



Dipartimento di Informatica
Corso di Laurea Triennale in Informatica
Applicata e Data Analytics

Relazione Progetto Machine Learning

Professore:
Lorenzo Putzu

Studenti:
Senette Davide
Aresu Andrea
Floris Theo
Lodo Edoardo
Sanna Massimo

Anno Accademico 2023/2024

Contents

1	Introduzione	2
1.1	Librerie	2
1.2	L'applicazione	4
1.2.1	Dati da Analizzare	4
1.2.2	Addestra Modelli	4
1.2.3	Miglior combinazione	5
2	Analisi dei Dati	6
3	Classificatori	10
3.1	K-Nearest Neighbor	10
3.1.1	Tuning KNN	10
3.1.2	Descrizione codice	13
3.2	Albero Decisionale	15
3.2.1	Tuning Albero Decisionale	15
3.2.2	Descrizione codice	17
3.3	Random Forest	18
3.3.1	Tuning Random Forest	18
3.3.2	Descrizione codice	20
3.4	Support Vector Machine	22
3.4.1	Tuning SVM	22
3.4.2	Descrizione codice	25
3.5	Artificial Neural Network	27
3.5.1	Tuning ANN	27
3.5.2	Descrizione codice	30
4	Tecniche di Pre-Processing	32
4.1	Tecniche di pre-processing utilizzate	32
4.2	Campionamento	32
4.3	Bilanciamento	33
4.4	Feature Selection	35
4.5	Combinazione di Attributi	37
4.6	Trasformazione di Attributi	40
5	Conclusioni	42

1 Introduzione

Nella seguente relazione andremo ad analizzare il data set 'Breast Cancer Wisconsin (Diagnostic) Data Set'. Lo scopo di questo progetto è di utilizzare diversi tipi di modelli ad apprendimento automatico di tipo predittivo, confrontarli tra di loro e utilizzare quello con un accuratezza maggiore per la diagnosi del cancro, così da individuare in quali casi esso risulti benigno ed in quali maligno.

Il data set selezionato contiene questi attributi:

- Id number
- Diagnosi (M = maligno, B = benigno)

Per ogni nucleo cellulare vengono calcolate dieci caratteristiche a valore reale:

- Raggio (Media delle distanze dal centro ai punti sul perimetro)
- Texture (deviazione standard dei valori in scala di grigi).
- Perimetro
- Area
- Smoothness (variazione locale nelle lunghezze dei raggi)
- Compattezza ($\frac{\text{perimetro}^2}{\text{area}-1.0}$)
- Concavità (gravità delle porzioni concave del contorno)
- Punti concavi (numero di porzioni concave del contorno)
- Simmetria
- Dimensione frattale ("approssimazione della linea costiera" - 1)

La media, lo standard error e il "peggiore" o il più grande (media dei tre valori più grandi) di queste caratteristiche sono stati calcolati per ciascuna immagine, risultando 30 attributi.

1.1 Librerie

Per la realizzazione del progetto, abbiamo utilizzato Python versione 3.12.1 insieme alle seguenti librerie:

Libreria	Versione
Numpy	1.26.3
Pandas	2.1.4
Matplotlib	3.8.2
Scikit-learn	1.3.2
Tk	8.6
Pillow	10.2.0
Seaborn	0.13.1
Imbalanced-learn	0.11.0
Scipy	1.11.4

Numpy e Pandas: La libreria np è stata usata per il calcolo della varianza e nel calcolo dei k Nearest Neighbour nel classificatore semplice custom. Pandas viene usata per l'estrapolazione dei dati dal file csv, e durante la creazione di alcune tecniche di pre processing.

Matplotlib e Seaborn: Matplotlib e Seaborn sono state scelte come librerie per la fase di data visualization: Matplotlib viene usata per la creazione dei box plot, della matrice di correlazione e per i plot dei modelli con e senza tuning. Seaborn per migliorare l'estetica dei box plot, della matrice di correlazione.

Sci-kit Learn: Sci-kit Learn è stata la libreria scelta per implementare i seguenti modelli di apprendimento, grazie alla sua affidabilità, flessibilità e vasta gamma di funzionalità. Di seguito, sono elencati i modelli di apprendimento che abbiamo sviluppato utilizzando le classi e le funzioni fornite da Sci-kit Learn:

- Decision Tree
- Support Vector Machine
- Artificial Neural Network

Tkinter (tk): Tkinter è stato utilizzato per la creazione dell'interfaccia grafica del nostro sistema. La sua semplicità e integrazione con Python lo rendono uno strumento potente per lo sviluppo di GUI.

Pillow: Pillow è un fork di PIL (Python Imaging Library) e ci ha fornito un insieme di strumenti essenziali per il lavoro con immagini. La sua compatibilità e facilità d'uso lo rendono uno strumento ideale per la gestione delle operazioni di elaborazione delle immagini nel nostro progetto.

Imbalanced-Learn: Abbiamo incorporato Imbalanced-Learn per affrontare problemi di sbilanciamento delle classi nei nostri dati di addestramento. Questa libreria fornisce tecniche di campionamento e bilanciamento che sono cruciali per migliorare le prestazioni dei modelli su dati con distribuzioni sproporzionate delle classi.

Scipy: è stata utilizzata per garantire la compatibilità con Imbalanced-Learn, poiché alcune funzionalità di Imbalanced-Learn dipendono da moduli specifici di SciPy per il loro corretto funzionamento.

1.2 L'applicazione

Abbiamo progettato un'applicazione GUI per facilitare la visualizzazione e l'analisi dei dati nel contesto del nostro progetto. Il menu principale offre diverse opzioni per esplorare e analizzare i dati in modo user-friendly.

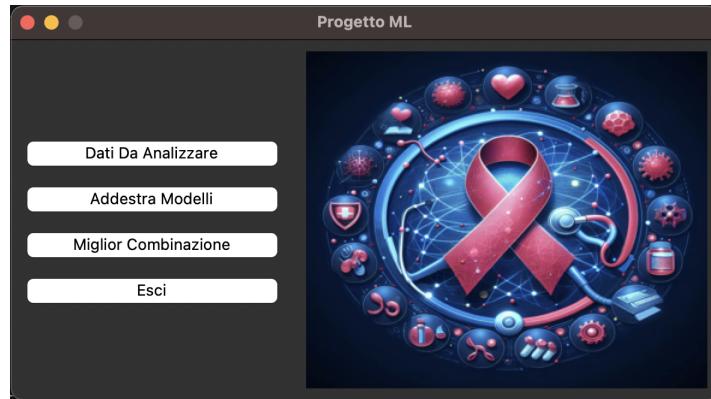


Figure 1: Homepage dell'app

Ecco una descrizione dettagliata delle funzionalità disponibili:

1.2.1 Dati da Analizzare

- **Informazioni dataset:** Visualizza tutte le etichette delle colonne del data set, il tail e l'head e fornisce il numero totale di righe.
- **Box Plot:** Permette all'utente di selezionare un attributo e visualizzarne il box plot per una rapida comprensione della distribuzione dei dati, e la presenza di outliers
- **Matrice di correlazione:** Mostra la matrice di correlazione tra le varie colonne del data set
- **Statistiche descrittive:** Permette di ottenere le statistiche descrittive come il massimo, il minimo, la media, la varianza e i quartili per ogni attributo del data set.

1.2.2 Addestra Modelli

- L'utente può selezionare il tipo di pre-processing desiderato e il modello di machine learning da utilizzare

- Dopo la selezione, l'applicazione esegue l'addestramento del modello e visualizza le performance ottenute

1.2.3 Miglior combinazione

- Questa opzione esegue un'esplorazione approfondita in tutto e per tutto dell'applicazione. Infatti entrando in questo lato del menù avremo due possibilità:
 1. La prima, dove leggiamo *"modelli"*, in questa sotto-sezione del menù, potremo verificare per ogni singolo modello, la miglior combinazione *"Modello - Pre-Processing"*.
 2. Nella seconda parte del sotto-menù troviamo, invece, la scritta *"Confronto tra modelli"*. Questa sotto-sezione si occupa invece, di effettuare il confronto tra tutti i modelli con la migliore tecnica di Pre-Processing, o per ogni singola tecnica di Pre-Processing, mostrando il risultato dentro un plot, contenente le curve ROC di tutti i modelli.

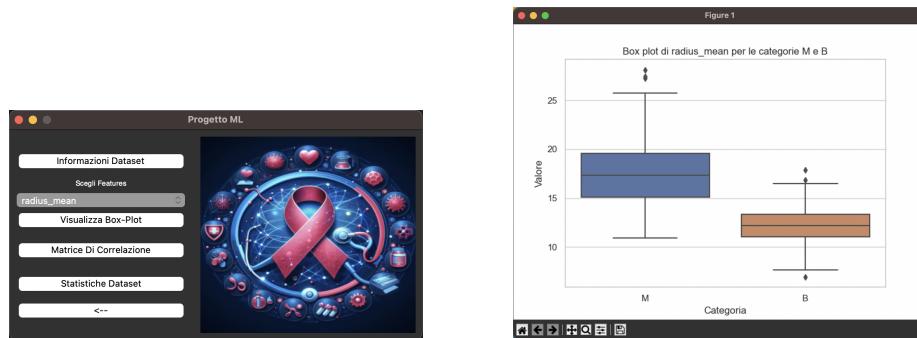


Figure 2: Funzionalità dell'applicazione

Con questa struttura, l'applicazione offre una vasta gamma di strumenti per analizzare e modellare i dati, permettendo agli utenti di esplorare approfonditamente la peculiarità del data set e valutare diverse configurazioni di modelli di machine learning ottimizzati per questo specifico data set.

2 Analisi dei Dati

Il data set è di tipo record oriented e contiene:

- 32 colonne
- 570 righe
- 2 classi
 - 357 record benigni
 - 212 record maligni

Tutti i valori delle feature sono registrati con quattro cifre significative. Avendo le classi e dovendo catalogare nuovi oggetti in base ad essa, ci troviamo davanti ad un problema di classificazione.

Per quanto riguarda l'analisi dei dati e la loro qualità, direttamente dal menù dell'applicazione è stata inserita la voce "dati da analizzare" che permette di visualizzare le informazioni contenute nel dataset e tramite la selezione di una determinata features anche di generare il box-plot riguardante ad essa, implementata tramite la libreria pandas e tramite la libreria seaborn per la rappresentazione dei box-plot.

Questa è una panoramica del dataset che mostra i nomi degli attributi usando la libreria pandas e i dati delle prime e ultime righe del dataset utilizzando in particolare la funzione `.head()` e `.tail()`:

Informazioni generali:		
#	Column	Non-Null Count Dtype
0	id	569 non-null int64
1	diagnosis	569 non-null object
2	radius_mean	569 non-null float64
3	texture_mean	569 non-null float64
4	perimeter_mean	569 non-null float64
5	area_mean	569 non-null float64
6	smoothness_mean	569 non-null float64
7	compactness_mean	569 non-null float64
8	concavity_mean	569 non-null float64
9	concave points_mean	569 non-null float64
10	symmetry_mean	569 non-null float64
11	fractal dimension_mean	569 non-null float64
12	radius_se	569 non-null float64
13	texture_se	569 non-null float64
14	perimeter_se	569 non-null float64
15	area_se	569 non-null float64
16	smoothness_se	569 non-null float64
17	compactness_se	569 non-null float64
18	concavity_se	569 non-null float64
19	concave points_se	569 non-null float64
20	symmetry_se	569 non-null float64
21	fractal dimension_se	569 non-null float64
22	radius_worst	569 non-null float64
23	texture_worst	569 non-null float64
24	perimeter_worst	569 non-null float64
25	area_worst	569 non-null float64
26	smoothness_worst	569 non-null float64
27	compactness_worst	569 non-null float64
28	concavity_worst	569 non-null float64
29	concave points_worst	569 non-null float64
30	symmetry_worst	569 non-null float64
31	fractal dimension_worst	569 non-null float64
32	Unnamed: 32	0 non-null float64

Figure 3: Informazioni generali sul data set

```

Prime righe del DataFrame:
   id diagnosis radius_mean ... symmetry_worst fractal_dimension_worst  Unnamed: 32
0    842302      M    17.98 ...          0.4601           0.11890        NaM
1    841930      M    20.57 ...          0.4596           0.10992        NaM
2    84100903     M    19.69 ...          0.3613           0.08758        NaM
3    8414801      M    11.42 ...          0.6398           0.17300        NaM
4    84158602      M    20.29 ...          0.2364           0.07678        NaM
[5 rows x 33 columns]

Ultime righe del DataFrame:
   id diagnosis radius_mean ... symmetry_worst fractal_dimension_worst  Unnamed: 32
564 924460      M    21.56 ...          0.2360           0.07115        NaM
565 9244682     M    20.13 ...          0.2372           0.06637        NaM
566 9246954     M    16.60 ...          0.2218           0.07825        NaM
567 9273741     M    20.60 ...          0.2307           0.13400        NaM
568 92751       B    17.9 ...          0.2871           0.07099        NaM
[5 rows x 33 columns]

```

Figure 4: Prime e ultime righe del data set

Di seguito la parte di codice che implementa la generazione dei box-plot

```

def BoxPlot(Feature):
    X, y = import_dataset()

    # Utilizza seaborn per un aspetto più attraente
    sns.set(style="whitegrid")

    # Crea il box plot per l'attributo in base alle etichette
    plt.figure(figsize=(7, 5))
    sns.boxplot(x=y, y=X[Feature])

    # Aggiungi etichette e titolo
    plt.xlabel('Categoria')
    plt.ylabel('Valore')
    plt.title(f'Box plot di {Feature} per le categorie M e B')
    # Mostra il grafico
    plt.show()

```

Figure 5: Codice implementazione box-plot

Vediamo in seguito la rappresentazione di alcuni box-plot significativi:

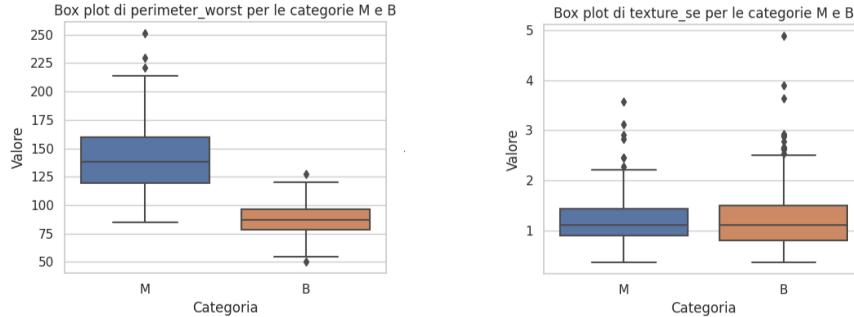


Figure 6

L'analisi dei boxplot evidenzia la presenza di outliers rappresentati dai punti situati al di fuori dei whisker. All'interno dei due rettangoli, notiamo una linea che indica la mediana delle due classi associate all'attributo texture-se. La posizione centrale della linea di mediana nella classe B (benigni) suggerisce una distribuzione equa dei dati, con valori distribuiti uniformemente da entrambi i

lati. Al contrario, nella classe M (maligni), osserviamo che i valori della distribuzione tendono a concentrarsi oltre la mediana, indicando una prevalenza di valori superiori.

Nella sezione "Statistiche dataset" accessibile dal menù, vengono mostrate tutte le statistiche riguardanti tutti i vari attributi e, dimostra l'assenza di valori mancanti e duplicati.

Inoltre, sono presenti i valori medi per ogni attributo e la deviazione standard che è una misura di dispersione o variabilità all'interno di un set di dati. In particolare, la deviazione standard di un attributo in un dataset quantifica quanto i valori di quell'attributo si discostano, in media, dal valore medio dell'attributo.

Valori medi:	
radius_mean	14.127292
texture_mean	19.289649
perimeter_mean	91.969033
area_mean	654.889104
smoothness_mean	0.096360
compactness_mean	0.104341
concavity_mean	0.088799

Figure 7: Esempio dei valori medi

Vengono mostrati anche:

Massimo: Rappresenta il massimo per ogni attributo.

Minimo: Rappresenta il valore minimo per ogni attributo.

Quartili: Forniscono una visione dettagliata della distribuzione dei dati rispetto ad un singolo valore di media.

Varianza: Anche essa, come per la deviazione standard, è una misura di dispersione o variabilità.

Massimo:		Minimo:	
radius_mean	28.11000	radius_mean	6.981000
texture_mean	39.28000	texture_mean	9.710000
perimeter_mean	188.50000	perimeter_mean	43.790000
area_mean	2501.00000	area_mean	143.500000

1° quartile:		Varianza:	
radius_mean	11.700000	radius_mean	12.397094
texture_mean	16.170000	texture_mean	18.466397
perimeter_mean	75.170000	perimeter_mean	589.402799
area_mean	420.300000	area_mean	123625.903080

Figure 8: Dati Statistici

Infine abbiamo creato una matrice di correlazione, utile per comprendere come le diverse variabili si influenzino reciprocamente. Dando uno sguardo alla matrice possiamo notare che ci sono diversi attributi con un'alta correlazione, ma anche diversi attributi con una correlazione bassa o inversa (0 e tendenti al -1)

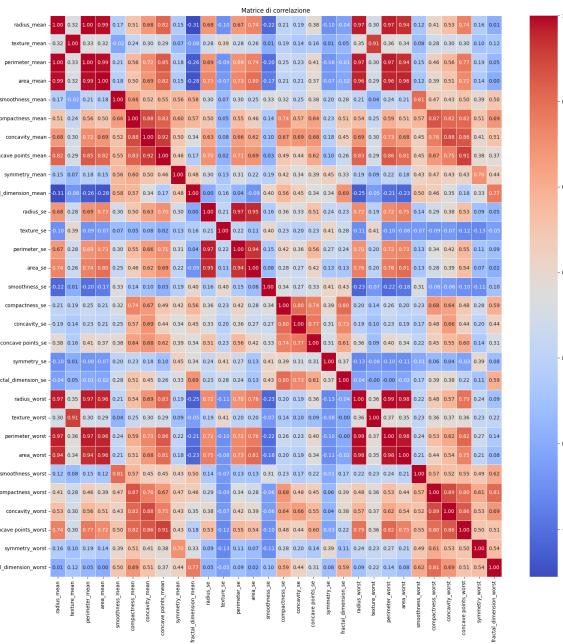


Figure 9: Matrice di correlazione

Abbiamo implementato la generazione della Matrice di correlazione nel modo seguente:

```
def Matrice_Correlazione():
    # Importa il dataset, considerando solo le variabili indipendenti X
    X, _ = import_dataset()

    # Crea una nuova figura per la matrice di correlazione
    plt.figure(num='Matrice di Correlazione', figsize=(20, 20))

    # Aggiunge il titolo alla figura
    plt.title('Matrice di correlazione')

    # Calcola la matrice di correlazione e arrotondala a 2 decimali
    corr_map = X.corr().round(2)

    # Utilizza Seaborn per visualizzare la matrice di correlazione come una heatmap
    sns.heatmap(corr_map, annot=True, cmap='coolwarm', fmt=".2f", linewidths=.5, ax=plt.gca())

    # Aggiungi un po' di spazio alla parte superiore e inferiore della heatmap
    plt.subplots_adjust(top=0.92, bottom=0.15)

    # Mostra la figura
    plt.show()
```

Figure 10: Codice Matrice di Correlazione

3 Classificatori

Abbiamo usato 5 tipi di classificatori, per poter analizzare come modelli di diversa natura potessero affrontare lo stesso problema. Essi sono:

- basati su istanze: K-Nearest Neighbor
- di tipo algoritmico: Albero decisionale
- di tipo classificatore multiplo: Random Forest
- di tipo geometrico: Support Vector Machine
- Artificial Neural Network

Per la creazione dei modelli dell’albero decisionale, del Support Vector Machine e dell’Artificial Neural Network, sono state usate le classi e le funzioni messe a disposizione dalla libreria Sci-kit learn di Python; il KNN e il Random Fortest sono modelli custom creati adhoc.

3.1 K-Nearest Neighbor

Il K-Nearest Neighbor è un algoritmo robusto e versatile, molto utilizzato anche come benchmark per classificatori più complessi, come *ANN* o *SVM*. Il KNN classifica nuovi data points in base alla similarità delle misure dei dati precedentemente registrati. Si tratta di uno dei modelli più usati sia per task di classificazione che di regressione, per la sua semplicità, ma nonostante ciò risulta molto potente in grado di superare le performance di modelli più complessi. È particolarmente indicato per data set etichettati, senza rumori e di dimensioni ridotte, per la sua caratteristica di essere un *lazy learner*. Queste sopra sono tutte caratteristiche del nostro data set.

Nel codice sono stati inseriti tre tipi di calcolo delle distanze, Euclidean, Manhattan e Chebychev, per poter poi confrontare quali hanno performance migliori.

3.1.1 Tuning KNN

Abbiamo implementato una procedura di tuning per il classificatore K-Nearest Neighbors (KNN) al fine di ottimizzare le prestazioni del modello su dati di test. La procedura esplora diverse distanze e valori di k al fine di individuare la combinazione ottimale.

```

def tuning_KNN_custom():
    # Suddivide il dataset in training set e test set
    train_x, test_x, train_y, test_y = split_dataset()

    # Definisci i range di valori per k e tipi di distanza
    k_range = list(range(1, 26))
    dist_range = ['Euclidean', 'Manhattan', 'Chebyshev']

    # Inizializza array per memorizzare le accuratezze durante il tuning
    acc_train = np.empty((len(dist_range), len(k_range)))
    acc_val = np.empty((len(dist_range), len(k_range)))
    acc_test = np.empty((len(dist_range), len(k_range)))

    # Loop attraverso le diverse distanze e valori di k
    for i, dist in enumerate(dist_range):
        for j, k in enumerate(k_range):
            # Crea un classificatore KNN personalizzato con la configurazione corrente
            clf = KNN_Classifier_Custom(k, distance_type=dist)
            print("Distanza: ", dist, " K: ", k)
            # Esegui una 5-fold cross-validation
            scores = cross_validate(estimator=clf, X=train_x, y=train_y, cv=5, n_jobs=10, return_train_score=True, return_estimator=True)
            score_train = scores['train_score']
            score_val = scores['test_score']

            # Calcola l'accuratezza media sul set di test
            score_test = np.mean([estimator.score(test_x, test_y) for estimator in scores['estimator']])

            # Memorizza le accuratezze medie nei rispettivi array
            acc_train[i, j] = score_train.mean()
            acc_val[i, j] = score_val.mean()
            acc_test[i, j] = score_test

    # Chiama la funzione per visualizzare i risultati del tuning del KNN
    plot_tuning_knn(acc_train, acc_val, acc_test, dist_range)

```

Figure 11: Codice in Python del tuning del classificatore KNN

- **Suddivisione del Dataset:** Il codice inizia suddividendo il dataset in un set di addestramento (train set) e un set di test (test set) utilizzando una funzione denominata `split_dataset()`.
- **Distanze esplorate:** Abbiamo considerato tre diverse misure di distanza: **Euclidean**, **Manhattan** e **Chebyshev**.
- **Valori di k :** Abbiamo esplorato valori di k da 1 a 25(inclusi).
- **Inizializzazione degli Array:** Vengono inizializzati tre array vuoti per memorizzare le accuratezze durante il processo di tuning. Gli array sono **acc_train** (accuratezze sul set di addestramento), **acc_val** (accuratezze sulla validation set) e **acc_test** (accuratezze sul set di test).
- **Loop di Tuning con Cross-Validation:** Viene eseguito un doppio loop attraverso i tipi di distanza e i valori di k . Per ogni combinazione di distanza e k , viene creato un classificatore KNN personalizzato utilizzando la classe **KNN_Classifier_Custom**. Successivamente, viene eseguita una 5-fold cross-validation sul set di addestramento (**train_x** e **train_y**). I risultati della cross-validation includono le accuratezze medie sul set di addestramento (**score_train**) e sul set di validation (**score_val**).

- **Calcolo dell'Accuratezza sul Test Set:** Dopo la cross-validation, viene calcolata l'accuratezza media sul set di test utilizzando i modelli addestrati durante la cross-validation. Questo viene fatto per ogni combinazione di distanza e k .
- **Memorizzazione delle Accuratezze:** Le accuratezze medie vengono memorizzate negli array `acc_train`, `acc_val` e `acc_test` corrispondenti alla posizione nel loop (l'indice i per la distanza e l'indice j per k).
- **Visualizzazione dei Risultati:** Infine, abbiamo utilizzato la funzione `plot_tuning_knn()` per visualizzare graficamente le variazioni nelle accuratezze del modello durante il tuning. Questo include l'analisi delle prestazioni su set di addestramento, di validazione e di test al variare della distanza e di k .

Sintesi dei Risultati del Tuning del KNN: I risultati del tuning indicano che il valore ottimale per k è 8, e la distanza che massimizza le prestazioni è la distanza di Manhattan. Questa configurazione ottimale ha dimostrato di evitare l'overfitting al set di validazione, garantendo nel contempo una buona capacità di generalizzazione del modello a nuovi dati, inclusi quelli nel set di test.

Pre-procesing	Accuratezza	Curva Roc (Area)	Sensibilità	Specificità	Precisione	F-score
Nessuno	0.930	0.916	0.969	0.863	0.925	0.946

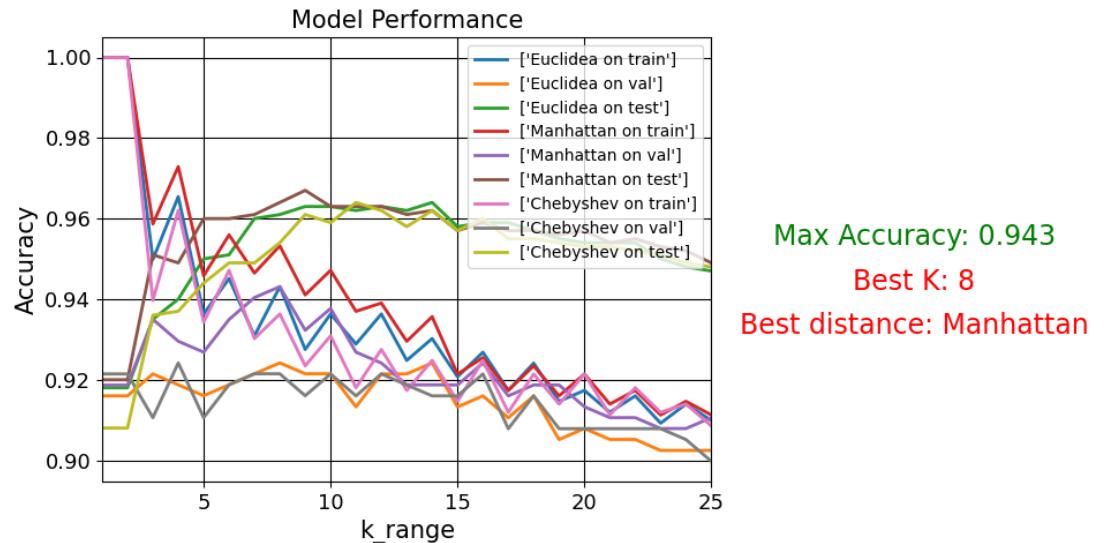


Figure 12: Risultato del Tuning su KNN

3.1.2 Descrizione codice

```

class KNN_Classifier_Custom(BaseEstimator, ClassifierMixin):
    """
    Importando i mixin BaseEstimator e ClassifierMixin e utilizzandoli come superclassi,
    si fornisce alla classe una struttura e un'implementazione che rendono il modello
    compatibile con l'API di scikit-learn. Questo tornerà particolarmente utile quando
    andrà a fare il tuning di k, potendo interfacciarmi direttamente con la funzione
    cross_validate offerta da sklearn.
    """

    def __init__(self, k=8, distance_type="Manhattan"):
        # Inizializzazione dell'istanza del classificatore
        # con i parametri specificati
        self.k = k
        self.distance_type = distance_type

    def fit(self, train_x, train_y):
        # Memorizzazione dei dati di addestramento
        self.train_x = train_x
        self.train_y = train_y

    def chebyshev_distances(self, row_test):
        # Calcolo delle distanze di Chebyshev tra una riga di test e
        # tutte le righe di addestramento
        distanza_chebyshev = []
        for i in range(len(self.train_x)):
            row_train = self.train_x.iloc[i, :]
            distanza_chebyshev.append(np.abs(row_train - row_test).max())
        return distanza_chebyshev

    def calcola_NN(self, distances):
        # Identificazione delle etichette delle k istanze più vicine
        etichette = []
        for i in range(0, self.k):
            indice_dist_minima = np.argmin(distances)
            etichette.append(self.train_y.iloc[indice_dist_minima])
            distances[indice_dist_minima] = float('inf')
        # Determinazione dell'etichetta di classe più comune tra le k istanze più vicine
        label_classe = Counter(etichette).most_common(1)[0][0]
        return label_classe

    def predict(self, test_x):
        # Predizione delle etichette di classe per le istanze di test
        predizioni = []
        for i in range(len(test_x)):
            row = test_x.iloc[i, :]
            # Calcolo delle distanze in base al tipo specificato
            if self.distance_type == "Euclidean":
                distances = euclidean_distances(self.train_x, [row])
            elif self.distance_type == "Manhattan":
                distances = manhattan_distances(self.train_x, [row])
            elif self.distance_type == "Chebyshev":
                distances = self.chebyshev_distances(row)
            else:
                # Tipo di distanza non supportato
                return None
            # Determinazione delle predizioni in base al valore di k
            if self.k > 1:
                predizioni.append(self.calcola_NN(distances))
            elif self.k == 1:
                indice_dist_minima = np.argmin(distances)
                predizioni.append(self.train_y.iloc[indice_dist_minima])
            else:
                # Valore di k non valido
                return None
        return predizioni

    def fit_predict(self, train_x, train_y, test_x):
        # Addestramento del modello e predizione sulle istanze di test
        self.fit(train_x, train_y)
        return self.predict(test_x)

```

Figure 13: Codice Python del modello KNN

- 1. Classe KNN_Classifier_Custom:** La classe è definita come un classificatore personalizzato (KNN_Classifier_Custom) che eredita dalle classi di mixin BaseEstimator e ClassifierMixin di scikit-learn. L'utilizzo di queste superclassi fornisce alla classe una struttura e un'implementazione compatibile con l'API di scikit-learn, rendendo più agevole l'interfacciamento con le funzioni offerte dalla libreria durante il tuning dei parametri.
- 2. Metodo __init__:** Il metodo di inizializzazione (__init__) imposta i parametri principali del classificatore, come il numero di vicini (k) e il tipo di distanza da utilizzare (distance_type), con valori predefiniti di 8 e "Manhattan", rispettivamente. Questi parametri, si sono rivelati i migliori per il Dataset preso in esame.
- 3. Distanza di Chebyshev (chebyshev_distances):** Questo metodo calcola la distanza di Chebyshev tra una riga di dati del set di test e tutte le righe del set di addestramento.
- 4. Metodo fit:** Il metodo fit memorizza i dati di addestramento (train_x e train_y) all'interno dell'istanza del classificatore.
- 5. Metodo calcola_NN:** Il metodo calcola_NN identifica le etichette delle k istanze più vicine e determina l'etichetta di classe più comune tra di esse.

6. **Metodo predict:** Il metodo predict effettua la predizione delle etichette di classe per le istanze di test. A seconda del tipo di distanza specificato, vengono utilizzate le distanze euclidee, di Manhattan o di Chebyshev. La predizione avviene sulla base del valore di k.
7. **Metodo fit_predict:** Questo metodo combina le operazioni di addestramento (fit) e predizione (predict). Prima, il modello viene addestrato sui dati di addestramento forniti e successivamente vengono effettuate le predizioni sulle istanze di test.

3.2 Albero Decisionale

Il Decisional Tree è una delle tecniche di classificazione maggiormente utilizzate. Rappresentano un modo efficiente per prendere decisioni, grazie alla suddivisione del problema in più sotto-insieme. Permette di rappresentare con un albero un insieme di regole di classificazione. È composto da una radice, rappresentato da un attributo iniziale dal quale partire per gli split; nodi interni o rami, i quali rappresentano gli attributi scelti dopo lo split; foglie, ovvero le classi scelte dopo aver percorso l'albero. Il nodo radice viene scelto tra tutti gli attributi in base al guadagno maggiore, calcolato in base all'indice di Gini.

3.2.1 Tuning Albero Decisionale

Il seguente codice è stato progettato per eseguire il tuning di un modello di Albero Decisionale al fine di determinare la profondità ottimale per massimizzare le prestazioni del modello. Questo processo è cruciale per assicurare che l'albero decisionale risponda al meglio ai dati specifici del nostro problema.

```
def tuning_Albero_decisionale():
    # Suddividi il dataset in training set e test set
    train_x, test_x, train_y, test_y = split_dataset()

    # Definisci la gamma di profondità massima degli alberi da esplorare
    max_depth_range = list(range(1, 26))

    # Liste per memorizzare le accuratezze durante il tuning
    acc_train = []
    acc_val = []
    acc_test = []

    # Loop attraverso le diverse profondità degli alberi
    for depth in max_depth_range:
        # Crea un classificatore ad albero decisionale con la profondità corrente
        clf = DecisionTreeClassifier(max_depth=depth, random_state=0)

        # Esegui una 10-fold cross-validation
        scores = cross_validate(estimator=clf, X=train_x, y=train_y, cv=10, n_jobs=10, return_train_score=True, return_estimator=True)

        # Memorizza le accuratezze medie per allenamento, validazione e test
        acc_train.append(np.mean(scores['train_score']))
        acc_val.append(np.mean(scores['test_score']))

        # Calcola l'accuratezza media sul set di test
        acc_test.append(np.mean([estimator.score(test_x, test_y) for estimator in scores['estimator']]))

    # Plotta i risultati del tuning
    plot_tuning_AlberoDecisionale(acc_train, acc_val, acc_test, max_depth_range)
```

Figure 14: Codice in Python del tuning del classificatore Albero Decisionale

- 1. Preparazione dei Dati:** La funzione inizia suddividendo il Dataset in un set di addestramento (train set) e un set di test (test set) utilizzando la funzione `split_dataset()`.
- 2. Definizione della Gamma di Profondità Massima:** Successivamente, un ciclo attraversa una gamma di valori di profondità dell'albero decisionale, da 1 a 25. Ogni iterazione del ciclo rappresenta un tentativo di addestrare e testare l'albero decisionale con una diversa profondità.
- 3. Inizializzazione delle Liste per le Accuratezze:** Vengono inizializzate tre liste vuote (`acc_train`, `acc_val`, `acc_test`) per memorizzare le accuratezze durante il processo di tuning. Queste liste saranno popolate con le accuratezze medie corrispondenti a ciascuna profondità dell'albero.

4. **Loop di Tuning con Cross-Validation:** Viene eseguito un loop attraverso i diversi valori della profondità degli alberi decisionali. Per ogni profondità, viene creato un classificatore ad albero decisionale utilizzando la classe **DecisionTreeClassifier**. Successivamente, viene eseguita una 10-fold cross-validation sul set di addestramento (**train_x** e **train_y**), e vengono calcolate le accuratezze medie per il set di addestramento, il set di validation e il set di test.
5. **Memorizzazione delle Accuratezze:** Le accuratezze medie vengono memorizzate nelle liste **acc_train**, **acc_val**, e **acc_test** corrispondenti alla profondità corrente dell'albero decisionale nel loop.
6. **Visualizzazione dei Risultati:** Una volta completato il tuning, la funzione chiama **plot_tuning_AlberoDecisionale()** per visualizzare graficamente i risultati del tuning dell'albero decisionale. Questa funzione prende in input le liste di accuratezze per il set di addestramento, il set di validation e il set di test, insieme ai valori della profondità.

Sintesi dei Risultati del Tuning dell’Albero Decisionale: I risultati del tuning indicano che la profondità massima ottimale per l’albero decisionale è 6. Questa configurazione ottimale ha dimostrato di evitare l’overfitting al set di validazione, garantendo nel contempo una buona capacità di generalizzazione del modello a nuovi dati, inclusi quelli nel set di test.

Pre-proccesing	Accuratezza	Curva Roc (Area)	Sensibilità	Specificità	Precisione	F-score
Nessuno	0.930	0.916	0.969	0.863	0.925	0.946

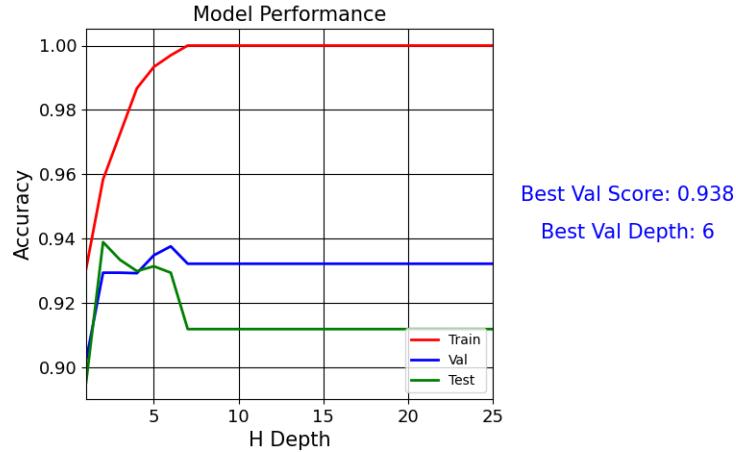


Figure 15: Risultato del Tuning su Decisional Tree

3.2.2 Descrizione codice

```
def decision_tree(train_x, train_y, test_x, depth=6):
    # Creazione di un classificatore ad albero decisionale
    # con profondità massima specificata (la migliore per questo dataset)
    dTree_clf = DecisionTreeClassifier(max_depth=depth, random_state=0)

    # Addestramento del classificatore utilizzando i dati di addestramento
    dTree_clf.fit(train_x, train_y)

    # Predizione delle etichette di classe per i dati di test
    y_pred = dTree_clf.predict(test_x)

    # Restituzione delle etichette predette
    return y_pred
```

Figure 16: Codice in Python del classificatore Albero Decisionale

Il codice, mostrato nella figura posta in alto, definisce una funzione `decision_tree` che implementa un classificatore ad albero decisionale. Questo classificatore viene addestrato su un set di dati di addestramento e successivamente utilizzato per effettuare previsioni su un set di dati di test. La profondità massima dell'albero è un parametro opzionale, e nel caso in cui non venga specificata, il valore predefinito è impostato a 6 che come discusso nella sezione sul tuning è la configurazione ottimale.

1. **DecisionTreeClassifier:** Questa classe proviene dalla libreria scikit-learn ed è utilizzata per creare un modello di albero decisionale.
2. **random_state:** Questo parametro è impostato su 0 per garantire la riproducibilità dei risultati. L'uso di un seed per la generazione di numeri casuali consente di ottenere gli stessi risultati in più esecuzioni.
3. **fit:** Questo metodo addestra il modello utilizzando i dati di addestramento (`train_x` e `train_y`), dove `train_x` sono le feature e `train_y` sono le etichette di classe corrispondenti.
4. **predict:** Dopo l'addestramento, il modello viene utilizzato per prevedere le etichette di classe per i dati di test (`test_x`).

Infine, le etichette predette (`y_pred`) vengono restituite dalla funzione.

3.3 Random Forest

Il Random Forest è un modello di classificazione ampiamente utilizzato in contesti caratterizzati da un elevato numero di dimensioni nei dati. È un modello versatile che dimostra efficacia nel mitigare l'overfitting attraverso la ponderazione delle decisioni di più sotto-alberi decisionali che lo compongono. Questa caratteristica lo rende particolarmente adatto a gestire il problema della maledizione della dimensionalità. La costruzione di ciascun sotto-albero avviene utilizzando un campione casuale di dati e selezionando attributi casuali. Questa strategia mira a garantire che ogni albero affronti il problema da un punto di vista unico, fornendo una diversificazione che contribuisce alla robustezza complessiva del modello.

Avendo molte dimensioni nel data set, abbiamo deciso di creare un Random Forest sfruttando l'Albero Decisionale, per comparare i risultati. Trattandosi di un classificatore composto abbiamo scelto di usare il meccanismo di majority voting.

3.3.1 Tuning Random Forest

La presente implementazione è stata concepita per eseguire il tuning di un modello Random Forest con l'obiettivo di determinare il numero ottimale di alberi nel forest. Questo processo riveste un'importanza fondamentale poiché influisce direttamente sulle prestazioni del modello. L'ottimizzazione del numero di alberi nel Random Forest consente di bilanciare accuratamente la complessità del modello, garantendo che risponda efficacemente ai dati specifici del nostro problema, migliorando così la sua capacità di generalizzazione a nuovi dati.

```
def tuning_Random_Forest(max_num_classifier=125):
    """
    Tuning del modello Random Forest variando il numero di alberi nel forest.

    Parameters:
    - max_num_classifier (int): Numero massimo di alberi da testare nel tuning.
    """
    # Suddivide il dataset in training set e test set
    train_x, test_x, train_y, test_y = split_dataset()

    # Liste per memorizzare le performance del modello per diversi numeri di alberi
    all_score_Random_Forest = []
    all_numbers_trees_forest = []

    # Loop attraverso i diversi numeri di alberi nel forest
    for i in range(2, max_num_classifier + 1):
        print(i)
        rf = RandomForestCustom(numero_alberi = i, random_state = 0)

        # Calcola l'accuracy del modello Random Forest con il numero corrente di alberi
        accuracy = accuracy_score(test_y, rf.fit_predict(train_x, train_y,test_x))
        # Memorizza l'accuracy e il numero corrente di alberi
        all_score_Random_Forest.append(accuracy)
        all_numbers_trees_forest.append(i)

    # Chiama la funzione per visualizzare i risultati del tuning del Random Forest
    plot_tuning_Random_Forest(all_score_Random_Forest, max_num_classifier, all_numbers_trees_forest)
```

Figure 17: Codice in Python del tuning del classificatore Random Forest Custom

- Suddivisione del Dataset:** La funzione inizia suddividendo il dataset in un set di addestramento (train set) e un set di test (test set) utilizzando la funzione `split_dataset()`.
- Inizializzazione delle Liste per le Performance:** Vengono inizializzate due liste vuote, `all_score_Random_Forest` e `all_numbers_trees_forest`, che saranno utilizzate per memorizzare le performance del modello e il numero corrente di alberi nella foresta, rispettivamente.
- Loop di Tuning con Random Forest:** Viene eseguito un loop attraverso i diversi numeri di alberi nel forest. Per ogni iterazione, viene creato un modello Random Forest personalizzato (`RandomForestCustom`) con il numero corrente di alberi. Successivamente, viene calcolata l'accuratezza del modello sul set di test utilizzando la funzione `accuracy_score`.
- Memorizzazione delle Performance:** L'accuratezza e il numero corrente di alberi vengono memorizzati nelle liste `all_score_Random_Forest` e `all_numbers_trees_forest` rispettivamente, per ogni iterazione nel loop.
- Plottaggio dei Risultati del Tuning:** Una volta completato il loop, la funzione chiama `plot_tuning_Random_forest` per visualizzare graficamente i risultati del tuning del Random Forest. Questa funzione prende in input le liste delle performance e dei numeri di alberi, insieme al numero massimo di alberi da testare (`max_num_classifier`).

Sintesi dei Risultati del Tuning del Random Forest: I risultati del tuning rivelano che il numero ottimale di alberi nel Random Forest per il nostro dataset è 28. Questa configurazione ottimale ha dimostrato di evitare l'overfitting al set di validazione, assicurando una robusta capacità di generalizzazione del modello a nuovi dati.

Pre-proccesing	Accuratezza	Curva Roc (Area)	Sensibilità	Specificità	Precisione	F-score
Nessuno	0.930	0.916	0.969	0.863	0.925	0.946

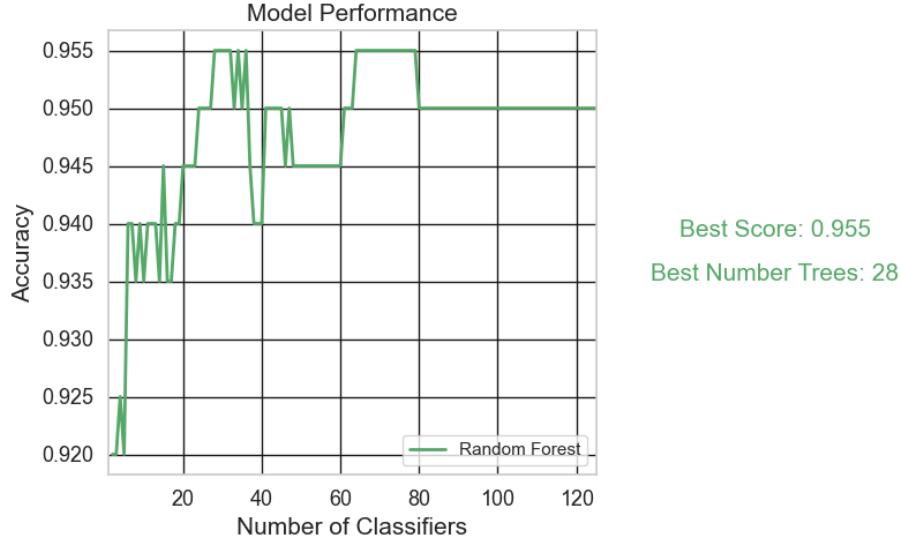


Figure 18: Risultato Tuning del Random Forest

3.3.2 Descrizione codice

```

class RandomForestCustom():
    def __init__(self, numero_alberi = 20, depth = 6, random_state = 0):
        self.numero_alberi = numero_alberi
        self.depth = depth
        self.random_state = random_state
        self.max_depth = depth #profondità massima dei singoli alberi
        self.colonne_utilizzate = [] #list per contenere i nomi delle colonne utilizzate per ciascun albero
        self.colonne_alberi = [] # lista per contenere le colonne utilizzate da ciascun albero
        self.numero_alberi = numero_alberi #numero massimo di alberi della foresta
        self.seed_random = random_state #seme per la random

    def fit(self, train_x, train_y):
        for i in range(0, self.numero_alberi):
            # Inserita il nome casuale
            random.seed(i + self.seed_random)
            np.random.seed(i + self.seed_random)

            # Determina il numero casuale di colonne da estrarre
            numero_colonne_da_estrarre = random.randint(1, len(train_x.columns))

            # Estrarre un numero casuale di colonne dal DataFrame
            colonne_casuali = np.random.choice(train_x.columns, size = numero_colonne_da_estrarre, replace=False)

            #Salvo le colonne selezionate, verranno utili in fase di test
            self.colonne_alberi.append(colonne_casuali)

            #Creare un nuovo DataFrame con le colonne estratte
            df_campione_colonne = train_x[colonne_casuali]

            #Esegui lo split del dataset per selezionare nuovi campioni, nuovo DataFrame campione
            train_x_campione, train_y_campione, _ = train_test_split(df_campione_colonne, train_y, test_size=0.35, random_state=i + self.seed_random)

            #Preparazione del modello con una profondità di default
            dtree_cif = DecisionTreeClassifier(max_depth = self.max_depth, random_state = i + self.seed_random)

            #Salvo l'albero nella foresta generata
            self.alberi_foresta.append(dtree_cif)

            #Addestramento
            dtree_cif.fit(train_x_campione, train_y_campione)

    def predict(self, test_x):
        labels_finali = []

        for i in range(len(test_x)):
            #Seleziona la riga del test set
            row = test_x.iloc[i, :]

            #calcola etichetta
            labels_finali.append(self.hard_voting(row))

        return labels_finali

    def hard_voting(self, record):
        # Votazione rigida (hard voting) per ottenere l'etichetta di classe più comune tra gli alberi
        etichette = []

        for i in range(len(self.alberi_foresta)):
            colonne_utilizzate = self.colonne_alberi[i]
            row_selected = record[colonne_utilizzate].to_frame().transpose()
            etichette.append(self.alberi_foresta[i].predict(row_selected)[0])

        # Tengo l'elemento con l'occorrenza massima (se sono uguali, ne sceglie uno casuale)
        label_classe = Counter(etichette).most_common(1)[0][0]

        return label_classe

    def fit_predict(self, train_x, train_y, test_x):
        # Addestramento del modello e predizione sulle istanze di test
        self.fit(train_x, train_y)
        return self.predict(test_x)

```

Figure 19: Codice Python del modello Random Forest custom

Abbiamo implementato un modello di Random Forest personalizzato, denominato **RandomForestCustom**, con la seguente struttura e funzionalità:

1. Inizializzazione del Random Forest:

- Il costruttore `__init__` inizializza il Random Forest con parametri come il numero massimo di alberi (**numero_alberi**), la profondità massima degli alberi (**depth**), e il seed per la randomizzazione (**random_state**).
- Vengono inizializzate liste vuote (**alberi_foresta** e **colonne_alberi**) per contenere gli alberi della foresta e le colonne utilizzate da ciascun albero, rispettivamente.

2. Metodo di Addestramento (fit):

- Il metodo `fit` addestra il Random Forest. Per ogni albero nella foresta:
 - Si imposta il seed casuale per garantire la riproducibilità dell'addestramento.
 - Si estrae un numero casuale di colonne dal dataset.
 - Si crea un nuovo DataFrame contenente solo le colonne estratte.
 - Si effettua uno split del dataset per selezionare un campione casuale dal nuovo DataFrame campione.
 - Si istanzia un albero di decisione e lo si addestra sul campione.
 - Si salva l'albero e le colonne utilizzate nelle rispettive liste.

3. Metodo di Predizione (predict):

- Il metodo `predict` predice le etichette per un insieme di istanze di test. Utilizza il metodo **hard_voting** per ottenere la previsione finale per ciascuna istanza.

4. Metodo di Hard Voting (hard_voting):

- Il metodo `hard_voting` implementa la votazione rigida, ovvero seleziona l'etichetta di classe più comune tra gli alberi per una determinata istanza.

5. Metodo di Addestramento e Predizione (fit_predict):

- Il metodo `fit_predict` addestra il modello e predice le etichette sulle istanze di test in una singola chiamata.

3.4 Support Vector Machine

Le Support Vector Machines (SVMs) sono potenti modelli di apprendimento supervisionato utilizzati per classificazione e regressione. Le loro principali caratteristiche includono l'efficacia in spazi ad alta dimensionalità, l'efficienza nella gestione della memoria tramite l'uso di vettori di supporto, e la versatilità nella scelta delle funzioni kernel. In Scikit-learn, una comune implementazione è il Support Vector Classifier (SVC), noto per la sua formulazione matematica simile a quella insegnata nelle lezioni. Questo classificatore, adatto anche per problemi multi-classe, segue l'approccio one-vs-one.

3.4.1 Tuning SVM

```

def gridSearchSVM(train_x, train_y, param_grid):
    # Creazione di un classificatore di tipo SVM
    svm_clf = SVC()
    # Numero di fold per la Cross-validation
    n_folds = 5
    # Creazione di un oggetto di tipo GridSearchCV
    grid_search_cv = GridSearchCV(svm_clf, param_grid, cv=n_folds)
    # Esecuzione della ricerca degli iperparametri
    grid_search_cv.fit(train_x, train_y)
    # Stampa risultati
    migliori_parametri = grid_search_cv.best_params_
    # Ottieni l'accuratezza associata alla combinazione ottimale
    best_accuracy = grid_search_cv.best_score_
    return migliori_parametri, best_accuracy

def tuning_SVM():
    train_x, test_x, train_y, test_y = split_dataset()
    # Creazione della griglia di iperparametri per SVM lineare
    param_grid_linear = {'kernel': ['linear'], 'C': [0.01, 0.1, 1, 10, 100]}

    # Creazione della griglia di iperparametri per SVM con RBF
    param_grid_rbf = {'kernel': ['rbf'], 'C': [0.01, 0.1, 1, 10, 100], 'gamma': [0.025, 0.05, 0.1]}

    # dataset moons
    scaler = StandardScaler()
    scld_train_x1 = scaler.fit_transform(train_x)

    #risultato per tuning SVM lineare con ottimizzazione di C
    param_tuning_linear, acc_tuning_linear = gridSearchSVM(scld_train_x1, train_y, param_grid_linear)

    #risultato per tuning SVM con kernel RBF con ottimizzazione di C e gamma
    param_tuning_rbf, acc_tuning_rbf = gridSearchSVM(scld_train_x1, train_y, param_grid_rbf)
    type_model = "SVM"
    #trovo il migliore e lo restituisco
    if(acc_tuning_linear > acc_tuning_rbf):
        esito_cancro = SVM_classifier(train_x, train_y, test_x, param_tuning_linear)
        #plot
        dataset senza_preprocessing(esito_cancro, test_y, type_model,best_c = param_tuning_linear.get('C'),
                                    best_kernel = param_tuning_linear.get('kernel'))
    else:
        esito_cancro = SVM_classifier(train_x, train_y, test_x, param_tuning_rbf)
        #plot
        dataset senza_preprocessing(esito_cancro, test_y, type_model,best_c = param_tuning_rbf.get('C'),
                                    best_kernel = param_tuning_rbf.get('kernel'), best_gamma= param_tuning_rbf.get('gamma'))

```

Figure 20: Codice in Python del tuning del classificatore SVM

Nel corso del nostro processo di ottimizzazione del modello Support Vector Machine (SVM), abbiamo implementato una procedura avanzata di tuning parametrico al fine di massimizzare le prestazioni del modello. Questa procedura si basa su una ricerca esaustiva degli iperparametri ottimali attraverso l'utilizzo della funzione gridSearchSVM. Ora, esploreremo dettagliatamente la logica e l'implementazione di questa procedura all'interno della funzione tuning-SVM,

che gestisce il tuning di un modello SVM con kernel lineare e RBF (Radial Basis Function).

1. **Suddivisione del Dataset:** La funzione inizia suddividendo il dataset in un set di addestramento (train set) e un set di test (test set) utilizzando la funzione `split_dataset()`.
2. **Creazione delle Griglie di Iperparametri:** Vengono definite due griglie di iperparametri per la SVM. Una per il kernel lineare (`param_grid_linear`) con diverse opzioni per il parametro di regolarizzazione C. L'altra per il kernel RBF (`param_grid_rbf`) con opzioni per i parametri di regolarizzazione C e gamma.
3. **Preprocessing del Dataset:** Abbiamo standardizzato il set di addestramento utilizzando uno scaler di tipo `StandardScaler`, per garantire uniformità nei dati di `train_x`, prima di applicare la procedura di tuning. Il risultato viene memorizzato in `scl_train_x1`.
4. **Funzione di Tuning SVM (gridSearchSVM):**
 - La funzione `gridSearchSVM` prende in input i dati di addestramento (`train_x`, `train_y`) e una griglia di iperparametri (`param_grid`) per la SVM.
 - Crea un classificatore SVM (`SVC`) e un oggetto `GridSearchCV` per eseguire una ricerca degli iperparametri.
 - Esegue la ricerca degli iperparametri utilizzando la cross-validation a 5 fold e restituisce i migliori parametri e l'accuratezza associata alla combinazione ottimale.
5. **Tuning della SVM Lineare:** Viene eseguito il tuning della SVM con kernel lineare utilizzando la funzione `gridSearchSVM` con la griglia di iperparametri `param_grid_linear`. Il risultato del tuning (parametri ottimali e accuratezza) viene memorizzato in `param_tuning_linear` e `acc_tuning_linear`.
6. **Tuning della SVM con Kernel RBF:** Viene eseguito il tuning della SVM con kernel RBF utilizzando la funzione `gridSearchSVM` con la griglia di iperparametri `param_grid_rbf`. Il risultato del tuning (parametri ottimali e accuratezza) viene memorizzato in `param_tuning_rbf` e `acc_tuning_rbf`.
7. **Selezione del Miglior Modello:**
 - Il codice determina quale SVM (lineare o con kernel RBF) ha ottenuto un'accuratezza migliore durante il tuning.
 - Se l'SVM lineare ha ottenuto un'accuratezza superiore, viene utilizzato il classificatore SVM lineare con i parametri ottimali per effettuare la classificazione su `test_x`. Viene poi visualizzato un grafico utilizzando la funzione `dataset_senza_preprocessing`.

- Altrimenti, viene utilizzato il classificatore SVM con kernel RBF con i parametri ottimali per effettuare la classificazione su `test_x`, e viene visualizzato un grafico corrispondente.

Sintesi dei Risultati del Tuning del SVM: I risultati del tuning indicano che i parametri ottimali per l'SVM sono $C = 0.1$ e il kernel migliore è risultato essere quello lineare. Questa configurazione ottimale ha dimostrato di fornire l'accuratezza più elevata nel nostro contesto di classificazione.

Pre-procesing	Accuratezza	Curva Roc (Area)	Sensibilità	Specificità	Precisione	F-score
Nessuno	0.970	0.965	0.984	0.945	0.969	0.977

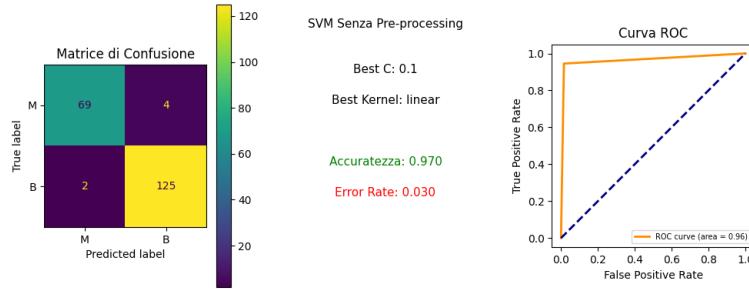


Figure 21: Accuratezza del classificatore SVM

3.4.2 Descrizione codice

```

def SVM_classifier(train_x, train_y, test_x, best_param=None):
    # Crea un oggetto StandardScaler per normalizzare i dati
    scaler = StandardScaler()

    # Adatta lo scaler ai dati di addestramento
    scaler.fit(train_x)

    # Verifica se sono forniti parametri ottimali
    if best_param is not None:
        # Controlla se il kernel è lineare
        if best_param.get('kernel') == 'linear':
            # Utilizza i parametri ottimali se forniti nel dizionario, altrimenti utilizza i valori di default
            svm_clf = SVC(C=best_param.get('C'), kernel=best_param.get('kernel'))
        else:
            # Utilizza i parametri ottimali se forniti nel dizionario, altrimenti utilizza i valori di default
            svm_clf = SVC(C=best_param.get('C'), kernel=best_param.get('kernel'), gamma=best_param.get('gamma'))
    else:
        # Parametri di default (i migliori per questo dataset)
        svm_clf = SVC(C=0.1, kernel='linear')

    # Addestra il classificatore SVM sui dati di addestramento normalizzati
    svm_clf.fit(scaler.transform(train_x), train_y)

    # Effettua previsioni sui dati di test normalizzati
    pred_y = svm_clf.predict(scaler.transform(test_x))

    # Restituisci le previsioni
    return pred_y

```

Figure 22: Codice in Python del classificatore SVM

La funzione **SVM_classifier** è progettata per addestrare un classificatore Support Vector Machine (SVM) utilizzando dati di addestramento, dati di test e, in modo opzionale, un dizionario di parametri ottimali. Ecco una spiegazione dettagliata del codice:

1. Standardizzazione dei dati:

La funzione inizia definendo un oggetto **StandardScaler** della libreria scikit-learn, che sarà utilizzato per normalizzare i dati. Questa standardizzazione è importante per assicurare che tutte le feature abbiano la stessa scala, ottimizzando così le prestazioni del modello SVM.

2. Adattamento dello scaler ai dati di addestramento:

Lo scaler viene adattato ai dati di addestramento utilizzando il metodo **fit**.

3. Gestione dei parametri ottimali (opzionale):

La funzione verifica se sono stati forniti parametri ottimali attraverso il parametro **best_param**. Se sì, la funzione utilizza questi parametri per creare l'oggetto **SVC** (Support Vector Classification) con il kernel specificato (lineare o non lineare).

4. Creazione dell'oggetto SVM:

In base alla tipologia di kernel, l'oggetto **SVC** viene creato con i parametri ottimali forniti o, in mancanza di essi, con valori di default che sono stati determinati come i migliori per il dataset in questione.

5. Addestramento del classificatore SVM:

Il classificatore SVM viene addestrato sui dati di addestramento normalizzati utilizzando il metodo **fit**.

6. Predizione sui dati di test:

La funzione effettua previsioni sui dati di test normalizzati utilizzando il metodo **predict**.

7. Restituzione delle previsioni:

Infine, le previsioni vengono restituite dalla funzione per ulteriori analisi o valutazioni.

La funzione **SVM_classifier** fornisce un'interfaccia semplice per addestrare un classificatore SVM con o senza l'utilizzo di parametri ottimali specificati. Questa funzione è utilizzata nella fase di tuning SVM nel codice precedente per generare le previsioni sul set di test in base ai parametri ottimali ottenuti durante il tuning.

3.5 Artificial Neural Network

L'ANN è un sistema di calcolo concepito per simulare artificialmente il comportamento del cervello umano. Come nei sistemi biologici, apprendere corrisponde a modificare il valore dei pesi delle connessioni sinaptiche in risposta a stimoli esterni. Questo modello è composto in strati, ogni strato elabora le informazioni, elaborate dallo strato precedente. Anche questo modello è utile per la classificazione di dati sia lineare che non lineari.

3.5.1 Tuning ANN

```
def tuning_ANN():
    # Suddivide il dataset in dati di addestramento e test
    train_x, test_x, train_y, test_y = split_dataset()

    # Ignora i FutureWarning per mantenere l'output pulito
    warnings.filterwarnings("ignore")

    # Definisci il set di parametri da esplorare nella ricerca della griglia
    GRID = [
        {'scaler': [StandardScaler(), MinMaxScaler()],
         'estimator': [MLPClassifier(max_iter=100,
                                     solver='sgd',
                                     random_state=0,
                                     learning_rate_init=0.2,
                                     early_stopping=True)],
         'estimator__hidden_layer_sizes': [(20), (30), (40), (40, 20), (50, 30), (50, 30, 10), (50, 40, 30, 20)],
         'estimator__alpha': [0.001, 0.01]
        }]
    # Crea un pipeline con uno scaler e un classificatore di rete neurale
    PIPELINE = Pipeline([('scaler', None), ('estimator', MLPClassifier())])

    # Specifica il numero di fold per la cross-validation
    n_folds = 5

    # Esegui la ricerca della griglia con cross-validation
    grid_search_cv = GridSearchCV(PIPELINE, param_grid=GRID, cv=n_folds)
    grid_search_cv.fit(train_x, train_y)

    # Trova la combinazione migliore di parametri ottenuti dalla ricerca della griglia
    best_combination = grid_search_cv.best_params_

    # Addestra la rete neurale con i migliori parametri trovati
    esito_cancro = Artificial_Neural_Network(train_x, train_y, test_x,
                                              best_combination.get('estimator__hidden_layer_sizes'),
                                              best_combination.get('scaler'),
                                              best_combination.get('estimator__alpha'))

    # Plot della valutazione del modello migliore trovato con il tuning
    dataset_senza_preprocessing(esito_cancro, test_y, 'Rete Neurale',
                                scaler=best_combination.get('scaler'),
                                alpha_value=best_combination.get('estimator__alpha'),
                                hidden_layer=best_combination.get('estimator__hidden_layer_sizes'))
```

Figure 23: Codice in Python del tuning del classificatore ANN

La funzione `tuning_ANN()` esegue il tuning di una rete neurale utilizzando una ricerca della griglia con cross-validation. Di seguito, una spiegazione dettagliata del codice:

1. **Suddivisione del Dataset:** La funzione inizia suddividendo il dataset in un set di addestramento (train set) e un set di test (test set) utilizzando la funzione `split_dataset()`.
2. **Ingnoro i FutureWarning:** Vengono ignorate i Warning per mantenere l'output pulito. Nota: i Warning ignorati NON riguardano errori presenti nel codice, ma semplici **warning di sistema**, sempre presenti, che sporcano l'output.
3. **Definizione del Set di Parametri da Esplorare (GRID):** Viene definito un set di parametri da esplorare durante la ricerca della griglia (**GRID**). Questi includono:
 - Scaler: StandardScaler e MinMaxScaler.
 - Estimator: Un classificatore di rete neurale con vari parametri come `hidden_layer_sizes` e `alpha`.
4. **Creazione del Pipeline:** Viene creato un pipeline (**PIPELINE**) composto da uno scaler e un classificatore di rete neurale. La pipeline coordina le operazioni di normalizzazione dei dati e addestramento del modello, semplificando l'implementazione e la gestione del processo di tuning dell'Artificial Neural Network (ANN)
5. **Specificazione del Numero di Fold per la Cross-Validation:** Viene specificato il numero di fold (`n_folds`) per la cross-validation.
6. **Esecuzione della Ricerca della Griglia con Cross-Validation:** Viene eseguita la ricerca della griglia con cross-validation utilizzando la classe `GridSearchCV` con il pipeline e il set di parametri definiti.
7. **Trova la Migliore Combinazione di Parametri:** Dopo la ricerca della griglia, vengono ottenuti i migliori parametri (`best_combination`) ottenuti dalla ricerca.
8. **Addestramento della Rete Neurale con i Migliori Parametri:** Viene addestrata una rete neurale utilizzando i migliori parametri ottenuti dalla ricerca della griglia utilizzando la funzione `Artificial_Neural_Network`.
9. **Plot della Valutazione del Modello Migliore:** Viene visualizzata la valutazione del modello migliore trovato durante il tuning utilizzando la funzione `dataset_senza_preprocessing`. I parametri ottimali, come lo scaler, l'alpha e la dimensione dello strato nascosto, sono passati come argomenti alla funzione di plotting.

Sintesi dei Risultati del Tuning del ANN: In conclusione è possibile affermare che dopo il tuning dei parametri, i valori ottimali sono i seguenti: **Best_layer = 30**, **best_alpha = 0.001**, e **best_scaler = MinMaxScaler()**. Questi rappresentano la combinazione di parametri che ha massimizzato le prestazioni del modello di rete neurale artificiale nell'ambito del problema di classificazione considerato.

Pre-procesing	Accuratezza	Curva Roc (Area)	Sensibilità	Specificità	Precisione	F-score
Nessuno	0.930	0.916	0.969	0.863	0.925	0.946



Figure 24: Risultato del Tuning su ANN

3.5.2 Descrizione codice

```

def Artificial_Neural_Network(train_x, train_y, test_x, hidden_layer=(30), scaler=MinMaxScaler(), alpha_value=0.0001):
    # Lista di scalar disponibili
    # si usa di default il migliore per questo dataset
    lista_scaler = [StandardScaler(), MinMaxScaler()]

    # Creazione di un classificatore MLP (Multi-Layer Perceptron)
    mlp = MLPClassifier(
        hidden_layer_sizes=hidden_layer,
        max_iter=100,
        alpha=alpha_value,
        solver="sgd",
        verbose=10,
        random_state=0,
        learning_rate_init=0.2,
        early_stopping=True
    )

    # Seleziona lo scalar specificato come parametro
    type_scaler = scaler

    # Trasforma i dati di addestramento utilizzando lo scalar
    scld_train_x = type_scaler.fit_transform(train_x)

    # Addestra il modello sulla base dei dati di addestramento trasformati
    mlp.fit(scld_train_x, train_y)

    # Trasforma i dati di test utilizzando lo stesso scalar
    scld_test_x = type_scaler.transform(test_x)

    # Ottieni le etichette predette
    predicted_labels = mlp.predict(scld_test_x)

    # Restituisce le etichette predette
    return predicted_labels

```

Figure 25: Codice in Python del classificatore ANN

La funzione **Artificial_Neural_Network** si occupa dell’addestramento e della previsione di una rete neurale artificiale (ANN) utilizzando la libreria scikit-learn. Ecco una spiegazione dettagliata del codice:

- Definizione della Funzione:** La funzione **Artificial_Neural_Network** accetta come input i dati di addestramento (**train_x** e **train_y**), i dati di test (**test_x**), e i parametri della rete neurale (**hidden_layer**, **scaler**, **alpha_value**).
- Selezione dello Scaler:** La funzione utilizza il valore di default (**MinMaxScaler()**) se non viene specificato uno scalar diverso come argomento.
- Creazione del Classificatore MLP:** Viene creato un classificatore Multi-Layer Perceptron (MLP) utilizzando la classe **MLPClassifier** di scikit-learn. I parametri del classificatore includono:
 - **hidden_layer_sizes:** specifica il numero di nodi nascosti in ciascuno strato (default è un singolo strato con 30 nodi).
 - **max_iter:** specifica il numero massimo di iterazioni di addestramento (default è 100).
 - **alpha:** parametro di regolarizzazione (default è 0.001).

- **solver:** specifica l'ottimizzatore utilizzato per l'addestramento (in questo caso, "sgd" per la discesa del gradiente stocastica).
 - **verbose:** controlla la quantità di informazioni stampate durante l'addestramento del modello. Impostandolo su valori diversi, si regola il livello di dettaglio degli output durante il processo di addestramento(di default è 10, quindi molto dettagliato).
 - **random_state:** specifica il seed per la riproducibilità (di default è 0).
 - **learning_rate_init:** specifica il tasso di apprendimento iniziale (di default è 0.2).
 - **early_stopping:** specifica se attivare l'arresto anticipato durante l'addestramento (di default è True).
4. **Trasformazione dei Dati di Addestramento:** I dati di addestramento (**train_x**) vengono trasformati utilizzando lo **scaler** selezionato (**type_scaler**).
 5. **Addestramento della Rete Neurale:** Il modello MLP viene addestrato sui dati di addestramento normalizzati utilizzando il metodo **fit**.
 6. **Trasformazione dei Dati di Test:** I dati di test (**test_x**) vengono trasformati utilizzando lo stesso scaler utilizzato per i dati di addestramento.
 7. **Previsione Utilizzando la Rete Neurale Addestrata:** Le etichette predette vengono ottenute applicando il modello addestrato ai dati di test normalizzati con il metodo **predict**.
 8. **Restituzione delle Etichette Predette:** Le etichette predette vengono restituite dalla funzione.

4 Tecniche di Pre-Processing

Gli algoritmi di Machine Learning raramente lavorano su dei Dataset che presentano fin da subito le caratteristiche ideali. Per questo motivo vengono applicate delle tecniche volte a migliorare il funzionamento degli algoritmi che sono di nostro interesse.

4.1 Tecniche di pre-processing utilizzate

Le tecniche di Pre-Processing sono molteplici, noi ne abbiamo usato 5, e sono:

- Campionamento
- Bilanciamento
 - Undersampling
 - Oversampling
 - Mix
- Feature selection
- Combinazione di Attributi
- Trasformazione di Attributi

Analizziamo le Tecniche di Pre-Processing da noi utilizzate, una ad una.

4.2 Campionamento

Come tecnica di Campionamento abbiamo utilizzato la **Stratificazione**. La stratificazione, nell'ambito della divisione di un dataset, è un metodo che assicura che la suddivisione mantenga proporzioni simili delle classi rispetto all'intero dataset originale. In altre parole, serve a garantire una rappresentazione equilibrata delle classi nelle suddivisioni, che può essere cruciale per evitare distorsioni durante la fase di addestramento o valutazione dei modelli di machine learning, specialmente quando si tratta di dataset con classi sbilanciate.

```
def stratificazione(X = None, y = None):
    if(X is None and y is None):
        X, y = import_dataset()
    #split con stratificazione
    train_x, test_x, train_y, test_y = train_test_split(X, y, random_state=0, test_size=0.35, stratify=y)
    return train_x, test_x, train_y, test_y
```

Figure 26: Codice in Python del Campionamento

Pre-procesing	Accuratezza	Curva Roc (Area)	Sensibilità	Specificità	Precisione	F-score
Campionamento	0.945	0.932	0.984	0.880	0.932	0.957

Table 1: Misure Valutazione Knn sul campionamento

Pre-procesing	Accuratezza	Curva Roc (Area)	Sensibilità	Specificità	Precisione	F-score
Campionamento	0.880	0.888	0.856	0.920	0.947	0.899

Table 2: Misure Valutazione Albero Decisionale sul campionamento

Pre-procesing	Accuratezza	Curva Roc (Area)	Sensibilità	Specificità	Precisione	F-score
Campionamento	0.940	0.941	0.936	0.947	0.967	0.951

Table 3: Misure Valutazione Random Forest sul campionamento

Pre-procesing	Accuratezza	Curva Roc (Area)	Sensibilità	Specificità	Precisione	F-score
Campionamento	0.960	0.952	0.984	0.920	0.953	0.969

Table 4: Misure Valutazione SVM sul campionamento

Pre-procesing	Accuratezza	Curva Roc (Area)	Sensibilità	Specificità	Precisione	F-score
Campionamento	0.960	0.957	0.968	0.947	0.968	0.968

Table 5: Misure Valutazione ANN sul campionamento

4.3 Bilanciamento

Come seconda tecnica di Pre-Processing abbiamo utilizzato il **Bilanciamento**. Il bilanciamento si riferisce al processo di gestione degli squilibri nella distribuzione delle classi di un insieme di dati. Quando si lavora con dataset in cui le classi target sono rappresentate in modo non uniforme, cioè alcune classi sono sottorappresentate rispetto ad altre, si può verificare uno squilibrio di classe. Questo squilibrio può influenzare negativamente le prestazioni del modello, in quanto il modello può avere una tendenza a predire più frequentemente le classi sovrarappresentate a scapito di quelle sottorappresentate. Le tecniche di Bilanciamento che abbiamo utilizzato sono le seguenti:

- **Undersampling:** Consiste nel ridurre il numero di istanze della classe maggioritaria (più frequente) in modo da bilanciare la distribuzione delle classi. Questo può essere fatto utilizzando metodi più o meno sofisticati.
- **Oversampling:** Consiste nel aumentare il numero di istanze della classe minoritaria (meno frequente) in modo da bilanciare la distribuzione delle classi.
- **Mix:** Consiste in una combinazione di tecniche di Undersampling e Oversampling, questo metodo ci consente di ottenere un Dataset con esattamente il 50% di record della classe maggioritaria e il restante 50% di record della classe minoritaria, ottenendo un bilanciamento perfetto del Dataset iniziale.

```

def bilanciamento(type = "Oversampling", tecnica="Random"):
    #import dataset
    X, y = import_dataset()
    if (type == "Undersampling"):
        if tecnica == "Random":
            rus = RandomUnderSampler(random_state=0)
            X_resampled, y_resampled = rus.fit_resample(X, y)

        elif tecnica == "IHT":
            # Mappiamo le etichette da stringhe a numeri interi (M --> 1 e B--> 0)
            y = y.replace({'M': 1, 'B': 0})

            iht = InstanceHardnessThreshold(random_state=0)
            X_resampled, y_resampled = iht.fit_resample(X, y)

            # Rimappiamo le etichette numeriche a stringhe
            y_resampled = y_resampled.replace({1: 'M', 0: 'B'})

        elif tecnica == "NearMiss_v1":
            nm = NearMiss(version=1)
            X_resampled, y_resampled = nm.fit_resample(X, y)

        elif tecnica == "NearMiss_v2":
            nm = NearMiss(version=2)
            X_resampled, y_resampled = nm.fit_resample(X, y)

        elif tecnica == "ClusterCentroids":
            warnings.filterwarnings("ignore")

            cc = ClusterCentroids(random_state=0)
            X_resampled, y_resampled = cc.fit_resample(X, y)

    elif tecnica == "Random":
        ros = RandomOverSampler(random_state=0)
        X_resampled, y_resampled = ros.fit_resample(X, y)

    elif(tecnica == "SMOTE"):
        sm = SMOTE(random_state=0)
        X_resampled, y_resampled = sm.fit_resample(X, y)

    elif(tecnica == "ADASYN"):
        ada = ADASYN(random_state=0)
        X_resampled, y_resampled = ada.fit_resample(X, y)

    else:
        raise ValueError("Tipo di bilanciamento non supportato: {}".format(type))

else:
    raise ValueError("Tipo di bilanciamento non supportato: {}".format(type))

#restituisco il dataset corrispondente bilanciato
train_x, test_x, train_y, test_y = split_dataset(X_resampled, y_resampled)
return train_x, test_x, train_y, test_y

```

Figure 27: Codice in Python del Bilanciamento

Pre-procesing	Accuratezza	Curva Roc (Area)	Sensibilità	Specificità	Precisione	F-score
Under-Random	0.919	0.918	0.987	0.849	0.872	0.926
Under-IHT	0.961	0.961	1.000	0.921	0.929	0.963
Under-NearMiss_v1	0.858	0.859	0.921	0.795	0.824	0.870
Under-NearMiss_v2	0.879	0.878	0.947	0.808	0.837	0.889
Under-ClusterCentroids	0.899	0.897	1.000	0.795	0.835	0.910
Over-Random	0.932	0.932	0.934	0.930	0.941	0.937
Over-SMOTE	0.960	0.961	0.949	0.974	0.977	0.963
Over-ADASYN	0.936	0.939	0.912	0.965	0.969	0.940
Mix Under_Over	0.970	0.970	0.980	0.960	0.961	0.970

Table 6: Misure di Valutazione KNN sul Bilanciamento

Pre-procesing	Accuratezza	Curva Roc (Area)	Sensibilità	Specificità	Precisione	F-score
Under-Random	0.926	0.925	0.961	0.890	0.901	0.930
Under-IHT	0.987	0.987	1.000	0.974	0.975	0.988
Under-NearMiss_v1	0.866	0.865	0.908	0.822	0.841	0.873
Under-NearMiss_v2	0.899	0.898	0.961	0.836	0.859	0.907
Under-ClusterCentroids	0.879	0.878	0.961	0.822	0.849	0.901
Over-Random	0.960	0.963	0.934	0.991	0.992	0.962
Over-SMOTE	0.976	0.977	0.971	0.982	0.985	0.978
Over-ADASYN	0.948	0.950	0.934	0.965	0.970	0.952
Mix Under_Over	0.970	0.970	0.970	0.970	0.970	0.970

Table 7: Misure di Valutazione Albero Decisionale sul Bilanciamento

Pre-proccesing	Accuratezza	Curva Roc (Area)	Sensibilità	Specificità	Precisione	F-score
Under-Random	0.933	0.932	0.987	0.877	0.893	0.938
Under-ITH	0.994	0.993	1.000	0.987	0.988	0.994
Under-NearMiss_v1	0.913	0.912	0.961	0.863	0.880	0.918
Under-NearMiss_v2	0.933	0.932	0.987	0.877	0.893	0.938
Under-ClusterCentroids	0.926	0.925	0.947	0.849	0.867	0.906
Over-Random	0.988	0.989	0.978	1.000	1.000	0.989
Over-SMOTE	0.976	0.976	0.978	0.974	0.978	0.978
Over-ADASYN	0.976	0.977	0.971	0.982	0.985	0.978
Mix Under_Over	1.000	1.000	1.000	1.000	1.000	1.000

Table 8: Misure di Valutazione Random Forest sul Bilanciamento

Pre-proccesing	Accuratezza	Curva Roc (Area)	Sensibilità	Specificità	Precisione	F-score
Under-Random	0.960	0.959	0.987	0.932	0.938	0.962
Under-ITH	0.987	0.987	1.000	0.974	0.975	0.988
Under-NearMiss_v1	0.946	0.946	0.947	0.945	0.947	0.947
Under-NearMiss_v2	0.960	0.959	0.974	0.945	0.949	0.961
Under-ClusterCentroids	0.980	0.979	1.000	0.959	0.962	0.981
Over-Random	0.988	0.988	0.993	0.982	0.985	0.989
Over-SMOTE	0.980	0.979	0.993	0.965	0.971	0.982
Over-ADASYN	0.976	0.975	0.985	0.965	0.971	0.978
Mix Under_Over	0.995	0.995	1.000	0.990	0.990	0.995

Table 9: Misure di Valutazione SVM sul Bilanciamento

Pre-proccesing	Accuratezza	Curva Roc (Area)	Sensibilità	Specificità	Precisione	F-score
Under-Random	0.980	0.980	0.974	0.986	0.987	0.980
Under-ITH	0.981	0.980	1.000	0.961	0.963	0.981
Under-NearMiss_v1	0.933	0.933	0.921	0.945	0.946	0.933
Under-NearMiss_v2	0.960	0.960	0.947	0.973	0.973	0.960
Under-ClusterCentroids	0.953	0.953	0.934	0.945	0.947	0.940
Over-Random	0.956	0.955	0.971	0.939	0.950	0.960
Over-SMOTE	0.960	0.959	0.971	0.947	0.957	0.964
Over-ADASYN	0.964	0.966	0.949	0.982	0.985	0.967
Mix Under_Over	0.990	0.990	1.000	0.980	0.980	0.990

Table 10: Misure di Valutazione ANN sul Bilanciamento

4.4 Feature Selection

Per quanto riguarda la terza tecnica di Pre-Processing abbiamo optato per la Feature Selection, o Selezione Degli Attributi. La Feature Selection è una tecnica di Pre-Processing che consiste nel selezionare un sottoinsieme rilevante di caratteristiche (Record) dal dataset originale, eliminando così gli attributi **Ridondanti** e quelli **Irrilevanti** prima di addestrare un modello. L'obiettivo

principale è quindi, quello di ridurre la complessità del modello, migliorare le prestazioni e ridurre il rischio di overfitting.

```
def Feature_Selection(tecnicia = "Correlation-Based"):
    #importo librarie
    X, y = import_dataset()

    if tecnicia == "Correlation-Based":
        # Memorizzo la matrice di correlazione
        corr_df = X.corr()

        # Seleziono gli indici che superano la soglia di correlazione
        indexes = np.where(corr_df > 0.99)

        # Ottengo gli indici delle colonne da rimuovere
        cols_to_remove = set()
        for i, j in zip(*indexes):
            if i != j and i not in cols_to_remove and j not in cols_to_remove:
                cols_to_remove.add(j)

        # Rimuovo le colonne correlate
        cols_to_remove_list = list(cols_to_remove)
        X_reduced = X.drop(X.columns[cols_to_remove_list], axis=1)

    elif tecnicia == "Variance Threshold":
        selector = VarianceThreshold(threshold=1)
        X_reduced = pd.DataFrame(selector.fit_transform(X), columns=X.columns[selector.get_support()])

    elif tecnicia == "Select_K Best":
        selector = SelectKBest(mutual_info_classif, k=10)
        X_reduced = pd.DataFrame(selector.fit_transform(X, y), columns=X.columns[selector.get_support()])

    elif tecnicia == "Sequential Feature":
        warnings.filterwarnings("ignore")

        sfs = SequentialFeatureSelector(KNeighborsClassifier(n_neighbors=1), n_features_to_select=10)
        X_reduced = pd.DataFrame(sfs.fit_transform(X, y), columns=X.columns[list(sfs.get_support(indices=True))])

        warnings.filterwarnings("default")
    else:
        raise ValueError("Tipo di Feature Selection non supportato: {}".format(type))

    #restituisco il ridotto dataset corrispondente
    train_X, test_X, train_y, test_y = split_dataset(X_reduced, y)
    return train_X, test_X, train_y, test_y
```

Figure 28: Codice in Python del Feature Selection

Pre-proccesing	Accuratezza	Curva Roc (Area)	Sensibilità	Specificità	Precisione	F-score
F_S- Correlation Based	0.935	0.928	0.953	0.904	0.945	0.949
F_S- Variance Threshold	0.960	0.954	0.976	0.932	0.961	0.969
F_S- Select K Best	0.955	0.950	0.969	0.932	0.961	0.965
F_S- Sequential Feature	0.945	0.939	0.961	0.918	0.953	0.957

Table 11: Misure Valutazione Knn con Feature Selection

Pre-proccesing	Accuratezza	Curva Roc (Area)	Sensibilità	Specificità	Precisione	F-score
F_S- Correlation Based	0.925	0.926	0.921	0.932	0.959	0.940
F_S- Variance Threshold	0.945	0.945	0.945	0.945	0.968	0.956
F_S- Select K Best	0.920	0.914	0.937	0.890	0.937	0.937
F_S- Sequential Feature	0.930	0.919	0.961	0.877	0.931	0.946

Table 12: Misure Valutazione Albero Decisionale con Feature Selection

Pre-proccesing	Accuratezza	Curva Roc (Area)	Sensibilità	Specificità	Precisione	F-score
F_S- Correlation Based	0.945	0.936	0.969	0.904	0.946	0.957
F_S- Variance Threshold	0.970	0.971	0.969	0.973	0.984	0.976
F_S- Select K Best	0.940	0.932	0.961	0.904	0.946	0.953
F_S- Sequential Feature	0.915	0.907	0.937	0.877	0.930	0.933

Table 13: Misure Valutazione Random Forest con Feature Selection

Pre-procesing	Accuratezza	Curva Roc (Area)	Sensibilità	Specificità	Precisione	F-score
F_S- Correlation Based	0.960	0.951	0.984	0.918	0.954	0.969
F_S- Variance Threshold	0.970	0.968	0.976	0.959	0.976	0.976
F_S- Select K Best	0.935	0.923	0.969	0.877	0.932	0.950
F_S- Sequential Feature	0.950	0.934	0.992	0.877	0.933	0.962

Table 14: Misure Valutazione SVM con Feature Selection

Pre-procesing	Accuratezza	Curva Roc (Area)	Sensibilità	Specificità	Precisione	F-score
F_S- Correlation Based	0.915	0.898	0.961	0.836	0.910	0.935
F_S- Variance Threshold	0.950	0.949	0.953	0.945	0.968	0.960
F_S- Select K Best	0.925	0.906	0.976	0.836	0.912	0.943
F_S- Sequential Feature	0.900	0.872	0.976	0.767	0.879	0.925

Table 15: Misure Valutazione ANN con Feature Selection

4.5 Combinazione di Attributi

Come quarta tecnica di Pre-Processing abbiamo scelto la Combinazione di Attributi. Questa tecnica permette di combinare diversi attributi, ottenendo così un Dataset più ridotto e maggiormente rappresentativo.

Le tecniche per la Combinazione di Attributi da noi adoperate sono molteplici e sono:

- **PCA:** Acronimo di Principal Component Analysis, l'obiettivo principale di questa tecnica è rappresentare un insieme di dati complesso in modo più compatto, mantenendo però la maggior parte delle informazioni rilevanti.

```
if tecnica == "PCA":
    pca = PCA(n_components=10, random_state= 0)
    X_reduced = pd.DataFrame(pca.fit_transform(X), columns=[f'PC{i+1}' for i in range(10)])
```

Figure 29: Codice in Python di PCA per la Combinazione di Attributi

- **Sparse Projection:** La "Sparse Projection" (Proiezione Sparsa) è una tecnica di riduzione della dimensionalità. Questa tecnica fa parte di un gruppo di metodi noti come "Random Projection," che mirano a ridurre le dimensioni di un insieme di dati mantenendo comunque le caratteristiche di interesse. La matrice 2-D viene generata come una matrice randomica sparsa.

```

    elif tecnica == "Sparse Projection":
        srp = SparseRandomProjection(n_components=10, random_state= 0)
        X_reduced = pd.DataFrame(srp.fit_transform(X), columns=[f'SRP{i+1}' for i in range(10)])

```

Figure 30: Codice in Python di Sparse Projection per la Combinazione di Attributi

- **Gaussian Projection:** La "Gaussian Projection" è molto simile alla precedente tecnica di riduzione della dimensionalità, la matrice 2-D viene realizzata con distribuzione Gaussiana densa.

```

    elif tecnica == "Gaussian Projection":
        grp = GaussianRandomProjection(n_components=10, random_state= 0)
        X_reduced = pd.DataFrame(grp.fit_transform(X), columns=[f'GRP{i+1}' for i in range(10)])

```

Figure 31: Codice in Python di Gaussian Projection per la Combinazione di Attributi

- **Feature Agglomeration:** La "Feature Agglomeration" è un approccio che coinvolge l'aggregazione delle caratteristiche simili o correlate in gruppi o cluster. Questo porta ad ottenere un Dataset più bilanciato senza escludere caratteristiche fondamentali di quest'ultimo.

```

    elif tecnica == "Feature Agglomeration":
        fa = FeatureAgglomeration(n_clusters=10)
        X_reduced = pd.DataFrame(fa.fit_transform(X), columns=[f'Cluster{i+1}' for i in range(10)])

```

Figure 32: Codice in Python di Feature Agglomeration per la Combinazione di Attributi

Pre-procesing	Accuratezza	Curva Roc (Area)	Sensibilità	Specificità	Precisione	F-score
C_A- PCA	0.947	0.935	0.986	0.884	0.933	0.959
C_A- Sparse Projection	0.930	0.916	0.972	0.860	0.920	0.945
C_A- Gaussian Projection	0.956	0.946	0.986	0.907	0.946	0.966
C_A- Feature Agglomeration	0.939	0.923	0.986	0.860	0.921	0.952

Table 16: Misure Valutazione Knn con Combinazione Attributi

Pre-procesing	Accuratezza	Curva Roc (Area)	Sensibilità	Specificità	Precisione	F-score
C_A- PCA	0.930	0.925	0.944	0.907	0.944	0.944
C_A- Sparse Projection	0.939	0.932	0.958	0.907	0.944	0.951
C_A- Gaussian Projection	0.939	0.937	0.944	0.930	0.957	0.950
C_A- Feature Agglomeration	0.947	0.949	0.944	0.953	0.971	0.957

Table 17: Misure Valutazione Albero Decisionale con Combinazione Attributi

Pre-procesing	Accuratezza	Curva Roc (Area)	Sensibilità	Specificità	Precisione	F-score
C_A- PCA	0.939	0.919	1.000	0.837	0.910	0.953
C_A- Sparse Projection	0.965	0.958	0.986	0.930	0.959	0.972
C_A- Gaussian Projection	0.947	0.939	0.972	0.907	0.945	0.958
C_A- Feature Agglomeration	0.965	0.958	0.986	0.930	0.959	0.972

Table 18: Misure Valutazione Random Forest con Combinazione Attributi

Pre-procesing	Accuratezza	Curva Roc (Area)	Sensibilità	Specificità	Precisione	F-score
C_A- PCA	0.947	0.935	0.986	0.884	0.933	0.959
C_A- Sparse Projection	0.956	0.946	0.986	0.907	0.946	0.966
C_A- Gaussian Projection	0.930	0.907	1.000	0.814	0.899	0.947
C_A- Feature Agglomeration	0.965	0.958	0.986	0.930	0.959	0.972

Table 19: Misure Valutazione SVM con Combinazione Attributi

Pre-procesing	Accuratezza	Curva Roc (Area)	Sensibilità	Specificità	Precisione	F-score
C_A- PCA	0.956	0.946	0.986	0.907	0.946	0.966
C_A- Sparse Projection	0.947	0.930	1.000	0.860	0.922	0.959
C_A- Gaussian Projection	0.939	0.923	0.986	0.860	0.921	0.952
C_A- Feature Agglomeration	0.965	0.958	0.986	0.930	0.959	0.972

Table 20: Misure Valutazione ANN con Combinazione Attributi

4.6 Trasformazione di Attributi

Come quinta ed ultima tecnica di Pre-Processing abbiamo scelto di adoperare la Trasformazione di Attributi ha come scopo quello di aiutare gli algoritmi di Machine Learning a convergere. Per fare questo abbiamo adottato 3 tipi di trasformazioni:

1. **Standard Scaler:** Lo Standard Scaler è un comune metodo di standardizzazione che prevede la trasformazione dei dati per centralizzarli. Questo processo comporta la rimozione del valore medio da ciascuna caratteristica e la successiva scalatura, ottenuta dividendo le caratteristiche non costanti per la loro deviazione standard: $\frac{x - \text{mean}(x)}{\text{stdev}(x)}$

Questo processo aiuta a garantire che le diverse caratteristiche abbiano **la stessa scala**, il che può essere importante in algoritmi di Machine Learning che sono *sensibili alle differenze di scala* tra le caratteristiche.

2. **Min Max scaler:** Una standardizzazione alternativa è Min Max Scaler. Questo scaler trasforma i dati in modo che essi siano compresi entro un intervallo specifico, comunemente tra 0 e 1.
Si basa sulla seguente formula: $\frac{x - \text{min}(x)}{\text{max}(x) - \text{min}(x)}$
3. **Normalizzazione:** La normalizzazione è il processo di scalatura dei singoli record in modo che abbiano norma unitaria. Può essere eseguita con differenti norme (l1, l2 e lmax) o con la norma indotta dal prodotto scalare.

```
def Trasformazione_Attributi(tecnicia="Standard Scaler"):
    # Import dataset
    X, y = import_dataset()

    if tecnicia == "Standard Scaler":
        scaler = StandardScaler()
        X_transformed = pd.DataFrame(scaler.fit_transform(X), columns=X.columns)

    elif tecnicia == "MinMaxScaler":
        scaler = MinMaxScaler()
        X_transformed = pd.DataFrame(scaler.fit_transform(X), columns=X.columns)

    elif tecnicia == "Normalize":
        X_transformed = pd.DataFrame(normalize(X, norm='l1'), columns=X.columns)

    else:
        raise ValueError("Tipo di Trasformazione Attributi non supportata: {}".format(tecnicia))

    # Suddivide il dataset trasformato
    train_x, test_x, train_y, test_y = train_test_split(X_transformed, y, test_size=0.2, random_state=0)

    # Restituisce il dataset trasformato e suddiviso
    return train_x, test_x, train_y, test_y
```

Figure 33: Codice in Python per la Trasformazione di Attributi

Pre-proccesing	Accuratezza	Curva Roc (Area)	Sensibilità	Specificità	Precisione	F-score
T_A- Standard Scaler	0.956	0.953	0.970	0.936	0.956	0.963
T_A- Min Max scaler	0.956	0.953	0.970	0.936	0.956	0.963
T_A- Normalize	0.956	0.950	0.985	0.915	0.943	0.964

Table 21: Misure Valutazione Knn con Trasformazione Attributi

Pre-proccesing	Accuratezza	Curva Roc (Area)	Sensibilità	Specificità	Precisione	F-score
T_A- Standard Scaler	0.947	0.946	0.955	0.936	0.955	0.955
T_A- Min Max scaler	0.947	0.946	0.955	0.936	0.955	0.955
T_A- Normalize	0.956	0.960	0.940	0.979	0.984	0.962

Table 22: Misure Valutazione Albero Decisionale con Trasformazione Attributi

Pre-proccesing	Accuratezza	Curva Roc (Area)	Sensibilità	Specificità	Precisione	F-score
T_A- Standard Scaler	0.947	0.949	0.940	0.957	0.969	0.955
T_A- Min Max scaler	0.947	0.949	0.940	0.957	0.969	0.955
T_A- Normalize	0.965	0.967	0.955	0.979	0.985	0.970

Table 23: Misure Valutazione Random Forest con Trasformazione Attributi

Pre-proccesing	Accuratezza	Curva Roc (Area)	Sensibilità	Specificità	Precisione	F-score
T_A- Standard Scaler	0.974	0.971	0.985	0.957	0.971	0.978
T_A- Min Max scaler	0.974	0.971	0.985	0.957	0.971	0.978
T_A- Normalize	0.974	0.968	1.000	0.936	0.957	0.978

Table 24: Misure Valutazione SVM con Trasformazione Attributi

Pre-proccesing	Accuratezza	Curva Roc (Area)	Sensibilità	Specificità	Precisione	F-score
T_A- Standard Scaler	0.939	0.929	0.985	0.872	0.872	0.950
T_A- Min Max scaler	0.939	0.929	0.985	0.872	0.872	0.950
T_A- Normalize	0.947	0.949	0.940	0.957	0.969	0.955

Table 25: Misure Valutazione ANN con Trasformazione Attributi

5 Conclusioni

Per fornire una panoramica completa delle performance ottenute durante il corso di questo studio, abbiamo condotto un'analisi comparativa tra varie combinazioni di classificatori e tecniche di pre-processing. Di seguito i risultati:

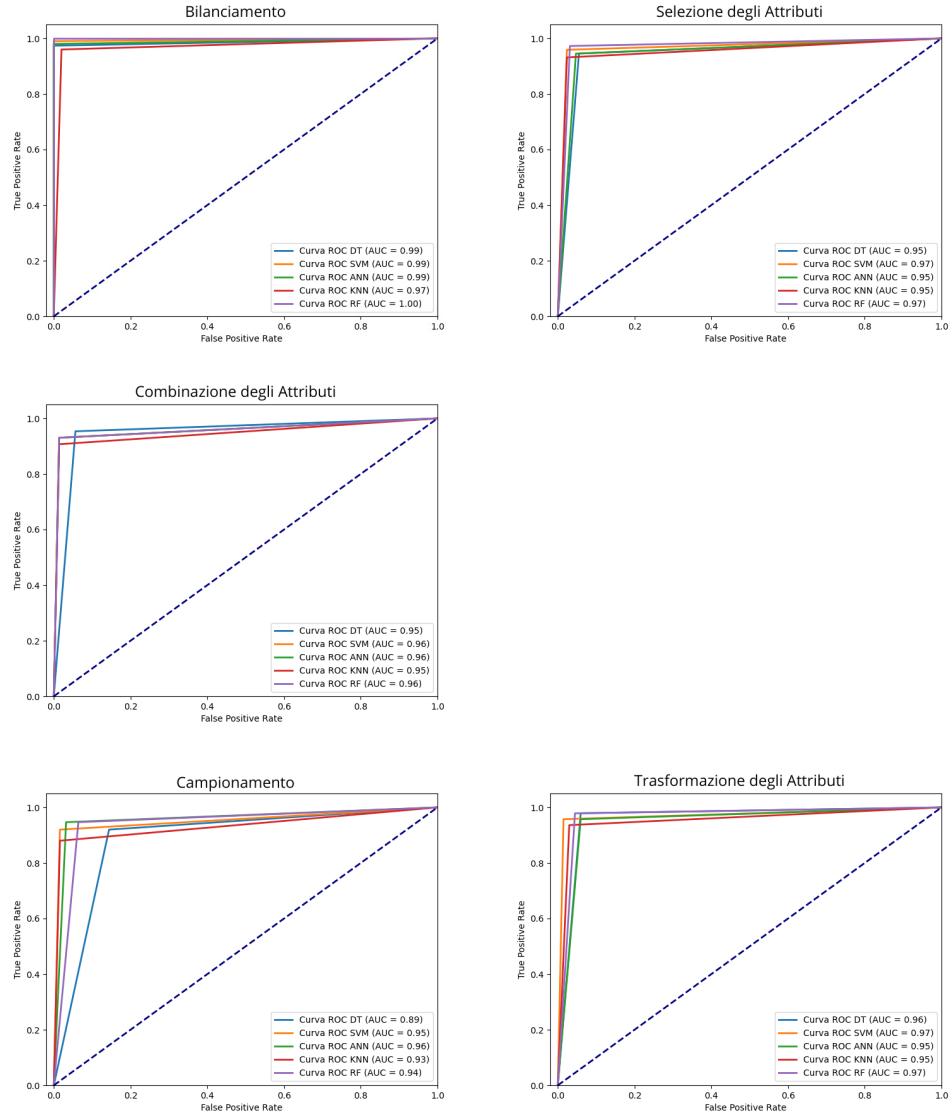


Figure 34: Comparazione dei risultati per ogni tecnica di Pre Processing

Si può notare come, in generale, la tecnica di bilanciamento sia quella che performi meglio tra tutte.

Successivamente abbiamo comparato le migliori combinazioni 'classificatori-tecniche di pre processing' con i risultati dei modelli senza pre processing.

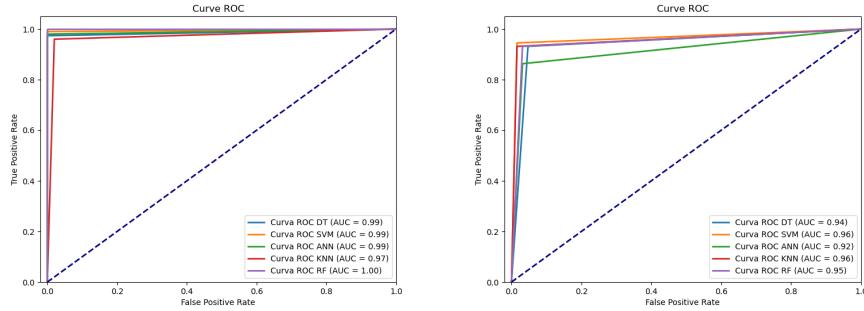


Figure 35: Comparazione delle prestazioni tra le migliori combinazioni di classificatori - Pre Processing (sinistra) e classificatori senza Pre Processing (Destra)

La figura a sinistra riporta i confronti tra le curve ROC delle diverse tecniche di pre-processing utilizzate durante il nostro studio. L'analisi di queste curve fornisce una visione approfondita delle prestazioni dei classificatori.

Nella figura a destra, presentiamo un confronto diretto tra i classificatori senza alcuna forma di pre-processing.

Questa analisi mira a evidenziare l'efficacia dell'implementazione congiunta di modelli di machine learning e strategie di manipolazione del data set.

In conclusione è emerso che il classificatore Random Forest si è dimostrato estremamente efficace nella classificazione del nostro dataset. Con un'accuratezza del 95.5% senza l'utilizzo di tecniche di pre-processing, e addirittura del 100% con l'implementazione di una combinazione di tecniche quali undersampling (SMOTE) e oversampling (IHT).

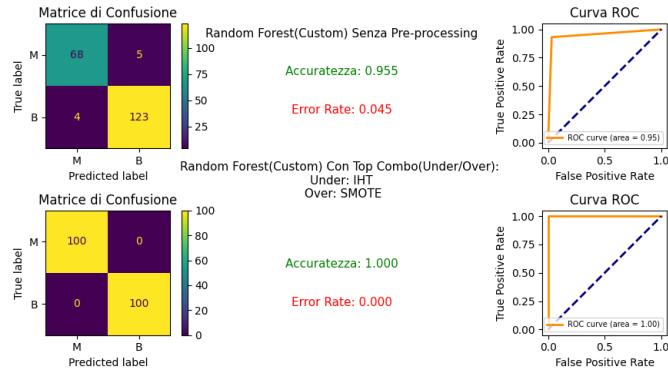


Figure 36: Miglior classificatore

Tuttavia, il raggiungimento di un’accuratezza del 100% ci fa sollevare dei dubbi sulla generalizzazione del modello.

Un risultato così straordinario è sicuramente indicativo di un possibile overfitting. Pensiamo che un motivo per il quale questo accade sia dato dall’utilizzo combinato di queste tecniche, che potrebbe portare il modello a memorizzare particolari configurazioni di dati, le quali non riflettono fedelmente la distribuzione reale del data set, andando così ad addestrare un modello altamente specializzato sul nostro data set di partenza, ma con scarsa capacità di generalizzare su nuovi dati, compromettendo di fatto la sua utilità.