# MACHINE LEARNING & DATA MINING

Tecniche di machine learning per lo studio di qualità di un vino

Studente:
FRANCESCO ROSSI

Docente:
Prof. Ivan Serina
Prof. Alfonso Emilio Gerevini

# INDICE

# INTRODUZIONE

Il vino (dal latino vinum) è una bevanda alcolica ottenuta dall'uva, generalmente Vitis vinifera, fermentata senza l'aggiunta di zuccheri, acidi, enzimi, acqua o altri nutrienti.

Il vino è prodotto da migliaia di anni. Le prime tracce conosciute di vino provengono dalla Georgia (cca. 6000 a.C.), dall'Iran (cca. 5000 a.C.) e dalla Sicilia (cca. 4000 a.C.), sebbene vi siano prove di una bevanda alcolica simile consumata in precedenza in Cina (cca. 7000 a.C). La prima azienda vinicola conosciuta è la cantina Areni-1 di 6.100 anni in Armenia. Il vino raggiunse i Balcani nel 4500 aC ed era consumato e celebrato nell'antica Grecia, Tracia e Roma. Nel corso della storia, il vino è stato consumato per i suoi effetti inebrianti.

Il vino ha svolto a lungo un ruolo importante nella religione. Il vino rosso era associato al sangue dagli antichi egizi ed era usato sia dal culto greco di Dioniso che dai romani nei loro Baccanali; L'ebraismo lo incorpora anche nel Kiddush e il cristianesimo nell'Eucaristia.

Il lievito consuma lo zucchero dell'uva e lo converte in etanolo e anidride carbonica. Diverse varietà di uva e ceppi di lieviti producono diversi stili di vino. Queste variazioni derivano dalle complesse interazioni tra lo sviluppo biochimico dell'uva, le reazioni coinvolte nella fermentazione, il terroir e il processo produttivo. Molti paesi promuovono denominazioni legali volte a definire stili e qualità del vino. Questi in genere limitano l'origine geografica e le varietà di uva consentite, nonché altri aspetti della produzione del vino. I vini non ottenuti da uve includono vino di riso e vini di frutta come prugna, ciliegia, melograno e sambuco.

Il seguente lavoro si occupa creare e testare la bontà di vari sistemi e modelli del machine learning tramite l’ausilio di un dataset, per stabilire la qualità di un vino sulla base dei suoi parametri e componenti chimici.

**Panoramica sul caso di studio**

Il nostro campo di studio riguarda la qualità del vino, prodotto in tutto il mondo. Il vino appartiene alla famiglia degli alcolici ed è considerato anche parte della ricca cultura. Ci sono vari benefici per la salute del vino (http://www.wideopeneats.com/10-health-benefits-get-drinking-daily-glass-wine/).

Esistono un gran numero di occupazioni e professioni che fanno parte dell'industria del vino, che vanno dalle persone che coltivano l'uva, preparano il vino, lo imbottigliano, lo vendono, lo valutano, lo commercializzano e infine danno consigli ai clienti e servono il vino. In questo studio, scopriamo importanti componenti chimici correlati del vino. Acidi importanti che sono associati alla qualità del vino. Troveremo i giusti componenti che servono nel giusto rapporto per fare un vino di qualità.

**Valutazione dei risultati**

La metrica di valutazione utilizzata è il *Mean F1-Score*. L'F1-Score, comunemente usato nel recupero delle informazioni, misura l'accuratezza utilizzando le misure precisione statistica p e la recall r. La precisione è il rapporto tra i veri positivi (tp) e tutti i positivi previsti (tp + fp). Il recall è il rapporto tra i veri positivi e tutti i positivi effettivi (tp + fn). Il punteggio F1 è dato da:

$$F1 = 2\frac{p \cdot r}{p + r} \text{ where } p = \frac{tp}{tp + fp}, \quad r = \frac{tp}{tp + fn}$$

F1 metric pesa la recall e la precisione equamente, e un buon algoritmo massimizzerà entrambe le misure contemporaneamente.

Pertanto, prestazioni moderatamente buone su entrambi saranno favorite rispetto a prestazioni estremamente buone su uno e prestazioni scarse sull'altro.

# Capitolo 1
## FASE DI PRE-PROCESSING

La prima fase del lavoro è stata svolta analizzando il dataset iniziale. Il dataset utilizzato comprendeva 13 colonne e circa 4000 righe.

Le colonne erano così identificate:

- Id
- *Fixed acidity* - la maggior parte degli acidi coinvolti nel vino o fissi o non volatili (non evaporano facilmente).
- *volatile acidity* - la quantità di acido acetico nel vino, che a livelli troppo elevati può portare a uno sgradevole sapore di aceto.
- *citric acid* - trovato in piccole quantità, l'acido citrico può aggiungere 'freschezza' e dare sapore ai vini.
- *residual sugar* - la quantità di zucchero che rimane dopo l'arresto della fermentazione, è raro trovare vini con meno di 1 grammo/litro e vini con più di 45 grammi/litro vengono considerati dolci.
- *chlorides* – la quantità di sale nel vino.
- *free sulfur dioxide* - la forma libera di SO2 che esiste in equilibrio tra SO2 molecolare (come gas disciolto) e ione bisolfito; previene la crescita microbica e l'ossidazione del vino.
- *total sulfur dioxide* - quantità di forme libere e vincolate di S02; a basse concentrazioni, l'SO2 è per lo più non rilevabile nel vino, ma a concentrazioni di SO2 libera superiori a 50 ppm, l'SO2 diventa evidente nel naso e nel gusto del vino.
- *density* - la densità dell'acqua è vicina a quella del vino a seconda della percentuale di alcol e del contenuto di zucchero.
- *pH* - descrive quanto è acido o basico un vino su una scala da 0 (molto acido) a 14 (molto basico); la maggior parte dei vini sono tra 3-4 sulla scala del pH.

- *sulphates* - un additivo per il vino che può contribuire ai livelli di anidride solforosa (S02), che agisce come antimicrobico e antiossidante.
- *alcohol* – la percentuale di alchol contenuta nel vino.
- *quality* - 0 -- Disappointing; 1 – Good

Analizzando il dataset si è subito visto che alcune colonne contenevano valori riportati in scale diverse (es. alcuni valori riportati in kg mentre altri in g), si è quindi proceduto a ri-scalare tutti i valori su una scala comune per ciascuna colonna. Questa operazione è stata fatta direttamente su file.

Successivamente si è andati a cercare le colonne considerate inutili o meno importati ai fini dello studio, per questo è stata rimossa la colonna denominata id (non conteneva informazione).

Infine, il dataset conteneva alcuni dati mancanti che sono stati riempiti, in questo caso, con un valore medio. In certi casi la fase di pre-procesing è stata estesa a scalare i dati prima di essere passati ai metodi, tramite lo standardScaller o con il minMaxScaller (tra 0 e 1).

# Capitolo 2
## ANALISI DELLE TECNICHE UTILIZZATE

Vediamo ora le varie tecniche di machine learning utilizzate per lo studio della qualità con le relative implementazioni.

Prima di tutto il dataset è stato diviso in due: l'80% usato come training set e il restante 20 come validation set.

```
[5]  xTrain, xTest, yTrain, yTest = train_test_split(train_X_imp, train_y, train_size = 0.8, random_state = 0)
```

### 2.1 **Alberi di decisione**

L'albero di decisione è la prima tecnica che è stata vista ed utilizzata. Solitamente più un albero è lungo più è specializzato ed il rischio di overfitting aumenta. Gli alberi di decisione non hanno portato a grandi risultati, sono stati fatti variare principalmente gli iper-parametri relativi all'altezza dell'albero, al max numero di foglie, numero di samples minime per foglia. Si è visto come aumentando l'altezza dell'albero i risultati miglioravano, in particolare in combinazione con il numero massimo di nodi foglia.

Come criterio si è sempre usato il 'gini' perché il cambio di esso non portava a nessuna variazione nei risultati finali. Vediamo il risultato migliore ottenuto:

```
my_model = DecisionTreeClassifier(random_state=0,criterion='gini',
                                  splitter='best',max_depth=14,max_leaf_nodes=300,
                                  min_samples_leaf=2)
my_model.fit(xTrain, yTrain)
print(my_model.score(xTrain, yTrain))
print(my_model.score(xTest,yTest))
```

```
y_pred = my_model.predict(xTest)
print("Confusion Matrix:")
print(confusion_matrix(yTest, y_pred))
print("Classification Report:")
print(classification_report(yTest, y_pred), "\n\n")
```

```
Confusion Matrix:
[[172  78]
 [ 78 370]]
Classification Report:
              precision    recall  f1-score   support

           0       0.69      0.69      0.69       250
           1       0.83      0.83      0.83       448

    accuracy                           0.78       698
   macro avg       0.76      0.76      0.76       698
weighted avg       0.78      0.78      0.78       698
```

Risultati sul testset finale:

| (Al 30%)  public score | (Al 100%) private score |
|---|---|
| 0.77363 | 0.78404 |

### 2.2 **SVM**

Oltre alla classificazione lineare è possibile fare uso delle SVM per svolgere efficacemente la classificazione non lineare utilizzando il metodi *kernel*, mappando implicitamente i loro ingressi in uno spazio delle caratteristiche multi-dimensionale.

La tecnica delle support vector machines ha portato a risultati leggermente migliori degli alberi, i test sono stati eseguiti con vari kernel, polinomiale, rbf, lineare, sigmoide, combinati con altri iper-parametri principali. Prima però, il dataset è stato opportunamente scalato e standardizzato.

- Il kernel polinomiale con gradi 2 e 3 hanno portato a risultati simili tra di loro intorno al 77%, anche se il grado 2 migliore di poco del 3. In genere aumentando il grado si riesce ad adattare meglio la funzione al problema, ma troppo adattamento porta all'overfitting.
- Il kernel lineare non è adatto per questo tipo di problema risulta troppo semplice, e neanche variando il parametro C si sono ottenuti risultati validi.
- Il sigmoide non è stato molto efficace, si sono registrati risultati intorno al 76%.
- Il risultato migliore è stato usando il kernel rbf, con coefficiente C di 5.

Il coefficiente C è legato al margine, per valori elevati di C, l'ottimizzatore sceglierà un iperpiano con margine più piccolo, questo significa una maggiore rigidità nella classificazione. Al contrario, un valore molto piccolo di C farà sì che l'ottimizzatore cerchi un iperpiano di separazione con margini maggiori, ottenendo un modello più "elastico".

I parametri "degree" e "coef0" vanno presi in considerazione solo nel caso di kernel polinomiale. Per il kernel sigmoide è necessario il parametro "coef0", che in generale indica il termine noto. Per tentare di ottenere un buon risultato è stata utilizzata anche la tecnica della grid search. Questo però, in relazione al tempo di ricerca, che aumenta all'aumentare delle dimensioni della griglia, non ha portato benefici, considerando che i risultati ottenuti con le SVM si sono registrati anche con un semplice albero. Perciò il trade-off è stato negativo. Di seguito sono riportati i 2 risultati migliori trovati:

```
svm = SVC(kernel="rbf", random_state=0, degree=12, C=5,
          gamma='auto', coef0=100, decision_function_shape="ovr")
svm.fit(xTrain, yTrain)
```

```
y_pred = my_model.predict(xTest)
print("Confusion Matrix:")
print(confusion_matrix(yTest, y_pred))
print("Classification Report:")
print(classification_report(yTest, y_pred), "\n\n")
```

```
Confusion Matrix:
[[163  87]
 [ 67 381]]
Classification Report:
              precision    recall  f1-score   support

           0       0.71      0.65      0.68       250
           1       0.81      0.85      0.83       448

    accuracy                           0.78       698
   macro avg       0.76      0.75      0.76       698
weighted avg       0.78      0.78      0.78       698
```

Risultati sul testset finale:

| (Al 30%) public score | (Al 100%) private score |
|---|---|
| 0.77936 | 0.79631 |

Secondo test con kernel rbf:

```
svm = SVC(kernel="rbf", random_state=0, degree=12, C=7,
          gamma='scale', coef0=500, decision_function_shape="ovr")
svm.fit(xTrain, yTrain)
```

```
Confusion Matrix:
[[167  83]
 [ 67 381]]
Classification Report:
              precision    recall  f1-score   support

           0       0.71      0.67      0.69       250
           1       0.82      0.85      0.84       448

    accuracy                           0.79       698
   macro avg       0.77      0.76      0.76       698
weighted avg       0.78      0.79      0.78       698
```

Risultati sul testset finale:

| (Al 30%)  public score | (Al 100%) private score |
|---|---|
| 0.77363 | 0.80245 |

### 2.3 **Logistic regression**

La regressione logistica ha portato scarsi risultati, tutti al di sotto del 70%, con tutti i tipi di solver quali: newton, lbfgs, liblinear, sag, saga, e varie combinazioni di parametri provati. Sono stati fatti variare i seguenti parametri: il tipo di solver, il num. di iterazioni del solver, la tolleranza legata allo stop, il parametro C (lo stesso delle SVM) per il quale riducendolo aumenta la regolarizzazione, viceversa porta a risultati più elastici. Alla luce dei risultati possiamo affermare che questa tecnica è da scartare su questo tipo di problema. Vediamo qui sotto un esempio di un risultato ottenuto:

```
 #  solver usati : {'newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga'}
log_reg = LogisticRegression(solver="newton-cg", random_state=0,
                             max_iter=5000,  C=0.500, tol=0.0001)
print(log_reg)
my_model = log_reg
my_model.fit(xTrain, yTrain)
print(my_model.score(xTrain, yTrain))
print(my_model.score(xTest, yTest))
```

```
LogisticRegression(C=0.5, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=5000,
                   multi_class='auto', n_jobs=None, penalty='l2',
                   random_state=0, solver='newton-cg', tol=0.0001, verbose=0,
                   warm_start=False)
0.7380867072733788
0.7063037249283668
```

Risultati sul testset finale:

| (Al 30%)  public score | (Al 100%) private score |
|---|---|
| 0.64328 | 0.66721 |

### 2.4 **Reti neurali**

Le reti neurali hanno prodotto discreti risultati, anche se il numero di parametri e variabili da prendere in considerazione è stato elevato. Proprio per questo, considerando i risultati in linea con gli altri metodi gli si può attribuire una trade-off negativo. Innanzitutto, è stato necessario implementare a mano la misura “f1-score” perché non implementata direttamente nel pacchetto Tensorflow:

```
def recall_m(y_true, y_pred):
    y_true = K.ones_like(y_true)
    true_positives = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
    all_positives = K.sum(K.round(K.clip(y_true, 0, 1)))

    recall = true_positives / (all_positives + K.epsilon())
    return recall

def precision_m(y_true, y_pred):
    y_true = K.ones_like(y_true)
    true_positives = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))

    predicted_positives = K.sum(K.round(K.clip(y_pred, 0, 1)))
    precision = true_positives / (predicted_positives + K.epsilon())
    return precision

def f1_score(y_true, y_pred):
    precision = precision_m(y_true, y_pred)
    recall = recall_m(y_true, y_pred)
    return 2*((precision*recall)/(precision+recall+K.epsilon()))
```

Le reti sono state testate in varie configurazioni con varie funzioni di attivazione: ***sigmoid***, ***relu***, ***tanh, elu, exponential***, (alcune si sono rivelate molto meno efficaci di altre), vari livelli di profondità, un numero di unità variabile (scegliendo casualmente sia profondità che num. di unità), e vari ottimizzatori. Risultati migliori si sono avuti aumentando il “batch_size”. Inoltre, aumentando il “batch size” il tempo di addestramento si riduceva perché venivano presi in considerazione meno insiemi e quindi meno operazioni, e i risultati nel complesso miglioravano.

Mentre diminuendo il “batch size” i tempi aumentavano, la funzione f1 risultava più instabile e con molte oscillazioni.

Aumentando il learnig rate invece le funzioni accuracy, e f1 erano più instabili rispetto a learnig rate bassi, ma le distanze tra le prove su train e validation erano spesso più vicine. Learning rete bassi hanno portato a funzioni risultanti più stabili e migliori, ma le distanze tra i 2 set erano più marcate.

Ad esempio l’utilizzo di learning rate tra lo 0.001-0.015 hanno portato i risultati migliori sul f1 score, circa 80%, con “batch_size”=200-300, “epoch”=500-800 e loss function=BinaryCrossentropy.

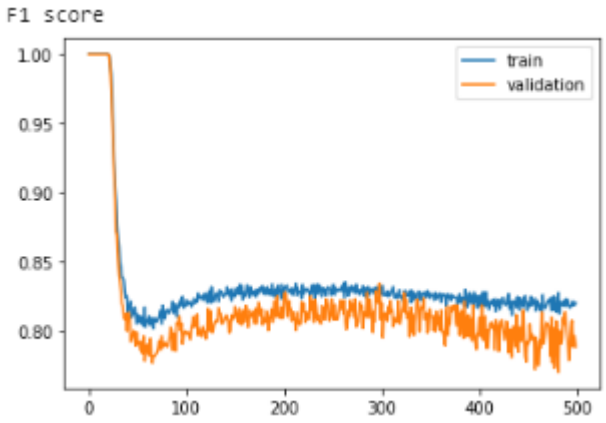
Una comparazione tra ottimizzatori quali: adadelta, adam, adagrad e RMSprop, fissati i parametri di batch, epoche, Lrate come sopra, ha portato a risultati tutti molto vicini tra loro, con la differenza che con i primi due le funzioni f1 e accuracy erano più stabili e con meno oscillazioni rispetto a RSMSprop. Adagrad ha invece presentato risultati che stanno più nel mezzo rispetto ai 2 casi descritti.

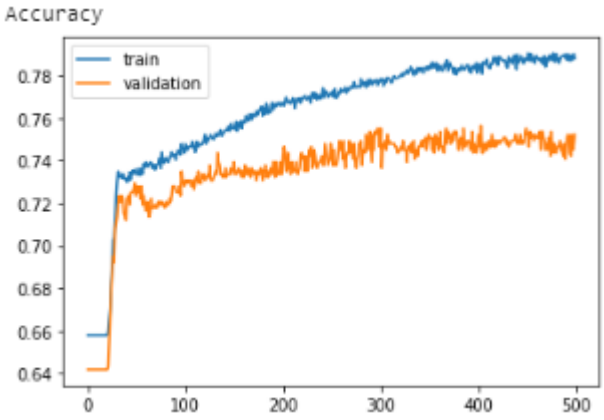
Di seguito è riportato uno dei risultati ottenuti, non il migliore. Per i migliori ottenuti, si è usata sempre la stessa rete che è la seguente:

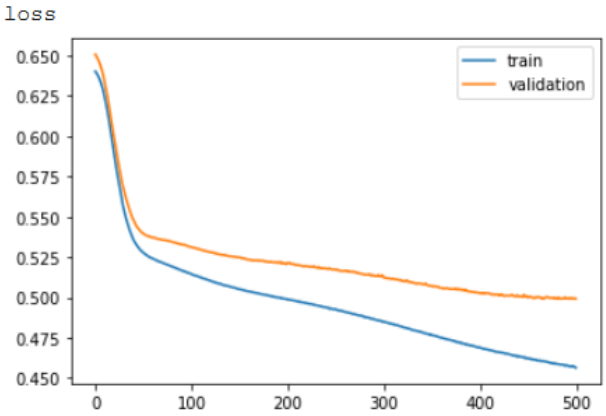
```
#Initialize the constructor
model = Sequential()
#Add an input layer
model.add(Dense(12, activation='sigmoid', input_shape=(11,)))
#Add one hidden layer
model.add(Dense(11, activation='sigmoid'))
#Add one hidden layer
model.add(Dense(4, activation='sigmoid'))
#Add an output layer
model.add(Dense(1, activation='sigmoid'))
```

```
model.compile(loss='binary_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy',tf.keras.metrics.AUC(), f1_score])

training_phase =model.fit(xTrain, yTrain ,epochs=500,
                          batch_size=150,  validation_data=(xTest,yTest),
                           verbose=1)
```






Risultati finali:

| (Al 30%) public score | (Al 100%) private score |
|---|---|
| 0.77936 | 0.78404 |

2.5 **Random forest**

La tecnica della random forest rappresenta un tipo di modello ensemble, che si avvale di più decision tree. Ciò significa che una random forest combina molti alberi decisionali in un unico modello.

La tecnica si è rivelata molto efficace per la classificazione del problema proposto, infatti si sono registrati risultati migliori della media in tutti i test, facendo variare principalmente: la profondità, il max numero di nodi foglia, max samples, numero di stimatori.

```
RandomForestClassifier(bootstrap=True,ccp_alpha=0.0,class_weight=None,
                       criterion='gini',max_depth=15,
                       max_features='auto',max_leaf_nodes=300,max_samples=1500,
                       min_impurity_decrease=0.0,min_impurity_split=None,
                       min_samples_leaf=1,min_samples_split=2,min_weight_fraction_leaf=0.0,
                       n_estimators=2000, n_jobs=None, oob_score=False,
                       random_state=42,)
```

```
Confusion Matrix:
[[174  76]
 [ 49 399]]
Classification Report:
              precision    recall  f1-score   support

           0       0.78      0.70      0.74       250
           1       0.84      0.89      0.86       448

    accuracy                           0.82       698
   macro avg       0.81      0.79      0.80       698
weighted avg       0.82      0.82      0.82       698
```

Risultati finali:

| (Al 30%)  public score | (Al 100%) private score |
|---|---|
| 0.80802 | 0.82208 |

Questo è un risultato leggermente migliore del precedente, notare l'incremento sostanzioso di stimatori e l'attivazione del oob:

```
RandomForestClassifier(n_estimators=8000,max_leaf_nodes=300,random_state=42,oob_score=True,
                       max_samples=2000,max_depth=None,min_samples_split=2))
```

```
Confusion Matrix:
[[172  78]
 [ 47 401]]
Classification Report:
              precision    recall  f1-score   support

           0       0.79      0.69      0.73       250
           1       0.84      0.90      0.87       448

    accuracy                           0.82       698
   macro avg       0.81      0.79      0.80       698
weighted avg       0.82      0.82      0.82       698
```

Risultati finali:

| (Al 30%)  public score | (Al 100%) private score |
|---|---|
| 0. 81948 | 0. 82822 |

## 2.6 **Boosting**

Nel Boosting più modelli vengono generati consecutivamente dando sempre più peso agli errori effettuati nei modelli precedenti. In questo modo si creano modelli via via più "attenti" agli aspetti che hanno causato inesattezze nei modelli precedenti, ottenendo infine un modello aggregato avente migliore accuratezza di ciascun modello che lo costituisce, ma con un alto rischio di overfitting.

La tecnica del boosting è stata utilizzata con l'algoritmo *Ada*, ed è la tecnica con la quale si è anche ottenuto uno dei risultati migliori tra tutti i modelli provati. E' stato utilizzato come modello di base un extra tree, un classificatore simile alla random forest, che a differenza di questo non utilizza la tecnica del bootstrap e in fase di split va a fare una selezione random e non sul valore ottimo.

```
my_model= ExtraTreesClassifier(n_estimators=2000, criterion='gini', max_depth=30, min_samples_split=2,
                               min_samples_leaf=2, min_weight_fraction_leaf=0.0, max_features='auto',
                               max_leaf_nodes=300, min_impurity_decrease=0.0,bootstrap=True,
                               oob_score=False,random_state=42, class_weight=None, max_samples=None)
```

```
ada_clf = AdaBoostClassifier(base_estimator= my_model, n_estimators=500,
                             algorithm="SAMME", learning_rate=0.5, random_state=42)
ada_clf.fit(xTrain, yTrain)
```

```
Confusion Matrix:
[[172  78]
 [ 42 406]]
Classification Report:
              precision    recall  f1-score   support

           0       0.80      0.69      0.74       250
           1       0.84      0.91      0.87       448

    accuracy                           0.83       698
   macro avg       0.82      0.80      0.81       698
weighted avg       0.83      0.83      0.82       698
```

Risultati finali:

| (Al 30%)  public score | (Al 100%) private score |
|---|---|
| 0.83190 | 0.82808 |

2.7 **Voting**

Il voting è una tecnica che si serve di un insieme di classificatori per fare una predizione finale basata sulla loro probabilità più alta di scegliere una classe come output. Aggrega i risultati di ciascun classificatore utilizzato e prevede la classe di output in base alla maggior parte dei voti.

Sono state usate 2 tecniche di voto Hard e Soft:

-Hard voting: la predizione di output è quella che riceve più voti, cioè la classe che ha la probabilità più alta di essere predetta.

-Soft voting: la classe di output è la predizione basata sulla media delle probabilità assegnate alle varie classi da ogni predittore.

Il miglior risultato ottenuto dopo vari test è stato quello di utilizzare come classificatori Ada boost, random forest e SVM, tramite la tecnica del soft voting.

```
ada_clf = AdaBoostClassifier( RandomForestClassifier(n_estimators=5000, max_leaf_nodes=320,
                              random_state=42,oob_score= True,bootstrap=True, max_samples= 2500,
                              max_depth=25,min_samples_split=2), n_estimators=1000,
                              algorithm="SAMME.R", learning_rate=0.05, random_state=42)
rnd_clf = RandomForestClassifier(n_estimators=10000, max_leaf_nodes=320, random_state=42,
                                 oob_score= True,bootstrap=True, max_samples= 1500,
                                 max_depth=25,min_samples_split=2)
svm_clf = Pipeline([ ("scaler", StandardScaler()),
                     ("svm_clf", SVC(kernel="rbf", C=7,gamma='scale',
                   probability=True, coef0=500))])

voting_clf = VotingClassifier(
    estimators=[('ada', ada_clf), ('rf', rnd_clf), ('svc', svm_clf)],
    voting='soft')
```

```
Confusion Matrix:
[[170  80]
 [ 53 395]]
Classification Report:
              precision    recall  f1-score   support

           0       0.76      0.68      0.72       250
           1       0.83      0.88      0.86       448

    accuracy                           0.81       698
   macro avg       0.80      0.78      0.79       698
weighted avg       0.81      0.81      0.81       698
```

Risultati finali:

| (Al 30%)  public score | (Al 100%) private score |
|---|---|
| 0. 80981 | 0. 79083 |

## 2.8 **Nearest Neighbors**

La tecnica del nearest neighbors è stata testata tramite l'algoritmo K- nearest neighbors. In questo caso l'apprendimento è basato sui k punti più vicini ad ogni "query point". I risultati ottenuti sono stati sempre sotto la media, di seguito viene comunque riportato uno dei tentativi che hanno ottenuto un "miglior" punteggio, utilizzando l'algoritmo "kd tree", gli altri utilizzati sono stati: "ball_tree", "Brute". E' stata compiuta una ricerca più completa anche utilizzando una grid search combinando parametri quali: algoritmo, numero di vicini, "p" il tipo di distanza, weights e "leaf size". Una ricerca del genere allunga i tempi ma permette di creare tutte le combinazioni tra i parametri e restituisce la combinazione che ha ottenuto il miglior risultato.

```
my_model=KNeighborsClassifier(n_neighbors=10,weights='distance',
                          n_jobs=-1,leaf_size=300, p=1,algorithm='kd_tree')

my_model.fit(xTrain,yTrain)
```

```
my_model.score(xTrain,yTrain)
```

```
1.0
```

```
my_model.score(xTest,yTest)
```

```
0.7421203438395415
```

```
Confusion Matrix:
[[142 108]
 [ 72 376]]
Classification Report:
              precision    recall  f1-score   support

           0       0.66      0.57      0.61       250
           1       0.78      0.84      0.81       448

    accuracy                           0.74       698
   macro avg       0.72      0.70      0.71       698
weighted avg       0.74      0.74      0.74       698
```

## 2.9 **XGBoost**

XGBoost è un'implementazione specifica del metodo Gradient Boosting che utilizza approssimazioni più accurate per trovare il miglior modello ad albero. Un vantaggio è che è un metodo "regolarizzato" e questo evita spesso l'overfitting. Il classificatore ha ottenuto prestazioni tutto sommato buone. Qui riportato un esempio di esecuzione.

```
my_model = XGBClassifier(max_depth=100, learning_rate=0.01, n_estimators=5000,silent=None,
                         objective='binary:logistic',booster='gbtree', n_jobs=1, nthread=None, gamma=1,
                         min_child_weight=1, max_delta_step=0, subsample=1, colsample_bytree=1,
                         colsample_bylevel=1, colsample_bynode=1,reg_alpha=0, reg_lambda=1, scale_pos_weight=1,
                         base_score=0.1, random_state=42,)
my_model.fit(xTrain, yTrain)
```

```
Confusion Matrix:
[[172  78]
 [ 55 393]]
Classification Report:
              precision    recall  f1-score   support

           0       0.76      0.69      0.72       250
           1       0.83      0.88      0.86       448

    accuracy                           0.81       698
   macro avg       0.80      0.78      0.79       698
weighted avg       0.81      0.81      0.81       698
```

# CONCLUSIONI

Dai risultati ottenuti si può notare come i metodi di insieme siano stati molto più performanti dei classificatori singoli per via della loro robustezza, ed in generale i metodi basati su alberi si sono comportati meglio rispetto agli altri, in termini di risultati finali, considerando anche i tempi di attesa. Possiamo dire che per questo tipo di dataset sono stati un buon trade-off.

I risultati migliori si sono avuti con l'adaboost applicando un extra-tree come modello di base e con una random forest. Le prestazioni peggiori invece, si sono registrate con la logistic regression e con una SVM che utilizza il kernel lineare.

Infine, per tentare di ottenere risultati migliori si potrebbe procedere con una fase di pre-processing dei dati più robusta, ed in secondo luogo esplorare più nel dettaglio le architetture basate su reti neurali.