



# POLITECNICO DI BARI

DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELL'INFORMAZIONE

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA DEI SISTEMI MEDICALI

---

RELAZIONE

**Big Data Analytics**

## **Classificazione tramite algoritmi di ML di segnali provenienti da un ECG**

**Professori:**

Prof.ssa S. Colucci

Ing. F. Berloco

**Studentesse:**

Altamura Benedetta

Balestrucci Serena

---

**ANNO ACCADEMICO 2023-2024**

# Indice

<b>1</b>	<b>Dataset di ECG</b>	<b>1</b>
1.1	Descrizione dataset . . . . .	1
1.2	Segnale ECG . . . . .	2
<b>2</b>	<b>Inizializzazione di Spark</b>	<b>8</b>
2.1	Pipeline . . . . .	9
<b>3</b>	<b>Pre-processing dei dati</b>	<b>11</b>
3.1	Controllo dei valori mancanti . . . . .	11
3.2	Bilanciamento . . . . .	12
3.3	PCA . . . . .	14
<b>4</b>	<b>Algoritmi di classificazione</b>	<b>18</b>
4.1	Random Forest . . . . .	18
4.2	Rete Neurale (ANN) . . . . .	20
4.3	Valutazione dei modelli . . . . .	21
4.3.1	Matrice di confusione . . . . .	22
4.3.2	Curve Roc . . . . .	24
4.3.3	Tempi di esecuzione . . . . .	27
<b>5</b>	<b>Simulazione con Spark Streaming delle predizioni sui dati di test</b>	<b>28</b>

# Capitolo 1

## Dataset di ECG

In questo lavoro verranno realizzati e confrontati due algoritmi di classificazione di Machine Learning (ML) con il fine di classificare segnali ECG in cinque classi. Il database è stato analizzato con il framework Spark. Successivamente, è stata simulata l'analisi in tempo reale utilizzando Spark Structured Streaming, per valutare l'applicazione di tali algoritmi in un contesto di streaming continuo dei dati.

### 1.1 Descrizione dataset

Il dataset utilizzato è composto da una raccolta di segnali di battito cardiaco, è stato scaricato dal link di seguito riportato <https://www.kaggle.com/datasets/shayanfazeli/heartbeat>. Questo set di dati è costituito da due file CSV:

- `mitbih_train.csv`
- `mitbih_test.csv`

utilizzati rispettivamente per l'addestramento dei modelli di classificazione e per la valutazione degli stessi. Ciascuno di questi file CSV contiene una matrice, in cui ogni riga rappresenta un tipo di battito cardiaco attraverso l'ampiezza che il segnale raggiunge in un dato momento. L'elemento finale di ogni riga denota la classe a cui appartiene il record. I segnali corrispondono alle forme dell'elettrocardiogramma (ECG) dei battiti cardiaci per i casi normali e per i casi affetti da diverse arit-

mie e infarto del miocardio. Questi segnali sono preelaborati e segmentati, ciascun segmento corrisponde a un battito cardiaco quindi a una categoria.

Di seguito sono riportati dettagli sul dataset:

- Numero di campioni: 109446. Di cui 87554 per il training e 21892 per il test
- Numero di categorie: 5
- Numero di features (colonne escluse l'ultima) : 187
- Frequenza di campionamento: 125Hz
- Classi:
  - 'Battito normale': 0
  - 'Battito prematuro sopraventricolare': 1
  - 'Contrazione ventricolare prematura': 2
  - 'Fusione del battito ventricolare e del battito normale': 3
  - 'Battito non classificabile': 4

In figura 1.1 e 1.2 sono riportati il numero di campioni presenti in ogni classe rispettivamente nel dataframe di train e test.

```

Number of records (train):
+-----+-----+
|_c187|count|
+-----+-----+
|  0.0|72471|
|  1.0| 2223|
|  3.0|  641|
|  2.0| 5788|
|  4.0| 6431|
+-----+-----+

```

Figura 1.1

```

Numeber of records (test):
+-----+-----+
|_c187|count|
+-----+-----+
|  0.0|18118|
|  1.0|  556|
|  2.0| 1448|
|  4.0| 1608|
|  3.0|  162|
+-----+-----+

```

Figura 1.2

## 1.2 Segnale ECG

L'elettrocardiogramma (ECG) è uno strumento diagnostico fondamentale in cardiologia, utilizzato per registrare l'attività elettrica del cuore. Un ECG rileva i segnali

elettrici prodotti dal cuore durante il ciclo cardiaco, permettendo di valutare il ritmo e la funzione cardiaca. L'ECG viene eseguito applicando elettrodi sulla pelle in posizioni specifiche del torace e degli arti, i quali rilevano le variazioni del potenziale elettrico. Le registrazioni prodotte dall'ECG mostrano le onde P, QRS e T, che corrispondono rispettivamente alla depolarizzazione degli atri, alla depolarizzazione dei ventricoli e alla ripolarizzazione dei ventricoli.

Un tracciato ECG presenta il tempo sulle ascisse e il voltaggio sulle ordinate. Sull'asse delle ascisse, il tempo è solitamente rappresentato in millisecondi (ms) o secondi (s), il voltaggio è espresso in millivolt (mV). Questa configurazione consente di misurare la durata delle varie onde e intervalli (come l'onda P, il complesso QRS e l'onda T) e di valutare l'ampiezza delle onde elettriche prodotte dal cuore, fornendo informazioni cruciali per l'analisi della funzione cardiaca e delle aritmie.

Nel dataset considerato sono presenti le seguenti tipologie di aritmie cardiache:

- Battito prematuro sopraventricolare (SVPB): Anche noto come extrasistole sopraventricolare, è un battito prematuro che origina nelle camere superiori del cuore (atri). Si verifica quando un impulso elettrico parte da una parte dell'atrio diversa dal nodo senoatriale, portando a un battito anticipato. Questi battiti prematuri sono generalmente benigni ma possono causare sensazioni di palpitazioni o "salti" nel battito cardiaco. Di seguito si riportano esempi di tracciato ottenuto dal dataset

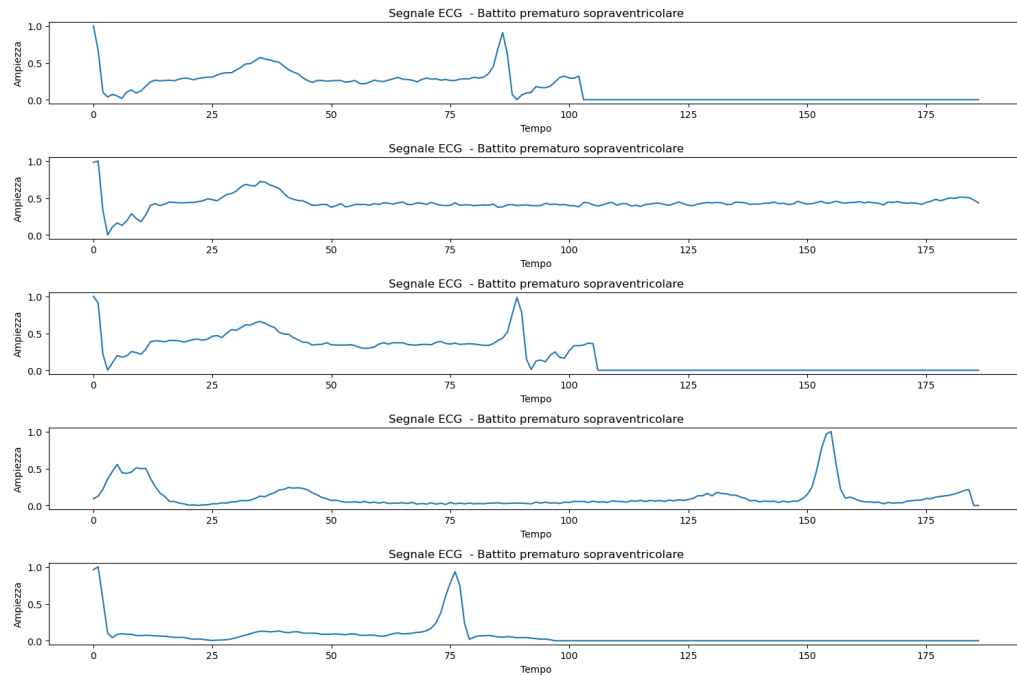


Figura 1.3: Segnale ECG del battito prematuro sopraventricolare

- Contrazione ventricolare prematura (PVC): Conosciuta anche come extrasistole ventricolare, è un battito prematuro che origina nei ventricoli del cuore. Questo tipo di battito è causato da un impulso elettrico che parte in modo anomalo dai ventricoli, provocando un battito prematuro e spesso più forte. Le PVC possono essere avvertite come palpitazioni o come una sensazione di "cuore in gola". In molti casi, sono innocue, ma frequenti PVC possono essere indicativi di una sottostante patologia cardiaca.

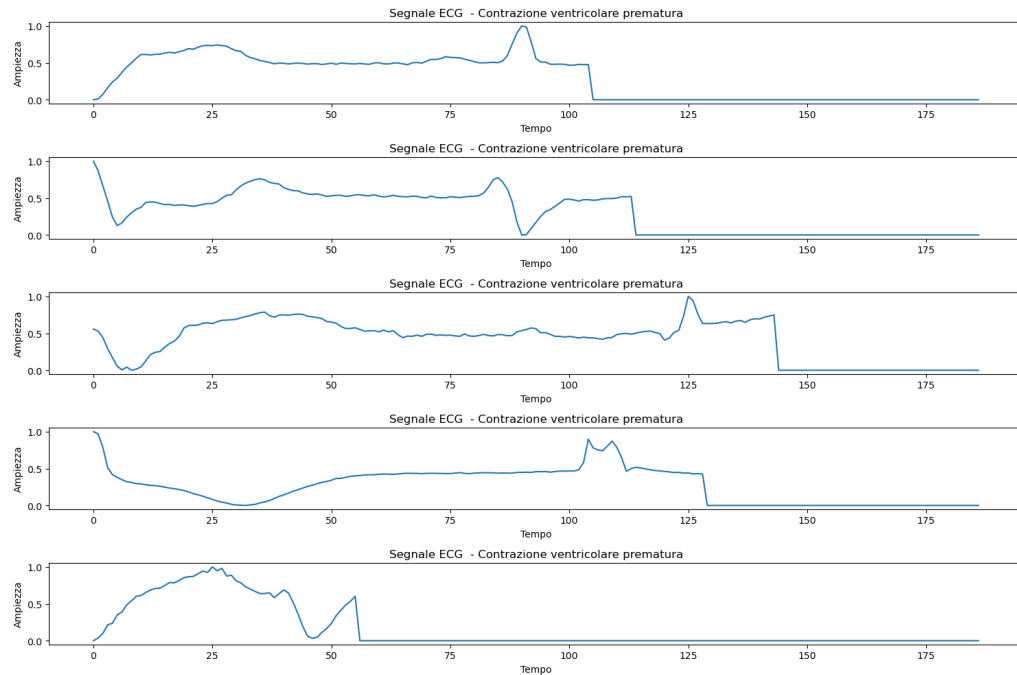


Figura 1.4: Segnale ECG della contrazione ventricolare prematura

- Fusione del battito ventricolare e del battito normale: Questo fenomeno si verifica quando un battito prematuro (PVC) e un battito normale (originato dal nodo senoatriale) avvengono quasi simultaneamente, risultando in un battito di fusione. Il risultato è un complesso QRS ibrido che ha caratteristiche sia del battito normale che del battito prematuro. Questo tipo di battito è un indicatore che i ventricoli sono attivati contemporaneamente da due diverse fonti elettriche.

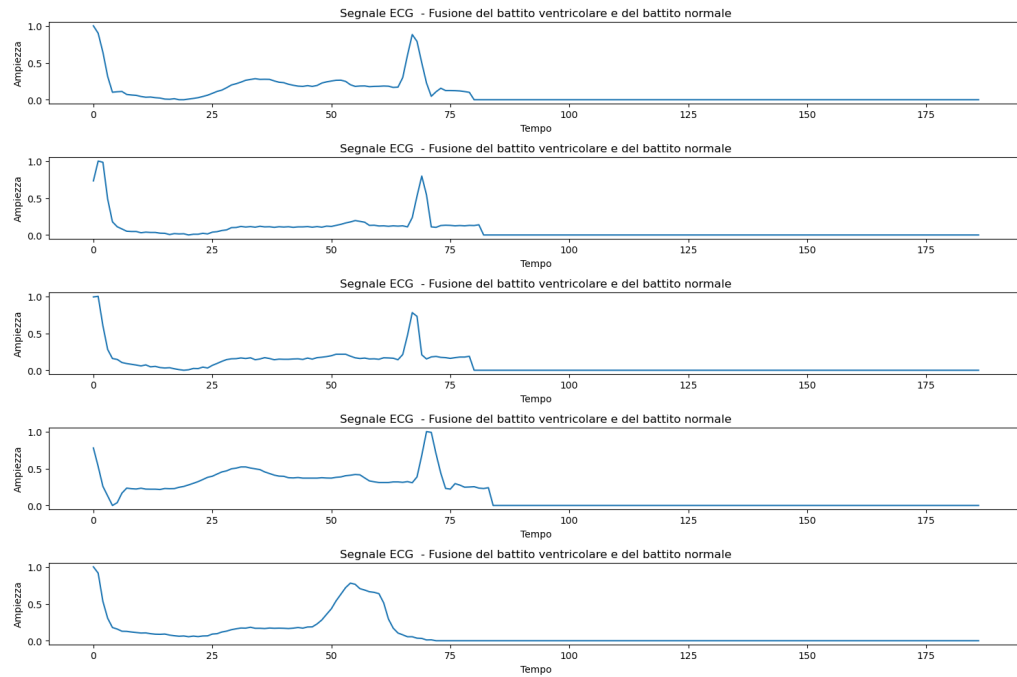


Figura 1.5: Segnale ECG di fusione del battito ventricolare e del battito normale

Per completezza si riportano anche degli esempi di campioni di dato classificato come normale e non classificabile, rispettivamente rappresentati in figura 1.6 e 1.7.

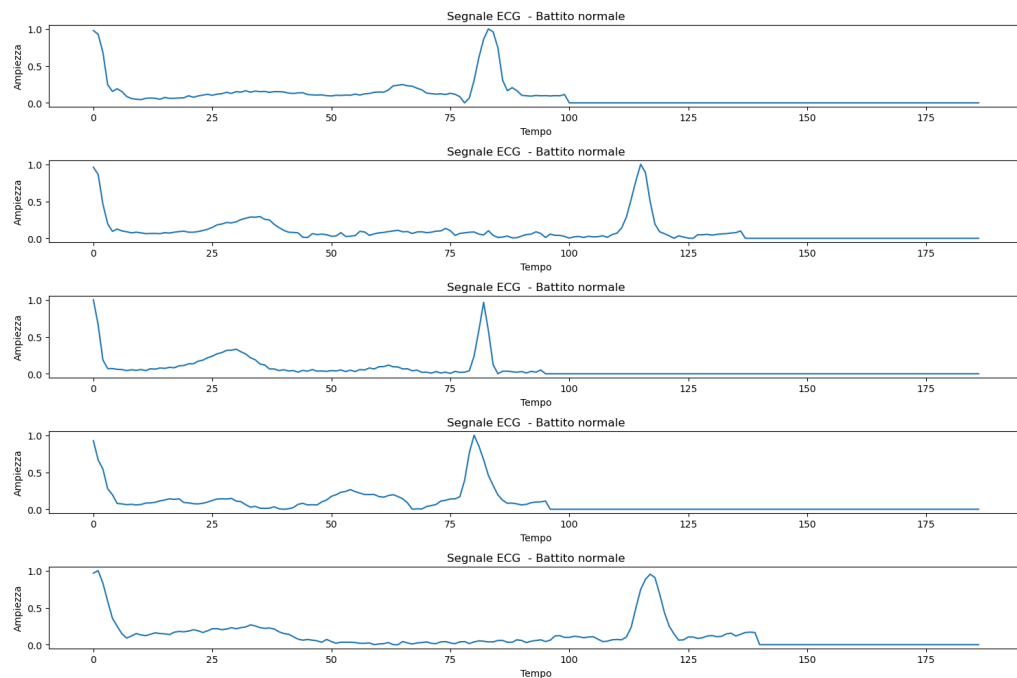


Figura 1.6: Segnale ECG di un battito normale



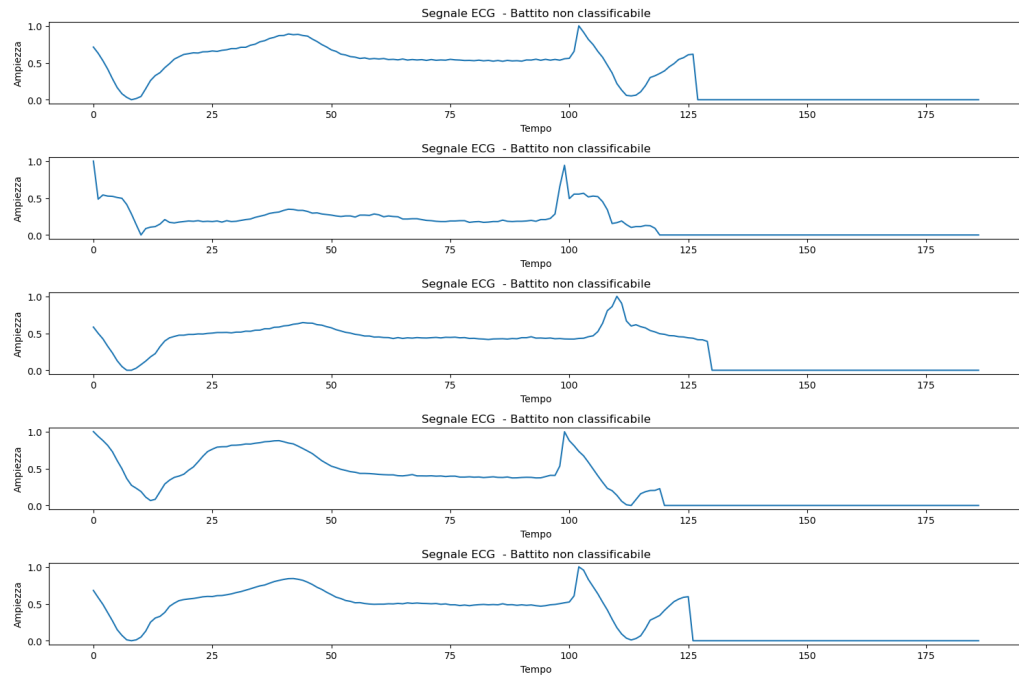
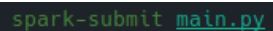


Figura 1.7: Segnale ECG di un battito non classificabile

## Capitolo 2

# Inizializzazione di Spark

La sessione Spark viene avviata e inizializzata quando il file viene eseguito



```
spark-submit main.py
```

Figura 2.1

La configurazione della sessione Spark è ottimizzata per gestire grandi quantità di dati e migliorare le prestazioni, infatti sono state impostate le seguenti configurazioni:

- `spark.driver.memory`: Imposta la quantità di memoria disponibile per il driver Spark a 7 GB, migliorando le capacità di gestione dei dati del driver.
- `spark.executor.memory`: Alloca 6 GB di memoria per gli esecutori Spark, che eseguono le operazioni sui dati distribuiti.
- `spark.driver.maxResultSize`: Limita la dimensione massima dei risultati del driver a 4 GB, prevenendo problemi di memoria.
- `spark.sql.debug.maxToStringFields`: Aumenta il numero massimo di campi visualizzabili durante il debug, facilitando la risoluzione dei problemi.
- `spark.sql.shuffle.partitions`: Imposta il numero di partizioni di shuffle a 200, ottimizzando le operazioni di shuffle durante l'elaborazione dei dati.

Un ulteriore aspetto che è stato considerato riguarda la gestione efficiente della persistenza dei dati. Questo è particolarmente rilevante quando si lavora con dataset di grandi dimensioni utilizzando PySpark. Una delle tecniche utilizzate per migliorare le prestazioni dell'elaborazione dei dati è l'uso del metodo `persist` di PySpark, con un livello di storage specifico.

In particolare si è deciso di persistere i `DataFrame` utilizzando il livello di storage `MEMORY_AND_DISK`; dunque, i dati del `DataFrame` verranno memorizzati in memoria (RAM) e, se la memoria non è sufficiente, verranno scritti su disco.

L'uso del metodo `persist` con il livello di storage `MEMORY_AND_DISK` offre vari vantaggi:

- **Riduzione del Tempo di Calcolo:** Poiché i dati sono già memorizzati e pronti per l'uso, si riduce significativamente il tempo necessario per ricalcolare le stesse operazioni ripetutamente.
- **Ottimizzazione delle Risorse:** Permette una gestione più efficiente delle risorse di memoria e disco, soprattutto in scenari di elaborazione dati intensiva.
- **Miglioramento delle Prestazioni Globali:** In combinazione con altre ottimizzazioni, persistere i dati contribuisce a migliorare le prestazioni globali del sistema di analisi dei dati.

Per monitorare le prestazioni del programma, viene calcolato il tempo totale di esecuzione. Dunque, si memorizza il tempo di inizio dell'esecuzione e il tempo totale trascorso può essere calcolato sottraendo questo valore dal tempo corrente al termine dell'esecuzione. I tempi di esecuzione delle varie sezioni del codice verranno approfonditi nella sezione 4.3.3.

## 2.1 Pipeline

La pipeline seguita per il seguente elaborato è la seguente e sarà discussa nel dettaglio nei paragrafi successivi.

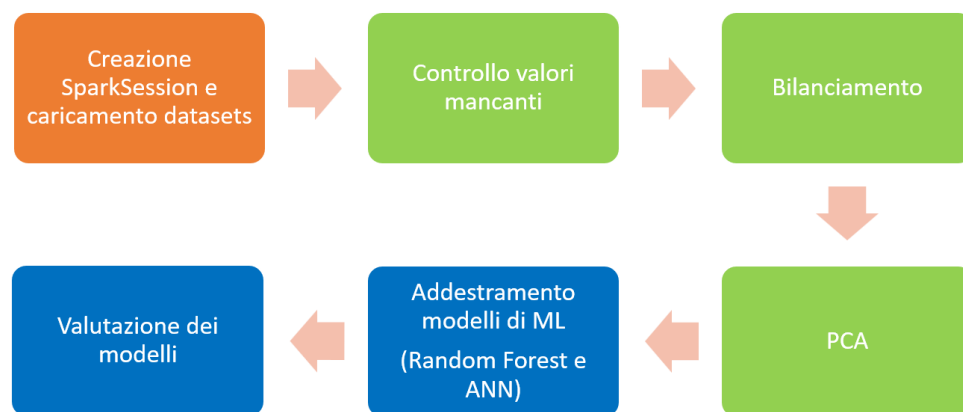


Figura 2.2

## Capitolo 3

# Pre-processing dei dati

Affinchè i dati fossero idonei per l'addestramento dei modelli di Machine Learning sono state effettuate delle operazioni di pre processing su di essi.

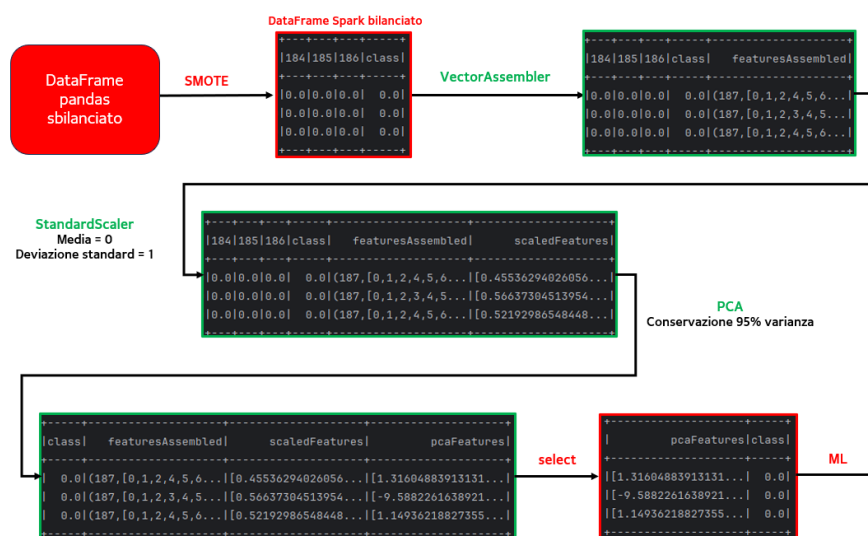


Figura 3.1: Rappresentazione grafica della pipeline. In verde sono evidenziati gli effettivi stages usati nell'oggetto pipeline.

### 3.1 Controllo dei valori mancanti

Nella funzione di preprocessing, si è incluso un controllo per identificare i valori mancanti nei dati di addestramento e nei dati di test utilizzando PySpark. Questo controllo è essenziale per comprendere la qualità dei dati e per determinare le azioni

correttive necessarie prima di procedere con l'analisi o l'addestramento del modello. Attraverso PySpark viene selezionata ogni colonna del DataFrame e viene calcolato il numero di valori nulli. Questo viene fatto tramite una trasformazione che converte i valori nulli in interi (1 se il valore è nullo, altrimenti 0) e li somma. Il risultato di questa operazione viene raccolto in un elenco di colonne e il conteggio dei valori nulli viene memorizzato in un dizionario.

```
missing_values_train = train_df.select(
    [spark_sum(col(c).isNull().cast("int")).alias(c) for c in train_df.columns]
).collect()
missing_values_train_dict = {col: missing_values_train[0][col] for col in train_df.columns}
total_missing_train = sum(missing_values_train_dict.values())
print("Missing values in the training data:")
print(missing_values_train_dict)
print(f"Total missing values in training data: {total_missing_train}")
```

Figura 3.2

Calcolando la somma totale di tutti i valori mancanti nelle diverse colonne si è osservato che nel DataFrame elaborato non sono preseti valori nulli.

```
Missing values in the training data:
{'_c0': 0, '_c1': 0, '_c2': 0, '_c3': 0,
Total missing values in training data: 0
Missing values in the testing data:
{'_c0': 0, '_c1': 0, '_c2': 0, '_c3': 0,
Total missing values in testing data: 0
```

Figura 3.3

## 3.2 Bilanciamento

Vista la grande disomogeneità del dataset come si evidenzia in figura 3.6, come prima operazione di pre processing si è effettuato il bilanciamento.

SMOTE risulta essere vantaggioso perchè creando nuovi esempi della classe minoritaria, può aiutare i modelli di machine learning a imparare meglio le caratteristiche della classe minoritaria, migliorando così le performance generali. Inoltre, a differenza dell'oversampling semplice che replica gli esempi della classe minoritaria, SMOTE genera nuovi esempi, riducendo il rischio di overfitting.

A tale scopo è stato utilizzato SMOTE che in Python è implementato utilizzando la libreria imbalanced-learn.

```
from imblearn.over_sampling import SMOTE
```

Figura 3.4

Affinchè potessero essere utilizzate queste librerie è stato necessario convertire i dati di training e di test vengono convertiti da DataFrame Spark a Dataframe Pandas per poter applicare SMOTE.

```
smote = SMOTE(random_state=42)
X_train_res, y_train_res = smote.fit_resample(X_train, y_train)
```

Figura 3.5

Di seguito si può osservare la nuova distribuzione delle classi prima e dopo il resampling e visualizzare questa distribuzione con un grafico a barre. In particolare dopo lo smote si avranno 72471 campioni per classe.

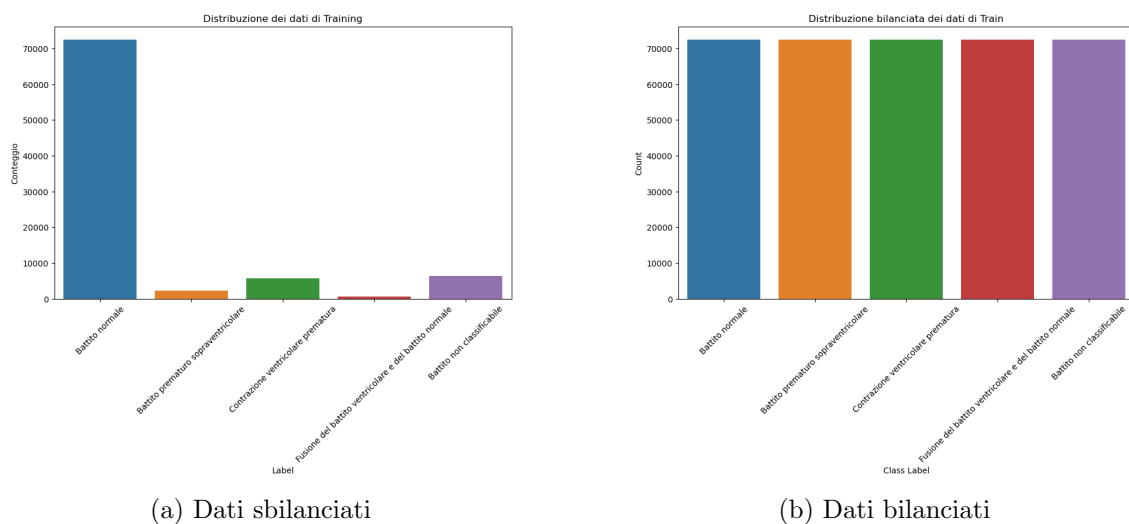


Figura 3.6: Conteggio dati di Train

Si è inoltre implementato un secondo tipo di bilanciamento di tipo undersampling.

```
def undersample_majority_classes(df, target_col, seed=42):
    minority_class_count = df.groupBy(target_col).count().agg({"count": "min"}).collect()[0][0]
    print("Minority class records: ")
    print(minority_class_count)
    classes = df.select(target_col).distinct().collect()
    sampled_df = None

    for cls in classes:
        cls_df = df.filter(F.col(target_col) == cls[target_col])
        fraction = minority_class_count / cls_df.count()
        undersampled_cls_df = cls_df.sample(withReplacement=False, fraction=fraction, seed=seed)
        sampled_df = undersampled_cls_df if sampled_df is None else sampled_df.union(undersampled_cls_df)

    return sampled_df
```

Figura 3.7

Questa funzione ha il compito di sottocampionare le classi maggioritarie. Per farlo si calcola il numero di istanze della classe con il minor numero di campioni e per ogni classe si sottocampiona le istanze in base al numero di istanze della classe minoritaria. Infine, si riuniscono tutti i sottocampionamenti in un unico dataframe. La seguente metodologia di bilanciamento non è stata considerata nella realizzazione del modello finale poichè si ottenevano livelli di accuratezza molto bassi nei modelli di ML. Dunque per le successive elaborazioni si considerano i dati ottenuti dopo l'oversampling (SMOTE).

### 3.3 PCA

L'Analisi delle Componenti Principali (PCA) è una tecnica di riduzione della dimensionalità utilizzata in statistica e machine learning per trasformare un dataset di caratteristiche correlate in un nuovo insieme di caratteristiche non correlate, chiamate componenti principali. Questo metodo è particolarmente utile quando si lavora con dataset ad alta dimensionalità, poiché può semplificare l'analisi riducendo il numero di variabili senza perdere informazioni significative. PCA funziona trovando le direzioni (componenti principali) lungo le quali la varianza dei dati è massimizzata. Le prime componenti principali catturano la maggior parte della varianza presente nei dati originali, permettendo di ridurre la dimensionalità del dataset mantenendo il più possibile l'informazione.

L'implementazione di seguito riportata prevende l'utilizzo della libreria `pyspark.ml.feature` che fornisce le classi `StandardScaler` e `PCA`, che sono strumenti utili per la standar-



dizzazione dei dati e la riduzione della dimensionalità.

```
from pyspark.ml.feature import StandardScaler, PCA
```

Figura 3.8

Affinchè potessero essere utilizzate queste librerie è stato necessario convertire i dati di training `X_train_res` e le label `y_train_res` in un `DataFrame Spark`.

```
# Conversione dati bilanciati in DataFrame Spark
resampled_pd = pd.concat([X_train_res, y_train_res], axis=1)
resampled_pd.columns = list(X_train.columns) + ['label'] # Ensure the label column is named correctly
schema = StructType([StructField(col, DoubleType(), nullable=True) for col in X_train.columns] +
                    [StructField(name='label', DoubleType(), nullable=True)])

train_df_resampled = spark.createDataFrame(resampled_pd, schema)
train_last_column_name = train_df_resampled.columns[-1]
train_df_resampled = train_df_resampled.withColumnRenamed(train_last_column_name, "class").repartition(200)
train_df_resampled.persist(StorageLevel.MEMORY_AND_DISK)
print("Number of records (train_df_resampled): ")
print(train_df_resampled.count())

test_last_column_name = test_df.columns[-1]
test_df = test_df.withColumnRenamed(test_last_column_name, "class").repartition(200)
test_df.persist(StorageLevel.MEMORY_AND_DISK)
print("Number of records (test_df): ")
print(test_df.count())
```

Figura 3.9

La creazione della pipeline di preprocessing creata in seguito, serve a concatenare vari passaggi di trasformazione dei dati in una sequenza ordinata e applicarli in modo coerente sia ai dati di training che ai dati di test. In questo esempio, la pipeline prevede:

- 'VectorAssembler' che combina un insieme di colonne che rappresentano caratteristiche in un singolo vettore di caratteristiche. Il risultato sarà una nuova colonna chiamata "featuresAssembled".
- 'StandardScaler' è usato per scalare le caratteristiche in modo che abbiano media zero e deviazione standard unitaria. Il risultato sarà una nuova colonna chiamata "scaledFeatures" che contiene le caratteristiche scalate.

Questo processo garantisce che i dati siano correttamente preprocessati e pronti per l'addestramento del modello di machine learning.

```
feature_cols = train_df_resampled.columns[:-1]
assembler = VectorAssembler(inputCols=feature_cols, outputCol="featuresAssembled")
scaler = StandardScaler(inputCol="featuresAssembled", outputCol="scaledFeatures", withStd=True, withMean=True)

pipeline = Pipeline(stages=[assembler, scaler])
pipeline_model = pipeline.fit(train_df_resampled)
train_df_scaled = pipeline_model.transform(train_df_resampled)
test_df_scaled = pipeline_model.transform(test_df)
```

Figura 3.10

Successivamente viene calcolato il numero di componenti principali che spiegano una certa percentuale della varianza totale.

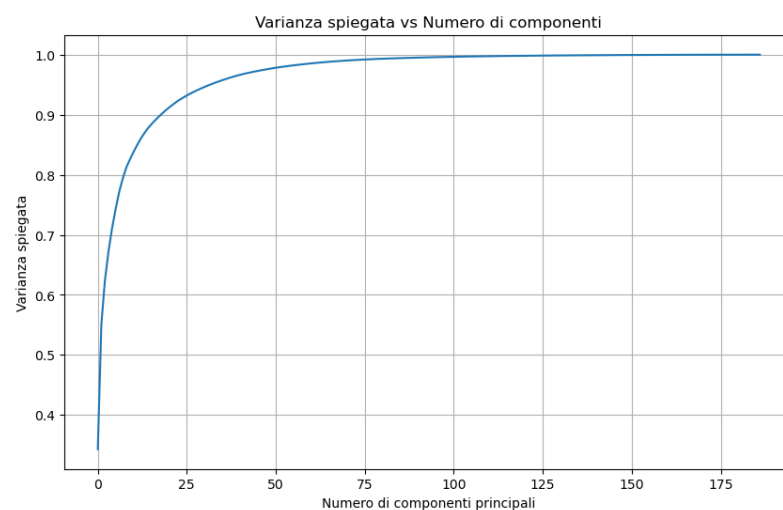


Figura 3.11: Varianza spiegata VS Numero di componenti principali

Col fine di determinare quante componenti principali sono necessarie per spiegare il 95% della varianza nei dati.

```
# PCA iniziale per calcolare il numero di componenti
pca_full = PCA(k=len(feature_cols), inputCol="scaledFeatures", outputCol="pcaFeaturesFull")
pca_model_full = pca_full.fit(train_df_scaled)

# Numero di componenti che spiegano il 95% della varianza
explained_variance = pca_model_full.explainedVariance.toArray()
explained_variance_cumsum = pd.Series(explained_variance).cumsum()
n_components = (explained_variance_cumsum >= 0.95).idxmax()+1
print(f'Number of components explaining 95% of variance: {n_components}')
```

Figura 3.12

```
Numero di componenti per spiegare il 95% della varianza: 33
```

Figura 3.13

Infine si applica la trasformazione PCA ai dati di training e test generandoli "pcaFeatures". Infine dei dataset di training e test con PCA si selezionano le caratteristiche PCA "pcaFeatures" e delle etichette indicizzate "class". I dataset così ottenuti verranno dati in pasto agli algoritmi di ML.

```
# PCA con il numero corretto di componenti
pca = PCA(k=n_components, inputCol="scaledFeatures", outputCol="pcaFeatures")
pca_model = pca.fit(train_df_scaled)
train_pca = pca_model.transform(train_df_scaled).select("pcaFeatures", "class")
test_pca = pca_model.transform(test_df_scaled).select("pcaFeatures", "class")

train_df_resampled.show(3)
test_pca.show(3)

train_df_resampled.unpersist()
test_df.unpersist()
return train_pca, test_pca
```

Figura 3.14

## Capitolo 4

# Algoritmi di classificazione

Al fine di realizzare un modello di classificazione dei tracciati ottenuti dall'elettrocardiogramma sono stati confrontati due algoritmi di classificazione: Random Forest e Rete neurale.

### 4.1 Random Forest

Il Random Forest costruisce una "foresta" di alberi decisionali (decision trees) durante il processo di addestramento. Ogni albero nella foresta viene addestrato su un sottoinsieme casuale del dataset e seleziona un sottoinsieme casuale delle caratteristiche per determinare le decisioni. Questa combinazione di multiple decisioni su sottoinsiemi diversi aiuta a ridurre il problema dell'overfitting che può affliggere i singoli alberi decisionali.

In questa fase, abbiamo addestrato e valutato un modello di Random Forest per la classificazione multiclass con il dataset di segnali ECG pre-elaborato, utilizzando la cross-validation durante il processo di addestramento per garantire robustezza.

Sono state importate librerie essenziali per il machine learning, la visualizzazione dei dati e la valutazione delle prestazioni del modello.

```
from pyspark import StorageLevel
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.ml.classification import RandomForestClassifier
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from pyspark.mllib.evaluation import MulticlassMetrics
from sklearn.metrics import roc_curve, auc
from sklearn.preprocessing import label_binarize
```

Figura 4.1

Inizialmente è stato inizializzato un modello di Random Forest con le colonne delle etichette e delle caratteristiche specificate e viene costruita una griglia di parametri che esplora diverse combinazioni di numTrees (numero di alberi) e maxDepth (profondità massima degli alberi) per ottimizzare il modello.

```
# Inizializzazione RandomForestClassifier
rf = RandomForestClassifier(labelCol="class",
                           featuresCol="pcaFeatures",
                           numTrees=100,
                           seed=42,
                           minInstancesPerNode=2,
                           maxDepth=11)

# Creazione griglia di parametri per la cross-validazione
paramGrid = (ParamGridBuilder()
             .addGrid(rf.numTrees, values=[50, 100])
             .addGrid(rf.maxDepth, values=[5, 11])
             .build())
```

Per misurare l'accuratezza del modello durante la cross-validation viene definito un evaluator. Il modello Random Forest, la griglia di parametri e l'evaluator vengono utilizzati dall'oggetto CrossValidator, con 3 fold, per addestrare il modello sui dati di training. Viene selezionato il miglior modello ottenuto dalla cross-validation per fare previsioni sui dati di test.

```
evaluator = MulticlassClassificationEvaluator(labelCol="class", predictionCol="prediction", metricName="accuracy")
cv = CrossValidator(estimator=rf, estimatorParamMaps=paramGrid, evaluator=evaluator, numFolds=3)

print("Model training with cross-validation...")
cv_model = cv.fit(train_df_pca)
best_model = cv_model.bestModel
```

Figura 4.2

## 4.2 Rete Neurale (ANN)

Come secondo modello è stata implementata una rete neurale artificiale utilizzando la libreria 'MultilayerPerceptronClassifier'

Dunque è stato inizializzato un classificatore MLP (Multilayer Perceptron) con i seguenti parametri

- labelCol: colonna delle etichette indicizzate.
- featuresCol: colonna delle componenti principali ottenute con la PCA.
- maxIter: numero massimo di iterazioni.
- blockSize: dimensione del blocco di input per l'addestramento.
- seed: seme per la generazione di numeri casuali.

```
# Inizializzazione del MultilayerPerceptronClassifier
mlp = MultilayerPerceptronClassifier(labelCol="class", featuresCol="pcaFeatures", maxIter=50, blockSize=64,
                                     seed=1234)
```

Figura 4.3

Anche in questo caso viene creato un oggetto CrossValidator per eseguire la validazione incrociata (3 fold) utilizzando il modello MLP, la griglia di parametri e un evaluator che misura l'accuratezza.

La griglia di parametri è stata costruita per l'ottimizzazione degli iperparametri, infatti sono presenti diverse configurazioni di livelli e iterazioni massime alla griglia. In particolare i layers1 e layers2 sono costituiti da due strati neurali rispettivamente da 70 e 20 neuroni per il primo e 10 e 5 per il secondo.

```
# Creazione di una griglia di parametri per il tuning degli iperparametri
paramGrid = ParamGridBuilder() \
    .addGrid(mlp.layers, values=[layers1, layers2]) \
    .addGrid(mlp.maxIter, values=[10, 450]) \
    .build()
```

Figura 4.4

A questo punto il modello viene addestrato utilizzando la cross-validazione e si seleziona il miglior modello (`best_model`) ottenuto per effettuare le predizioni sul dataset di test

```
predictions = best_model.transform(test_df_pca)
```

Figura 4.5

### 4.3 Valutazione dei modelli

Attraverso la classe 'MulticlassMetrics' è stata possibile valutare le metriche di precisione per valutare le prestazioni complessive del modello sui dati di test.

```
for label in classes.keys():
    class_accuracy = metrics.accuracy
    class_precision = metrics.precision(float(label))
    class_recall = metrics.recall(float(label))
    class_f1 = metrics.fMeasure(float(label))
    print(f"\nMetrics for class {classes[label]} (label {label}):")
    print(f" Accuracy: {class_accuracy:.4f}")
    print(f" Precision: {class_precision:.4f}")
    print(f" Recall: {class_recall:.4f}")
    print(f" F1 Score: {class_f1:.4f}")
```

Figura 4.6

- Accuratezza: misura la proporzione di previsioni corrette rispetto al numero totale di previsioni effettuate. Viene calcolata come il rapporto tra il numero di predizioni corrette e il numero totale di campioni.
- F1 Score: è la media armonica della precisione e del recall. Fornisce un bilanciamento tra precisione e recall, risultando utile quando è importante considerare entrambi. L'F1 score varia tra 0 e 1, dove 1 indica la miglior performance possibile.
- Precisione: misura la proporzione di vere predizioni positive tra tutte le predizioni positive effettuate. È particolarmente utile quando i costi degli errori falsi positivi sono alti.
- Recall: misura la proporzione di veri positivi identificati correttamente rispetto al numero totale di veri positivi presenti nel dataset. È importante quando è cruciale catturare il maggior numero possibile di veri positivi.

I risultati delle predizioni sul dataframe di test sono riportati nelle figure 4.7 e 4.8.

<pre> Metrics for class Normal beat (label 0): Accuracy: 0.9469 Precision: 0.9869 Recall: 0.9542 F1 Score: 0.9783  Metrics for class Supraventricular premature beat (label 1): Accuracy: 0.9469 Precision: 0.4896 Recall: 0.7644 F1 Score: 0.5969  Metrics for class Premature ventricular contraction (label 2): Accuracy: 0.9469 Precision: 0.8397 Recall: 0.9151 F1 Score: 0.8757  Metrics for class Fusion of ventricular and normal beat (label 3): Accuracy: 0.9469 Precision: 0.5236 Recall: 0.8218 F1 Score: 0.6394  Metrics for class Unclassifiable beat (label 4): Accuracy: 0.9469 Precision: 0.9313 Recall: 0.9689 F1 Score: 0.9497 </pre>	<pre> Metrics for class Normal beat (label 0): Accuracy: 0.9284 Precision: 0.9821 Recall: 0.9364 F1 Score: 0.9587  Metrics for class Supraventricular premature beat (label 1): Accuracy: 0.9284 Precision: 0.5138 Recall: 0.7356 F1 Score: 0.6050  Metrics for class Premature ventricular contraction (label 2): Accuracy: 0.9284 Precision: 0.8380 Recall: 0.8930 F1 Score: 0.8646  Metrics for class Fusion of ventricular and normal beat (label 3): Accuracy: 0.9284 Precision: 0.2872 Recall: 0.8580 F1 Score: 0.3337  Metrics for class Unclassifiable beat (label 4): Accuracy: 0.9284 Precision: 0.9452 Recall: 0.9447 F1 Score: 0.9449 </pre>
--	--

Figura 4.7: Risultati modello ANN

Figura 4.8: Risultati modello Random Forest

### 4.3.1 Matrice di confusione

E' stata inoltre implementata la matrice di confusione che è una tabella che descrive le prestazioni di un modello di classificazione. È una tabella di contingenza con due dimensioni "True" (veri) e "Predicted" (previsti) e una dimensione per ciascuna classe. Le dimensioni tipiche per una classificazione binaria (due classi) sono:

- Vero Positivo (TP): Il numero di casi positivi che sono stati correttamente identificati come positivi.
- Falso Positivo (FP): Il numero di casi negativi che sono stati erroneamente identificati come positivi.
- Vero Negativo (TN): Il numero di casi negativi che sono stati correttamente identificati come negativi.
- Falso Negativo (FN): Il numero di casi positivi che sono stati erroneamente identificati come negativi.



Per una classificazione multiclasse, la matrice di confusione viene estesa per includere più classi e i conteggi vengono effettuati per ciascuna coppia di classi vera-prevista. Ogni cella (i,j) nella matrice mostra il numero di istanze che appartengono alla classe i e sono state classificate come classe j. Di seguito si riportano i risultati ottenuti per l'algoritmo di Random Forest

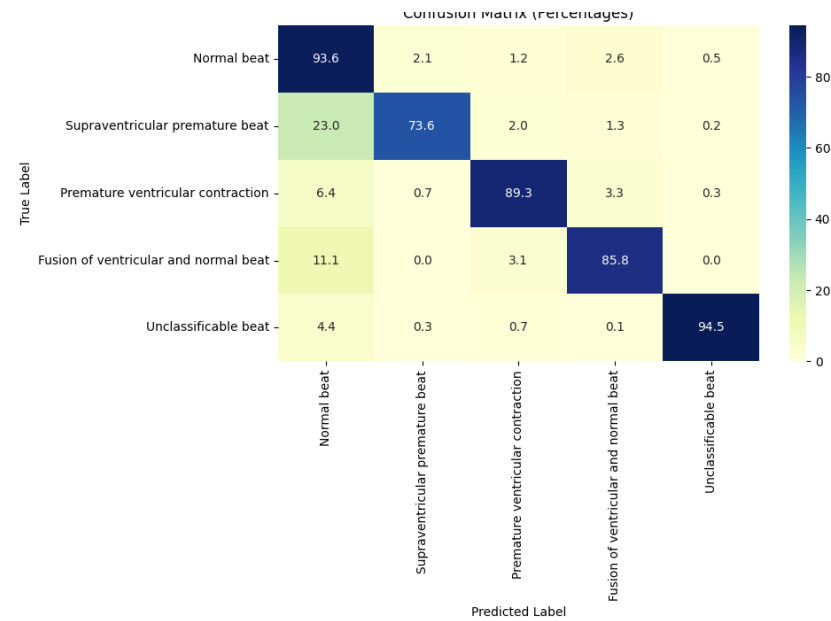


Figura 4.9

e della rete neurale artificiale

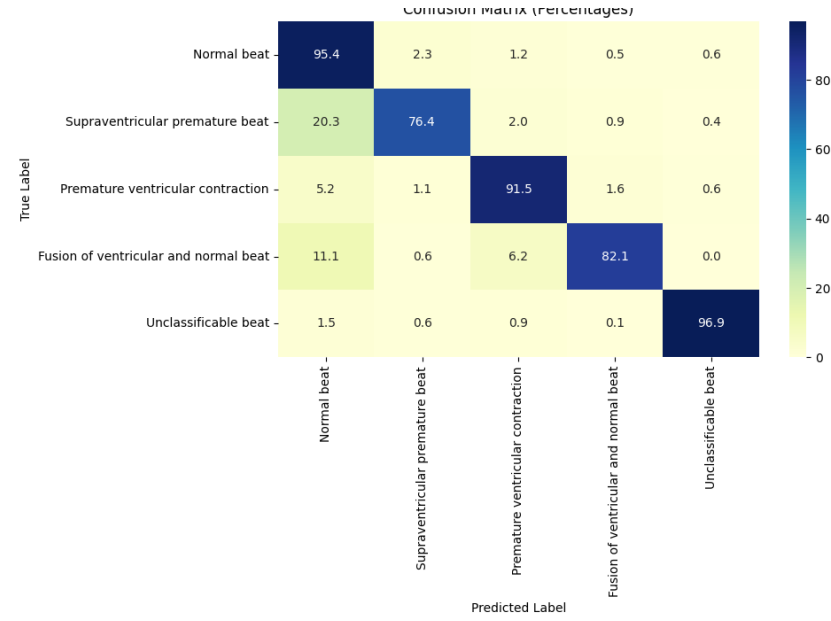
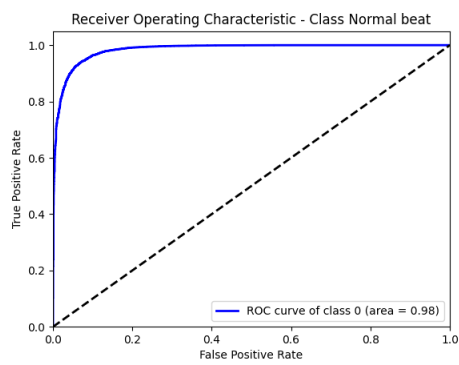


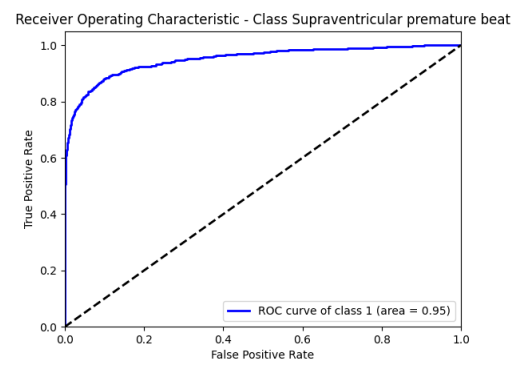
Figura 4.10

### 4.3.2 Curve Roc

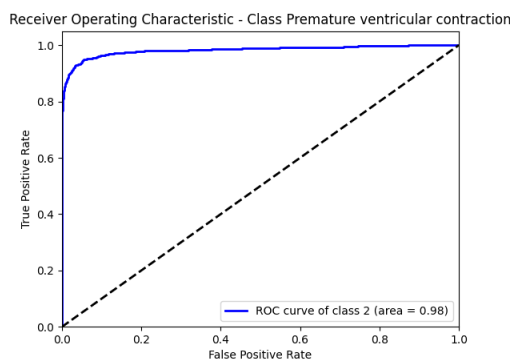
Infine, sono state implementate le curve ROC (Receiver Operating Characteristic) che sono uno strumento grafico utilizzato per valutare le prestazioni di un modello di classificazione binaria. Nel contesto della classificazione multiclasse, le curve ROC possono essere estese considerando l'approccio "one-vs-all", in cui si costruisce una curva ROC per ogni classe trattandola come una classe positiva e tutte le altre classi come negative. Ogni punto sulla curva rappresenta una coppia (Tasso di Falsi Positivi, Tasso di Veri Positivi) ottenuta variando la soglia di classificazione. L'area sotto la curva (AUC - Area Under the Curve) è un indicatore dell'accuratezza del modello: un AUC di 0.5 indica un modello che non ha capacità discriminativa (equivalente a una classificazione casuale), mentre un AUC di 1 indica un modello perfetto. Le curve ROC sono utili perché permettono di confrontare diversi modelli e di scegliere la soglia ottimale che bilancia falsi positivi e falsi negativi in base alle esigenze specifiche dell'applicazione. Di seguito sono riportate le curve ROC corrispondenti alle 5 classi nel caso dell'algoritmo di Random Forest



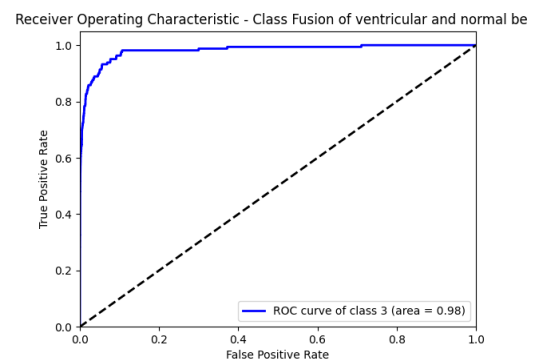
(a) classe 0



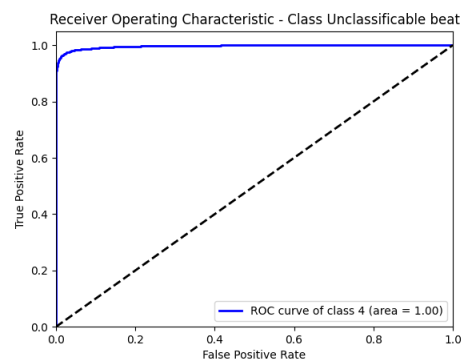
(b) classe 1



(c) classe 2



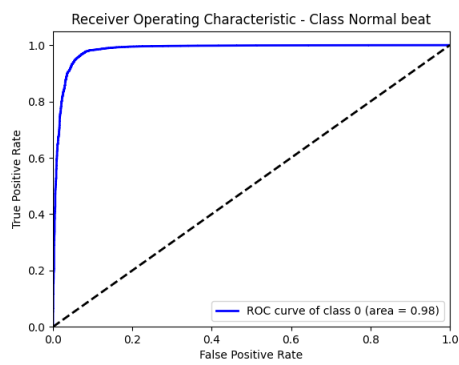
(d) classe 3



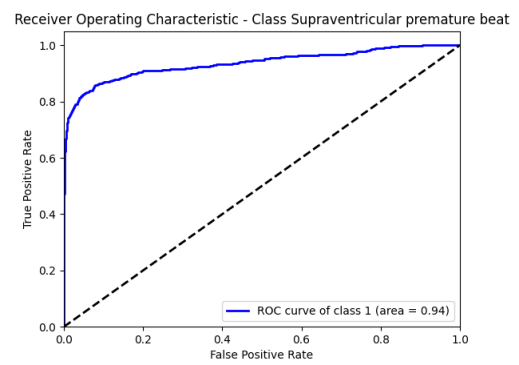
(e) classe 4

Figura 4.11: Curve ROC Random Forest

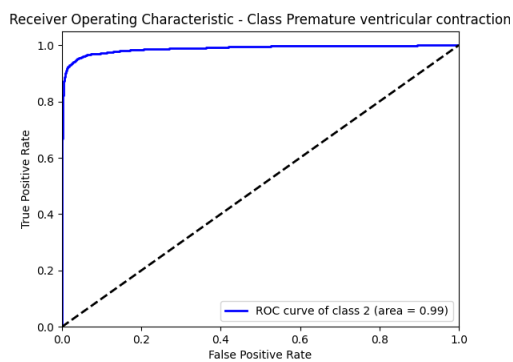
Per quanto riguarda le curve ROC ottenute con ANN si ottiene



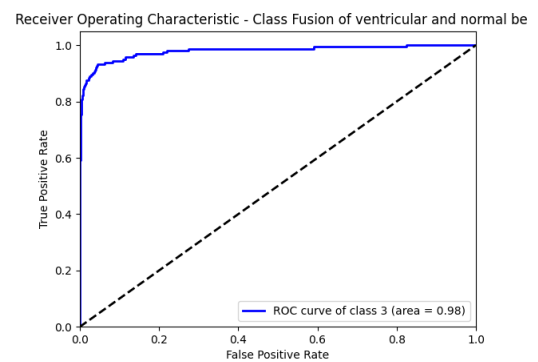
(a) classe 0



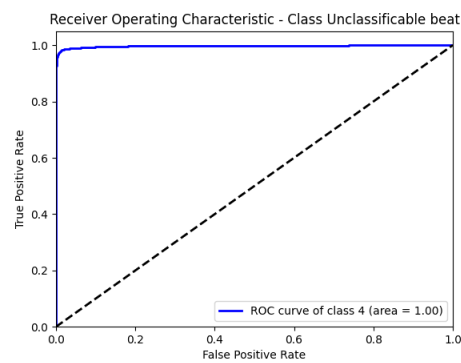
(b) classe 1



(c) classe 2



(d) classe 3



(e) classe 4

Figura 4.12: Curve ROC Rete neurale artificiale

### 4.3.3 Tempi di esecuzione

I tempi di esecuzione relativi al codice completo, alle sole operazioni di pre-processing e alle sole operazioni di addestramento e test di ciascuno dei due modelli sono stati memorizzati in file testuali, come mostrato nelle figure. 4.13 e 4.14.

```
Pre-processing execution time: 86.76 seconds
ANN model execution time: 1270.84 seconds
Total execution time: 1357.60 seconds
```

Figura 4.13: Modello ANN

```
Pre-processing execution time: 83.06 seconds
RF model execution time: 1243.26 seconds
Total execution time: 1326.33 seconds
```

Figura 4.14: Modello Random Forest

## Capitolo 5

# Simulazione con Spark Streaming delle predizioni sui dati di test

Infine, è stata simulata una situazione di streaming dei dati in cui i dati di test vengono letti da un file CSV e, attraverso l'uso di Spark Structured Streaming, questi dati vengono processati in tempo reale per ottenere le predizioni del modello.

Spark Structured Streaming, è un'API di alto livello introdotta in Spark 2.0 per la costruzione di applicazioni di streaming ma è diverso dall'API originale di Spark Streaming, che si basa su `SparkContext` e `StreamingContext`.

Structured Streaming è progettato per essere più semplice e più potente rispetto a Spark Streaming, infatti non richiede l'uso esplicito di `SparkContext` e `StreamingContext`. Invece, utilizza `SparkSession` per gestire il contesto di Spark e permette di lavorare con dati di streaming come se fossero batch, utilizzando `DataFrame` e `Dataset` API.

Per simulare lo streaming dei dati, è stato creato uno script Python che divide il dataset di test in più file CSV più piccoli, generandoli a intervalli regolari di 5 secondi. In questa fase, dunque, viene letto il file CSV originale contenente il dataset di test. In base alla dimensione del dataset e alla dimensione del chunk posta uguale a 100, viene calcolato il numero di file CSV che verranno creati; in modo che ogni chunk di dati viene scritto in un nuovo file CSV nella cartella di output. Tra la

scrittura di un file e l'altro viene inserito un delay di 5 secondi per simulare lo streaming dei dati.

```
File creato: /home/serena/Documenti/Progetto BigDataAnalytics/pythonProject_EC6/Dataset/csv_streaming/output_1.csv
File creato: /home/serena/Documenti/Progetto BigDataAnalytics/pythonProject_EC6/Dataset/csv_streaming/output_2.csv
File creato: /home/serena/Documenti/Progetto BigDataAnalytics/pythonProject_EC6/Dataset/csv_streaming/output_3.csv
File creato: /home/serena/Documenti/Progetto BigDataAnalytics/pythonProject_EC6/Dataset/csv_streaming/output_4.csv
File creato: /home/serena/Documenti/Progetto BigDataAnalytics/pythonProject_EC6/Dataset/csv_streaming/output_5.csv
```

Figura 5.1

A questo punto si è passati alla simulazione dello streaming vera e propria. Risulta fondamentale il caricamento del modello di preprocessing precedentemente creato e del modello di predizione precedentemente addestrato.

```
# Carica il modello di preprocessing completo (PipelineModel che include VectorAssembler, StandardScaler e PCA)
preprocessing_pipeline = PipelineModel.load("/home/serena/Documenti/Progetto BigDataAnalytics/pythonProject_EC6/pipeline_model")

# Carica il modello di predizione (CrossValidatorModel)
prediction_model = CrossValidatorModel.load("/home/serena/Documenti/Progetto BigDataAnalytics/pythonProject_EC6/ann_model")
```

Figura 5.2

Il metodo 'readStream' è il punto di ingresso per la lettura di dati di streaming in Spark. A differenza di read, che legge i dati statici (batch), readStream consente di leggere i dati che vengono continuamente generati, trattandoli come un flusso in tempo reale. Questo è essenziale per applicazioni che richiedono l'elaborazione immediata dei dati non appena arrivano, come nel nostro caso con i file CSV generati ogni 5 minuti. Nella funzione sono state specificate opzioni che includono il tipo di separatore o header e viene specificato lo schema che ci si aspetta di trovare nei file CSV. Si specifica, inoltre, il 'csv\_folder\_path' cioè il percorso della cartella contenente i file CSV che verranno letti come flusso di dati. Spark monitora questa cartella e legge automaticamente qualsiasi nuovo file CSV che vi viene aggiunto.

```
# Leggi i file CSV come uno stream con lo schema specificato
csv_stream = (
    spark.readStream
        .option(key="sep", value=",")
        .option(key="header", value="false")
        .schema(schema)
        .csv(csv_folder_path)
)
```

Figura 5.3

Applicato il modello di preprocessing e di ML ai dati in streaming i risultati delle predizioni vengono scritti in console in tempo reale. Per fare questo si utilizza il metodo 'writeStream' che viene utilizzato per scrivere i dati in una destinazione di streaming, in questo caso la console.

	[prediction]
15986, -0.13808820441066245, 0.028288412830639053, -0.357565701356243, -0.1456664126878476, 0.66592806438528801, 0.6389247182041337, -0.9186102159888726, 0.9349328612996876, 1.5779709478630526]	[0.0]
40492239743734535, 0.1382044351050143, -0.07331776491412009, 0.08236219379772639, -0.0438703326446501, 0.08040957223189423, -0.043625845274131055, -0.022978914453728343, -0.26933636807773015, 0.10420451228177413]	[1.0]
1683406341444, -1.5567755905452891, 0.4551405577360716, 0.2661419259788335, -0.9079840865907107, 0.508492903971528, -0.4595908757793712, 0.06316157804467618, 0.8923617118655532, -0.8202347257016983]	[2.0]
-0.2794812368426346, -0.09221624514913011, 1.5560787142631778, -0.3057364328050408, -1.5310568204756743, 0.5493829484097946, 0.4845847627059443, -0.7937599645458671, -0.6450768064537358]	[2.0]
5238772, -0.6825830016645236, -0.296688814032736, -0.21576405566267387, 0.6951115190605704, -0.936732037296511, 0.9134383427364815, 0.18283833955599388, -0.1912269158152221, 1.1732384803480191]	[4.0]
-0.881813203235216, -1.4956407905121183, -1.0178918372024994, 2.1638524616105437, -0.4656976533634623, -0.36528306565393618, -0.1992224212122153, 1.6114001661737784, -0.499203237130836]	[2.0]
1108408, 0.325814879690355, 0.41297516317803132, -0.4002346233863997, 1.1148440508144443, -0.42364150491024981, 0.14442222383644721, 0.3358424049711724, -0.9433933327388442, 0.5738469718090584]	[1.0]
14504, 1.8101874057768014, 2.3923569279528644, -0.36412878084627474, -0.6122923535135296, -0.441093373792479, 1.1970876435487716, 0.08519492358267792, 1.134342635203786, -1.7617256935801855]	[0.0]

Figura 5.4: Esempio di predizione in tempo reale relativo ad uno dei batch.

Il metodo avvia un'azione di scrittura continua che aggiorna la destinazione con i dati trasformati.

Quando si utilizza 'writeStream', è possibile specificare diverse modalità di output, in questo caso si è utilizzata la modalità 'append' che aggiunge solo le nuove righe risultanti all'output.

```
# Scrivi il risultato nello stream di output (console)
query = (
    predictions.writeStream
        .outputMode("append")
        .format("console")
        .option("truncate", "false") # Opzione per evitare il troncamento dell'output in console
        .start()
)

# Avvia lo stream
query.awaitTermination()
```

Figura 5.5