Esame Data Analytics
ML-25m

Riccardo Mioli - 983525
riccardo.mioli2@studio.unibo.it

Metodologia

Data pipeline:

- **Data Acquisition**: utilizzo di Pandas per la creazione di un unico dataframe e la verifica della consistenza dei dati;
- **Data Pre-processing**: discretizzazione in bin (0.5 stelle) delle valutazioni assegnate dagli utenti , dimensionality reduction e data resampling;

- **Modeling**: modelli classici (NB, SVM, RF) e MLP;
- **Performance Analysis**: cross validation per modelli classici e train validation e test set per MLP. Metriche: precision, recall, accuracy, f1;
- **Visualization**: durante tutto lo studio.

Data Acquisition

- Il dataset a disposizione e` composto da **diversi file**, che verranno uniti mediante la funzione di **merge**.
- I generi dei film sono in formato pipe separated value, la funzione di **one-hot encoding** permette di trasformare il genere in una colonna e di utilizzare una variabile qualitativa binaria per evidenziare la presenza/assenza di un genere.

```
# One hot encoding for pipe separated genres
genres = movies["genres"].str.get_dummies()
movies = movies.merge(genres, on="movieId")
```

- Segue il merge dei generi, nel dataframe “globale” e il **calcolo del voto medio** per ogni film.

```
# Calc. avg. rating foreach movie
y = ratings.groupby("movieId")["rating"].mean()
movies = movies.merge(y, on="movieId")
```

Data Acquisition

Come ultima operazione al termine della Data Acquisition vengono svolti i seguenti controlli sulla **consistenza dei dati**:

1. Eventuale presenza di dati NA
2. Valori sotto lo 0 nel genoma/valutazioni
3. Caratteristiche con valori superiori a 1 (genomi)
4. Votazioni maggiori di 5

```
logger.info("NA values: %i." % movies.isna().sum().sum())
# Check data integrity
df_info = movies.describe().loc[["min", "max"], :]
condition0 = df_info < 0
logger.info("Values under 0: %i." % condition0.sum().sum())
condition1 = df_info.loc[:, df_info.columns != "rating"] > 1
logger.info("Characteristics over 1: %i." % condition1.sum().sum())
condition2 = df_info.loc[:, df_info.columns == "rating"] > 5
logger.info("Ratings over 5: %i." % condition2.sum().sum())
```

```
INFO:__main__:NA values: 0.
INFO:__main__:Values under 0: 0.
INFO:__main__:Characteristics over 1: 0.
INFO:__main__:Ratings over 5: 0.
```

Non risultano anomalie nelle osservazioni, nessuna operazione di data cleanup da effettuare.

Pre-Processing

Per impostare un problema di classificazione multiclasse sono stati discretizzati i voti in classi pari a 0.5 stelle.

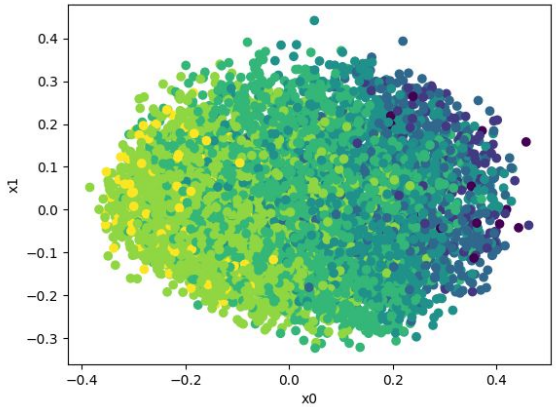
```
bins = [0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5]
binned_ratings = pd.cut(
    ratings, bins=bins, labels=bins[1:])
label_encoder = LabelEncoder()
label_encoder.fit(binned_ratings)
binned_ratings = label_encoder.transform(binned_ratings)
```

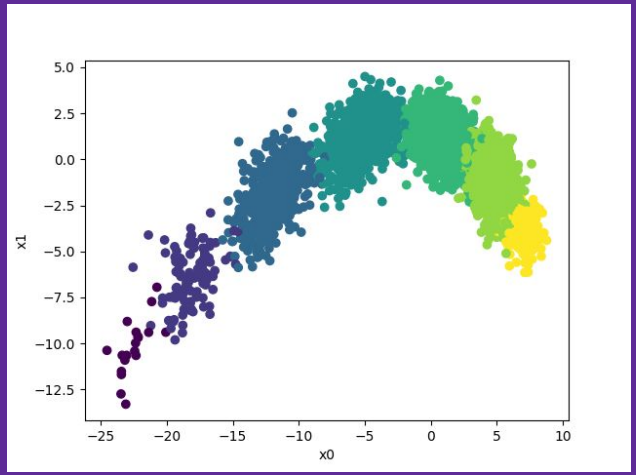
A seguire sono stati scalati i dati mediante varie tecniche, verra` mantenuta quella che garantisce le prestazioni migliori.

```
logger.info("Scaling data")
scaler = Normalizer(copy=False, norm='l2')
# scaler = StandardScaler(copy=False)
# scaler = MinMaxScaler(copy=False)
df = scaler.fit_transform(df.to_numpy())
```

Pre-Processing

La dimensionality reduction ha utilizzato due tecniche: la **principal component analysis** e la **linear discriminant analysis**.



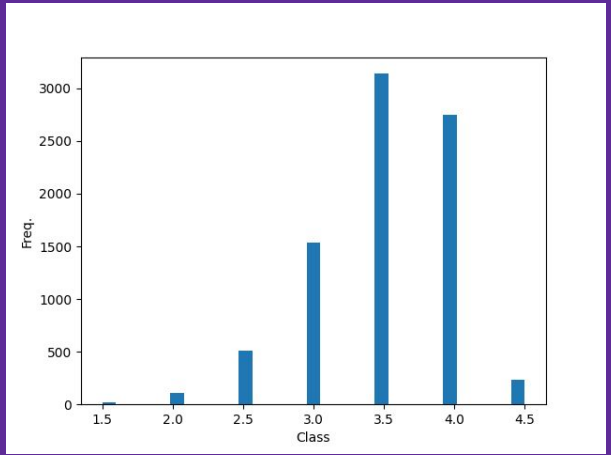


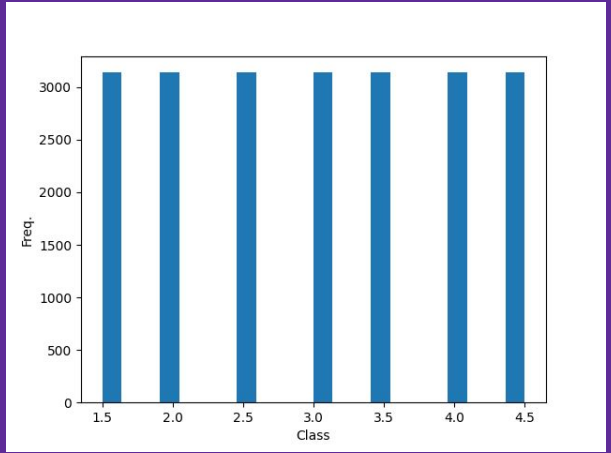
Dai due scatterplot (PCA a sinistra, LDA a destra) e` chiaramente visibile la differenza di approccio delle due tecniche di dimensionality reduction.

Pre-Processing

Da un'analisi della distribuzione delle classi emerge un importante **sbilanciamento**, pertanto vengono testati alcuni algoritmi per il ribilanciamento delle classi.

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





Modeling (Modelli classici)

Per i modelli classici l'ottimizzazione degli **iperparametri** viene gestita mediante una **random search** con **cross validation**.

Scikit implementa out of the box tutto il necessario per la random search, e` sufficiente passare in input alla funzione uno stimatore, gli iperparametri su cui effettuare la ricerca e una funzione di scoring.

```
# 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(Xr_train, yr_train)
btm['random_forest'] = rf.best_estimator_
dump(rf.best_estimator_, os.path.join(
    __DUMP_MODELS_PATH, 'random_forest.joblib'))
```

Modeling (Modelli classici)

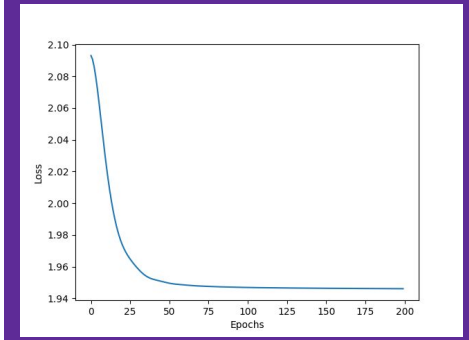
La random search e` stata impostata con **10 iterazioni sulla matrice degli iperparametri** e **5 fold di cross-validation per ogni combinazione** di parametri.
Viene tenuto lo stimatore che ha avuto la **migliore accuracy** media nelle 5 cross validazioni.

```
Fitting 5 folds for each of 10 candidates, totalling 50 fits
INFO:__main__:Random forest best params: {'n_estimators': 200, 'min_samples_split': 4, 'max_depth': 15, 'criterion': 'gini'}
Random forest accuracy score: 0.995223
```

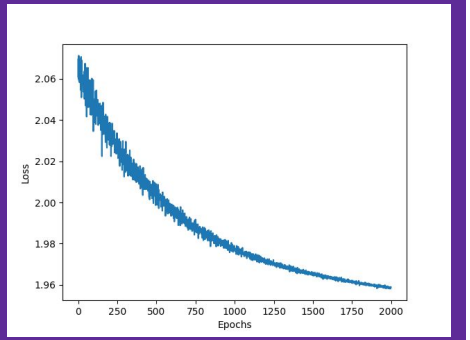
Modeling (MLP)

L'implementazione della rete neurale e` simile a quella vista durante le lezioni del corso, il **processo** e` stato **iterativo**.

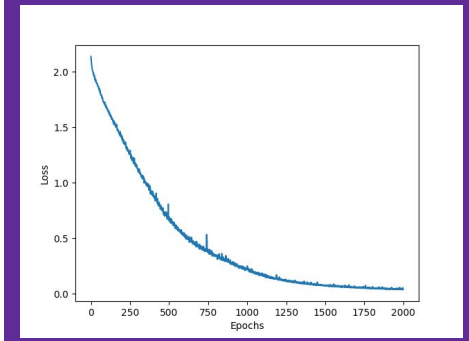
1. Stocastic gradient descent
2. Minibatch
3. Batchnorm
4. Dropout
5. Early stopping



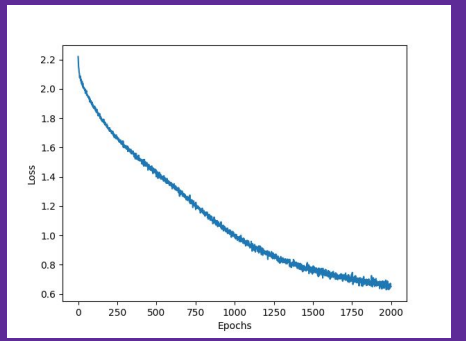
SGD



Minibatch



Batchnorm



Dropout

Modeling (MLP)

Per l'ottimizzazione degli iperparametri con random search, **Pytorch** non offre un modulo integrato; per questo motivo l'ottimizzazione viene effettuata tramite **ray-tune**.

ray-tune **non gestisce la cross validation**, per semplicita` e` stato scelto l'approccio **train-test-validation set**.

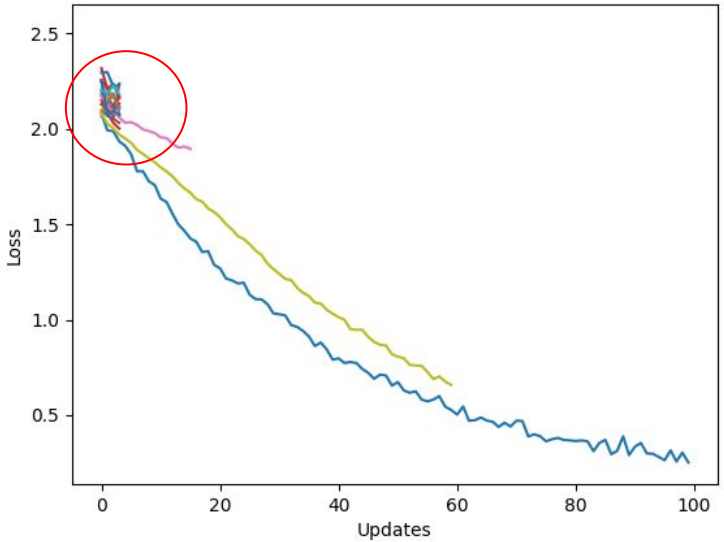
In ray-tune viene definita una **funzione di train** (diversa da quella di Pytorch) e una metrica da massimizzare. Viene mantenuto il modello che garantisce le migliori performance.

Modeling (MLP)

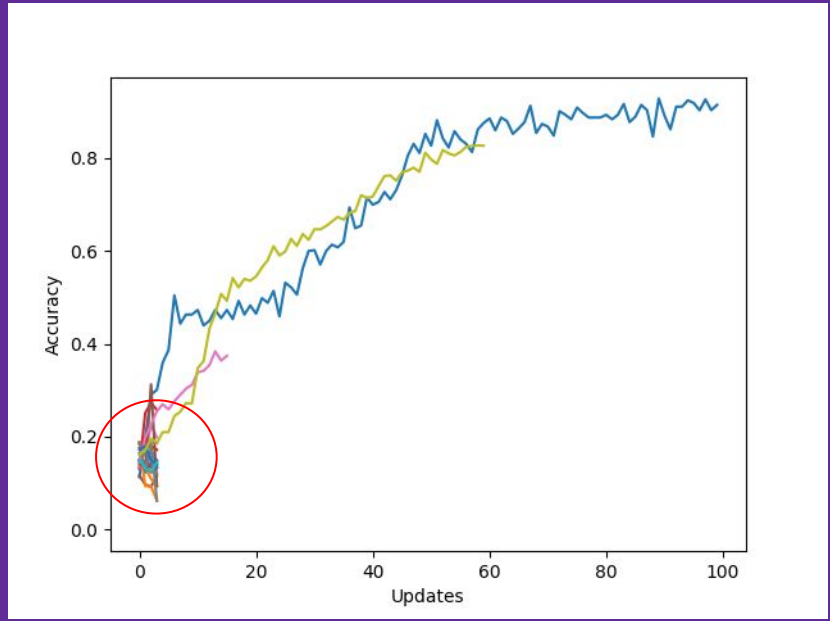
```
def train_nn(config: dict, X_train, y_train):
    train_loader = DataLoader(
        Dataset(X_train, y_train), config['batch_size'], shuffle=True, drop_last=True)
    val_loader = DataLoader(Dataset(X_val, y_val), X_val.shape[0])
    mlp = NeuralNetwork(X_train.shape[1], config['hidden_layer_size'], 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)
    # Local plot (just for this specific training session)
    plt.plot(range(0, len(loss)), loss)
    plot(["Updates", "Loss"], "mlp_loss_progr_minib_bnorm_drop")
    return mlp

ray.init(log_to_driver=False, logging_level=logging.CRITICAL)
results = tune.run(tune.with_parameters(train_nn, X_train=X_train, y_train=y_train), config=configs,
                   scheduler=ASHAScheduler(metric="accuracy", mode="max"),
                   local_dir=os.path.realpath("."), verbose=0,
                   search_alg=BasicVariantGenerator(random_state=np.random.RandomState(__SEED)), num_samples=50, resources_per_trial=tune_res)
```

L’addestramento di una rete neurale puo` richiedere molto tempo, per questo e` stato utilizzato l’ASHA scheduler.

Modeling (MLP) - ASHA Scheduler



Progressione Loss



Progressione Accuracy

Performance evaluation

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

Fase di train dei modelli

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

Test dei modelli addestrati