Data Analytics
Laurea magistrale in Informatica
Università di Bologna

Classificazione dati tabellari MovieLens

Qazim Muçodema
Keran Jegasothy

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

19-06-2022

Contents

1 Introduzione

In questo elaborato presentiamo un progetto il quale cerca di predire il voto medio di un film. Il dataset utilizzato per lo svolgimento dello studio contiene dati provenienti da MovieLens, esso contiene rating e tag per più di 60000 film e raccolti da più di 150k utenti negli anni 1996-2019. Per effettuare la nostra analisi su questi dati abbiamo costruito un unico dataset da quelli messi disponibili da MovieLens che contiene tutte le caratteristiche dei vari film e il voto medio delle valutazione di tutti gli utenti. In seguito presentiamo la metodologia e gli algoritmi implementati per creare dei modelli in grado di effettuare la previsione che ci interessa. In questo caso si tratta di modelli supervisionati in quanto oltre alle features abbiamo anche la nostra variabile di output. Infine presentiamo i risultati del nostro studio che prevede la performance dei nostri modelli.

2 Metodologia

Dall'analisi del problema abbiamo dedotto che abbiamo abbiamo a che fare con un problema di predizione. Valutando la nostra variabile di output, ossia il voto medio di un film abbiamo deciso di procedere con uno studio di regressione invece che di classificazione. In seguito precediamo con la spiegazione delle metodologie utilizzate durante lo svolgimento di questo progetto.

2.1 Data acquisition

Per quanto riguarda l'acquisizione dei dati MovieLens mette a disposizione i seguenti file in formato csv:

- **Movies.csv** dataset che contiene tutti i dati relativi a id, nome e genere di un singolo film
- **Ratings.csv dataset** che contiene tutti i dati relativi alle valutazioni di ogni singolo utente ad un film e il timestamp in cui ha effettuato la propria valutazione
- **genome-tags.csv dataset** che associa il tag id con il tag
- **genome-scores.csv** associa la rilevanza di un tag ad un singolo film

Utilizzando i seguenti file a disposizione abbiamo creato un unico dataset che utilizzeremo in seguito per i nostri modelli dove abbiamo come features le rilevanze di ogni tag e calcoliamo la media del rating utenti che rappresenta la nostra variabile di output calcolata attraverso le valutazioni di ogni singolo utente sui film che ha visualizzato.

2.2 Features primary analysis

Il dataset che abbiamo generato e che andremo ad utilizzare per costruire i modelli predittivi, contiene centinaia di features e di conseguenza abbiamo deciso di svolgere un'analisi preliminare ad alto livello per capire cosa ci rappresentano i dati e quali sono quelle features che vengono considerate importanti per la generazione del voto medio degli utenti. Per quest'analisi preliminare ci siamo concentrati sulla relazione tra le nostre variabili di input e la nostra variabile di output che vogliamo stimare. Per stimare queste relazioni abbiamo utilizzato due indici molto conosciuto che elenchiamo di seguito:

1. Correlation
2. Mutual Information

2.3 Pre-processing

Prima di procedere con la fase di modellazione abbiamo applicato le principali metodologie di pre-processing apprese durante il corso. Abbiamo gestito i valori mancanti, standardizzato, regolarizzato i dati, effettuato la principal component analysis. Per quanto riguarda il bilanciamento dei dati nella letteratura abbiamo notato che è uno step di pre-processing principalmente utilizzato per i problemi di classificazione e non di regressione, per questo motivo abbiamo deciso di non effettuare un bilanciamento del dataset in quanto come variabile di output non abbiamo delle classi.

2.3.1 Valori mancanti

All'interno del nostro dataset abbiamo analizzato la mancanza di valori tra le nostre features e abbiamo notato che per la maggior parte dei film a disposizione non erano presenti la rilevanza dei nostri tag. Questo era dovuto che non per la mancanza nei file di origine. Quindi per quei film avevamo a disposizione unicamente la valutazione media degli utenti. Abbiamo valutato necessario l'eliminazione di questi film in quanto impossibile effettuare lo studio solo con la variabile di output. Abbiamo eliminato i nostri campioni effettuando una drop dei valori null. La dimensione rimanente dei nostri campioni è di 13816.

2.3.2 Feature scaling

Lo scaling delle features viene considerato come uno degli elementi più importanti all'interno della fase di pre-processing all'interno della nostra pipeline. Nella letteratura ci sono diversi modi per normalizzare il dataset, ma tra quelli più utilizzati siamo riusciti ad individuare i seguenti:

1. Normalization o Min-Max Scaling
2. Standardization o Z-Score Normalization

Normalization utilizza la seguente formula: Xnew = (X - Xmin)/(Xmax - Xmin). Come si può notare vengono utilizzati i valori minimi e massimi e la presenza di outliers da molti problemi a questa tipologia di normalizzazione del dataset.
Dall'altra parte abbiamo la **Standardization** che utilizza la seguente formula: Xnew = (X - mean)/Std la quale è preferita in quanto non è influenzata dagli outliers come nel caso della normalizzazione.

2.3.3 Principal Component Analysis

La riduzione della dimensionalità del nostro dataset è stato un punto centrale nella fase di pre-processing in quanto la presenza di molte variabili appesantisce i modelli e li rallenta molto. Per questo motivo abbiamo utilizzato la **Principal Component Analysis** per ridurre le nostre variabili e tra varie prove abbiamo deciso di utilizzare 100 componenti che rappresentano circa il 70% del nostro dataset. Riducendo così in modo drastico la dimensionalità e mantenendo una buona rappresentazione dei dati.

2.4 Modeling

Durante la fare di *Modeling* abbiamo sperimentando con diversi modelli adatti per il problema di regressione in modo tale da individuare quale fosse il migliore per il nostro problema tra cui:

- K-Nearest Neighbour
- Support Vector Machine

- Decision tree
- Gradient Boosting Decision Tree
- Random Forest
- Feed Forward Neural Network

Prima della fase di training di questi modelli abbiamo effettuato quello che nella letteratura viene definita come **Resampling** in cui i dati a disposizione vengono suddivisi per svolgere il training e testing del modello. Abbiamo utilizzato la **K-folds cross-validation** per ottenere una valutazione più robusta di questi modelli. Per quanto riguarda invece il tuning degli hiperparametri abbiamo utilizzato **grid search** come di seguito:

2.4.1 K-Nearest Neighbour

Per il modello di K-Nearest Neighbour abbiamo cercato di effettuare il tuning dei seguenti hiperparametri:

- *numero neighbors*: range tra 2 e 15
- *weights*: uniform o distance

2.4.2 Support Vector Machine

Per il modello di Support Vector Machine abbiamo cercato di effettuare il tuning dei seguenti hiperparametri:

- *kernel*: linear, poly, rbf o sigmoid

2.4.3 Decision tree

Per il modello di Decision tree abbiamo cercato di effettuare il tuning dei seguenti hiperparametri:

- *profondità massima*: range tra 4 e 20
- *weights*: squared error, friedman mse, absolute error o poisson

2.4.4 Gradient Boosting Decision Tree

Per il modello di Gradient Boosting Decision Tree abbiamo cercato di effettuare il tuning dei seguenti hiperparametri:

- *numero alberi*: range tra 100 e 1000
- *learning rate*: range tra 0,01 e 0,1

2.4.5 Random Forest

Per il modello di Random Forest abbiamo cercato di effettuare il tuning dei seguenti hiperparametri:

- *numero alberi*: range tra 100 e 1000
- *profondità massima*: range tra 4 e 20
- *weights*: squared error, absolute error o poisson

2.4.6 Feed Forward Neural Network

Per il modello di Feed Forward Neural Network abbiamo cercato di effettuare il tuning dei seguenti hiperparametri:

- *hidden size*: 8, 16, 32
- *numero epoche*: 100, 200, 500
- *batch size*: 8, 16, 32
- *optimizer*: Momentum e Adam

3 Implementazione

Il progetto è stato sviluppato interamente in Python utilizzando le seguenti librarie: Pandas, Numpy, Scikit-learn e PyTorch. Esso consiste nei seguenti file in python:

- Import.py
- PreProcessing.py
- ClassicalPrediction.py
- MovilensDatasetClass.py
- NNArchitecture.py
- NNRegressor.py

In seguito vedremo le funzioni principali della nostra pipeline

3.1 Import.py

Questo script viene utilizzato principalmente per concatenare i dati dei vari dataset e per ottenere un dataset unico con tutte le features a nostro interesse. Nel nostro caso produrrà un dataset con tante features quanti i tag dei film e la variabile AVG_RATING che stiamo cercando di predire salvandolo in un file CSV, in questo modo ogni volta che sarà necessario il dataset sarà necessario solamente leggere il file.

3.2 PreProcessing.py

In questo script vengono effettuati varie tipologie di pre-processing per pulire i dati a nostra disposizione. La gestione dei valori mancati riguarda l'eliminazione dei campioni con valori null in quanto per molti film non sono stati associati dei tag, di conseguenza nessuno dei tag è nullo e quindi sono da togliere siccome ci troviamo nella situazione in cui questi campioni non hanno nessuna feature:

```
1 def manageMissingValues(dataset):
2     cleaned_dataset = dataset.dropna()
3     return cleaned_dataset
```

Listing 1: Gestione valori mancanti

In seguito procediamo con la normalizzazione dei dati utilizzando delle librerie su Scikit-learn. Come abbiamo indicato precedentemente abbiamo due approcci fondamentali per cui si può utilizzare la Z - score normalization o la MinMax scaling in base a cosa decide l'utente:

```
def scalingDataset(X, zscore):
    if (zscore == True):
        # Z normalization
        zscaler = StandardScaler()
        Xz = zscaler.fit_transform(X)
        X = pd.DataFrame(Xz, columns=X.columns)
    else:
        # Normalization
        scaler = MinMaxScaler(feature_range=(-1, 1))
        X_ = scaler.fit_transform(X)
        X = pd.DataFrame(X_, columns=X.columns)
    return X
```

Listing 2: Normalization

Una volta normalizzato il dataset ci troviamo in una situazione in cui abbiamo molte features e di conseguenza cerchiamo di fare un'eliminazione di quelle meno rilevanti considerando la loro varianza e la relazione con la variabile di output:

```
def featureSelection(X, Y, varianceTreshold, percentage):
    print('************************** Feature Selection -  BEGIN **********************')
    X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.30, random_state=7)
    X = lowVarianceFilter(X, X_train, varianceTreshold)
    print('-------------------------------------------------------------------------------')
    #X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.30, random_state=7)
    # X, Y = bestFeaturesFiltering(X, X_train, y_train, percentage)
    print('************************** Feature Selection -  END ************************')
    return X, Y
```

Listing 3: Feature selection

Procediamo con la regolarizzazione L2:

```
def L2_reg(X):
    x_norm1 = np.linalg.norm(X, ord=2)
    x_normalized = X / x_norm1
    return x_normalized
```

Listing 4: L1

In seguito procediamo con la principal component analysis:

```
def principalComponentAnalysis(X_train, X_test, components):
    pca = decomposition.PCA(n_components=components)
    pca.fit(X_train)
    # round the explained variance
    u = [round(i * 100, 2) for i in pca.explained_variance_ratio_]
    print("------------------------PRINCIPAL COMPONENT ANALYSIS-----------------")
    print("pca.explained_variance_ratio: ", u)
    print("sum pca explained_variance_ratio ", np.sum(u))
    X_train_t = pca.transform(X_train)
    X_test_t = pca.transform(X_test)
    #print(pd.DataFrame(pca.components_, columns=X_train.columns))
    #plt.plot(pd.DataFrame(pca.components_, columns=X_train.columns))
    # plt.plot(pca.components_)
    return X_train_t, X_test_t
```

Listing 5: PCA

3.3 ClassicalPrediction.py

In questa sezione vedremo l'implementazione degli algoritmi nominati nel capitolo precedente per la risoluzione del problema di regressione. Inizialmente applichiamo la parte di pre-processing al nostro dataset e in seguito procediamo con la fase di modeling. Abbiamo implementato un metodo per sfruttare la potenza di K-fold in modo tale da poterlo eseguire per i diversi modelli che intendiamo implementare. Oltre al cross-validation abbiamo utilizzato anche una grid - search in modo tale da poter svolgere il tuning degli hiperparametri e andare a scegliere la combinazione che offre l'accuratezza maggiore per un singolo modello:

```
def KFold_val(algorithm, hp_candidates , X_train, X_test, y_train, y_test):
    seed = 13
    kfold = KFold(n_splits=10, shuffle=True, random_state=seed)
    grid = GridSearchCV(estimator=algorithm, param_grid=hp_candidates, cv=kfold, scoring='r2')
    grid.fit(X_train, y_train)
    # Get the results
    print(grid.best_score_)
    print(grid.best_estimator_)
    print(grid.best_params_)
    # Evaluate
    predictions = grid.best_estimator_.predict(X_test)
    print("Test:", r2_score(y_test, predictions))
```

Listing 6: Feature selection

Una volta che abbiamo costruito il metodo per il cross-validation e di grid-search abbiamo costruito i seguenti metodi per ogni modello su cui intendiamo esperimentare:

```
def KNeighborsKFold(X_train, X_test, y_train, y_test):
    algorithm = KNeighborsRegressor()
    hp_candidates = [{'n_neighbors': [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 14, 15], 'weights':
    ['uniform', 'distance']}]
    KFold_val(algorithm, hp_candidates , X_train, X_test, y_train, y_test)
```

Listing 7: KNN

```
def SVRKFold(X_train, X_test, y_train, y_test):
    algorithm = SVR()
    hp_candidates = [{'kernel': ['linear']}]
    KFold_val(algorithm, hp_candidates, X_train, X_test, y_train, y_test)
```

Listing 8: SVM

```
def DecisionTreeFold(X_train, X_test, y_train, y_test):
    algorithm = DecisionTreeRegressor()
    hp_candidates = [{'max_depth': range(4,20,1), 'criterion': ['squared_error', 'friedman_mse
    ', 'absolute_error', 'poisson']}]
    KFold_val(algorithm, hp_candidates, X_train, X_test, y_train, y_test)
```

Listing 9: Decision Tree

```
def RandomForestFold(X_train, X_test, y_train, y_test):
    algorithm = RandomForestRegressor()
    hp_candidates = [{'n_estimators': range(100, 1000, 100), 'max_depth': range(4,20,4), '
    criterion': ['squared_error',  'absolute_error', 'poisson']}]
    KFold_val(algorithm, hp_candidates, X_train, X_test, y_train, y_test)
```

Listing 10: Random Forest

```
def gradientBoostingFold(X_train, X_test, y_train, y_test):
    algorithm = GradientBoostingRegressor()
    hp_candidates = {'n_estimators': range(1000, 5000, 100), 'learning_rate': [0.01, 0.1,
    0.05]}
    KFold_val(algorithm, hp_candidates, X_train, X_test, y_train, y_test)
```

Listing 11: Feature selection

3.4 MovilensDatasetClass.py

In seguito estendiamo la classe **Dataset** della libreria di Pytorch per creare la nostra classe apposita per il dataset. Richiamiamo la fase di pre-processing per avere un dataset pulito, normalizzato e standardizzato. Utilizzeremo questa classe ottimizzata per il l'implementazione della nostra rete neurale su Pytorch.

```
class MovieLensDataset(Dataset):

    def __init__(self):
        df = pd.read_csv('ImportDataset.csv')
        X, y, X_train, X_test, y_train, y_test = preprocessing(df, True, 0.02, 10, True, 20,
    True)

        X= torch.from_numpy(np.float64(X)).float()
        y = torch.from_numpy(np.float64(y.to_numpy())).float()
        print(X)
        self.num_var = 1
        self.X = torch.FloatTensor(X)
        self.y = torch.FloatTensor(y)
    def __getitem__(self, index):
        return self.X[index, :], self.y[index]

    def __len__(self):
        return len(self.X)
```

Listing 12: Feature selection

3.5 NNArchitecture.py

In seguito estendiamo la classe **Module** per costruire il nostro modello di rete neurale e utilizzarlo nello script python successivo. La nostra rete neurale sarà definita dall'oggetto model come visto di seguito.

```
class MultipleRegression(nn.Module):
    def __init__(self, input_size, hidden_size,num_var):
        super(MultipleRegression, self).__init__()
        self.input_size = input_size
        dropout = 0.2
        self.model = nn.Sequential(
            nn.Linear(input_size, hidden_size),
            nn.BatchNorm1d(hidden_size),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(hidden_size, hidden_size),
            nn.BatchNorm1d(hidden_size),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(hidden_size, hidden_size),
            nn.BatchNorm1d(hidden_size),
            nn.ReLU(),
            nn.Dropout(dropout),
```

```
            nn.Linear(hidden_size, num_var),
            nn.ReLU()

        )
    def forward(self, inputs):
        return self.model(inputs)
```

Listing 13: Feature selection

3.6 NNRegressor.py

Il dataset viene letto attraverso la nostra classe **MovieLensDataset**. In seguito suddividiamo il nostro dataset in diversi parti utilizzando la funzione Subset in quanto stiamo implementando una rete neurale su batch. Il tuning degli hiperparametri viene effettuato attraverso la variabile hyperparameters che attraverso la funzione *itertools.product* contiene tutte le possibili combinazioni degli hiperparametri. In questo modo ci è possibile effettuare il training e testing del nostro modello per ogni combinazione possibile. Si nota si cerca di garantire una replicabilità degli esperimenti fissando i seed.
Nel loop per ogni combinazione di hiperparametri viene instanziato un nuovo modello utilizzando la nostra classe **MultipleRegression** vista nello snippet precedente. Come criterio utilizziamo la MSE. Instanziamo l'ottimizzatore e procediamo con il training e il testing del modello:

```
dataset = MovieLensDataset()
train_idx, val_idx = train_test_split(np.arange(len(dataset)), test_size=0.2,  random_state
    =42)
train_subset = Subset(dataset, train_idx)
val_subset = Subset(dataset, val_idx)
val_loader = DataLoader(val_subset, batch_size=1, shuffle=True)

hyperparameters = itertools.product(hidden_size, num_epochs, batch)
for hidden_size, num_epochs, batch in hyperparameters:
    train_loader = DataLoader(train_subset, batch_size=batch, shuffle=True)
    model = MultipleRegression(dataset.X.shape[1], hidden_size, dataset.num_var)
    model.to(device)
    criterion = torch.nn.MSELoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
    #optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

    test_model(model, val_loader)
    model, loss_values = train_model(model, criterion, optimizer, num_epochs, train_loader,
    device)
    test_model(model, val_loader)

    plt.plot(loss_values)
    plt.title("Number of epochs: {} - Hidden size: {} - Mini batch: {}".format(num_epochs,
    hidden_size, batch ))
    plt.show()
```

Listing 14: Model

Per il training della rete neurale è necessario mettere il modello in modalità training attraverso la funzione *model.train*in modo tale da attivare il drop out e la batch normalization. Procediamo con il loop di training. Azzeriamo il gradiente ad ogni ciclo. Chiamiamo il modello al quale passiamo l'input al modulo forward e vacciamo inferenza in modo tale da ottenere le predizioni. In seguito calcoliamo la lost sulle predizioni utilizzando squeeze per eliminare le dimensioni in eccesso se presenti. Alla fine facciamo una *backward* per aggiornare i pesi e facciamo lo step dell'ottimizzatore per aggiornare i parametri ottenuti.

```
def train_model(model, criterion, optimizer, epochs, data_loader, device):
    model.train()
```

```
    loss_values = []
    for epoch in range(epochs):
        for data, targets in data_loader:
            data, targets = data.to(device), targets.to(device)
            optimizer.zero_grad()

            # Forward pass
            y_pred = model(data)

            # Compute Loss
            loss = criterion(y_pred.squeeze(), targets)
            loss_values.append(loss.item())
            print('Epoch {} train loss: {}'.format(epoch, loss.item()))

            # Backward pass
            loss.backward()
            optimizer.step()

    return model, loss_values
```

Listing 15: Training NN

In seguito procediamo con il testing per valutare l'accuratezza del modello:

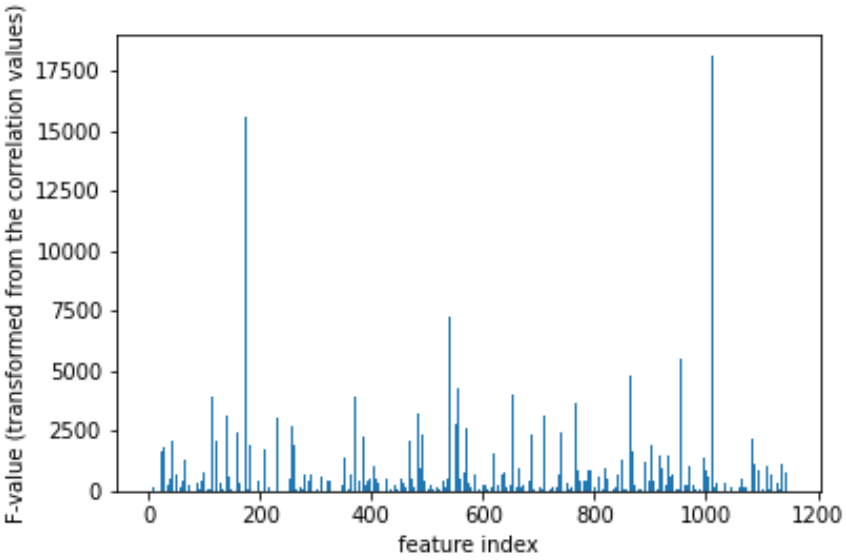
```
def test_model(model, data_loader):
    model.eval()
    y_pred = []
    y_test = []
    for data, targets in data_loader:
        y_pred.append(model(data))
        y_test.append(targets)
    y_pred = torch.stack(y_pred).squeeze()
    y_test = torch.stack(y_test).squeeze()

    print("    R2", r2_score(y_test.detach().numpy(), y_pred.detach().numpy()))
```

Listing 16: Testing NN selection

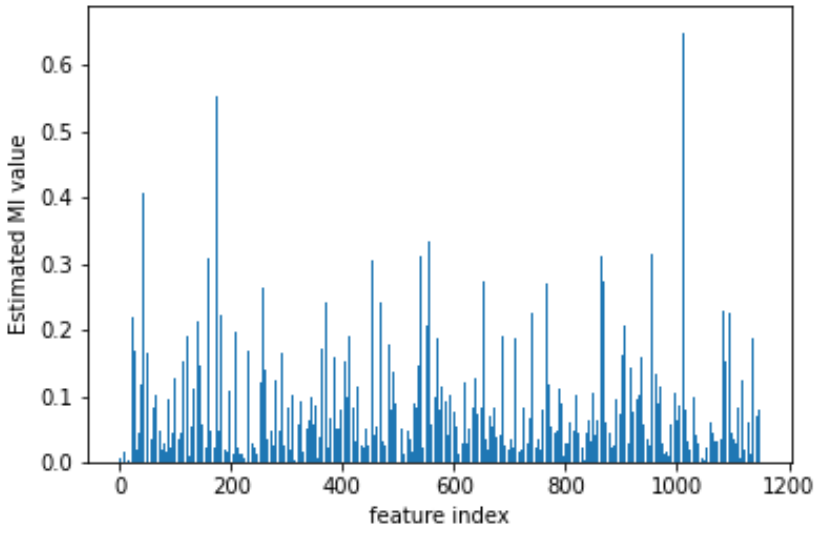
4 Risultati

4.1 Features primary analysis

Come riportato nella sezione 2.2 come analizi iniziale per conoscere meglio il dataset abbiamo utilizzati gli indici di Pearson e di Mutual Information per capire la relazione tra le diverse variabili e il nostro output. In seguito riportiamo i grafici ottenuti da questa analisi:



Dall'immagine riusciamo ad individuare in un certo modo le features più importanti e che avranno un impatto maggiore nella nostra stima del voto medio degli utenti ottenuti attraverso la corelazione di Pearson. In seguito invece consideriamo l'indice Mutual Information:



Notiamo che a differenza della corelazione abbiamo molte più variabili considerate rilevanti per il nostro problema. Questa nella nostra analisi l'abbiamo considerata un indicatore di presenza di rumore all'interno del nostro dataset e in modo tale da minimizzare questo rumore abbiamo proseguito con la fase di pre-processing.

4.2 Valutazione dei modelli

Per il confronto dei modelli abbiamo utilizzato al metrica **R2**. R2 indica il grado per il quale il modello spiega la varianza nei dati. In altre parole ci indica quanto il modello sia migliore rispetto a una predizione utilizzando il valore medio. Se il risultato è 1 possiamo dire che abbiamo una predizione perfetta, se invece è 0 significa che il modello da risultati buoni quanto prendendo la media come predittore. Se invece è negativo significa che il predittore è peggiore rispetto a prendere il valore medio.
Ora vediamo i risultati che abbiamo ottenuto tra i vari modelli in seguito al tuning dei parametri:

Modello	PCA	PCA Stand	PCA Norm	PCA Stand Reg	PCA Norm Reg
KNN	0,83	0,80	0,81	0,78	0,83
SVM	0,62	0,76	0,78	0,80	0,78
Decision Tree	0,71	0,75	0,74	0,71	0,75
Random Forest	0,85	0,84	0,86	0,85	0,85
Gradient Boosting	0,56	0,62	0,62	0,62	0,62
Neural Network	0,58	0,68	0,67	0,64	0,66

In tutti i casi è stato utilizzato la principal component analysis in quanto serve una riduzione della dimensionalità in quanto abbiamo la presenza di più di 1000 features all'interno del nostro dataset.
In termini di modelli più performanti possiamo dire che il Random Forest è stato più accurato nella predizione dei voto medio degli utenti.