Spambase, a binary classification problem

Alessandro Soccol[1], Marco Cosseddu [2], Giovanni Murgia[3], Pietro Cosseddu[4], Arturo Rodriguez[5]

Università degli Studi di Cagliari
Corso di Laurea in Informatica Applicata e Data Analytics 2022/2023

ABSTRACT

Spambase è un dataset contenente numerose email classificate come spam (1) o non spam (0), sulla base di criteri come la presenza di specifiche parole o caratteri.
Il nostro obiettivo, oltre all'analisi dei dati e della loro qualità, è quello di documentare come il dataset reagisce all'utilizzo di diverse tecniche di preprocessing e classificazione, al fine di creare modelli in grado di fornire buoni risultati di predizione e confrontarli tra loro.
Due degli algoritmi di classificazione usati sono implementati da noi, gli altri tre fanno riferimento a funzioni della libreria scikit-learn. Per il preprocessing e l'analisi dei dati, ci siamo affidati alle librerie pandas, numpy e imbalanced-learn.

1 Introduzione: Il Dataset

Il dataset, risalente al 1999, proviene dai laboratori della HP (Hewlett-Packard), e contiene diverse email classificate come spam (1) o non spam (0) sulla base di criteri come la presenza di specifiche parole o caratteri.

Il dataset contiente 4601 istanze, delle quali 1813 sono etichettate come spam (39.4%), e 2788 sono etichettate come non spam (60.6%). Notiamo un leggero sbilanciamento tra le due classi.

Gli attributi sono 58, e assumono valori continui, a differenza dell'ultimo, che indica l'etichetta di classe. I primi 48 attributi rappresentano la percentuale di frequenza di specifiche parole all'interno dell'e-mail, e assumono valori nell'insieme dei numeri reali in un intervallo [0,100]. Nello specifico sono calcolati secondo la seguente formula: *100 * (numero di volte che la parola appare nella e-mail)/totale di parole nell'e-mail*.
I successivi 6 attributi rappresentano invece la percentuale di frequenza di specifici caratteri all'interno dell'e-mail, anch'essi assumono valori nell'intervallo reale [0,100] e sono calcolati analogamente con la seguente formula: *100 * (numero di volte che il carattere appare nella e-mail)/totale di caratteri nell'e-mail*. Gli attributi "capital_run_length_average", "capital_run_length_longest", "capital_run_length_total", rappresentano rispettivamente la lunghezza media delle sequenze ininterrotte di elttere maiuscole, la lunghezza della più lunga sequenza ininterrotta di lettere maiuscole e il numero totale di lettere maiuscole.
L'ultimo attributo è un attributo nominale e indica se l'email è catalogata come spam (1) o meno (0).

[1] Matricola: 00057
[2] Matricola: 00010
[3] Matricola: 00047
[4] Matricola: 00008
[5] Matricola: 00054

1.1 Fasi preliminari

Abbiamo creato un modulo chiamato “libraries.py”, contenente tutti gli import delle librerie che ci serviranno nel corso del programma. Per ora facciamo riferimento alle librerie pandas e numpy.

Il dataset si trova all’interno di un file chiamato “spambase.data”, leggibile in formato csv cambiando l’estensione in “.csv”. I nomi degli attributi si trovano in un altro file chiamato “spambase.names” che contiene anche altre informazioni riguardo la struttura del dataset. Per comodità abbiamo separato i nomi degli attributi con il relativo tipo di dato in un altro file chiamato “features.txt”.

Abbiamo aperto il file “features.txt” in lettura, eliminando i new line, separando il nome degli attributi dal tipo con il delimitatore “:” e ottenendo una lista della forma [“<nome attributo>”,”<tipo>”,”<nome attributo2>,”<tipo2>”,….,”<nome attributo n>”,”<tipon>”] chiamata “raw”. Dopodichè abbiamo separato il tipo di dato dai nomi delle features veri e propri, in due liste separate, rispettivamente: “dataType” e “features”. Alla lista features, dato che non è stato specificato come le altre features il nome dell’attributo che identifica le rispettive etichette di classe, abbiamo aggiunto la label “spam”.

```
# Otteniamo i nomi delle attributi con il tipo, separatamente
with open("spambase/features.txt","r") as f:
    lines=f.readlines()
# Dati in formato raw da manipolare che contengono il nome dell'attributo separato
da ":" e il tipo dell'attributo
raw=[x.strip().split(":") for x in lines]
# Attributi
features=[x[0] for x in raw]+["spam"]
# Tipi di dato
dataType=[x[1].strip()[:-1] for x in raw]
```

Con la funzione “read_csv”, appartenente alla libreria pandas, abbiamo aperto il nostro dataset, specificando i parametri “names=features” e “index_cols=false”, rispettivamente per assegnare manualmente i nomi degli attributi estratti precedentemente, altrimenti avrebbe usato i dati stessi per identificarli, e per evitare che usi il primo attributo come indice per i dati stessi.

```
# Leggiamo il dataset
df=pd.read_csv("spambase/spambase.csv",names=features,index_col=False)
```

	word_freq_make	word_freq_address	word_freq_all	word_freq_3d	word_freq_our	word_freq_over	word_freq_remove	word_freq_internet	wor
0	0.00	0.64	0.64	0.0	0.32	0.00	0.00	0.00	
1	0.21	0.28	0.50	0.0	0.14	0.28	0.21	0.07	
2	0.06	0.00	0.71	0.0	1.23	0.19	0.19	0.12	
3	0.00	0.00	0.00	0.0	0.63	0.00	0.31	0.63	
4	0.00	0.00	0.00	0.0	0.63	0.00	0.31	0.63	
...	...	...	...	...	...	...	...	...	
4596	0.31	0.00	0.62	0.0	0.00	0.31	0.00	0.00	
4597	0.00	0.00	0.00	0.0	0.00	0.00	0.00	0.00	
4598	0.30	0.00	0.30	0.0	0.00	0.00	0.00	0.00	
4599	0.96	0.00	0.00	0.0	0.32	0.00	0.00	0.00	
4600	0.00	0.00	0.65	0.0	0.00	0.00	0.00	0.00	

4601 rows × 58 columns

Dopodichè, con un'indicizzazione adeguata, separiamo i dati (X) dalle etichette di classe (y). Creiamo un vettore "label" contenente le etichette di classe, trasformiamo il vettore y in un vettore numpy ed infine mostriamo la shape della matrice X.

```
# Separiamo gli attributi dall'attributo che definisce l'etichetta di classe
X=df.iloc[:,:-1] # Non prendiamo l'ultimo attributo (etichetta di classe)
y=df.iloc[:,-1] # Prendiamo in considerazione solo l'attributo che definisce
l'etichetta di classe
label=np.unique(y) #
y=y.to_numpy() # Trasformiamo in un vettore numpy
X.shape
```

2 Menù iniziale

All'esecuzione del file "main.py", viene mostrato un menù che permette di scegliere uno dei cinque classificatori che abbiamo scelto oppure di fare un'analisi dei dati. Per ogni classificatore, c'è la possibilità di fare o meno del preprocessing, facendo una trasformazione dei dati, usando metodi di riduzione della dimensionalità o di bilanciamento delle classi.

3 Analisi dei dati e della loro qualità

Per quanto riguarda l'analisi dei dati e la loro qualità, viene fatta una prima panoramica grazie alla libreria pandas, ed in particolare alla funzione ".describe()", che restituisce una matrice descrittiva di misure di posizione e dispersione per ogni attributo.

```
X.describe()
```

	word_freq_make	word_freq_address	word_freq_all	word_freq_3d	word_freq_our	word_freq_over	word_freq_remove	word_freq_internet	wo
count	4601.000000	4601.000000	4601.000000	4601.000000	4601.000000	4601.000000	4601.000000	4601.000000	
mean	0.104553	0.213015	0.280656	0.065425	0.312223	0.095901	0.114208	0.105295	
std	0.305358	1.290575	0.504143	1.395151	0.672513	0.273824	0.391441	0.401071	
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	
25%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	
50%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	
75%	0.000000	0.000000	0.420000	0.000000	0.380000	0.000000	0.000000	0.000000	
max	4.540000	14.280000	5.100000	42.810000	10.000000	5.880000	7.270000	11.110000	

Figura 1 Matrice descrittiva delle features, output non per intero

Abbiamo poi cercato la presenza o meno di valori nulli, usando sempre la libreria pandas, ed in particolare la funzione ".info()" che ci mostra per ogni attributo il numero di valori non nulli.

```
X.info()
```

```
Output exceeds the size limit. Open the full output data in a text editor
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4601 entries, 0 to 4600
Data columns (total 57 columns):
 #   Column                      Non-Null Count  Dtype
---  ------                      --------------  -----
 0   word_freq_make              4601 non-null   float64
 1   word_freq_address           4601 non-null   float64
 2   word_freq_all               4601 non-null   float64
 3   word_freq_3d                4601 non-null   float64
 4   word_freq_our               4601 non-null   float64
 5   word_freq_over              4601 non-null   float64
 6   word_freq_remove            4601 non-null   float64
 7   word_freq_internet          4601 non-null   float64
 8   word_freq_order             4601 non-null   float64
 9   word_freq_mail              4601 non-null   float64
 10  word_freq_receive           4601 non-null   float64
 11  word_freq_will              4601 non-null   float64
 12  word_freq_people            4601 non-null   float64
 13  word_freq_report            4601 non-null   float64
 14  word_freq_addresses         4601 non-null   float64
 15  word_freq_free              4601 non-null   float64
 16  word_freq_business          4601 non-null   float64
 17  word_freq_email             4601 non-null   float64
 18  word_freq_you               4601 non-null   float64
 19  word_freq_credit            4601 non-null   float64
```

Figura 2 Output della funzione info(), non per intero

Abbiamo usato poi due funzioni della libreria pandas combinate, per essere certi della mancanza di valori nulli per ogni attributo. "isnull()" controlla se sono presenti valori come "NaN", "None" o "NaT", esso è unito insieme ad "any" che ci permette di controllare lungo un asse, in questo caso "axis=0" per indicare le colonne.

```
#https://pandas.pydata.org/docs/reference/api/pandas.isnull.html
#https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.any.html
X.isnull().any(axis=0)
```

```
Output exceeds the size limit. Open the full output data in a text editor
word_freq_make                False
word_freq_address             False
word_freq_all                 False
word_freq_3d                  False
word_freq_our                 False
word_freq_over                False
word_freq_remove              False
word_freq_internet            False
word_freq_order               False
word_freq_mail                False
word_freq_receive             False
word_freq_will                False
word_freq_people              False
word_freq_report              False
word_freq_addresses           False
word_freq_free                False
word_freq_business            False
word_freq_email               False
word_freq_you                 False
word_freq_credit              False
word_freq_your                False
word_freq_font                False
word_freq_000                 False
word_freq_money               False
word_freq_hp                  False
word_freq_hpl                 False
word_freq_george              False
word_freq_650                 False
```

Figura 3 Output non per intero

Abbiamo poi controllato la presenza di outliers, guardando i valori che per ogni attributo si trovano oltre il 90esimo percentile e prima del decimo percentile.

```
X[(X<X.quantile(0.10)) | (X>X.quantile(0.90))]
```

	word_freq_make	word_freq_address	word_freq_all	word_freq_3d	word_freq_our	word_freq_over	word_freq_remove	word_freq_internet	wor
0	NaN	0.64	NaN	NaN	NaN	NaN	NaN	NaN	
1	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
2	NaN	NaN	NaN	NaN	1.23	NaN	NaN	NaN	
3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	0.63	
4	NaN	NaN	NaN	NaN	NaN	NaN	NaN	0.63	
...	...	...	...	...	...	...	...	...	
4596	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
4597	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
4598	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
4599	0.96	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
4600	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	

4601 rows × 57 columns

Un secondo controllo, individua come outliers gli elementi che si trovano oltre i baffi di un boxplot. In output la matrice mostra “NaN” per tutti i valori che non sono outliers.

```
Q1=X.quantile(0.25)
Q3=X.quantile(0.75)
IQR=Q3-Q1
outliersMatrix=X[(X<Q1-1.5*IQR) | (X>Q3+1.5*IQR) & (X<X.quantile(0.10)) |
(X>X.quantile(0.90))]
outliersMatrix
```

	word_freq_make	word_freq_address	word_freq_all	word_freq_3d	word_freq_our	word_freq_over	word_freq_remove	word_freq_intern
0	NaN	0.64	NaN	NaN	NaN	NaN	NaN	Na
1	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Na
2	NaN	NaN	NaN	NaN	1.23	NaN	NaN	Na
3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	0.6
4	NaN	NaN	NaN	NaN	NaN	NaN	NaN	0.6
...	...	...	...	...	...	...	...	
4596	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Na
4597	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Na
4598	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Na
4599	0.96	NaN	NaN	NaN	NaN	NaN	NaN	Na
4600	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Na

4601 rows × 57 columns

Abbiamo contato, per entrambi i metodi, quanti outliers ci sono per ogni attributo sottraendo il numero di records al numero di valori “NaN” nella matrice precedente.

```
outliersCount=X.shape[0]-outliersMatrix.isnull().sum()
outliersCount
```

```
Output exceeds the size limit. Open the full output data in a text editor
word_freq_make                458
word_freq_address             455
word_freq_all                 454
word_freq_3d                   47
word_freq_our                 459
word_freq_over                446
word_freq_remove              460
word_freq_internet            448
word_freq_order               459
word_freq_mail                455
word_freq_receive             459
word_freq_will                460
word_freq_people              458
word_freq_report              357
word_freq_addresses           336
word_freq_free                456
word_freq_business            458
word_freq_email               459
word_freq_you                 459
word_freq_credit              424
word_freq_your                460
word_freq_font                117
word_freq_000                 457
word_freq_money               460
word_freq_hp                  454
word_freq_hpl                 459
word_freq_george              460
word_freq_650                 460
word_freq_lab                 372
word_freq_labs                459
```

Figura 4 Output non per intero

Abbiamo infine creato una matrice di correlazione e usando il colore “coolwarm”, abbiamo individuato gli attributi con un alto coefficiente.

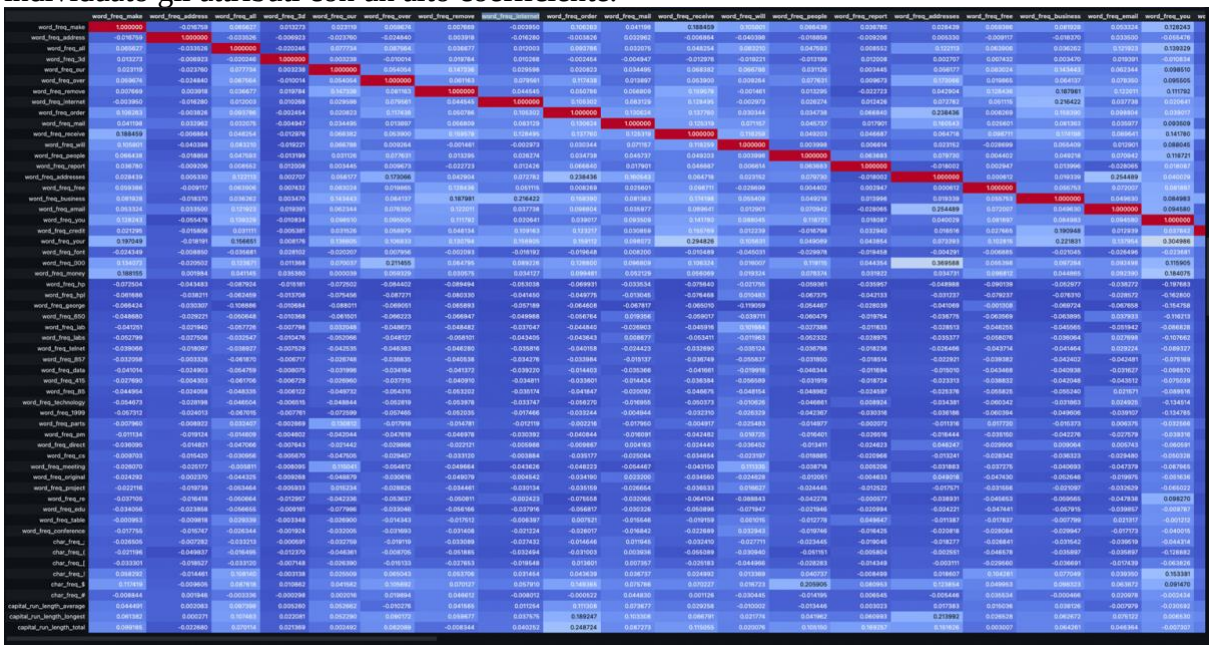

Figura 5 Output non per intero

Osservando la matrice di correlazione per intera, possiamo notare che ci sono diversi attributi con un’alta correlazione (per esempio maggiore di 0.8), ci è utile saperlo per le successive fasi di feature selection.

4 Il primo classificatore: Random Forest

Come primo classificatore dei tre a scelta forniti da Scikit-learn, abbiamo deciso di utilizzare il, un classificatore multiplo molto usato. Questo classificatore addestra diversi alberi decisionali, che casualmente creano un modello su una partizione random delle feature e del dataset. L’idea alla base del Random Forest è che gli alberi si specializzano su parti diverse del dataset, e una volta combinati offrono una predizione molto precisa.

Prima di tutto otteniamo i dati dalla funzione csv_open() importata dal file csv_open.py, e facciamo un campionamento stratificato per ottenere i dati di training e di test, partizionando il 25% dei dati come test set ed il restante 75% dei dati come training set.

```
X, y, dataType, features = csv_open()
train_x, test_x, train_y, test_y = train_test_split(X, y, random_state=0,
test_size=0.25, stratify=y)
```

Creiamo un oggetto classificatore RandomForestClassifier, senza specificare nessun parametro.

```
rf = RandomForestClassifier()
```

Definiamo un dizionario contenente gli iperparametri da testare usando GridSearchCV.

```
params={"n_estimators":list(range(10,80,10)),# Quanti alberi decisionali nella
random forest?
            "criterion":["gini","entropy","log_loss"], # Quale misura considerare
per valutare la qualità di uno split?
```

```
            "max_depth":[None]+list(range(4,34,10)), # Quale massima profondità
consideriamo? None significa che costruire l'intero albero decisionale senza un
limite alla profondità.
            "max_features":["sqrt","log2"] # Quante feature consideriamo quando
cerchiamo lo split migliore?
            }
```

- "n_estimator" indica il numero di alberi decisionali della random forest;
- "criterion" indica quali misure considerare per valutare la qualità di uno split;
- "max_depth" indica la massima profondità degli alberi decisionali,;
- "max_features" indica quante features consideriamo quando cerchiamo lo split migliore.

Avremmo potuto includere altri iperparametri, per esempio il peso delle classi o "Bootstrap" che indica se devono essere usati campioni di bootstrap per costruire gli alberi decisionali.

Effettuiamo la GridSearch e salviamo in results i vari modelli ottenuti:

```
rf_grid = GridSearchCV(estimator=rf, cv=10,param_grid=params,n_jobs=-1)
results = rf_grid.fit(train_x, train_y)
```

GridSearchCV ha come parametri: "estimator" a cui è assegnato il classificatore su cui fare il tuning; il numero di iperparametri ed i valori con il parametro "param_grid=params" che permette a GridSearchCV di testare tutti gli iperparametri descritti sopra; "cv=10" indica di usare la ten-fold cross validation ed in particolare se specificato un valore intero esegue una Stratified Cross-Validation, come specificato nella documentazione; ed infine "n_jobs=-1" indica di usare tutti i processori.

Creiamo e addestriamo un nuovo classificatore, avente i parametri del modello che ha dato i migliori risultati dopo la GridSearch. "best_params_" è un attributo di un oggetto GridSearchCV che contiene un dizionario contente i parametri migliori per il Random Forest.

```
rf_best = rf.set_params(**results.best_params_)
```

Rieseguiamo l'addestramento usando i parametri ottimali

```
rf_best.fit(train_x, train_y)
```

Facciamo una predizione

```
pred_y = rf_best.predict(test_x)
```

Calcoliamo delle metriche, che ci mostrano le performance del classificatore finale usando la funzione metrics che calcola al suo interno una matrice di confusione che sfrutta per calcolare TPR, TNR, FPR, FNR, Precision, Recall, F-Score. Stampiamo a video anche i parametri migliori e l'accuracy, che risulta abbastanza affidabile dato il leggero sbilanciamento.

```
print("#### Performance Random Forest ####\n")
print("\nBEST PARAMS:",results.best_params_)
metrics(test_y,pred_y,np.unique(test_y))
```

Performances ottenute senza preprocessing:

TPR	TNR	FPR	FNR	Precision	Recal	F-Score
0.98	**0.94**	**0.06**	**0.02**	**0.96**	**0.98**	**0.97**

5 Il secondo classificatore: kNearestNeighbour

Abbiamo deciso di utilizzare il classificatore kNearestNeighbour, come secondo classificatore a scelta fornito dalla libreria Scikit-Learn.

Questo classificatore basato su distanze è uno dei più conosciuti, dato il suo classico utilizzo come esempio accademico. Il kNearestNeighbour tuttavia, a differenza della maggior parte dei classificatori non fa una vera e propria induzione di un modello, ma si limita a copiare in memoria il training set integralmente e ad eseguire un confronto tra l'elemento da classificare e tutti gli oggetti memorizzati. Questo processo, genera una matrice delle distanze, tra gli esempi del training set e il record da classificare, e a quest'ultimo verrà assegnata la label di classe maggioritaria corrispondente ai k oggetti estratti, con distanza minore. Nel caso semplice dell' 1-NearestNeighbour (k=1), viene assegnata la label dell'elemento più simile tra tutti quelli presenti nel training set.

Prima di tutto otteniamo i dati dalla funzione csv_open() importata dal file csv_open.py, e facciamo un campionamento stratificato per ottenere i dati di training e di test, partizionando il 25% dei dati come test set ed il restante 75% dei dati come training set.

```
X, y, dataType, features = csv_open()
train_x, test_x, train_y, test_y = train_test_split(X, y, random_state=0,
test_size=0.25, stratify=y)
```

Creiamo un oggetto KNeighboursClassifier, rappresentante il classificatore da utilizzare, senza nessun parametro.

```
kNN = KNeighborsClassifier()
```

Definiamo un dizionario contentente varie combinazioni di iperparametri, che verranno poi testate dalla GridSearchCV.

```
params={
            'n_neighbors':[1,100],
            'weights':('uniform','distance'
            'metric':('cityblock', 'euclidean', 'cosine'
            }
```

- "n_neighbors" indica il numero di elementi da prendere in considerazione per l'assegnazione della classe (k);

- “weights” indica il peso da assegnare ai k più vicini presi in considerazione, se è “uniform” tutti gli oggetti sono pesati allo stesso modo, se è “distance” il peso è inversamente proporzionale alla distanza dall’oggetto da classificare, in modo tale da favorire l’influenza dei record più vicini;
- “metric” indica il tipo di metrica da utilizzare per calcolare la distanza.

Effettuiamo la GridSearch e salviamo in results i vari modelli ottenuti:

```
kNN_grid = GridSearchCV(kNN, params, cv=10, n_jobs=-1)
results = kNN_grid.fit(train_x, train_y)
```

GridSearchCV ha come parametri: “estimator” a cui è assegnato il classificatore su cui fare il tuning; il numero di iperparametri ed i valori con il parametro “param_grid=params” che permette a GridSearchCV di testare tutti gli iperparametri descritti sopra; “cv=10” indica di usare la ten-fold cross validation ed in particolare se specificato un valore intero esegue una Stratified Cross-Validation, come specificato nella documentazione; ed infine “n_jobs=-1” indica di usare tutti i processori.

Creiamo e addestriamo un nuovo classificatore, avente i parametri del modello che ha dato i migliori risultati dopo la GridSearch.

```
kNN_best = kNN.set_params(**results.best_params_)
kNN_best.fit(train_x, train_y)
```

Infine effettuiamo delle predizioni sulla partizione di test.

```
pred_y = kNN_best.predict(test_x)
```

Stampiamo a video i risultati ottenuti grazie alla funzione metrics.

```
metrics(test_y,pred_y,np.unique(test_y))
```

Performance ottenute senza preprocessing

TPR	TNR	FPR	FNR	Precision	Recall	F-Score
0.94	**0.88**	**0.12**	**0.6**	**0.92**	**0.94**	**0.93**

6 Il terzo classificatore: Decision Tree

Abbiamo deciso di utilizzare il classificatore DecisionTreeClassifier come terzo classificatore a scelta fornito dalla libreria Scikit-Learn.

Il grande vantaggio che ci dà l’implementazione di un decision tree classifier, consiste nella sua interpretabilità: a differenza di tante tecniche di analisi dati, spesso basate su somme

ponderate di valori, il criterio con cui opera un albero decisionale è chiarissimo. In ogni nodo dell'albero c'è un criterio in base al quale si prosegue per uno dei suoi rami sottostanti, fino ad arrivare a una decisione.

```
X, y, dataType, features = csv_open()
train_x, test_x, train_y, test_y = train_test_split(X, y, random_state=0,
test_size=0.25, stratify=y)
```

Prima di tutto otteniamo i dati dalla funzione csv_open nel file csv_open e facciamo un campionamento stratificato per dividere i dati ottenuti in training e test set, rispettivamente 70% per il training set e 30% per il test set.

```
dTree = DecisionTreeClassifier()
```

Utilizziamo la funzione fornitaci da Scikit-Learn per inizializzare un oggetto dTree che sarà il nostro classificatore.

```
params_dTree = { #Definiamo un dizionario con tutti i parametri che varieranno
durante il tuning
        'criterion': ["entropy", "gini"], # le funzioni da usare per misurare la
qualità di uno split
        'max_depth': [None]+list(range(2,30,2)), # Indica la massima profondità
dell'albero, "None" indica che espande l'albero finchè tutte le foglie sono pure
oppure fino a quando tutte le foglie contengono meno di min_samples_split
campioni.
        'max_features': ["sqrt", "log2"], #indica il numero di attributi da
considerare quando si cerca il miglior split
        'min_samples_leaf': list(range(1,10,1)), # indica il numero minimo di
elementi richiesto per essere in un nodo foglia
        'min_samples_split': list(range(2,10,1)) # indica il numero minimo di
elementi richiesti per fare uno split in un nodo interno
    }
```

Dopo aver fatto ciò, definiamo un dizionario params_dTree, il quale conterrà i parametri da testare usando la funzione GridSearchCV, anch'essa fornita dalla libreria Scikit-Learn.

- -"criterion" indica quali misure utilizzare per valutare la qualità di uno split;
- "max_depth" indica la massima profondità dell'albero decisionale;
- "max_features" indica quante features consideriamo quando cerchiamo lo split migliore;
- "min_samples_leaf" indica il numero minimo di campioni richiesti per diventare un nodo foglia;
- "min_samples_split" indica il numero minimo di campioni richiesti per dividere un nodo interno.

Avremmo potuto prendere in considerazione anche altri parametri utilizzando la GridSearchCV, ma per comodità abbiamo considerato solo i parametri descritti sopra.

```
dTree_grid = GridSearchCV(estimator = dTree, param_grid = params_dTree, cv=10,
n_jobs=-1
```

GridSearchCV ha come parametri: "estimator" a cui è assegnato il classificatore su cui fare il tuning; il numero di iperparametri ed i valori con il parametro "param_grid=params" che permette a GridSearchCV di testare tutti gli iperparametri descritti sopra; "cv=10" indica di usare la ten-fold cross validation ed in particolare se specificato un valore intero esegue una

Stratified Cross-Validation, come specificato nella documentazione; ed infine “n_jobs=-1” indica di usare tutti i processori.

Eseguiamo l’addestramento del classificatore usando gli iperparametri che specificherà GridSearchCV.

```
results_dTree = dTree_grid.fit(train_x, train_y)
```

Eseguiamo l’addestramento del classificatore generato utilizzando la GridSearchCV.

```
dTree_best = dTree.set_params(**results_dTree.best_params_)
```

Assegnamo i parametri migliori ottenuti con il tuning del classificatore, al nuovo classificatore “dTree_best”, utilizzando l’attributo “best_params_” di GridSearchCV, il quale contiene un dizionario con i parametri che sono risultati migliori dopo il tuning.

```
dTree_best.fit(train_x, train_y)
```

Eseguiamo l’addestramento del nuovo classificatore “dTree_best”.

```
pred_y = dTree_best.predict(test_x)
```

Gli facciamo eseguire una predizione.

```
metrics(test_y, pred_y, np.unique(test_y))
```

Utilizzando la funzione metrics, calcoliamo le metriche che ci mostrano le performance del classificatore finale. La funzione metric utilizza una matrice di confusione per calcolare: Accuracy, TPR, TNR, FPR, FNR, Precision, Recall, F-Score.
Per questo specifico dataset consideriamo anche l’accuracy perché le classi non sono troppo sbilanciate.

Performance ottenute senza preprocessing

TPR	TNR	FPR	FNR	Precision	Recall	F-Score
0.94	**0.91**	**0.09**	**0.06**	**0.94**	**0.94**	**0.94**

7 Il classificatore semplice custom

Come classificatore semplice custom abbiamo scelto di usare una rivisitazione del Naive Bayes adattata ai tipi di attributo che abbiamo, quindi senza considerare la possibilità di avere feature di tipo diverso da quelli continui. Inoltre è possibile applicare diverse tecniche di preprocessing ai dati per vedere come cambiano le prestazioni.

Il classificatore implementa quattro metodi e un costruttore che dichiara le strutture dati necessarie. I quattro metodi sono: il fit che permette di fare l’addestramento, il predict che permette di fare una predizione e i due restanti permettono di calcolare la probabilità a priori di una classe e le probabilità a posteriori.

Il metodo di fit prende in input tre parametri: l’insieme di training, le relative etichette di classe ed i nomi degli attributi. Dopodiché all’interno di esso calcoliamo le probabilità a priori richiamando il metodo get_class_proba e dopodiché, per ogni attributo, calcoliamo la probabilità a posteriori richiamando il metodo get_att_proba che non fa altro che supporre che i dati si distribuiscano come una distribuzione di probabilità gaussiana, per poi

memorizzarli in un dizionario ed usarli in fase di predict. Quindi impostiamo a True il flag "fitted", che ci permette di non effettuare una predizione se prima non abbiamo eseguito la fase di addestramento.

```
def fit(self, train_x, train_y, a_names):
        self.a_names = a_names
        self.get_class_proba(train_y)  # calcolo le probabilità delle classi
        for col in range(0, np.shape(train_x)[1]):  # itero per il numero degli
attributi (57)
            self.get_att_proba(train_x[:, col], a_names[col], train_y)
            # print(train_x[:,col])
        self.fitted = True
def get_class_proba(self, y):
        n = len(y)
        y_vals, y_counts = np.unique(y, return_counts=True)  # unique restituisce
i possibili valor unici delle classi
        self.classes = y_vals  # e il conto dei diversi valori delle classi
        for i, val in enumerate(y_vals):  # calcolo delle probabilità per una
classe e per l'altra
            self.prior[val] = y_counts[i] / n # calcolo probabilità
# funzione che ha il compito di calcolare le probabilità di ogni attributo tra le
due possibili classi
    def get_att_proba(self, attribute, a_name, y):
        y_vals, y_counts = np.unique(y, return_counts=True)  # fa la stessa cosa
di prima, soltanto che i valori
        # cambiano da attributo ad attributo,
        # vengono analizzati tutti
        for i, y_val in enumerate(y_vals):
            y_val = str(y_val)
            # crea un dizionario formato da 114 chiavi 57 con classe0 e 57 con
classe 1
            # ogni chiave del dizionario è data dal nome dell'attributo più il
valore della classe corrispondente
            self.post[y_val + a_name] = {}

            # per ogni chiave calcolo anche
            a = attribute[y == int(y_val)].astype(float)  # i valori assunti
dall'attributo
            mu = np.mean(a)  # la media
            var = np.var(a)+0.00000000001 # la varianza, aggiungo un valore
minuscolo perchè usando le tecniche di undersampling la varianza diventa uguale a
0 e mi da problemi. Con tutte le altre tecniche di preprocessing non succede.
# in ogni caso dovrebbe essere un valore ininfluente sul risultato finale

            # inserisco nel dizionario sia la media che la varianza per ogni cella
presente
            self.post[y_val + a_name]["mu"] = mu
```

```
            self.post[y_val + a_name]["var"] = var
```

Il metodo predict prende in input due parametri: l'insieme di test e le etichette corrispondenti. Può essere eseguito solo se prima è stato fatto l'addestramento, quindi se fitted=True. Per ogni riga dell'insieme di test, assegnamo a "prod" la probabilità a priori e andiamo a prendere dal dizionario creato in fase di addestramento, le probabilità a posteriori per fare la predizione. Quindi prendiamo la media e la varianza calcolata prima e applichiamo la formula per calcolare la probabilità condizionata che appartenga a una classe.
Alla fine restituiamo la classe per cui la probabilità è maggiore.

```
def predict(self, test_x, test_y):
        if self.fitted: # posso fare una predizione solo se prima è stato
addestrato il modello
            # per ogni record del test set
            pred_y = []
            for row in test_x:  # itero per tutte le righe preseni nel test_x
                i_max = 0
                prob_max = 0
                for i, y_val in enumerate(self.classes):
                    print(i,y_val)
                    prod = self.prior[y_val]  # inizialmente la probabilità è
quella a priori per 0 o per 1
                    y_val = str(y_val)
                    for j, a_val in enumerate(row):
                        # si prendono sia media che varianza calcolate
precedentemente per ogni cella del dizionario
                        mu = self.post[y_val + self.a_names[j]]["mu"]
                        var = self.post[y_val + self.a_names[j]]["var"]
                        denominator = np.sqrt(2 * np.pi * var)
                        numerator = np.exp(-np.power(a_val - mu, 2) / (2 * var))
                        #print(numerator,denominator)
                        prod = (prod * numerator) / denominator
                    if prod > prob_max:
                        prob_max = prod
                        i_max = i
                    # vengono calcolate le probabilità per tutti i possibili
valori della classe
                    # e poi ci si salva solo la classe che aveva la probabilità
più alta

                pred_y = pred_y + [self.classes[i_max]]
            return pred_y
        else:
            print("Il classificatore non è ancora stato addestrato")
```

Performance ottenute senza preprocessing

TPR	TNR	FPR	FNR	Precision	Recall	F-Score
0.76	**0.96**	**0.4**	**0.24**	**0.97**	**0.76**	**0.85**

8 Il classificatore multiplo custom

Come classificatore multiplo custom da utilizzare abbiamo scelto un classificatore di ensemble basato sul majority voting. Come base del classificatore, abbiamo utilizzato il codice proveniente dalla soluzione al terzo esercizio del decimo tra i notebooks visti durante le lezioni di laboratorio.
Il classificatore è strutturato tramite una classe Python contenente:

- Un metodo di addestramento,"fit";
- Un metodo di predizione, "predict".

In fase di inizializzazione l'ensemble accetta una lista semplice che contiene l'insieme dei classificatori di base, la modalità di voting ("hard", "soft") e i pesi da associare alle predizioni fatte da ognuno dei classificatori di base. Nel caso non si vogliano considerare i pesi durante la fase di voting basta mettere al posto della lista dei pesi "None", questo nella fase di inizializzazione dell'ensemble verrà gestito in modo da dare lo stesso peso alle previsioni di tutti i classificatori semplici.

Il metodo di fit, prende come input oltre al training set anche le labels (y), inoltre per fare l'addestramento di ogni classificatore semplice utilizza una porzione diversa del training set.

```
For estimator in self.estimators:
      sub_train_x, _, sub_train_y, _ = train_test_split(x, y, test_size=0.20,
stratify=y, random_state = 42)
      estimator.fit(sub_train_x, sub_train_y)
```

Il metodo di predict, prende come input solo il test set ed ha il compito di restituire una predizione che tiene in considerazione delle predizioni fatte da tutti i classificatori. Il metodo inizialmente controlla che fitted sia uguale a True. Se la condizione è verificata si calcolano come prima cosa le predizioni di ogni classificatore sul test set.

```
For estimator in self.estimators: #si calcolano le probabilità di predizione per
ogni classificatore
                proba.append(estimator.predict_proba(test_x))
                predictions.append(estimator.predict(test_x))
proba = np.array(proba) #vengono trasformate in np.array per poter svolgere delle
operazioni numpy
predictions = np.array(predictions).T
```

Quella che prima era una lista semplice viene in seguito trasformata in un NumPy Array, così da utilizzare in seguito operazioni NumPy su di esso.
In seguito per ogni riga del test set viene fatta un'operazione di aggregazione delle predizioni dei classificatori. Questa operazione è fatta tramite la funzione "aggregate function" che prende in input le probabilità di ogni classificatore per quella riga, il tipo di voting, i pesi delle predizioni e le predizioni per record del test set di ogni classificatore. Se come tipologia

di voting si seleziona “hard” verrà restituita la classe votata dal maggior numero di classificatori. Nel caso di un “soft voting” si sommano per ogni label le probabilità date da ogni classificatore, e verrà restituita la classe con la somma (pesata o non) maggiore.

```
def aggregate_function(probability, voting_mode, weights, predictions):

    maggiore = []
    count_0 = 0
    count_1 = 0

    if voting_mode == 1: #hard voting
        for i in range(len(predictions)):
            if predictions[i] == 0:
                count_0 = count_0 + (1 * weights[i])
            else:
                count_1 = count_1 + (1 * weights[i])

        if count_0 > count_1:
            return 0
        else:
            return 1

    if voting_mode == 2: #soft voting
        maggiore.append(sum(probability[0][:]*weights))
        maggiore.append(sum(probability[1][:]*weights))
        #verrà restituito 0 se la classe 0 avrà avuto la probabilità più alta, in caso contrario verrà restituita la classe 1
        if (maggiore[0] > maggiore[1]):
            return 0
        else:
            return 1
```

In questo caso i classificatori che abbiamo utilizzato sono il kNN, il dTree e il Gaussian Naive Bayes, tutti presi dalla libreria scikit-learn. Per ognuno di questi è stato svolto il tuning degli iperparametri, attraverso la funzione “GridSearchCV”.

Infine calcoliamo delle metriche, che ci mostrano le performance del classificatore finale usando la funzione metrics che calcola al suo interno una matrice di confusione che sfrutta per calcolare Accuracy, TPR, TNR, FPR, FNR, Precision, Recall, F-Score. Consideriamo anche l’accuracy perché le classi non sono particolarmente sbilanciate.

```
Print(“#### PERFORMANCES Ensemble:####\n”)
metrics(test_y,pred_y,np.unique(test_y)
```

Performance ottenute senza preprocessing, usando hard voting senza pesi

TPR	TNR	FPR	FNR	Precision	Recall	F-Score
0.92	0.92	0.08	0.08	0.95	0.92	0.93

9 I classificatori a confronto con o senza standardizzazione dei dati

Abbiamo deciso di confrontare le performance dei classificatori con tre diverse trasformazioni dei dati: MinMax Scaler, che standardizza i dati per fare in modo che siano in un range ben definito tra un massimo ed un minimo, lo Standard Scaler, anche chiamato ZScore che fa in modo che i dati, per ogni attributo, abbiano media pari a zero e varianza pari ad uno ed infine la Normalizzazione che permette di trasformare i dati facendo in modo che abbiano norma unitaria.
Il codice riguardante la trasformazione dei dati, si trova nel file “preprocessing.py” che viene richiamato dalla funzione main() in base all’input dell’utente.

L’accuracy dei 5 classificatori, senza considerare altre operazioni di preprocessing sono :

	Random Forest	k-Nearest Neighbor	Decision Tree	Naive Bayes Custom	Ensemble Custom
MinMax Scaler	0.92	0.89	0.89	0.84	0.93
Standard Scaler	0.96	0.93	0.89	0.84	0.93
Normalizzazione	0.95	0.87	0.91	0.79	0.90
Senza nessuna Trasformazione	0.96	0.85	0.92	0.84	0.91

10 I classificatori a confronto con o senza l’uso di aggregazione e selezione di features

Per quanto riguarda la feature selection, abbiamo deciso di calcolare una matrice di correlazione e cercare gli attributi che hanno tra loro una correlazione maggiore di un certo valore soglia, da noi deciso arbitrariamente come 0.8. Avremmo potuto mettere qualsiasi altro valore soglia, l’importante è che sia un valore molto alto maggiore di 0 e minore o uguale di 1 in quanto un valore in questo range ci permette di affermare che due attributi variano nello stesso modo (all’aumentare dell’uno, aumenta l’altro e al diminuire di uno diminuisce anche l’altro) più è alto il coefficiente che viene restituito, maggiore sarà la correlazione.

```
featureCorr = np.where(corr_X > 0.80)
```

“featureCorr” ci restituisce due matrici che indicano i rispettivi indici per cui i corrispondenti attributi hanno una correlazione maggiore di 0.8.

Estraiamo gli indici che sono diversi tra loro (ossia scartiamo gli indici per cui la correlazione con lo stesso attributo è, ovviamente, uguale ad 1) considerando solo uno dei due indici, in questo modo stiamo estraendo solo uno dei due indici corrispondenti a due attributi che hanno una correlazione >0.8.

```
indexes = [featureCorr[0][i] for i in range(len(featureCorr[0])) if featureCorr[0][i] != featureCorr[1][i]]
```

Eliminiamo eventuali duplicati

```
indexes = np.unique(indexes)
```

Estraiamo i nomi delle features, sulla base dell’indice, perché ci serviranno per eliminare quegli attributi dal dataset, lasciando gli solo uno dei due attributi che hanno tra loro correlazione >0.8

```
indexesNames = [features[i] for i in indexes]
```

Riassegnamo al dataset originale il dataset con gli attributi rimossi. “columns” specifica le colonne da considerare, inplace=”false” indica di restituire una nuova copia del dataset senza quelle colonne e “axis=1” indica di considerare le colonne.

```
X = df.drop(columns=indexesNames, inplace=False, axis=1)
```

Infine, trasformiamo il dataset modificato in una matrice numpy.

```
X = X.to_numpy()
```

Inoltre abbiamo deciso di utilizzare delle funzioni per la selezione di features che fanno parte della libreria di Sklearn.

- VarianceThreshold: si basa sull'analisi della varianza, non consente di selezionare un numero prefissato di features, ma usa un valore di soglia "Threshold" per eliminare tutte le features con una varianza inferiore. Di default elimina tutte le features che hanno una varianza uguale a 0, ovvero quelle il cui valore non cambia per tutti i record. Come visto a lezione abbiamo deciso di utilizzare Threshold = 1;
- SelectKBest: fa lo scoring di tutte le features e permette di selezionare le prime k. Prende in input oltre il numero predefinito di features da selezionare anche una funzione che restituisce un singolo array con gli scores. La funzione utilizzata di default è la "f_classif", ma in alternativa abbiamo deciso di utilizzare la "chi2" e la "mutual_info_classif". Tutte queste funzioni di scoring vanno bene solo per problemi di classificazione, se il problema fosse stato di regressione avremmo dovuto utilizzare altre funzioni come la "f_regression" o la "r_regression". In entrambi i casi, "chi2" e "mutual_info_classif", il numero degli attributi è stato diviso della metà passando da 57 a 28;
- SequentialFeaturesSelector: Aggiunge (forward selection) o rimuove (backward elimination) features da un sottoinsieme di features usando un approccio greedy. Ad ogni istanza, l'estimator (classificatore) sceglie quale feature sia la migliore da aggiungere o rimuovere in base al cross validation score fatto da quello stesso classificatore. Come visto a lezione abbiamo deciso di utilizzare come estimator un KNeighborsClassifier con n_neighbors = 1. Il numero delle features da selezionare è come per gli approcci precedenti 28, ovvero la metà del numero originale di features presenti. Un altro parametro che avremmo potuto modificare anziché lasciare le scelte default, è la direzione della selezione degli attributi (forward / backward), questa funzione usa la forward selection di default.

	Random Forest	k-Nearest Neighbor	Decision Tree	Naive Bayes Custom	Ensemble Custom
FeatureCorrelation	1.0	0.89	1.0	0.99	0.99
VarianceThreshold	0.91	0.81	0.88	0.74	0.87
SelectKBest (chi2)	0.95	0.85	0.94	0.91	0.93
SelectKBest (mutual_info_classif)	0.95	0.83	0.93	0.90	0.9t
SequentialFeatureSelector	0.95	0.91	0.92	0.81	0.93
Senza selezione delle features	0.96	0.85	0.92	0.84	0.91

Per quanto riguarda la Feature Aggregation, abbiamo deciso di sfruttare i metodi visti a lezione come SparseRandomProjection, GaussianRandomProjection, FeatureAgglomeration e PCA permettendo di ridurre per ognuno il numero di attributi a 28, ossia la metà di quelli originali.

- GaussianRandomProjection permette di ridurre la dimensionalità del dataset specificando la nuova dimensionalità con il parametro "n_components" ottenendo una matrice con distribuzione gaussiana densa;
- SparseRandomProjection permette di ridurre la dimensionalità del dataset specificando la nuova dimensionalità con il parametro "n_components", garantisce una qualità di agglomerazione simile alla Gaussian Random Projection ma con una maggiore efficienza in termini di memoria e consentendo un calcolo più rapido.

In generale, sia nella SparseRandomProjection che nella GaussianRandomProjection quello che facciamo è proiettare i dati in uno spazio di dimensionalità inferiore preservando il più possibile le distanze tra i punti, sono due tecniche che usiamo al posto della PCA perché meno costose computazionalmente.

Per quanto riguarda la Principal Component Analysis, abbiamo una riduzione lineare della dimensionalità usando la Singular Value Decomposition dei dati per proiettarli in uno spazio dimensionale inferiore. Il parametro in input "n_components" indica il numero di componenti principali da tenere, se non è specificato allora vengono tenute tutte le componenti principali generate.

Nella tabella sotto è possibile vedere l'accuracy dei classificatori usando solamente le tecniche descritte sopra:

	Random Forest	k-Nearest Neighbor	Decision Tree	Naive Bayes Custom	Ensemble Custom
SparseRandomProjection	0.92	0.84	0.87	0.72	0.88
GaussianRandomProjection	0.85	0.84	0.81	0.66	0.81
Feature Agglomeration	0.94	0.84	0.91	0.82	0.91
PCA	0.93	0.88	0.88	0.81	0.90
Senza tecniche di Feature Aggregation	0.96	0.85	0.92	0.84	0.91

Notiamo che la Feature Agglomeration permette di ottenere performance migliori, rispetto agli altri metodi, con tutti i classificatori che abbiamo provato.

11 I classificatori a confronto con o senza l'uso di tecniche di bilanciamento

Per quanto riguarda la fase di bilanciamento abbiamo deciso di usare delle classi viste durante le lezioni di laboratorio che fanno parte della libreria imbalanced-learn. In tutti e 3 i metodi tra i parametri è stato cambiato solo il "random state", settato a 42 in modo tale da avere la stessa partizione del data set in diverse esecuzioni dell'algoritmo.
Dunque abbiamo scelto di utilizzare:

- RandomOversampling: è la strategia più naive e consiste nel generare nuovi record selezionando randomicamente, con rimpiazzo, record già presenti all'interno del data set.
- SMOTE: Il suo funzionamento si basa sul generare nuove osservazioni sintetiche della classe minoritaria, utilizzando la tecnica di interpolazione tra i punti di dati esistenti. Crea nuovi record che sono combinazioni lineari di coppie di record esistenti nella classe minoritaria.
- ADASYN: utilizza un procedimento simile allo SMOTE ma tiene in considerazione anche la densità dei dati della classe minoritaria. L'algoritmo tende a generare più osservazioni sintetiche in zone del dataset in cui la classe minoritaria è meno densa. In questo modo cerca di generare una distribuzione più elevata della classe minoritaria nel dataset.

Accuracy dei classificatori usando solamente tecniche di Oversampling:

	Random Forest	k-Nearest Neighbor	Decision Tree	Naive Bayes Custom	Ensemble Custom

RandomOverSampling	0.97	0.89	0.94	0.85	0.94
SMOTE	0.96	0.89	0.92	0.86	0.93
ADASYN	0.96	0.89	0.92	0.86	0.93
Senza nessuna tecnica di Oversampling	0.96	0.85	0.92	0.84	0.91

Si può notare che abbiamo quasi le stesse performance, per ogni classificatore, indipendentemente dal metodo che usiamo.

Per quanto riguarda le tecniche di UnderSampling abbiamo deciso di sfruttare i metodi visti a lezione come RandomUnderSampler, InstanceHardnessThreshold, NearMissV1, NearMissV2 e ClusterCentroids, anch'essi fanno parte della libreria "imbalanced-learn".

- RandomUnderSampler ci permette di eseguire un undersampling della classe maggioritaria selezionando gli elementi in maniera randomica con o senza rimpiazzamento.
- ClusterCentroids ci permette di fare undersampling generando centroidi usando metodi di clustering.
- InstanceHardnessThreshold usa criteri probabilistici che preservano quelli con più alta probabilità;
- NearMissV1 usa metodi basati su distanze che preservano i più vicini ai più vicini ed infine
- NearMissV2 usa criteri basati su distanze che preservano i più vicini ai più lontani.

Accuracy dei classificatory usando tecniche di Undersampling:

	Random Forest	k-Nearest Neighbor	Decision Tree	Naive Bayes Custom	Ensemble Custom
RandomUnderSampler	0.93	0.86	0.90	0.84	0.92
InstanceHardnessThreshold	0.98	0.93	0.95	0.88	0.95
NearMissV1	0.95	0.85	0.90	0.82	0.92
NearMissV2	0.94	0.81	0.90	0.85	0.91
ClusterCentroids	0.94	0.83	0.89	0.87	0.90
Senza nessuna tecnica di undersampling	0.96	0.85	0.92	0.84	0.91

Possiamo notare come la tecnica di undersampling che ci permette di ottenere prestazioni migliori sia InstanceHardnessThreshold.

N.B. ClusterCentroids quando usato mostra un messaggio riguardo un parametro "n_init" che non riguarda il metodo stesso; non è un messaggio di errore e "n_init" non è un parametro che è possibile specificare, per questo, potrebbe essere un parametro da specificare che dipende dal modo con cui è stato implementato il metodo stesso.

Per quanto riguarda la combinazione di tecniche di UnderSampling e OverSampling, abbiamo deciso di vedere le prestazioni dei classificatori sottocampionando la classe maggioritaria del dataset originale (classe 0, non spam, 2788 istanze) portandola a 2300 istanze e sovracampionando successivamente la classe minoritaria nel dataset originale (classe 1, spam, 1813 oggetti) portandola a 2301 istanze.

Per fare questo abbiamo creato per prima cosa una copia del dataset originale:

```
tmp=X.copy()
```

dopodiché abbiamo fatto undersampling specificando con il parametro "sampling_strategy" il nuovo numero di record della classe 0, la classe che stiamo sottocampionando.

```
#undersampling
iht = InstanceHardnessThreshold(sampling_strategy={
            0: 2300 # facciamo undersampling della classe maggioritaria 0
portandola a 2300 record.
        },random_state=42)
X_resampled, y_resampled = iht.fit_resample(tmp, y)
```

Dopodichè il nuovo dataset sottocampionato lo diamo in input al metodo SMOTE che ci permette di fare oversampling, specificando il nuovo numero di record che vogliamo della classe 1, specificando come prima il parametro "sampling_strategy".

```
#oversampling
sm = SMOTE(sampling_strategy={
            1: 2301 # facciamo oversampling della classe minoritaria 1, portandola
a 2301 record
        },random_state=42)
X_resampled, y_resampled = sm.fit_resample(X_resampled, y_resampled)
```

L'accuracy dei classificatori usando una combinazione di undersampling e oversampling è :

	Random Forest	k-Nearest Neighbor	Decision Tree	Naive Bayes Custom	Ensemble Custom
Combinazione di undersampling e oversampling	0.98	0.92	0.96	0.86	0.95
Senza nessuna combinazione di Undersampling e Oversampling	0.96	0.85	0.92	0.84	0.91

12 Confronto dei classificatori con e senza tecniche di preprocessing:

- Nella prima curva ROC sono rappresentate le performance dei classificatori senza nessuna tecnica di preprocessing. Notiamo che il classificatore multiplo di Ensemble, combinando le votazioni di altri classificatori semplici, raggiunge le performance migliori;
- Nella seconda curva ROC abbiamo deciso di mettere insieme tutte le tecniche di preprocessing migliori per ogni classificatore studiato. Per il RandomForest, dTree, myNBayes e myEnsemble abbiamo deciso di usare la feature correlation, che fa parte della sezione sulla selezione di features. Invece per il kNN abbiamo deciso di utilizzare la tecnica di UnderSampling InstanceHardnessThreshold, che ci ha permesso di ottenere un'area sotto la curva di 0.91.

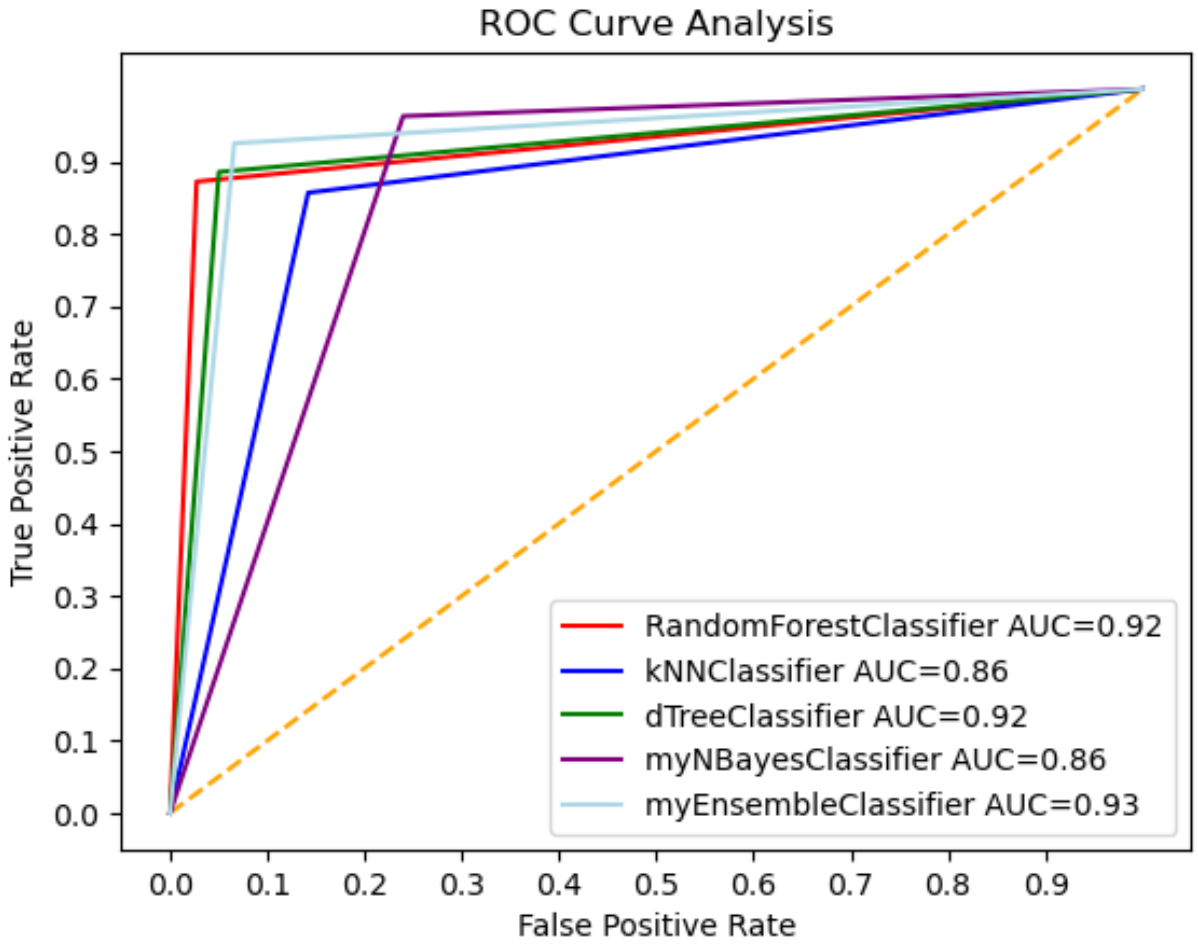

Figure 1 Curva roc senza applicare tecniche di preprocessing ai dati

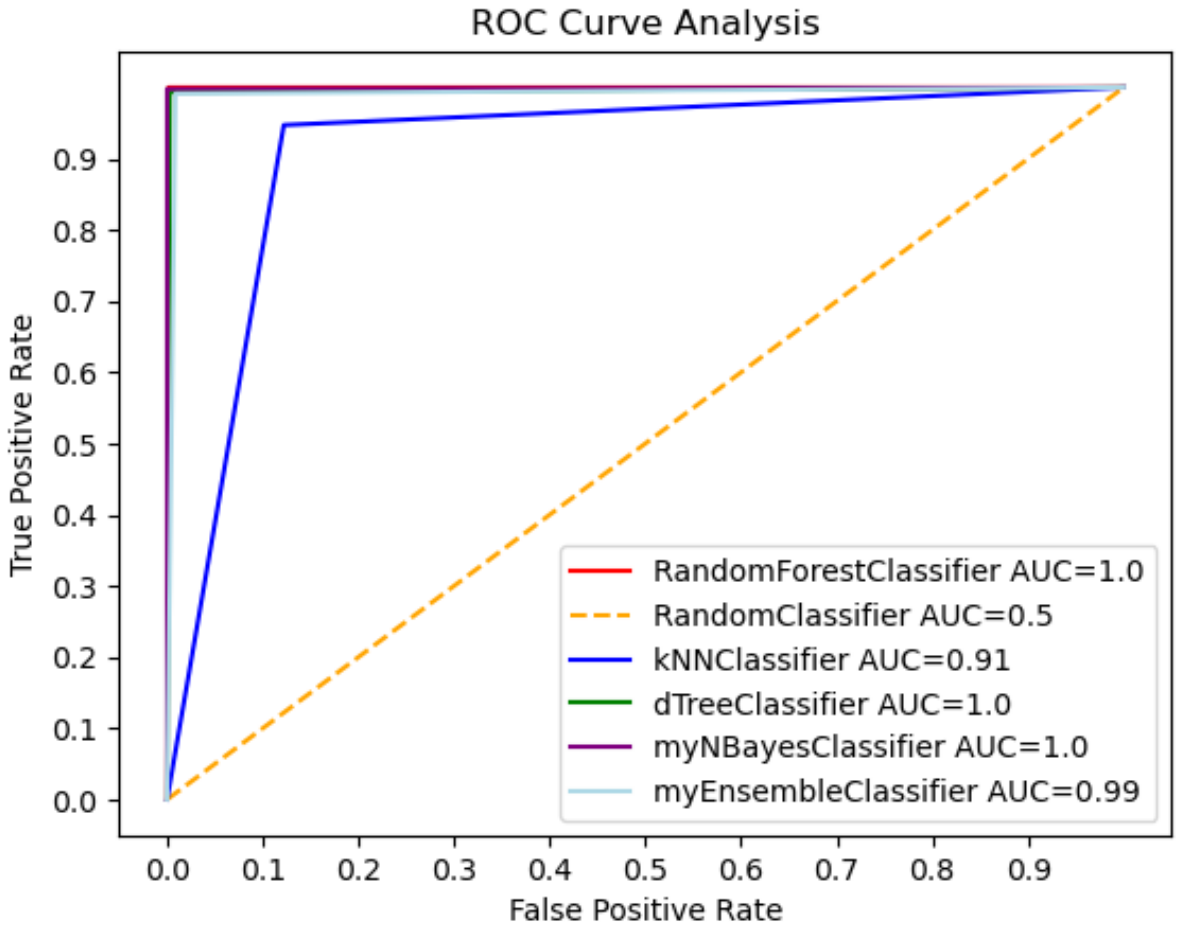

Figure 2 Curva ROC ottenuta dopo aver applicato le tecniche di preprocessing che ci danno le performance migliori.