UNIVERSITÀ DEGLI STUDI DI CATANIA

Dipartimento di Matematica e Informatica

Corso di Laurea Triennale in Informatica

Lemuel Puglisi

Autoencoder per la riduzione della dimensionalità di dataset molecolari e conseguente predizione di dati clinici

Relazione Progetto Finale

Relatore: Dott. Alaimo Salvatore

Correlatore: Prof. Ferro Alfredo

Correlatore: Dott. Micale Giovanni

Anno Accademico 2020 - 2021

Abstract

La ricerca sul carcinoma mammario è molto sviluppata a livello molecolare, tanto da aver definito dei particolari sottotipi molecolari di tumore al seno, individuabili attraverso profili dell'espressione genica del tessuto ammalato. Tali profili vengono estratti attraverso diverse tecniche di gene expression profiling, tra cui RNA Sequencing e Microarray. Il profilo dell'espressione genica è un dato ad alta dimensionalità poiché, in questo caso, fa riferimento alle migliaia di geni del genoma umano. In questa tesi, vogliamo proporre metodi complessi di riduzione della dimensionalità, quali i deep autoencoder, per effettuare una compressione dei profili dell'espressione genica che mantenga le proprietà significative dei dati originali. Una bassa dimensionalità aiuta i metodi di analisi dati a fornire risultati statisticamente significativi. Valuteremo il modello su due dataset di profili dell'espressione genica, uno ottenuto attraverso la tecnica RNA Sequencing, ed un altro ottenuto attraverso Microarray, entrambi contenenti anche i dati clinici dei rispettivi pazienti ammalati. Da questi risultati effettueremo la classificazione del rispettivo sottotipo molecolare, ottenendo una accuratezza media dell'82% per i dati RNA-Seq e del 60% per i dati Microarray. Alterando la struttura dell'autoencoder, riusciremo a migliorare le performance sino ad una accuratezza dell'85% e del 74% rispettivamente. Verranno monitorate le metriche sensitivity, specificity, precision e negative predicted value per ogni sottotipo molecolare, assicurando la produzione di un modello bilanciato. Il deep autoencoder super di gran lunga le performance di metodi classici di riduzione della dimensionalità, come la PCA.

Alla mia famiglia, che crede in me dal primo giorno.

Indice

Capitolo 1

Introduzione

1.1 Il carcinoma mammario

La ricerca sul cancro al seno ha scoperto sempre di più sui diversi tipi di cellule coinvolte. Questo ha portato a definire la malattia a livello molecolare: la classificazione molecolare del carcinoma mammario si riferisce ai pattern che queste cellule mostrano. Una comprensione più profonda dei vari sottotipi di cancro al seno ha consentito agli scienziati di sviluppare trattamenti mirati per risultati migliori. È frequente l'utilizzo di profili delle espressioni geniche delle cellule coinvolte, ottenuti attraverso tecniche come i microarray o il Next Generation Sequencing. A partire da tali profili, attraverso delle tecniche di analisi dati, si cerca di fare inferenza sul sottotipo di tumore al seno, o su altri parametri legati alla malattia.

1.2 Curse of dimensionality

Il profilo dell'espressione genica di un campione è un dato ad altissima dimensionalità: nel caso del tumore al seno si fa riferimento alla trascrizione di circa 20.000 geni per ogni record, approssimativamente il numero di geni identificati nel genoma umano. Quando la dimensione del dato è di gran lunga maggiore della numerosità di dati disponibili, si va incontro al fenomeno della curse of dimensionality [1]: all'aumentare della dimensionalità, il volume dello spazio aumenta così velocemente che i dati disponibili diventano sparsi. Per ottenere un risultato statisticamente valido da metodi di analisi dati, la quantità di dati necessari cresce esponenzialmente con la dimensionalità. Spesso, gli algoritmi di predizione o di clustering si basano sul rilevamento di aree in cui gli oggetti formano gruppi con proprietà simili; nei dati ad alta dimensione, gli oggetti tendono ad essere sparsi e dissimili in

molti modi, per cui gli algoritmi richiedono molti più dati per rilevare aree con significatività statistica.

1.3 Riduzione della dimensionalità

Per riduzione della dimensionalità si intende la trasformazione di dati ad alta dimensionalità, in dati a più bassa dimensionalità, eliminando informazioni ridondanti e correlate, e mantenendo intatte le proprietà significative legate ai dati originali. La riduzione della dimensionalità è utilizzata principalmente per combattere il fenomeno della curse of dimensionality. Il metodo più frequente è la Principal Component Analysis (PCA) [2], che consiste in una trasformazione lineare delle feature, proiettando quelle originarie in un nuovo spazio a più bassa dimensione, in cui gli assi corrispondono alle componenti principali, ovvero quelle direzioni per cui si ha una maggiore variabilità dei dati. Tuttavia, la PCA non riesce a coprire dipendenze non-lineari tra le feature, per cui risulta essere limitata nella sua assunzione di linearità. Applicando la PCA ai profili di espressioni geniche in input, ed utilizzando le prime due componenti principali, si ottiene il grafico 1.1. Dalla proiezione si nota la difficoltà nell'individuare cluster omogenei di dati con lo stesso sottotipo di tumore al seno.

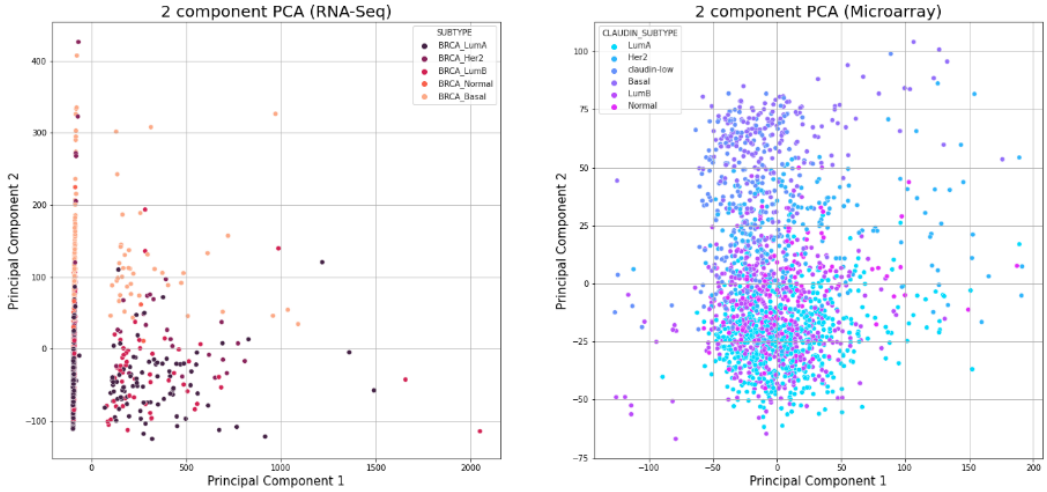

Figura 1.1: PCA. Plot dei dati sulle prime due componenti principali.

1.4 Autoencoder

Gli autoencoder sono reti neurali utilizzate per ridurre la dimensionalità dei dati in uno spazio latente a più bassa dimensione, concatenando trasformazioni lineari e/o non lineari. Parleremo della loro struttura e del loro funzionamento nella sezione 2.7. Al contrario della PCA, gli autoencoder, o in generale le reti neurali, sono degli approssimatori universali [3] e riescono a rilevare dipendenze non lineari tra le feature. Utilizzeremo gli autoencoder per comprimere i profili di espressioni geniche in uno spazio latente a basse dimensioni, tentando una riduzione di 2 o 3 ordini di grandezza. Valideremo la significatività dei dati ridotti tentando di predire il sottotipo di tumore al seno legato al campione.

Capitolo 2

Conoscenze preliminari

In questa sezione sono introdotti i concetti necessari alla comprensione della tesi. Lo strumento principale è l'autoencoder, per cui è necessario avere chiaro il funzionamento di una rete neurale, della sua architettura, delle funzioni di attivazione, funzioni loss, metriche di validazione. Inoltre è necessaria una introduzione ai modelli ensemble, utilizzati per predire i dati clinici. Importante è la comprensione della provenienza dei dati di dominio biologico: verrà fornita una introduzione al Next Generation Sequencing e ai Microarray.

2.1 Gene expression profiling

Per gene expression profiling (analisi dell'espressione genica) intendiamo la misura dell'attività, ovvero l'espressione, di migliaia di geni alla volta, per creare una immagine globale della funzione cellulare. Nei tessuti affetti da tumore, ad esempio, possiamo utilizzare tali profili per confrontare una cellula normale ed una cellula mutata. I profili si ottengono attraverso vari metodi, come i Microarray o le tecniche Next Generation Sequencing. Descriviamo brevemente il loro funzionamento nei paragrafi successivi.

2.1.1 RNA-Seq

RNA-Seq (RNA Sequencing) è una tecnica che fa uso di Next Generation Sequencing e serve a descrivere quali geni sono attivi e quanto sono trascritti. In generale, prendono il nome di Next Generation Sequencing, o Massive Parallel Sequencing, tutti gli approcci di sequenziamento ad alto throughput. RNA-Seq è definibile tecnica ad alto throughput, in quanto esegue milioni di letture in parallelo [4]. Una volta effettuate le letture, esse si selezionano in base alla qualità e si allineano su un genoma di riferimento. Ad ogni lettura si

assegnano dei geni annotati (o meno, nel caso non siano noti). Si effettua un conteggio dei geni e si normalizza attraverso tecniche apposite, definendo il profilo dell'espressione genica appartenente alle cellule del tessuto analizzato.

2.1.2 Microarray

Un microarray è un insieme di sonde microscopiche fissate in una superficie solida in vetro, plastica o chip di silicio. Le sonde sono disposte a matrice (array bidimensionale) sulla superficie e permettono di analizzare simultaneamente la presenza di geni all'interno del campione. Dal risultato delle sonda si crea un profilo dell'espressione genica del campione.

2.2 Reti neurali biologiche

Nel campo dell'apprendimento automatico, una rete neurale articiale è un modello computazionale ispirato alle reti neurali biologiche. In biologia le reti neurali cerebrali sono composte da neuroni, i quali attraverso le connessioni sinaptiche trasmettono informazioni. Nel corpo cellulare del neurone, detto soma, si ramificano le fibre minori, i dendriti, e la fibra principale, detta assone, che si allontana dal neurone per portare l'output ad altri neuroni. Il passaggio delle informazioni avviene attraverso dei neurotrasmettitori, che vengono captati dai neuroni attraverso dei recettori. Essi si accumulano tra il corpo sinaptico del neurone e l'esterno, generando una differenza di potenziale. Se questo potenziale supera una certa soglia, detta di attivazione, viene prodotto un impulso: il neurone propaga, a sua volta, neurotrasmettitori dalle sinapsi dell'assone.

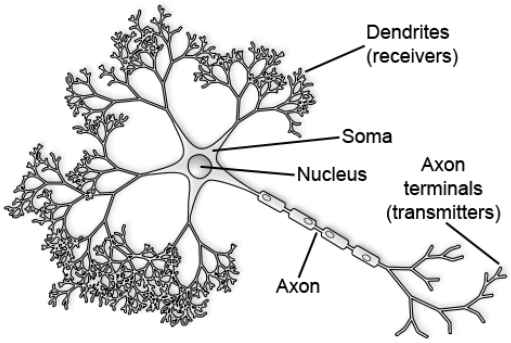

Figura 2.1: Un neurone.

2.3 Reti neurali artificiali

Così come una rete neurale biologica, una rete neurale artificiale è un modello computazionale composto da "neuroni" artificiali, vagamente ispirati al funzionamento dei neuroni reali. La rete neurale dispone i propri neuroni artificiali in layer comunicanti. I neuroni di un layer prendono dati in input dall'output del layer precedente e danno in output dei valori al layer successivo. Questi collegamenti sono pesati da coefficienti, che vengono stimati a partire dai dati attraverso una fase di apprendimento, cercando di ottenere il minor errore di predizione possibile.

2.3.1 Neurone artificiale

Un neurone artificiale prende in ingresso n input $(x_1, ..., x_n)$, pesati rispettivamente con n coefficienti o pesi $(w_1, ..., w_n)$, che rappresentano l'efficacia delle connessioni sinaptiche dei dendriti. Tali parametri saranno calcolati a partire dai dati durante la fase di apprendimento. Esiste un ulteriore peso, detto *costante di bias*, che si considera collegato ad un input fittizio con valore costante 1, questo peso è utile per tarare il punto di lavoro ottimale del neurone. Il neurone somma i prodotti tra gli input ed i corrispettivi pesi (compresa la costante di bias) e produce un valore z. Dopodiché, sulla base di una funzione di attivazione φ a cui viene passato il valore z, produce un valore di output.

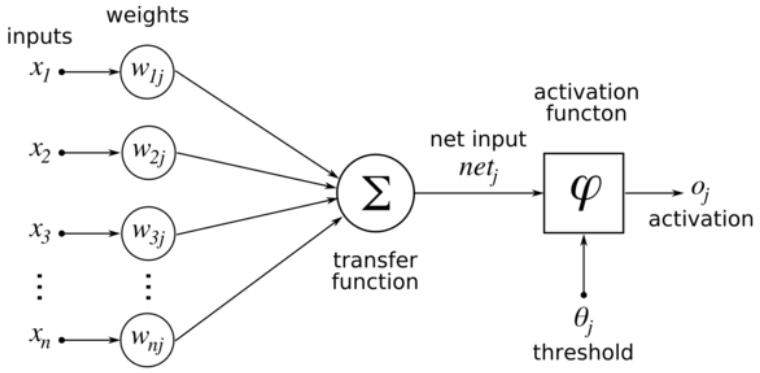

Figura 2.2: Percettrone.

2.3.2 Layer

Come accennato, le reti neurali sono composte da gruppi di neuroni artificiali organizzati in layer (livelli). Tipicamente sono presenti un layer di input, un layer di output, ed uno o più layer intermedi (*hidden layer*). La presenza di più layer intermedi etichetta la rete neurale come deep neural network. Il layer di input è costituito da un vettore di n valori $\bar{x} = (x_1, \ldots, x_n)$. I layer intermedi hanno un numero arbitrario di neuroni ed ogni neurone si collega ad un numero arbitrario di output del layer precedente, per produrre un valore da inviare in output al layer successivo. Il layer di output è l'ultimo layer della rete e costituisce l'output del modello. Se ogni nodo di ogni layer è connesso a tutti gli output dei nodi del layer precedente, allora si parla di rete neurale densa, ed i layer sono totalmente connessi.

2.3.3 Funzione di attivazione

Ogni neurone sfrutta una funzione di attivazione φ per produrre un output. La funzione prende in input il prodotto scalare tra il vettore dei coefficienti del neurone $\bar{w} = (b, w_1, \ldots, w_n)$ ed il vettore di input $\bar{x} = (1, x_1, \ldots, x_n)$, per cui l'output del neurone sarà $\varphi(\bar{x} \cdot \bar{w})$. Tutti i neuroni nello stesso layer hanno la stessa funzione di attivazione. La scelta della funzione φ si lega alla scelta della funzione loss, ovvero la funzione che quantifica l'errore di predizione. La funzione di attivazione deve rispettare alcune proprietà, che permettono all'algoritmo di ottimizzazione del modello di lavorare in maniera stabile. La funzione deve essere continua e differenziabile, la derivata non deve saturare a zero, ne esplodere all'infinito nel suo dominio. Le ultime due proprietà prevengono rispettivamente lo stallo e l'instabilità numerica dell'algoritmo di ottimizzazione. Di seguito elencheremo le funzioni di attivazione utilizzate nella tesi.

2.3.3.1 Funzione logistica

La funzione logistica appartiene alla classe delle funzioni sigmoidee, aventi cioé una curva a forma di S. È definita in $\mathbb{R}$ ed ha valori nell'intervallo $[0, 1]$. La definizione è la seguente:

$$\sigma(x) = \frac{1}{1 + \exp(-x)} = \frac{\exp(x)}{1 + \exp(x)}$$

2.3.3.2 Tangente iperbolica

Anche la tangente iperbolica appartiene alla classe delle *funzioni sigmoidee*. È definita in $\mathbb{R}$ ed ha valori nell'intervallo $[-1, 1]$ e risulta essere una versione

scalata e traslata della funzione logistica. La definizione è la seguente:

$$\tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$$

2.3.3.3 ReLU

La funzione ReLU (Rectified Linear Unit) prende spunto dai raddrizzatori a singola semionda usati in elettronica per trasformare un segnale alternato in un segnale unidirezionale, facendo passare solo semionde positive. È formalmente definita come segue:

$$\text{ReLU}(x) = \max(0, x) = \begin{cases} x \text{ if } x \geq 0 \\ 0 \text{ if } x < 0 \end{cases}$$

La funzione ReLU non satura mai per valori di x positivi. Nella pratica, le reti neurali che utilizzano ReLU offrono uno speed-up significativo nella fase di training rispetto alle funzioni sigmoidee. Sia il calcolo della funzione che della sua derivata sono molto semplici e veloci da effettuare poiché non è richiesta l'esponenziazione. ReLU soffre di problemi di saturazione della sua derivata quando x è negativo.

2.3.3.4 Softmax

A differenza delle funzioni precedenti, la funzione softmax non opera sulla singola componente del vettore, ma sull'intero vettore. Sia $\bar{x} = (x_1, ..., x_n)$ un vettore di valori, la funzione softmax è definita come $\mu(\bar{x}) = (\mu(x_1), ..., \mu(x_n))$ dove il generico $\mu(x_i)$ è calcolato come segue:

$$\mu(x_i) = \frac{exp(x_i)}{\sum_{j=1}^{n} exp(x_j)}$$

La funzione ha valori in $[0, 1]$. La somma dei valori calcolati su ogni componente del vettore è 1, dunque $\mu(\bar{x})$ è una distribuzione di probabilità. Grazie all'esponenziale, le componenti del vettore con valori più alti ricevono valori molto più alti rispetto alle altre. In particolare se c'è una componente con un valore significativamente più alto rispetto a tutti gli altri, a questa componente corrisponderà un valore vicino ad 1, mentre alle restanti un valore prossimo allo 0. La softmax ha problemi di saturazione, che possono essere aggirati utilizzando come funzione costo l'*entropia incrociata*.

2.4 Funzioni loss

La loss function, o funzione obiettivo, è quella funzione da minimizzare durante la fase di apprendimento del modello. Quantifica la differenza tra le predizioni del modello ed i valori reali indicati nel training set. I coefficienti ottimali della rete sono quelli che minimizzano la funzione loss. Distinguiamo due tipologie di funzioni loss in base al problema che affronta la rete neurale: la **regression loss**, che da in output uno o più valori reali, e la **classification loss**, che da in output una distribuzione di probabilità.

2.4.1 Regression loss

Sia $\bar{x} = (x_1, ..., x_n)$ l'input, y l'output reale ed $\hat{y}$ il valore predetto dalla rete neurale. Le prime due regression loss function più comuni sono la **squared error loss**:

$$L(\hat{y}, y) = (\hat{y} - y)^2$$

e la **Huber loss**:

$$L_\delta(\hat{y}, y) = \begin{cases} (\hat{y} - y)^2 \text{ if } |\hat{y} - y| \leq \delta \\ 2\delta \times (|\bar{y} - y| - \frac{1}{2}\delta) \text{ otherwise} \end{cases}$$

dove δ è una costante. Per valori minori di δ, la Huber loss è definita come la squared error loss. Mentre la Huber loss penalizza di meno le differenze tra i valori reali e quelli predetti, la squared error loss tende all'infinito anche con errori relativamente piccoli. La variante Huber loss protegge il processo di learning nel caso di outlier, dove la squared error segnalerebbe un errore molto grande.

2.4.1.1 Mean Squared error

Le funzioni squared error ed Huber loss funzionano per un solo dato. Nel caso volessimo calcolare l'errore su un insieme di dati su cui è stata effettuata una predizione, dovremmo affidarci ad altri metodi. Sia $T = \{(x_1, y_1), ..., (x_n, y_n)\}$ il training set e $P = \{(x_1, \hat{y}_1), ..., (x_n, \hat{y}_n)\}$ l'insieme di coppie formate dai valori di input e dai rispettivi output predetti dalla rete. Definiamo il **Mean Squared Error** come la misura dell'errore quadratico medio:

$$\text{MSE}(P, T) = \frac{1}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2$$

La radice quadrata del MSE è definita **Root Mean Square Error** (RMSE). Nella pratica è preferibile il MSE perché il calcolo della derivata è più semplice (ai fini dell'utilizzo dell'algoritmo di discesa del gradiente). A causa del termine al quadrato, il MSE è molto sensibile agli outliers. Pochi outlier possono incrementare di molto il valore MSE, compromettendo il learning. Per attenuare questo effetto, conviene calcolare l'errore medio utilizzando la **Huber Loss**.

2.4.2 Classification loss

Poniamoci in un generale problema di classificazione con k classi $C_1, ..., C_k$. Supponiamo di avere un training set $T = \{(\bar{x}_1, \bar{p}_1), ..., (\bar{x}_n, \bar{p}_n)\}$ dove $\bar{x}$ è il vettore degli input e $\bar{p} = (p_1, ..., p_k)$ è una distribuzione di probabilità. La componente p_i del vettore $\bar{p}$ indica la probabilità che $\bar{x}$ appartenga alla classe C_i. Supponiamo che la rete neurale sia progettata, ad esempio attraverso una softmax, per produrre in output una distribuzione di probabilità $\bar{q} = (q_1, ..., q_n)$, dove q_i indica la probabilità predetta dal modello che $\bar{x}$ sia di classe C_i. Le funzioni di classification loss quantificano la **distanza** tra le distribuzioni di probabilità $\bar{p}$ e $\bar{q}$.

2.4.2.1 Entropia

Supponiamo di avere un alfabeto di n simboli. Si vuole trasmettere un messaggio utilizzando questi simboli attraverso un canale di informazione. Sia $\bar{p} = (p_1, ..., p_n)$ una distribuzione di probabilità. Supponiamo che in ogni punto del messaggio la probabilità di osservare l'i-esimo simbolo sia p_i. Il **teorema di Shannon** [5] afferma che, in un sistema di codifica ottimale, il numero medio di bit per simbolo necessari per codificare il messaggio è dato dall'entropia $H(\bar{p})$:

$$H(\bar{p}) = -\sum_{i=1}^{n} p_i \log_2 p_i$$

Il termine $-\log p_i$ indica il numero di bit necessari a rappresentare l'i-esimo simbolo usando lo schema di codifica ottimale. Qualunque altro schema utilizza in media più bit, dunque è sub-ottimale.

2.4.2.2 Entropia incrociata

Nella teoria dell'informazione, l'entropia incrociata (o cross-entropy) tra due distribuzioni di probabilità p e q, relative allo stesso insieme di eventi, misura

il numero *medio* di bit necessari ad identificare un evento estratto dall'insieme, nel caso sia utilizzato uno schema alternativo q anziché la vera distribuzione p. Supponiamo di cambiare lo schema di codifica, usando una diversa distribuzione di probabilità $\bar{q} = (q_1, ..., q_n)$. Con questo nuovo schema occorrono $-\log q_i$ bit per rappresentare l'i-esimo simbolo. Quanti bit per simbolo occorrono in media se usiamo questo schema di codifica sub-ottimale? Dalla definizione di entropia incrociata possiamo calcolarlo come segue:

$$C(\bar{p}||\bar{q}) = -\sum_{i=1}^{n} p_i \log_2 q_i$$

2.4.2.3 Divergenza di Kullback-Leibler

In generale sappiamo che $C(\bar{p}||\bar{q}) > H(\bar{p})$ poiché $\bar{q}$ è sub-ottimale, mentre $\bar{p}$ è ottimale. La differenza tra $C(\bar{p}||\bar{q})$ e $H(\bar{p})$, chiamata Divergenza di *Kullback-Leibler* (KL), misura il numero medio di bit in più che occorrono per ogni simbolo:

$$KL(\bar{p}||\bar{q}) = C(\bar{p}||\bar{q}) - H(\bar{p}) = \sum_{i=1}^{n} p_i \log_2 \frac{p_i}{q_i}$$

Minimizzare la divergenza di KL equivale a minimizzare l'entropia incrociata. Nella pratica si utilizza l'entropia incrociata come funzione loss. Essa è la più comune funzione di classification loss ed è spesso accoppiata con la funzione di attivazione *softmax* in un'unica funzione numericamente più stabile, la cui derivata è semplice da calcolare.

2.4.2.4 Binary Cross-Entropy Loss

Come abbiamo già detto, la cross-entropy può essere utilizzata per definire una funzione loss. Poniamoci nel caso binario in cui le sole due classi di appartenenza sono $\{0, 1\}$. Sia p_i la reale probabilità che la i-esima tupla appartenga alla classe 1. Sia q_i la probabilità stimata dall'algoritmo che la i-esima tupla appartenga alla classe 1. Utilizziamo una funzione di attivazione che dia in output delle probabilità, come la *softmax*. Data una tupla x, la softmax ci comunicherà che con probabilità $q_{y=1} = \hat{y}$ la tupla apparterrà alla classe 1, mentre con probabilità $q_{y=0} = 1 - \hat{y}$ la tupla apparterrà alla classe 0. Avendo fissato le notazioni $p \in \{y, 1-y\}$ e $q \in \{\hat{y}, 1-\hat{y}\}$ è possibile utilizzare la cross-entropy per misurare la dissimilarità tra p e q:

$$C(p \,||\, q) = -\sum_{i} p_i \times \log_2(q_i) = -y \times \log \hat{y} - (1-y) \times \log(1-\hat{y})$$

Utilizziamo la cross-entropy come loss function per una tupla ed ipotizziamo di voler calcolare la cross-entropy media per tutte le n tuple del training set:

$$J(w) = \frac{1}{n}\sum_{i=1}^{n}[-y \times \log \hat{y} - (1-y) \times \log(1-\hat{y})] =$$
$$= -\frac{1}{n}\sum_{i=1}^{n}[y \times \log \hat{y} + (1-y) \times \log(1-\hat{y})]$$

Tale funzione loss è chiamata **binary cross-entropy loss**.

2.5 Fase di apprendimento

La fase di apprendimento, o training, di una rete neurale consiste nel trovare i parametri (o pesi) che minimizzino la funzione loss sul training set. Per trovare tali parametri è necessario l'utilizzo di algoritmi di ottimizzazione, la maggior parte dei quali sono basati sul gradiente. Alcuni di questi sono: batch gradient descent, stochastic gradient descent, adam. Prima di passare agli algoritmi di ottimizzazione, definiamo *epoca* un passaggio dell'algoritmo sull'intero training set, e definiamo *batch* un insieme di dati utilizzato per l'aggiornamento dei pesi.

2.5.1 Algoritmo di discesa del gradiente

L'idea principale degli algoritmi basati sulla discesa del gradiente è la seguente: supponiamo di trovarci all'interno di un neurone artificiale con pesi inizializzati casualmente: $w = (w_1, w_2, \ldots, w_n)$. Omettiamo per semplicità la costante di bias e supponiamo che prenda in input un vettore di feature $x = (x_1, x_2, \ldots, x_n)$. Sia $J(w)$ la funzione loss, calcoliamo il gradiente $\nabla J(w)$: se la i-esima componente del gradiente è positiva, vorrà dire che il peso i-esimo contribuisce ad incrementare la funzione loss, anziché farla decrescere. Per raggiungere il minimo è necessario dirigersi nel verso opposto al gradiente, quindi aggiornare i pesi come segue:

$$w = w - \eta \nabla J(w)$$

Dove η è un iper-parametro, chiamato *learning rate*, che ci permette di stabilire la larghezza di ogni step di aggiornamento. Il processo convergerà ad un minimo locale.

2.5.2 Backpropagation

Supponiamo che la rete neurale abbia k layer. L'i-esimo layer avrà $n^{(i)}$ neuroni con $w^{(i)}$ pesi, per cui è possibile organizzare tutti i pesi in una matrice W_i di dimensione $n^{(i)} \times w^{(i)}$. Ogni layer necessita che i suoi pesi vengano aggiornati durante il training. L'algoritmo di backpropagation percorre la rete neurale nel verso opposto, partendo dall'output, e calcola il gradiente della funzione loss J rispetto ai vari layer utilizzando la regola della catena:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

Di volta in volta, i pesi del layer corrente vengono aggiornati come descritto nel paragrafo precedente.

2.6 Monitoraggio di qualità

Lo scopo finale del processo di training è minimizzare la loss media su nuovi input. Un problema abbastanza comune è quello di costruire un modello che funziona molto bene sul training set, ma con performance peggiori su dai mai visti prima, si parla quindi di *overfitting*. Le deep neural network sono particolarmente suscettibili all'overfitting, tuttavia esistono dei metodi di regolarizzazione che aiutano a ridurlo.

2.6.1 Aggiunta di penalità

Il metodo di discesa del gradiente potrebbe convergere ad un minimo locale, che non corrisponde al minimo assoluto della funzione di loss. Nella pratica, pesi più piccoli producono modelli migliori e generali. È possibile forzare l'algoritmo di discesa del gradiente a favorire soluzioni con pesi piccoli aggiungendo un *termine di penalità* alla loss function J usata dal modello:

$$J'(W) = J(W) + \alpha||W||^2$$

Dove α è un iper-parametro che permette di regolare l'impatto del termine di penalità. Più alti sono i pesi, maggiore è la norma del vettore dei pesi, quindi maggiore è la penalità introdotta.

2.6.2 Early Stopping

Nella pratica si osservano comportamenti diversi della funzione loss sul training set e sul test set. Nel training set la funzione loss decresce durante

l'addestramento. Nel test set potrebbe succedere che la funzione loss decresca sino a raggiungere un minimo e, dopo un certo numero di iterazioni, cresca (overfitting). Per evitare questo problema, si può fermare prematuramente il training non appena la loss smette di decrescere nel test set. Il rischio che si corre con l'early stopping è quello di produrre overfitting sul test set. Per evitare ciò si può costruire un validation set indipendente dai primi due. Il validation set non contribuisce in alcun modo al training della rete neurale, per cui è un buon candidato per effettuare la valutazione del modello.

2.6.3 Dropout

Il Dropout è una tecnica che ha lo scopo di creare un consenso sui valori ottimali dei pesi, considerando il risultato ottenuto da diverse sottoreti neurali. Sulla base di un iper-parametro chiamato *dropout rate*, ad ogni epoca si seleziona in maniera casuale una frazione di nodi della rete e si scarta temporaneamente. Sulla rete neurale ridotta si effettua una iterazione del metodo di discesa del gradiente. Dal momento in cui la rete neurale completa contiene più nodi rispetto a quella utilizzata durante il training, alla fine del processo di allenamento i pesi ottenuti vengono ri-scalati.

2.6.4 Batch normalization

La Batch Normalization è una tecnica per rendere il training delle deep neural network più veloce e più stabile. Il risultato di un hidden layer con n unità è un vettore di dimensione n, le cui componenti sono i risultati di una funzione di attivazione. La Batch Normalization normalizza questo vettore utilizzando la media e la varianza del batch corrente. Questa tecnica può essere applicata anche prima di passare i risultati alla funzione di attivazione.

2.7 Autoencoder

Gli autoencoder sono reti neurali con lo scopo di generare nuovi dati dapprima comprimendo l'input in uno spazio di variabili latenti (embedding space) e, successivamente, ricostruendo l'output sulla base delle informazioni acquisite. Sono formati da due componenti: un **encoder** $\bar{h} = f(\bar{x})$ che comprime l'input $\bar{x}$ in uno spazio di variabili latenti, ed un **decoder** $\bar{r} = g(\bar{h})$ che ricostruisce l'input sulla base delle informazioni raccolte. Se $\bar{h}$ è dimensionalmente minore di $\bar{x}$, allora durante la fase di apprendimento si forza

l'autoencoder ad estrarre solo le caratteristiche salienti dall'input.

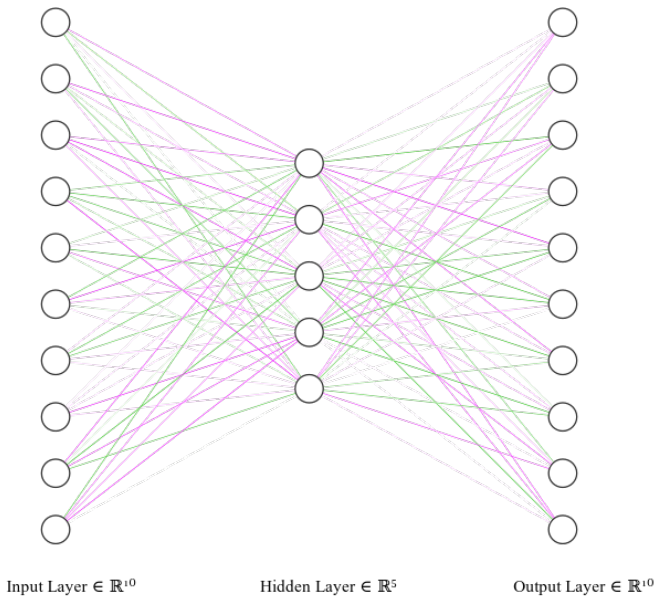

Figura 2.3: Vanilla autoencoder.

I principali utilizzi degli autoencoder sono la riduzione della dimensionalità e, attraverso alcune tecniche aggiuntive, la riduzione del rumore. L'autoencoder concettualmente più semplice (*vanilla autoencoder*), contiene un layer di input formato n unità, dove n è lo spazio originale dell'input, un hidden layer con $m < n$ unità, che rappresenta la componente *encoder*, ed infine un output layer con n unità che rappresenta la componente decoder e si occupa di ricostruire l'input.

2.7.1 Deep autoencoder

Anziché dedicare un solo layer ad ambo le componenti dell'autoencoder, è possibile costruire una deep neural network e dedicare più hidden layer all'estrazione delle feature dai dati in input. Questo tipo di architettura prende il nome di **deep autoencoder**. Attraversando la rete i dati verranno compressi in dimensioni progressivamente minori, sino ad arrivare all'ultimo layer dedicato alla componente encoder. Nei layer successivi, appartenenti al decoder, la dimensione dei layer aumenterà sino a ritornare alla dimensione di

input. Grazie alle capacità di estrazione di feature intermedie dai dati, dimostriamo empiricamente che un autoencoder multistrato fornisce risultati migliori rispetto ad un vanilla autoencoder.

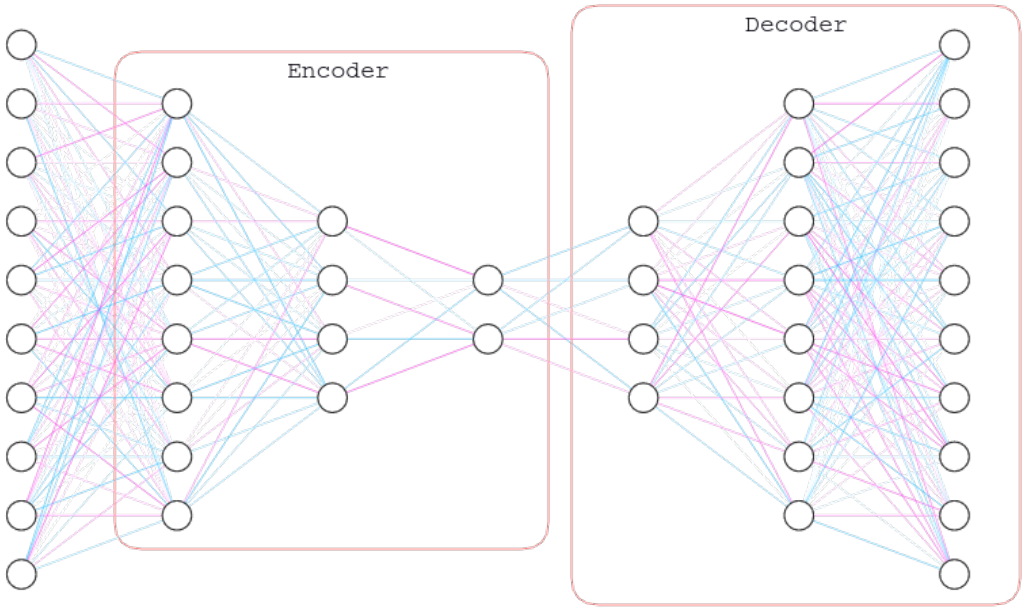

Figura 2.4: Deep autoencoder.

2.8 Alberi decisionali

Gli alberi decisionali sono uno strumento noto nei campi del machine learning, data mining e della statistica. Ogni nodo interno dell'albero contiene un test condizionale su uno o più attributi, tale test stabilisce quale sottoalbero visitare. Se ogni nodo valuta una sola feature, allora si parla di *alberi univariati*, se una condizione combina più feature, allora si parla di *alberi multivariati*. Ogni foglia contiene una *etichetta di classe*. Alcune importanti osservazioni:

- La classe di una osservazione si ottiene seguendo il percorso che va dalla radice dell'albero ad una foglia che determina la classe, sulla base dei test residenti sui nodi interni.
- Ad ogni nodo interno X è possibile associare l'insieme S_X delle tuple che soddisfano le condizioni valutate, partendo dalla radice sino ad arrivare al nodo X.

2.9 Ensemble learning

L'apprendimento ensemble (*ensemble learning*) combina due o più modelli di apprendimento al fine di ottenere migliori performance di predizione rispetto a modelli presi singolarmente. In altre parole, vanno combinate ipotesi multiple al fine di ottenere una migliore ipotesi predittiva. L'ensemble learning richiede molta più computazione, quindi ha senso solo se è usato per combinare algoritmi di apprendimento veloci (es. alberi decisionali), che non hanno elevata accuratezza se presi singolarmente. Il problema principale consiste nell'addestrare i classificatori e nel combinarne i risultati.

2.9.1 Bootstrap

Una tecnica comune di ensemble learning è il **bootstrap** (o bagging). Si prendono in considerazione k modelli deboli e si divide il training set in k sottoinsiemi mediante un campionamento casuale. Ogni modello viene allenato con una porzione di training set differente. Nel caso della regressione, la variabile target si predice effettuando una media dei risultati dei k predittori deboli, mentre nel caso della classificazione vince l'etichetta di maggioranza. Un esempio pratico è la **Random Forest**, che utilizza gli alberi decisionali come modelli deboli ed introduce il concetto di bagging sugli attributi, quindi una selezione parziale degli attributi su ogni predittore.

2.9.2 Gradient boosting

Il **boosting** è una tecnica ensemble, in cui i vari predittori sono creati in maniera sequenziale e dipendente. L'idea chiave è quella di far imparare il predittore successivo dagli errori del predittore corrente. Il **Gradient Boosting** è una tecnica di boosting. L'algoritmo XGBoost (*eXtreme Gradient Boosting*) [6] è una implementazione efficiente del Gradient Boosting. Supponiamo di avere una funzione loss J, vogliamo eseguire delle predizioni che rendano J minima. Consideriamo come modelli base degli alberi decisionali. Supponiamo di trovarci all'i-esima iterazione dell'algoritmo e di aver già costruito l'albero decisionale T_i. Effettuiamo la predizione della variabile target per ogni osservazione, sfruttando gli alberi $T_1, T_2, \ldots, T_i$. Calcoliamo il residuo di ogni predizione come la differenza tra il valore reale e quello predetto. All'iterazione $i + 1$ si costruisce un albero decisionale T_{i+1} che approssimi una ricostruzione dei residui appena ottenuti. In questo modo, l'albero T_{i+1} cercherà di sopperire agli errori di predizione degli alberi precedenti.

Capitolo 3

Preprocessing dei dati

3.1 Analisi del dataset

Sono stati utilizzati due dataset che descrivono il profilo dell'espressione genica di campioni appartenenti ai pazienti affetti da tumore al seno. I dati del primo dataset sono estratti attraverso la tecnica RNA-Seq, descritta brevemente nel paragrafo 2.1.1, mentre il secondo dataset contiene dati estratti attraverso Microarray, descritto nel paragrafo 2.1.2. I dati saranno analizzati attraverso la libreria Pandas [7] per python e, inoltre, verrà incluso il codice necessario a riprodurre l'analisi. Nel prossimo paragrafo ci soffermeremo sul formato dei dati.

3.1.1 Formato dei dati

In entrambi i dataset, i dati sono serializzati in formato tsv (tab separated values). Ogni riga della matrice referenzia un gene. La prima colonna riporta lo HUGO symbol, ovvero la nomenclatura HUGO (Human Genome Organisation) assegnata al gene, mentre la seconda colonna riporta l'identificativo del gene nei database NCBI Entrez. La prima riga, escluse le prime due colonne, conterrà gli identificativi dei campioni analizzati. Escludendo le colonne e righe precedenti, l'elemento generico $M_{i,j}$ della matrice conterrà un valore quantitativo e normalizzato appartenente al campione j rispetto al gene i. Due esempi campionati dai dataset reali sono visualizzabili nelle tabelle 3.1 e 3.2.

Tabella 3.1: Campione d'esempio estratto dal dataset RNA-Seq.

Hugo_Symbol	Entrez_Gene_Id	TCGA-3C-AAAU-01	TCGA-3C-AALI-01
AARS	16	2.6487	2.8250
AARS2	57505	1.7437	0.7079
AARSD1	80755	0.0130	-0.3148
AASDH	132949	-1.4142	-2.9724
AASDHPPT	60496	-0.0873	-0.9247
AASS	10157	-6.4356	-5.3694
AATF	26574	2.4122	9.5609
AATK	9625	4.9418	1.6275
ABAT	18	0.2588	-1.3113
ABCA10	10349	-4.5057	-4.3554

Tabella 3.2: Campione d'esempio estratto dal dataset Microarray.

Hugo_Symbol	Entrez_Gene_Id	MB-0362	MB-0346
MARS2	92935	-0.9398	2.1340
MAGEC1	9947	-0.0432	-0.3771
PDIA6	10130	0.7415	1.8047
FLJ41130	401113	NaN	NaN
SLC25A25	114789	0.5859	-0.8084
FGFBP2	83888	0.6062	-0.2120
LRRC23	10233	2.2915	0.1723
PNCK	139728	-0.0308	-0.2286
RPS24	6229	-1.1657	1.2938
NRDE2	55051	0.9131	0.6910

3.1.2 Dataset RNA-Seq

I file il cui prefisso è "data_RNA_Seq_v2_mRNA" contengono i dati molecolari. Ogni file contiene circa 1080 record e applica una diversa normalizzazione (es. z-scores). L'unico file esente da normalizzazione è denominato "raw_counts". Empiricamente si osserva che il file "median all sample ref normal Zscores" risponde meglio alla compressione attraverso autoencoder. Con il termine "dataset RNA_Seq" ci riferiremo a quest'ultimo. Il modulo python "ngs" conterrà alcune funzioni per manipolare i dataset RNA-Seq. Le prime due funzioni necessarie alla fase di preprocessing sono: "get_data_sources" e "get_ds" (codice A.1 dell'appendice). Servono rispettivamente ad indicizzare tutti i file contenenti i dati RNA-Seq e ad estrarne il contenuto già preprocessato sotto forma di dataframe Pandas.

3.1.3 Dataset Microarray

I file il cui prefisso è "data_mRNA" contengono dati molecolari. Anche in questo caso, ad ognuno dei file è applicata una diversa normalizzazione. La numerosità del dataset ammonta a circa 1900 record. In questo caso, si è osservata una migliore responsività alla compressione utilizzando il file "median all sample Zscores", per cui quest'ultimo verrà referenziato quando si parlerà di dataset Microarray. Come fatto per i dataset RNA-Seq, creiamo un modulo python "microarray" che contenga le stesse funzioni (codice A.2 dell'appendice).

3.2 Ottenere un dataframe

Vogliamo lavorare su un dataset le cui righe corrispondano alle osservazioni (campioni) e colonne corrispondano alle feature (geni). Scegliamo un identificativo tra lo HUGO symbol e l'Entrez Gene ID per distinguere i geni. Per il dataset RNA_Seq è stato scelto l'Entrez Gene ID a causa di alcuni geni per cui lo HUGO symbol risulta mancante. Al contrario, per il dataset Microarray è stato scelto lo HUGO symbol come identificativo, essendo sempre definito.

3.2.1 Preprocessing

Nei moduli ngs e microarray la funzione "get_ds" passa il dataframe importato da file tsv ad una funzione chiamata "adapt". Tale funzione è definita nel modulo preprocess (codice A.3 dell'appendice) e, oltre a prendere in input il

dataframe, richiede un parametro "column", ovvero una stringa che indichi la colonna che fungerà da identificativo per i geni. La pipeline è la seguente:

- conserva gli identificativi dei geni (in base al parametro *column*)
- traspone la matrice
- sostituisce i valori NaN con lo 0
- rimuove le colonne contenenti gli identificativi dei geni dal dataframe
- inserisce gli identificativi dei geni come colonne del dataframe
- rimuove le colonne con valore costante (se non è indicato diversamente)

3.2.2 Dataframe risultante

La pipeline ritorna un dataframe pronto per essere processato. La trasposizione, oltre a scambiare le righe con le colonne, trasporta i nomi delle colonne in indici delle righe. Ogni riga del nuovo dataframe avrà come rispettivo indice l'identificativo del campione. Ad ogni colonna corrisponderà invece l'identificativo del gene. Possiamo visualizzare un esempio nelle tabelle 3.3 e 3.4. Le colonne costanti non costituiscono informazione utile, per cui vengono rimosse.

Tabella 3.3: Dataframe RNA-Seq dopo l'applicazione della pipeline.

Entrez_Gene_Id	100133144	100134869	10357	10431
TCGA-3C-AAAU-01	1.0625	1.4054	-2.6318	-2.3685
TCGA-3C-AALI-01	0.3242	1.8883	-1.1750	-0.9363
TCGA-3C-AALJ-01	0.6145	0.8671	2.8280	2.9686
TCGA-3C-AALK-01	0.6655	1.1569	2.4738	0.9625
TCGA-4H-AAAK-01	-0.0534	1.5834	-0.2282	0.4232
TCGA-5L-AAT0-01	-0.6896	1.4882	1.3096	0.6562
TCGA-5T-A9QA-01	-1.6804	0.8268	1.0050	3.1589
TCGA-A1-A0SB-01	0.9771	1.5793	1.4029	-0.2573
TCGA-A1-A0SD-01	0.3582	1.1918	-1.9085	1.5036
TCGA-A1-A0SE-01	0.4259	-0.2327	2.7901	1.9615

Tabella 3.4: Dataframe Microarray dopo l'applicazione della pipeline.

Hugo_Symbol	RERE	RNF165	CD049690	BC033982	PHF7
MB-0362	-0.7082	-0.4419	0.2236	-2.1485	-0.322
MB-0346	1.2179	0.414	0.2255	0.4763	-1.0921
MB-0386	0.0168	-0.6843	0.5691	-0.2446	0.283
MB-0574	-0.4248	-1.1139	0.3545	0.2618	-0.2864
MB-0503	0.4916	-0.6875	0.7865	-0.2695	0.0772
MB-0641	0.5156	-0.2522	-0.3715	-0.8391	-0.4976
MB-0201	-1.2105	-0.4124	1.9356	-0.677	-0.6453
MB-0218	-0.9309	-0.0023	-0.1612	0.9853	-0.0506
MB-0316	-0.2677	-0.891	1.0461	0.4264	-0.1191
MB-0189	-0.2058	0.1057	-0.0902	0.7944	-1.3048

Capitolo 4

Creazione dell'autoencoder

4.1 Preambolo

Ad ogni campione sono associate circa 20.000 feature geniche. I dati su cui si opera sono quindi ad altissima dimensionalità. A tal proposito si vuole allenare un autoencoder per comprimere tali dati in uno spazio latente (o embedding space) ridotto, mantenendo l'informazione trasportata dai dati. Si visualizzeranno i risultati per 3 diversi autoencoder: il primo codifica i dati in uno spazio latente a 150 dimensioni, il secondo a 50 ed il terzo a 25, quindi una riduzione di 2-3 ordini di grandezza rispetto all'input.

4.2 Ambiente di lavoro

Per la creazione degli autoencoder è stata sfruttata la libreria Keras [8]. Quest'ultima permette di definire in maniera semplice un rete neurale, concatenando layer di vario tipo. Il codice A.4 dell'appendice contiene un esempio di autoencoder formato da 1 solo hidden layer costruito attraverso le classi della libreria Keras. Per facilitare la generazione di diverse architetture di autoencoder è utile definire una funzione che generi un modello attraverso una struttura passata in input. La funzione prende il nome di "generate_deep_autoencoder" e si trova all'interno del modulo "autoencoder". La definizione è consultabile allo snippet di codice A.6 dell'appendice. La funzione prende in input 4 parametri: la dimensione dell'input, le specifiche della componente encoder, le specifiche della componente decoder, le specifiche generiche della rete neurale (loss, algoritmo di ottimizzazione, metriche da monitorare). Il codice A.5 dell'appendice sfrutta tale funzione per costruire un autoencoder con 2 layer per la componente encoder e 2 layer per la componente decoder.

4.3 Struttura dell'autoencoder

L'autoencoder avrà più hidden layer dedicati a ciascuna delle componenti. Ogni layer è connesso in maniera *densa* al layer precedente. Sia n la dimensione dell'input. La componente encoder avrà l layer con $e_1, e_2, \ldots, e_l$ unità, dove $n > e_1 > e_2 > \cdots > e_l$ ed e_l corrisponde alla dimensione dello spazio latente in cui si vogliono codificare i dati. La componente decoder avrà s layer con $d_1, d_2, \ldots, d_s$ unità, dove $e_l < d_1 < d_2 < \cdots < d_s$ e $d_s = n$ sono le unità del layer di output, che dovranno ricostruire l'input.

4.3.1 Funzione di attivazione

Oltre a rendere la fase di training molto più rapida, la funzione di attivazione $ReLU$, di cui abbiamo parlato nelle conoscenze preliminari, risulta essere molto efficace. Anche se utilizzando funzioni come la tangente iperbolica si ottiene pressoché la stessa loss, al momento di convalidare la qualità della compressione attraverso la classificazione del sottotipo di tumore al seno, i dati compressi utilizzando la ReLU ottengono una maggiore accuratezza (circa +10%). Escluso l'output layer, tutti gli altri layer utilizzeranno la funzione di attivazione ReLU. Essendo che i valori di input verranno normalizzati tra $[0, 1]$, allora sarà possibile utilizzare nell'output layer la funzione logistica (con le dovute note, che vedremo in seguito).

4.3.2 Funzione loss e algoritmo di ottimizzazione

Per alleviare l'effetto degli outlier sulla fase di training, la funzione loss selezionata è la Huber loss. Quando la differenza in valore assoluto tra la predizione ed il valore originale è troppo grande, la Huber loss mitiga l'errore non considerando il quadrato. L'algoritmo di ottimizzazione utilizzato è l'Adam, considerato una estensione dell'algoritmo di discesa del gradiente stocastico. Tuttavia, si riscontrano ottimi risultati anche utilizzando l'algoritmo RMSProp.

4.3.3 Stabilità e overfitting

Allo scopo di evitare l'overfitting si utilizza la tecnica Dropout, introdotta nelle conoscenze preliminari. Il Dropout funge da regolarizzazione della rete neurale. Nella pratica, viene inserito un layer speciale fornito da Keras, chiamato Dropout layer, posto dopo il layer di input. Questo layer imposta casualmente una frazione r (dropout rate) di nodi a 0, scalando i rimamenti input di $\frac{1}{(1-r)}$. Come suggerito nell'articolo originale [9], imponiamo che la

il valore massimo della norma dei pesi sia inferiore a 3. Per rendere la fase di training più stabile e più rapida, si applica la Batch Normalization alla fine dei layer, se specificato. Si osservano risultati migliori applicando la Batch Normalization solo ai layer della componente encoder. Nella pratica si pospone un layer speciale di Keras, chiamato BatchNormalization, ai layer sulla quale si vuole applicare la tecnica.

4.3.4 Creazione degli autoencoder

Volendo comprimere i dati in 3 diversi spazi latenti (150, 50 e 25 dimensioni), è utile creare degli autoencoder separati. La differenza tra gli autoencoder sarà nel numero complessivo di layer, mentre le scelte architetturali e implementative resteranno analoghe a come descritte nei paragrafi precedenti. Le strutture degli autoencoder sono consultabili nell'appendice ai codici A.7, A.8 e A.9. Essendo che nessuna conoscenza a priori sul dato in input è utilizzata per la costruzione e l'allenamento dei modelli, spesso ci riferiremo ad essi con il termine "*blind autoencoder*".

4.4 Fase di training

La procedura avviene allo stesso modo sia per il dataset RNA-Seq, che per il dataset Microarray. I risultati verranno confrontati alla fine del paragrafo.

4.4.1 Partizionamento del dataset

Partizioniamo il dataset in 3 insiemi distinti: training-set, test-set e validation-set. Il training set sarà utilizzato nella fase di learning, affiancato dal test-set, che non contribuirà attivamente al learning, ma servirà per monitorare le metriche su dati esterni e prevenire l'overfitting. Il validation set è completamente dissociato dalla fase di learning: il modello non ha alcun contatto con il validation-set se non per validare la sua efficacia, al termine della fase, su dati reali mai visti prima. Perché non utilizzare direttamente il test set, dato che non contribuisce attivamente al learning? Il problema risiede nell'Early Stopping: utilizzando questa tecnica, si ferma l'allenamento quando le metriche migliorano sul training set, ma peggiorano sul test set. Lo stop del training è compiuto sulla base dei dati nel test-set, per cui il modello risulta distorto per tali dati. Gli insiemi nel partizionamento hanno i rispettivi rapporti rispetto alla dimensione totale del dataset: $(81\%, 9\%, 10\%)$ su RNA-Seq e $(72\%, 8\%, 20\%)$ su Microarray (considerando rispettivamente training, test e validation).

4.4.2 Scaling

Prima di iniziare il training, scaliamo i dati nel range $[0, 1]$. Lo scaling utilizzato è il *rescaling* (o *min-max normalization*). Per ogni feature x si trova il valore minimo $\min(x)$ ed il massimo $\max(x)$. Dopodiché si effettua lo scaling nel seguente modo:

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

Il range $[0, 1]$ corrisponde proprio al codominio della funzione logistica nel layer di output. Applichiamo lo scaling al training set ed otteniamo dei minimi e dei massimi. Gli stessi minimi e massimi dovranno essere utilizzati per scalare il test set ed il validation set. Se così non fosse, si introdurrebbe un bias nel processo: immaginiamo di dover comprimere feature relative ad un solo (e nuovo) campione, come effettueremmo lo scaling? Abbiamo bisogno di parametri di riferimento, per cui, a meno di uno studio a priori, è possibile utilizzare i parametri del training set. Una nota dolente di questa tecnica è che, considerando minimi e massimi di un altro insieme di dati, i dati scalati potrebbero uscire dal range $[0, 1]$ e quindi non essere raggiungibili dalla funzione logistica. Tuttavia, allo scopo di veicolare l'autoencoder su una buona compressione dei dati, questo non risulta essere un grosso problema. Nel modulo "utilities" vengono implementate due funzioni, osservabili dal codice A.10 dell'appendice, attraverso la libreria scikit-learn [10]. Esse servono rispettivamente a scalare i dati ex-novo e ad effettuare lo scaling utilizzando dei parametri ottenuti precedentemente.

4.4.3 Fase di training

Oltre a valutare la Huber loss, monitoriamo il Mean Square Error (MSE) ed il Mean Absolute Error (MAE). Scaliamo opportunamente i dati (codice A.11 dell'appendice). Passiamo al modello i dati di training e di test, indicando che le variabili da prevedere sono esattamente le stesse variabili in input, per cui esso deve performare una ricostruzione dei dati. L'avvio della fase di training è mostrato nel codice A.12 dell'appendice.

4.5 Valutazione del training

Il grafico 4.1 mostra l'andamento del training: nelle ascisse troviamo le epoche, nelle ordinate troviamo il valore della Huber loss. Il training degli autoencoder che sfruttano dati RNA-Seq risulta essere più stabile. Tra le 20 e

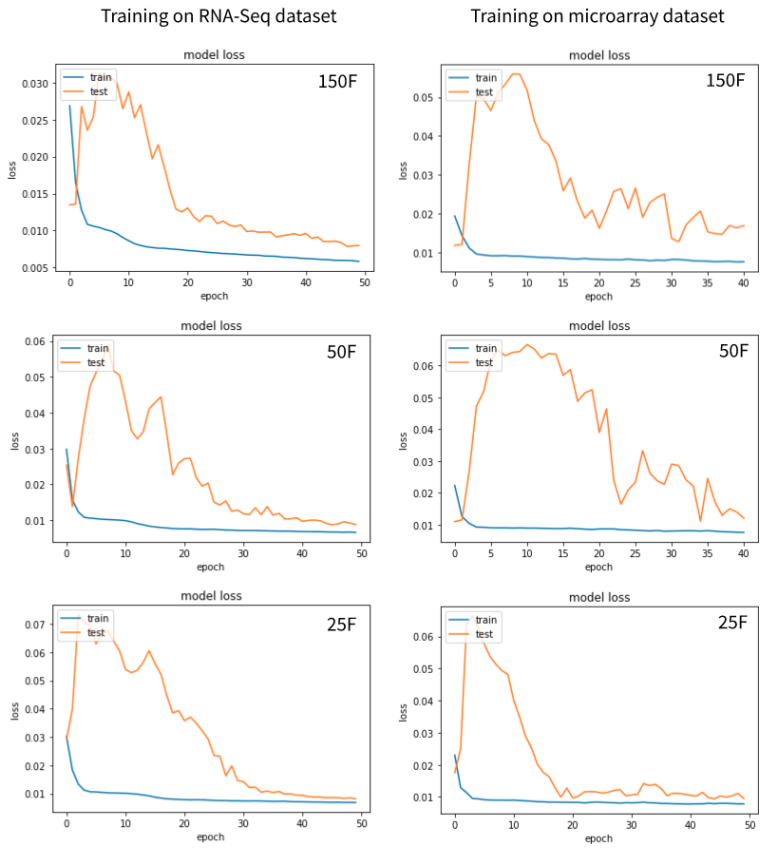

Figura 4.1: Storici della loss durante il training

Tabella 4.1: Metriche di valutazione dei modelli prodotti.

Dataset	RNA-Seq			Microarray		
Latent space	150	50	25	150	50	25
Huber loss	0.0079	0.0084	0.0081	0.0168	0.0134	0.0100
MSE	0.0159	0.0169	0.0164	0.0338	0.0268	0.0200
MAE	0.0936	0.0966	0.0955	0.1399	0.1231	0.1060

le 30 epoche, gli autoencoder mostrano una buona loss sia per i dati di training (curva blu), che per i dati di test (curva gialla). Con molta probabilità, questo fenomeno è indice di un modello ben generalizzato. Gli autoencoder che sfruttano dati Microarray hanno un training instabile: mentre la loss per i dati di training decresce in maniera monotona, la loss calcolata sui dati di test esegue frequentemente dei salti. Molto spesso, il training su questi modelli è fermato dalla tecnica Early Stopping, che vede deteriorare la loss sul test set. L'instabilità sul test set si riflette, con molta probabilità, in un modello che soffre di overfitting. Tuttavia, la loss sul test set decresce sufficientemente rispetto alla fase iniziale del training, per cui il modello risulta essere utilizzabile.

4.5.1 Validazione del modello

Utilizziamo il validation set per calcolare le metriche (MSE, MAE, Huber loss) su dati reali. Dalle osservazioni precedenti, i risultati ottenuti sui dati RNA-Seq ed i risultati ottenuti sui dati Microarray, mostrati nella tabella 4.1, sono pressoché attesi. Considerando che i dati sono scalati in un intervallo $[0, 1]$, abbiamo un errore di ricostruzione medio di 0.09 nel caso di RNA-Seq, e di 0.12 nel caso di Microarray. L'immagine 4.2 presenta delle heatmap, in cui ogni cella indica la differenza in valore assoluto tra il valore originale e la sua ricostruzione. Più la tonalità è violacea, minore è l'errore complessivo di ricostruzione.

4.5.2 Salvataggio dei dataset compressi

Estraiamo la componente encoder da ogni autoencoder attraverso lo snippet di codice A.13. Utilizziamo gli encoder per comprimere entrambi i dataset nei vari embedding space (150, 50 e 25 dimensioni) e salviamoli. I dataset compressi serviranno a svolgere i prossimi task di predizione.

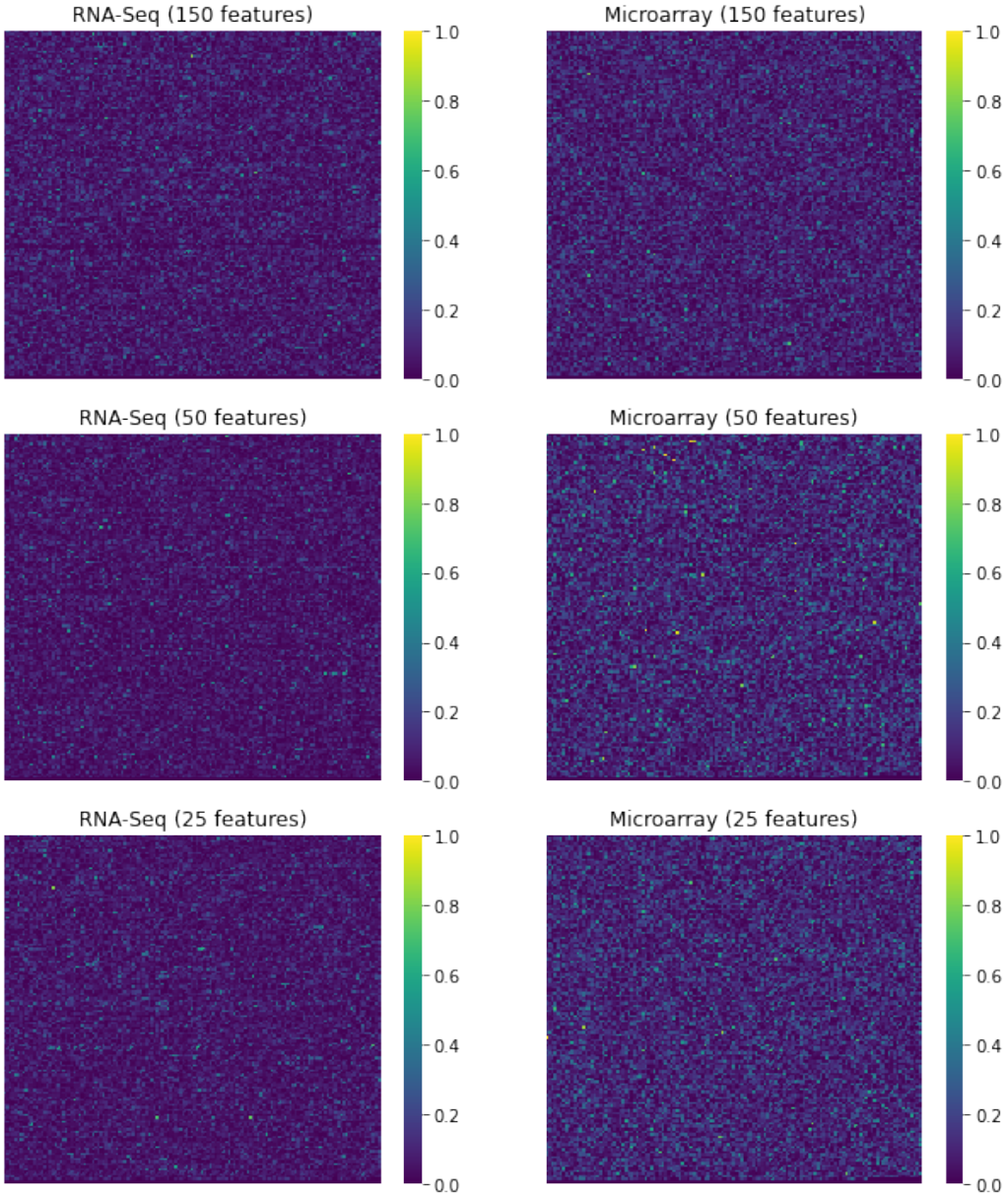

Figura 4.2: Heatmap degli errori di ricostruzione.

Capitolo 5

Predizione del sottotipo

5.1 Introduzione al problema

Il capitolo introduttivo accenna l'importanza della determinazione dei sottotipi di tumore al seno, allo scopo di sviluppare trattamenti mirati. Date delle feature geniche associate ad un campione, vogliamo predire con alta probabilità il sottotipo di cancro al seno. La compressione dei dataset permette di lavorare su uno spazio a più basse dimensioni, riassumendo i pattern assunti dalle cellule.

5.2 Estrazione dei sottotipi

Come visto nel capitolo 3, ad ogni record di feature geniche è associato un identificativo alfanumerico. Tale identificativo corrisponde all'ID assegnato al campione analizzato. Per ognuno dei dataset, sono presenti due ulteriori file: uno contenente informazioni sui campioni, ed uno contenente informazioni e dati clinici sui pazienti. Per ogni paziente possono essere presenti uno o più campioni. Il dataset dei campioni contiene l'ID del rispettivo paziente. Tra i dati clinici del paziente, è presente il sottotipo di tumore al seno: nel dati RNA-Seq, questo corrisponde al campo 'SUBTYPE', mentre nei dati Microarray, il campo è denominato 'CLAUDIN_SUBTYPE'. Per associare le feature geniche al rispettivo sottotipo di tumore è necessario estrarre l'ID del campione; ricercarlo nel dataset dei campioni; estrarre l'ID del paziente; ricercarlo nel dataset dei pazienti ed, infine, estrarre il sottotipo. I moduli "ngs" e "microarray" definiscono entrambi una funzione (codice A.14 e A.15 dell'appendice) chiamata "attach_label", che:

- prende in input un dataframe di feature geniche ed un dato clinico;

- associa ad ogni record il dato clinico, come descritto precedentemente;
- restituisce il dataframe arricchito con il dato.

Questa funzione astrae il processo di incrocio dei dati, ed è utilizzabile per arricchire i dataset compressi con la label da predire, che in questo caso è il sottotipo di tumore al seno. Un output di esempio è visualizzabile nelle tabelle 5.1 e 5.2.

Tabella 5.1: Porzione di dataset RNA-Seq compresso, arricchito di sottotipo

	0	1	...	24	SUBTYPE
TCGA-3C-AAAU-01	-0.6811	-0.7125	...	-0.6127	BRCA_LumA
TCGA-3C-AALI-01	-0.6811	-0.7125	...	8.1006	BRCA_Her2
TCGA-3C-AALJ-01	-0.6811	-0.7125	...	-0.6127	BRCA_LumB
TCGA-3C-AALK-01	-0.6811	-0.7125	...	6.9583	BRCA_LumA
TCGA-4H-AAAK-01	-0.6811	-0.7125	...	1.6100	BRCA_LumA

Tabella 5.2: Porzione di dataset Microarray compresso, arricchito di sottotipo

	0	1	...	24	CLAUDIN_SUBTYPE
MB-0362	2.1400	5.9944	...	13.2636	LumA
MB-0346	-0.6336	8.4901	...	18.3221	Her2
MB-0386	-0.6336	5.0948	...	7.3496	LumA
MB-0574	10.8585	20.1144	...	27.1986	LumA
MB-0503	0.1611	2.0011	...	6.2723	LumA

5.3 Classificazione multiclasse

I sottotipi di tumore al seno sono molteplici, per cui si dovrà performare una classificazione multiclasse. Identifichiamo i principali:

- Luminal A (LumA)

- Luminal B (LumB)
- Her2-enriched (Her2)
- Basal
- Normal-like

Affronteremo di seguito il problema del partizionamento del dataset in un task di classificazione multiclasse.

5.3.1 Partizioni eterogenee

La suddivisione dei dati tra tra i vari insiemi (training set, test set, validation set) va fatta con cautela: in entrambi gli insiemi devono ricadere tutte le classi. Quello che vogliamo ottenere è un *partizionamento eterogeneo*. Per ottenere ciò, è stato implementato un semplice algoritmo che si basa sulla seguente osservazione: supponiamo di avere un insieme S di numerosità n. Supponiamo che questo insieme sia partizionato in m classi $c_1, c_2, \ldots, c_m$ di numerosità $n_1, n_2, \ldots, n_m$ tale che $n_1 + n_2 + \cdots + n_m = n$. Sia $\alpha \in [0, 1]$, vogliamo estrarre un sottoinsieme $S_\alpha \subset S$ eterogeneo di numerosità $n \cdot \alpha$, ovvero un sottoinsieme che contenga elementi di tutte le classi e, per ogni classe c_i, via sia la stessa frequenza relativa $(\frac{n_i}{n})$ presente nell'insieme S di partenza. La strategia di campionamento consiste nel prelevare da ogni classe un numero di elementi pari a:

$$\frac{n_i}{n} \cdot (n \cdot \alpha) = n_i \cdot \alpha \quad \forall i = 1, \ldots, m$$

Si dimostra che la classe i-esima nel sottoinsieme abbia la stessa frequenza relativa della classe i-esima nell'insieme originale:

$$\frac{n_i \cdot \alpha}{n \cdot \alpha} = \frac{n_i}{n} \quad \forall i = 1, \ldots, m$$

Si dimostra che la numerosità del sottoinsieme S_α ottenuto unendo i sottoinsiemi prelevati da ogni classe è pari a $n \cdot \alpha$:

$$n_1\alpha + n_2\alpha + \cdots + n_m\alpha =$$
$$(n_1 + n_2 + \cdots + n_m) \cdot \alpha = n \cdot \alpha$$

L'algoritmo di partizionamento eterogeneo è implementato tramite la funzione *multilabel_train_test_split* (codice A.16 dell'appendice) del modulo utilities.

5.4 Comparazione dei classificatori

Una volta estratto il dataset, arricchito del sottotipo e partizionato in training set, test set e validation set, è il momento di selezionare i modelli che performeranno la classificazione del sottotipo. Il principale candidato è l'algoritmo XGBoost [6], un modello ensemble basato sul gradient boosting (paragrafo 2.9.2). Tuttavia, verranno utilizzati anche modelli alternativi, come Random Forest e Support Vector Machine. I modelli SVM e Random Forest sono forniti dalla libreria scikit-learn, mentre l'algoritmo XGBoost è implementato da una libreria a sé stante. Eseguiamo il training dei classificatori sul training set e validiamoli con il test set ed il validation set.

5.5 Analisi dei risultati

Una prima valutazione dei modelli è data dall'accuratezza, ovvero il rapporto tra il numero di predizioni corrette ed il numero di predizioni totali. L'accuratezza non riesce a rilevare alcune anomalie relative al modello (es. modello costante). A tale scopo, è necessario consultare la matrice di confusione. Per definizione, una matrice di confusione C è una matrice tale che l'elemento $C_{i,j}$ corrisponde al numero di record di classe i, predetti come classe j. Per ottenere una matrice di confusione *binaria* per ogni sottotipo, è sufficiente adottare una tecnica *one vs all*, considerando un task di classificazione binaria in cui si ricerca la presenza dello specifico sottotipo. In questo modo, calcolando metriche come la specificity o la sensitivity, si ha una panoramica più dettagliata del comportamento del classificatore in relazione ai singoli sottotipi. I risultati differiscono a seconda del dataset, per cui verranno analizzati singolarmente. Si noti che i risultati ottenuti nelle tabelle 5.3 e 5.4 sono frutto di esperimenti a sé stanti, effettuati considerando solo i sottotipi principali (Luminal A, Luminal B, Basal, Her2).

5.5.1 Risultati su dataset RNA-Seq

Tutti i modelli allenati su dati RNA-Seq ottengono una accuratezza intorno all' 82% sui set di validazione. I modelli prodotti risultano essere abbastanza generici. La matrice di confusione nell'immagine 5.1 mostra un comportamento analogo per tutti i modelli, in tutti gli embedding space. La diagonale mostra le predizioni corrette, ed ha valori molto più alti rispetto agli altri elementi, questo giustifica l'alta accuratezza dei modelli. L'errore più frequente avviene nei campioni affetti da tumore Luminal B, che viene predetto circa il 25% delle volte in maniera errata come Luminal A. Una ulteriore osservazio-

ne è la mancata rilevazione di tumori di tipo Normal-Like che, oltre ad essere rari, hanno pattern difficili da scovare rispetto alle altre classi. La tabella 5.3 contiene i valori di specificity, sensitiviy, precision e negative predictive value (NPV) calcolati sui sottotipi principali, su tutti gli embedding space, utilizzando come riferimento il modello XGBoost. Considerando accettabili i valori > 0.7, la sensitivity del sottotipo Her2 è frequentemente sotto la soglia, mentre tutti gli altri valori risultano essere ottimali. Una sensitivity bassa indica difficoltà di rilevazione del sottotipo.

Embedding space	**Metrics**	**LumA**	**LumB**	**Basal**	**Her2**
150	Sensitivity	0.94	0.72	0.97	0.44
	Specificity	0.89	0.91	0.99	0.99
	Precision	0.90	0.67	0.97	0.78
	NPV	0.93	0.93	0.99	0.95
50	Sensitivity	0.91	0.69	0.97	0.38
	Specificity	0.83	0.91	0.99	0.98
	Precision	0.86	0.68	0.97	0.67
	NPV	0.89	0.92	0.99	0.94
25	Sensitivity	0.90	0.54	0.91	0.38
	Specificity	0.83	0.89	0.99	0.96
	Precision	0.86	0.55	0.94	0.46
	NPV	0.88	0.88	0.98	0.94

Tabella 5.3: Metriche sulla matrice di confusione (dataset RNA-Seq).

5.5.2 Risultati su dataset Microarray

I modelli allenati su dati Microarray ottengono una accuratezza intorno al 60% sui set di validazione. Importante osservare che, rispetto ai dati RNA-Seq, i dati Microarray hanno una minore precisione. Osservando la matrice di confusione nell'immagine 5.2, notiamo che nei dati compressi in spazi latenti

a 150 e 50 dimensioni, gli errori principali sono dati dalla mancata distinzione delle classi Luminal A, Luminal B ed Her2, confusione di magnitudo maggiore rispetto a quella osservata nei dataset RNA-Seq. Anche qui, la classe Normal-Like non è rilevata correttamente. La tabella 5.4 contiene i valori di specificity, sensitiviy, precision e negative predictive value (NPV) calcolati sui sottotipi principali, su tutti gli embedding space, utilizzando come riferimento il modello XGBoost. Considerando accettabili i valori > 0.7, i problemi più frequenti sono una bassa sensitivity per la classe Luminal A ed una bassa precision per le classi Luminal B ed Her2. Mentre la sensitivity bassa indica una debole capacità di rilevazione del sottotipo, una precision bassa indica una maggiore probabilità di ottenere falsi positivi.

Embedding space	**Metrics**	**LumA**	**LumB**	**Basal**	**Her2**
150	Sensitivity	0.60	0.72	0.92	0.77
	Specificity	0.95	0.78	0.98	0.89
	Precision	0.90	0.58	0.86	0.53
	NPV	0.76	0.87	0.99	0.96
50	Sensitivity	0.67	0.68	0.95	0.68
	Specificity	0.93	0.82	0.96	0.90
	Precision	0.88	0.62	0.79	0.52
	NPV	0.78	0.86	0.99	0.94
25	Sensitivity	0.66	0.67	0.90	0.68
	Specificity	0.92	0.81	0.97	0.88
	Precision	0.87	0.60	0.84	0.49
	NPV	0.78	0.86	0.99	0.94

Tabella 5.4: Metriche sulla matrice di confusione (dataset Microarray).

5.5.3 Introduzione al biased autoencoder

Mentre i dati RNA-Seq compressi con un autoencoder blind danno ottimi risultati, i dati Microarray risultano essere meno precisi. Allo scopo di migliorare i risultati di classificazione, è possibile sacrificare la "blindness" degli autoencoder, veicolando la compressione sulla base del sottotipo di tumore a cui i record appartengono. Nella pratica, andremo a costruire degli autoencoder biased, con una struttura parzialmente diversa rispetto agli autoencoder visti precedentemente.

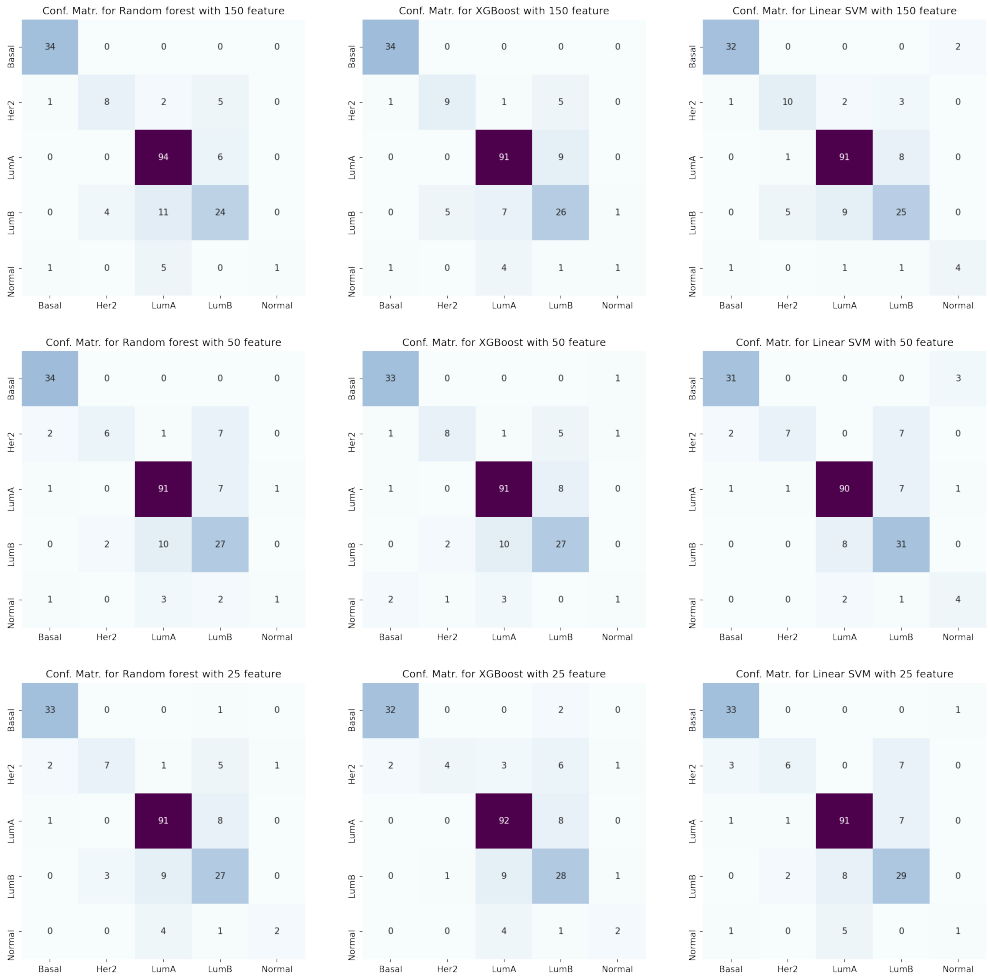

Figura 5.1: RNA-Seq, Matrici di confusione, classificazione del sottotipo.

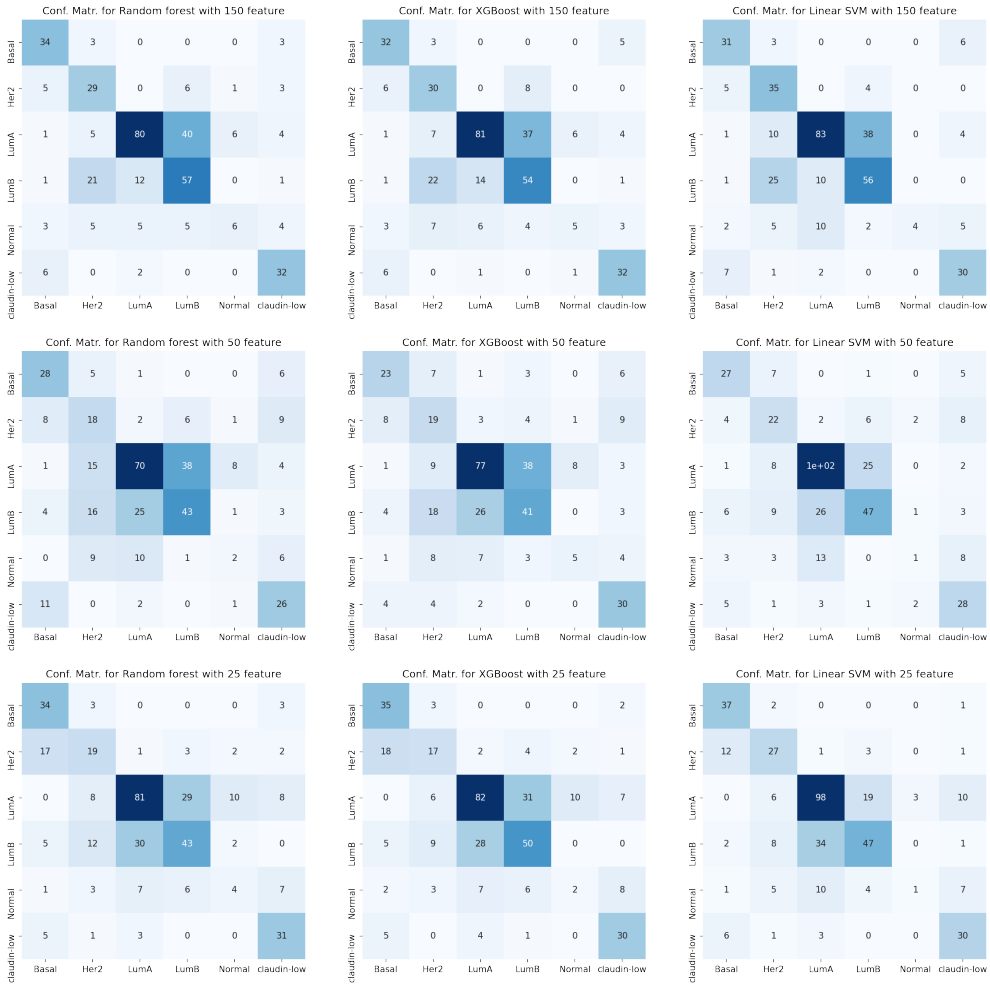

Figura 5.2: Microarray, Matrici di confusione, classificazione del sottotipo.

Capitolo 6

Biased autoencoder

6.1 Perdita della blindness

Una nota positiva dei blind autoencoder è la cosidetta *blindness*: il modello non ha nessuna conoscenza apriori del dato che tratterà. Il learning è guidato unicamente dall'obiettivo di ricostruire il dato in input, per cui nello spazio latente troveremo valori che sintetizzano i pattern salienti dei dati. È possibile aggiungere un bias alla compressione dei dati: sostituiamo la componente decoder, che si occupa della ricostruzione del dato, con un classificatore che prenda in input i dati dallo spazio latente e predica in output il rispettivo sottotipo. Così facendo, la fase di learning sarà guidata dal risultato della classificazione. Lo scopo è quello di individuare dei pattern nello spazio latente che permettano di distinguere al meglio i vari sottotipi di tumore al seno. Ci riferiremo a questo particolare modello con il nome di *biased autoencoder*.

6.2 Creazione del modello

Il biased autoencoder è composto da un encoder ed un classificatore. L'encoder rispetta la descrizione fornita nel paragrafo 4.3, per cui il suo output layer ha e_l unità, che corrispondono alla dimensione dell'embedding space. Il classificatore è una deep neural network che si collega all'output layer dell'encoder, e diminuisce progressivamente il numero di unità per layer sino all'output, che avrà tante unità quanti sono i sottotipi di tumore al seno. Collegando le due componenti, la fase di apprendimento della rete neurale risultante sarà guidata dall'output del classificatore. Essendoci ricondotti ad un task di classificazione, anche la funzione loss deve variare: utilizziamo la Categorical Cross-Entropy loss, ovvero una generalizzazione della Binary Cross-Entropy loss trattata nel paragrafo 2.4.2.4.

6.2.1 Implementazione

All'interno del modulo python "autoencoder" creiamo una nuova funzione chiamata "generate_biased_autoencoder" (codice A.17 dell'appendice). Il bisogno di creare una nuova funzione scaturisce poiché la struttura del modello differisce da un autoencoder classico, tuttavia il funzionamento è analogo alla funzione definita nel paragrafo 4.3.4. Creiamo un solo autoencoder che contenga layer per vari embedding space (150, 50 e 25 unità), la cui struttura è osservabile dal codice A.18 dell'appendice. Alleniamo lo stesso modello sui dati RNA-Seq e sui dati Microarray.

6.2.2 Fase di training e valutazione

Eseguiamo tutti gli step descritti nella sezione 4.4, quindi partizionamento del dataset, rescaling e avvio della fase di apprendimento. Oltre a monitorare la loss, teniamo traccia dell'accuratezza di classificazione del modello, così da poter consultare una metrica più intuitiva. Attraverso i grafici nell'immagine 6.1 osserviamo un andamento del training molto stabile, ed una convergenza intorno alla 20° epoca per i dati RNA-Seq, ed intorno alla 10° epoca per i dati Microarray. Il modello ottiene, sul validation set, una accuratezza dell'80% per i dati RNA-Seq e del 70% per i dati Microarray. Tuttavia, utilizzeremo tale modello esclusivamente per comprimere i dataset nei vari spazi latenti, e proseguiremo alla classificazione del sottotipo con i modelli utilizzati in precedenza.

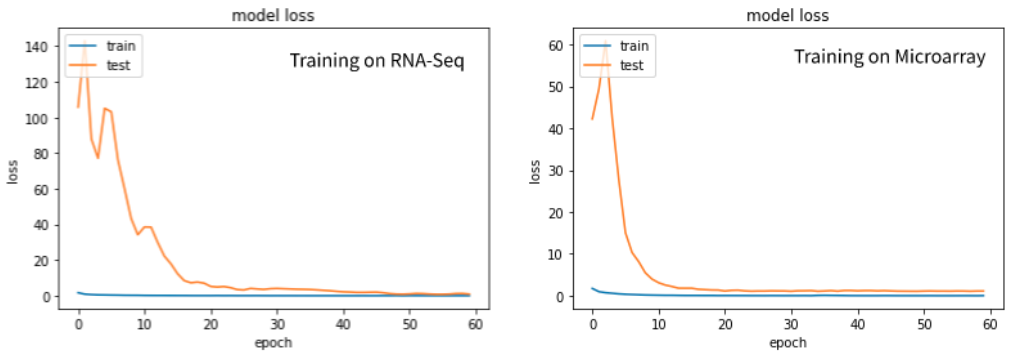

Figura 6.1: Training del biased autoencoder.

6.2.3 Salvataggio dei dataset compressi

Essendovi un layer per ogni spazio latente (150, 50 e 25 dimensioni) è possibile estrarre più componenti encoder dal modello, come fatto nel paragrafo 4.5.2. Attraverso gli encoder comprimiamo interamente i dataset e salviamoli.

6.3 Predizione del sottotipo

Eseguiamo tutti i passaggi illustrati nel capitolo precedente, stavolta utilizzando il dataset ottenuto dal biased autoencoder. Valutiamo singolarmente i risultati ottenuti con il dataset RNA-Seq e quelli ottenuti con il dataset Microarray. Prendiamo come riferimento i modelli precedenti, quindi XGBoost, Random Forest ed SVM. Anche stavolta, i risultati ottenuti nelle tabelle 6.1 e 6.2 sono frutto di esperimenti a sé stanti, effettuati considerando solo i sottotipi principali (Luminal A, Luminal B, Basal, Her2).

6.3.1 Risultati su dataset RNA-Seq

Validando i risultati dei modelli, si osserva immediatamente un guadagno in accuratezza, che in media risulta essere dell'85%. Consultando la matrice di confusione in figura 6.2, notiamo una migliore rilevazione del pattern della classe Normal-Like, prima totalmente anonimo. I risultati forniti dalla tabella 6.1 per i sottotipi principali, sono nettamente ed uniformemente migliori rispetto a quelli precedenti. Inoltre, il problema della bassa sensitivity per il sottotipo Her2 sembra essere stato risolto.

6.3.2 Risultati su dataset Microarray

I risultati sul dataset Microarray descrivono un grosso guadagno in accuratezza: una media del 74%, ovvero un guadagno di +14% rispetto alla classificazione sui dati prodotti dal blind autoencoder. Osservando la matrice di confusione in figura 6.3, notiamo valori più alti nella diagonale ed una forte rilevazione del sottotipo Normal-Like. Sussiste una leggera confusione del modello rispetto ai sottotipi Luminal A e Luminal B. Anche in questo caso, i risultati forniti dalla tabella 6.2 per i sottotipi principali, sono nettamente ed uniformemente migliori rispetto a quelli precedenti. Migliora la sensitivity per la classe Luminal A (circa +20%) e la precision per le classi Luminal B ed Her2 (circa +30%).

Embedding space	Metrics	LumA	LumB	Basal	Her2
150	Sensitivity	0.96	0.90	0.97	0.94
	Specificity	0.98	0.97	1.00	0.98
	Precision	0.98	0.90	1.00	0.79
	NPV	0.96	0.97	0.99	0.99
50	Sensitivity	0.98	0.79	1.00	0.94
	Specificity	0.94	0.99	0.99	0.98
	Precision	0.95	0.94	0.97	0.83
	NPV	0.98	0.95	1.00	0.99
25	Sensitivity	0.98	0.77	1.00	0.88
	Specificity	0.93	0.99	0.99	0.97
	Precision	0.94	0.97	0.97	0.74
	NPV	0.98	0.94	1.00	0.99

Tabella 6.1: Metriche sulla matrice di confusione (dataset RNA-Seq, biased).

Embedding space	Metrics	LumA	LumB	Basal	Her2
150	Sensitivity	0.83	0.89	0.98	0.91
	Specificity	0.97	0.91	0.98	0.97
	Precision	0.96	0.80	0.89	0.83
	NPV	0.88	0.95	1.00	0.98
50	Sensitivity	0.85	0.87	0.98	0.89
	Specificity	0.97	0.91	0.97	0.97
	Precision	0.95	0.81	0.85	0.85
	NPV	0.89	0.94	1.00	0.98
25	Sensitivity	0.87	0.90	1.00	0.89
	Specificity	0.97	0.91	0.98	0.97
	Precision	0.96	0.81	0.87	0.83
	NPV	0.87	0.96	1.00	0.98

Tabella 6.2: Metriche sulla matrice di confusione (dataset Microarray, biased).

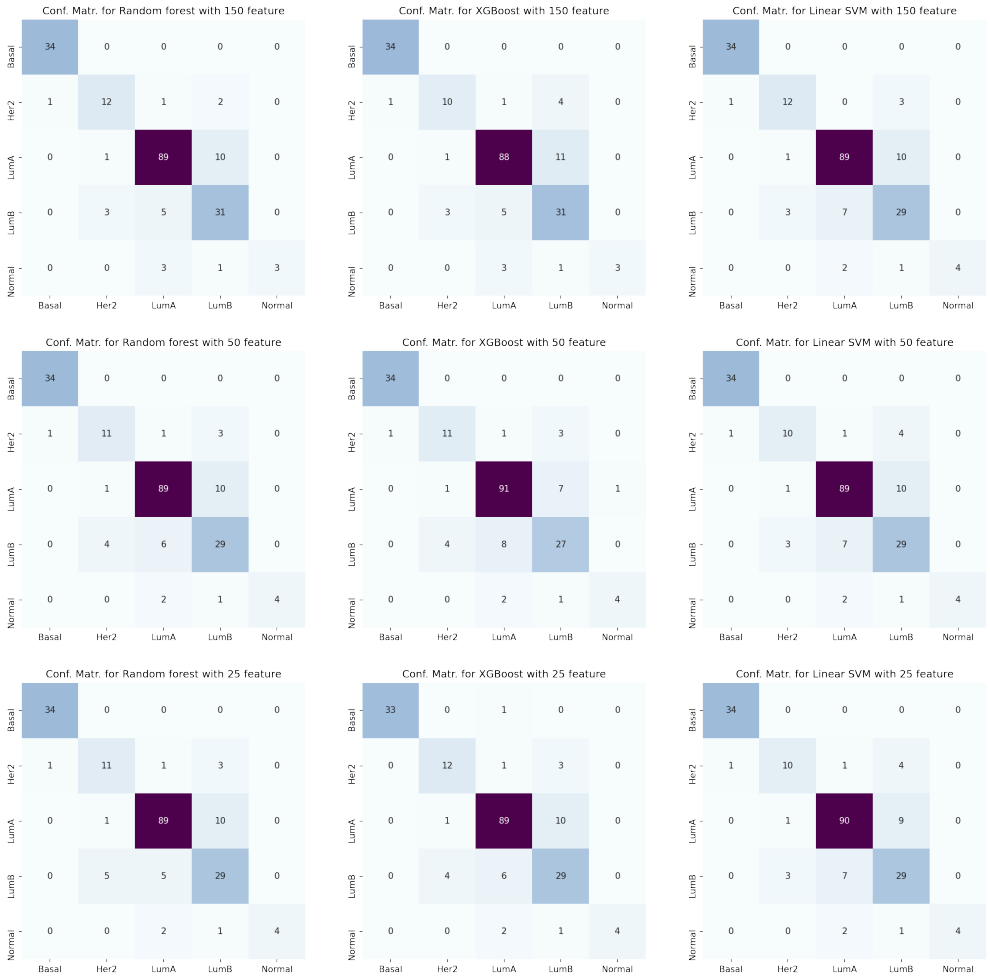

Figura 6.2: RNA-Seq (biased), Matrici di confusione, classificazione del sottotipo.

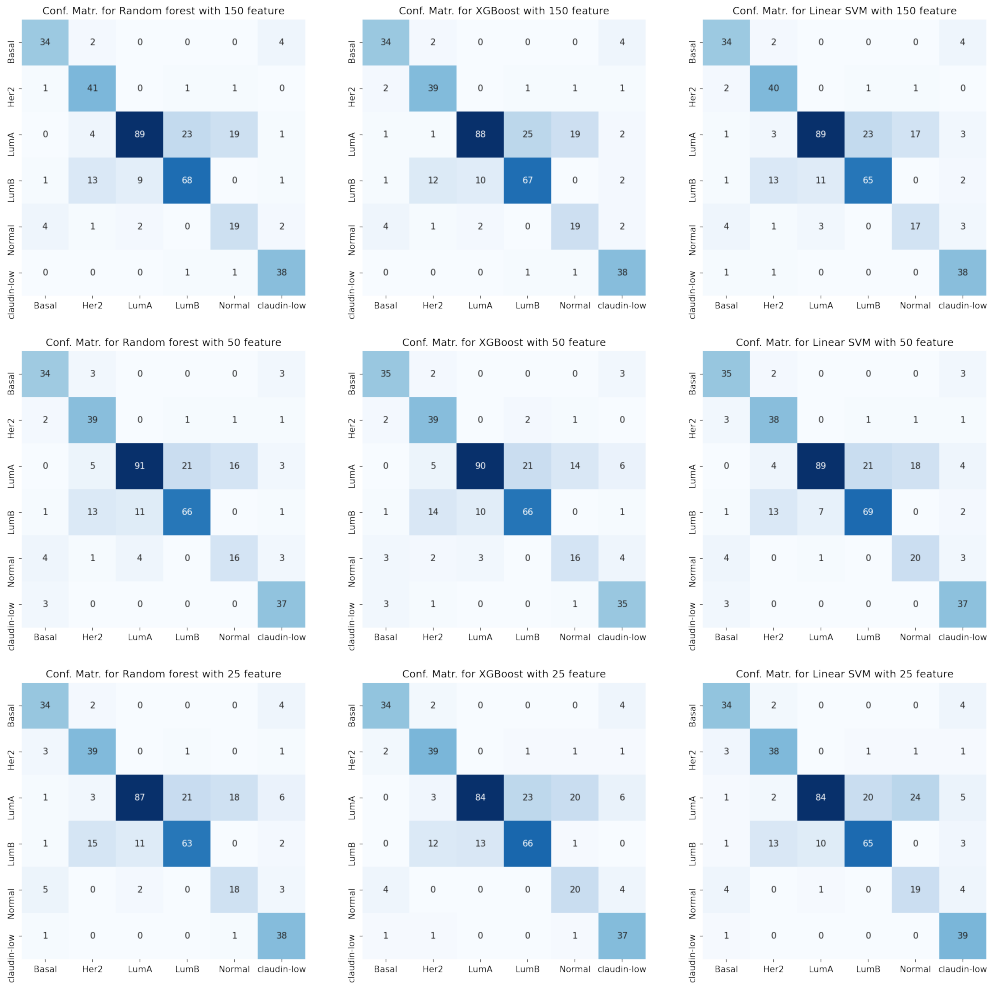

Figura 6.3: Microarray (biased), Matrici di confusione, classificazione del sottotipo.

Capitolo 7

Predizione della sopravvivenza

7.1 Introduzione

Un ulteriore dato clinico legato al paziente è la sopravvivenza (*survival*). Nella sua forma più semplice, essa indica il numero di mesi in cui il paziente sopravvive dopo la diagnosi del tumore, o dopo aver cominciato un trattamento. In questo caso, prende il nome di *Overall Survival*. Nel dataset RNA-Seq, i dati clinici relativi ai pazienti contengono anche altri tipi di survival, come la Disease-Specific Survial (DSS), Disease-Free Survival (DFS) e la Progression-Free Survival (PFS). A partire dai dati molecolari compressi attraverso gli autoencoder blind, vogliamo predire i valori di Overall Survival legati ai pazienti ammalati di tumore al seno.

7.1.1 Estrazione del dato

Sia nel dataset RNA-Seq, che nel dataset Microarray, la Overall Survival è contrassegnata con il nome "OS_MONTHS" (Oversall Survival Months). Utilizziamo la funzione "attach_label" introdotta nella sezione 5.2 per estrarre la sopravvivenza. Arrotondiamo i valori all'intero più vicino, sacrificando la precisione allo scopo di migliorare le performance dei modelli statistici.

7.2 Analisi del dato

Prima di procedere con una analisi di regressione, analizziamo il dato per capire se vi è una relazione con le feature estratte dai dati molecolari.

7.2.1 Analisi della covarianza

Utilizziamo i dataset compressi in uno spazio latente di dimensione 25. Calcoliamo la covarianza in valore assoluto tra ogni singola feature nello spazio latente e la Overall Survival y. Ordiniamo i risultati in ordine decrescente, quindi $f_1, f_2, \ldots, f_{25}$ dove $|Cov(f_1, y)| \geq |Cov(f_2, y)| \geq \cdots \geq |Cov(f_{25}, y)|$. Selezioniamo le prime tre feature f_1, f_2, f_3 più correlate, positivamente o negativamente, alla Overall Survival. Per ogni feature f_i selezionata, il grafico di dispersione in figura 7.2 mostra sull'asse x la feature f_i e sull'asse y la Overall Survival. I punti nel grafico non seguono un andamento ben preciso, per cui ci aspettiamo un errore medio di predizione discretamente alto.

7.2.2 Analisi della correlazione

Per ognuna delle 25 feature del dataset calcoliamo l'indice di correlazione di Pearson rispetto alla Overall Survival. Lo scopo è quello di scovare una eventuale relazione di linearità tra le feature e la sopravvivenza. Utilizziamo la mappa di calore in figura 7.1 per visualizzare i valori ottenuti. Sia per i dati RNA-Seq che per i dati Microarray abbiamo valori molto vicini allo 0, il che equivale ad una assenza di correlazione lineare. Questo è un punto a sfavore per i modelli di regressione lineari che dovranno predire il valore di sopravvivenza.

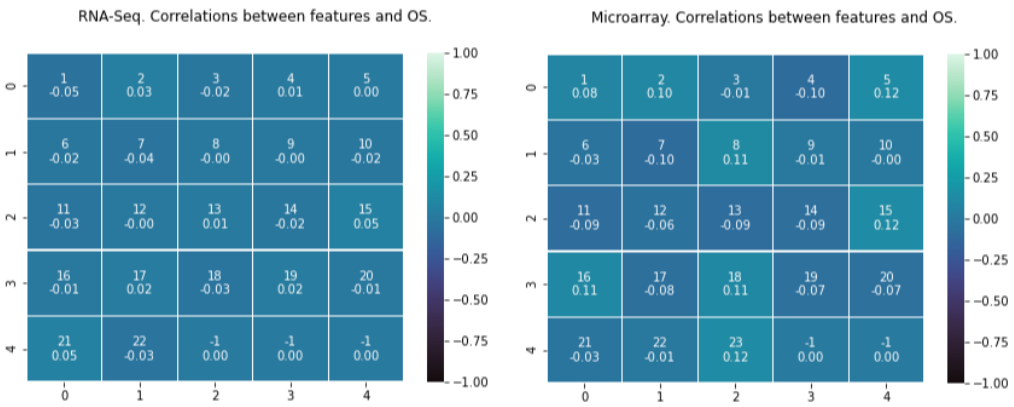

Figura 7.1: Indici di Pearson tra feature dei dataset ed Overall Survival.

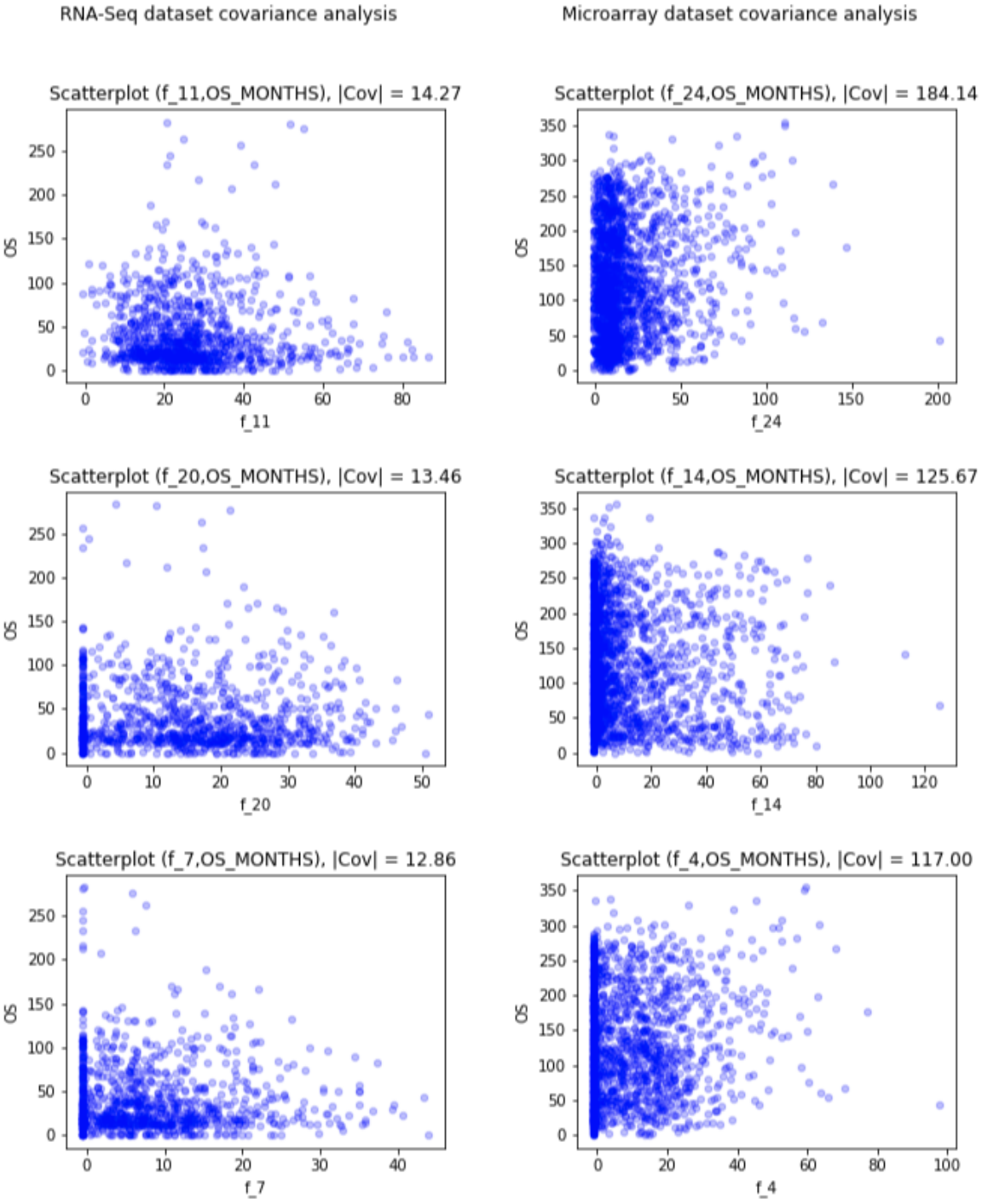

Figura 7.2: Grafico di dispersione tra le feature maggiormente correlate alla sopravvivenza.

7.3 Analisi di regressione

Attraverso la libreria python scikit-learn eseguiamo una analisi di regressione sui dati estratti precedentemente, considerando come variabili indipendenti tutte le feature contenute nei dataset compressi, e come variabile dipendente la Overall Survival. Utilizziamo i seguenti modelli lineari:

- Linear Regression
- Ridge Regression
- Lasso Regression
- Lasso-Lars Regression
- Bayesian Ridge Regression

Ed i seguenti meta-modelli non lineari:

- Gradient Boosting regression
- Random Forest regression

Analizzando le tabelle dei risultati 7.1 e 7.2, notiamo un Negative Mean Absolute Error (NMAE) di circa 30 mesi per i modelli allenati sui dati RNA-Seq, e circa 63 mesi per i modelli allenati su dati Microarray. Le analisi sulla covarianza e sulla correlazione giustificano parte dell'ampio errore ottenuto. Un'altra causa potrebbe essere la qualità dei dati, di cui parleremo nella prossimo paragrafo.

7.3.1 Qualità dei dati

Il dato "OS_MONTHS" estratto da entrambi i dataset RNA-Seq e Microarray indica il numero di mesi dalla diagnosi del tumore (o dal primo trattamento) al suo decesso. Tuttavia, il paziente potrebbe decedere per cause scorrelate al tumore, o decidere di interrompere il collezionamento dei dati inerenti la patologia. Una metrica più accurata è la Disease-Specific Survival, analoga alla Overall Survival, ma misurata solo su pazienti che decedono a causa della malattia. Una accortezza preliminare consiste nel raggruppare i pazienti in base allo stadio tumorale, o ad una serie di fattori che influenzano attivamente la sopravvivenza.

Embedding space	**150f**	**50f**	**25f**
Models	**NMAE**		
Linear regression	-33.71	-30.95	-30.45
Ridge regression	-33.56	-30.94	-30.45
Lasso	-31.33	-30.60	-30.35
Lasso-Lars	-29.92	-29.92	-29.91
Bayesian Ridge	-29.93	-29.96	-29.97
Gradient Boosting	-32.99	-32.66	-32.52
Random Forest	-32.36	-32.73	-32.01

Tabella 7.1: NMAE calcolato su modelli allenati su dati RNA-Seq.

Embedding space	**150f**	**50f**	**25f**
Models	**NMAE**		
Linear regression	-63.88	-64.72	-64.68
Ridge regression	-63.86	-64.15	-64.68
Lasso	-63.38	-64.03	-64.71
Lasso-Lars	-63.53	-64.20	-64.66
Bayesian Ridge	-62.87	-63.94	-64.30
Gradient Boosting	-64.68	-66.20	-66.87
Random Forest	-63.86	-65.98	-66.97

Tabella 7.2: NMAE calcolato su modelli allenati su dati Microarray.

Conclusione

Affinché la riduzione della dimensionalità sia efficace, le feature associate al dato devono essere correlate tra loro, quindi formare una struttura in uno spazio a più bassa dimensione. Se le feature sono legate da una dipendenza non lineare, allora algoritmi come la PCA non riescono a codificare in maniera effettiva le informazioni in uno spazio latente. Da questa trattazione emerge che i deep autoencoder sono particolarmente adatti a codificare dipendenze complesse tra le feature dei dati. Negli autoencoder blind, la ricostruzione del dato produce un errore molto basso, e la significatività della riduzione è confermata dalle ottime metriche di predizione del sottotipo di carcinoma al seno. Sacrificando la generalità fornita dal blind autoencoder, è possibile specializzare il modello su una certa proprietà del dato per rilevare pattern più complessi e migliorare le metriche di predizione. Di fatto, il biased autoencoder, specializzato sulla predizione del sottotipo, permette di rilevare nuove classi come la Normal-Like e aumenta ulteriormente le metriche analizzate. Un pregio delle deep neural networks, come il nostro deep autoencoder, è che le performance aumentano proporzionalmente alla quantità di dati. All'aumentare dei dati molecolari forniti dalla sanità, è possibile ottenere modelli via via più precisi, in grado di produrre dati in uno spazio latente che rappresentino una accurata sintesi delle proprietà trasportate dai dati originali.

Appendice A

Codice A.1: Modulo ngs

```
1
2  import pandas as pd
3  from modules.preprocess import adapt
4
5  def get_data_sources():
6      return [...]
7
8  def get_ds(csv_index=1,
9      ds_dir="datasets",
10     delete_const=True):
11     ds_list = get_data_sources()
12     path = f'{ds_dir}/{ds_list[csv_index]}'
13     ds = pd.read_csv(path, sep="\t")
14     return adapt(ds, 'Entrez_Gene_Id", delete_const)
```

Codice A.2: Modulo microarray

```
1
2  import pandas as pd
3  from modules.preprocess import adapt
4
5  def get_data_sources():
6      return [...]
7
8  def get_ds(csv_index=1,
9      ds_dir="datasets",
10     delete_const=True):
11     ds_list = get_data_sources()
12     path = f'{ds_dir}/{ds_list[csv_index]}'
13     ds = pd.read_csv(path, sep="\t")
14     return adapt(ds, 'Hugo_Symbol", delete_const)
```

Codice A.3: Modulo preprocess

```
1
2  def adapt(ds, colname, delete_constants=True):
3      """
4      Apply preprocessing to the dataset
5      """
6      colnames = ds[colname]
7      col_to_drop = ['Hugo_Symbol', 'Entrez_Gene_Id']
8      tmp = ds.transpose().fillna(0).drop(col_to_drop)
9      tmp.columns = colnames
10     if (delete_constants):
11         tmp = delete_constant_cols(tmp)
12     return tmp
13
14
15 def delete_constant_cols(ds):
16     """
17     Delete constant columns in
18     a dataframe
19     """
20     constant_cols = find_constant_cols(ds)
21     return ds.drop(constant_cols, axis=1)
22
23
24 def is_constant(arr):
25     """
26     Return true if the array
27     contains const. values
28     """
29     res = arr[0] == arr
30     return res.all()
31
32
33 def find_constant_cols(ds):
34     """
35     Return a list of the constant
36     columns in a dataframe
37     """
38     constant_cols = []
39     for col in ds.columns:
40         if (is_constant(ds[col].to_numpy())):
41             constant_cols.append(col)
42     return constant_cols
```

Codice A.4: Vanilla autoencoder con Keras

```
# definiamo il layer di input
encoder_input = Input(shape=(100, ))

# definiamo l'hidden layer (encoder)
encoder = Dense(25, activation='relu')(encoder_input)

# definiamo l'output layer (decoder)
decoder = Dense(100, activation='sigmoid')(encoder)

# creiamo il modello passando i layer
autoencoder = Model(encoder_input, decoder)
autoencoder.compile(optimizer='adam', loss='mse')
```

Codice A.5: Creazione di un autoencoder

```
encoder_components = [
    {'latent_space': 50, 'activation': 'relu',
    'batch_norm_output': True},
    {'latent_space': 25, 'activation': 'relu',
    'batch_norm_output': True},
]
decoder_components = [
    {'latent_space': 50,  'activation': 'relu',
    'batch_norm_output': False},
    {'latent_space': 100, 'activation': 'sigmoid',
    'batch_norm_output': False}
]
autoencoder = generate_deep_autoencoder(100,
    encoder_components, decoder_components,
    {'optimizer': 'adam',})
```

Codice A.6: Funzione che generalizza la creazione di un autoencoder

```
def generate_deep_autoencoder(input_dim,
    encoder_components,
    decoder_components,
    options = {}):

    optimizer = options.get('optimizer', 'adam')
    loss_func = options.get('loss_function',\
        Huber(delta=1.0, reduction="auto", name="huber_loss"))
    dropout   = options.get('dropout', True)
    metrics   = options.get('metrics', ['mae', 'mse'])
    # learning_rate = options.get('learning_rate', 10)
    # momentum = options.get('momentum', .9)

    autoenc = inputs = Input(shape=(input_dim, ))
    firstcomp = encoder_components[0]
    if (dropout):
        autoenc = Dropout(rate=.2)(autoenc)
    autoenc = __add_dense_layer(autoenc, firstcomp, dropout)

    for ecomp in encoder_components[1:]:
        autoenc = __add_dense_layer(autoenc, ecomp, dropout)
    for dcomp in decoder_components[0:-1]:
        autoenc = __add_dense_layer(autoenc, dcomp, dropout)

    lastcomp = decoder_components[-1]
    lastcomp['batch_norm_output'] = False # just to make sure
    autoenc = __add_dense_layer(autoenc, lastcomp, False)

    autoencoder = Model(inputs, autoenc)
    autoencoder.compile(optimizer=optimizer, loss=loss_func, metrics=metrics)
    return autoencoder

def _add_dense_layer(model, comp, dropout):
    if (dropout):
        model = Dense(
            comp['latent_space'],
            activation=comp['activation'],
            kernel_constraint=maxnorm(3))(model)
    else:
        model = Dense(
            comp['latent_space'],
            activation=comp['activation'])(model)
    if (comp['batch_norm_output']):
        model = BatchNormalization(axis=1)(model)
    return model
```

Codice A.7: Autoencoder (emb. space 150)

```
ds = ngs.get_ds(csv_index = 1)
_, ny = ds.shape
encoder_components = [
    {'latent_space': 15000, 'activation': 'relu', 'batch_norm_output': True},
    {'latent_space': 7500,  'activation': 'relu', 'batch_norm_output': True},
    {'latent_space': 3000,  'activation': 'relu', 'batch_norm_output': True},
    {'latent_space': 1000,  'activation': 'relu', 'batch_norm_output': True},
    {'latent_space': 150,   'activation': 'relu', 'batch_norm_output': True},
]
decoder_components = [
    {'latent_space': 1000,  'activation': 'relu', 'batch_norm_output': False},
    {'latent_space': 3000,  'activation': 'relu', 'batch_norm_output': False},
    {'latent_space': 7500,  'activation': 'relu', 'batch_norm_output': False},
    {'latent_space': 15000, 'activation': 'relu', 'batch_norm_output': False},
    {'latent_space': ny,    'activation': 'sigmoid', 'batch_norm_output': False}
]
autoencoder = generate_deep_autoencoder(ny, encoder_components, decoder_components)
```

Codice A.8: Autoencoder(emb. space 50)

```
ds = ngs.get_ds(csv_index = 1)
_, ny = ds.shape
encoder_components = [
    {'latent_space': 15000, 'activation': 'relu', 'batch_norm_output': True},
    {'latent_space': 7500,  'activation': 'relu', 'batch_norm_output': True},
    {'latent_space': 3000,  'activation': 'relu', 'batch_norm_output': True},
    {'latent_space': 1000,  'activation': 'relu', 'batch_norm_output': True},
    {'latent_space': 150,   'activation': 'relu', 'batch_norm_output': True},
    {'latent_space': 50,    'activation': 'relu', 'batch_norm_output': True},
]
decoder_components = [
    {'latent_space': 150,   'activation': 'relu', 'batch_norm_output': False},
    {'latent_space': 1000,  'activation': 'relu', 'batch_norm_output': False},
    {'latent_space': 3000,  'activation': 'relu', 'batch_norm_output': False},
    {'latent_space': 7500,  'activation': 'relu', 'batch_norm_output': False},
    {'latent_space': 15000, 'activation': 'relu', 'batch_norm_output': False},
    {'latent_space': ny,    'activation': 'sigmoid', 'batch_norm_output': False}
]
autoencoder = generate_deep_autoencoder(ny, encoder_components, decoder_components)
```

Codice A.9: Autoencoder (emb. space 25)

```
ds = ngs.get_ds(csv_index = 1)
_, ny = ds.shape
encoder_components = [
    {'latent_space': 15000, 'activation': 'relu', 'batch_norm_output': True},
    {'latent_space': 7500,  'activation': 'relu', 'batch_norm_output': True},
    {'latent_space': 3000,  'activation': 'relu', 'batch_norm_output': True},
    {'latent_space': 1000,  'activation': 'relu', 'batch_norm_output': True},
    {'latent_space': 150,   'activation': 'relu', 'batch_norm_output': True},
    {'latent_space': 25,    'activation': 'relu', 'batch_norm_output': True},
]
decoder_components = [
    {'latent_space': 150,   'activation': 'relu', 'batch_norm_output': False},
    {'latent_space': 1000,  'activation': 'relu', 'batch_norm_output': False},
    {'latent_space': 3000,  'activation': 'relu', 'batch_norm_output': False},
    {'latent_space': 7500,  'activation': 'relu', 'batch_norm_output': False},
    {'latent_space': 15000, 'activation': 'relu', 'batch_norm_output': False},
    {'latent_space': ny,    'activation': 'sigmoid', 'batch_norm_output': False}
]
autoencoder = generate_deep_autoencoder(ny, encoder_components, decoder_components)
```

Codice A.10: funzioni di rescaling (min-max norm.)

```
from sklearn.preprocessing import MinMaxScaler

def normalize_sets(train_df, test_df, scaler = MinMaxScaler()):
    """ Normalize train and test set between 0 and 1 """
    # saving indexes and column names
    # before normalization
    tr_colnames = train_df.columns
    tr_rownames = train_df.index
    te_colnames = test_df.columns
    te_rownames = test_df.index
    # normalize
    scaler.fit(X=train_df.to_numpy())
    output_train_df = pd.DataFrame(scaler.transform(train_df.to_numpy()))
    output_test_df  = pd.DataFrame(scaler.transform(test_df.to_numpy()))
    # restoring the prev  columns names
    # and indexes
    output_train_df.columns = tr_colnames
    output_train_df.index = tr_rownames
    output_test_df.columns = te_colnames
    output_test_df.index = te_rownames
    # return sets
    return output_train_df, output_test_df, scaler


def normalize_with_pretrained_scaler(scaler, dataset):
    colnames = dataset.columns
    rownames = dataset.index
    norm_dataset = pd.DataFrame(scaler.transform(dataset.to_numpy()))
    norm_dataset.columns = colnames
    norm_dataset.index = rownames
    return norm_dataset
```

Codice A.11: partizionamento e rescaling dei set

```
x_train,x_valid=train_test_split(ds, test_perc = .1)
x_train,x_test=train_test_split(x_train, test_perc = .1)
x_train,x_test,scaler= normalize_sets(x_train, x_test)
x_valid=normalize_with_pretrained_scaler(scaler, x_valid)
```

Codice A.12: Inizio fase di training

```
from keras.callbacks import EarlyStopping

earlystopping = EarlyStopping(
    monitor='val_loss',
    verbose=1,
    patience=40)

hist = autoencoder.fit(x_train, x_train,
            epochs=50,
            batch_size=256,
            shuffle=True,
            verbose=1,
            callbacks=[earlystopping],
            validation_data=(x_test, x_test))
```

Codice A.13: Estrazione componente encoder

```
def extract_encoder(autoencoder, index):
    """
    Extract the encoder component from an
    autoencoder passing the full trained
    autoencoder and the index of the last
    encoder layer.
    """
    out = autoencoder.layers[index].output
    return Model(inputs=autoencoder.input,
                 outputs=out)
```

Codice A.14: Funzione attach_label (modulo ngs)

```
def attach_label(samples_ds, label):
    clinical_ds = get_clinical_ds()

    # extract the label
    samples_ids = samples_ds.index.to_numpy()
    subtypes_ds = [
        find_patient_clinical_data(sid, [label], clinical_ds)
        for sid in samples_ids
    ]
    subtypes_ds = pd.DataFrame(subtypes_ds, columns=[label])
    subtypes_ds.index = samples_ds.index

    # concat to the genes dataset
    df = pd.concat( [ samples_ds, subtypes_ds[label] ], axis=1)
    rows_without_label = df[label].isnull()
    return df[~rows_without_label]

def get_clinical_ds():
    return pd.read_csv('datasets/ngs/data_clinical_patient.tsv',
        skiprows=4, sep='\t')

def find_patient_clinical_data(sample_id, clinical_features, clinical_ds):
    patient_id = '-'.join(sample_id.split('-')[:3])
    bio = clinical_ds[clinical_ds['PATIENT_ID'] == patient_id].iloc[0]
    return bio[clinical_features]
```

Codice A.15: Funzione attach_label (modulo microarray)

```

def attach_label(samples_ds, label):
    clinical_ds = get_clinical_ds()

    # extract the label
    samples_ids = samples_ds.index.to_numpy()
    subtypes_ds = [
        find_patient_clinical_data(sid, [label], clinical_ds)
        for sid in samples_ids
    ]
    subtypes_ds = pd.DataFrame(subtypes_ds, columns=[label])
    subtypes_ds.index = samples_ds.index

    # concat to the genes dataset
    df = pd.concat( [ samples_ds, subtypes_ds[label] ], axis=1)
    rows_without_label = df[label].isnull()
    return df[~rows_without_label]

def get_clinical_ds():
    return pd.read_csv('datasets/microarray/data_clinical_patient.txt',
        skiprows=4, sep='\t')

def find_patient_clinical_data(sample_id, clinical_features, clinical_ds):
    # in this case patient_id and sample_id are the same
    bio = clinical_ds[clinical_ds['PATIENT_ID'] == sample_id].iloc[0]
    return bio[clinical_features]
```

Codice A.16: Algoritmo di partizionamento eterogeneo

```
def multilabel_train_test_split(ds, label_column, test_perc = .1):
    """
    Returns 2 dataframes, a training set containing
    (100 - test_perc * 100) percent of the dataset
    and a test set containing (test_perc * 100) percent
    of the dataset. This method ensures that every label
    is contained in each set.
    """
    if (test_perc < 0 or test_perc > 1):
        return None, None
    labels = ds[label_column].unique()
    n, _ = ds.shape
    trn_subset = pd.DataFrame(data=None, columns=ds.columns)
    tst_subset = pd.DataFrame(data=None, columns=ds.columns)
    for label in labels:
        label_class_data = ds[ds[label_column] == label]
        label_class_rows = len(label_class_data)
        test_subset_rows = round((label_class_rows / n) * (n * test_perc))
        tst_subset_to_add = label_class_data.iloc[:test_subset_rows]
        trn_subset_to_add = label_class_data.iloc[test_subset_rows:]
        tst_subset = pd.concat([tst_subset, tst_subset_to_add])
        trn_subset = pd.concat([trn_subset, trn_subset_to_add])
    return shuffle(trn_subset), shuffle(tst_subset)
```

Codice A.17: Funzione generatrice per biased autoencoder

```
def generate_biased_autoencoder(input_dim,
    encoder_components, classifier_components, options = {}):

    optimizer = options.get('optimizer', 'adam')
    loss_func = options.get('loss_function', 'categorical_crossentropy')
    dropout   = options.get('dropout', True)
    metrics   = options.get('metrics', ['accuracy'])

    # creating the encoder component of the autoencoder as usually
    autoenc = inputs = Input(shape=(input_dim, ))
    firstcomp = encoder_components[0]
    if (dropout):
        autoenc = Dropout(rate=.2)(autoenc)
    autoenc = __add_dense_layer(autoenc, firstcomp, dropout)
    for ecomp in encoder_components[1:]:
        autoenc = __add_dense_layer(autoenc, ecomp, dropout)
    # replacing the decoder part with a classifier
    for ccomp in classifier_components[0:-1]:
        autoenc = __add_dense_layer(autoenc, ccomp, dropout)

    lastcomp = classifier_components[-1]
    lastcomp['batch_norm_output'] = False # just to make sure
    autoenc = __add_dense_layer(autoenc, lastcomp, False)

    autoencoder = Model(inputs, autoenc)
    autoencoder.compile(optimizer=optimizer, loss=loss_func, metrics=metrics)
    return autoencoder
```

Codice A.18: Generazione biased Autoencoder

```
ds = ngs.get_ds(csv_index = 1)
_, ny = ds.shape

encoder_components = [
    {'latent_space': 15000, 'activation': 'relu', 'batch_norm_output': True},
    {'latent_space': 7500,  'activation': 'relu', 'batch_norm_output': True},
    {'latent_space': 3000,  'activation': 'relu', 'batch_norm_output': True},
    {'latent_space': 1000,  'activation': 'relu', 'batch_norm_output': True},
    {'latent_space': 150,   'activation': 'relu', 'batch_norm_output': True},
]

classifier_components = [
    {'latent_space': 100,      'activation': 'relu',    'batch_norm_output': True},
    {'latent_space': 50,       'activation': 'relu',    'batch_norm_output': True},
    {'latent_space': 25,       'activation': 'relu',    'batch_norm_output': True},
    {'latent_space': nclasses, 'activation': 'softmax', 'batch_norm_output': False}
]

autoencoder = generate_biased_autoencoder(ny,
    classifier_components,
    decoder_components)
```

Bibliografia

[1] Eamonn Keogh and Abdullah Mueen. *Curse of Dimensionality*, pages 314–315. Springer US, Boston, MA, 2017.

[2] Andrzej Maćkiewicz and Waldemar Ratajczak. Principal components analysis (pca). *Computers & Geosciences*, 19(3):303–342, 1993.

[3] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[4] Zhong Wang, Mark Gerstein, and Michael Snyder. Rna-seq: a revolutionary tool for transcriptomics. *Nature reviews. Genetics*, 10(1):57–63, Jan 2009. 19015660[pmid].

[5] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27(3):379–423, 1948.

[6] Tianqi Chen and Carlos Guestrin. Xgboost. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Aug 2016.

[7] The pandas development team. pandas-dev/pandas: Pandas, February 2020.

[8] François Chollet et al. Keras. `https://keras.io`, 2015.

[9] Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[10] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Du-

chesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.