# Classificazione segnali ECG

Benedetta Altamura, Serena Balestrucci

Politecnico di Bari
Dipartimento di Ingegneria Elettrica e dell'Informazione
Corso di Laurea Magistrale in Ingegneria dei Sistemi Medicali

Anno Accademico 2023-2024

# Introduzione

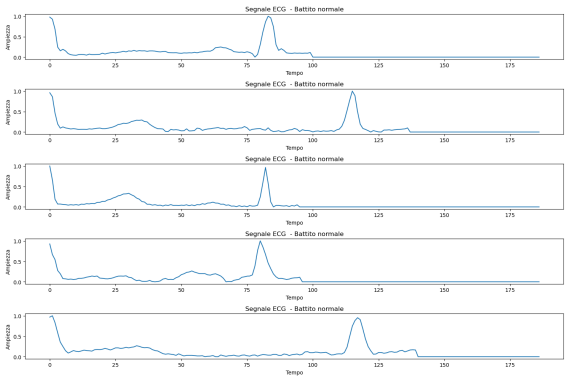
- **OBIETTIVO:** Realizzare e confrontare due algoritmi di classificazione multiclasse di tracciati di ECG e simulare la predizione su batch di dati in streaming
- **TECNOLOGIE:** Apache Spark, PyCharm
- **MODULI DI SPARK:** Spark SQL, Spark Streaming, MLlib


# Descrizione dataset

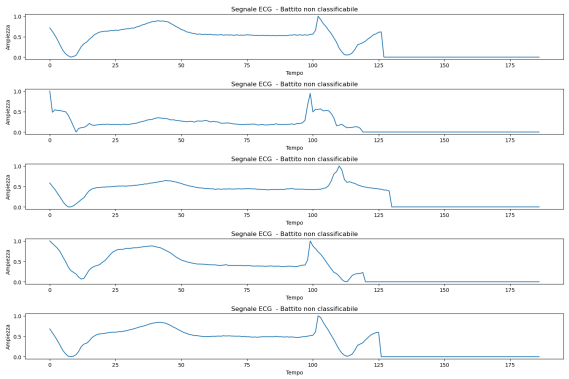
Il dataset utilizzato è composto da una raccolta di segnali di battito cardiaco, scaricato dal link di seguito riportato `https://www.kaggle.com/datasets/shayanfazeli/heartbeat`. all'interno del quale sono presenti i csv 'mitbih_train.csv' e 'mitbih_test.csv'

- **Numero di campioni:** 109446 (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 (label):**
  - 'Battito normale': 0
  - 'Battito prematuro sopraventricolare': 1
  - 'Contrazione ventricolare prematura': 2
  - 'Fusione del battito ventricolare e del battito normale': 3
  - 'Battito non classificabile': 4

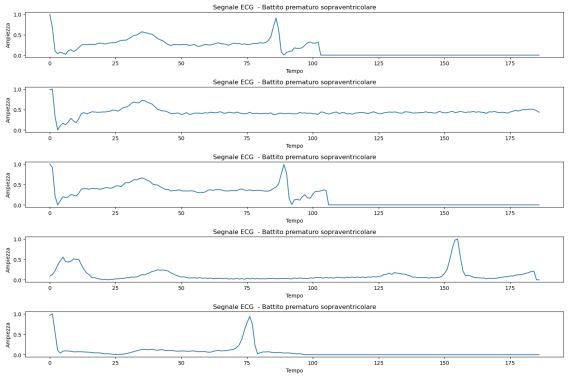
# Esempi di tracciati ECG



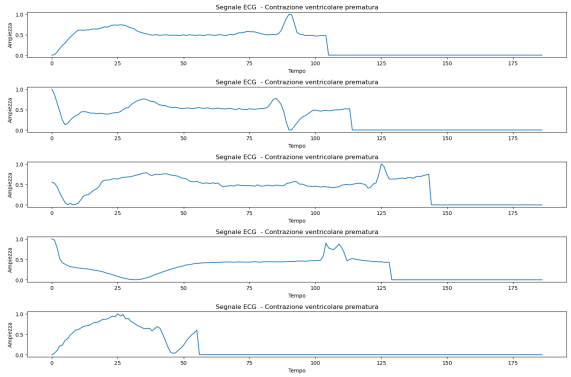

Battito normale



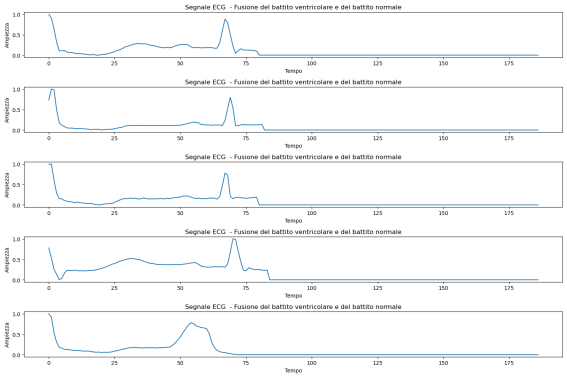

Battito non classificabile




Battito prematuro sopra ventricolare




Contrazione ventricolare prematura

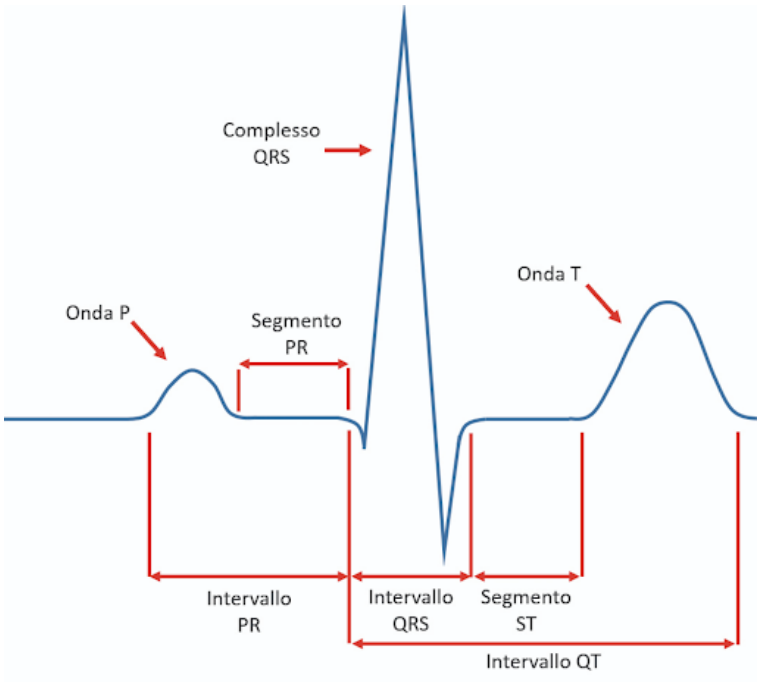



Fusione del battito ventricolare e del battito normale

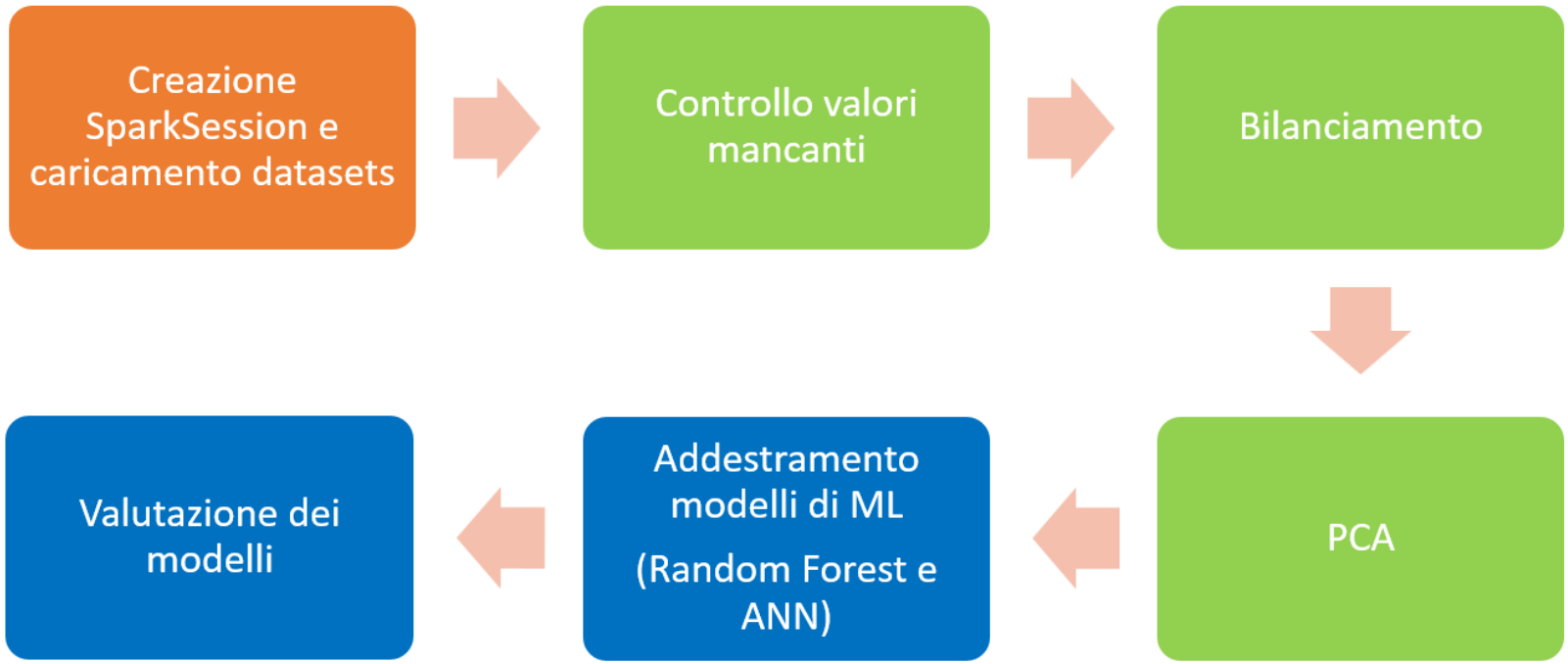
# Segnale ECG

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

- Le registrazioni mostrano le onde P, QRS e T.
- Sull'asse delle ascisse, il tempo è solitamente rappresentato in millisecondi (ms) o secondi (s).
- Il voltaggio è espresso in millivolt (mV).



# Workflow di creazione dei modelli di ML



# Configurazione della SparkSession

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

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

La configurazione della SparkSession è ottimizzata per gestire grandi quantità di dati e migliorare le prestazioni. Sono state impostate le seguenti configurazioni:

```
spark = SparkSession.builder \
    .appName("ECG Analyst") \
    .config("spark.driver.memory", "8g") \
    .config("spark.executor.memory", "6g") \
    .config("spark.driver.maxResultsize", "4g") \
    .config("spark.sql.debug.maxToStringFields", "1000") \
    .config("spark.sql.shuffle.partitions", "200") \
    .getOrCreate()
```

Per la gestione efficiente della persistenza dei dati, si è deciso di persistere i DataFrame utilizzando il livello di storage MEMORY_AND_DISK

# Caricamento del dataset

- Creazione di un Dataframe basandoci sul contenuto di un file CSV

```
train_df = spark.read.csv(train_path, header=False, inferSchema=True)
test_df = spark.read.csv(test_path, header=False, inferSchema=True)
```

- Conversione del Dataframe Spark in un Dataframe Pandas per applicare lo SMOTE sui dati di training

```
train_pd = train_df.toPandas()
test_pd = test_df.toPandas()
```

- Creazione di un DataFrame Spark dai dati di training concatenati utilizzando lo schema definito. Il ripartizionamento è utile per migliorare le prestazioni di elaborazione distribuita.

```
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).repartition(200)
```

# Pipeline Pre-Processing

Obiettivo: rendere i dati idonei per l'addestramento dei modelli di ML


In verde sono evidenziati gli effettivi stages usati nell'oggetto pipeline.

# Controllo dei Valori Mancanti

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

```
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())
```

```
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
```

# Bilanciamento dei Dati

Vista la grande disomogeneità del dataset è stato effettuato il bilanciamento seguendo due approcci:

- Undersampling
- Oversampling con SMOTE

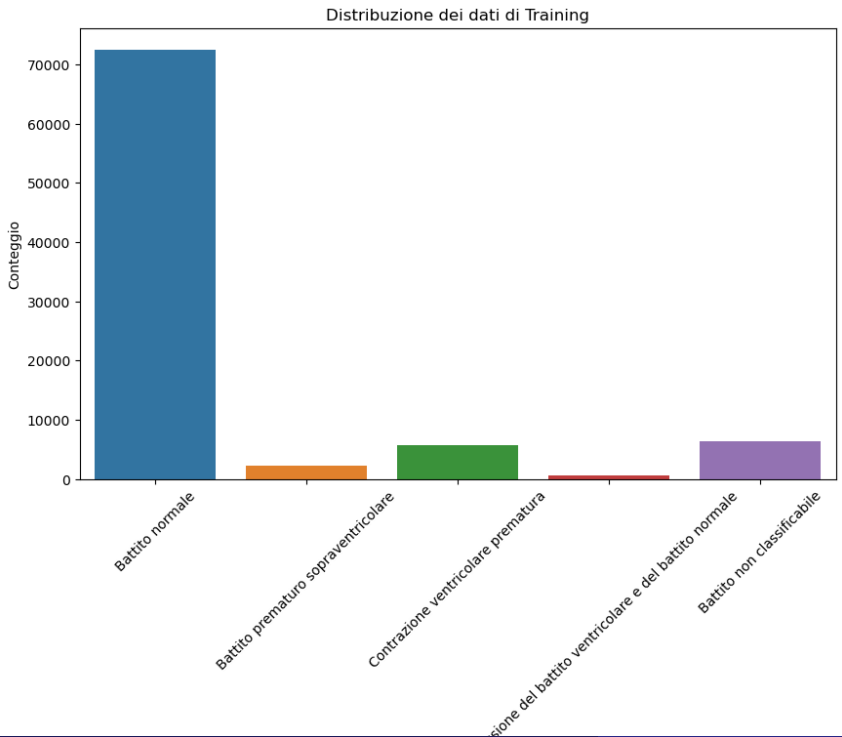
Un dataset di train non bilanciato non è idoneo all'addestramento di un modello di predizione per diversi motivi:

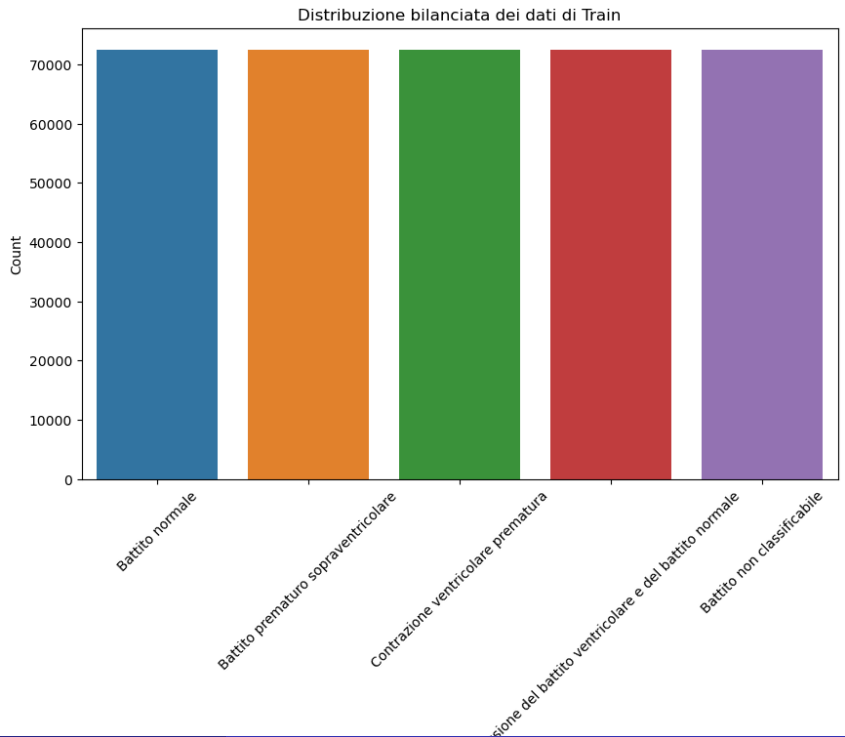
- I modelli tendono a favorire le classi maggioritarie e a trascurare le classi minoritarie.
- Problemi di generalizzazione del modello su nuovi dati che possono essere bilanciati diversamente rispetto al dataset di addestramento.
- La metrica di accuratezza può risultare ingannevolmente alta perché il modello potrebbe semplicemente imparare a predire sempre la classe maggioritaria.

# SMOTE

- Utilizzo della libreria imbalanced-learn in Python.
- Conversione dei dati di training e di test da DataFrame Spark a DataFrame Pandas per poter applicare SMOTE.

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



# Undersampling

```python
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
```

Questa funzione sottocampiona le classi maggioritarie calcolando il numero di istanze della classe con il minor numero di campioni.

# 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.

- 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.

# Implementazione PCA

L'implementazione di seguito riportata prevede l'utilizzo della libreria pyspark.ml.feature che fornisce le classi StandardScaler e PCA.

```
from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer, VectorAssembler, StandardScaler, PCA
```

A monte dell'applicazione della PCA si sono attuate le seguenti trasformazioni all'interno della pipeline:

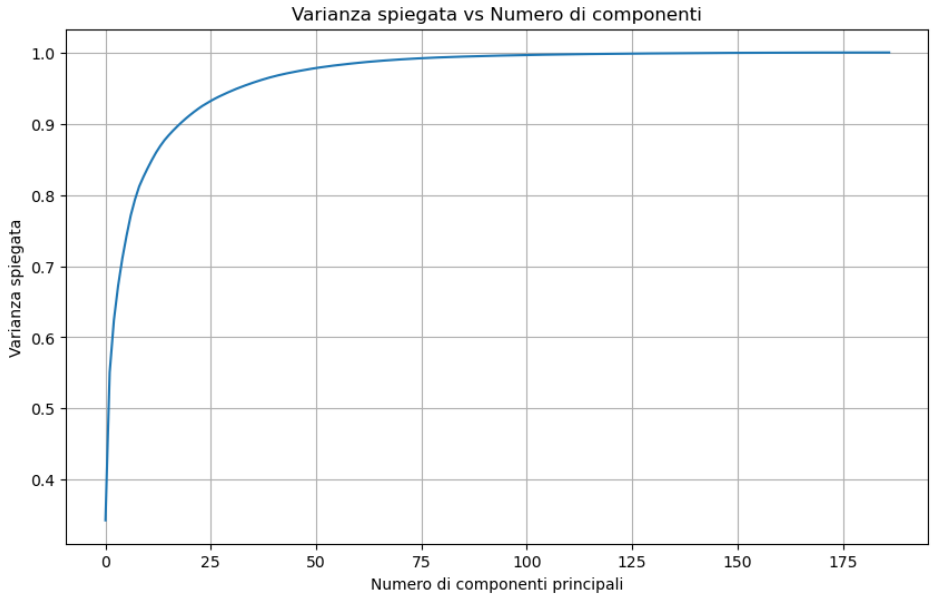
- `VectorAssembler` combina un insieme di colonne che rappresentano caratteristiche in un singolo vettore di caratteristiche.
- `StandardScaler` scala le caratteristiche in modo che abbiano media zero e deviazione standard unitaria.

```
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)
```

# Implementazione PCA

Il numero di componenti principali che spiegano il 95% di varianza totale risulta pari a 33

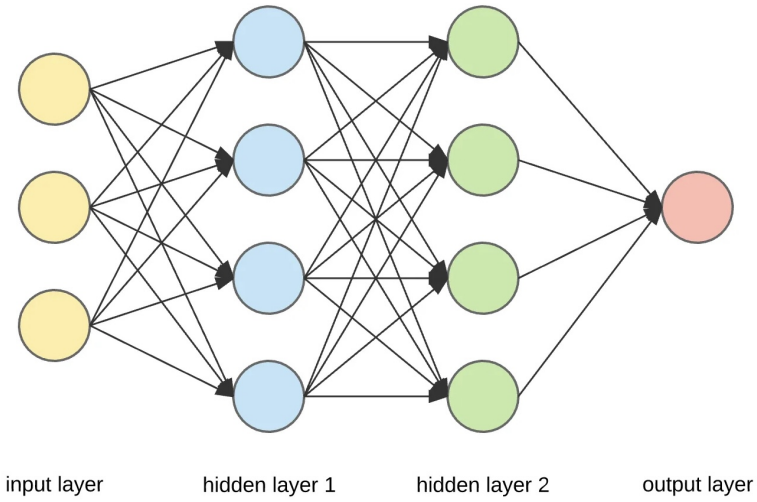



Si applica la trasformazione PCA ai dati di training e test generando "pcaFeatures" considerando n_components=33

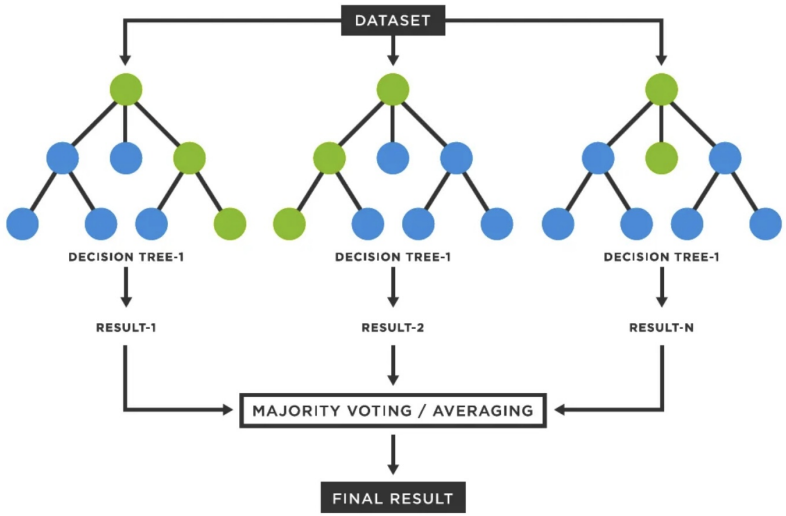
```
pca = PCA(k=n_components, inputCol="scaledFeatures", outputCol="pcaFeatures")
pca_model = pca.fit(train_df_scaled)
train_pca = pca_model.transform(train_df_scaled).select("pcaFeatures", "class")
test_pca = pca_model.transform(test_df_scaled).select("pcaFeatures", "class")
```

# Modelli 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.




Modello ANN




Modello Random Forest

```
from pyspark.ml.classification import MultilayerPerceptronClassifier
```

```
from pyspark.ml.classification import RandomForestClassifier
```

# Addestramento del Modello

Entrambi i modelli sono stato addestrati utilizzando la **cross-validation**.

```
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.ml.tuning import ParamGridBuilder, CrossValidator
```

Il modello di ML, la paramGrid e l'evaluator vengono utilizzati dall'oggetto `CrossValidator`, con 3 fold, per addestrare il modello sui dati di training per ogni combinazione di iperparametri

```
# Definizione layers per il modello di rete neurale
input_layer_size = train_df_pca.select("pcaFeatures").first()[0].size
layers1 = [input_layer_size, 70, 20, len(classes)]
layers2 = [input_layer_size, 10, 5, len(classes)]

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

# 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()
```

```
# 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())
```

Sul modello migliore trovato si effettua la predizione dui dati di test

# Valutazione dei Modelli

Attraverso la classe '`MulticlassMetrics`' è stato possibile valutare le prestazioni complessive del modello sui dati di test.

- Accuratezza: numero di previsioni corrette (TP e TN) diviso per il numero di campioni
- Precisione: numero di campioni previsti come positivi che sono effettivamente positivi
- Recall: numero di campioni positivi catturati dalla previsione positiva
- F1 Score: media armonica della precisione e del recall

# Risultati delle Predizioni

```
Metrics for class Normal beat (label 0):
 Accuracy: 0.9469
  Precision: 0.9869
  Recall: 0.9542
  F1 Score: 0.9703

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: 0.8210
  F1 Score: 0.6394

Metrics for class Unclassificable beat (label 4):
 Accuracy: 0.9469
  Precision: 0.9313
  Recall: 0.9689
  F1 Score: 0.9497
```

Modello ANN

```
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: 0.2072
  Recall: 0.8580
  F1 Score: 0.3337

Metrics for class Unclassificable beat (label 4):
 Accuracy: 0.9284
  Precision: 0.9452
  Recall: 0.9447
  F1 Score: 0.9449
```

Modello Random Forest

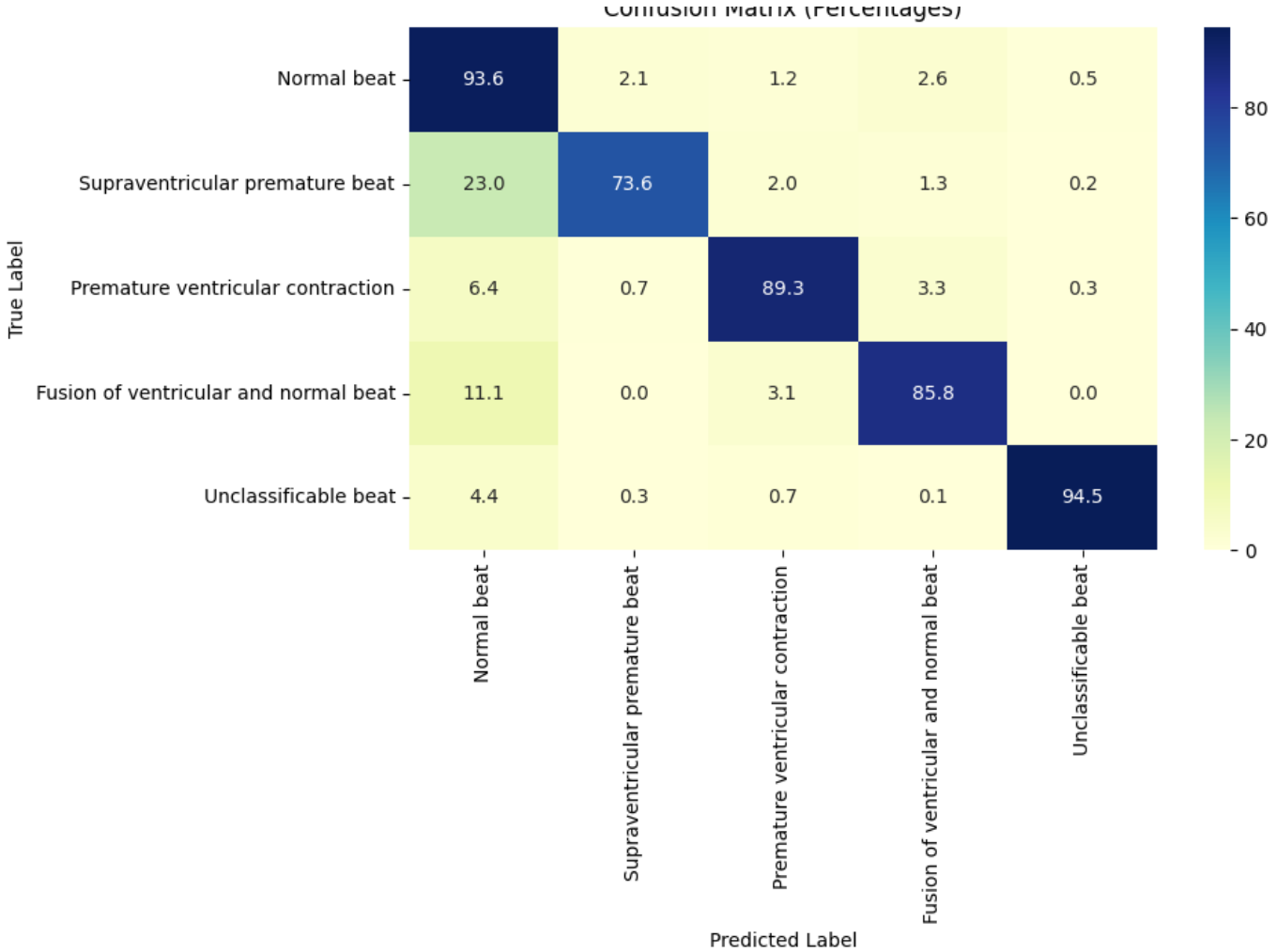
# Matrice di Confusione

La matrice di confusione è una tabella che descrive le prestazioni di un modello di classificazione.

- **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.

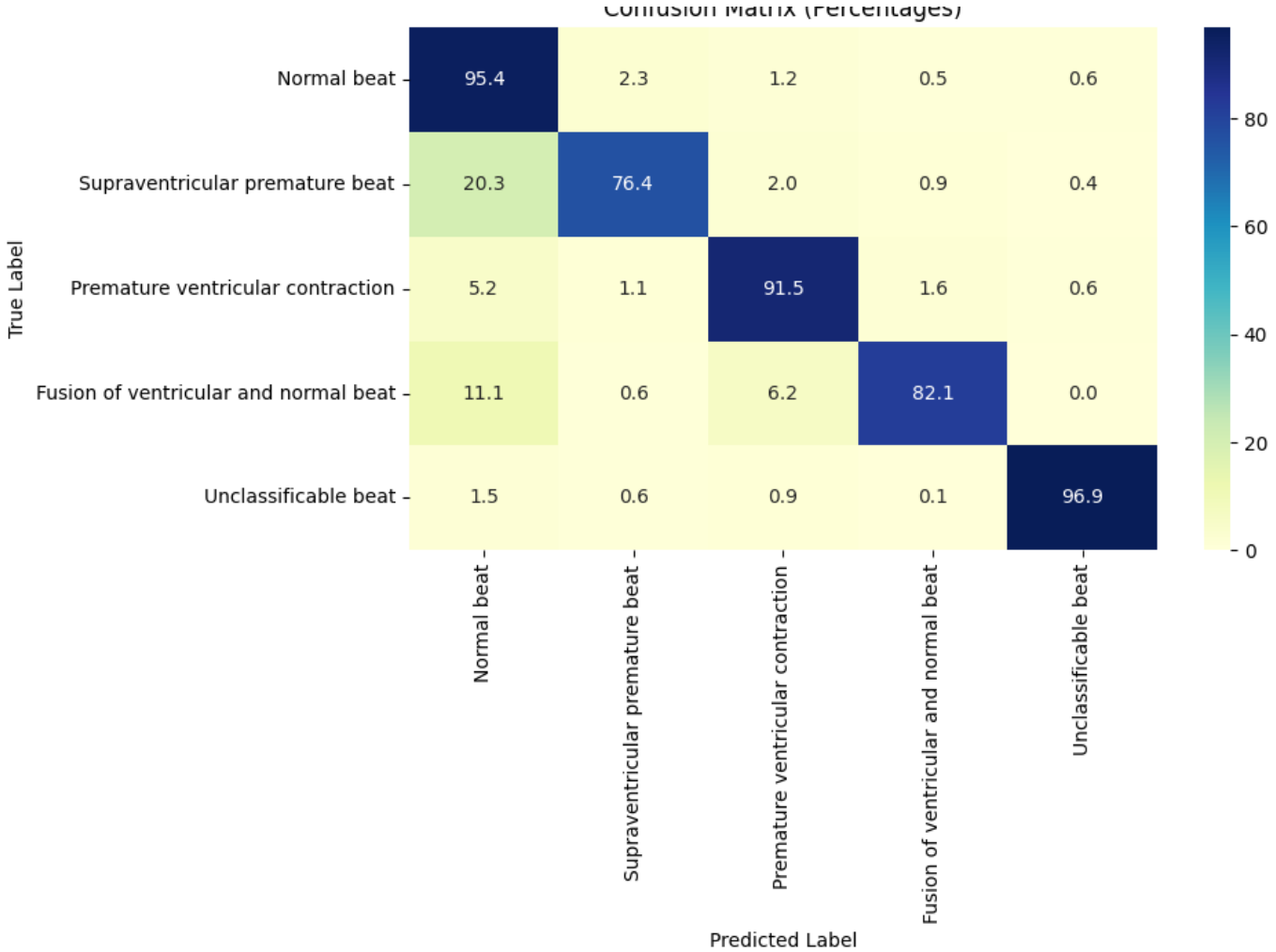
# Matrice di Confusione - Random Forest

Di seguito si riportano i risultati ottenuti per l'algoritmo di Random Forest:



# Matrice di Confusione - ANN

Di seguito si riportano i risultati ottenuti per la rete neurale artificiale:

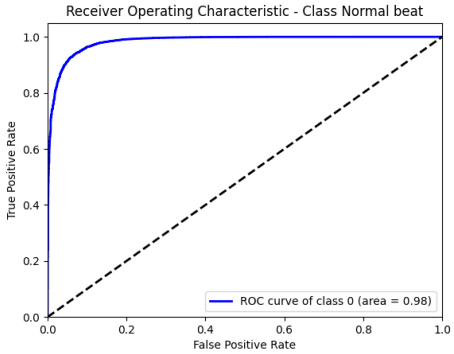


# Curve ROC

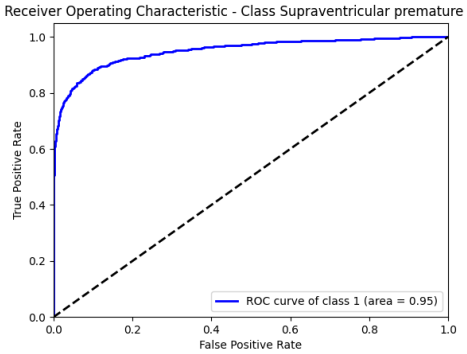
Le curve ROC (*Receiver Operating Characteristic*) sono uno strumento grafico utilizzato per valutare le prestazioni di un modello di classificazione binaria.

- Nel contesto della classificazione multiclasse, le curve ROC possono essere estese considerando l'**approccio "one-vs-all"**.
- L'area sotto la curva (**AUC** - *Area Under the Curve*) è un indicatore dell'accuratezza del modello.

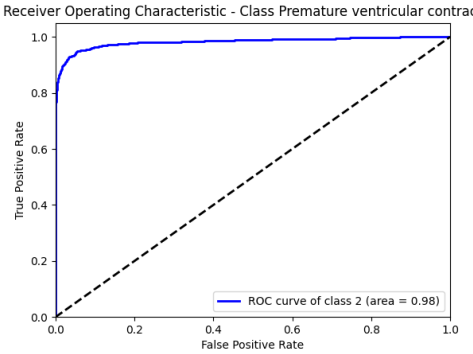
# Curve ROC - Random Forest



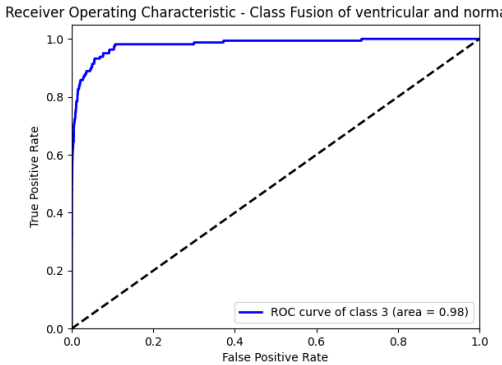

(a) Classe 0



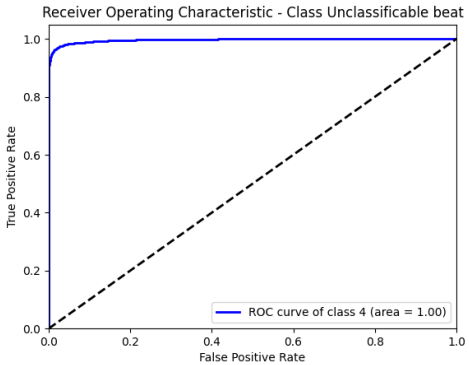

(b) Classe 1




(c) Classe 2

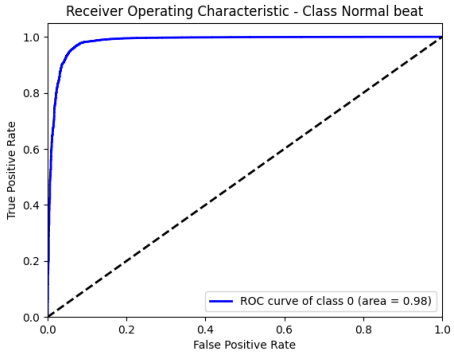



(a) Classe 3

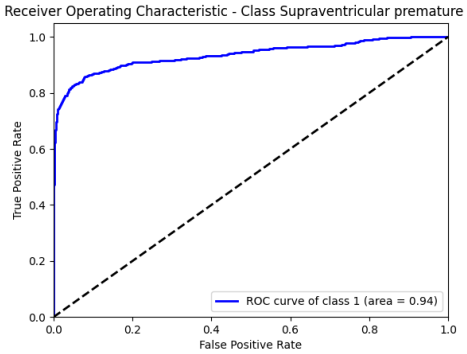



(b) Classe 4

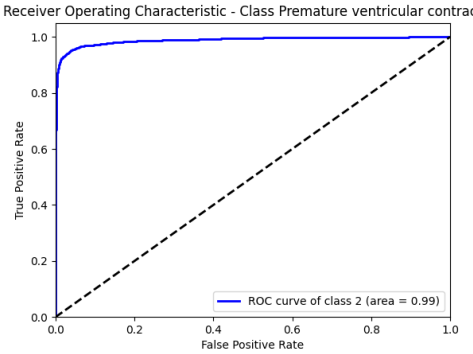
# Curve ROC - ANN



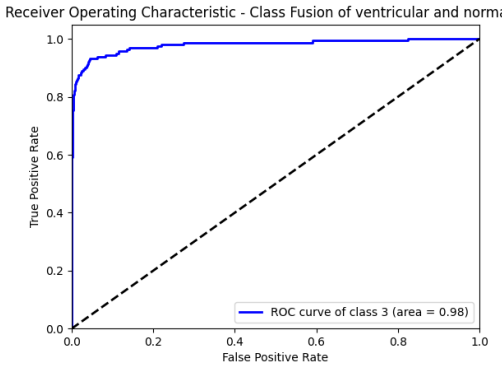

(a) Classe 0



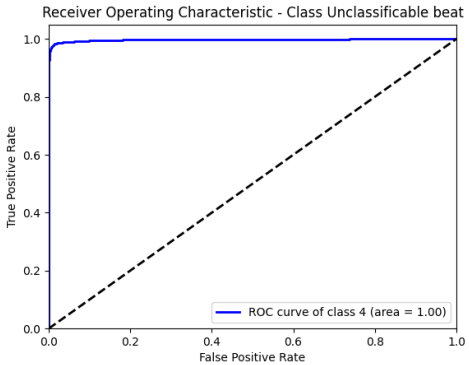

(b) Classe 1




(c) Classe 2




(a) Classe 3




(b) Classe 4

# Simulazione streaming

**Structured Streaming** è un motore di elaborazione dei flussi scalabile e tollerante ai guasti, basato su Spark SQL, che consente di esprimere computazioni di streaming come batch su dati statici.

- metodo `readStream()` → legge dati da una sorgente, per la quale si possono specificare formato, schema ed opzioni.
- metodo `writeStream()` → specifica la modalità di output, il formato e le opzioni.

# Simulazione streaming