
ring-type


spore-print-color
2000
1750
1500
1250
1000
750
500
250
0
black
pink
gray
white
brown
purple
green
spore-print-color


habitat
40000
30000
20000
10000
0
leaves
woods
meadows
grasses
heaths
waste
paths
urban
habitat



Dall'analisi dei grafici si può notare che alcune variabili hanno una predominanza quasi totale di un singolo valore, questo ci porta ad ipotizzare che tali variabili non saranno quindi utili ai fini della classificazione, in quanto non sono in grado di essere utilizzate per discriminare fra le due classi poiché hanno quasi sempre lo stesso valore. (*veil_type, veil_color, ring_type, does_bruise_bleed)*.

2. PRE-PROCESSING DEI DATI

Nella fase di pre-processing dei dati ci siamo concentrati sopratutto su 2 aspetti, che ci sembravano fondamentali nell'ottica di ottenere un dataset migliore per un modello di machine learning:

- Rimozione dei valori di outlier
- Analisi e ricostruzione dei missing values

2.1 Rimozione dei valori di outlier

Nella prima fase del preprocessing abbiamo deciso di concentrarci sulla rimozione dei valori di outlier, che avrebbero potuto dar fastidio sia nella fase di stima dei missing values, sia nella fase di training e selezione dei modelli di ML.
Abbiamo quindi definito una funzione (figura 2.1) che rimuove dal dataset tutti i valori maggiori del valore medio + 3 volte la deviazione standard. Questa treshold è stata decisa arbitrariamente da noi pensando alle varie dimensioni dei funghi presenti in natura. Questo ci ha permesso di eliminare i valori troppo distanti dalla media mantenendo però un buon livello di generalizzazione nel dataset.

```
[ ]  # definizione funzione per rimozione outlier
     def remove_outliers(df,columns,n_std):
         for col in columns:
             print('Working on column: {}'.format(col))
             mean = df[col].mean()
             sd = df[col].std()
             df = df[(df[col] <= mean+(n_std*sd))]
         return df
```

Figura 2.1: funzione per la rimozione degli outliers

Nelle figure sottostanti è riportata una visualizzazione grafica, tramite boxplot, degli attributi numerici del dataset prima e dopo la rimozione degli outlier. I dati iniziali (figura 2.2) fanno riferimento ai dati mostrati nella figura 1.2.

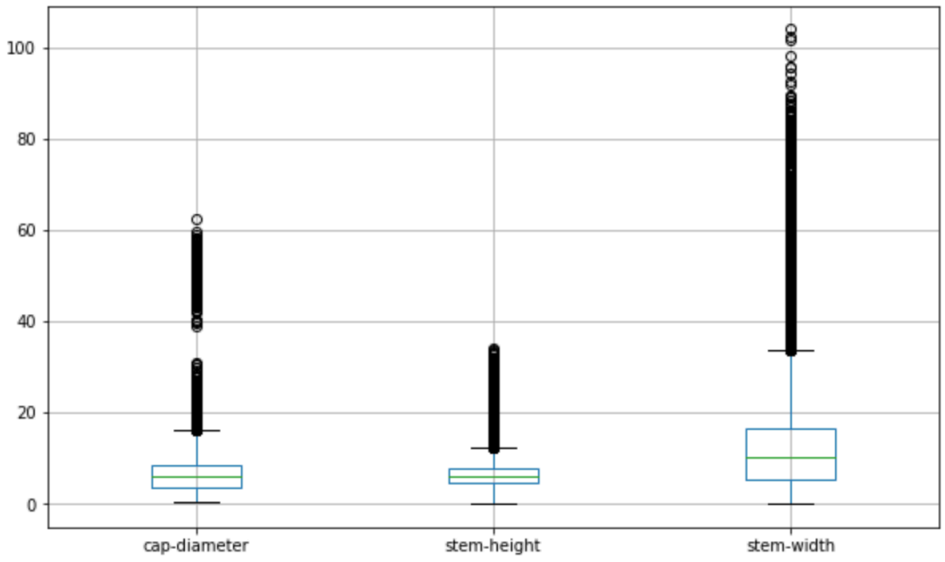


Figura 2.2: boxplot delle variabili numeriche prima della rimozione degli outlier

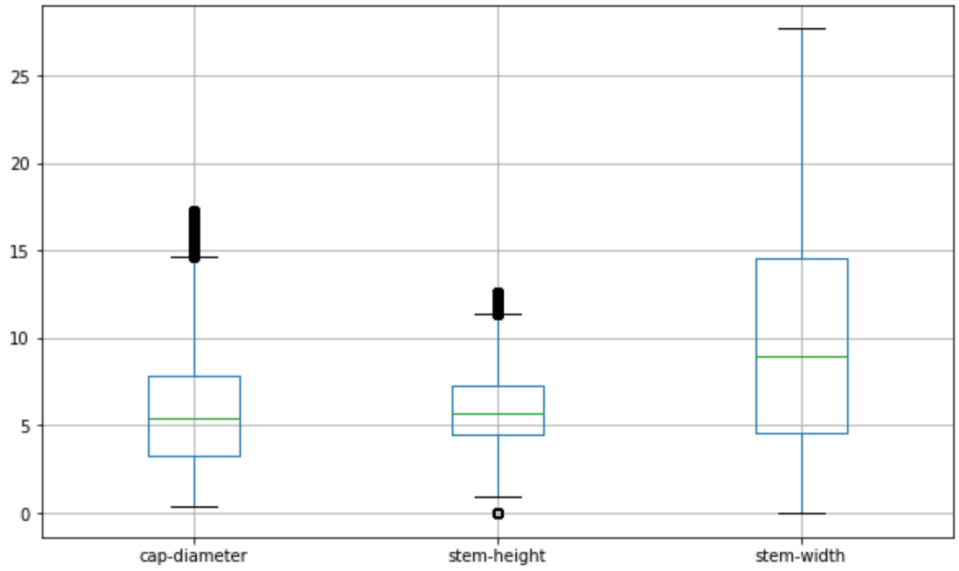


Figura 2.3: boxplot delle variabili numeriche dopo la rimozione degli outliers

	cap-diameter	stem-height	stem-width
count	54614.000000	54614.000000	54614.000000
mean	5.801475	5.953751	10.063094
std	3.276876	2.301243	6.699734
min	0.380000	0.000000	0.000000
25%	3.290000	4.490000	4.560000
50%	5.380000	5.730000	8.920000
75%	7.850000	7.250000	14.540000
max	17.260000	12.560000	27.690000

Figura 2.4: analisi statistica del dataset dopo la rimozione degli outliers

Dalla figura 2.4 è possibile notare come siano stati eliminati 6455 record, pari al 10.57% dell'intero dataset. Abbiamo ritenuto tale perdita di istanze sopportabile, in quanto ha permesso di ottenere dei dati più omogenei su cui lavorare.
Per completezza sono riportate di seguito anche le distribuzioni dei valori delle variabili numeriche.

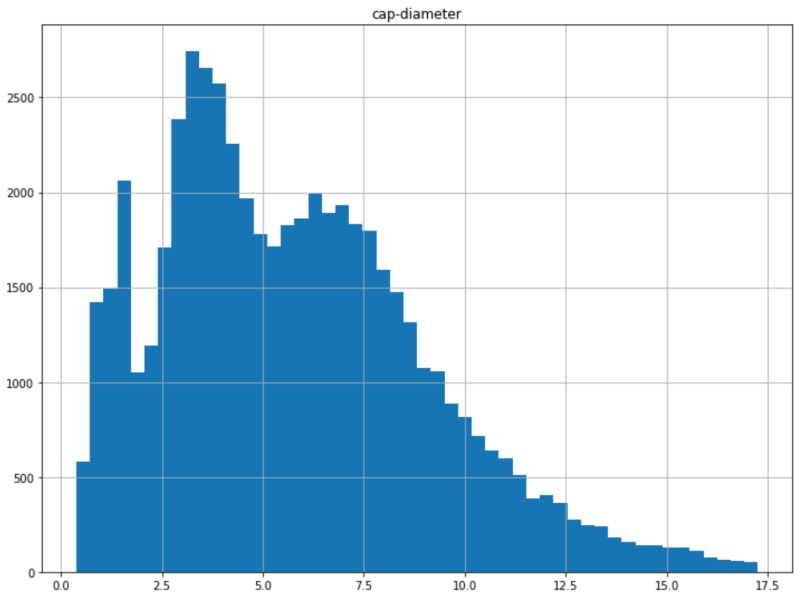


Figura 2.5: distribuzione valori di cap-diameter dopo eliminazione outliers

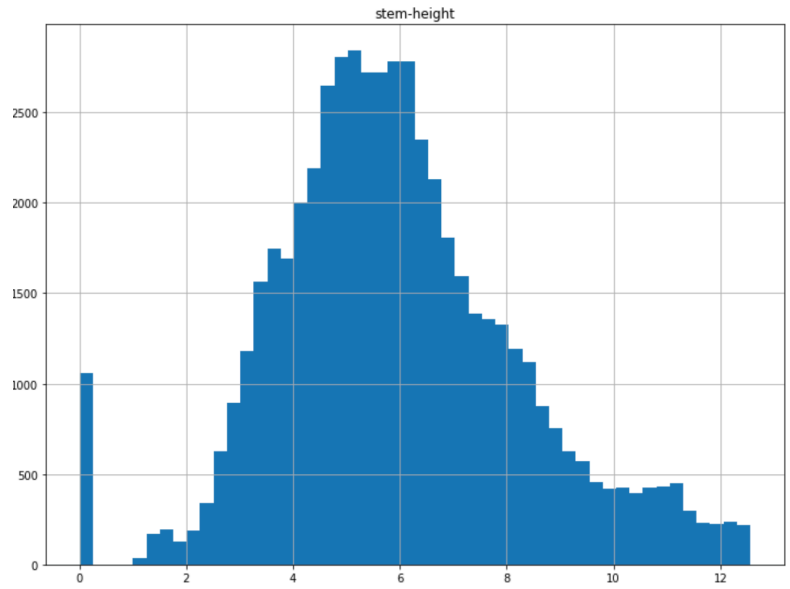


Figura 2.6: distribuzione valori di stem-height dopo eliminazione outliers

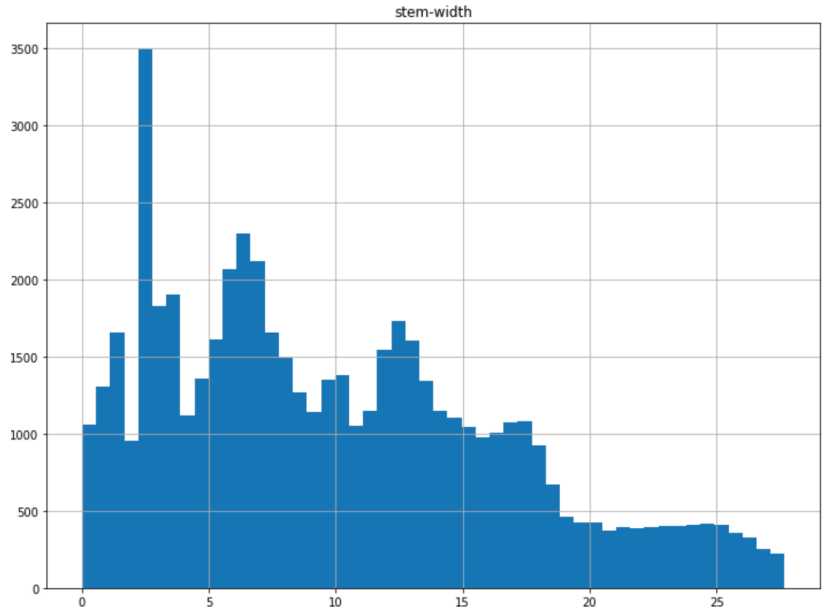


Figura 2.7: distribuzione valori di stem-width dopo eliminazione outliers

2.2 Analisi e ricostruzione missing values

Dopo aver rimosso i valori di outlier, nella seconda fase del preprocessing, abbiamo deciso di analizzare i missing values presenti nel dataset.
Tali valori mancanti, infatti, potrebbero costituire un problema per l'addestramento dei modelli.
Come prima cosa abbiamo calcolato la percentuale di missing values per ogni feature presente nel dataset, e abbiamo ottenuto i seguenti risultati:

- Cap-Surface = 23%
- Gill-Attachment = 16%
- Gill-Spacing = 41%
- Stem-Root = 84%
- Stem-Surface = 62%
- Veil-Type = 95%
- Veil-Color = 88%
- Ring-Type = 4%
- Spore-Print-Color = 90%

Abbiamo successivamente deciso di eliminare le features con un valore di missing values > 50%, in quanto abbiamo ritenuto che non ci fossero informazioni sufficienti per stimare i valori mancanti a partire da quelli presenti. Di conseguenza abbiamo droppato gli attributi Stem-Root, Stem-Surface, Veil-Type, Veil-Color e Spore-Print-Color dal dataset, perdendo quindi informazioni riguardo alla radice e al velo dei funghi.

Dopo la rimozione dei seguenti valori, il dataset si presenta come riportato nella figura 2.8.

```
class                       0
cap-diameter                0
cap-shape                   0
cap-surface             14120
cap-color                   0
does-bruise-or-bleed        0
gill-attachment          9884
gill-spacing            25063
gill-color                  0
stem-height                 0
stem-width                  0
stem-color                  0
has-ring                    0
ring-type                2471
habitat                     0
season                      0
```

Figura 2.8: conto dei missing values nel dataset dopo rimozione iniziale

Abbiamo quindi deciso di rimediare alla presenza dei missing values provando ad utilizzare 4 diverse tecniche, in modo da poter poi decidere in seguito all'addestramento dei modelli quale si è rivelata la migliore.
Abbiamo salvato l'output di ogni tecnica in un nuovo dataframe.

2.2.1 Rimozione totale delle features con missing values

Come prima tecnica provata abbiamo deciso di eliminare tutti gli attributi contenenti missing values. Questo metodo è molto drastico, ma abbiamo ritenuto che potesse funzionare in quanto nel dataset rimarrebbero comunque molte features su cui i modelli potrebbero basarsi per la predizione della classe.

Attraverso il metodo `dataset.dropna()` abbiamo quindi eliminato anche le colonne Cap-Surface, Gill-Attachment, Gill-Spacing e Ring-Type.
Fortunatamente, per tutte e 3 le caratteristiche, Cap, Gill e Ring, abbiamo ancora nel dataset delle features in grado di fornire informazioni, di conseguenza la perdita di dati è solamente parziale.

2.2.2 Ricostruzione missing values tramite most frequent value

La seconda tecnica che abbiamo deciso di provare è la stima dei valori mancanti tramite l'analisi dei valori più frequenti assunti da ogni attributo all'interno del dataset.
Utilizzando tale tecnica abbiamo quindi ottenuto una stima molto rozza, in quanto non vengono usate nozioni di similarità fra le varie istanze del dataset, ma abbiamo eliminato la perdita di informazioni.

Per la ricostruzione dei valori abbiamo utilizzato il metodo `simpleImputer` di `sklearn` con la tecnica di stima "`most-frequent`". Tale scelta è stata obbligata in quanto, essendo i valori mancanti dei valori nominali, non avrebbe avuto senso calcolare la media o la mediana.
Attraverso il codice mostrato in figura 2.9 abbiamo quindi provveduto a ricostruire i valori mancanti.

```
train_set_metodo2 = SimpleImputer(strategy="most_frequent").fit_transform(train_set)
train_set_metodo2 = array_to_dataframe(arr=train_set_metodo2, columns=train_set.columns)
train_set_metodo2
```

Figura 2.9: ricostruzione dei missing values tramite most frequent value

2.2.3 Ricostruzione missing values tramite K nearest neighbors

Per questa terza tecnica abbiamo deciso di utilizzare un metodo di stima più avanzato, che tenesse in considerazione anche la similarità fra le varie istanze del dataset, al fine di cercare di ottenere una ricostruzione dei valori più veritiera.
L'imputazione dei missing values tramite la tecnica del KNN, infatti, consiste nel calcolare la distanza fra le varie istanze del dataset e le istanze contenenti dei missing values, in modo da stimarli successivamente usando il valore medio delle K istanze più vicine a quelle contenenti i missing values.
Per effettuare un'analisi di questo tipo abbiamo dovuto prima mappare le features nominali del nostro dataset in features numeriche, in modo da consentire il calcolo della distanza fra le istanze.

```python
# converto gli attributi da oggetti a numeri con il LabelEncoder
from sklearn.preprocessing import LabelEncoder
class MultiColumnLabelEncoder:

    def __init__(self, columns=None):
        self.columns = columns # array of column names to encode


    def fit(self, X, y=None):
        self.encoders = {}
        columns = X.columns if self.columns is None else self.columns
        for col in columns:
            self.encoders[col] = LabelEncoder().fit(X[col])
        return self


    def transform(self, X):
        output = X.copy()
        columns = X.columns if self.columns is None else self.columns
        for col in columns:
            output[col] = self.encoders[col].transform(X[col])
        return output


    def fit_transform(self, X, y=None):
        return self.fit(X,y).transform(X)


    def inverse_transform(self, X):
        output = X.copy()
        columns = X.columns if self.columns is None else self.columns
        for col in columns:
            output[col] = self.encoders[col].inverse_transform(X[col])
        return output
```

Figura 2.10: implementazione del MultiColumnLabelEncoder

Per effettuare tale mapping abbiamo utilizzato un `MultiColumnLabelEncoder`, la cui implementazione è mostrata nella figura 2.10.
Successivamente abbiamo effettuato la stima tramite il `KNNimputer`, abbiamo castato i valori float ad interi ed abbiamo riconvertito il dataset in formato nominale.
Effettuando il cast abbiamo ovviamente stimato ulteriormente i valori, ma questo ci ha permesso di ricondurci ad una situazione simile al dataset iniziale, perciò abbiamo considerato sopportabile l'errore di arrotondamento derivato da tale cast.

2.2.4 Ricostruzione missing values tramite Bayesan Ridge

L'ultima tecnica che abbiamo tentato per la stima dei missing values è la regressione lineare bayesiana. Tale tecnica consiste nel descrivere la media di una variabile tramite una combinazione lineare di altre variabili al fine di ottenere una probabilità a posteriori, che viene poi utilizzata per effettuare la regressione vera e propria. La distribuzione di probabilità a priori delle variabili viene calcolata in modo analitico a partire dal dataset.
L'ordine con cui vengono stimate le variabili è deciso tramite una schedulazione round Robin.

Anche in questo caso, prima di poter eseguire la stima, è stato necessario mappare le variabili nominali in variabili numeriche.
La stima è stata poi effettuata utilizzando il metodo `IterativeImputer` di `sklearn`.

3. FEATURES SELECTION

Al termine del preprocessing abbiamo deciso di effettuare un processo di selezione delle migliori features in quanto abbiamo ritenuto che quelle originarie del dataset fossero troppe per garantire un addestramento ottimale dei modelli.
Abbiamo quindi svolto diverse analisi sui dataset ottenuti dal preprocessing.

3.1 Analisi della correlazione fra le features

Come primo step abbiamo deciso di analizzare la correlazione fra le features del dataset, in quanto trovando features con un alto livello di correlazione permetterebbe di snellire il dataset, mantenendo solo una delle variabili correlate. Tramite il codice presente in figura 3.1 abbiamo quindi provveduto a creare la matrice delle correlazioni presente nella figura 3.2.

```
plt.figure(figsize=(20,12))
plt.title("Train set metodo 2",fontsize=30,fontweight="bold")
cor = train_set_metodo2_float.corr()
sns.heatmap(cor, annot=True, cmap=plt.cm.Reds)
plt.show()
```

Figura 3.1: codice per generazione matrice delle correlazioni

Dall'analisi dell'output possiamo vedere che le features "Cap-Diameter" e "Stem-Width" presentano una correlazione di 0.83, valore che abbiamo giudicato essere alto. Ciò significa che al variare di una varia anche l'altra.
Abbiamo quindi deciso di eliminare la variabile "Stem-Width" in quanto abbiamo ritenuto che la correlazione fosse abbastanza alta da giustificare la rimozione senza incorrere in una perdita di informazioni eccessiva.

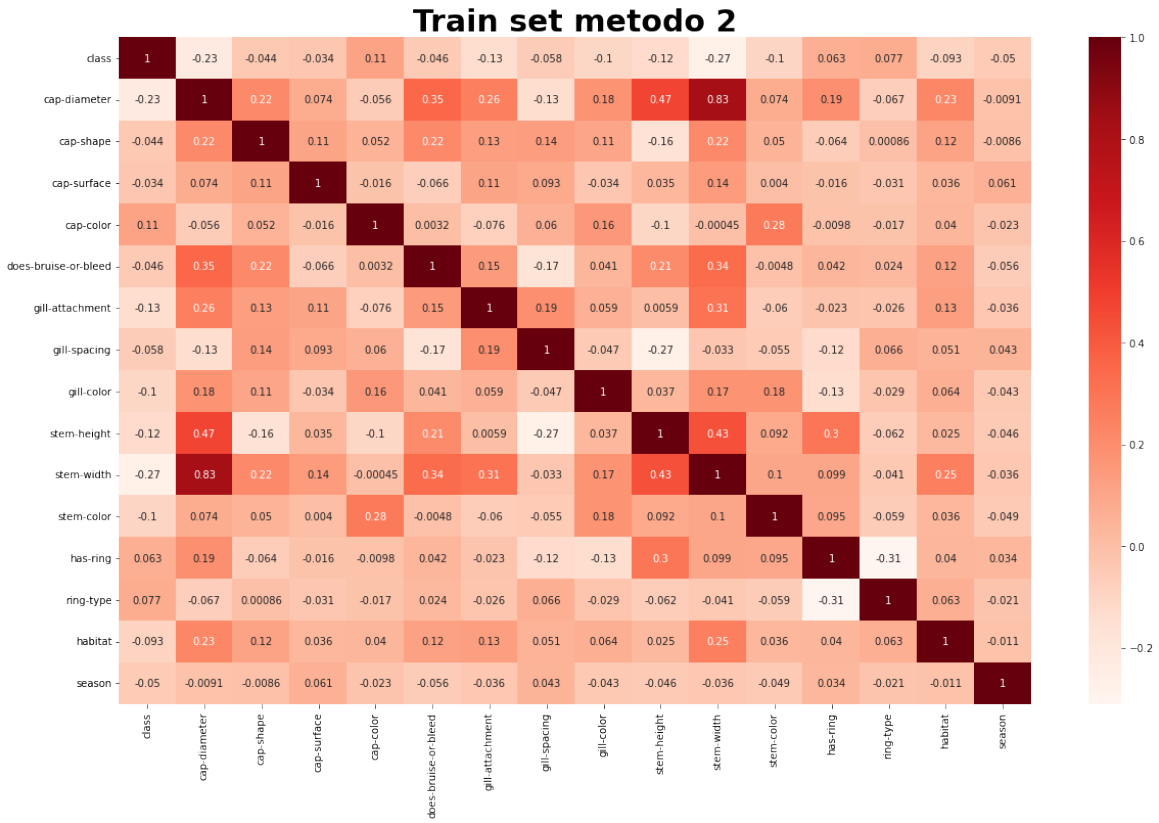


Figura 3.2: matrice delle correlazioni

3.2 Feature Importance tramite Univariate Analysis

Le seconda tecnica che abbiamo deciso di utilizzare è stata una analisi di tipo “Univariate”, ovvero un tipo di analisi che viene eseguita su una variabile alla volta, calcolando degli indici statistici.
In particolare, per le variabili nominali viene calcolata la frequenza e la moda, mentre per quelle numeriche la media e la mediana.
Tali indici vengono poi usati per definire l’importanza di una variabile.
Dall’analisi eseguita in questo modo abbiamo ricavato le variabili ordinate in ordine di importanza e abbiamo notato che l’importanza delle variabili è coerente sui 4 dataset. Ciò è una prova in più riguardo alla bontà della stima dei missing values fatta in precedenza.
Il codice e l’output sono visibili nelle figure seguenti.

```python
def get_feature_univ_scores(univ_fselector, columns, sort_by: str = "score"):
    return pd.DataFrame(
        data=[univ_fselector.scores_, univ_fselector.pvalues_],
        index=["score", "pvalue"],
        columns=columns,
    ).T.sort_values(by=sort_by, ascending=False)
```

```python
def FS_Kbest(ds: pd.DataFrame):
  # Create the feature selector
  kbest = SelectKBest(k=3)
  kbest.fit(ds, ds["class"])
  # Print selected columns
  print(ds.columns[kbest.get_support()])
  # Feature selection scores and pvalues analysis
  print(get_feature_univ_scores(kbest, train_set_metodo4.columns, sort_by="score"))
```

```
Train set metodo 2 Analysis
Index(['class', 'cap-diameter', 'gill-attachment'], dtype='object')
                           score         pvalue
class                        inf   0.000000e+00
cap-diameter         2993.040791   0.000000e+00
gill-attachment       989.118611  3.509822e-215
stem-height           851.141616  1.102041e-185
cap-color             730.323278  8.626208e-160
gill-color            571.438829  1.211418e-125
stem-color            555.374949  3.483447e-122
habitat               475.845873  4.820882e-105
ring-type             328.183022   3.918334e-73
has-ring              216.993271   5.087112e-49
gill-spacing          187.287755   1.461153e-42
season                137.687260   9.314446e-32
does-bruise-or-bleed  117.155817   2.828634e-27
cap-shape             104.050349   2.074446e-24
cap-surface            62.622926   2.549985e-15
```

Figura 3.3: codice per la univariate analysis

Figura 3.4: output della univariate analysis

3.3 Feature Importance tramite Random Forest

Come ultima tecnica abbiamo deciso di utilizzare le Random Forest per visionare l'importanza delle variabili, in quanto esse sono in grado di calcolarla.
Abbiamo quindi provveduto a fittare un `RandomForestClassifier` sui nostri dataset. Il codice e l'output sono visibili nelle figure seguenti.
In particolare abbiamo deciso di utilizzare Gini come misura per lo split, e di effettuare il bootstrap sul dataset.

```
forest2 = RandomForestClassifier(random_state=42,oob_score = True, min_samples_leaf=1,max_features="sqrt",
                                 bootstrap=True,warm_start=True)
forest2.fit(X_train2.values, Y_train2.values)

importances = forest2.feature_importances_
std = np.std([
    tree.feature_importances_ for tree in forest2.estimators_], axis=0)

forest_importances2 = pd.Series(importances, index=X_train2.columns)

fig, ax = plt.subplots()
forest_importances2.plot.bar(yerr=std, ax=ax)
ax.set_title("Feature importances using Gini")
ax.set_ylabel("Mean decrease in impurity")
fig.tight_layout()
```

Figura 3.5: codice per feature importance random forest

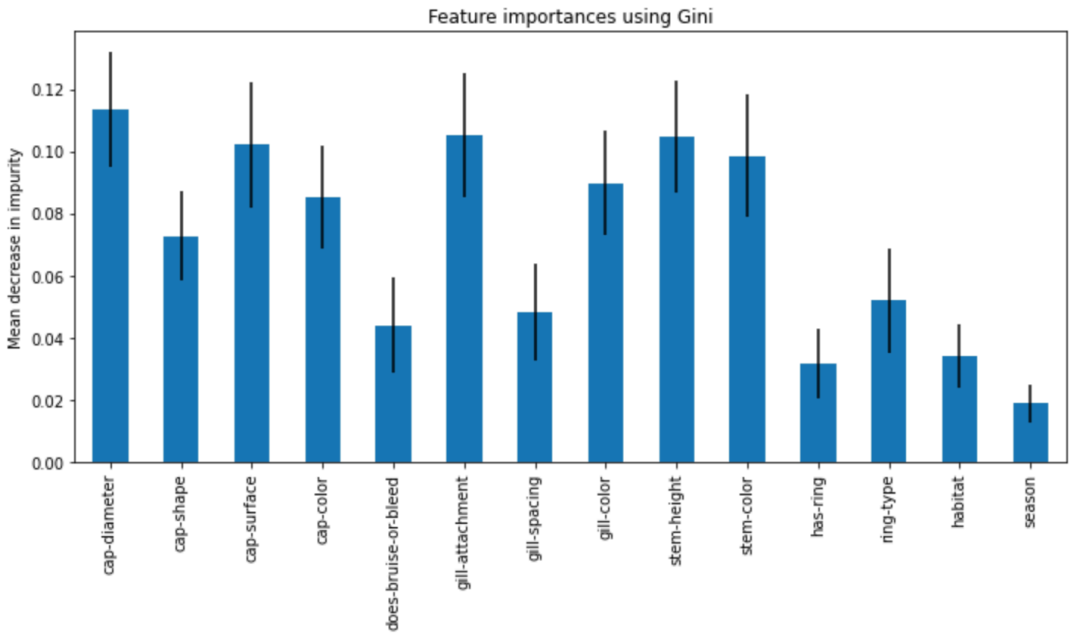


Figura 3.6: output feature importance random forest

In definitiva, dopo le precedenti analisi, abbiamo constatato che i risultati erano coerenti fra loro e che entrambi i metodi utilizzati attribuivano lo stesso ordine di importanza alle variabili.
Di conseguenza abbiamo deciso di droppare le ultime 4 variabili, in modo da raggiungere un numero di features pari a 10, che abbiamo ritenuto giusto per rappresentare una buona parte delle informazioni contenute nel dataset e per addestrare i modelli.

Abbiamo quindi eliminato le seguenti features: “habitat", “season", “ring-type" ,”has-ring”.

4. ADDESTRAMENTO E SCELTA DEI MODELLI

Nell'ultima fase del nostro progetto abbiamo addestrato diversi modelli di Machine Learning al fine di decidere quale fosse il migliore nel task di predizione della classe dei funghi.
Innanzitutto abbiamo splittato i 4 dataset ottenuti dalle tecniche di stima dei missing values in train test e test set.
Successivamente abbiamo addestrato i seguenti modelli.

4.1 Support Vector Machine

Il primo modello che abbiamo provato è quello delle Support Vector Machine. In particolare abbiamo provato le seguenti configurazioni:

- SVM con kernel lineare e C=1
- SVM con kernel RBF e C=10

In particolare abbiamo scelto di aumentare il valore di C al secondo tentativo poiché nel nostro dataset, nonostante la pulizia, sono rimasti dei valori di outlier.

```
#Utilizziamo come primo modello una SVC con kernel lineare.
svc=SVC(kernel='linear',C=1)

# fit classifier to training set1
svc.fit(X_train1,y_train1)

# make predictions on test set1
y_pred1=svc.predict(X_test1)
```

Figura 4.1: codice SVM lineare

```
svc=SVC(C=10)

# fit classifier to training set1
svc.fit(X_train1,y_train1)

# make predictions on test set1
y_pred1=svc.predict(X_test1)
```

Figura 4.2: codice SVM con kernel Gaussiano

```
Model 1 accuracy score with default hyperparameters with linear kernel (dataset1): 0.7361
Model 2 accuracy score with default hyperparameters with linear kernel (dataset2): 0.6285
Model 3 accuracy score with default hyperparameters with linear kernel (dataset3): 0.6327
Model 4 accuracy score with default hyperparameters with linear kernel (dataset4): 0.6276

Model 1 accuracy score with default hyperparameters with C=10 (dataset1): 0.9914
Model 2 accuracy score with default hyperparameters with C=10 (dataset2): 0.9563
Model 3 accuracy score with default hyperparameters with C=10 (dataset3): 0.9469
Model 4 accuracy score with default hyperparameters with C=10 (dataset4): 0.9552
```

Figura 4.3: risultati con entrambe le configurazioni

Come si può evincere dai risultati rappresentati nella figura 4.3, il modello con il kernel lineare non ha dato buoni risultati. Al contrario, il secondo modello, quello con kernel gaussiano e penalizzazione per gli outlier = 10 ha dato ottimi risultati. Questo probabilmente è dovuto ad una migliore separabilità delle classi in una dimensione maggiore rispetto a quella standard.

Un altro spunto di riflessione è dato dal fatto che i risultati migliori, in entrambi i casi, sono ottenuti sul primo dataset, quello ottenuto per rimozione delle features contenenti missing values. L'under-performance dei restanti 3 dataset potrebbe essere ricondotta alle stime e alle approssimazioni fatte nella stima dei missing values, che potrebbero aver "confuso" il modello durante l'addestramento.

4.2 Extra Trees

Avendo già utilizzato le Random Forest per la selezione delle variabili nella fase di feature selection, abbiamo optato per l'utilizzo del modello Extra Trees in questa fase.
Tale modello è simile alle Random Forest, ma aggiunge ancora più randomizzazione.
Anche in questo caso abbiamo utilizzato Gini come misura di split, e abbiamo deciso di utilizzare 30 stimatori.
Il codice e i risultati sono riportati di seguito.

```
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.model_selection import train_test_split

extra_tree_forest = ExtraTreesClassifier(oob_score=True, bootstrap=True,n_estimators=30,
                            criterion = "gini", max_features = "auto", random_state=42)

extra_tree_forest.fit(X_train1,y_train1)
y_pred1 = extra_tree_forest.predict(X_test1)
feature_importance1 = extra_tree_forest.feature_importances_
```

Figura 4.4: codice Extra Trees

```
Model 1 accuracy score with extra trees (dataset1): 0.9996
Model 2 accuracy score with extra trees (dataset2): 0.9975
Model 3 accuracy score with extra trees (dataset3): 0.9987
Model 4 accuracy score with extra trees (dataset4): 0.9990
```

Figura 4.5: risultati Extra Trees

Anche in questo caso il dataset che ha ottenuto i migliori risultati è stato il primo, come pronosticato. In generale però con questo modello tutti e 4 i dataset si sono comportati bene, raggiungendo uno score di accuracy superiore al 99%. Inoltre tale modello, rispetto al precedente, ha fornito i risultati sopracitati in un tempo decisamente più ridotto.
Abbiamo anche valutato nuovamente l'importanza delle variabili, ottenendo un risultato coerente con quanto trovato con le RF.

4.3 Bagging

La terza tecnica provata è stata una tecnica di tipo Ensemble, in particolare un Bagging Classifier che utilizza 500 Decision Tree.

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

clf = DecisionTreeClassifier(random_state=424)
clf.fit(X_train1,y_train1)
ypred1_albero = clf.predict(X_test1)
print(clf.tree_.max_depth)

bag_clf = BaggingClassifier(
    DecisionTreeClassifier(random_state=424), n_estimators=500,
      bootstrap=True, random_state=42)
bag_clf.fit(X_train1, y_train1)
y_pred1 = bag_clf.predict(X_test1)
```

Figura 4.6: codice Bagging

```
Model 1 accuracy score with bagging with 1 tree (dataset1): 0.9960
Model 1 accuracy score with bagging with 500 trees (dataset1): 0.9968
Model 2 accuracy score with bagging with 500 trees (dataset2): 0.9961
Model 3 accuracy score with bagging with 500 trees (dataset3): 0.9952
Model 4 accuracy score with bagging with 500 trees (dataset4): 0.9971
```

Figura 4.7: risultati Bagging

Come si può notare nella figura 4.7, anche questa tecnica ha fornito ottimi esiti, raggiungendo un'accuratezza del 99%.
Abbiamo tuttavia notato che tale score viene raggiunto anche da un classificatore con un singolo albero, ciò significa che il lavoro fatto dal bagging risulta inutile.
Questo è probabilmente dovuto alla semplicità del dataset in uso.

4.4 Neural Networks

La quarta tecnica testata è quella delle Neural Networks. Abbiamo deciso di provare anche questo modello in quanto, rispetto ai precedenti, si tratta di un modello blackbox, ovvero di un modello che non permette di capire come si giunge al risultato finale.

Abbiamo utilizzato una struttura molto semplice e basilare, utilizzando una rete Feed Forward con 6 livelli densi, funzione di attivazione relu, dropout finale con tasso di 0.2 e livello di classificazione finale con softmax.
Abbiamo poi compilato la rete utilizzando Adam come optimizer e l'abbiamo addestrata per 10 epoche.
La rete, il codice e i risultati sono visibili nelle figure seguenti.

```
numhiddens=5
numhneurons=50
model = tf.keras.Sequential()
model.add(tf.keras.layers.Flatten(input_dim=X_train1.shape[1]))
model.add(tf.keras.layers.Dense(200, activation='relu'))
for _ in range(numhiddens):
    model.add(tf.keras.layers.Dense(numhneurons, activation='relu'))
model.add(tf.keras.layers.Dropout(0.2))
model.add(tf.keras.layers.Dense(2,activation='softmax'))
```

Figura 4.8: codice rete neurale

```
157/157 - 0s - loss: 0.0075 - accuracy: 0.9978 - 260ms/epoch - 2ms/step

Test accuracy: 0.9978026151657104
```

Figura 4.9: risultato rete neurale

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten (Flatten)           (None, 10)                0

 dense (Dense)               (None, 200)               2200

 dense_1 (Dense)             (None, 50)                10050

 dense_2 (Dense)             (None, 50)                2550

 dense_3 (Dense)             (None, 50)                2550

 dense_4 (Dense)             (None, 50)                2550

 dense_5 (Dense)             (None, 50)                2550

 dropout (Dropout)           (None, 50)                0

 dense_6 (Dense)             (None, 2)                 102

=================================================================
Total params: 22,552
Trainable params: 22,552
Non-trainable params: 0
_________________________________________________________________
```

Figura 4.10: struttura rete neurale

Anche in questo caso i risultati sono stati ottimi. Probabilmente tale modello è anche over-dimensionato rispetto alla difficoltà del problema, che poteva essere risolto anche tramite tecniche molto più semplici, come visto in precedenza.

4.5 Auto-Sklearn

L'ultima tecnica che abbiamo testato è stata Auto-Sklearn, un metodo in grado di eseguire il compito di scelta del modello più adeguato in modo automatico. Abbiamo testato anche questa tecnica più che altro per avere un confronto con quanto fatto da noi nella selezione e nel tuning manuale dei modelli.
Abbiamo deciso di impostare il metodo con 4 diversi modelli fra cui scegliere e con 3 tipi diversi di preprocessing., come visibile nella figura sottostante.

```
import autosklearn.classification
import sklearn.metrics
from sklearn.model_selection import train_test_split, StratifiedKFold
from autosklearn.classification import AutoSklearnClassifier
from autosklearn.metrics import (accuracy,
                                 f1,
                                 roc_auc,
                                 precision,
                                 average_precision,
                                 recall,
                                 log_loss)
```

```
clf = AutoSklearnClassifier(
    time_left_for_this_task=300,
    per_run_time_limit=30,
    include={
        "classifier": ["decision_tree", "extra_trees", "random_forest","adaboost"],
        "feature_preprocessor": [
            "no_preprocessing",
            "polynomial",
            "select_percentile_classification",
        ],
    },
    ensemble_kwargs={"ensemble_size": 1},
)
```

```
clf.fit(X_train1_cat, y_train1_cat)
```

Figura 4.11: codice Auto-Sklearn

Il risultato migliore è stato ottenuto dal modello Random Forest con nessun tipo di preprocessing, tale modello ha ottenuto un'accuracy di 0.9998.
Tale risultato ci ha permesso di validare il preprocessing manuale fatto da noi e illustrato nelle pagine precedenti, in quanto il metodo automatico non ha ritenuto necessario effettuarne di ulteriore.
Il risultato ottenuto si posiziona poi in linea con i risultati ottenuti precedentemente anche dai modelli addestrati in modo manuale da noi.

5. CONCLUSIONI

In conclusione abbiamo potuto constatare che tutti i modelli, ad eccezione delle SVM con kernel lineare, hanno prodotto risultati ottimi nel test di classificazione binaria su tutti e 4 dataset ottenuti dopo la stima dei missing values.
Questo è probabilmente dovuto al preprocessing molto forte fatto sul dataset, che lo ha messo in condizione di essere appreso bene dai modelli, e dalla semplicità dello stesso.
Probabilmente i modelli che abbiamo addestrato hanno individuato un piccolo subset di variabili in grado di predire sempre correttamente il valore della classe, ciò ci ha portati a pensare che saremmo quindi potuti essere anche più aggressivi nella fase di feature selection.

Infine è importante notare che tutte le tecniche di stima dei missing values hanno portato a risultati coerenti e comparabili, questo può significare che:

- le tecniche prese in esame hanno funzionato correttamente nonostante l'approssimazione dovuta alla trasformazione delle features da nominali a numeriche.
- le variabili di cui sono stati stimati i missing values non erano importanti ai fini della classificazione.