Relazione Data Analytics

Riccardo Mioli - 983525[1]

Alma Mater Studiorum - Università di Bologna, Via Zamboni, 33, 40126 Bologna, Italia. riccardo.mioli2@studio.unibo.it[1]

Keywords: Data Analytics · ml-25m · Random Forest · Naive Bayes · Support Vector Machine · Multi Layer Perceptron

1 Introduzione

Il progetto svolto consiste in uno studio di analisi di dati basato sul dataset `ml-25m`[1]. I dati presi in considerazione riguardano valutazioni cinematografiche pubblicate su MovieLens[2], un celebre reccomender system in cui gli utenti possono assegnare valutazioni a film e aggiungere tag.

Durante lo svolgimento del progetto è stata replicata una data pipeline simile a quanto visto durante le lezioni del corso. Di seguito verranno affrontate le fasi di progettazione e implementazione della suddetta pipeline.

2 Metodologia

Prima di procedere alla progettazione e allo sviluppo della data pipeline è stato esaminato il dataset; questa operazione si è resa necessaria al fine di comprendere quali fossero i dati a disposizione e le procedure di pre-processing da svolgere.

Per lo svolgimento del presente studio sono stati utilizzati tutti i file contenuti nell'achivio del dataset[3], ad eccezione di: `links.csv`, in quanto contenente riferimenti a siti esterni (imdb e tmdb), inutili allo scopo del progetto, e `tags.csv`. Il motivo del non utilizzo dei tag è che questi sono assegnati dagli utenti, e dunque potrebbero contenere imprecisioni ed errori difficilmente verificabili, inoltre nel file `movies.csv` è contenuta una colonna `genres` che, insieme ai `genome-scores`, aiuterà a definire adeguatamente le caratteristiche dei film presi in esame.

Visionati i dati a disposizione e appurate le procedure da svolgere, le fasi pianificate per la data pipeline sono:

- Data Acquisition: in questa fase i diversi file del dataset verranno letti mediante l'utilizzo della libreria `pandas`. I singoli dataframe saranno poi uniti tra loro per crearne uno unico.
  Il file `ratings.csv` contiene i punteggi assegnati ad ogni film dai singoli utenti; per ogni film verrà quindi calcolato il punteggio medio dei suoi voti, questo confluirà poi nella colonna `rating` del dataframe finale.
- Data Pre-processing: i punteggi medi, ottenuti dalla scorsa fase della pipeline, verranno discretizzati in bin corrispondenti a 0.5 stelle; in questo modo sarà possile lavorare ad un problema di classificazione.

Considerando l'enorme numero di genomi per ogni film, che si concretizza in oltre 1000 colonne, verranno testate due tecniche di dimensionality reduction, ossia la Principal Component Analysis e la Linear Discriminant Analysis. Sulla base delle performance dei modelli addestrati verrà mantenuto il metodo di dimensionalty reduction che porta ad ottenere le migliori prestazioni. Lo stesso processo sarà seguito per lo scaling dei dati.
Infine saranno svolte operazioni di data augmentation mediante la libreria `imbalanced learn`; anche in questo caso verranno testati vari algoritmi (e.g. Random Oversampling e SMOTE) e verrà mantenuto quello che garantisce le migliori performance per il modello.

- Modeling: in questa fase verranno implementati i vari modelli richiesti dalle specifiche del progetto. Le librerie utilizzate a tal fine saranno `scikit-learn` per i modelli classici (Random Forest, Naive Bayes, Support Vector Machines) e `torch` per il Multi Layer Perceptron.
  Per i modelli facenti parte di `scikit-learn`, il tuning degli iperparametri verrà effettuato attraverso una Random Search, svolta utilizzando gli strumenti già disponibili all'interno libreria; per quanto riguarda l'ottimizzazione degli iperparametri della rete neurale, anche in questo caso verrà utilizzata la tecnica di random search, tuttavia questa verrà gestita utilizzando la libreria `ray-tune`.
- Performance analysis: l'analisi delle performance verrà effettuata in due momenti: a seguito del training con cross-validation e infine su un test set non utilizzato per la fase di apprendimento; in questo modo sarà possibile valutare se vi è stato overfitting sui dati durante la fase di training.
  Le metriche utilizzate per la valutazione delle performance saranno: precision, recall, accuracy ed f1 score.
- Visualization: la fase di visualization si svolgerà durante tutto lo studio, in quanto utile a una comprensione visiva di quanto sta avvenendo (e.g. discesa del gradiente, resampling, etc).

3 Implementazione

3.1 Data Acquisition

La lettura dei dati avviene mediante una semplice chiamata alla funzione `read_csv` di `pandas`[1].

```
movies = pd.read_csv(os.path.join(path, "movies.csv"),
    index_col="movieId", usecols=["movieId", "genres"])
```

Listing 1.1: Lettura dati

La colonna `genres` del dataset movies è tuttavia in formato pipe separated values, inoltre il numero di generi per film è variabile. Questo problema è

[1] Per brevità si mostrerà il caricamento solamente del file `movies.csv`, per gli altri file l'operazione è identica.

semplicemente risolvibile utilizzando il metodo `get_dummies`, che permette di effettuare in modo automatico l'one hot encoding delle variabili indicanti il genere dei film.

```
genres = movies["genres"].str.get_dummies()
```

Listing 1.2: One hot encoding dei generi.

Un'ulteriore operazione che si è reso necessario svolgere prima di poter procedere all'unione dei dataset è stata la trasposizione del dataframe contenente i valori per ogni genoma.

Il file .csv dei genomi è così strutturato: `movieId, tagId, relevance`. La funzione `pivot` permette di specificare l'indice per ogni riga e le colonne da trasformare in righe, in questo modo è possibile effettuare il reshaping dei genomi trasponendo i dati.

```
genome_scores = genome_scores.pivot(index="movieId", columns=
    "tag", values="relevance")
```

Listing 1.3: Reshaping genoma.

Infine è stato calcolato il voto medio per ogni film. Tale operazione viene portata a termine in due passaggi: le recensioni vengono raggruppate mediante l'operatore `groupby` di `pandas` e a seguire viene chiamata la funzione `mean` su ogni gruppo.

```
y = ratings.groupby("movieId")["rating"].mean()
```

Listing 1.4: Calcolo voto medio per ogni film.

I singoli dataframe originati dalle operazioni precedenti sono stati "fusi" mediante la funzione `merge`[2].

```
movies = movies.merge(y, on="movieId")
```

Listing 1.5: Merge dataframe.

La funzione di merge, per impostazione di default, esegue una inner join tra i dataframe, dunque i film per cui non sono presenti recensioni, o dati sul genoma, non verranno inclusi nel dataframe finale, che consiste di 13816 righe e 1149 colonne.

Al termine della fase di data acquisition è stata verificata l'integrità dei dati mediante alcue semplici operazioni booleane sul dataframe; nello specifico le condizionie verificate sono state:

1. che non vi fossero film con genere, genoma o voti con valori inferiori a 0;
2. che le colonne riguardanti il genere e il genoma non avessero valori superiori a 1;
3. che non vi fossero film con valutazione superiore a 5.

Una qualsiasi violazione di questi vincoli potrebbe indicare un sample con errori di inserimento, tuttavia dai controlli è emerso come non siano presenti sample con attributi i cui valori presentano anomalie rispetto alla scala stabilita.

[2] Anche in questo caso l'operazione è ripetitiva e verrà mostrata solamente una volta.

```
df_info = movies.describe().loc[["min", "max"], :]
condition0 = df_info < 0
condition1 = df_info.loc[:, df_info.columns != "rating"] > 1
condition2 = df_info.loc[:, df_info.columns == "rating"] > 5
```

Listing 1.6: Verifica consistenza colonne dataset.

3.2 Data Pre-Processing

La prima operazione svolta nella fase di pre-processing è stata il binning dei voti medi per ciascun film, questa operazione è stata effettuata al fine di impostare il progetto come un problema di classificazione.

```
bins = [0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5]
binned_ratings = pd.cut(ratings, bins=bins, labels=bins[1:])
label_encoder = LabelEncoder()
label_encoder.fit(binned_ratings)
binned_ratings = label_encoder.transform(binned_ratings)
```

Listing 1.7: Binning dati e encoding.

A seguito del binning in intervalli pari a 0.5 stelle, il voto medio per ogni film è stato trasformato in una label; il motivo dell'operazione deriva dal fatto che gli oversampler inclusi nella libreria `imbalanced-learn` richiedono che l'etichetta per un sample non sia un dato numerico continuo.

I dati a disposizione sono stati poi riscalati utilizzando gli scaler inclusi nel modulo `preprocess` di `scikit`. Lo scaling dei dati è stato effettuando testando le seguenti tecniche: normalizzazione (l1 e l2), min max scaling e standardizzazione.

```
scaler = Normalizer(copy=False, norm='l2') #l1
# scaler = StandardScaler(copy=False)
# scaler = MinMaxScaler(copy=False)
df = scaler.fit_transform(df.to_numpy())
```

Listing 1.8: Scaling dei sample.

A seguito della fase di scaling è stata svolta la fase di dimensionality reduction e il resampling dei dati. Per la fase di resampling dei dati verranno testati diversi algoritmi tra cui SMOTE, che permette di creare nuovi esempi "sintetici" sulla base di k-sample vicini[3].

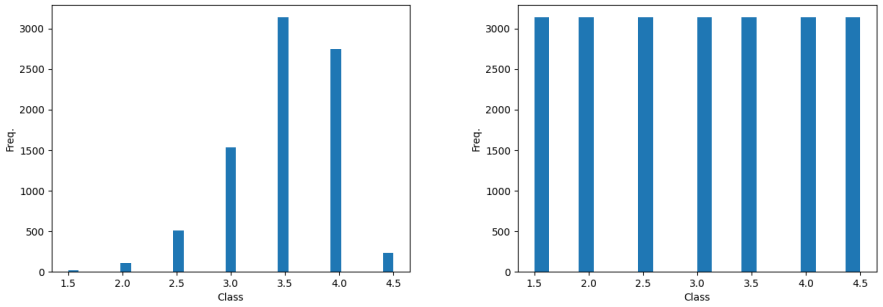
```
# projector = PCA(0.8)
projector = LinearDiscriminantAnalysis()
projector.fit(X_train, y_train)  # y_train is automatically
    ignored in PCA
X_train = projector.transform(X_train)
X_val = projector.transform(X_val)
```

[3] Quanto descritto, a differenza del random oversampling, che si limita a replicare più volte le stesse osservazioni, consente di andare a "popolare" zone del feature space per le quali si hanno un esiguo numero di sample.

```
X_test = projector.transform(X_test)
# oversampler = RandomOverSampler()
oversampler = SMOTE(k_neighbors=6)  #4,5,6,7,8
X_train, y_train = oversampler.fit_resample(X_train, y_train)
```

Listing 1.9: Dimensionality reduction e resampling dati.

Nella figura 1 è possibile visionare lo stato del bilanciamento delle classi prima del resampling e in seguito a tale operazione. Prima del ribilanciamento appare evidente come i film con valutazioni non appartenenti alle classi 3.5 o 4 siano sottorappresentati, qualora il dataset non venisse ribilanciato si potrebbero riscontrare performance scadenti per queste classi sui modelli addestrati. A seguito del resampling tutte le classi sono ugualmente rappresentate, ciò è visibile nella figura 1b.



(a) Distribuzione classi pre resampling. (b) Distrubuzione classi post resampling.

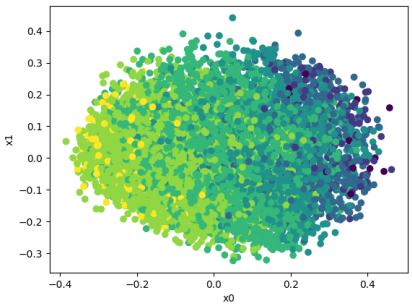
Figura 1: Sbilanciamento classi prima e dopo il resampling.

Nella figura 2 appare invece evidente l'effetto dei due metodi di dimensionality reduction testati. Se la Pricipal Componenet Analysis, figura 2a, crea nuove feature proiettandole sulle componenti principali (i.e. direzioni di maggiore varianza), la Linear Discriminant Analysis, figura 2b, minimizza la varianza intraclasse e massimizza quella interclasse, questo effetto è ben visibile in quanto nel nuovo spazio creato da LDA, le classi, distinguibili per il colore assegnato ai sample, sono visibilmente raggruppate.

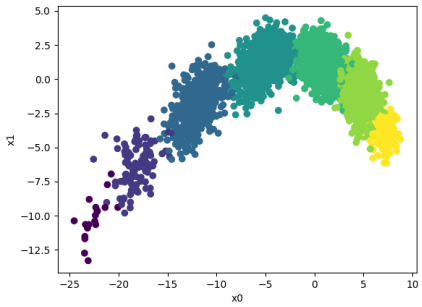
3.3 Modeling

La fase di modeling consiste nell'addestramento dei vari modelli. Come accennato nella sezione 2, l'ottimizzazione degli iperparametri si baserà su una Random Search con cross-validation; entrambe le tecniche sono già implementate nella classe `RandomizedSearchCV`, inclusa nella libreria `scikit`.

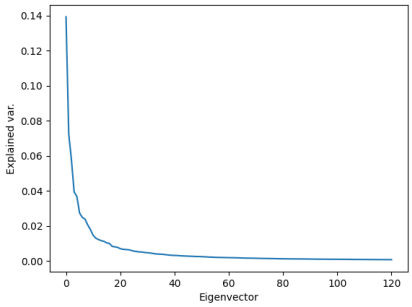
Al fine di garantire un adeguato numero di combinazioni di iperparametri testati, verranno effettuate 10 iterazioni per ogni matrice di iperparametri. Il



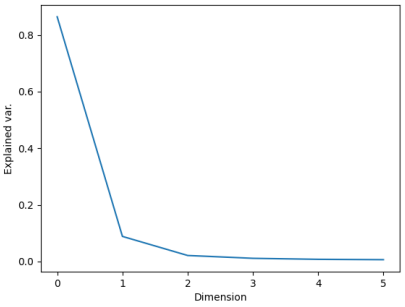
(a) PCA scatterplot.



(b) LDA Scatterplot.



(c) PCA explained variance ratio.



(d) LDA explained variance ratio.

Figura 2: PCA e LDA dimensionality reduction.

numero di fold è invece stato fissato a 5, in modo da testare ogni combinazione di iperparametri un sufficente numero di volte ed escludere quindi che una buona accuratezza possa derivare da un campione di dati favorevole.

```
def train_models(X_train: np.ndarray, Y_train: np.ndarray,
    x_test: np.ndarray, y_test):
    # Params for hyperparams tuner
    n_jobs = os.cpu_count()-1
    n_iter = 10
    cv = 5
    scoring = "accuracy"
    btm = {}  # best trained models
    # Random forest classifier
    hyperparams = {"n_estimators": list(range(100, 350, 50)),
     'criterion': ['gini', 'entropy'], 'max_depth': list(
    range(10, 30, 5))+[None], 'min_samples_split': list(range
    (2, 11, 2))}
    estimator = RandomForestClassifier(n_jobs, random_state=
    __SEED)
    rf = RandomizedSearchCV(estimator, hyperparams, n_jobs=
    n_jobs, verbose=verbose, cv=cv, scoring=scoring, n_iter=
    n_iter)
    rf.fit(X_train, Y_train)
    btm['random_forest'] = rf.best_estimator_
    dump(rf.best_estimator_, os.path.join(__DUMP_MODELS_PATH,
     'random_forest.joblib'))
    # SVM
    hyperparams = {'kernel': ['linear', 'rbf', 'poly', '
    sigmoid'], 'degree': list(range(2, 5)), 'tol': np.
    linspace(1e-3, 1e-5, 5), 'C': np.linspace(1, 5, 5)}
    estimator = SVC()
    svc = RandomizedSearchCV(estimator, hyperparams, n_jobs=
    n_jobs, verbose=verbose, cv=cv, scoring=scoring, n_iter=
    n_iter)
    svc.fit(X_train, Y_train)
    btm['support_vector'] = svc.best_estimator_
    dump(svc.best_estimator_, os.path.join(__DUMP_MODELS_PATH
    , 'support_vector.joblib'))
```

Listing 1.10: Random search modelli classici.

Per ottimizzare i tempi di esecuzione le random search vengono lanciate in parallelo su un numero di core pari a n-1 rispetto a quelli disponibili, questo approccio è conservativo ed è stato messo in atto per consentire l'utilizzo del calcolatore anche per altri scopi durante l'addestramento dei modelli. Il problema non si porrà per l'addestramento della rete neurale, che avverrà in GPU.

Nel listato precedente sono ben visibili tutti gli iperparametri che vengono ottimizzati per i modelli classici[4], al termine dell'addestramento il miglior modello tra quelli addestrati viene salvato in un dizionario, che verrà restituito dalla

[4] Nel listato non viene riportato l'addestramento del modello relativo a Naive Bayes, la

funzione di train, e in un archivio compresso mediante `joblib` per consentire di caricarlo e utilizzarlo in un secondo momento.

Per la creazione della rete neurale è stata impiegata la libreria `pytorch`. Per questioni di spazio non verrà riportato l'intero codice della rete, bensì ci si soffermerà sulle sue caratteristiche principali.

```
self.activation_function = torch.nn.ReLU()
# Building the layers
layers = OrderedDict()
layers[str(len(layers))] = torch.nn.Dropout(p=dropout_prob)
layers[str(len(layers))] = torch.nn.Linear(input_layer_size,
    hidden_layer_size)
layers[str(len(layers))] = torch.nn.BatchNorm1d(
    hidden_layer_size)
layers[str(len(layers))] = self.activation_function
for i in range(0, number_hidden_layers):
    layers[str(len(layers))] = torch.nn.Dropout(p=
    dropout_prob)
    layers[str(len(layers))] = torch.nn.Linear(
    hidden_layer_size, hidden_layer_size)
    layers[str(len(layers))] = torch.nn.BatchNorm1d(
    hidden_layer_size)
    layers[str(len(layers))] = self.activation_function
    layers[str(len(layers))] = torch.nn.Linear(
    hidden_layer_size, output_layer_size)
self.linear_relu_stack = torch.nn.Sequential(layers)
```

Listing 1.11: Architettura rete neurale.

Come è possibile vedere dal listato precedente, la rete neurale utilizza il dropout, per evitare overfitting sui dati, e la batchnorm, in modo da non subire l'effetto dell'internal covariate shift, ossia lo spostamento ad ogni layer di media e varianza causato dai pesi di quel layer. La funzione di attivazione scelta, considerato il problema di classificazione, è la Rectified Linear Unit.

Il training della rete neurale verrà gestito mediante minibatch, in questo modo sarà possibile effettuare un maggior numero di aggiornamenti dei pesi per ogni epoca in quanto questa operazione verrà svolta per ogni batch, pertanto l'apprendimento risulterà più rapido. La fase di train sarà poi velocizzata ulteriormente anche grazie all'utilizzo del momentum.

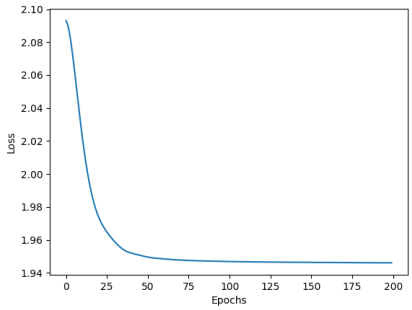
L'ultima aggiunta effettuata alla funzione di train è stata la funzionalità di early stopping, in questo modo l'apprendimento della rete neurale verrà fermato automaticamente se la loss sul validation set[5] diventa maggiore di quella del training set, ossia se vi è un principio di overfit sui dati di train.

---

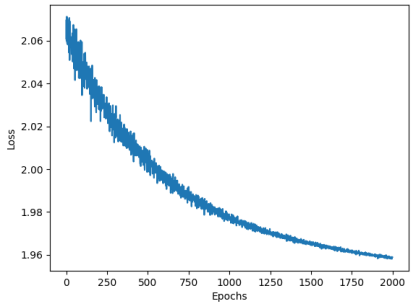
scelta deriva dal fatto che non sono presenti iperparametri da ottimizzare e dunque la fase di training si limita a una semplice chiamata `fit`.

[5] Il modulo per la random search di `ray-tune` non consente di svolgere la k-fold cross validation in modo automatico, per questioni di semplicità è stato quindi utilizzato un validation set.

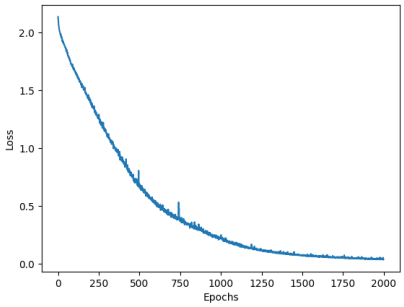
Nella figura 3 è possibile vedere l'evoluzione della loss per ogni modifica apportata durante l'implementazione della rete neurale. Se nella figura 3a l'andamento descrescente della loss è poco rumoroso, come tipico per una discesa del gradiente senza minibatch, in figura 3b è possibile notare come gli aggiornamenti più frequenti dei pesi comportino una maggiore rumorosità nella discesa del gradiente. In seguito, con l'aggiunta della batch normalization (figura 3c) è evidente come la rumorosità migliori dal momento che ad ogni layer i dati vengono ricentrati. Infine l'aggiunta del dropout 3d tende a "piegare" la discesa del gradiente e a renderla livemente più rumorosa, ciò deriva dallo spegnimento randomico di alcuni neuroni, che determina un apprendimento più lento e difficoltoso.



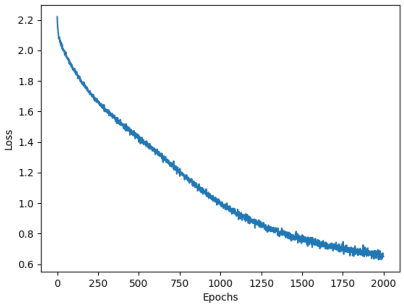
(a) Loss SGD.



(b) Loss con minibatch.



(c) Loss batchnorm.



(d) Loss con dropout.

Figura 3: Andamento loss a seguito della modifica dell'architettura della rete e della modalità di training.

Come accennato in precedenza, il tuning degli iperparametri della rete neurale è stato gestito con `ray-tune`, il funzionamento è il seguente: alla libreria viene fornita in input una funzione che si occupa di inizializzare un data loader e gli iperparametri della rete neurale, i parametri vengono però passati alla funzione

di train da `ray-tune` sulla base del sampling randomico effettuato inizialmente sulla matrice degli iperparametri.

```
configs = {"hidden_layer_size": hidden_layer_size, "
    number_hidden_layers": number_hidden_layers, "
    learning_rate": learning_rate, "momentum": momentum, "
    batch_size": batch_size, "epochs": epochs, "dropout_prob"
    : dropout_prob}

def train_nn(config: dict, X_train, y_train):
    train_loader = DataLoader(Dataset(X_train, y_train),
    config['batch_size'], shuffle=True, drop_last=True)
    val_loader = DataLoader(Dataset(X_val, y_val), X_val.
    shape[0])
    mlp = NeuralNetwork(X_train.shape[1], config['
    hidden_layer_size'], y_train.max(
    ).astype(int)+1, config['number_hidden_layers'], config['
    dropout_prob'])
    logger.info(mlp)
    optimizer = torch.optim.SGD(
        mlp.parameters(), config['learning_rate'], config['
    momentum'])
    mlp, loss = mlp._train(torch.nn.CrossEntropyLoss(
    ), optimizer, config['epochs'], train_loader, val_loader)

results = tune.run(tune.with_parameters(train_nn, X_train=
    X_train, y_train=y_train), config=configs,
    scheduler=ASHAScheduler(metric="accuracy", mode="max"),
    num_samples=10, resources_per_trial=tune_res)
```

Listing 1.12: Configurazione random search con ray tune.

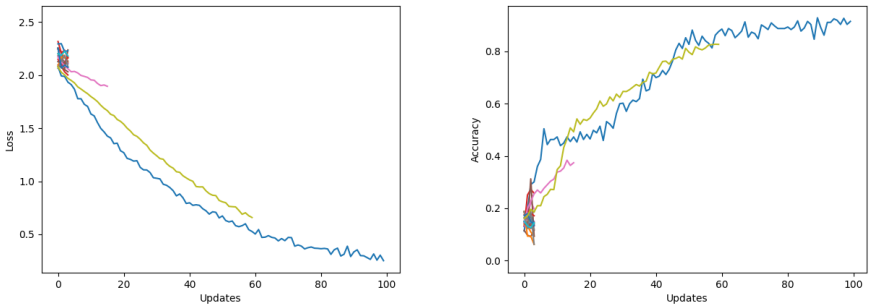
Nel listato precedente è possibile notare come il tuning faccia utilizzo di uno scheduler, nello specifico l'`AsyncHyperBandScheduler`[6]. L'utilità di questa componente è quella di terminare l'apprendimento dei modelli che, per una data metrica, in questo caso l'accuracy, abbiano performance inferiori rispetto agli altri modelli in corso di addestramento.

Il risultato, all'atto pratico, è quello di poter ingrandire lo spazio di ricerca della random search, in quanto non sarà più necessario attendere che tutti i modelli vengano addestrati per poter scegliere il migliore; quanto descritto risulta evidente nell figure 4a e 4b.

4 Conclusione

La valutazione dei modelli addestrati prende in considerazione le seguenti metriche:

[6] Per una chiara disamina si rimanda a [4].



(a) Evoluzione delle loss per i modelli addestrati con ASHA scheduler.

(b) Evoluzione delle accuracy per i modelli addestrati con ASHA scheduler.

Figura 4: Evoluzione dei training con ASHA scheduler, nelle immagini è chiaramente visibile come alcune delle fasi di train vengano terminate prima di altre in quanto infruttuose.

- precision: quantifica il numero di sample predetti correttamente rispetto al totale dei sample a cui è stata assegnata una certa classe.

$$Precision = \frac{TP}{TP + FP} \tag{1}$$

- recall: quantifica il numero di sample predetti correttamente rispetto al totale di quelli che sarebbero dovuti essere classificati come appartenenti a una specifica classe.

$$Recall = \frac{TP}{TP + FN} \tag{2}$$

- accuracy: indica il numero di sample predetti correttamente rispetto al totale delle classificazioni effettuate.

$$Accuracy = \frac{TP + TN}{TP + FN + TN + FP} \tag{3}$$

- F1 score: è la media armonica tra precision e recall.

$$F1 = 2 * \frac{1}{\frac{1}{Precision} + \frac{1}{Recall}} \tag{4}$$

Terminata la fase implementativa[7] è stato lanciato il tuning degli iperparametri e sono state confrontate le performance dei modelli testando i vari metodi per

[7] Lo split dei dati in train, test e validation set non è stato riportato in quanto giudicato di poca importanza. In questa sede ci limitiamo a citare come il 60% del dataset sia stato destinato alla fase di train, il 20% alla validazione della rete neurale e il restante 20% come testset.

la dimensionality reduction e lo scaling delle feature. I migliori risultati sono stati ottenuti utilizzando LDA e Normalizzazione con norma l2. Nella tabella a seguire vengono riportati i risultati della fase di addestramento.

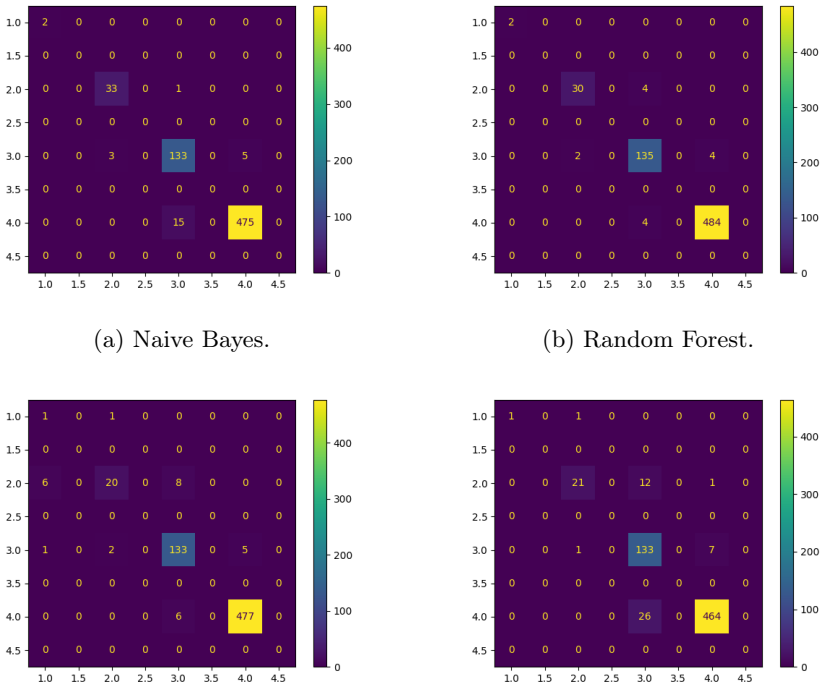
Model	**Accuracy**
Naive Bayes	0.947
Random Forest	0.995
Support Vector	0.994
Neural Network	0.984

Concluso l'addestramento, tutti i modelli sono stati testati sul test-set, i risultati per le metriche precedentemente disusse sono riportati nella tabella a seguire. Nella figura 5 è invece possibile visionare le matrici di confusione.

Model	**Precision**	**Recall**	**F1**	**Accuracy**
Naive Bayes	0.909	0.958	0.930	0.947
Random Forest	0.958	0.947	0.952	0.964
Support Vector	0.0.797	0.832	0.794	0.951
Neural Network	0.878	0.824	0.830	0.920

Riferimenti bibliografici

1. F. Maxwell Harper and Joseph A. Konstan. 2015. The MovieLens Datasets: History and Context. *ACM Transactions on Interactive Intelligent Systems (TiiS)* 5, 4: 19:1–19:19. https://doi.org/10.1145/2827872.
2. GroupLens Research. Movielens. https://movielens.org/
3. MovieLens 25M Dataset. https://movielens.org/ https://grouplens.org/datasets/movielens/25m/
4. L. Li. 2018. Massively Parallel Hyperparameter Optimization. *Machine Learning, Carnegie Mellon University.* https://blog.ml.cmu.edu/2018/12/12/massively-parallel-hyperparameter-optimization/



(a) Naive Bayes.

(b) Random Forest.

(c) Support Vector.

(d) Rete neurale.

Figura 5: Matrici di confusione per i modelli addestrati.