Urinalysis Test Results

By Elka Segura

Anno Accademico 2023/2024

INTRODUZIONE

Il cervello umano è ciò che ispira l'architettura delle reti neurali. Le cellule del cervello umano, chiamate neuroni, formano una rete complessa e altamente interconnessa e si inviano segnali elettrici tra loro per aiutare gli esseri umani a elaborare le informazioni. Allo stesso modo, una rete neurale artificiale è costituita da neuroni artificiali che lavorano insieme per risolvere un problema. I neuroni artificiali sono moduli software, chiamati nodi, e le reti neurali artificiali sono programmi software o algoritmi che utilizzano essenzialmente sistemi informatici per risolvere calcoli matematici.

Le reti neurali sono costituite da neuroni, che a loro volta sono raggruppati in strati: ogni neurone di ogni strato è connesso a tutti i neuroni dello strato precedente. In ogni neurone verranno effettuate una serie di operazioni che, ottimizzando, faremo sì che la nostra rete impari.

Questo lavoro consiste nel costruire una rete neurale che preveda la classificazione del test UTIs, delle infezioni delle vie urinarie. Le infezioni del tratto urinario possono essere analizzate analizzando un tratto urinario. L'urina viene esaminata al microscopio per identificare batteri o globuli bianchi, che indicano un'infezione. Il medico può anche seguire un sistema urinario. Questo test esamina l'urina per raccogliere e identificare batteri e infezioni che possono causare infezioni del tratto urinario.

L'insieme dei dati che utilizziamo è stato raccolto da una clinica locale nel Mindanao settentrionale, nelle Filippine, ed è compreso tra aprile 2020 e gennaio 2023.

PRE-PROCESAMIENTO DEI DATI

Prima di costruire e addestrare una rete neurale, è fondamentale preparare adeguatamente i dati di input e di output. Eliminare i dati errati, duplicati o mancanti. Modificare i dati in modo che abbiano una scala simile per facilitare il processo di ottimizzazione. Convertire le variabili categoriali in un formato numerico adatto al modello.

Il set di dati è composto da 1436 righe × 16 colonne

Age (l'età del paziente) Nota: alcuni pazienti hanno mesi, quindi l'età di questi pazienti viene preelaborata dividendola per cento) ad esempio, 8 MESI, 8/100 = 0,08

Gender (Il sesso del paziente) Nota: maschio o femmina

Color (colore dell'urina)

Transparency (trasparenza delle urine)

Glucose (il glucosio è un tipo di zucchero e la sua presenza nelle urine può essere un indicatore importante di determinate condizioni di salute)

Protein (la presenza di proteine nelle urine è uno dei parametri esaminati per valutare la funzionalità renale e rilevarne le potenzialità)

pH (il livello del pH misura l'acidità o l'alcalinità delle urine)

Specific Gravity (la gravità specifica dell'urina è una misura della concentrazione di particelle nell'urina rispetto all'acqua)

WBC (White Blood Cells) Nota: noti anche come leucociti, i globuli bianchi sono una parte cruciale del sistema immunitario)

RBC (Red Blood Cells) Nota: i globuli rossi sono responsabili del trasporto dell'ossigeno in tutto il corpo

Epithelial Cells (le cellule epiteliali sono cellule che rivestono le superfici e le cavità del corpo, compreso il tratto urinario)

Mucous Threads (i fili mucosi sono filamenti di muco che possono essere presenti nelle urine)

Amorphous Urates (gli urati amorfi sono formazioni non cristalline nelle urine costituite da acido urico)

Bacteria (presenza di batteri nelle urine)

Diagnosis (UTI Diagnosis) NEGATIVE o POSITIVE

Il pre-processing dei dati si riferisce alla trasformazione dei dati grezzi in un formato più adatto e comprensibile per l'algoritmo. Questo processo include la gestione dei valori nulli, la normalizzazione delle variabili munerici, la codifica delle variabili categoriali e la gestione dei valori anomali. Per questo set di dati sono stati controllati i valori nulli e ne è stato ottenuto uno, per il quale è stata rimossa l'intera riga.

Con la funzione info() sono stati ottenuti i tipi di dati per ciascuna variabile nel set di dati. Abbiamo 3 variabili [Age, pH, Specific Gravity] che sono numeriche, mentre le restanti 12 variabili sono categoriali. [Gender, Transparency, Glucose, Protein, WBC, RBC, Epithelial Cells, Mucous Threads, Amorphous Urates, Bacteria, Diagnosis].

```
import pandas as pd
import numpy as np
archivo = 'C:/Users/2davi/OneDrive/Desktop/2 Anno/Statistica Per la Azienda(COZZUCOLI)/Progetto 2/urinalysis_tests.csv'
df = pd.read_csv(archivo, delimiter=',')
print(df.shape)
df.drop(df.columns[0:1], axis=1, inplace=True)
df = df.dropna()
df.info()
```

```
(1436, 16)
<class 'pandas.core.frame.DataFrame'>
Index: 1435 entries, 0 to 1435
Data columns (total 15 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   Age               1435 non-null   float64
 1   Gender            1435 non-null   object 
 2   Color             1435 non-null   object 
 3   Transparency      1435 non-null   object 
 4   Glucose           1435 non-null   object 
 5   Protein           1435 non-null   object 
 6   pH                1435 non-null   float64
 7   Specific Gravity  1435 non-null   float64
 8   WBC               1435 non-null   object 
 9   RBC               1435 non-null   object 
 10  Epithelial Cells  1435 non-null   object 
 11  Mucous Threads    1435 non-null   object 
 12  Amorphous Urates  1435 non-null   object 
 13  Bacteria          1435 non-null   object 
 14  Diagnosis         1435 non-null   object 
dtypes: float64(3), object(12)
memory usage: 179.4+ KB
```

```
df.head()
```

	Age	Gender	Color	Transparency	Glucose	Protein	pH	Specific Gravity	WBC	RBC	Epithelial Cells	Mucous Threads	Amorphous Urates	Bacteria	Diagnosis
0	76.0	FEMALE	LIGHT YELLOW	CLEAR	NEGATIVE	NEGATIVE	5.0	1.010	1-3	0-2	OCCASIONAL	RARE	NONE SEEN	OCCASIONAL	NEGATIVE
1	9.0	MALE	DARK YELLOW	SLIGHTLY HAZY	NEGATIVE	1+	5.0	1.030	1-3	0-2	RARE	FEW	FEW	MODERATE	NEGATIVE
2	12.0	MALE	LIGHT YELLOW	SLIGHTLY HAZY	NEGATIVE	TRACE	5.0	1.030	0-3	0-2	RARE	FEW	MODERATE	RARE	NEGATIVE
3	77.0	MALE	BROWN	CLOUDY	NEGATIVE	1+	6.0	1.020	5-8	LOADED	RARE	RARE	NONE SEEN	FEW	NEGATIVE
4	29.0	FEMALE	YELLOW	HAZY	NEGATIVE	TRACE	6.0	1.025	1-4	0-2	RARE	RARE	NONE SEEN	FEW	NEGATIVE

Con la funzione describe() abbiamo ottenuto il minimo, il massimo, la media e altri valori delle variabili numeriche. Come avevamo accennato in precedenza, dobbiamo standardizzare i dati che saranno input nella nostra rete neurale.

```
df.describe()
```

	Age	pH	Specific Gravity
count	1435.000000	1435.000000	1435.000000
mean	27.213937	6.052962	1.015847
std	23.466950	0.598682	0.007287
min	0.010000	5.000000	1.005000
25%	6.000000	6.000000	1.010000
50%	23.000000	6.000000	1.015000
75%	45.000000	6.500000	1.020000
max	92.000000	8.000000	1.030000

Per le variabili numeriche useremo normalizzazione min-max. Questa tecnica ridimensiona i valori dei dati in modo che rientrino in un intervallo specifico, ad esempio tra 0 e 1. Ciò si ottiene utilizzando la formula: X_norma = (X - X_min) / (X_max - X_min) Dove X è il valore originale, X_min è il valore minimo e X_max è il valore massimo. La normalizzazione min-max è utile quando i dati hanno scale diverse e si desidera confrontare o mettere in relazione le variabili tra loro. Questa tecnica garantisce che tutti i dati siano nello stesso intervallo e impedisce che le variabili con valori più grandi dominino il modello.

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
df['Age'] = scaler.fit_transform(df[['Age']])
df['pH'] = scaler.fit_transform(df[['pH']])
df['Specific Gravity'] = scaler.fit_transform(df[['Specific Gravity']])
df.describe()
```

	Age	pH	Specific Gravity
count	1435.000000	1435.000000	1435.000000
mean	0.295727	0.350987	0.433868
std	0.255103	0.199561	0.291481
min	0.000000	0.000000	0.000000
25%	0.065116	0.333333	0.200000
50%	0.249918	0.333333	0.400000
75%	0.489075	0.500000	0.600000
max	1.000000	1.000000	1.000000

Per trattare le variabili categoriali utilizzeremo il metodo di codifica chiamato One Hot Encoding. Questa strategia consiste nel creare una colonna binaria (che può contenere solo i valori 0 o 1) per ogni valore univoco esistente nella variabile categoriale che stiamo codificando, e contrassegnare con un 1 la colonna corrispondente al valore presente in ciascuna record, lasciando le altre colonne con un valore pari a 0. Ma prima di convertire le variabili categoriali in dummy, verranno apportate modifiche pertinenti con l'obiettivo di ridurre al massimo le categorie possibili per ciascuna variabile. Uno svantaggio di questo metodo è che stiamo aumentando la dimensionalità del set di dati (ovvero, aumentando il numero di colonne o caratteristiche categoriche da cui addestrare il modello), il che può essere problematico se il numero di campioni disponibili non è sufficientemente elevato per addestrare la rete. Per la variabile Color, abbiamo unito le categorie ['REDDISH', 'REDDISH YELLOW', 'LIGHT RED', 'BROWN'] nella categoria RED che in questo caso abbiamo una frequenza bassa per ogni categoria. Successivamente descriviamo le altre variabili con ciascuno dei loro livelli.

```
df['Color'] = df['Color'].replace(('REDDISH', 'REDDISH YELLOW', 'LIGHT RED', 'BROWN'), 'RED')
df.Color.value_counts()
```

```
Color
YELLOW          710
LIGHT YELLOW    341
DARK YELLOW     248
STRAW           116
AMBER            15
RED               5
Name: count, dtype: int64
```

```
df.Transparency.value_counts()
```

```
Transparency
CLEAR           1123
SLIGHTLY HAZY    172
HAZY             104
TURBID            20
CLOUDY            16
Name: count, dtype: int64
```

```
df.Glucose.value_counts()
```

```
Glucose
NEGATIVE    1348
2+            24
3+            23
1+            15
TRACE         13
4+            12
Name: count, dtype: int64
```

```
df.Protein.value_counts()
```

```
Protein
NEGATIVE    804
TRACE       491
1+           94
2+           41
3+            5
Name: count, dtype: int64
```

```
df['pH'].value_counts()
```

```
pH
0.333333    758
0.500000    305
0.000000    237
0.666667     96
0.833333     27
1.000000     12
Name: count, dtype: int64
```

Le variabili WBC e RBC si riferiscono rispettivamente alla conta dei globuli bianchi e rossi del paziente. Queste variabili come set singolo hanno 91 tipi di categorie che si riflettono in intervalli. [0-2, 1-3...]. La procedura adottata dall'autore originario del lavoro prevede la riduzione di 91 categorie a 13 divisioni che contengono parzialmente insiemi delle categorie originarie. Per ridurre il numero di categorie, la nostra proposta è di ridurre a 9 categorie I valori di queste colonne che sono indicati da un segno maggiore (">") come ">50" e ">100", indicando che il numero di leucociti o eritrociti supera 50 o 100 e, infine, alcuni punti di dati rappresentati come stringhe, inclusi "LOADED" e "TNTC" saranno rappresentato >250 che significa che sono non numerabile.

```
df.WBC.value_counts()
```

```
WBC
0-2      448
1-2      150
0-1      136
2-4      114
1-3      106
        ...
10-16      1
48-55      1
7-8        1
15-22      1
8-11       1
Name: count, Length: 75, dtype: int64
```

```
df.RBC.value_counts()
```

```
RBC
0-2       617
0-1       312
1-2       123
1-3        87
2-4        72
0-3        49
3-5        27
4-6        24
>50        13
5-7        10
>100       10
8-10       10
```

La seguente funzione crea una nuova colonna numerica in cui mi dà il valore medio per ogni soggetto. Per non perdere l'ordine della variabile viene effettuato un trattamento che passa

il valore originale, che è una stringa, alla funzione e ne restituisce il valore medio. Che un soggetto abbia una conta di globuli bianchi o rossi compresa tra [3-5], corrisponde ad un valore medio di 4. Successivamente si effettua la divisione dei valori medi in 9 intervalli.

```
from typing import Union
def custom_sort(value: str) -> Union[int, float]:
    APPROXIMATE_COUNT = 250
    if ">" in value:
        return int(value.replace(">", ""))
    elif value.isalpha():
        return APPROXIMATE_COUNT
    else:
        start, end = map(int, value.split('-'))
        return start + (end - start) / 2



sorted_values = [custom_sort(val) for val in df['WBC']]
sorted_values1 = [custom_sort(val1) for val1 in df['RBC']]

# Convertir la lista en una Serie de Pandas
sorted_series = pd.Series(sorted_values, name='Sorted_WBC')
sorted_series1 = pd.Series(sorted_values1, name='Sorted_RBC')
# Concatenar la nueva Serie al DataFrame
df = pd.concat([df, sorted_series, sorted_series1], axis=1)
df.isna().sum()
df= df.dropna()
```

Per apportare le modifiche pertinenti alle variabili WBC e RBC, viene applicata una funzione che restituisce ciascun valore medio di ciascun intervallo (91 categorie). Da questo valore creeremo le 9 nuove categorie che rappresenteranno i seguenti intervalli [0_5, 6_10, 11_15, 16_20, 21_30, 31_40, 41_50, >50, >100, >250].

```python
# Definir los intervalos de los bins
bins = [0, 5, 10, 15, 20, 30, 50, 100, 249, 300]

# Etiquetas para los bins
labels = ['0_5', '6_10', '11_15', '16_20', '21_30', '31_50', '>50', '>100', '>250']

# Aplicar la función cut para dividir la variable en bins
df['WBC_S'] = pd.cut(df['Sorted_WBC'], bins=bins, labels=labels, right=False)
df['RBC_S'] = pd.cut(df['Sorted_RBC'], bins=bins, labels=labels, right=False)
```

```python
df['WBC_S'].value_counts()
```

```
WBC_S
0_5      1110
6_10      153
11_15      54
16_20      35
>250       26
>50        23
21_30      21
31_50       8
>100        4
Name: count, dtype: int64
```

```python
df['Epithelial Cells'].value_counts()
```

```
Epithelial Cells
RARE           740
FEW            347
MODERATE       188
PLENTY         121
OCCASIONAL      19
NONE SEEN       16
LOADED           3
Name: count, dtype: int64
```

```
df['Mucous Threads'].value_counts()
```

```
Mucous Threads
NONE SEEN     500
FEW           381
RARE          275
MODERATE      221
PLENTY         36
OCCASIONAL     21
Name: count, dtype: int64
```

```
df['Amorphous Urates'].value_counts()
```

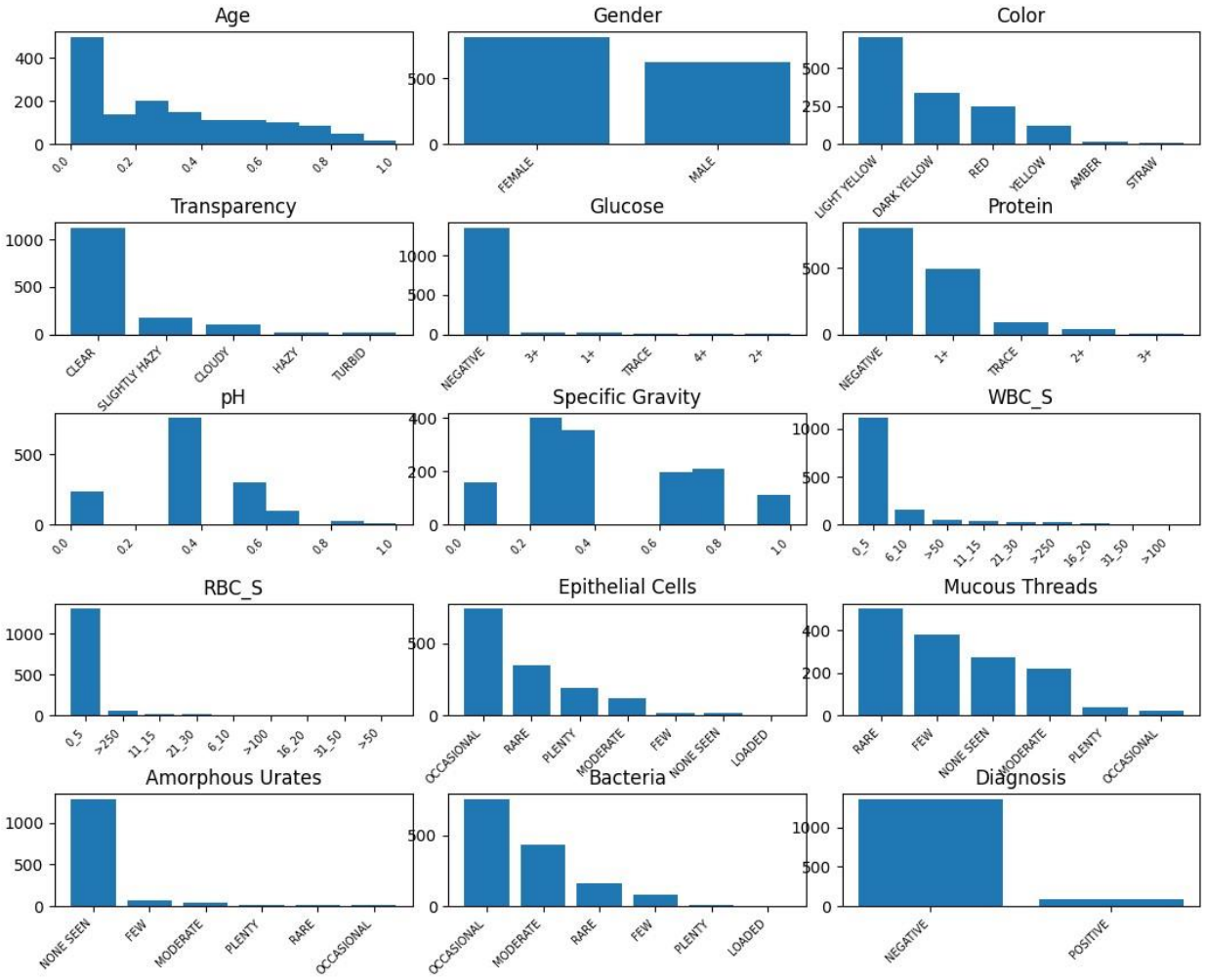
```
Amorphous Urates
NONE SEEN     1282
FEW             72
RARE            42
MODERATE        19
PLENTY          11
OCCASIONAL       8
Name: count, dtype: int64
```

```
df.Bacteria.value_counts()
```

```
Bacteria
RARE          753
FEW           434
MODERATE      158
PLENTY         77
OCCASIONAL      8
LOADED          4
Name: count, dtype: int64
```

```
df.Diagnosis.value_counts()
```

```
Diagnosis
NEGATIVE    1353
POSITIVE      81
Name: count, dtype: int64
```



Come avevamo detto in precedenza, le variabili categoriali vengono trasformate in dummy. Definiamo una piccola funzione che restituisce la conversione delle variabili categoriali in variabili binarie, questo con l'obiettivo di ottenere le variabili trasformate che serviranno come input alla rete neurale.

```
<class 'pandas.core.frame.DataFrame'>
Index: 1434 entries, 0 to 1434
Data columns (total 71 columns):
 #   Column                        Non-Null Count  Dtype
---  ------                        --------------  -----
 0   Age                           1434 non-null   float64
 1   pH                            1434 non-null   float64
 2   Specific Gravity              1434 non-null   float64
 3   Gender_FEMALE                 1434 non-null   int32
 4   Gender_MALE                   1434 non-null   int32
 5   Color_LIGHT YELLOW            1434 non-null   int32
 6   Color_DARK YELLOW             1434 non-null   int32
 7   Color_RED                     1434 non-null   int32
 8   Color_YELLOW                  1434 non-null   int32
 9   Color_AMBER                   1434 non-null   int32
 10  Color_STRAW                   1434 non-null   int32
 11  Transparency_CLEAR            1434 non-null   int32
 12  Transparency_SLIGHTLY HAZY    1434 non-null   int32
 13  Transparency_CLOUDY           1434 non-null   int32
 14  Transparency_HAZY             1434 non-null   int32
 15  Transparency_TURBID           1434 non-null   int32
 16  Glucose_NEGATIVE              1434 non-null   int32
 17  Glucose_3+                    1434 non-null   int32
 18  Glucose_1+                    1434 non-null   int32
 19  Glucose_TRACE                 1434 non-null   int32
 20  Glucose_4+                    1434 non-null   int32
 21  Glucose_2+                    1434 non-null   int32
 22  Protein_NEGATIVE              1434 non-null   int32
 23  Protein_1+                    1434 non-null   int32
 24  Protein_TRACE                 1434 non-null   int32
 25  Protein_2+                    1434 non-null   int32
 26  Protein_3+                    1434 non-null   int32
 27  WBC_S_0_5                     1434 non-null   int32
 28  WBC_S_6_10                    1434 non-null   int32
 29  WBC_S_>50                     1434 non-null   int32
 30  WBC_S_11_15                   1434 non-null   int32
 31  WBC_S_21_30                   1434 non-null   int32
 32  WBC_S_>250                    1434 non-null   int32
 33  WBC_S_16_20                   1434 non-null   int32
 34  WBC_S_31_50                   1434 non-null   int32
 35  WBC_S_>100                    1434 non-null   int32
```

```
 36  RBC_S_0_5                    1434 non-null   int32
 37  RBC_S_>250                   1434 non-null   int32
 38  RBC_S_11_15                  1434 non-null   int32
 39  RBC_S_21_30                  1434 non-null   int32
 40  RBC_S_6_10                   1434 non-null   int32
 41  RBC_S_>100                   1434 non-null   int32
 42  RBC_S_16_20                  1434 non-null   int32
 43  RBC_S_31_50                  1434 non-null   int32
 44  RBC_S_>50                    1434 non-null   int32
 45  Epithelial Cells_OCCASIONAL  1434 non-null   int32
 46  Epithelial Cells_RARE        1434 non-null   int32
 47  Epithelial Cells_PLENTY      1434 non-null   int32
 48  Epithelial Cells_MODERATE    1434 non-null   int32
 49  Epithelial Cells_FEW         1434 non-null   int32
 50  Epithelial Cells_NONE SEEN   1434 non-null   int32
 51  Epithelial Cells_LOADED      1434 non-null   int32
 52  Mucous Threads_RARE          1434 non-null   int32
 53  Mucous Threads_FEW           1434 non-null   int32
 54  Mucous Threads_NONE SEEN     1434 non-null   int32
 55  Mucous Threads_MODERATE      1434 non-null   int32
 56  Mucous Threads_PLENTY        1434 non-null   int32
 57  Mucous Threads_OCCASIONAL    1434 non-null   int32
 58  Amorphous Urates_NONE SEEN   1434 non-null   int32
 59  Amorphous Urates_FEW         1434 non-null   int32
 60  Amorphous Urates_MODERATE    1434 non-null   int32
 61  Amorphous Urates_PLENTY      1434 non-null   int32
 62  Amorphous Urates_RARE        1434 non-null   int32
 63  Amorphous Urates_OCCASIONAL  1434 non-null   int32
 64  Bacteria_OCCASIONAL          1434 non-null   int32
 65  Bacteria_MODERATE            1434 non-null   int32
 66  Bacteria_RARE                1434 non-null   int32
 67  Bacteria_FEW                 1434 non-null   int32
 68  Bacteria_PLENTY              1434 non-null   int32
 69  Bacteria_LOADED              1434 non-null   int32
 70  Diagnosi                     1434 non-null   int64
dtypes: float64(3), int32(67), int64(1)
memory usage: 431.3 KB
```

COSTRUZIONE DELLA RETE NEURALE

Dopo che il dataset vengono preparati, i dati vengono quindi generalmente suddivisi in set di training, validation e test; In questo caso dividiamo il dataset solo in train_set e test_set perche ci ritroviamo con un data set sbilanciato e poche unità che rappresentano la classe positiva. Viene definita l'architettura della rete neurale, compreso il numero di strati, il numero di neuroni in ogni strato, le funzioni di attivazione, in questo caso utilizziamo il RELU per gli strati intermedi e per l'output utilizziamo il Sigmoide. Questo viene fatto utilizzando

le librerie TensorFlow e Keras. La funzione di attivazione ReLu applica una trasformazione non lineare molto semplice, attivando il neurone solo se l'input è superiore a zero. Mentre il valore di ingresso è inferiore a zero, il valore di uscita è zero, ma quando è superiore a zero, il valore di uscita aumenta linearmente con il valore di ingresso. La funzione sigmoide trasforma i valori nell'intervallo (-inf, +inf) in valori nell'intervallo (0, 1). Questa è ancora la funzione spesso utilizzata nei neuroni nello strato di output dei modelli di classificazione binaria, poiché il suo output può essere interpretato come probabilità.

Il processo di addestramento di una rete neurale consiste nell'aggiustare il valore dei pesi e dei bias in modo tale che le previsioni generate abbiano il minor errore possibile. Grazie a ciò, il modello è in grado di identificare quali predittori hanno la maggiore influenza e come sono correlati tra loro e con la variabile di risposta. Avvia la rete con valori casuali dei pesi e del bias. Per ogni osservazione di addestramento (X, y), calcola l'errore commesso dalla rete quando effettua le sue previsioni. Media degli errori di tutte le osservazioni. Identificare la responsabilità che ciascun peso e bias ha avuto nell'errore di previsione. Modificare leggermente i pesi e il bias della rete (proporzionali alla sua responsabilità per l'errore) nella giusta direzione in modo da ridurre l'errore. Ripeti i passaggi precedenti finché la rete è abbastanza buona. Tutto questo attraverso della combinazione di più metodi matematici, in particolare l'algoritmo di backpropagation e l'ottimizzazione della discesa del gradiente.

Il codice seguente importa diverse classi e funzioni dalle librerie tensorflow.keras e sklearn.model_selection che andiamo a utilizzare.

- Da ***tensorflow.keras.models*** import ***Sequential***: importa la classe Sequential da tensorflow.keras.models. Sequenziale è un modello di rete neurale sequenziale, il che significa che i livelli vengono aggiunti in sequenza uno dopo l'altro.
- Da ***tensorflow.keras.layers*** import ***Dense***, Input, Concatenate, Embedding: importa diversi livelli che possono essere utilizzati per creare modelli di rete neurale con tensorflow.keras. I layer Denso costruisce uno strato di rete neurale completamente connesso e i layer Input costruisce un livello di input utilizzato per specificare la dimensione dei dati di input.
- Da ***sklearn.model_selection*** import ***train_test_split***: importa la funzione train_test_split da sklearn.model_selection. train_test_split viene utilizzato per suddividere i dati in set di training e test, utile per valutare le prestazioni del modello.

In sintesi, questo codice prepara l'ambiente per la creazione e l'addestramento di modelli di rete neurale utilizzando TensorFlow e Keras e anche per la suddivisione dei set di dati in set di training e test utilizzando scikit-learn.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Input, Concatenate, Embedding
from sklearn.model_selection import train_test_split
```

Selezioniamo tutte le colonne di un DataFrame df_dummy elaborato, tranne la colonna 'Diagnosi', che è la nostra variabile target, che sarà l'input della rete neurale che salviamo nella variabile ***X***. Quindi selezioniamo solo la colonna 'Diagnosi' e lo salviamo nella variabile ***y***.

```
X = df_dummy.drop('Diagnosi', axis=1)
y = df_dummy['Diagnosi']
```

Utilizziamo quindi la funzione train_test_split di scikit-learn per dividere i set di dati X e y in set di training e test. Ecco una descrizione di ciò che fa ciascun argomento nella funzione train_test_split:

- ***X***: Insieme di caratteristiche/ variabili indipendenti.
- ***y***: set di etichette (valore della variabile target).
- test_size: proporzione del set di dati da utilizzare come set di test. In questo caso è impostato su 0,3, il che significa che il 30% del set di dati verrà utilizzato come set di test.
- ***random_state***: seme per garantire la riproducibilità della suddivisione. Impostandolo ti assicuri di ottenere la stessa suddivisione ogni volta che esegui il codice.

Il risultato della funzione train_test_split è costituito da quattro set di dati: X_train, X_test, y_train e y_test. Questi rappresentano i dati delle funzionalità e delle etichette rispettivamente per il set di training e il set di test.

```
# División de datos
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

Utilizziamo un modello sequenziale, in cui sono definiti i seguenti attributi: per produrre un output compreso tra 0 e 1, che viene interpretato come la probabilità che l'input appartenga alla classe positiva.

- ***Input(shape=(X_train.shape[1],)):*** Definisce il livello di input del modello, che in questo caso rappresenta il numero di feature nel variabili indipendenti ottenute 70 salvate nel set di addestramento X_train. Ciò consente alla rete neurale di accettare dati di input con la stessa forma dei dati di addestramento. Ora attiviamo due livelli nascosti interconnessi:
- ***Dense(64, attivazione='relu')***: aggiungiamo uno strato densamente connesso con 64 neuroni alla rete neurale. All'uscita di questo livello viene applicata la funzione di

attivazione 'relu' (unità lineare rettificata). Lo strato denso è completamente connesso ai neuroni dello strato precedente.

- ***Dense(32, attivazione='relu'):*** aggiungiamo un altro strato densamente connesso con 32 neuroni alla rete neurale. Come il livello precedente, utilizza la funzione di attivazione "relu". Per gli output si attiva un nodo:
- ***Dense(1, attivazione='sigmoid'):*** aggiungiamo lo strato di output della rete neurale. Questo è uno strato denso con un singolo neurone, poiché il problema è di classificazione binaria (l'output è 0 o 1). La funzione di attivazione "sigmoide" viene qui utilizzata.

```
# Definición del modelo
model = Sequential([
    Input(shape=(X_train.shape[1],)),
    Dense(64, activation='relu'),
    Dense(32, activation='relu'),
    Dense(1, activation='sigmoid')
])
```

Il modello viene compilato utilizzando l'ottimizzatore "rmsprop". Il RMSprop (Root Mean Square Propagation) è un algoritmo di ottimizzazione comunemente utilizzato nell'allenamento delle reti neuronali. A differenza della tradizionale discesa del gradiente Stocastico, RMSprop adatta il compito di apprendimento a ciascun parametro individualmente, in modo da aiutare a convergere più rapidamente su problemi con gradienti di diversa natura. Anche se RMSprop non richiede un compito di apprendimento esplicito, è anche possibile controllarlo indirettamente tramite la regolazione di altri iperparametri. Uno di questi iperparametri è il "learning_rate" (learning_rate) che influenza l'entità delle regolazioni del peso durante . Il "tasso di apprendimento" iniziale predefinito nell'ottimizzatore è 0.001. Il tasso di apprendimento o learning_rate stabilisce la velocità con cui i parametri di un modello possono cambiare man mano che viene ottimizzato (appreso). Questo iperparametro è uno dei più complicati da stabilire, poiché dipende fortemente dai dati e interagisce con il resto degli iperparametri. Se il tasso di apprendimento è molto elevato, il processo di ottimizzazione può passare da una regione all'altra senza che il modello sia in grado di apprendere. Se, invece, il tasso di apprendimento è molto basso, il processo formativo potrebbe richiedere troppo tempo e non essere completato. La funzione di perdita (loss function) è specificata come "binary_crossentropy", che viene comunemente utilizzata per problemi di classificazione binaria. La metrica di valutazione è specificata come "accuratezza", che calcolerà l'accuratezza del modello durante l'addestramento e la valutazione. Il modello viene addestrato utilizzando i dati di addestramento. Valutiamo il modello utilizzando i dati di test X_test e y_test. La perdita e l'accuratezza del modello sui dati del test vengono calcolate e visualizzate. Le previsioni vengono effettuate utilizzando il modello addestrato sui dati del test X_test.

```python
# Compilación del modelo
model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['accuracy'])

# Entrenamiento del modelo
history = model.fit(X_train, y_train, epochs=50, batch_size=32, validation_data=(X_test, y_test))

# Evaluación del modelo
loss, accuracy = model.evaluate(X_test, y_test)
print(f"Loss: {loss}, Accuracy: {accuracy}")

# Predicciones
predictions = model.predict(X_test)
```

```
Epoch 1/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 1s 12ms/step - accuracy: 0.8872 - loss: 0.3887 - val_accuracy: 0.9304 - val_loss: 0.2450
Epoch 2/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9428 - loss: 0.1869 - val_accuracy: 0.9304 - val_loss: 0.2312
Epoch 3/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9529 - loss: 0.1479 - val_accuracy: 0.9304 - val_loss: 0.2201
Epoch 4/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9509 - loss: 0.1617 - val_accuracy: 0.9281 - val_loss: 0.2101
Epoch 5/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9664 - loss: 0.1072 - val_accuracy: 0.9304 - val_loss: 0.2029
Epoch 6/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9639 - loss: 0.1295 - val_accuracy: 0.9327 - val_loss: 0.2110
Epoch 7/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9652 - loss: 0.1155 - val_accuracy: 0.9304 - val_loss: 0.2137
Epoch 8/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9690 - loss: 0.1294 - val_accuracy: 0.9327 - val_loss: 0.2174
Epoch 9/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9696 - loss: 0.0958 - val_accuracy: 0.9327 - val_loss: 0.1991
Epoch 10/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9774 - loss: 0.0928 - val_accuracy: 0.9327 - val_loss: 0.2056
Epoch 11/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9789 - loss: 0.0936 - val_accuracy: 0.9397 - val_loss: 0.1965
Epoch 12/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9779 - loss: 0.0902 - val_accuracy: 0.9420 - val_loss: 0.1940
Epoch 13/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9680 - loss: 0.1157 - val_accuracy: 0.9327 - val_loss: 0.2141
Epoch 14/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9731 - loss: 0.1067 - val_accuracy: 0.9443 - val_loss: 0.1961
Epoch 15/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9739 - loss: 0.0845 - val_accuracy: 0.9443 - val_loss: 0.1988
Epoch 16/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9862 - loss: 0.0674 - val_accuracy: 0.9420 - val_loss: 0.2077
Epoch 17/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9817 - loss: 0.0823 - val_accuracy: 0.9466 - val_loss: 0.2054
Epoch 18/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9828 - loss: 0.0750 - val_accuracy: 0.9374 - val_loss: 0.2162
Epoch 19/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9775 - loss: 0.0800 - val_accuracy: 0.9443 - val_loss: 0.2022
```

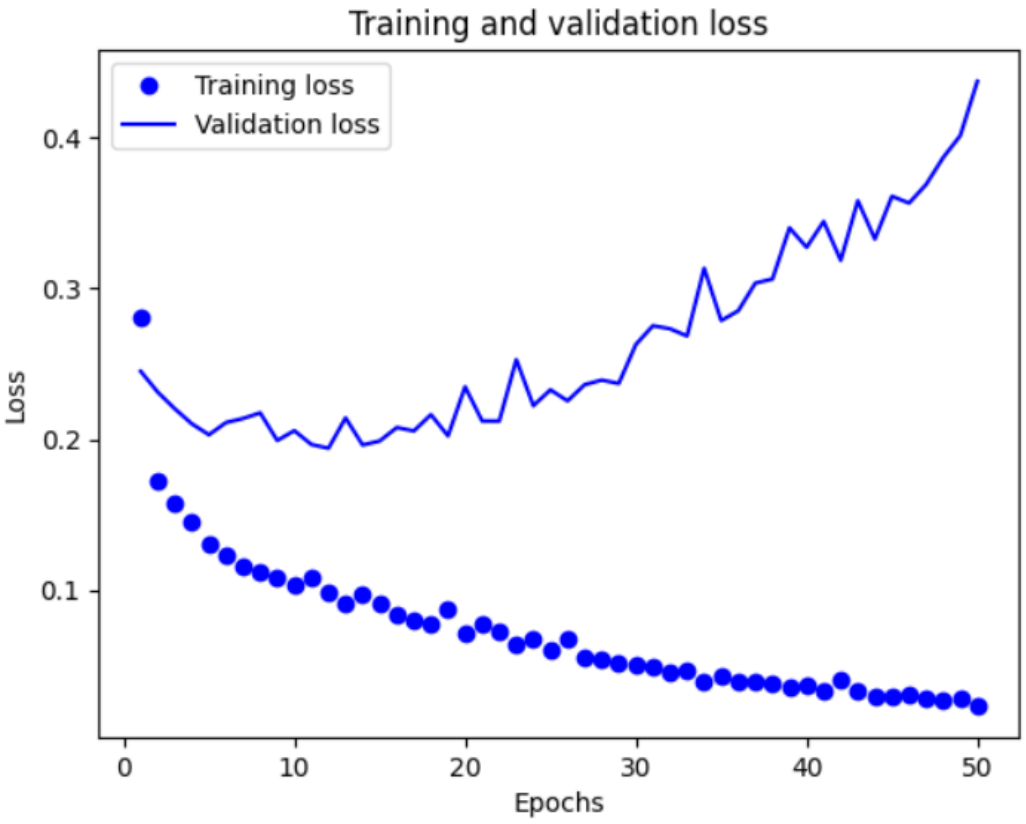
```
Epoch 20/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9805 - loss: 0.0763 - val_accuracy: 0.9350 - val_loss: 0.2347
Epoch 21/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9794 - loss: 0.0898 - val_accuracy: 0.9466 - val_loss: 0.2120
Epoch 22/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9863 - loss: 0.0603 - val_accuracy: 0.9443 - val_loss: 0.2120
Epoch 23/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9769 - loss: 0.0689 - val_accuracy: 0.9350 - val_loss: 0.2527
Epoch 24/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9814 - loss: 0.0660 - val_accuracy: 0.9443 - val_loss: 0.2222
Epoch 25/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9817 - loss: 0.0582 - val_accuracy: 0.9466 - val_loss: 0.2327
Epoch 26/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9832 - loss: 0.0576 - val_accuracy: 0.9443 - val_loss: 0.2253
Epoch 27/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9818 - loss: 0.0581 - val_accuracy: 0.9443 - val_loss: 0.2361
Epoch 28/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9811 - loss: 0.0583 - val_accuracy: 0.9443 - val_loss: 0.2391
Epoch 29/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9899 - loss: 0.0392 - val_accuracy: 0.9490 - val_loss: 0.2369
Epoch 30/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9797 - loss: 0.0564 - val_accuracy: 0.9443 - val_loss: 0.2627
Epoch 31/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9844 - loss: 0.0489 - val_accuracy: 0.9420 - val_loss: 0.2751
Epoch 32/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9806 - loss: 0.0574 - val_accuracy: 0.9443 - val_loss: 0.2731
Epoch 33/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9846 - loss: 0.0437 - val_accuracy: 0.9443 - val_loss: 0.2683
Epoch 34/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9833 - loss: 0.0557 - val_accuracy: 0.9420 - val_loss: 0.3132
Epoch 35/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9869 - loss: 0.0371 - val_accuracy: 0.9466 - val_loss: 0.2785
Epoch 36/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9911 - loss: 0.0307 - val_accuracy: 0.9466 - val_loss: 0.2851
Epoch 37/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9887 - loss: 0.0361 - val_accuracy: 0.9420 - val_loss: 0.3034
Epoch 38/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9932 - loss: 0.0297 - val_accuracy: 0.9466 - val_loss: 0.3060
Epoch 39/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9912 - loss: 0.0305 - val_accuracy: 0.9420 - val_loss: 0.3400

Epoch 40/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9897 - loss: 0.0342 - val_accuracy: 0.9420 - val_loss: 0.3270
Epoch 41/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9890 - loss: 0.0298 - val_accuracy: 0.9397 - val_loss: 0.3442
Epoch 42/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9878 - loss: 0.0274 - val_accuracy: 0.9420 - val_loss: 0.3184
Epoch 43/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9918 - loss: 0.0292 - val_accuracy: 0.9420 - val_loss: 0.3579
Epoch 44/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9936 - loss: 0.0195 - val_accuracy: 0.9420 - val_loss: 0.3325
Epoch 45/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9897 - loss: 0.0308 - val_accuracy: 0.9420 - val_loss: 0.3609
Epoch 46/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9873 - loss: 0.0384 - val_accuracy: 0.9466 - val_loss: 0.3564
Epoch 47/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9909 - loss: 0.0259 - val_accuracy: 0.9420 - val_loss: 0.3687
Epoch 48/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9888 - loss: 0.0354 - val_accuracy: 0.9420 - val_loss: 0.3866
Epoch 49/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9944 - loss: 0.0187 - val_accuracy: 0.9397 - val_loss: 0.4011
Epoch 50/50
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9918 - loss: 0.0259 - val_accuracy: 0.9420 - val_loss: 0.4370
14/14 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9499 - loss: 0.4238
Loss: 0.4370463788509369, Accuracy: 0.94199538230896
14/14 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step
```

La funzione di perdita (***loss function***) è una parte fondamentale del processo di addestramento del modello di rete neurale. Questa funzione valuta le prestazioni del modello ad ogni iterazione durante l'addestramento, confrontando le previsioni del modello con i valori effettivi dei dati di addestramento. L'obiettivo durante l'addestramento è ridurre al minimo la funzione di perdita, il che significa che il modello fa previsioni più vicine ai valori effettivi. In questo caso, la funzione di perdita utilizzata è la "Binary Crossentropy". Questa funzione di perdita è comunemente utilizzata nei problemi di classificazione binaria, quando il modello deve prevedere tra due classi mutuamente esclusive, come "sì" o "no", "positivo" o "negativo". La funzione di perdita di Binary Crossentropy calcola la perdita tra le etichette vere e le previsioni del modello per ciascun esempio di training. La crossentropia binaria penalizza pesantemente le previsioni errate. Se l'etichetta vera è 1 e la previsione del modello è vicina a 0, o se l'etichetta vera è 0 e la previsione del modello è vicina a 1, la perdita sarà elevata. D'altra parte, se la previsione del modello è vicina all'etichetta reale, la perdita sarà bassa. Nell'addestramento del modello, l'obiettivo è ridurre al minimo la Binary Crossentropy, il che significa che il modello sta imparando a fare previsioni che si adattino il più fedelmente possibile alle etichette reali.

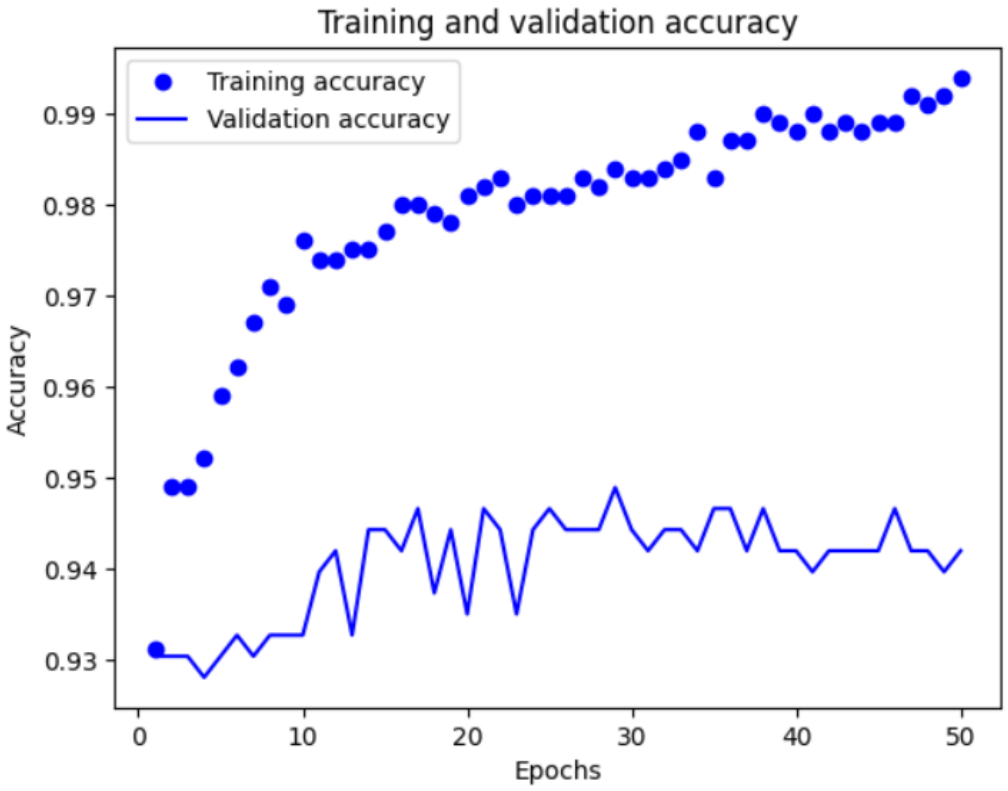
PLOTING

Come vediamo nell'output seguente, la funzione di perdita nel set di dati di training diminuisce all'aumentare del numero di epoche, che è il comportamento desiderato, inizia con un valore di .3887 e termina con una perdita di .0259. Questo comportamento non è lo stesso quando si utilizza il set di dati di test (test_set), che all'aumentare delle epoche aumenta la funzione di perdita. Per quanto riguarda l'accuratezza, notiamo che nel training set ottiene un valore del 99,18% nelle ultime epoche, ma nel test set questa accuratezza si riduce al 94,2%. Potremmo cadere in un eccesso di adattamento, overfitting.

```
plt.clf()
loss = history.history["loss"]
val_loss = history.history["val_loss"]
epochs = range(1, len(loss) + 1)
plt.plot(epochs, loss, "bo", label="Training loss")
plt.plot(epochs, val_loss, "b", label="Validation loss")
plt.title("Training and validation loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.show()
```



```
plt.clf() #Clears the figure
acc = history.history["accuracy"]
val_acc = history.history["val_accuracy"]
plt.plot(epochs, acc, "bo", label="Training accuracy")
plt.plot(epochs, val_acc, "b", label="Validation accuracy")
plt.title("Training and validation accuracy")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend()
plt.show()
```



Per una migliore comprensione dell'accuratezza ottenuta, includiamo gli altri punteggi: ***Accuracy***: 0.9419953596287703, ***Precision***: 0.7272727272727273, ***Recall***: 0.2666666666 6666666, ***F1-score***: 0.3902439024390244, ***Specificity***: 0.9925187032418953, ***ROC AUC***: 0. 7452202826267664.

RESULTATI PREVI

Come puoi vedere l'accuratezza ottenuta è pari a circa il 94,19%, bisogna precisare che questa accuratezza può variare in base al numero di volte in cui si ruota il codice. Rispetto all'articolo a cui abbiamo fatto riferimento in precedenza, abbiamo aggiunto i risultati ottenuti per la valutazione. L'autore afferma che il modello MLP (Perceptron multistrato) si distingue come il miglior classificatore tra gli altri modelli con una accuratezza del 92,06%

tenendo conto degli altri attributi, Precision, Recall, F1-Score, Specificity e AUC-ROC. L'autore prosegue dicendo che inoltre, Random Forest e XGBoost possono avere un punteggio di accuratezza importante, ma esiste una disparità tra Precision, Recall, F1-Score, Specificity, AUC-ROC, che porta a un punteggio basso dal MLP. Questo è un problema comune soprattutto quando si ha a che fare con dati sbilanciati, in questo caso i valori positivi sono la classe minoritaria. I risultati ottenuti sono abbastanza simili a quelli ottenuti dall'autore del paper, con accuratezza simile, non tanto per gli altri parametri ma accettabili. Possiamo aggiungere che il pre-processing iniziale dei dati in questo lavoro è diversa da quella effettuata dall'autore dell'articolo. Nonostante ciò, il modello ottiene un accuratezza elevata, abbastanza simile a quelli ottenuti nella ricerca originale. Di seguito sono riportati i risultati ottenuti nella ricerca dell'autore citato in questo lavoro.

	Model	Accuracy	Precision	Recall	F1-Score	Specificity	AUC-ROC
0	LogisticRegression	0.484919	0.080508	0.791667	0.146154	0.466830	0.629249
1	RandomForestClassifier	0.951276	0.615385	0.333333	0.432432	0.987715	0.660524
2	SVC	0.895592	0.230769	0.375000	0.285714	0.926290	0.650645
3	XGBClassifier	0.953596	0.625000	0.416667	0.500000	0.985258	0.700962
4	LGBMClassifier	0.944316	0.500000	0.375000	0.428571	0.977887	0.676443
5	MLP	0.920635	0.908497	0.896774	0.902597	0.928889	0.918693

Modifica de la rete costruita

Gli iperparametri della rete modificabili cercano la soluzione migliore ed evitano l'overfitting. I parametri maggiormente modificati sono il learning_rate di cui abbiamo discusso in precedenza, il parametro epochs, il parametro batch_size e l'ottimizzatore. Possiamo anche includere la struttura della rete, eliminando nodi e strai, oppure aggiungendo, ma l'architettura implementata ottiene buoni risultati. I parametri su cui non ci concentreremo sono epochs, learning_rate, batch_size e l'ottimizzatore

Il parametro "*epochs*" si riferisce al numero di volte in cui il modello vedrà l'intero set di dati durante il processo di addestramento. Ogni iterazione dell'intero set di dati è nota come "*epochs*". Durante ogni epochs, il modello adegua i propri pesi attraverso il processo di backpropagation dell'errore, in cui i gradienti vengono calcolati rispetto alla funzione di perdita e utilizzati per aggiornare i parametri del modello (pesi e bias) attraverso un algoritmo di ottimizzazione, come la discesa stocastica del gradiente (SGD), Adam e RMSprop. Il numero di epoche è un iperparametro che viene ottimizzato durante l'addestramento del modello e può influire in modo significativo sulle prestazioni del modello. Se vengono utilizzate troppe epoche, esiste il rischio di overfitting, ovvero il modello si adatta eccessivamente ai dati di addestramento e non si generalizza bene ai nuovi dati. D'altra parte, se vengono utilizzate troppo poche epoche, il modello potrebbe non apprendere abbastanza e ottenere prestazioni scadenti. Pertanto, il valore delle epoche viene

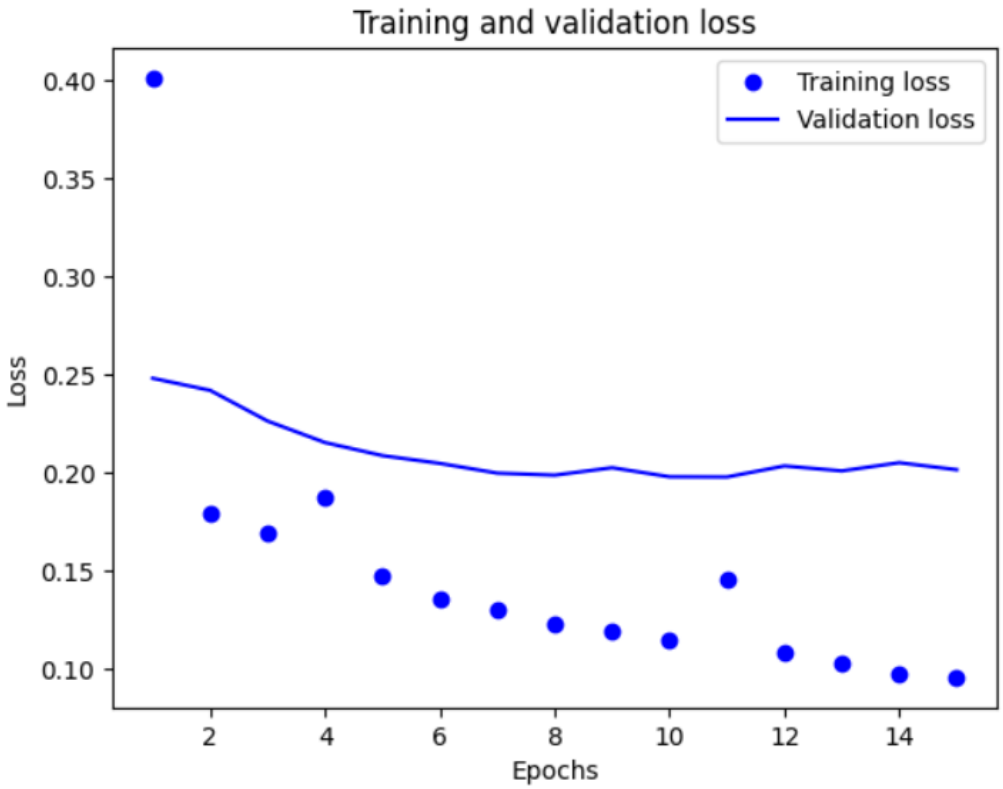
tipicamente scelto valutando le prestazioni del modello su un set di validazione durante l'addestramento con valori diversi di questo parametro. Questo viene fatto per trovare il giusto equilibrio tra adattamento del modello e generalizzabilità.

Il parametro *batch_size* si riferisce al numero di esempi di addestramento utilizzati in un'iterazione per calcolare il gradiente e aggiornare i pesi del modello. Durante il processo di addestramento, i dati di addestramento vengono divisi in batch e il modello viene addestrato utilizzando un batch alla volta. Dopo aver elaborato una serie completa di esempi di addestramento, il gradiente della funzione di perdita viene calcolato rispetto ai pesi del modello e i pesi vengono aggiornati utilizzando un algoritmo di ottimizzazione, come la discesa del gradiente stocastico (SGD), Adam e RMSprop. La dimensione del batch (batch_size) è un iperparametro che viene ottimizzato durante il training del modello e può influire sulle prestazioni e sulla convergenza del modello. Alcuni valori comuni per la dimensione del lotto sono solitamente compresi tra 32, 64, 128 o 256, ma non esiste una regola fissa.

```
from keras.optimizers import RMSprop, Adam
model_1 = Sequential([
    Input(shape=(X_train.shape[1],)),
    Dense(64, activation='relu'),
    Dense(32, activation='relu'),
    Dense(1, activation='sigmoid')
])
#Definir el valor de la tasa de aprendizaje
learning_rate = 0.001
# Crear un objeto optimizador RMSprop con la tasa de aprendizaje especificada
optimizer = Adam(learning_rate=learning_rate)

# Compilación del modelo
model_1.compile(optimizer=optimizer, loss='binary_crossentropy', metrics=['accuracy'])

# Entrenamiento del modelo
history_1=model_1.fit(X_train, y_train, epochs=15, batch_size=40, validation_data=(X_test, y_test))

# Evaluación del modelo
loss, accuracy = model_1.evaluate(X_test, y_test)
print(f"Loss: {loss}, Accuracy: {accuracy}")

# Predicciones
predictions_1 = model_1.predict(X_test)
```

```
Epoch 1/15
26/26 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - accuracy: 0.7666 - loss: 0.5414 - val_accuracy: 0.9304 - val_loss: 0.2482
Epoch 2/15
26/26 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9478 - loss: 0.1921 - val_accuracy: 0.9304 - val_loss: 0.2421
Epoch 3/15
26/26 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9499 - loss: 0.1658 - val_accuracy: 0.9304 - val_loss: 0.2264
Epoch 4/15
26/26 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9516 - loss: 0.1548 - val_accuracy: 0.9304 - val_loss: 0.2154
Epoch 5/15
26/26 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9464 - loss: 0.1554 - val_accuracy: 0.9304 - val_loss: 0.2088
Epoch 6/15
26/26 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9479 - loss: 0.1538 - val_accuracy: 0.9304 - val_loss: 0.2048
Epoch 7/15
26/26 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9607 - loss: 0.1197 - val_accuracy: 0.9281 - val_loss: 0.1999
Epoch 8/15
26/26 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9679 - loss: 0.1072 - val_accuracy: 0.9281 - val_loss: 0.1988
Epoch 9/15
26/26 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9617 - loss: 0.1592 - val_accuracy: 0.9281 - val_loss: 0.2026
Epoch 10/15
26/26 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9629 - loss: 0.1215 - val_accuracy: 0.9374 - val_loss: 0.1980
Epoch 11/15
26/26 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9692 - loss: 0.1227 - val_accuracy: 0.9374 - val_loss: 0.1979
Epoch 12/15
26/26 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9713 - loss: 0.1048 - val_accuracy: 0.9350 - val_loss: 0.2035
Epoch 13/15
26/26 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9732 - loss: 0.1034 - val_accuracy: 0.9350 - val_loss: 0.2010
Epoch 14/15
26/26 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9680 - loss: 0.1160 - val_accuracy: 0.9374 - val_loss: 0.2052
Epoch 15/15
26/26 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9748 - loss: 0.0970 - val_accuracy: 0.9374 - val_loss: 0.2016
14/14 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9445 - loss: 0.2060
Loss: 0.20045830309391022, Accuracy: 0.9373549818992615
14/14 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step
```

```
plt.clf()
loss = history_1.history["loss"]
val_loss = history_1.history["val_loss"]
epochs = range(1, len(loss) + 1)
plt.plot(epochs, loss, "bo", label="Training loss")
plt.plot(epochs, val_loss, "b", label="Validation loss")
plt.title("Training and validation loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.show()
```



Come vediamo, la rete precedente è stata modificata nei parametri epochs=15, batch_size=40 e anche il l'ottimizzatore= Adam. Vedendo graficamente la funzione di perdita del modello precedente, possiamo dedurre che dopo di 10 epoche si verifica un overfitting dei dati e qui la funzione di perdita inizia a crescere nel primo modello. L'accuratezza di 93.73% diminuisce rispetto alla precedente, ma è comunque accettabile

RESAMPLING

Quando si addestra un modello di classificazione con la variabile sbilanciata, incontreremo alcuni problemi. Ciò accade perché il modello di dati della classe dominante supererà quelli della classe meno frequentemente. Generalmente, nei database che hanno una variabile target sbilanciata, la classe con la frequenza più bassa è proprio quella che ci interessa prevedere, il che rende i problemi ancora maggiori.

Poiché una delle classi ha una frequenza molto elevata, il modello costruito con dati sbilanciati può avere una precisione molto elevata e tuttavia non prevedere correttamente eventuali osservazioni per la classe con una frequenza inferiore. Ciò può dare la falsa impressione che il modello funzioni bene quando in realtà non è così.

Per risolvere questi problemi generati da un database sbilanciato possiamo ricorrere a due soluzioni che consistono nel bilanciare i dati della variabile target: undersampling e oversampling. In questo caso abbiamo che la classe positiva è minoritaria con 81 unità. Chiamamo al dataset formato con le variabili dummy che abbiamo formato precedentemente.

La tecnica che utilizzeremo è il oversamplig, poiché con il undersampling perderemmo informazioni riducendo la classe maggioritaria. Il oversampling è una tecnica che consiste nell'aumentare il numero di record di classi con minore frequenza finché il dataset non avrà un numero equilibrato tra le classi della variabile target. Per aumentare il numero di record, possiamo duplicare casualmente i record della classe meno frequentemente. Tuttavia, ciò farà sì che molte informazioni siano identiche, il che può influenzare il modello.

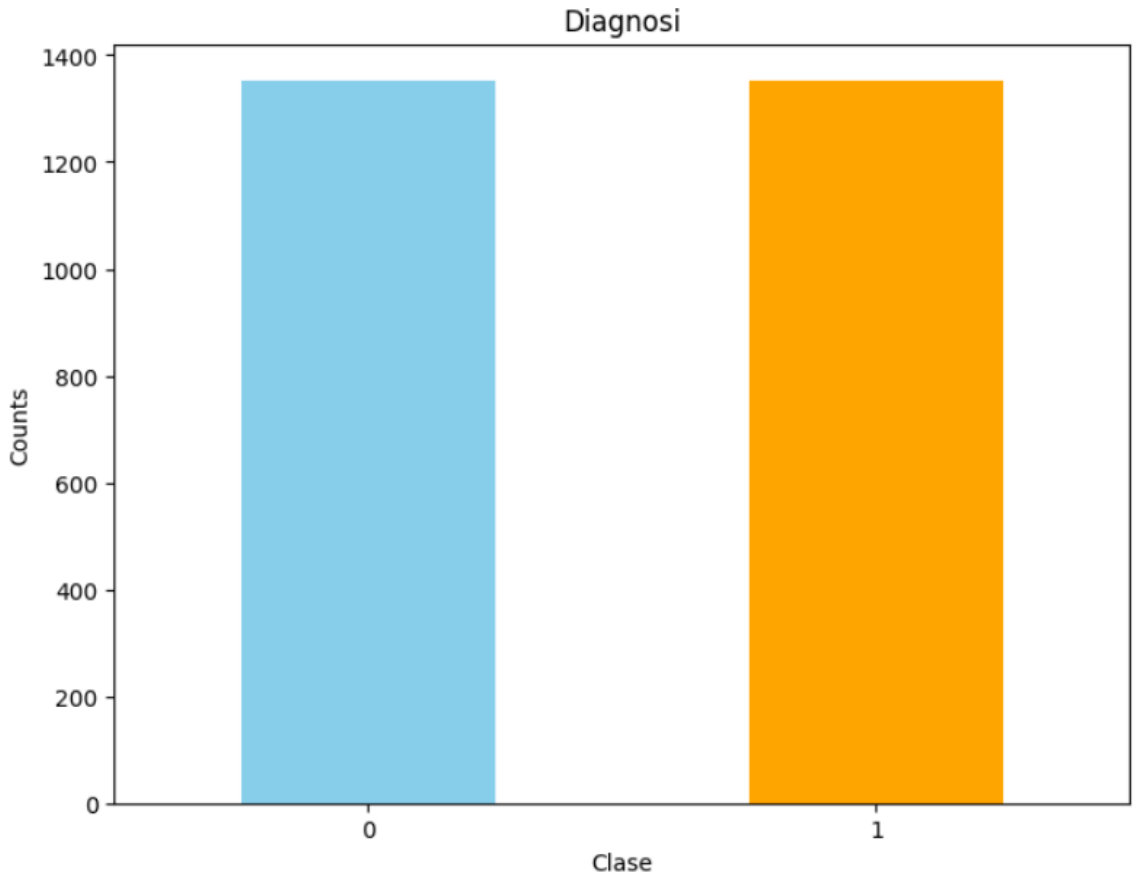
Un vantaggio di questa tecnica è che nessuna informazione viene persa dai record che hanno avuto la classe più frequentemente. Ciò fa sì che il set di dati abbia molti record per alimentare gli algoritmi di apprendimento automatico. A sua volta, i tempi di archiviazione ed elaborazione aumentano in modo significativo e vi è la possibilità di sovradimensionare i dati che sono stati duplicati. Questo overfitting si verifica quando il modello diventa molto efficace nel prevedere i risultati dai dati di training, ma non si generalizza bene ai nuovi dati.

Per evitare di avere troppi dati identici si può utilizzare la tecnica SMOTE, che consiste nel sintetizzare nuove informazioni a partire da informazioni esistenti. Questi dati “sintetici” sono relativamente vicini ai dati reali, ma non sono identici.

SMOTE (Synthetic Minority Over-sampling Technique) è una tecnica utilizzata per affrontare lo squilibrio di classi nei set di dati di machine learning, in particolare nei problemi di classificazione binaria in cui una classe è significativamente più grande dell'altra. L'idea principale alla base di SMOTE è aumentare il numero di campioni nella classe minoritaria generando campioni sintetici che assomiglino ai campioni esistenti in quella classe.

```
from imblearn.over_sampling import SMOTE

X = df_dummy.drop('Diagnosi', axis=1)  # DataFrame de características
y = df_dummy['Diagnosi']  # Serie de etiquetas

# Instanciamos SMOTE
oversampler = SMOTE(random_state=42, k_neighbors=5)

# Aplicamos SMOTE al conjunto de datos
X_resampled, y_resampled = oversampler.fit_resample(X, y)

# Convertimos los datos sobremuestreados en un nuevo DataFrame
df_resampled = pd.concat([pd.DataFrame(X_resampled), pd.DataFrame(y_resampled, columns=['Diagnosi'])], axis=1)
```

Dopo aver applicato la funzione SMOTE vediamo nel grafico seguente come le classi sono distribuite in modo equilibrato, dalla creazione dei campioni sintetici.



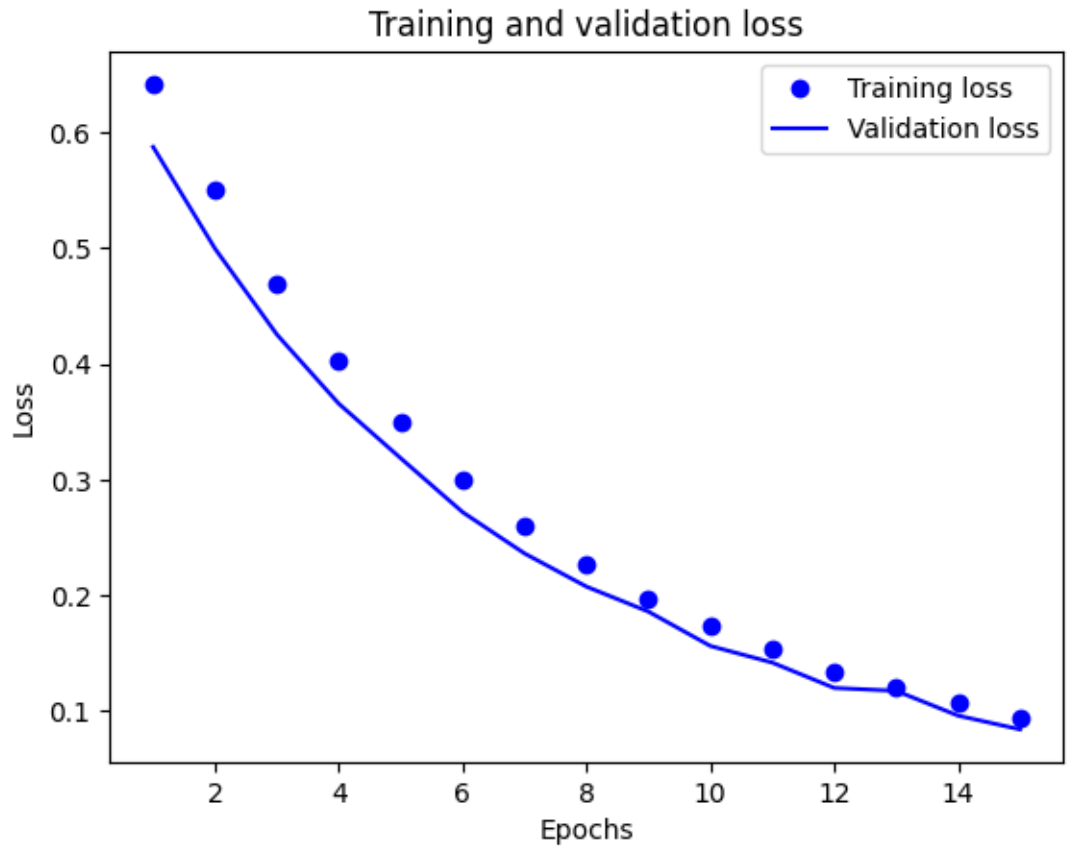

```
Diagnosi
0    1353
1    1353
Name: count, dtype: int64
```

RETE COSTRUITA CON IL RESAMPLING

```python
# División de datos
A = df_resampled.drop('Diagnosi', axis=1)
b = df_resampled['Diagnosi']
X_t, X_t, y_t, y_t = train_test_split(A, b, test_size=0.3, random_state=42)

from keras.optimizers import RMSprop, Adam
model_2 = Sequential([
    Input(shape=(X_train.shape[1],)),
    Dense(64, activation='relu'),
    Dense(32, activation='relu'),
    Dense(1, activation='sigmoid')
])
#Definir el valor de la tasa de aprendizaje
learning_rate = 0.001
# Crear un objeto optimizador RMSprop con la tasa de aprendizaje especificada
optimizer1 = RMSprop(learning_rate=learning_rate)

# Compilación del modelo
model_2.compile(optimizer=optimizer1, loss='binary_crossentropy', metrics=['accuracy'])

# Entrenamiento del modelo
history_2=model_2.fit(X_t, y_t, epochs=15, batch_size=40, validation_data=(X_t, y_t))

# Evaluación del modelo
loss2, accuracy2 = model_2.evaluate(X_t, y_t)
print(f"Loss: {loss2}, Accuracy: {accuracy2}")

# Predicciones
predictions_2 = model_2.predict(X_t)
```

```
Epoch 1/15
21/21 [==============================] - 1s 19ms/step - loss: 0.6419 - accuracy: 0.6059 - val_loss: 0.5874 - val_accuracy: 0.7537
Epoch 2/15
21/21 [==============================] - 0s 6ms/step - loss: 0.5505 - accuracy: 0.7869 - val_loss: 0.4995 - val_accuracy: 0.8116
Epoch 3/15
21/21 [==============================] - 0s 4ms/step - loss: 0.4687 - accuracy: 0.8140 - val_loss: 0.4252 - val_accuracy: 0.8190
Epoch 4/15
21/21 [==============================] - 0s 4ms/step - loss: 0.4034 - accuracy: 0.8264 - val_loss: 0.3654 - val_accuracy: 0.8288
Epoch 5/15
21/21 [==============================] - 0s 5ms/step - loss: 0.3492 - accuracy: 0.8387 - val_loss: 0.3183 - val_accuracy: 0.8473
Epoch 6/15
21/21 [==============================] - 0s 5ms/step - loss: 0.3002 - accuracy: 0.8744 - val_loss: 0.2716 - val_accuracy: 0.9027
Epoch 7/15
21/21 [==============================] - 0s 5ms/step - loss: 0.2599 - accuracy: 0.9076 - val_loss: 0.2360 - val_accuracy: 0.9249
Epoch 8/15
21/21 [==============================] - 0s 6ms/step - loss: 0.2258 - accuracy: 0.9286 - val_loss: 0.2073 - val_accuracy: 0.9421
Epoch 9/15
21/21 [==============================] - 0s 4ms/step - loss: 0.1959 - accuracy: 0.9409 - val_loss: 0.1857 - val_accuracy: 0.9544
Epoch 10/15
21/21 [==============================] - 0s 4ms/step - loss: 0.1732 - accuracy: 0.9532 - val_loss: 0.1560 - val_accuracy: 0.9631
Epoch 11/15
21/21 [==============================] - 0s 6ms/step - loss: 0.1530 - accuracy: 0.9581 - val_loss: 0.1416 - val_accuracy: 0.9631
Epoch 12/15
21/21 [==============================] - 0s 6ms/step - loss: 0.1341 - accuracy: 0.9692 - val_loss: 0.1198 - val_accuracy: 0.9680
Epoch 13/15
21/21 [==============================] - 0s 5ms/step - loss: 0.1202 - accuracy: 0.9704 - val_loss: 0.1172 - val_accuracy: 0.9717
Epoch 14/15
21/21 [==============================] - 0s 5ms/step - loss: 0.1077 - accuracy: 0.9766 - val_loss: 0.0958 - val_accuracy: 0.9828
Epoch 15/15
21/21 [==============================] - 0s 5ms/step - loss: 0.0942 - accuracy: 0.9840 - val_loss: 0.0839 - val_accuracy: 0.9852
26/26 [==============================] - 0s 2ms/step - loss: 0.0839 - accuracy: 0.9852
Loss: 0.08391764014959335, Accuracy: 0.9852216839790344
26/26 [==============================] - 0s 1ms/step
```

```
plt.clf()
loss = history_2.history["loss"]
val_loss = history_2.history["val_loss"]
epochs = range(1, len(loss) + 1)
plt.plot(epochs, loss, "bo", label="Training loss")
plt.plot(epochs, val_loss, "b", label="Validation loss")
plt.title("Training and validation loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.show()
```




```
26/26 [==============================] - 0s 1ms/step
```

```
Accuracy: 0.9852216748768473, Precision: 0.9948849104859335, Recall: 0.974937343358396, F1-score: 0.9848101265822785, Specificity: 0.9951573849878934, ROC AUC: 0.9958036738334941
```

RISULTATI E CONCLUSIONI

In base ai risultati forniti, il modello ha una *Accuracy* del 98,52%, il che significa che circa il 98,52% delle previsioni effettuate dal modello sono corrette. È una misura generale delle prestazioni del modello in tutte le classi. La *Precision* del modello è del 99.48%. Ciò indica che sul totale dei casi classificati come positivi dal modello, circa il 99.48% appartiene effettivamente alla classe dei positivi, valutando le prestazioni del modello nell'identificare la classe positiva e nel minimizzare i falsi positivi. Il *Recall* del modello è del 97,49%. Ciò significa che il modello è in grado di classificare correttamente circa il 97,49% di tutte le istanze positive nel set di dati. La specificità del modello è del 99.51%. Ciò indica che del totale dei casi classificati come negativi dal modello, circa il 99.51% appartiene effettivamente alla classe negativa, valutando le prestazioni del modello nell'identificare la classe negativa e nel ridurre al minimo i falsi negativi. L'area sotto la curva ROC (ROC AUC) è 99,58%. Questa metrica fornisce una misura della capacità del modello di distinguere tra classi positive e negative. Maggiore è il valore, migliori saranno le prestazioni del modello. In generale, i risultati mostrano che il modello ha prestazioni buone, con valori elevati in tutte le metriche valutate. La precisione, il recall e la specificità sono tutti elevati, indicando che il modello è in grado di fare previsioni accurate sia per la classe positiva che per quella negativa. Inoltre, l'area sotto la curva ROC è molto elevata, indicando una buona capacità di discriminazione del modello. Rispetto ai modelli precedenti, questo ha caratteristiche migliori. I modelli precedenti classificavano meglio la classe negativa a causa dello squilibrio del dataset e quindi avevano una precisione, recall e F1-score inferiore.

RIFERIMENTI BIBLIOGRAFICI

Optimizing UTI Diagnosis with Machine Learning and Artificial Neural Network for Reducing Misdiagnoses by Agdeppa et al. (2023).