Tecniche di classificazione per il riconoscimento di un tumore al cervello

Documentazione progettuale

Progetto del corso di Intelligenza Artificiale
Università degli Studi di Bergamo

Del Prete Giovanni - 1035205

August 5, 2020

1

Introduzione

In questo documento si espone il progetto sviluppato dal sottoscritto per il corso di Intelligenza Artificiale presso l'Università degli Studi di Bergamo. Il progetto consiste nel confrontare diverse tecniche di classificazione binaria di immagini provenienti da diverse TAC al cervello per il riconoscimento di un tumore.
Si ha a disposizione un dataset in formato csv che racchiude le caratteristiche dimensionali e statistiche delle immagini raccolte dalle TAC.

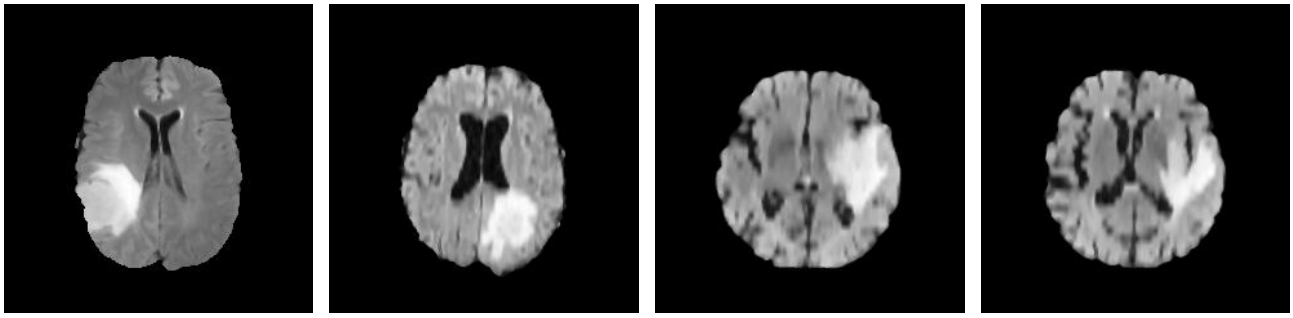


Figure 1.1: Esempio di quattro TAC al cervello

Il file *bt_dataset_t3.csv*, disponibile sul sito *www.kaggle.com*, contiene le seguenti colonne (feature):

- Nome dell'immagine (ImageX, dove X indica il numero della TAC)
- Valori di target (1 se è un tumore, 0 se non è un tumore)
- Media
- Varianza
- Deviazione Standard
- Skewness
- Curtosi
- Contrasto (intensità)
- Energia
- ASM (Angular Second Moment), ovvero un indice che rappresenta l'uniformità del livello di grigio nell'immagine
- Entropia

- Omogeneità
- Dissimilarità
- Correlazione
- Ruvidezza dell'immagine

In pratica le macchie presenti nelle TAC al cervello vengono studiate come distribuzioni statistiche estrapolate da analisi di immagini.
L'obiettivo è quindi utilizzare le tecniche KNN, Regressione Logistica, Kernel SVM e Multi-Layer Perceptron per riconoscere quali macchie possono essere considerate tumori e quali no.
Gli strumenti utilizzati sono diversi moduli Python, come *Pandas*, *Scikit-Learn* e *Matplotlib*.
Le diverse tecniche son state implementate in quattro file di estensione .py per facilità di lettura. In questo documento si illustreranno i punti chiave dei codici.
Prima di procedere con l'implementazione delle tecniche di classificazione qui sopra indicate, è stato necessario fare *preprocessing* sul dataset in quanto vi sono colonne non utili (come il nome delle immagini) e alcuni dati NaN e infiniti che non sono compatibili con gli algoritmi.

2

Data Preprocessing

Il dataset è formato da numeri in formato float, ma in alcuni casi si hanno dei NaN o dei *inf*. Occorre quindi sistemare il dataset.

```
brain_tumor_data = pd.read_csv(r"Data/bt_dataset_t3.csv")

del brain_tumor_data["Image"]
brain_tumor_data = brain_tumor_data.replace(np.inf, 999)
bt_rep = brain_tumor_data.mean()
brain_tumor_data = brain_tumor_data.fillna(bt_rep)

brain_tumor_data.to_csv(r"Data/bt_dataset_t3_fixed.csv")
```

Mediante il modulo Pandas si va a leggere il file csv che costituisce il dataset e lo si trasforma in un *dataframe*, in modo da poter eseguire le operazioni di preprocessing. La trasformazione in dataframe verrà eseguita in ogni algoritmo per poter eseguire le tecniche di classificazione.
Per i dati inf indicano un valori che tendono a infinito. Per mantenere la loro caratteristica si è pensato di sostituire i dati di tipo inf con un valore molto alto, ad esempio 999.

```
brain_tumor_data = brain_tumor_data.replace(np.inf, 999)
```

I dati NaN sono dati che non si hanno in possesso. L'idea è quindi quella di renderli influenti negli algoritmi, perciò può essere una buona idea sostituirli con la media della colonna (feature).

```
bt_rep = brain_tumor_data.mean()
```

Infine con l'istruzione di Pandas *to_csv()* si salvano tali operazioni in un nuovo csv che costituirà il dataset aggiustato. Il nuovo csv verrà utilizzato come input negli algoritmi che implementano le tecniche di classificazione. In questo modo si evitano errori legati all'inconsistenza dei dati.

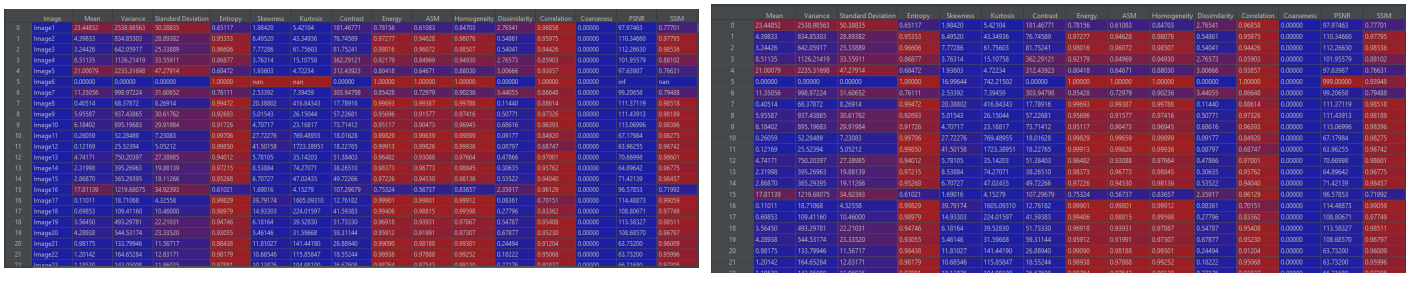


Figure 2.1: A sinistra il dataframe con dati inconsistenti, a desta il dataframe fixato.

L'IDE utilizzato è *PyCharm* e grazie al suo comodo strumento di debugging è stato semplice visualizzare la struttura del dataframe e quindi valutare il corretto funzionamento del codice.

3

Regressione Logistica

3.1 Regressione Logistica con tutte le feature

La prima tecnica implementata per la classificazione binaria è la *Regressione Logistica*.
Come accennato nei capitoli precedenti, l'obiettivo è quello di creare un modello in grado di riconoscere quando un immagine rappresenta un tumore al cervello oppure no con le migliori prestazioni possibili. L'obiettivo di tale tecnica è quella di trovare il *decision boundary* che suddivida nel modo più corretto possibile i target nella classe positiva (1 = è un tumore) e nella classe negativa (0 = non è un tumore). Il primo passo, oltre a quello di leggere il file csv fixato e convertirlo in dataset con Pandas, è quello di portare tutti i dati nella stessa scala di valori. Ciò è utile perchè si hanno dei range di valori molto diversi e l'algoritmo potrebbe dare più importanza ai dati con range più grande. Per questo motivo si è utilizzata la classe di scikit-learn *StandardScaler*. Tale classe permette semplicemente di standardizzare il dataset, ovvero portare tutti i dati in una distribuzione con media 0 e varianza 1. Perciò applicando il modello StandardScaler sui dati si ottiene un dataset con tutti i dati nella stessa scala.
Dal dataset occorre estrarre i *regressori* (matrice X) e i *target* (vettore Y). La standardizzazione ovviamente viene applicata alla matrice X.

```
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

X = brain_tumor_data.drop("Target", axis=1).values
Y = brain_tumor_data["Target"].values
ss = StandardScaler()
X = ss.fit_transform(X)

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.3)
```

Per la validazione del modello si sa che è necessario avere dei dati per il *training* del modello e per il test. Si è ricorso quindi al metodo *train_test_split* di scikit-learn che applica tale divisione. Si è scelto di utilizzare il 70% dei dati per la fase di training e il 30% per la fase di test.

A questo punto si costruisce il modello di Regressione Logistica. Per farlo si utilizza la classe *LogisticRegression* di scikit-learn.

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, log_loss

log_reg = LogisticRegression()
log_reg.fit(X_train, Y_train)
Y_pred = log_reg.predict(X_test)
Y_pred_prob = log_reg.predict_proba(X_test)

acc = accuracy_score(Y_test, Y_pred)
log_loss_score = log_loss(Y_test, Y_pred_prob)
```

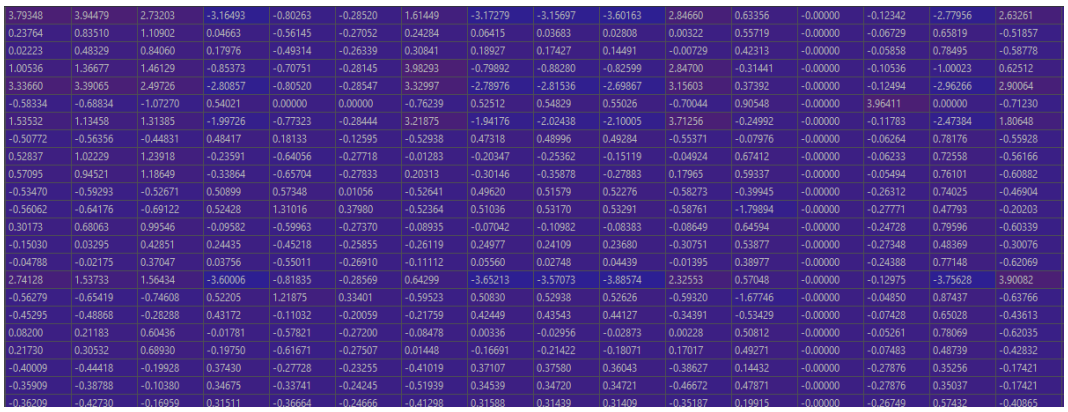


Figure 3.1: una parte della matrice X standardizzata

Con il metodo fit() di LogisticRegression si esegue la fase di training e, ovviamente, in input al metodo si passano i dati di train generati dal metodo *train_test_split*.
Per valutare la bontà del modello si calcolano l' *accuracy* (o accuratezza) e la *Negative Log-likelihood* (o *log loss*). L'accuracy è un indice che va da 0 a 1 che conta quante classificazioni son state fatte correttamente, mentre la log loss tiene conto della probabilità di appartenenza ad una determinata classe. Si punta ad avere un accuracy più alta possibile e una log loss più bassa possibile.

3.2 Regressione Logistica con Dimensionality Reduction

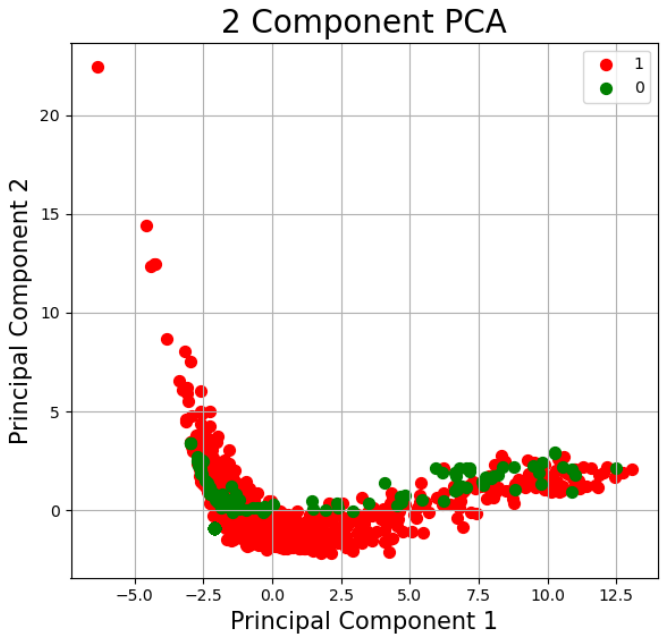
Il dataset è formato da 17 feature. Può essere una buona mossa applicare una tecnica di Dimensionality Reduction per poi fare classificazione?
Per capirlo si è utilizzata la tecnica *PCA* in 2D, 3D e di dimensione tale da spiegare almeno il 95% di varianza dei dati.

```
from sklearn.decomposition import PCA

pca = PCA(n_components=2)
principal_components = pca.fit_transform(X)
principalDF = pd.DataFrame(data=principal_components,
columns=['principal component 1', 'principal component 2'])

finalDf = pd.concat([principalDF, brain_tumor_data[['Target']]], axis=1)

X = finalDf.drop("Target", axis=1)
Y = finalDf["Target"].values

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.3)
```

Si è utilizzata la classe *PCA* di scikit-learn. Tale classe, applicando il metodo *fit_transform()* sulla matrice X, seleziona le componenti principali (2 nel caso di PCA 2D e 3 nel caso di PCA 3D) in una matrice. Per utilizzare i dati nella tecnica di Regressione Logistica è necessario averli in formato dataframe, perciò mediante i metodi di Pandas *DataFrame()* e *concat()* si è costruito il nuovo dataframe con le sole componenti principali. Dopodiché si suddividono i dati di training e di test e si applica la Regressione Logistica nello stesso modo in cui è stato applicato nella sezione precedente.
Uno degli scopi della tecnica PCA è quella della *Data Visualization*. Infatti per le tecniche di PCA in 2D e 3D vi sono porzioni di codice per la proiezione dei dati in 2D e 3D. Per la rappresentazione in 2D è sufficiente costruire uno scatter dove negli assi vi sono le due componenti principali. I target rappresentati possono essere rossi se il valore è 1 (cioè è un tumore) e verdi se il valore è 0 (non è un tumore).

```
fig = plt.figure(figsize=(8, 8))
ax = fig.add_subplot(1, 1, 1)
ax.set_xlabel('Principal Component 1', fontsize=15)
ax.set_ylabel('Principal Component 2', fontsize=15)
ax.set_title('2 Component PCA', fontsize=20)


targets = [1, 0]
colors = ['r', 'g']
for target, color in zip(targets, colors):
indicesToKeep = finalDf['Target'] == target
ax.scatter(finalDf.loc[indicesToKeep, 'principal component 1']
, finalDf.loc[indicesToKeep, 'principal component 2']
, c=color
, s=50)
ax.legend(targets)
ax.grid()
plt.show()
```

Il risultato è il seguente:

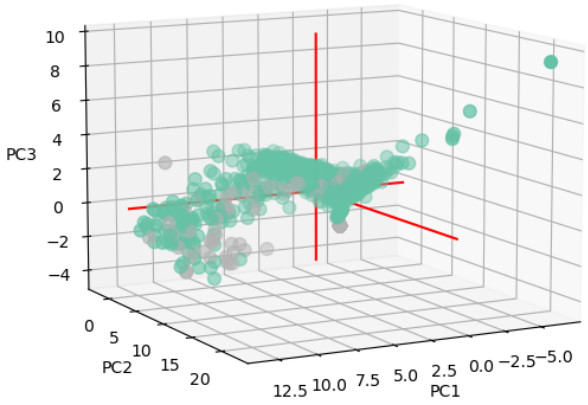


Per quanto riguarda la Data Visualization in 3D si utilizza la classe *Axes3D* del modulo *mpl_toolkits*. La classe Axes3D permette di costruire un diagramma cartesiano con tre assi. Sui tre assi si hanno le tre componenti principali.
L'idea di base è quella di costruire uno scatter (come nel caso precedente) di tre dimensioni mediante matplotlib e passarlo in input alla classe Axes3D che permette la rappresentazione cartesiana.

```
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure()
finalDf['Target'] = pd.Categorical(finalDf['Target'])
my_color = finalDf['Target'].cat.codes
ax = Axes3D(fig)
ax.scatter(finalDf['PCA1'], finalDf['PCA2'], finalDf['PCA3'], c=my_color, cmap="
    Set2_r", s=60)

xAxisLine = ((min(finalDf['PCA1']), max(finalDf['PCA1'])), (0, 0), (0, 0))
ax.plot(xAxisLine[0], xAxisLine[1], xAxisLine[2], 'r')
yAxisLine = ((0, 0), (min(finalDf['PCA2']), max(finalDf['PCA2'])), (0, 0))
```

```
ax.plot(yAxisLine[0], yAxisLine[1], yAxisLine[2], 'r')
zAxisLine = ((0, 0), (0, 0), (min(finalDf['PCA3']), max(finalDf['PCA3'])))
ax.plot(zAxisLine[0], zAxisLine[1], zAxisLine[2], 'r')

ax.set_xlabel("PC1")
ax.set_ylabel("PC2")
ax.set_zlabel("PC3")
ax.set_title("PCA on the iris data set")
plt.show()
```

Il risultato è il seguente:



3.3 Conclusione sulla Regressione Logistica

La regressione logistica utilizzando tutte le feature è quella che dà i risultati migliori, sia in termini di accuracy e di log loss.
Per quanto riguarda l'utilizzo della tecnica PCA, il risultato migliore è dato dalla PCA con parametro il 95% di varianza spiegata. In particolare, applicando dimensionality reduction a due dimensioni con PCA 2D ha peggiorato le prestazioni del modello. Ciò era intuibile guardando il grafico in due dimensioni, dove è difficile trovare un decision boundary lineare che suddivida la classe negativa con quella positiva. La situazione migliora di molto nel caso di PCA 3D, dove le prestazioni sono di poco peggiori a quelle di Regressione Logistica con PCA 95%.

```
======== LOGISTIC REGRESSION WITH ALL FEATURE =========
ACCURACY: 0.9736842105263158
LOG LOSS: 0.0629653063230046

========================================================
========== LOGISTIC REGRESSION WITH PCA 2D ============
ACCURACY: 0.8765182186234818
LOG LOSS: 0.373229693053158

========================================================
========== LOGISTIC REGRESSION WITH PCA 3D ============
ACCURACY: 0.937246963562753
LOG LOSS: 0.2911021573676872

========================================================
========== LOGISTIC REGRESSION WITH PCA .95 ===========
ACCURACY: 0.9412955465587044
LOG LOSS: 0.1795537157456496

========================================================
```

Figure 3.2: Visualizzazione dei risultati stampati sulla console Python.

4

K-Nearest Neighbors

Un'altra tecnica implementata per il problema di classificazione delle immagini delle TAC al cervello è il *K-Nearest Neighbor*. A differenza della Regressione Logistica, KNN si basa sulla classificazione con decision boundary non lineare.
In particolare è importante capire quanti vicini considerare nel modello, ovvero per quale K si ottiene un risultato migliore. La classe di scikit-learn che permette di implementare l'algoritmo KNN è *KNeighborsClassifier*. Tale classe ha come campo principale il campo *n_neighbors* che dà possibilità di settare il K dell'algoritmo. Di default il K è pari a 5.

```
from sklearn.neighbors import KNeighborsClassifier

X = brain_tumor_data.drop("Target", axis=1).values
Y = brain_tumor_data["Target"].values
ss = StandardScaler()
X = ss.fit_transform(X)

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.3)

knn = KNeighborsClassifier()
knn.fit(X_train, Y_train)

Y_pred = knn.predict(X_test)
Y_prob = knn.predict_proba(X_test)

acc = accuracy_score(Y_test, Y_pred)
log_loss_score = log_loss(Y_test, Y_prob)

print("ACCURACY: " + str(acc) + "\n")
print("LOG LOSS: " + str(log_loss_score) + "\n")
```

Il primo tentativo è quello di effettuare la classificazione KNN con K pari al valore di default, ovvero 5. Si ha un accuracy pari a circa 0.95, mentre il log loss è pari a 0.5 e quindi abbastanza alto.
A questo punto si vuole verificare per quale K si ha il risultato migliore in termini di accuracy e log loss. Per fare ciò si implementano più modelli per più K e si visualizzano le metriche su grafici per valutare per quali K il modello ha le migliori prestazioni.
L'idea è quella di utilizzare un ciclo for per costruire un modello con K che va da 1 a 20; *Ks* è un array che contiene i diversi valori di K che ad ogni iterazione viene modificato e di seguito si effettuano le fasi di train e di test. A questo punto si calcolano le metriche, ovvero l'accuracy e log loss.

```
Ks = [1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
accuracys = np.array([])
scores = np.array([])
for K in Ks:
print("============ "+"K="+str(K)+" ============")
knn = KNeighborsClassifier(n_neighbors=K)
knn.fit(X_train, Y_train)

Y_pred_train = knn.predict(X_train)
```

```
Y_prob_train = knn.predict_proba(X_train)

Y_pred = knn.predict(X_test)
Y_prob = knn.predict_proba(X_test)
Y_pred = knn.predict(X_test)
Y_prob = knn.predict_proba(X_test)

acc = accuracy_score(Y_test, Y_pred)
accuracys = np.append(accuracys,acc)
log_loss_score = log_loss(Y_test, Y_prob)
scores = np.append(scores, log_loss_score)

print("ACCURACY: " + str(acc))
print("LOG LOSS: " + str(log_loss_score) + "\n")
print("==========================================")
```

Per il plotting delle metriche in funzione dei diversi K si utilizza semplicemente, ancora una volta, il modulo matplotlib di Python.
Nel passo precedente, ovvero nel ciclo for, si salvano le metriche in due array numpy che verranno utilizzati proprio in questo passo.

```
x_plot = np.array(Ks)
y_plot = accuracys
y_plot_2 = scores

plt.plot(x_plot, y_plot, 'r--', label="Accuracy")
plt.xticks(np.arange(x_plot.min(), x_plot.max(), 1))
plt.legend()
plt.show()

plt.plot(x_plot, y_plot_2, 'bs', label='Log loss score')
plt.xticks(np.arange(x_plot.min(), x_plot.max(), 1))
plt.legend()
plt.show()
```

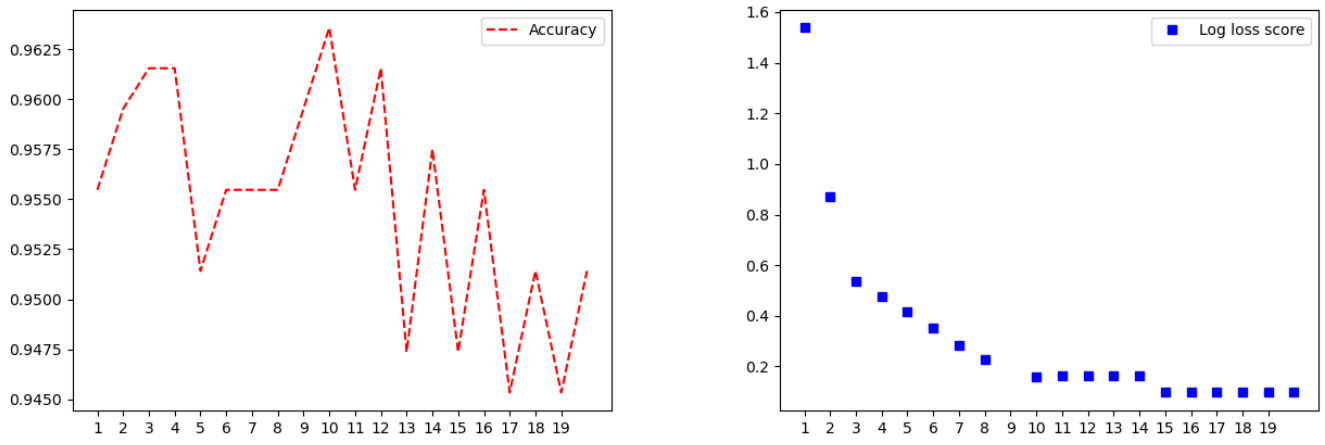


Figure 4.1: A sinistra il plot delle accuracy, a desta quello delle log loss.

In termini di accuracy i valori oscillano tra 0.94 a 0.96 con un andamento piuttosto irregolare. In qualsiasi caso, per K che va da 1 a 20 si ha un buon risultato in termini di accuracy.
Per quanto riguarda la log loss si nota dal grafico un andamento simile ad un ramo di iperbole. Per K molto bassi (ad es. $K = 1$ e $K = 2$) si ha log loss molto alto. All'aumentare del K si ha log loss molto più basso fino a raggiungere un valore limite pari a circa 0.1 per $K > 15$.

4.1 Conclusioni finali sul KNN

Dal punto di vista dell'accuracy utilizzare un K basso o alto non comporta grosse differenze. Il grafico della log loss suggerisce di utilizzare un K alto in quanto si ha un legame decrescente al crescere di K fino a raggiungere il valore limite.
Un buon valore di K per ottenere un ottimo trade off tra accuracy e log loss è 17 (accuracy = 0.96 e logloss = 0.1).

5

Kernel SVM

In questo capitolo si mostra la classificazione mediante *SVM* con kernel trick.
Anceh in questo caso si utilizzano le tecniche di PCA per la dimensionality reduction, come nel caso della regressione logistica.
I kernel utilizzati sono quello lineare, gaussiano e la funzione sigmoide.
Il modello SVM è incluso nella classe *SVC* di scikit-learn.
Un problema notato nei capitoli precedenti è che le due classi non sono molto divisibili da un decision boundary lineare. Infatti ci si aspettano prestazioni migliori utilizzando il kernel gaussiano o con la funzione sigmoide.

```
138 from sklearn.svm import SVC
139
140 pca = PCA(n_components=2)
141
142 principal_components = pca.fit_transform(X)
143 principalDF = pd.DataFrame(data=principal_components,
144 columns=['principal component 1', 'principal component 2'])
145
146 finalDf = pd.concat([principalDF, brain_tumor_data[['Target']]], axis=1)
147
148 X = finalDf.drop("Target", axis=1)
149 Y = finalDf["Target"].values
150
151 X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.3)
152
153 svc = SVC(kernel="linear", probability=True)
154 svc.fit(X_train, Y_train)
155 score_linear2d = svc.score(X_test, Y_test)
156 print("=========== PCA 2D ===========")
157 print("ACCURACY WITH LINEAR KERNEL: " + str(score_linear2d))
158
159 svc = SVC(kernel="rbf", probability=True)
160 svc.fit(X_train, Y_train)
161 score_rbf2d = svc.score(X_test, Y_test)
162 print("ACCURACY WITH GAUSSIAN KERNEL: " + str(score_rbf2d))
163
164 svc = SVC(kernel="sigmoid", probability=True)
165 svc.fit(X_train, Y_train)
166 score_sigmoid2d = svc.score(X_test, Y_test)
167 print("ACCURACY WITH SIGMOID KERNEL: " + str(score_sigmoid2d))
```

Nella classe SVC vengono settati i campi *kernel* e *probability*. Il campo kernel permette di specificare quale tipo di kernel utilizzare per misurare la somiglianza tra due sample. Il campo probability viene settato a *true* per includere il calcolo della probabilità di accuratezza delle predizioni. Ciò rallenta di non poco l'algoritmo ma può essere utile per avere un ulteriore feedback sulle prestazioni del modello.

5.1 Conclusioni su classificazione con SVM

```
========== PCA 2D ==========
ACCURACY WITH LINEAR KERNEL: 0.8866396761133604
ACCURACY WITH GAUSSIAN KERNEL: 0.9534412955465587
ACCURACY WITH SIGMOID KERNEL: 0.8502024291497976
========== PCA 3D ==========
ACCURACY WITH LINEAR KERNEL:0.937246963562753
ACCURACY WITH GAUSSIAN KERNEL:0.9433198380566802
ACCURACY WITH SIGMOID KERNEL:0.9433198380566802
========== PCA .95 ==========
ACCURACY WITH LINEAR KERNEL: 0.9493927125506073
ACCURACY WITH GAUSSIAN KERNEL: 0.9574898785425101
ACCURACY WITH SIGMOID KERNEL: 0.9291497975708503
```

Figure 5.1: Risultati stampati sulla console Python

Come ci si aspettava la tecnica migliore risulta quella ottenuta con kernel gaussiano e con PCA 95%.
Come si può notare però, il contributo maggiore viene fornito dal tipo di kernel. Infatti per il dataset in questione il kernel più performante è quello gaussiano a prescindere dal tipo di PCA utilizzato. Infatti nei tre casi di PCA si nota che le prestazioni del modello SVM con kernel gaussiano cambiano di poco, mentre il tipo di tecnica PCA utilizzata può influire molto se si utilizzano kernel lineari o sigmoidali.

6

Classificazione con Multi-Layer Perceptron

In questo capitolo si utilizza una tecnica basata su rete neurale, ovvero il *Multi-Layer Perceptron*. Mediante scikit-learn è possibile decidere quanti hidden layer inserire e di quali dimensioni. La classe che permette di costruire un MLP è *MLPClassifier*. Inoltre è possibile selezionare il tipo di funzione di attivazione nei neuroni interni:

- *identity*: funzione identità $f(x) = x$;
- *logistic*: funzione sigmoide tipica della regressione logistica $f(x) = \frac{1}{1+e^{-x}}$
- *tanh*: funzione tangente iperbolica $f(x) = \tanh(x)$
- *relu*: funzione lineare $f(x) = \max(0, x)$

```
168 from sklearn.neural_network import MLPClassifier
169
170 X = brain_tumor_data.drop("Target", axis=1).values
171 Y = brain_tumor_data["Target"].values
172 ss = StandardScaler()
173 X = ss.fit_transform(X)
174
175 X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.3)
176
177 mlp = MLPClassifier(hidden_layer_sizes=(100,), activation='relu', verbose=True,
        max_iter=300)
178 mlp.fit(X_train, Y_train)
179
180 y_pred_train = mlp.predict(X_train)
181 y_prob_train = mlp.predict_proba(X_train)
182
183 y_pred = mlp.predict(X_test)
184 y_prob = mlp.predict_proba(X_test)
185
186 accuracy_train = accuracy_score(Y_train, y_pred_train)
187
188 loss_train = log_loss(Y_train, y_prob_train)
189
190 print("ACCURACY: TRAIN=%.4f" % accuracy_train)
191 print("LOG LOSS: TRAIN=%.4f" % loss_train)
```

Il modello adottato è un MLP con un hidden layer formato da 100 neuroni e con funzione di attivazione lineare (relu). I campi *verbose* e *max_iter* sono utili per la fase di addestramento della rete neurale; il primo è utile per visualizzare in maniera dinamica e testuale la fase di addestramento, mentre il secondo permette di inserire il massimo numero di iterazioni nella fase di addestramento. Sperimentalmente il massimo di 300 iterazioni sono risultati sufficienti per raggiungere l'obiettivo.
La fase di addestramento della rete viene iterata finchè non vi sono 10 epoche consecutive dove il loss

score non diminuisce di almeno 10^4. Le metriche calcolate, come nelle altre tecniche, sono l'accuracy e il log loss.

Con questo modello si ha un accuracy pari a circa 0.98 e una log loss di circa 0.04.

```
Iteration 234, loss = 0.04219188
Iteration 235, loss = 0.04179304
Iteration 236, loss = 0.04255709
Iteration 237, loss = 0.04292363
Iteration 238, loss = 0.04230913
Iteration 239, loss = 0.04233116
Iteration 240, loss = 0.04231918
Iteration 241, loss = 0.04185933
Iteration 242, loss = 0.04196515
Iteration 243, loss = 0.04109270
Iteration 244, loss = 0.04151047
Iteration 245, loss = 0.04103798
Iteration 246, loss = 0.04141281
Iteration 247, loss = 0.04094330
Iteration 248, loss = 0.04210819
Iteration 249, loss = 0.04167614
Iteration 250, loss = 0.04129295
Iteration 251, loss = 0.04118068
Iteration 252, loss = 0.04128724
Iteration 253, loss = 0.04189682
Iteration 254, loss = 0.04096718
Training loss did not improve more than tol=0.000100 for 10 consecutive epochs. Stopping.
ACCURACY: TRAIN=0.9843
LOG LOSS: TRAIN=0.0403
```

Figure 6.1: Fase di addestramento e metriche risultanti.

Un altro tentativo è quello di inserire un ulteriore hidden layer di 100 neuroni sempre con funzione di attivazione lineare. Con tale modello si raggiunge la saturazione dell'addestramento (ovvero le 10 epoche consecutive dove il loss score non diminuisce di almeno 10^4) molto prima (123 iterazioni) e si raggiunge un accuracy pari a circa 0.99 e una log loss pari a circa 0.03.
A questo punto si prova ad aggiungere un ulteriore hidden layer. In questo caso l'addestramento si conclude con 81 iterazioni, ma l'accuracy diminuisce a circa 0.98 e la log loss è pari a circa 0.03.
Si prova ora a cambiare la funzione di attivazione settandola come funzione sigmoide. In questo caso, anche con più hidden layer, si ha un accuracy pari a circa a 0.98 e una log loss a circa a 0.05.

6.1 Conclusioni su MLP

In base ai risultati ottenuti, per il dataset a disposizione, il modello con prestazioni migliori è senza dubbio il Multi-Layer Perceptron con due hidden layer di 100 neuroni l'uno e con funzione di attivazione lineare. Una limitazione, non poco importante, è il fatto che la fase di training può occupare molto tempo.

Bibliografia

- Machine Learning con Python: corso pratico di Profession AI
  https://www.udemy.com/course/machine-learning-pratico
  Corso di tutorial iniziale che mi ha fornito gli strumenti di base per svolgere il progetto.
- Documentazione di scikit-learn
  https://scikit-learn.org/stable/index.html
- Slide del corso di Intelligenza Artificiale tenuto dal prof. Francesco Trovò
  https://trovo.faculty.polimi.it/aibg_2018_2019.html