A.A. 2020/2021

# DIGIT RECOGNIZER
# MACHINE LEARNING & DATA MINING

—

**Giovanni Baselli  Matricola 718969**

## INTRODUZIONE

Lo scopo di questo testo è quello di documentare il lavoro svolto nella risoluzione della competizione Kaggle “Digit Recognizer” (https://www.kaggle.com/c/digit-recognizer).

Al seguente link è possibile accedere al file di Google Colaboratory dove è presente il codice, l’implementazione e la valutazione dei modelli utilizzati per la risoluzione:

https://colab.research.google.com/drive/1z2KB3LH2D1siJ0boqWVg9L3FlTvoYqjq?usp=sharing

Questa competizione permette di sfruttare diversi algoritmi di classificazione per il riconoscimento delle immagini, entrando di fatto nel campo della Computer Vision, ovvero nell’ambito della comprensione automatica e nel riconoscimento delle immagini.

**Descrizione dei dati**

Viene fornito un dataset ampiamente utilizzato in Computer Vision, il MNIST ("Modified National Institute of Standards and Technology"), rilasciato nel 1999 e contenente un grosso set di immagini di caratteri scritti a mano. Questo risulta essere un buon dataset per l'applicazione di tecniche di apprendimento e di riconoscimento su dati provenienti dal mondo reale senza spendere troppo tempo nelle fasi di preprocessing (in ogni caso necessario) e di formattazione.

I file forniti dalla competizione Kaggle sono i file *train.csv* e *test.csv*, i quali contengono un set di caratteri, in particolare i numeri da 0 a 9 scritti a mano. Ciascuno di essi è stato opportunamente isolato, ridimensionato e centrato all'interno di immagini 28x28 pixel per un totale di 784 pixel per immagine. Ogni pixel ha associato un valore intero tra 0 e 255 (compresi) che indica il livello di luminosità di quel pixel: più il valore è alto, più il pixel è scuro (e perciò indica la presenza di inchiostro).

Il training set, caricato nel file *train.csv*, è composto da 42000 righe e 785 colonne. La prima è la colonna 'label' contenente il valore numerico del numero scritto a mano associato a ciascun record, mentre le restanti 784 sono i valori numerici dei singoli pixel, 784 colonne denominate pixel**x** con **x** compreso tra 0 e 783.

Il test set, caricato nel file *test.csv*, è composto da 28000 righe e 784 colonne. Infatti non presenta la colonna target 'label' al fine di effettuare le predizioni e poi fornire i risultati a Kaggle che ne conoscerà gli effettivi valori associati e ne potrà valutare le performance.

Viene fornito in esempio anche un file *sample_submission.csv* utile a comprendere il formato per la consegna su Kaggle della propria soluzione. Si compone di una colonna 'ImageId' contenente i numeri ordinati da 0 a 27999 relativi alla lista di record del test set, e di una colonna 'Label' con i valori numerici predetti dal modello sviluppato.

La metrica di valutazione utilizzata da Kaggle è l'accuratezza (accuracy) e quindi la frazione di immagini correttamente classificate sul totale del test set.

# EDA & PREPROCESSING

***EDA (Exploratory Data Analysis)***

- STRUTTURA

| train_data.info() | test_data.info() |
|---|---|
| RangeIndex: 42000 entries, 0 to 41999<br>Columns: 785 entries, label to pixel783<br>dtypes: int64(785)<br>memory usage: 251.5 MB | RangeIndex: 28000 entries, 0 to 27999<br>Columns: 784 entries, pixel0 to pixel783<br>dtypes: int64(784)<br>memory usage: 167.5 MB |

Come definito nell'introduzione, i due set di dati presentano 42000 esempi e 785 colonne per il training set e 28000 esempi e 784 colonne per il test set. I valori contenuti nelle celle sono tutti valori interi.

- **MISSING VALUES**

Risulta importante verificare la presenza di Missing Values all'interno del dataset. In caso siano presenti è necessario trattarli attraverso tecniche di sostituzione (ad esempio, sostituendo i NaN con valori nulli (0), con valori ricorrenti nel dataset relativi a quella particolare feature, con valori medi, ecc.) oppure tecniche di eliminazione (ad esempio, eliminando le feature che presentano NaN oppure eliminando i record relativi).

In questo caso il dataset fornito dalla competizione risulta non avere alcun dato NaN.

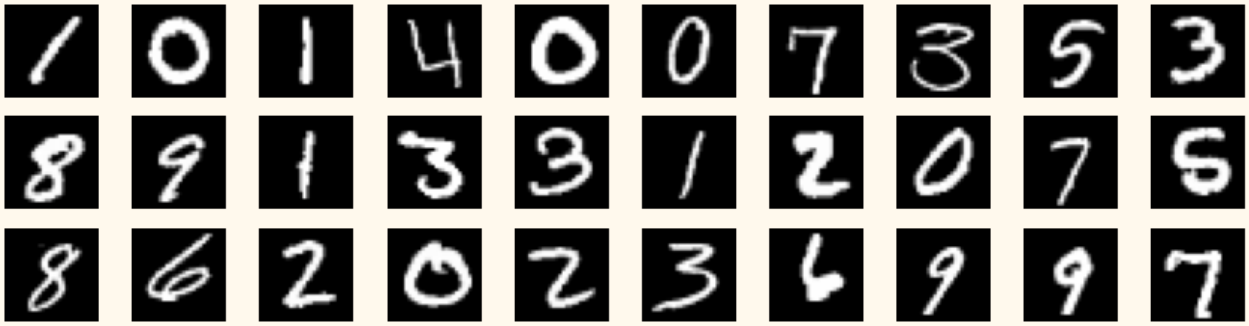
```
train_data.isnull().sum().sum()==0

return: True
```

- **DATA VISUALIZATION**

Importante è anche analizzare il contenuto effettivo di ogni record rappresentando graficamente i dati. Attraverso il reshape è possibile ridimensionare ogni riga del dataset rendendo il formato da 1x784 a 28x28.

Nell'immagine seguente si può vedere la rappresentazione grafica delle prime 30 immagini del training set. In questo caso è stato utilizzato *cmap='gray'*.

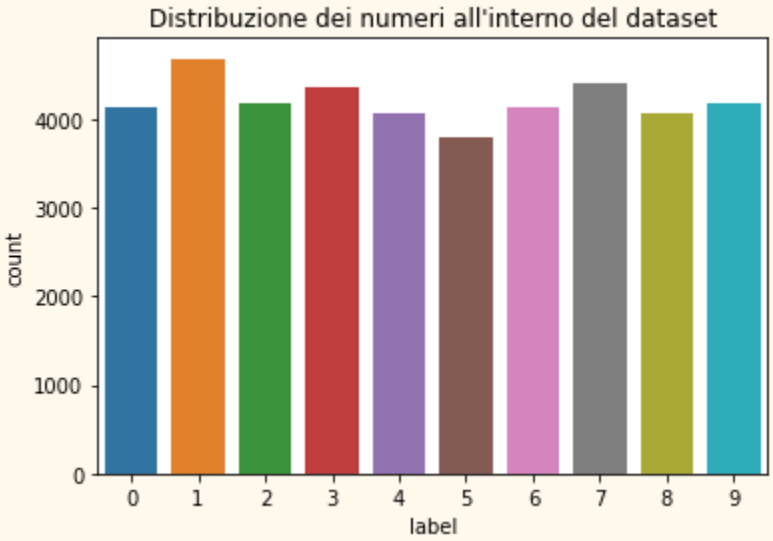


Si possono notare sia l'isolamento che la centratura di ogni numero all'interno di ogni singola immagine.

- **ANDAMENTO DELL'INTENSITÀ DEI PIXEL**

Attraverso il metodo *describe()* è possibile ricavare una descrizione statistica dei dati, ovvero tutte quelle informazioni riguardo alla dispersione, alla tendenza, alla dimensione dei dati all'interno del dataset quali valore medio, deviazione standard, valori minimo e massimo e percentili. In particolare, nel nostro caso, ci permette di comprendere, per ogni pixel, se esso presenta tendenzialmente dei valori o meno. Dalla tabella data in output si può notare come certi pixel presentino sempre intensità 0 ("mancanza di inchiostro") mentre altri presentino intensità 255 ("presenza di inchiostro"). Infatti, grazie al centramento dei digit è comprensibile il fatto che i pixel di bordo in ogni immagine presentino sempre valore zero in quanto non verranno mai occupati dal testo scritto, mentre pixel centrali all'immagine saranno probabilmente quelli più occupati.

- **DISTRIBUZIONE DEI DIGIT NEL DATASET DI TRAINING**



# PREPROCESSING

Come già descritto a livello introduttivo, il dataset MNIST fornito da Kaggle risulta essere già stato adattato per essere utilizzato senza un elevato preprocessing. Tuttavia, risulta comunque importante occuparsi di una serie di accorgimenti atti a semplificare maggiormente le operazioni di fitting degli algoritmi e soprattutto ad adattare i dati correttamente in base al modello da utilizzare. Queste operazioni sono la normalizzazione, lo scaling e il reshaping per tutti gli algoritmi, mentre altre operazioni utili quali il one-hot encoding e Image Data Augmentation aiutano le CNN ad accrescere le loro specifiche performance.

- **NORMALIZATION**

Ogni pixel presenta un valore compreso nel range [0,255]. Questi valori possono essere quindi degli interi molto alti e in grado di mettere in difficoltà l'apprendimento attraverso i diversi modelli. Ecco perché il miglior approccio risulta essere quello della normalizzazione dei dati. In questo caso, essendo riconosciuto il range di valori che assumono i pixel, è sufficiente dividere i valori dei set di dati per 255, ottenendo quindi un nuovo range [0,1] per pixel.

```
train_data=train_data/255.0
test_data=test_data/255.0
```

- **SCALING**

Il processo di scaling permette di standardizzare un dataset rimuovendo il valore medio da ogni elemento e dividendo il risultato per la deviazione standard lungo quell'asse. Questo processo di standardizzazione permette di migliorare le prestazioni di certi algoritmi di apprendimento (come ad esempio nel kernel RBF delle Support Vector Machine) le cui funzioni obiettivo presuppongono che tutte le features siano centrate attorno a zero e abbiano varianza dello stesso ordine. Features con varianza maggiore potrebbero dominare la funzione obiettivo e rendere lo stimatore incapace di apprendere correttamente le features con valori minori.

```
#scaling
from sklearn.preprocessing import scale
scaled_train = scale(train_data)
scaled_test = scale(test_data)
```

- **RESHAPING**

Altri algoritmi, come accade per le CNN (Convolutional Neural Networks), migliorano nelle prestazioni attuando il reshape dei dati. I dati di training e testing vengono scaricati come Dataframe composti da vettori in una dimensione composti da 784 features. Attraverso il reshape si ottengono delle matrici 3D, 28x28x1. In particolare, le prime due dimensioni indicano il formato effettivo delle immagini, mentre l'ultima dimensione è quella richiesta da Keras per l'implementazione dei modelli e corrisponde al numero di canali. In questo caso si ha un unico canale perché le immagini del MNIST sono in scala di grigi.

```
# Reshape delle immagini
img_size = 28
X_cnn=train_data.values.reshape(-1, img_size, img_size, 1)
```

- **ONE-HOT ENCODING**

Il modello CNN fornisce i risultati in un vettore di previsioni per ciascuna classe. Per far sì che il CNN sia in grado di apprendere correttamente e che quindi ci sia il corretto legame tra i vettori di uscita (il label previsto) e il dato in ingresso (il label effettivo), viene sfruttata la tecnica del one-hot encoding sulla feature 'label'. Questo metodo permette di rimuovere una categorical variable, nel nostro caso la variabile 'label' composta dai valori compresi tra 0 e 9, e sostituirla con una nuova variabile binaria, una per valore assumibile dalla variabile precedente, associata ad ogni valore unico intero. In questo caso vengono creati dei vettori di dimensione 10 che presentano tutti valore 0 ad eccezione dell'elemento di posizione associata alla cifra della label.

Ad esempio: se label=4, il one-hot vector associato a quella label sarà [0,0,0,0,1,0,0,0,0,0].

La CNN fornirà in uscita un vettore strutturato allo stesso modo, da cui sarà possibile ricavare nuovamente la label numerica associata.

```
y_train = to_categorical(y_train)
```

- **IMAGE DATA AUGMENTATION**

Un ulteriore elemento importante del preprocessing delle immagini per il riconoscimento di numeri scritti a mano è la Image Data Augmentation. Questo metodo viene utilizzato in particolare in combinazione con i modelli CNN per incrementarne le performance attraverso la traslazione, la rotazione, la deformazione delle immagini. In particolare, l'utilizzo di questo metodo con le CNN permette di incrementare l'accuratezza anche dello 0.71%.

Questo metodo è disponibile come funzione *ImageDataGenerator* all'interno del package *tensorflow.keras.preprocessing.image*. Permette di specificare le operazioni quali rotazione, zoom, shift, etc. da eseguire sulle immagini attraverso un insieme di parametri, alcuni dei quali sono specificati nel riquadro sottostante.

```
#Image Augmentation

datagen = ImageDataGenerator(
      zca_whitening=False,  # applica ZCA whitening
      rotation_range=10,  # ruota le immagini di 10°
      zoom_range = 0.1, # ingrandisce le immagini del 10%
      width_shift_range=0.1,  # sposta le immagini orizzontalmente del 10%
      height_shift_range=0.1,  # sposta le immagini verticalmente del 10%
      horizontal_flip=False,  # capovolge le immagini
      vertical_flip=False  # capovolge le immagini
)
```

L'operazione di fitting della CNN viene quindi in seguito eseguita non solo sfruttando il training set fornito, ma anche le immagini aggiuntive che sono state create attraverso ImageDataGenerator. Tutto ciò permette di evitare il problema dell'overfitting, di accrescere il numero di immagini su cui costruire l'apprendimento e di conseguenza aumenta l'accuratezza finale della CNN.

La funzione *fit_generator* di *Sequential()* qui sotto definita permette di eseguire il fit della rete neurale convoluzionale, oltre alla validazione del modello.

```
history = model_cnn.fit_generator(datagen.flow(X_train,y_train, batch_size=batch_size),
                                  epochs = n_epochs, validation_data = (X_valid,y_valid),
                                  verbose = 2, steps_per_epoch=X_train.shape[0] // batch_size,
                                  callbacks=[learning_rate_reduction, early_stop])
```

## MODELLI

Per risolvere il problema del riconoscimento di caratteri scritti a mano, i modelli di Machine Learning utilizzabili risultano essere molteplici. In particolare, gli algoritmi di classificazione implementati sono:

- *SVM*
- *K-NEAREST NEIGHBORS*
- *NAIVE BAYES*
- *DECISION TREE*

Gli ensable methods utilizzati sono invece

- *RANDOM FOREST*
- *XGBOOST*

Le precedenti due liste di algoritmi sono state implementate utilizzando il package Scikit-Learn con il linguaggio di programmazione Python.

Infine, essendo un problema di riconoscimento delle immagini, diviene utile e soprattutto efficiente l'utilizzo di modelli di Deep Learning. In questo caso mi sono occupato della costruzione di una *RETE NEURALE CONVOLUZIONALE* o *CNN*.

In quest'ultimo caso, la CNN è stata implementata usando TensorFlow con il package Keras, sempre utilizzando il linguaggio Pyhton. In particolare, la libreria Keras contiene numerose implementazioni delle più utilizzate componenti delle reti neurali quali le tipologie di layer, di funzioni target o di trasferimento, ottimizzatori e tanti altri strumenti utili. Nel paragrafo relativo alla risoluzione con CNN sono descritti in modo più dettagliato gli elementi utilizzati.

Per comparare gli algoritmi si è utilizzata come metrica principale l'**accuratezza sul test set,** parametro che viene fornito direttamente dalla competizione Kaggle in seguito alla Submission del risultato ottenuto. Inoltre sono stati valutati anche i tempi di esecuzione sul test set (sia per i fit che per le prediction) e precision, recall e f1-score sul validation set.

## SVM

SVM (Support Vector Machines) è un algoritmo di Machine Learning di tipo supervisionato. Durante la fase di training, SVM riconosce le relazioni tra ogni dato e il valore target nel training set, posizionando i dati in uno spazio n-dimensionale le n features, e quindi per ogni valore della feature relativa si avrà una coordinata spaziale. I punti vengono classificati trovando l'iperpiano che divide tra due classi. SVM sceglie così i support vectors in modo da trovare la suddivisione perfetta tra le classi.

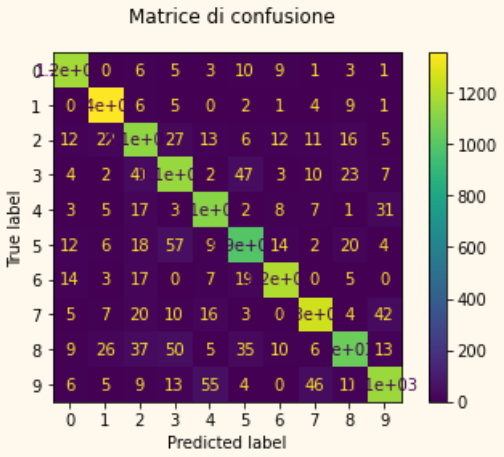
SVM si compone principalmente di due tipologie di algoritmi, quelli lineari e quelli non lineari. In particolare, risulta importante anche la scelta della kernel function associata per la risoluzione del problema. SVM risulta in ogni caso un metodo molto efficiente per risolvere problemi di classificazione.

Per implementare questo modello è stato utilizzato il modulo sklearn.svm, in particolare il modello SVC.

I modelli implementati sono due: SVM con kernel lineare e SVM con kernel RBF.

- **SVM con kernel lineare** $K(\vec{x}_i, \vec{x}_j) = \vec{x}_i \cdot \vec{x}_j$

Il modello SVM lineare sembra funzionare bene, con un'accuratezza sul validation set del 92%. Anche i valori ricavati dalle metriche sottostanti risultano essere buoni.



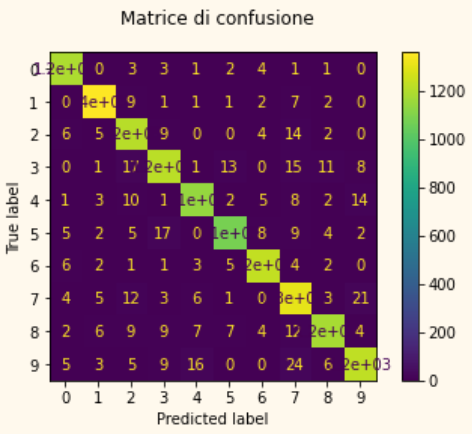

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.95 | 0.97 | 0.96 | 1204 |
| 1 | 0.95 | 0.98 | 0.96 | 1381 |
| 2 | 0.87 | 0.90 | 0.88 | 1252 |
| 3 | 0.87 | 0.89 | 0.88 | 1285 |
| 4 | 0.91 | 0.94 | 0.92 | 1190 |
| 5 | 0.89 | 0.87 | 0.88 | 1132 |
| 6 | 0.95 | 0.95 | 0.95 | 1257 |
| 7 | 0.94 | 0.92 | 0.93 | 1363 |
| 8 | 0.92 | 0.85 | 0.88 | 1239 |
| 9 | 0.92 | 0.89 | 0.90 | 1297 |
| accuracy | | | 0.92 | 12600 |
| macro avg | 0.92 | 0.92 | 0.92 | 12600 |
| weighted avg | 0.92 | 0.92 | 0.92 | 12600 |

- **SVM con kernel RBF o Gaussiano** $K(\vec{x}_i, \vec{x}_j) = \exp(-\gamma \left\| \vec{x}_i - \vec{x}_j \right\|^2)$

Con l'utilizzo del kernel RBF si ottiene un netto miglioramento delle prestazioni. Questo modello è quello maggiormente utilizzato in quanto il kernel RBF computa la similarità o quanto sono vicini due punti del dataset fra loro. In particolare, gli iperparametri utilizzati per il modello migliore, ricavati utilizzando GridSearchCV, sono *gamma*=0.001 e *C*=10.

Il parametro *gamma,* utilizzato nel kernel RBF, ha un valore basso, e ciò significa che ogni singolo esempio di training influenza a livello basso la scelta dei support vectors. In particolare, indica che la curvatura del decision boundary sarà minore.

Il parametro C ha anch'esso un valore basso, quindi si avrà un errore minore nella classificazione, in questo caso.




| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.98 | 0.99 | 0.98 | 1204 |
| 1 | 0.98 | 0.98 | 0.98 | 1381 |
| 2 | 0.94 | 0.97 | 0.96 | 1252 |
| 3 | 0.96 | 0.95 | 0.95 | 1285 |
| 4 | 0.97 | 0.96 | 0.97 | 1190 |
| 5 | 0.97 | 0.95 | 0.96 | 1132 |
| 6 | 0.98 | 0.98 | 0.98 | 1257 |
| 7 | 0.93 | 0.96 | 0.95 | 1363 |
| 8 | 0.97 | 0.95 | 0.96 | 1239 |
| 9 | 0.96 | 0.95 | 0.95 | 1297 |
| accuracy | | | 0.96 | 12600 |
| macro avg | 0.96 | 0.96 | 0.96 | 12600 |
| weighted avg | 0.96 | 0.96 | 0.96 | 12600 |

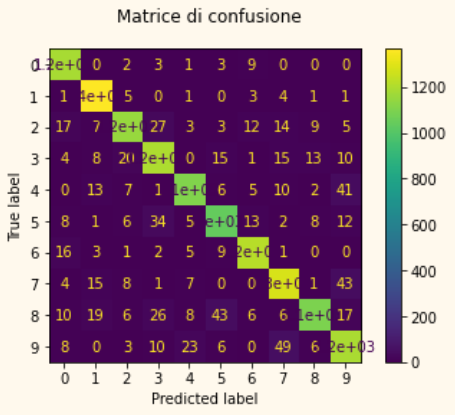
## k-NEAREST NEIGHBORS

KNN è un altro particolare modello di classificazione. La ricerca dei "nearest neighbors", ovvero dei "vicini più vicini", è il problema seguente: dato un set $S$ di $n$ punti in uno spazio metrico $X$, lo scopo è quello di pre-processare questi punti in modo che, dato un qualsiasi elemento di n punti $q \in X$, i punti più vicini a $q$ possano essere registrati velocemente. Questo problema viene chiamato anche *closest-point problem* o *post office problem*.

KNN viene quindi utilizzato anche nel campo del riconoscimento delle immagini basandosi, infatti, sulla similarità dei caratteri scritti a mano. Viene usato per assegnare una nuova osservazione ad una delle classi più comuni tra i k vicini ad un dato elemento, le classi già

conosciute. La classificazione viene poi effettuata in base al valore di soglia determinato con la media dei k dati che sembra essere più vicina.

Per implementare KNN è stato utilizzato il modello KNeighborsClassifier.




| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.95 | 0.99 | 0.97 | 1204 |
| 1 | 0.95 | 0.99 | 0.97 | 1381 |
| 2 | 0.95 | 0.92 | 0.94 | 1252 |
| 3 | 0.92 | 0.93 | 0.93 | 1285 |
| 4 | 0.95 | 0.93 | 0.94 | 1190 |
| 5 | 0.92 | 0.92 | 0.92 | 1132 |
| 6 | 0.96 | 0.97 | 0.97 | 1257 |
| 7 | 0.93 | 0.94 | 0.93 | 1363 |
| 8 | 0.96 | 0.89 | 0.92 | 1239 |
| 9 | 0.90 | 0.92 | 0.91 | 1297 |
| accuracy | | | 0.94 | 12600 |
| macro avg | 0.94 | 0.94 | 0.94 | 12600 |
| weighted avg | 0.94 | 0.94 | 0.94 | 12600 |

## NAIVE BAYES

Per risolvere il problema della Digit Recognition è possibile sfruttare anche il modello Naive Bayes. La strategia di predizione potrebbe essere quella di sfruttare le probabilità, come ad esempio la probabilità che un'immagine rappresenti un certo numero dati i suoi pixel.

$$\arg\max_Y P(Y|X_1,\ldots,X_n)$$

Dati Y la classe da predire e $X_1$, $X_2$, ..., $X_N$ i pixel relativi, possiamo sfruttare il teorema di Bayes e calcolare:

$$P(Y|X_1,\ldots,X_n) = \frac{P(X_1,\ldots,X_n|Y)P(Y)}{P(X_1,\ldots,X_n)}$$

Tuttavia questi valori di probabilità risultano difficili da calcolare, soprattutto a causa dell'elevato numero di parametri coinvolti (abbiamo 784 features per ogni immagine).

Attraverso l'assunzione di indipendenza tra i parametri dettata dall'algoritmo Naive Bayes

$$P(X_1,\ldots,X_n|Y) = \prod_{i=1}^{n} P(X_i|Y)$$

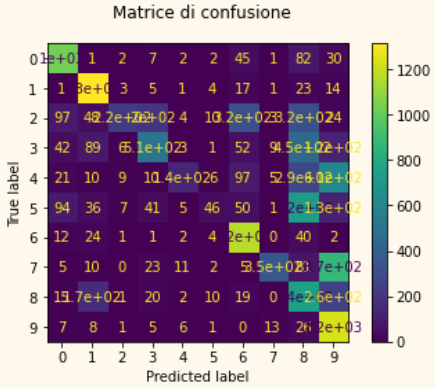
si passa da $2(2^n-1)$ a 2n parametri per la modellizzazione.

Il numero risulta quindi alto ma visibilmente ridotto rispetto al caso non Naive.

Ho utilizzato due modelli specifici di Naive Bayes, quello Gaussiano e quello di Bernoulli, al fine di verificarne le funzionalità e le differenze.

- **Gaussian Naive Bayes**

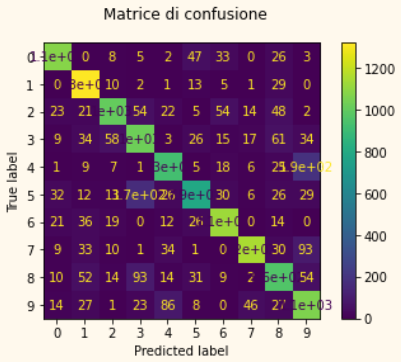
Questo modello, oltre all'assunzione di indipendenza dei parametri, aggiunge l'assunzione di distribuzione normale. Infatti, di solito, Gaussian Naive Bayes viene utilizzato nel caso in cui le features siano a valori continui. Nel nostro caso, le features risultano avere carattere binario, e come dimostrano anche la matrice di confusione e le metriche sottostanti si può notare come questo genere di modello non sia efficiente per la risoluzione del problema di digit recognition.




| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.78 | 0.86 | 0.82 | 1204 |
| 1 | 0.77 | 0.95 | 0.85 | 1381 |
| 2 | 0.88 | 0.18 | 0.30 | 1252 |
| 3 | 0.62 | 0.39 | 0.48 | 1285 |
| 4 | 0.80 | 0.12 | 0.20 | 1190 |
| 5 | 0.53 | 0.04 | 0.08 | 1132 |
| 6 | 0.66 | 0.93 | 0.77 | 1257 |
| 7 | 0.91 | 0.26 | 0.40 | 1363 |
| 8 | 0.27 | 0.60 | 0.37 | 1239 |
| 9 | 0.37 | 0.95 | 0.54 | 1297 |
| accuracy | | | 0.54 | 12600 |
| macro avg | 0.66 | 0.53 | 0.48 | 12600 |
| weighted avg | 0.66 | 0.54 | 0.49 | 12600 |

- **Bernoulli Naive Bayes**

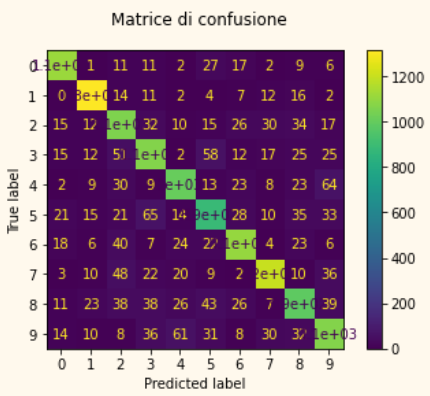
Si ha invece un netto miglioramento con il caso del modello di Bernoulli. In questo caso l'assunzione è la presenza di valori binari. In particolare, avremo che il valore 1 indicherà la presenza di colore (pixel colorato), e al contrario il valore 0 ne indicherà l'assenza (pixel vuoto). I valori assunti dalle features sono dei valori compresi tra 0 e 1, ma vengono arrotondati al valore intero più vicino. Dalle metriche sottostanti si evince un miglioramento.




| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.90 | 0.90 | 0.90 | 1204 |
| 1 | 0.85 | 0.96 | 0.90 | 1381 |
| 2 | 0.88 | 0.81 | 0.84 | 1252 |
| 3 | 0.75 | 0.80 | 0.77 | 1285 |
| 4 | 0.82 | 0.78 | 0.80 | 1190 |
| 5 | 0.83 | 0.70 | 0.76 | 1132 |
| 6 | 0.87 | 0.90 | 0.89 | 1257 |
| 7 | 0.93 | 0.85 | 0.88 | 1363 |
| 8 | 0.77 | 0.77 | 0.77 | 1239 |
| 9 | 0.72 | 0.82 | 0.77 | 1297 |
| accuracy | | | 0.83 | 12600 |
| macro avg | 0.83 | 0.83 | 0.83 | 12600 |
| weighted avg | 0.83 | 0.83 | 0.83 | 12600 |

## DECISION TREE

I Decision Tree sono dei modelli che permettono di risolvere un problema attraverso una struttura ad albero comprensibile, in grado di semplificare l'implementazione stessa dell'algoritmo. Le decisioni prese da un Decision Tree consistono in piccole scelte sequenziali che permettono di separare ogni nodo in due rami in base al valore dell'attributo prescelto per la separazione.



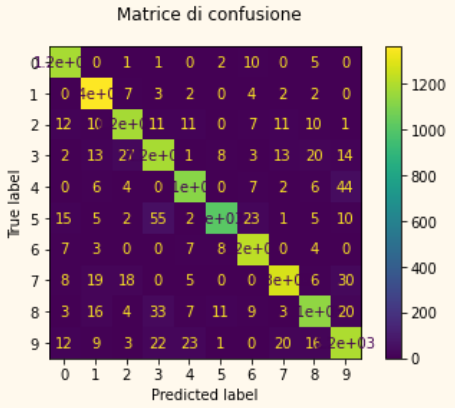

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.92 | 0.93 | 0.92 | 1204 |
| 1 | 0.93 | 0.95 | 0.94 | 1381 |
| 2 | 0.80 | 0.85 | 0.82 | 1252 |
| 3 | 0.82 | 0.83 | 0.83 | 1285 |
| 4 | 0.86 | 0.85 | 0.86 | 1190 |
| 5 | 0.80 | 0.79 | 0.79 | 1132 |
| 6 | 0.88 | 0.88 | 0.88 | 1257 |
| 7 | 0.91 | 0.88 | 0.90 | 1363 |
| 8 | 0.83 | 0.80 | 0.81 | 1239 |
| 9 | 0.82 | 0.82 | 0.82 | 1297 |
| accuracy | | | 0.86 | 12600 |
| macro avg | 0.86 | 0.86 | 0.86 | 12600 |
| weighted avg | 0.86 | 0.86 | 0.86 | 12600 |

## RANDOM FOREST

L'algoritmo Random Forest può essere usato sia in problemi di classificazione che di regressione come accade per i Decision Tree. La logica di questo algoritmo è quella di utilizzare più Decision Trees e produrre dei risultati medi sulla base di questi alberi. La randomicità che viene sottolineata nel nome sta nella creazione e costruzione di questi alberi: quando viene effettuato lo splitting, piuttosto che cercare direttamente l'attributo migliore, lo cerca per un sottoinsieme random di attributi. Questo fattore porta alla creazione di più alberi differenti tra loro, ognuno dei quali darà un risultato di accuratezza differente. L'insieme di tutti questi Decision Trees porta all'aumento dell'accuratezza totale della predizione.

Il RandomForestClassifier migliore implementato presentava una serie di iperparametri specificatamente settati: max_depth=20; max_features=4; min_samples_leaf=5; n_estimators=200.

Questa combinazione di iperparametri ha permesso di ottenere un insieme di alberi non troppo profondi rispetto al numero di features per cui fare gli split e di aumentare di molto l'accuratezza del modello finale.



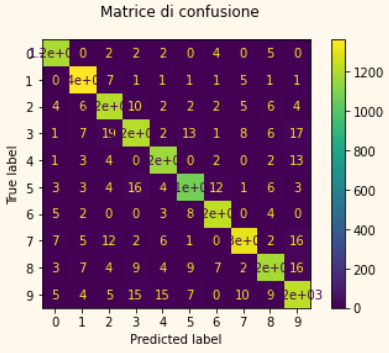

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.95 | 0.98 | 0.97 | 1204 |
| 1 | 0.94 | 0.99 | 0.96 | 1381 |
| 2 | 0.95 | 0.94 | 0.94 | 1252 |
| 3 | 0.90 | 0.92 | 0.91 | 1285 |
| 4 | 0.95 | 0.94 | 0.95 | 1190 |
| 5 | 0.97 | 0.90 | 0.93 | 1132 |
| 6 | 0.95 | 0.98 | 0.96 | 1257 |
| 7 | 0.96 | 0.94 | 0.95 | 1363 |
| 8 | 0.94 | 0.91 | 0.93 | 1239 |
| 9 | 0.91 | 0.92 | 0.91 | 1297 |
| accuracy | | | 0.94 | 12600 |
| macro avg | 0.94 | 0.94 | 0.94 | 12600 |
| weighted avg | 0.94 | 0.94 | 0.94 | 12600 |

## XGBOOST

Per la risoluzione di questo problema di digit recognition è stato utilizzato anche l'algoritmo XGBoost, uno di quelli maggiormente utilizzati all'interno delle competizioni Kaggle. Questo perché risulta essere ampiamente applicabile, scalabile e molto efficiente, in quanto è in grado di lavorare efficientemente con gli alberi sfruttando la parallelizzazione, il calcolo distribuito e l'ottimizzazione della cache.

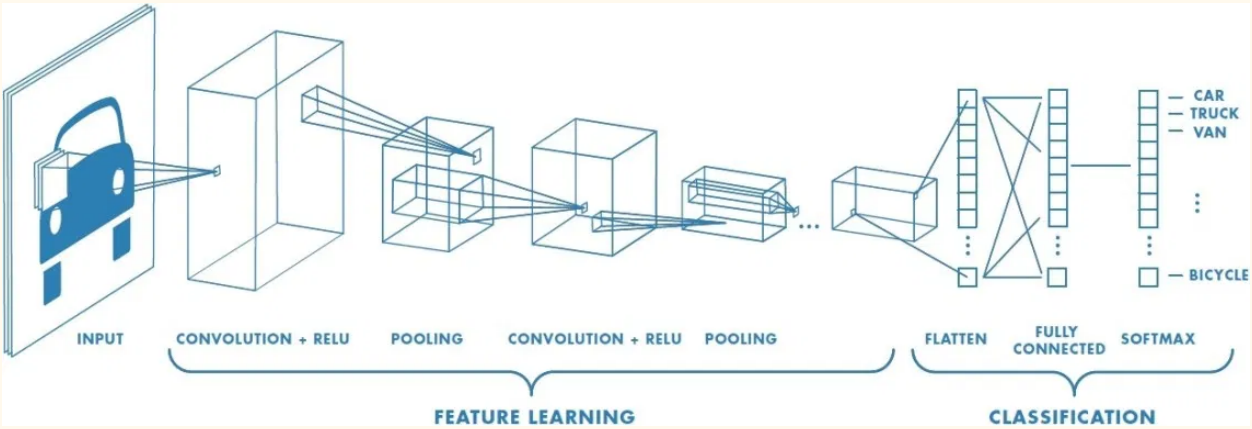
Nonostante l'elevato numero di features, talvolta non necessarie in quanto presentano dei valori sempre nulli o sempre elevati, XGBoost riesce a lavorare tranquillamente con tutte ed è in grado di computare la feature importance ad ogni iterazione dell'algoritmo. Non è necessario, pertanto, eseguire alcuna operazione di riduzione delle feature non necessarie in preprocessing.

Utilizzando un elevato numero di stimatori si ha un aumento dell'efficienza dell'algoritmo. I risultati sottostanti sono relativi all'utilizzo di un XGBClassifier con 500 stimatori e un valore di *eta*=0.1. La presenza di metriche a valori molto buoni si scontra con un execution time molto elevato, dovuto al training dei 500 stimatori.




| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.98 | 0.99 | 0.98 | 1204 |
| 1 | 0.97 | 0.99 | 0.98 | 1381 |
| 2 | 0.96 | 0.97 | 0.96 | 1252 |
| 3 | 0.96 | 0.94 | 0.95 | 1285 |
| 4 | 0.97 | 0.98 | 0.97 | 1190 |
| 5 | 0.96 | 0.95 | 0.96 | 1132 |
| 6 | 0.98 | 0.98 | 0.98 | 1257 |
| 7 | 0.98 | 0.96 | 0.97 | 1363 |
| 8 | 0.97 | 0.95 | 0.96 | 1239 |
| 9 | 0.95 | 0.95 | 0.95 | 1297 |
| accuracy | | | 0.97 | 12600 |
| macro avg | 0.97 | 0.97 | 0.97 | 12600 |
| weighted avg | 0.97 | 0.97 | 0.97 | 12600 |

## CNN




Una CNN o Rete Neurale Convoluzionale è una tipologia speciale di rete neurale multi-layer utilizzata per il riconoscimento automatico dei pattern visuali direttamente dai pixel delle immagini, sfruttando un preprocessing minimale.

Esse funzionano come tutte le reti neurali: hanno un input layer, degli hidden layer, che effettuano i calcoli tramite funzioni di attivazione, e un layer di output con il risultato. Ciò che cambia è la presenza della convoluzione al posto di una generica moltiplicazione matriciale in almeno uno dei suoi livelli. Ogni layer ospita una "feature map", ovvero la feature specifica che i nodi si preoccupano di cercare. Si ottiene, come risultato di un layer, una matrice leggermente più piccola dell'originale (o della stessa dimensione, se si usa zero padding), ovvero la feature map stessa. Ad ogni layer di neuroni si possono applicare eventualmente più filtri in modo da generare più feature map. Tipicamente ad ogni layer convoluzionale si fa seguire uno di max-pooling, riducendo via via la dimensione della matrice, ma aumentando il livello di "astrazione".

Per poter apprendere features complicate e funzioni in grado di rappresentare astrazioni di alto livello, le architetture devono essere di tipo "deep". Ciò consiste in un numero elevato di neuroni e in molteplici livelli di calcolo di non linearità, fattori in grado di aumentare la latenza.

*Convoluzione*: è un'operazione matematica in cui si "riassume" un tensore, una matrice o un vettore in un elemento più piccolo.

*Kernel* (o matrice di convoluzione): descrive un filtro a cui viene passata un'immagine in input. Questo kernel si sposta sull'immagine, di default da sinistra verso destra, dall'alto verso il basso, applicando un prodotto di convoluzione, il cui output sarà un'immagine filtrata.

$$G(x,y) = \omega * F(x,y) = \sum_{\delta x=-k_i}^{k_i} \sum_{\delta y=-k_j}^{k_j} \omega(\delta x, \delta y) \cdot F(x+\delta x, y+\delta y)$$

where $\omega$ is a kernel and $-k_i \leq \delta x \leq k_i, -k_j \leq \delta y \leq k_j$ its elements.

Ad esempio, prendiamo un kernel di convoluzione 3x3 che filtra un'immagine 9x9. Questo kernel si sposta su tutta l'immagine per catturare tutti i quadrati della stessa dimensione 3x3. Il prodotto di convoluzione è una moltiplicazione per elemento. La somma dei prodotti effettuati fornisce un risultato che viene associato al pixel risultante sull'immagine filtrata.

Il modello implementato è ***Sequential***, un modello appropriato per la costruzione di una pila di layers in cui ciascuno di essi ha esattamente un tensore di input e uno di output. Esso viene poi costruito attraverso gli add sul modello sfruttando i seguenti layer:

1- ***Conv2D***: è un layer convoluzionale in due dimensioni, utilizzato ad esempio per la convoluzione spaziale sulle immagini. Questo layer crea un kernel convoluzionale, convoluto con il layer di input per produrre un tensore di output.

2- ***MaxPooling2D***: layer con operazione di pooling 2D. Questo layer viene aggiunto dopo un layer convoluzionale, in particolare se è presente una non linearità (nel nostro caso ReLU) nella mappa delle caratteristiche del layer convoluzionale precedente, e utilizza il max-pooling per ridurre la dimensione di un'immagine, suddividendola in blocchi e tenendo solo l'elemento del blocco con valore più alto. In questo modo è possibile ridurre il problema dell'overfitting e vengono mantenute solamente le aree con maggiore attivazione.

3- ***Dropout***: layer di regolarizzazione. Permette di rimuovere in modo probabilistico degli input da un layer, i quali possono essere variabili di input del data sample o valori provenienti dai layer precedenti. Deve essere specificato un dropout_rate, ovvero la probabilità di settare ogni input al layer come valore zero.

4- ***Flatten***: layer di appiattimento. Consiste nel convertire i dati in un array unidimensionale per l'inserimento nel livello successivo, appiattendo quindi l'output di un livello per creare un unico vettore di features.

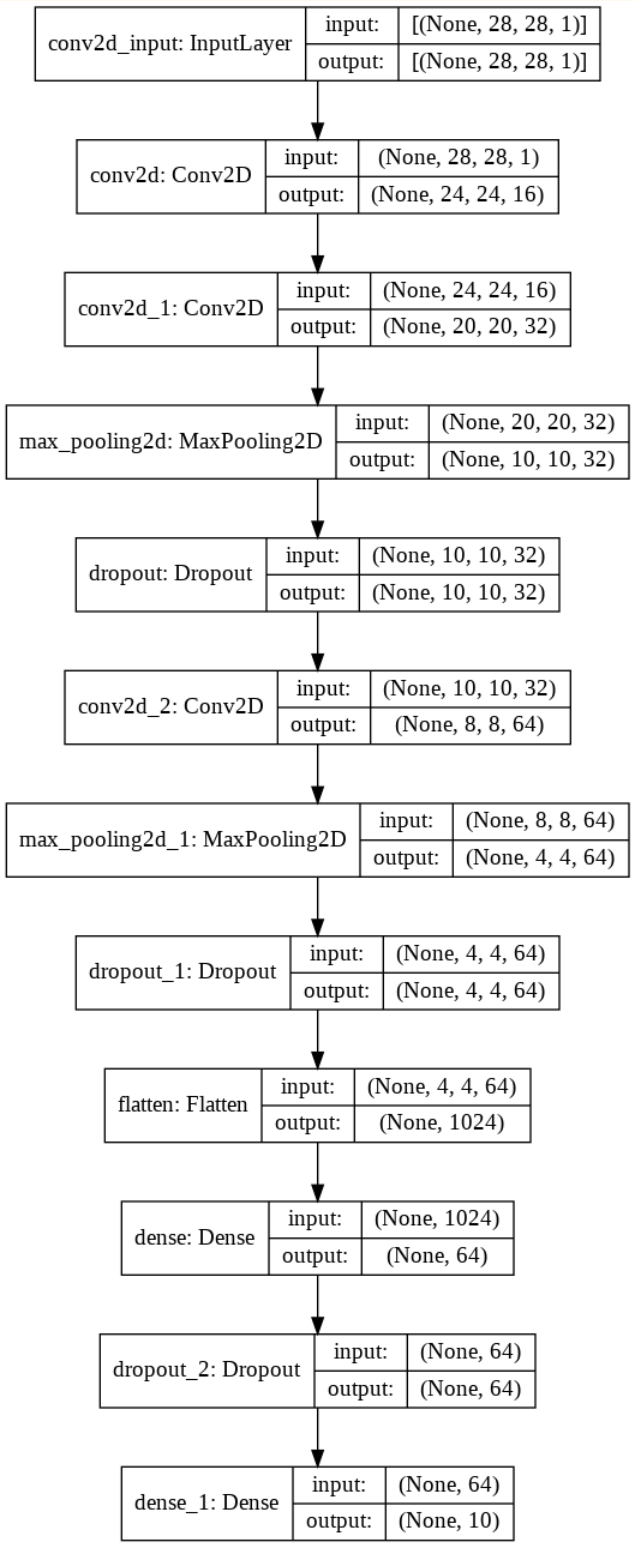
5- ***Dense***: è un tipico layer lineare delle reti neurali fully-connected, ovvero ogni neurone nello strato denso riceve input da tutti i neuroni dello strato precedente. Viene tendenzialmente utilizzato anch'esso per modificare la dimensione del vettore. Nel nostro caso, viene utilizzato anche come output layer in modo da ottenere i one-hot vector associati ai label predetti.

***ADAM optimizer***: è un ottimizzatore che implementa l'algoritmo di Adam. Questo algoritmo è un metodo stocastico di discesa del gradiente basato sulla stima adattativa dei momenti del primo e del secondo ordine, computazionalmente efficiente, richiedente poca memoria e adatto a problemi di grandi dimensioni in termini di dati e parametri.

***Callbacks***: una callback è un'azione eseguibile durante la fase di addestramento di una rete neurale. Nel nostro caso sono state implementate due callbacks:

- *ReduceLROnPlateau*: se non si vede un aumento dell'accuratezza entro un numero prestabilito di epoche, si ha una diminuzione del learning rate, fattore che può risultare vantaggioso per il modello;
- *EarlyStopping*: quando il valore di accuratezza smette di aumentare avanzando con le epoche, si ha l'interruzione dell'apprendimento dopo un numero prestabilito di epoche di "patience".

- **ARCHITETTURA**




Entrambi i modelli di CNN implementati si compongono di un layer di input, uno di output e una serie di layer nascosti presentati nell'immagine a fianco e spiegati alla pagina precedente. In entrambi i casi per l'addestramento sono stati provati più numeri di epoche massime, quindi un numero variabile di epoche.

Le immagini sono state riscalate ad una dimensione 28x28 (784 celle nella rappresentazione unidimensionale precedente), quindi il numero di neuroni negli strati di input e nascosti si ipotizza essere 784.

Il layer finale, di output, restituisce un array di 10 elementi, ovvero il one-hot vector associato alla label predetta.

- **CNN senza Image Augmentation**

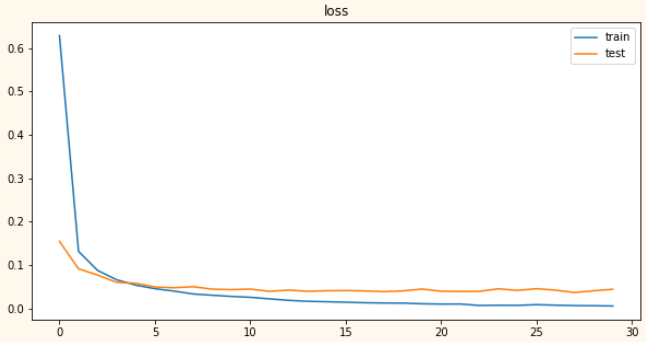
Il primo modello di CNN è stato implementato senza l'utilizzo dell'Image Data Augmentation, al fine di comprenderne l'andamento utilizzando solamente i dati di training di base forniti dalla competizione.

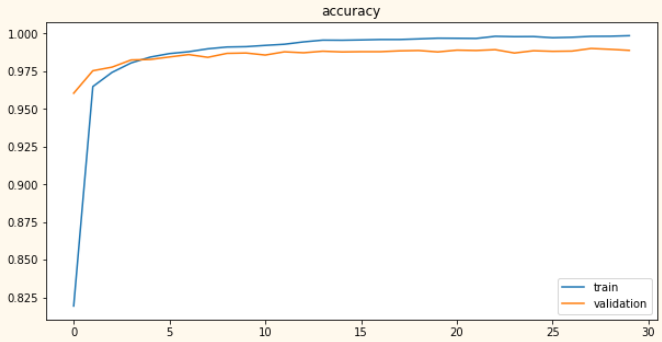
Dalla tabella sottostante si può notare come il modello sia già di suo molto efficiente, con in particolare un'accuratezza del 99% sul validation set.

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.99 | 1.00 | 0.99 | 1204 |
| 1 | 0.99 | 0.99 | 0.99 | 1381 |
| 2 | 0.98 | 0.99 | 0.99 | 1252 |
| 3 | 0.99 | 0.99 | 0.99 | 1285 |
| 4 | 0.99 | 0.99 | 0.99 | 1190 |
| 5 | 1.00 | 0.99 | 0.99 | 1132 |
| 6 | 0.99 | 1.00 | 0.99 | 1257 |
| 7 | 0.99 | 0.99 | 0.99 | 1363 |
| 8 | 0.98 | 0.98 | 0.98 | 1239 |
| 9 | 0.99 | 0.98 | 0.98 | 1297 |
| accuracy | | | 0.99 | 12600 |
| macro avg | 0.99 | 0.99 | 0.99 | 12600 |
| weighted avg | 0.99 | 0.99 | 0.99 | 12600 |

Durante la fase di fitting della rete neurale, grazie alla funzione *fit* è stato possibile anche l'esecuzione diretta della fase di validation. Il processo di validation è stato eseguito alla fine di ogni epoca, mostrando perciò nella fase di fitting anche l'andamento dell'accuratezza del modello epoca per epoca. In totale sono state eseguite 30 epoche nel caso presentato.

In seguito sono raffigurati i grafici di *loss* e *accuracy* del modello senza Data Augmentation.




Si può notare come la crescita dell'accuracy (e quindi la decrescita del loss) avanzi lentamente con l'aumentare delle epoche e sia molto elevata già a partire dalla seconda epoca. Sul validation set si ha una crescita più lenta, stabile intorno allo 0.9888.

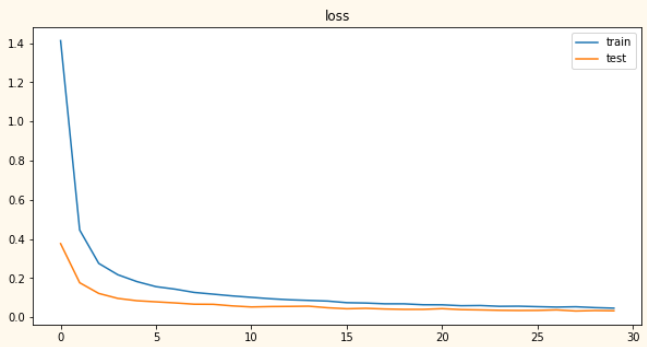
Vediamo ora se ci sono dei miglioramenti con l'utilizzo dell'Image Augmentation

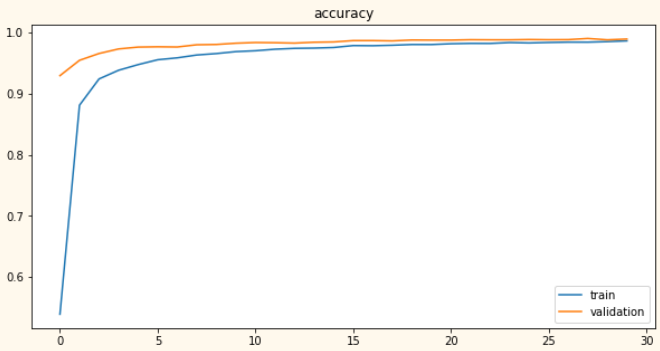
- **CNN con Image Augmentation**

Anche in questo caso i valori ricavati delle metriche risultano essere molto elevati per accuracy, precision, recall e f1-score. Tuttavia, semplicemente analizzando questi valori non si notano grandi differenze dal modello precedente.

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 1.00 | 1.00 | 1.00 | 620 |
| 1 | 1.00 | 0.98 | 0.99 | 697 |
| 2 | 0.98 | 0.99 | 0.98 | 617 |
| 3 | 0.99 | 0.99 | 0.99 | 653 |
| 4 | 0.99 | 0.99 | 0.99 | 583 |
| 5 | 0.99 | 0.99 | 0.99 | 593 |
| 6 | 0.99 | 0.99 | 0.99 | 610 |
| 7 | 0.99 | 0.99 | 0.99 | 668 |
| 8 | 0.98 | 0.99 | 0.98 | 605 |
| 9 | 0.99 | 0.99 | 0.99 | 654 |
| accuracy | | | 0.99 | 6300 |
| macro avg | 0.99 | 0.99 | 0.99 | 6300 |
| weighted avg | 0.99 | 0.99 | 0.99 | 6300 |

Osservando invece i grafici di *loss* e *accuracy* si possono notare invece degli andamenti leggermente diversi: in una prima fase di vede una crescita più lenta dell'accuratezza sui dati di training, ma avanzando si raggiungono nuovamente valori molto elevati. Si nota anche una crescita dell'accuratezza sul validation set, che si aggira attorno ad un valore massimo di 0.9892.




I risultati sono ancora più evidenti sul test set fornito dalla competizione Kaggle. Infatti, in seguito alla submission dei due modelli, si sono ottenuti i seguenti score:

- 0.98864 per la CNN senza Image Augmentation
- 0.99150 per la CNN con Image Augmentation

## RISULTATI

Nella seguente tabella sono presentati i risultati ottenuti eseguendo la submission delle predizioni sul test set fornito dalla competizione Kaggle. Come già specificato, lo score ottenuto è l'accuracy del modello. Inoltre, sono stati inseriti i tempi di esecuzione, sia per la fase di fitting che per quella di predizione, al fine di comprendere anche l'andamento temporale dei diversi modelli e perfezionarne la valutazione.

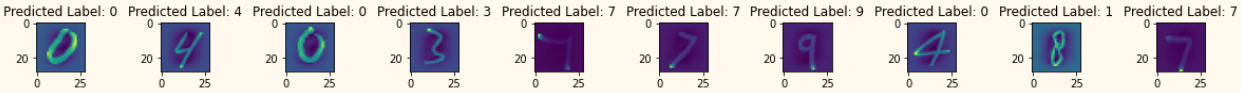
| **MODELLO UTILIZZATO** | ***Accuracy (dalla Kaggle Submission)*** | ***Fit time*** | ***Prediction Time (sul Test set)*** |
|---|---|---|---|
| SVM - modello lineare | *0.91564* | *2min 12sec* | *3min 15s* |
| SVM - modello non lineare | *0.96460* | *3min 29sec* | *4min 55s* |
| k-NEAREST NEIGHBORS | *0.93446* | *7.8s* | *22min 8s* |
| Gaussian Naive Bayes | *0.13382* | *308ms* | *1.29s* |
| Bernoulli Naive Bayes | *0.83292* | *271ms* | *389ms* |
| Decision Tree | *0.77425* | *9.92s* | *54.7ms* |
| Random Forest | *0.92614* | *6.92s* | *1.74s* |
| XGBoost | *0.92760* | *34min 56s* | *9.31s* |
| CNN - no Image Augmentation | *0.98864* | *24min 10s* | *10.1s* |
| CNN - Image Augmentation | *0.99150* | *31min 37s* | *10.2s* |

| VALORE PEGGIORE | VALORE MIGLIORE |
|---|---|

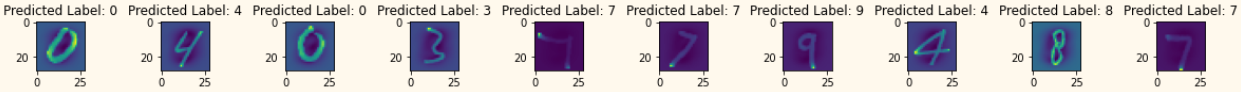
# PREDIZIONI

In seguito verranno mostrate 10 predizioni effettuate per ciascun modello sfruttato sui primi 10 esempi inseriti nei dati di validazione.

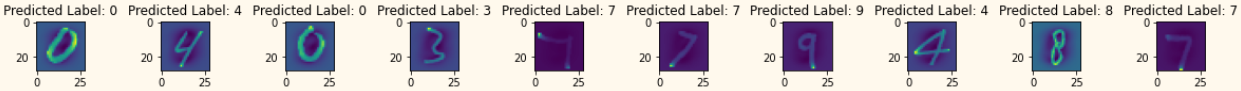
### SVM lineare



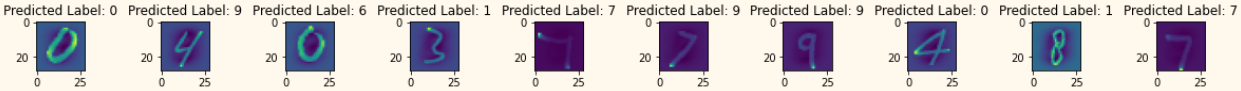

### SVM RBF



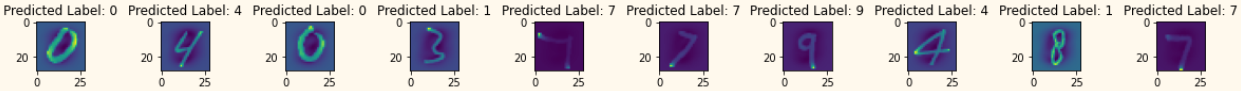

### k-NEAREST NEIGHBORS



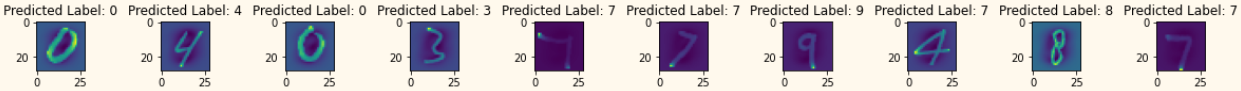

### GAUSSIAN NAIVE BAYES



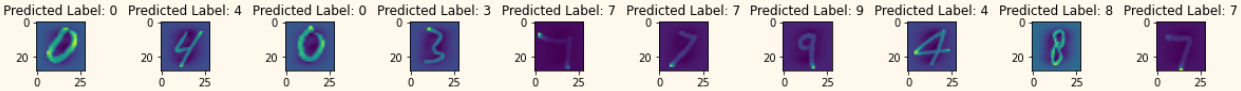

### BERNOULLI NAIVE BAYES



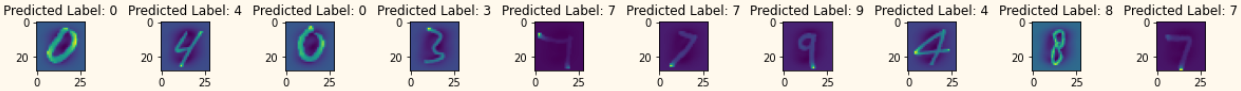

### DECISION TREE



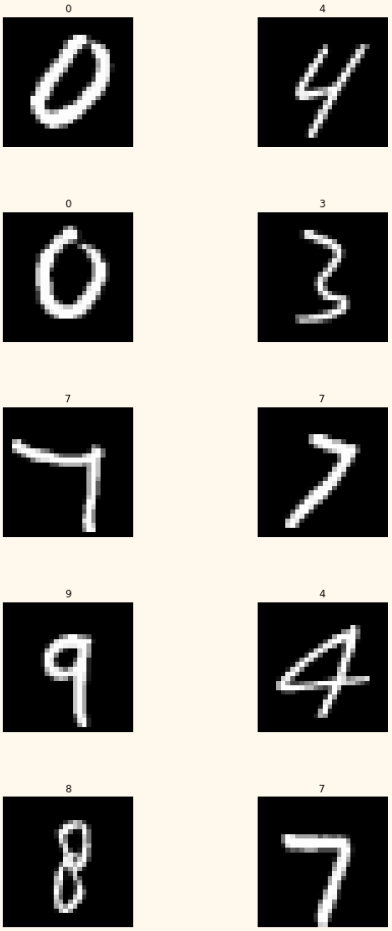
## RANDOM FOREST



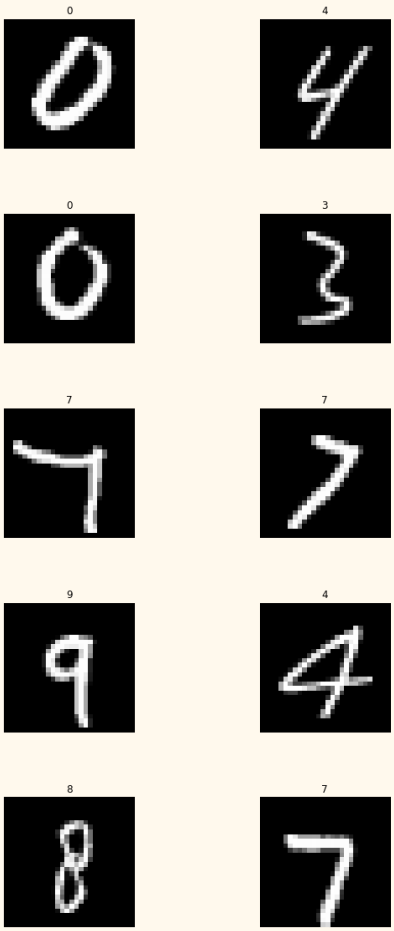

## XGBOOST




## CNN senza Image Augmentation




## CNN con Image Augmentation



# CONCLUSIONI

L'implementazione dei diversi modelli ha avuto successo e ha reso possibile effettuare anche un'analisi complessiva dell'applicabilità degli stessi.

Sia dai dati registrati nella tabella riassuntiva che dalla visualizzazione degli esempi di predizioni si evince quali siano gli algoritmi preferibili e quali non lo siano per le operazioni di digit recognition. In particolare:

- *Gaussian Naive Bayes* risulta essere il modello peggiore in quanto poco adatto al tipo di problema e ai valori che le features assumono;
- *Decision Tree* risulta essere non il più adatto a tale problema, in quanto il numero di features risulta essere molto elevato, gli alberi costruiti risultano potenzialmente complessi da comprendere, e inoltre esistono metodi che ne possono sfruttare maggiormente le capacità quali *Random Forest* e *XGBoost*;
- la maggior parte dei modelli risulta avere un'accuratezza intorno o superiore al 90%, un valore elevato ma che tuttavia lascia un margine di errore del 10% non trascurabile. Tra questi abbiamo *Random Forest, SVM lineare, XGBoost, Bernoulli Naive Bayes e k-Nearest Neighbors*;
- SVM non lineare, con kernel RBF, risulta essere l'algoritmo di classificazione di Machine Learning migliore, con un'accuratezza del 96.46% sul test set a scapito di un execution time più elevato;
- le due CNN sono i migliori modelli di deep learning (oltre ad essere gli unici) e risultano essere i migliori modelli per la digit recognition. Il problema principale risiede nell'execution time, tra i 20 minuti di fitting per il modello senza Image Augmentation ai 34 minuti per quello con Image Augmentation;
- l'utilizzo di Image Data Augmentation aumenta di fatto l'accuratezza, pertanto il miglior modello risulta essere la CNN con Image Augmentation (**99.15%**).

## CONSIDERAZIONI SULL'ESPERIENZA

Applicare le conoscenze ottenute durante il corso, adattarle, verificare il funzionamento di diversi algoritmi di Machine Learning e partecipare a questa competizione Kaggle è stato un percorso interessante e molto stimolante. Entrare a far parte di una community grande e spinta all'aiuto reciproco nella risoluzione dei problemi ha aiutato molto nel processo stesso di superamento della competizione da parte mia.

La possibilità di approfondire gli aspetti già visti e quella di entrare nel merito di un argomento ancora più complesso, il Deep Learning, ha reso la competizione ancora più accattivante, ma soprattutto la materia stessa ancora più accattivante.

L'implementazione della CNN mi ha permesso di raggiungere il 1832° posto nella competizione su 7064 team partecipanti, una posizione ancora bassa ma comunque importante per una prima esperienza con il Machine Learning e il Deep Learning.