DMML 2024: Final Project

Francesco Peria, Maria Paola Sforza Fogliani, Andrea Vitali

In [33]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import sklearn
import seaborn as sns
```

In [35]:
```
pd.set_option('display.max_columns', 100)
df = pd.read_csv('A) Data Understanding & Preparation Output.csv', sep=',', skipinitialspace=True)
df.head()
```

Out[35]:

	emotion	vocal_channel	emotional_intensity	statement	repetition	sex	length_ms	zero_crossings_sum	mfcc_mean	mfcc_st
0	fearful	0.0	0.0	0.0	1.0	0.0	3737.0	16995.0	-33.485947	134.65486
1	angry	0.0	0.0	0.0	0.0	0.0	3904.0	13906.0	-29.502108	130.48563
2	happy	0.0	1.0	0.0	1.0	0.0	4671.0	18723.0	-30.532463	126.57711
3	surprised	0.0	0.0	1.0	0.0	0.0	3637.0	11617.0	-36.059555	159.72516
4	happy	1.0	1.0	0.0	1.0	0.0	4404.0	15137.0	-31.405996	122.12582

In [37]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2452 entries, 0 to 2451
Data columns (total 28 columns):
 #   Column               Non-Null Count  Dtype  
---  ------               --------------  -----  
 0   emotion              2452 non-null   object 
 1   vocal_channel        2452 non-null   float64
 2   emotional_intensity  2452 non-null   float64
 3   statement            2452 non-null   float64
 4   repetition           2452 non-null   float64
 5   sex                  2452 non-null   float64
 6   length_ms            2452 non-null   float64
 7   zero_crossings_sum   2452 non-null   float64
 8   mfcc_mean            2452 non-null   float64
 9   mfcc_std             2452 non-null   float64
 10  mfcc_min             2452 non-null   float64
 11  mfcc_max             2452 non-null   float64
 12  sc_mean              2452 non-null   float64
 13  sc_std               2452 non-null   float64
 14  sc_min               2452 non-null   float64
 15  sc_max               2452 non-null   float64
 16  sc_kur               2452 non-null   float64
 17  sc_skew              2452 non-null   float64
 18  stft_mean            2452 non-null   float64
 19  stft_std             2452 non-null   float64
 20  stft_min             2452 non-null   float64
 21  stft_kur             2452 non-null   float64
 22  stft_skew            2452 non-null   float64
 23  std                  2452 non-null   float64
 24  min                  2452 non-null   float64
 25  max                  2452 non-null   float64
 26  kur                  2452 non-null   float64
 27  skew                 2452 non-null   float64
dtypes: float64(27), object(1)
memory usage: 536.5+ KB
```

C) Classification

In questa sezione, esploreremo, analizzeremo e valuteremo diversi metodi di classificazione al fine di prevedere due variabili categoriche fondamentali nel nostro dataset: Emotion e Sex.

Sex

Decision Tree on Sex

Iniziamo con il modello chiamato **Decision Tree** che costruisce una struttura ad albero che simula una serie di split logici basati sulle

variabili usate per l'addestramento per arrivare a una conclusione sulla variabile target. Per utilizzare il **Decision Tree**, dobbiamo far sì che gli attributi categorici nel nostro dataframe siano trasformati in variabili binarie (0 e 1) per consentire al modello di apprendere da tali attributi. Questo passaggio, chiamato binarizzazione, è già stato fatto per tutti gli attributi categorici del dataset con solo due valori distinti nella sezione di clustering.

Tuttavia, per quanto riguarda l'attributo Emotion, poiché esso presenta più di due valori distinti, adotteremo la tecnica di **one-hot encoding**. Questa tecnica trasforma ciascun valore univoco dell'attributo Emotion in una variabile binaria separata creando una colonna per ogni emozione possibile, come "happy", "sad", "angry", ecc., con valori 1 o 0 a seconda della presenza o assenza di quella specifica emozione in ciascuna osservazione. Questa trasformazione preserva l'integrità delle informazioni categoriche senza introdurre un ordine fittizio tra le categorie.

In [49]:
```
df_noobj = pd.get_dummies(df, columns=['emotion'])
```

Abbiamo quindi creato un dataframe chiamato **df_noobj** privo di oggetti

In [364…
```
df_noobj.dtypes
```

Out[364…
```
vocal_channel          float64
emotional_intensity    float64
statement              float64
repetition             float64
sex                    float64
length_ms              float64
zero_crossings_sum     float64
mfcc_mean              float64
mfcc_std               float64
mfcc_min               float64
mfcc_max               float64
sc_mean                float64
sc_std                 float64
sc_min                 float64
sc_max                 float64
sc_kur                 float64
sc_skew                float64
stft_mean              float64
stft_std               float64
stft_min               float64
stft_kur               float64
stft_skew              float64
std                    float64
min                    float64
max                    float64
kur                    float64
skew                   float64
emotion_angry             bool
emotion_calm              bool
emotion_disgust           bool
emotion_fearful           bool
emotion_happy             bool
emotion_neutral           bool
emotion_sad               bool
emotion_surprised         bool
dtype: object
```

In [366…
```
df_noobj
```

Out[366…

	vocal_channel	emotional_intensity	statement	repetition	sex	length_ms	zero_crossings_sum	mfcc_mean	mfcc_std	mf
0	0.0	0.0	0.0	1.0	0.0	3737.0	16995.0	-33.485947	134.654860	-755
1	0.0	0.0	0.0	0.0	0.0	3904.0	13906.0	-29.502108	130.485630	-713
2	0.0	1.0	0.0	1.0	0.0	4671.0	18723.0	-30.532463	126.577110	-726
3	0.0	0.0	1.0	0.0	0.0	3637.0	11617.0	-36.059555	159.725160	-842
4	1.0	1.0	0.0	1.0	0.0	4404.0	15137.0	-31.405996	122.125824	-700
...	...	...	...	...	...	...	...	...	...	
2447	0.0	1.0	1.0	0.0	1.0	4605.0	9871.0	-30.225578	158.845500	-855
2448	0.0	0.0	0.0	0.0	1.0	4171.0	8963.0	-31.160332	157.499700	-825
2449	1.0	1.0	0.0	1.0	1.0	5239.0	9765.0	-26.135280	138.133210	-768
2450	0.0	0.0	1.0	0.0	1.0	3737.0	9716.0	-28.242815	159.943400	-868
2451	1.0	0.0	0.0	1.0	1.0	3837.0	9427.0	-29.019236	149.188950	-799

2452 rows × 35 columns

Scegliamo come variabile target l'attributo **sex** e prendiamo tutte le altre colonne come quelle che useremo per allenare il modello predittivo del Decision Tree

In [369…
```
target = 'sex'
columns = [c for c in df_noobj.columns if c not in target]
```

In [371…
```
columns
```

Out[371…
```
['vocal_channel',
 'emotional_intensity',
 'statement',
 'repetition',
 'length_ms',
 'zero_crossings_sum',
 'mfcc_mean',
 'mfcc_std',
 'mfcc_min',
 'mfcc_max',
 'sc_mean',
 'sc_std',
 'sc_min',
 'sc_max',
 'sc_kur',
 'sc_skew',
 'stft_mean',
 'stft_std',
 'stft_min',
 'stft_kur',
 'stft_skew',
 'std',
 'min',
 'max',
 'kur',
 'skew',
 'emotion_angry',
 'emotion_calm',
 'emotion_disgust',
 'emotion_fearful',
 'emotion_happy',
 'emotion_neutral',
 'emotion_sad',
 'emotion_surprised']
```

Splittiamo il dataset in due parti, una riservata al **Training & Validation Set** composta dall'80% dei dati e il **Test Set** composto dal 20% dei dati. All'interno della porzione Training & Validation Set, un 20% viene riservato al Validation Set (16% del totale) e il rimanente al Traning Set (64%). Abbiamo scelto questa proporzione in quanto è una di quelle più comunemente usate nei modelli predittivi, ma successivamente andremo a controllare se esistono partizioni che migliorano il modello. La creazione dei due set è stratificata (con il parametro stratify settato a y) e riproducibile (con il parametro random_state=0). L'importanza di creare un Validation Set che sia indipendente permette di valutare e ottimizzare gli iperparametri del modello senza toccare il Test Set. Questo approccio può aiutare a prevenire l'overfitting sul Test Set.

In [374…
```
X = df_noobj[columns].values
y = df_noobj[target].values
```

In [376…
```
from sklearn.model_selection import train_test_split
```

In [378…
```
X_train_val, X_test, y_train_val, y_test = train_test_split(X, y, test_size=0.2, stratify=y, random_state=0)
```

In [380…
```
X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val, test_size=0.2, stratify=y_train_val
```

In [382…
```
from sklearn.tree import DecisionTreeClassifier
```

In [384…
```
from sklearn.metrics import accuracy_score, confusion_matrix
```

Dopo aver importato l'algoritmo DecisionTreeClassifier, lo inizializziamo usando il **gini index** per misurare la qualità degli split. In questo primo passaggio, in maniera arbitraria, decidiamo di non mettere un limite alla **profondità dell'albero**, di impostare come 2 **il numero minimo di campioni necessari per splittare un nodo**, e come 1 **il numero minimo di campioni che devono essere inclusi in una foglia**. L'albero viene prima addestrato sul Traning set che abbiamo scelto e successivamente testato sul Validation set. L'accuracy ottenuta è 0.90 che indica che il modello ha predetto correttamente la classe di appartenenza nel 90% dei casi nel set di dati di validazione.

In [387…
```
clf = DecisionTreeClassifier(
    criterion='gini',
    max_depth=None,
    min_samples_split=2,
    min_samples_leaf=1,
    ccp_alpha=0.0,
```

```
    random_state=0
)
clf.fit(X_train, y_train)

y_pred = clf.predict(X_val)
accuracy_score(y_val, y_pred)
```

Out[387… 0.9033078880407125

Visualizziamo la **confusion matrix**, dove:

- Nella prima cella (in alto a sinistra) abbiamo i True Positives (TP)
- Nella seconda cella (in alto a destra) abbiamo i False Negatives (FN)
- Nella terza cella (in basso a sinistra) abbiamo i False Positives (FP)
- Nella quarta cella (in basso a destra) abbiamo i True Negatives (TN)

Il numero di TP e TN rispetto al FP e FN rispecchia l'alta accuratezza del modello

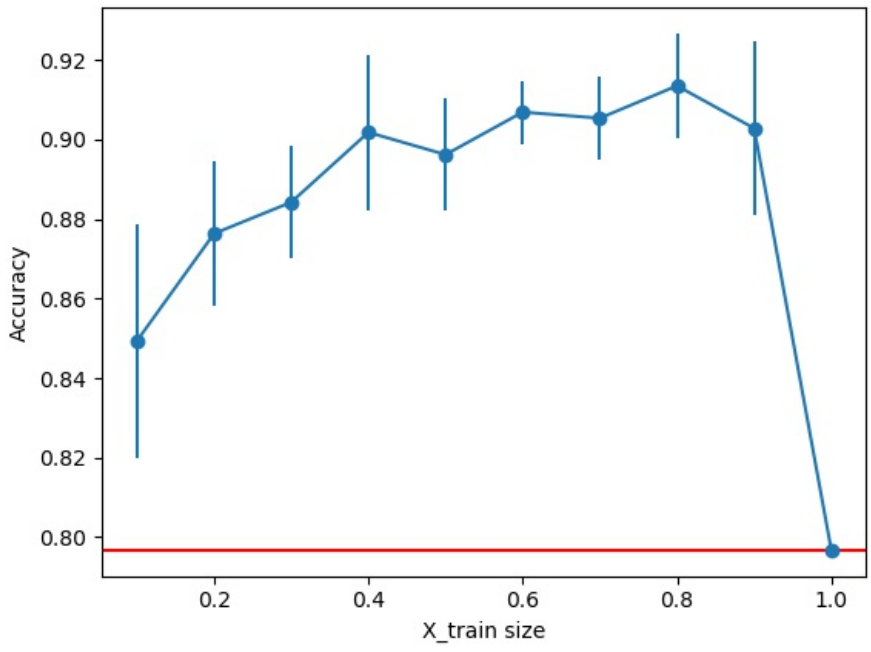
In [76]:
```
confusion_matrix(y_val, y_pred)
```

Out[76]:
```
array([[170,  23],
       [ 15, 185]])
```

Con le seguenti due celle di codice valutiamo come varia l'accuratezza al variare della dimensione del Training Set. Dal grafico sembrerebbe che assegnare l'80% del dataset al Training Set dia i risultati migliori per l'accuratezza.

In [401…
```
accuracy_mean_list = list()
accuracy_std_list = list()
for p in np.arange(0.1, 1.0, 0.1):
    accuracy_list_p = list()
    for i in range(0, 10):
        index = np.random.choice(np.arange(0, len(X_train)), int(len(X_train) * p), replace=False)
        clf = DecisionTreeClassifier(
            criterion='gini',
            max_depth=None,
            min_samples_split=2,
            min_samples_leaf=1,
            ccp_alpha=0.0,
            random_state=0)
        clf.fit(X_train[index], y_train[index])

        y_pred = clf.predict(X_val)
        accuracy_list_p.append(accuracy_score(y_val, y_pred))
    accuracy_mean_list.append(np.mean(accuracy_list_p))
    accuracy_std_list.append(np.std(accuracy_list_p))
```

In [402…
```
accuracy_mean_list.append(0.7967914438502673)
accuracy_std_list.append(0.0)

plt.errorbar(x=np.arange(0.1, 1.1, 0.1), y=accuracy_mean_list,
             yerr=accuracy_std_list, marker='o')
plt.axhline(y=0.7967914438502673, color='r')
plt.ylabel('Accuracy')
plt.xlabel('X_train size')
plt.show()
```



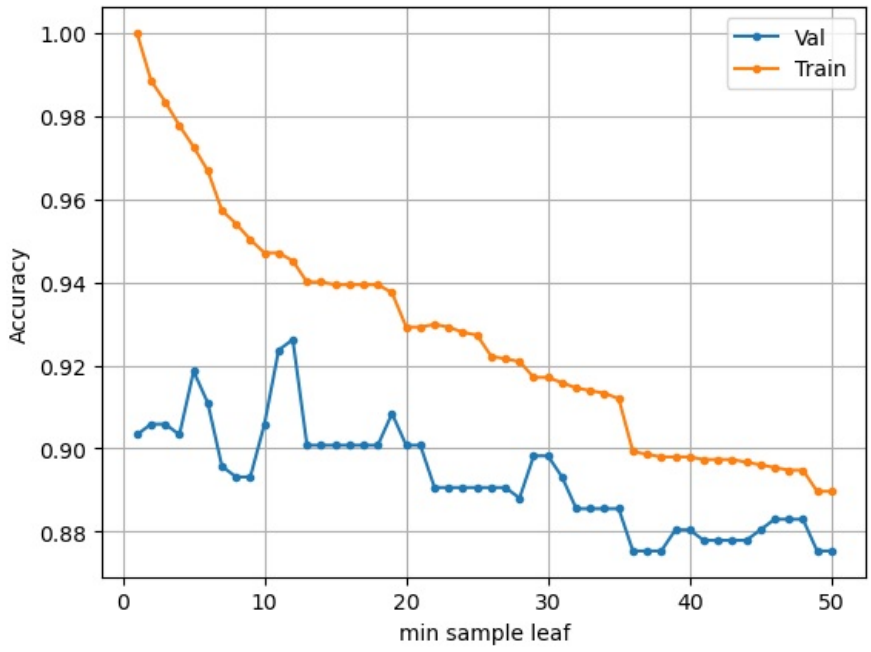
Nelle seguenti celle andiamo a esplorare come la selezione degli **iperparametri** per il Decision Tree vari l'accuratezza. Ne andremo ad analizzare 3:

1. *il numero minimo di campioni che deve avere ogni foglia dell'albero decisionale*
2. *il numero minimo di campioni necessari perché un nodo sia splittabile*
3. *la massima profondità dell'albero decisionale*.

Con le seguenti due celle di codice valutiamo come varia l'accuratezza con **il numero minimo di campioni che deve avere ogni foglia dell'albero decisionale**. Dal grafico sembrerebbe che i valori di 11 e 12 siano quelli che danno un'accuratezza migliore. Generalmente quando possibile, è preferibile scegliere numeri più alti per evitare overfitting quindi per le successive analisi imposteremo il parametro a 12.

In [405…
```
min_sample_leaf_list = np.arange(1, 50+1, 1)
accuracy_val_list = list()
accuracy_train_list = list()
for min_sample_leaf in min_sample_leaf_list:
    clf = DecisionTreeClassifier(
        criterion='gini',
        max_depth=None,
        min_samples_split=2,
        min_samples_leaf=min_sample_leaf,
        ccp_alpha=0.0,
        random_state=0
    )
    clf.fit(X_train, y_train)

    y_pred = clf.predict(X_val)
    accuracy_val_list.append(accuracy_score(y_val, y_pred))

    y_pred_train = clf.predict(X_train)
    accuracy_train_list.append(accuracy_score(y_train, y_pred_train))
```

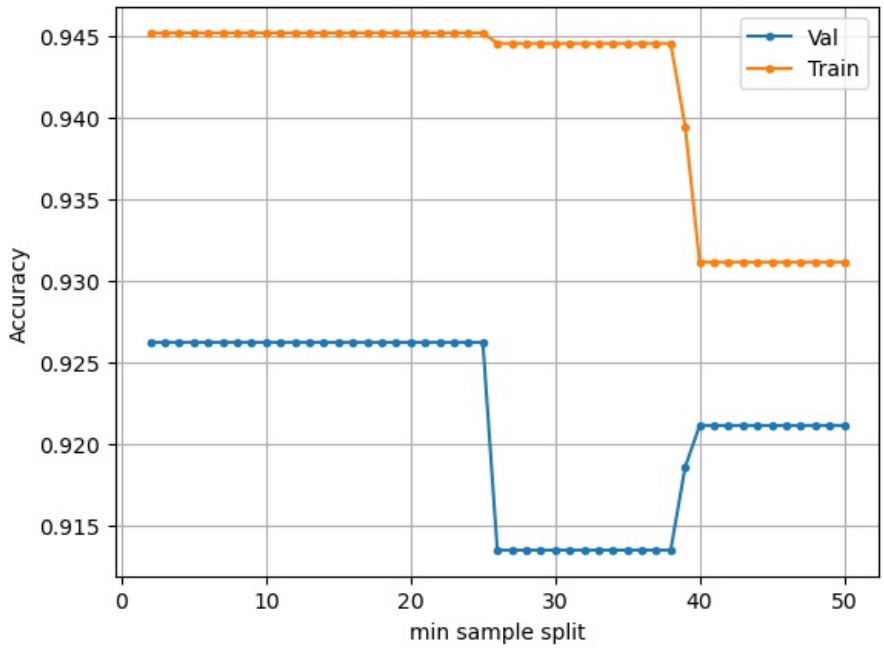
In [406…
```
plt.plot(min_sample_leaf_list, accuracy_val_list, label='Val', marker='.')
plt.plot(min_sample_leaf_list, accuracy_train_list, label='Train', marker='.')
plt.ylabel('Accuracy')
plt.xlabel('min sample leaf')
plt.grid('white')
plt.legend()
plt.show()
```



Con le seguenti due celle di codice andiamo a invece valutare come varia l'accuratezza con **il numero minimo di campioni necessari perché un nodo sia splittabile**. Dal grafico sembrerebbe che l'accuratezza sia ottimizzata con valori <=25. Per minimizzare il rischio di overfitting e avere un modello predittivo più robusto è generalmente consigliabile avere valori più alti e quindi scegliamo 25 per l'analisi successiva.

In [410…
```
min_sample_split_list = np.arange(2, 50+1, 1)
min_sample_split_list

accuracy_val_list = list()
accuracy_train_list = list()
for min_sample_split in min_sample_split_list:
    clf = DecisionTreeClassifier(
```

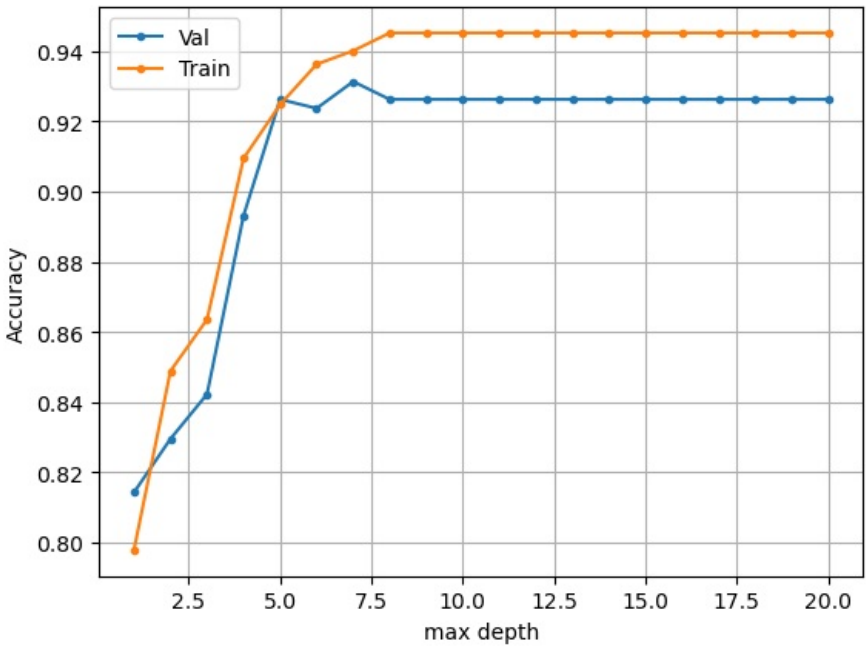
```
        criterion='gini',
        max_depth=None,
        min_samples_split=min_sample_split,
        min_samples_leaf=12,
        ccp_alpha=0.0,
        random_state=0
    )
    clf.fit(X_train, y_train)

    y_pred = clf.predict(X_val)
    accuracy_val_list.append(accuracy_score(y_val, y_pred))

    y_pred_train = clf.predict(X_train)
    accuracy_train_list.append(accuracy_score(y_train, y_pred_train))

plt.plot(min_sample_split_list, accuracy_val_list, label='Val', marker='.')
plt.plot(min_sample_split_list, accuracy_train_list, label='Train', marker='.')
plt.ylabel('Accuracy')
plt.xlabel('min sample split')
plt.grid('white')
plt.legend()
plt.show()
```



Per concludere, esamineremo come varia l'accuratezza al variare della **massima profondità** dell'albero decisionale. Dall'analisi grafica, sembra che l'accuratezza raggiunga il massimo con valore di profondità 7.

In [412…

```
max_depth_list = np.arange(1, 20+1, 1).tolist() + [None]

accuracy_val_list = list()
accuracy_train_list = list()
for max_depth in max_depth_list:
    clf = DecisionTreeClassifier(
        criterion='gini',
        max_depth=max_depth,
        min_samples_split=25,
        min_samples_leaf=12,
        ccp_alpha=0.0,
        random_state=0
    )
    clf.fit(X_train, y_train)

    y_pred = clf.predict(X_val)
    accuracy_val_list.append(accuracy_score(y_val, y_pred))

    y_pred_train = clf.predict(X_train)
    accuracy_train_list.append(accuracy_score(y_train, y_pred_train))

plt.plot(max_depth_list, accuracy_val_list, label='Val', marker='.')
plt.plot(max_depth_list, accuracy_train_list, label='Train', marker='.')
plt.ylabel('Accuracy')
plt.xlabel('max depth')
plt.grid('white')
plt.legend()
plt.show()
```



Proviamo adesso ad usare **RandomizedSearch** che è una tecnica automatica di ottimizzazione della libreria **scikit-learn** per trovare i migliori parametri da impostare per il DecisionTree.

In [95]:
```
from sklearn.model_selection import RandomizedSearchCV
```

In [99]:
```
param_dict = {
    'max_depth': np.arange(1, 20+1, 1).tolist(),
    'min_samples_split': np.arange(2, 50+1, 1),
    'min_samples_leaf': np.arange(1, 50+1, 1),
    'ccp_alpha': np.arange(0.0,1, 0.1)
}
```

In [101…
```
random = RandomizedSearchCV(clf, param_dict, cv=5, scoring='accuracy', refit=True, n_iter=500, random_state=0)
random.fit(X_train_val, y_train_val)
random.best_params_
```

Out[101…
```
{'min_samples_split': 15,
 'min_samples_leaf': 7,
 'max_depth': 12,
 'ccp_alpha': 0.0}
```

In [102…
```
random.best_estimator_
```

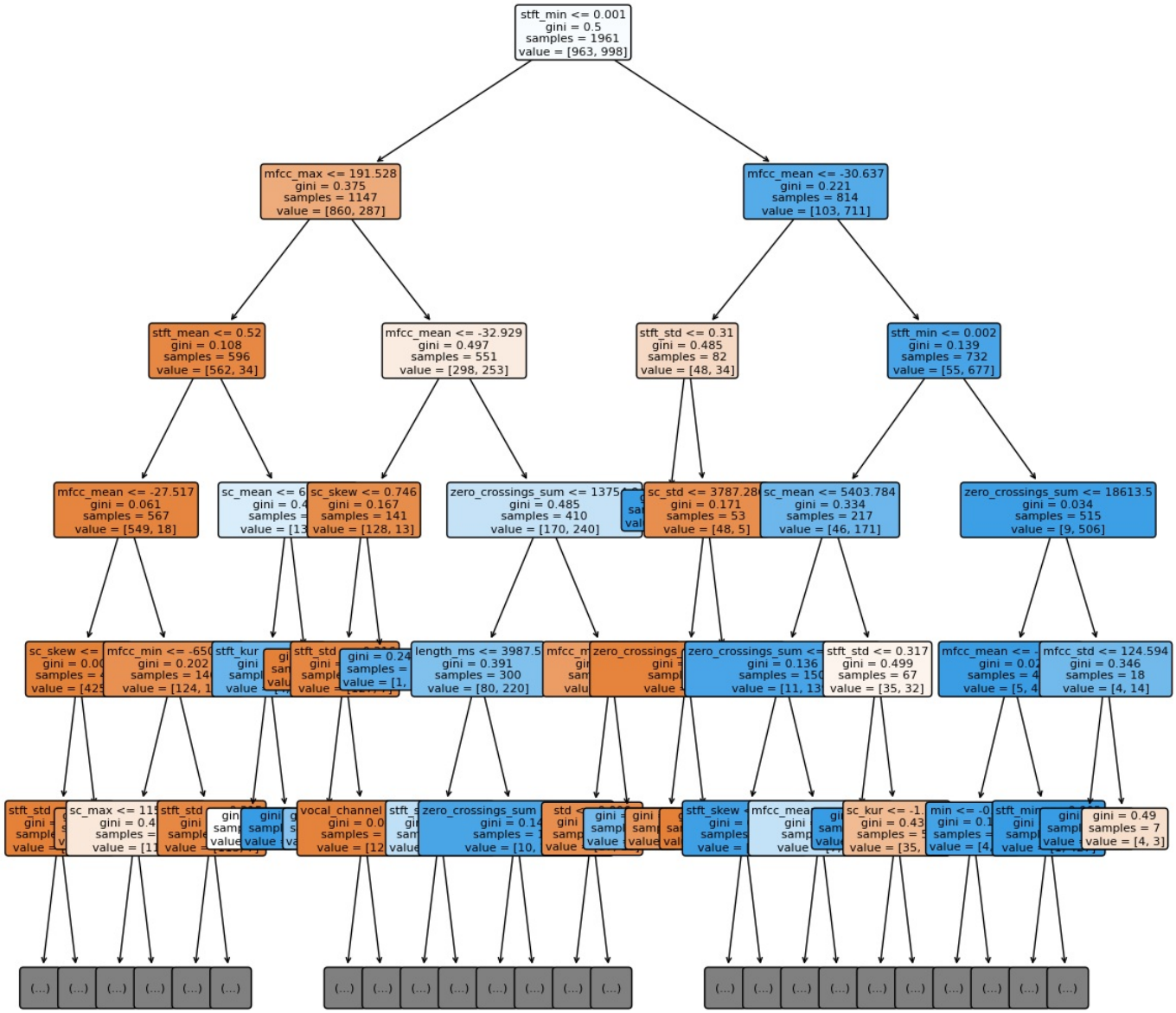
Out[102…
```
DecisionTreeClassifier(max_depth=12, min_samples_leaf=7, min_samples_split=15,
                       random_state=0)
```

Si può vedere che i valori trovati da **RandomizedSearch** non sono gli stessi di quelli che abbiamo precendente individuato cambiando un parametro alla volta. Abbiamo deciso di usare i parametri di RandomizedSearch visto che l'algoritmo è più efficiente nel valutare e ottimizzare le diverse combinazione dei tre parametri.

In [104…
```
clf = random.best_estimator_
```

In [105…
```
from sklearn.tree import plot_tree, export_text
```

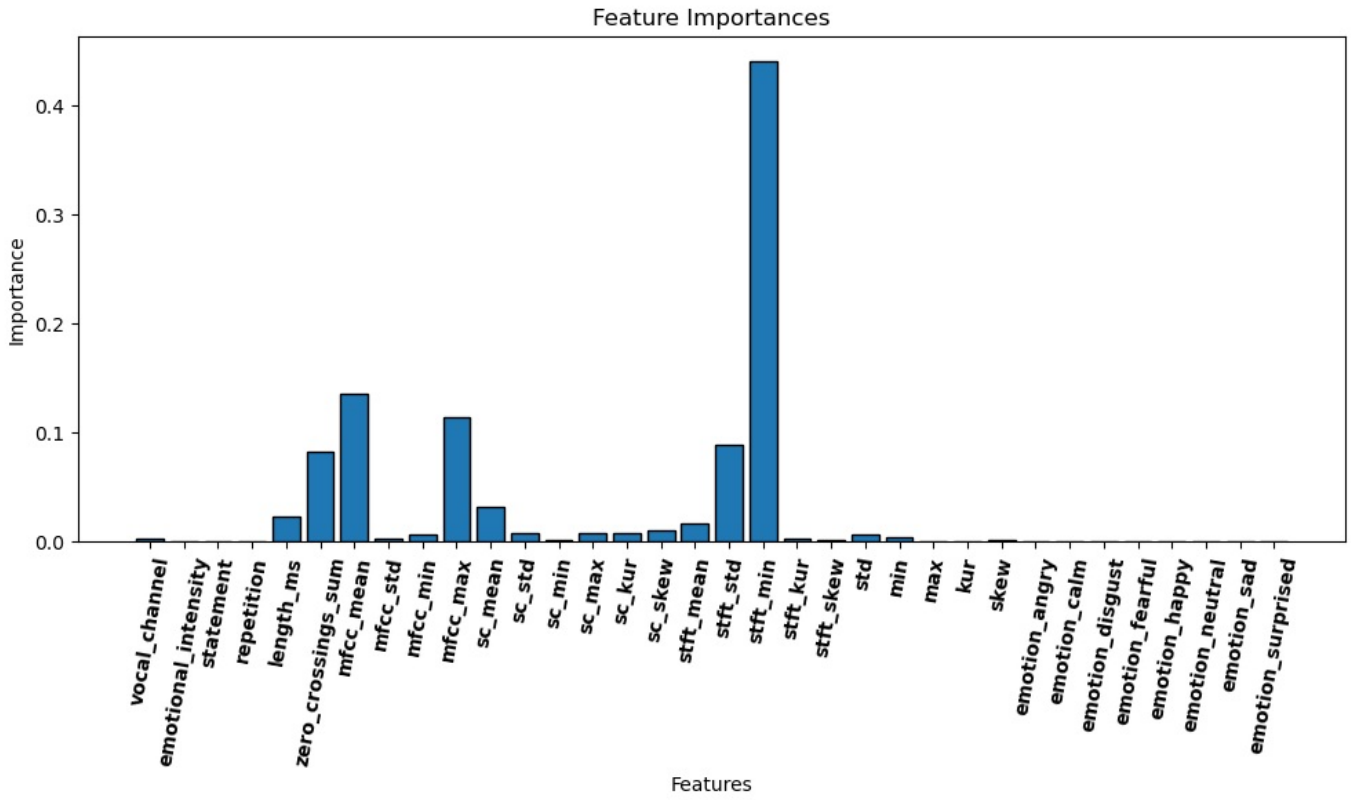
In [106…
```
plt.figure(figsize=(14, 14))
plot_tree(clf,
          feature_names=columns,
          filled=True,
          rounded=True,
          fontsize=8,
          max_depth=5
          )
plt.show()
```



Con la seguente barchart visualizziamo quali sono le features del dataset più importanti per prevedere il valore "Sex" con il modello del DecisionTree. La feature più importante sembra essere "stft_min" seguita da "mfcc_max" e "mfcc_mean"

In [108…

```
plt.figure(figsize=(10, 6))
plt.bar(columns, clf.feature_importances_, edgecolor='k')
plt.xticks(rotation=80, fontweight='bold')
plt.title('Feature Importances')
plt.xlabel('Features')
plt.ylabel('Importance')
plt.tight_layout()
plt.show()
```



Nel Classification Report stampato sotto, possiamo vedere che l'accuratezza del Decision Tree sul Test set è elevata (0.90), indicando una buona capacità di classificazione generale. Tuttavia, l'analisi di Precision, Recall, F1-Score e Confusion Matrix rivela un piccolo sbilanciamento nelle performance tra le classi. In particolare, il modello mostra una maggiore precisione nel prevedere la classe 0 (F), con meno falsi positivi, ma soffre di un numero più alto di falsi negativi per la stessa classe. Al contrario, per la classe 1 (M), si osserva un incremento dei falsi positivi rispetto ai falsi negativi.

In [110…
```
y_pred = clf.predict(X_test)

accuracy_score(y_test, y_pred)
```

Out[110… 0.8961303462321792

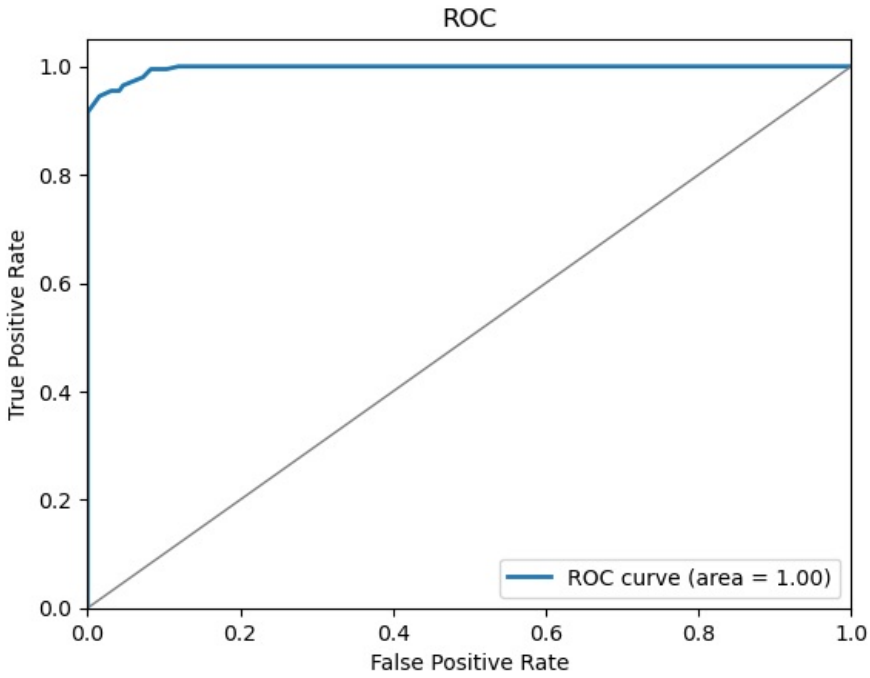
In [111…
```
from sklearn.metrics import precision_score, recall_score, f1_score,classification_report
precision = precision_score(y_test, y_pred,average='weighted')
recall = recall_score(y_test, y_pred,average='weighted')
f1 = f1_score(y_test, y_pred,average='weighted')
conf_matrix = confusion_matrix(y_test, y_pred)
y_pred = clf.predict(X_test)
print(classification_report(y_pred, y_test))

y_pred_trainval = clf.predict(X_train_val)

print(classification_report(y_pred_trainval, y_train_val))
```

```
              precision    recall  f1-score   support

         0.0       0.91      0.88      0.90       248
         1.0       0.88      0.91      0.90       243

    accuracy                           0.90       491
   macro avg       0.90      0.90      0.90       491
weighted avg       0.90      0.90      0.90       491

              precision    recall  f1-score   support

         0.0       0.95      0.96      0.96       958
         1.0       0.96      0.96      0.96      1003

    accuracy                           0.96      1961
   macro avg       0.96      0.96      0.96      1961
weighted avg       0.96      0.96      0.96      1961
```

Plottiamo adesso la **ROC** che è uno strumento grafico usato per valutare le prestazioni di un modello predittivo su una variabile target binaria. La curva traccia il tasso di TP rispetto a FP. Calcoliamo l'area sotto alla curva che risulta essere 1 che indica un modello altamente performante. Questo valore è consistente con l'alta accuratezza calcolata in precedenza e implica che il modello predittivo è efficiente sulla variabile target Sex.

In [113…
```
from sklearn.metrics import roc_curve, auc
```

```
import matplotlib.pyplot as plt

y_pred = clf.predict_proba(X_val)[:, 1]

fpr, tpr, thresholds = roc_curve(y_val, y_pred)
roc_auc = auc(fpr, tpr)

plt.figure()
plt.plot(fpr, tpr, lw=2, label='ROC curve (area = %0.2f)' % roc_auc)
plt.plot([0, 1], [0, 1], color='gray', lw=1)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC')
plt.legend(loc="lower right")
plt.show()
```



KNN on Sex

Sebbene il Decision Tree abbia dato ottimi risultati sulla variabile target Sex, esploriamo ora un approccio differente adottando il **K-Nearest Neighbors (KNN)** come classificatore. Questo modello si basa sull'idea che le osservazioni vicine nello spazio dimensionale del dataset tendono ad appartenere alla stessa classe. In pratica, la classificazione di un nuovo campione viene determinata dalla maggioranza delle classi tra i suoi K vicini più prossimi.

Iniziamo con un numero di K=5 (il numero di *vicini* in base ai quali è fatta la classificazione). Il dataset è diviso in due parti: una riservata al **Training & Validation Set** composta dall'80% dei dati e il **Test Set** composto dal 20% dei dati. All'interno della porzione Training & Validation Set, un 20% viene riservato al Validation Set (16% del totale) e il rimanente al Traning Set (64%). Abbiamo scelto questa proporzione in quanto è una di quelle più comunemente usate nei modelli predittivi, ma successivamente andremo a controllare se esistono partizioni che migliorano il modello. La creazione dei due set è stratificata (con il parametro stratify settato a y) e riproducibile (con il parametro random_state=0)

Adottando il KNN, vogliamo valutare come un approccio basato sulla prossimità possa confrontarsi con la metodologia di decisione gerarchica utilizzata dai Decision Tree.

In [125…

```
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report

target = 'sex'
columns = [c for c in df_noobj.columns if c not in target]

X = df_noobj[columns].values
Y = df_noobj[target].values


scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

X_train_val, X_test, y_train_val, y_test = train_test_split(X_scaled, Y, test_size=0.2, stratify=Y, random_stat
X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val, test_size=0.2, stratify=y_train_val
```

```
print("Training set", X_train.shape, y_train.shape)
print("Validation set", X_val.shape, y_val.shape)
print("Test set", X_test.shape, y_test.shape)
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)
y_pred = knn.predict(X_val)
y_pred_train = knn.predict(X_train)
y_pred_trainval=knn.predict(X_train_val)

print(accuracy_score(y_val, y_pred), accuracy_score(y_train, y_pred_train))
print(classification_report(y_val, y_pred))
print(classification_report(y_pred_trainval, y_train_val))
```

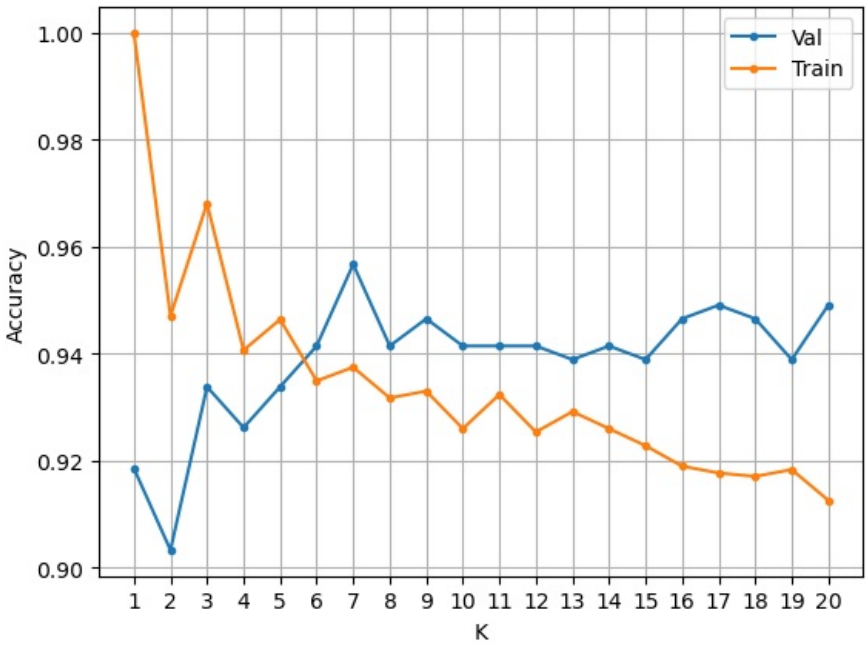
```
Training set (1568, 34) (1568,)
Validation set (393, 34) (393,)
Test set (491, 34) (491,)
0.9338422391857506 0.9464285714285714
              precision    recall  f1-score   support

         0.0       0.95      0.92      0.93       193
         1.0       0.92      0.95      0.94       200

    accuracy                           0.93       393
   macro avg       0.93      0.93      0.93       393
weighted avg       0.93      0.93      0.93       393

              precision    recall  f1-score   support

         0.0       0.95      0.94      0.94       977
         1.0       0.94      0.95      0.94       984

    accuracy                           0.94      1961
   macro avg       0.94      0.94      0.94      1961
weighted avg       0.94      0.94      0.94      1961
```

L'accuratezza risulta già molto alta, ma comunque cerchiamo di ottimizzarla variando il parametro **K**.

In [128…
```
acc_val_list = list()
acc_train_list = list()
for k in np.arange(1, 20+1, 1):
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train, y_train)
    y_pred = knn.predict(X_val)
    y_pred_train = knn.predict(X_train)
    acc_val_list.append(accuracy_score(y_val, y_pred))
    acc_train_list.append(accuracy_score(y_train, y_pred_train))
```

Da questo plot sembrerebbe che il miglior numero di K sia 7 però questo potrebbe essere influenzato dalla scelta del training set e del test set che è randomica

In [131…
```
plt.plot(np.arange(1, 20+1, 1), acc_val_list, label='Val', marker='.')
plt.plot(np.arange(1, 20+1, 1), acc_train_list, label='Train', marker='.')
plt.xlabel('K')
plt.ylabel('Accuracy')
plt.xticks(np.arange(1, 20+1, 1))
plt.grid()
plt.legend()
plt.show()
```

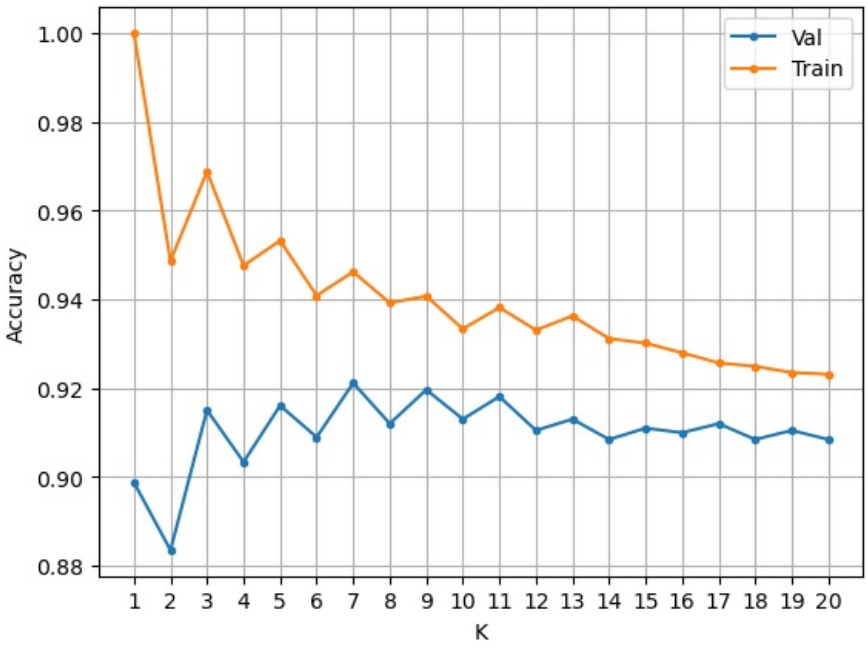


Per ovviare al problema della scelta randomica del training set possiamo usare una tecnica chiamata **Repeated Holdout**. Questo metodo prevede la ripetizione della divisione del dataset in Training Set e Test Set per diverse iterazioni, consentendo di valutare la stabilità e l'affidabilità del modello su più suddivisioni dei dati. In questo caso, abbiamo eseguito la valutazione su 5 diverse combinazioni di Training Set e Test Set. Dall'analisi dei risultati, attraverso la visualizzazione, abbiamo identificato che K=7 offre le migliori prestazioni complessive.

In [134…
```
nbr_holdout = 5
acc_val_list_all = list()
acc_train_list_all = list()
for i in range(nbr_holdout):

    X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val,
                                                      test_size=0.2,
                                                      stratify=y_train_val,
                                                      random_state=i)

    acc_val_list = list()
    acc_train_list = list()
    for k in np.arange(1, 20+1, 1):
        knn = KNeighborsClassifier(n_neighbors=k)
        knn.fit(X_train, y_train)
        y_pred = knn.predict(X_val)
        y_pred_train = knn.predict(X_train)
        acc_val_list.append(accuracy_score(y_val, y_pred))
        acc_train_list.append(accuracy_score(y_train, y_pred_train))

    acc_val_list_all.append(acc_val_list)
    acc_train_list_all.append(acc_train_list)
```

In [135…
```
acc_val_list_all = np.array(acc_val_list_all)
acc_train_list_all = np.array(acc_train_list_all)
```

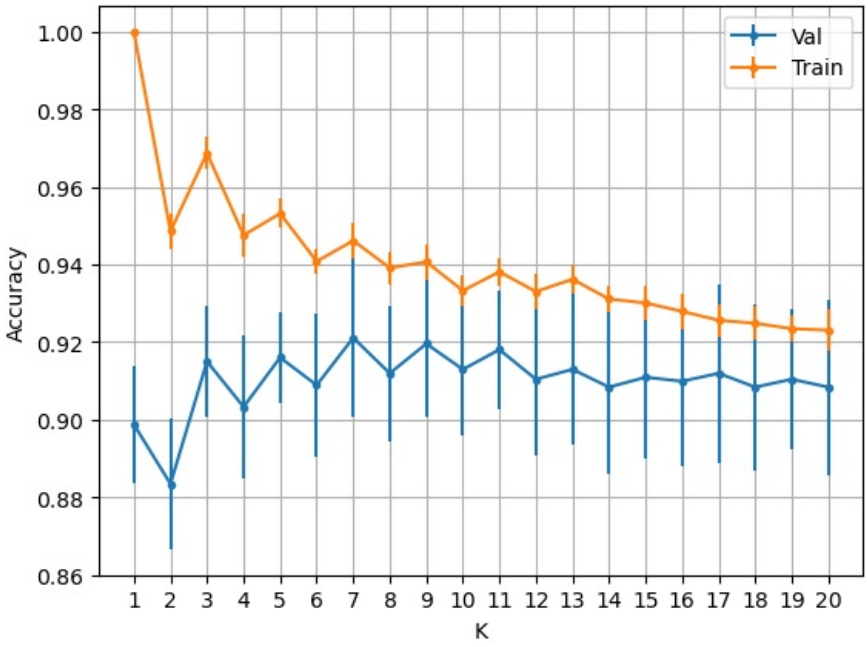
In [136…
```
plt.plot(np.arange(1, 20+1, 1), np.mean(acc_val_list_all, axis=0), label='Val', marker='.')
plt.plot(np.arange(1, 20+1, 1), np.mean(acc_train_list_all, axis=0), label='Train', marker='.')
plt.xlabel('K')
plt.ylabel('Accuracy')
plt.xticks(np.arange(1, 20+1, 1))
plt.grid()
plt.legend()
plt.show()
```



Facciamo lo stesso plot includendo la deviazione stardard per ogni valore di K. Questo ci permette di valutare non solo la performance media ma anche la variabilità delle prestazioni. Dall'analisi, K=7 si conferma essere il valore che offre le migliori prestazioni complessive, combinando una buona accuratezza con una variabilità relativamente bassa.

In [141…

```
plt.errorbar(x=np.arange(1, 20+1, 1),
             y=np.mean(acc_val_list_all, axis=0),
             yerr=np.std(acc_val_list_all, axis=0),
             label='Val', marker='.')

plt.errorbar(x=np.arange(1, 20+1, 1),
             y=np.mean(acc_train_list_all, axis=0),
             yerr=np.std(acc_train_list_all, axis=0),
             label='Train', marker='.')

plt.xlabel('K')
plt.ylabel('Accuracy')
plt.xticks(np.arange(1, 20+1, 1))
plt.grid()
plt.legend()
plt.show()
```



Cross Validation

A questo punto è importante però sottolineare che il problema della scelta casuale dei Test Set e Training Set potrebbe persistere, dato che, in teoria, la distribuzione della variabile target potrebbe essere sbilanciata, influenzando così i risultati dei plot precedentemente analizzati. Per mitigare questo potenziale bias, possiamo ricorrere alla tecnica della **Cross Validation**. Questo metodo ci permette di valutare più accuratamente il nostro modello attraverso diverse iterazioni, utilizzando ogni volta diverse suddivisioni del dataset in Training Set e Test Set, assicurando così una più equa rappresentazione della variabile target in ogni fase di valutazione.

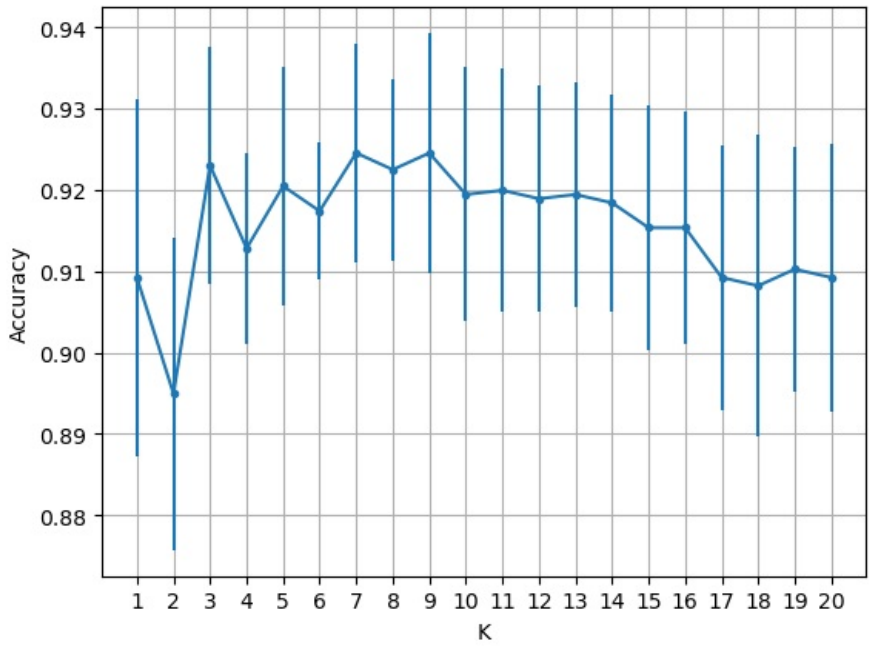
Utilizzando i parametri definiti sotto, la funzione cross_val_score organizza X_train_val in 10 sottoinsiemi. Per ogni iterazione di validazione incrociata, 9 di questi sottoinsiemi sono utilizzati per l'addestramento del modello, mentre quello rimanente serve come Test Set. In ogni iterazione, un diverso sottoinsieme è selezionato come test set, mentre i sottoinsiemi restanti costituiscono il training set. Questo processo garantisce che ogni sottoinsieme venga utilizzato almeno una volta come test set. Con k=7 per il nostro classificatore KNN, la funzione procede attraverso tutte le 10 iterazioni possibili, calcolando un valore di accuratezza per ciascuna.

In [146…
```
from sklearn.model_selection import cross_val_score
knn = KNeighborsClassifier(n_neighbors=7)
scores = cross_val_score(knn, X_train_val, y_train_val, cv=10)
```

Realizzando un ciclo che varia il valore di K da 1 a 20 e utilizzando la funzione di validazione incrociata per ogni valore, possiamo plottare un grafico che mostra come l'accuratezza varia in funzione di K. Dai risultati ottenuti, K=9 emerge come il valore che massimizza l'accuratezza. Tuttavia, anche K=7 si conferma una scelta valida, mostrando elevate prestazioni.

In [149…
```
acc_list_mean = list()
acc_list_std = list()
for k in np.arange(1, 20+1, 1):
    knn = KNeighborsClassifier(n_neighbors=k)
    scores = cross_val_score(knn, X_train_val, y_train_val, cv=10)
    acc_list_mean.append(np.mean(scores))
    acc_list_std.append(np.std(scores))
```

In [150…
```
plt.errorbar(x=np.arange(1, 20+1, 1),
             y=acc_list_mean,
             yerr=acc_list_std,
             marker='.')

plt.xlabel('K')
plt.ylabel('Accuracy')
plt.xticks(np.arange(1, 20+1, 1))
plt.grid()
plt.show()
```



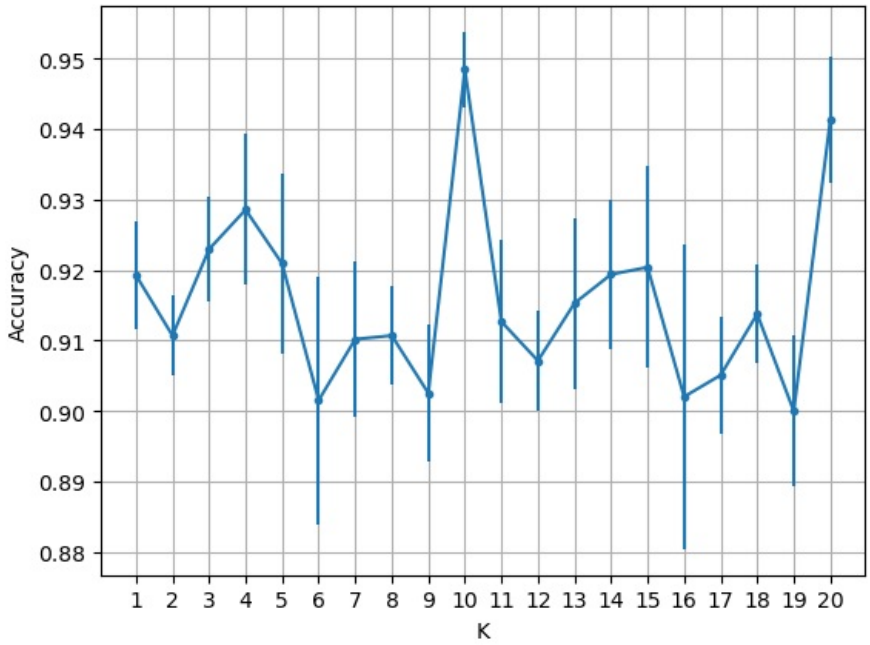
Per minimizzare l'impatto della selezione casuale dei set di dati, intrinseco nella funzione cross_val_score, possiamo ripetere il processo di cross-validation più volte. In questo caso, scegliamo di eseguire 5 ripetizioni facendo quindi 2 cicli *for*. Questo approccio aiuta a stabilizzare le stime di accuratezza. K=10 sembra offrire migliori prestazioni complessive, suggerendo che sia il valore ottimale per il nostro classificatore KNN in questo contesto.

In [154…
```
nbr_repetitions = 5
acc_list_all = list()
for i in range(nbr_repetitions):
    acc_list = list()
    for k in np.arange(1, 20+1, 1):
        knn = KNeighborsClassifier(n_neighbors=k)
        scores = cross_val_score(knn, X_train_val, y_train_val, cv=10,
                                 scoring='accuracy')
        acc_list.append(scores)

    acc_list_all.append(acc_list)
```

In [155…
```
acc_list_all = np.array(acc_list_all)
acc_list_all.shape
acc_list_all.reshape(50, 20)
np.mean(acc_list_all.reshape(50, 20), axis=0)
```

Out[155…
```
array([0.91928934, 0.91071429, 0.92295918, 0.92857143, 0.92091837,
       0.90153061, 0.91020408, 0.91071429, 0.90255102, 0.94846939,
       0.91269036, 0.90714286, 0.91530612, 0.91938776, 0.92040816,
       0.90204082, 0.90510204, 0.91377551, 0.9       , 0.94132653])
```

In [156…
```
plt.errorbar(x=np.arange(1, 20+1, 1),
             y=np.mean(acc_list_all.reshape(50, 20), axis=0),
             yerr=np.std(acc_list_all.reshape(50, 20), axis=0),
             marker='.')

plt.xlabel('K')
plt.ylabel('Accuracy')
plt.xticks(np.arange(1, 20+1, 1))
plt.grid()
plt.show()
```



Stampiamo i report di classificazione per il Test set e il Training set con K=10. Il metodo predittivo ha un'accuratezza molto alta per entrambi i sessi ma con uno sbilanciamento nella precision e nella recall. Questo sbilanciamento implica che il modello è preciso nel prevedere 0 (Sex=F) con un basso numero di FP (falsi positivi), ma con un numero più alto di FN (falsi negativi) suggerendo una tendenza a sottovalutare questa classe . Per 'M' (Sex=1), si verifica la situazione opposta: ci sono più FP rispetto a FN, indicando una propensione a sovrastimare questa classe.

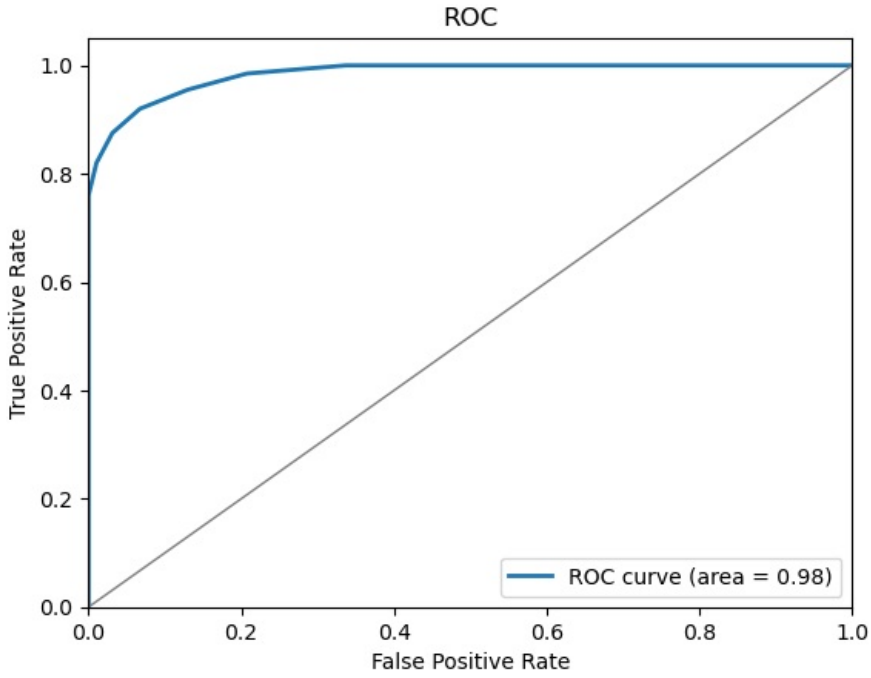
In [158…
```
knn = KNeighborsClassifier(n_neighbors=10)
knn.fit(X_train_val, y_train_val)

y_pred = knn.predict(X_test)
print(classification_report(y_pred, y_test))

y_pred_trainval = knn.predict(X_train_val)
print(classification_report(y_pred_trainval, y_train_val))
```

```
              precision    recall  f1-score   support

         0.0       0.94      0.89      0.91       256
         1.0       0.88      0.94      0.91       235

    accuracy                           0.91       491
   macro avg       0.91      0.91      0.91       491
weighted avg       0.91      0.91      0.91       491

              precision    recall  f1-score   support

         0.0       0.97      0.91      0.94      1028
         1.0       0.91      0.97      0.94       933

    accuracy                           0.94      1961
   macro avg       0.94      0.94      0.94      1961
weighted avg       0.94      0.94      0.94      1961
```

Plottiamo adesso la **ROC** che è uno strumento grafico usato per valutare le prestazioni di un modello predittivo su una variabile target

binaria. La curva traccia il tasso di TP rispetto a FP. Calcoliamo l'area sotto alla curva che risulta essere 0.98 che indica un modello altamente performante. Questo valore è consistente con l'alta accuratezza calcolata in precedenza e implica che il modello predittivo è efficiente sulla variabile target Sex.

In [166…
```
from sklearn.metrics import roc_curve, auc
import matplotlib.pyplot as plt

y_pred = knn.predict_proba(X_val)[:, 1]

fpr, tpr, thresholds = roc_curve(y_val, y_pred)
roc_auc = auc(fpr, tpr)

plt.figure()
plt.plot(fpr, tpr, lw=2, label='ROC curve (area = %0.2f)' % roc_auc)
plt.plot([0, 1], [0, 1], color='gray', lw=1)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC')
plt.legend(loc="lower right")
plt.show()
```

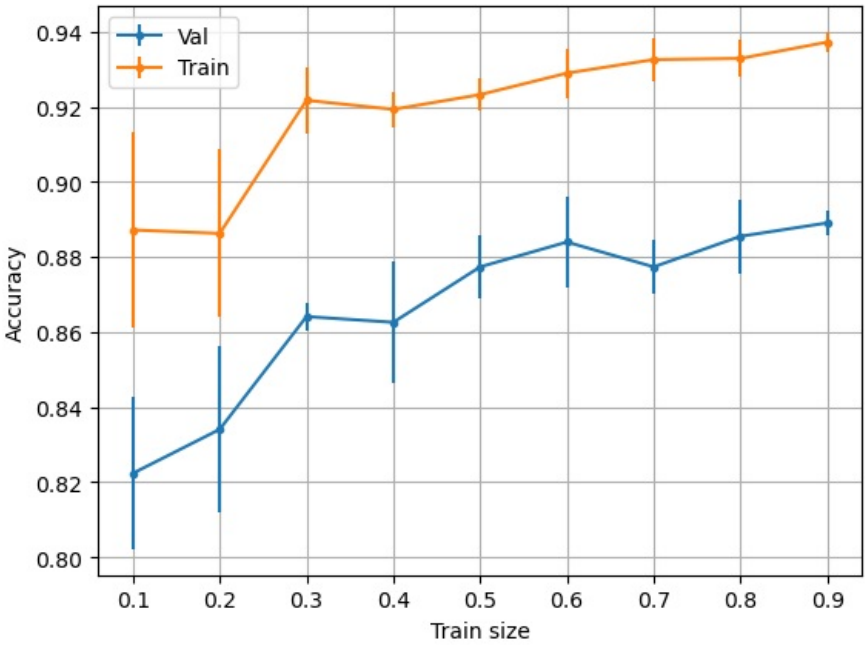


Per comprendere meglio come la dimensione del Training Set influisca sull'accuratezza del nostro modello, possiamo utilizzare quella che è nota come **Learning Curve**. Questa curva ci permette di visualizzare la relazione tra l'accuratezza del modello e la quantità di dati utilizzati per l'addestramento, il che è particolarmente rilevante quando si lavora con dataset di grandi dimensioni. Una riduzione mirata della dimensione del Training Set può, infatti, aumentare l'efficienza computazionale del modello senza sacrificare significativamente le prestazioni. Nel caso del nostro dataset, sembra che utilizzare l'80% dei dati come Training Set offra l'accuratezza migliore, anche se la variazione è minima tra il 60% e l'80%. Questo approccio potrebbe essere applicato anche focalizzandoci su altri indicatori di performance, come la precisione o la recall.

In [169…
```
nbr_repetitions = 5
acc_val_list_all = list()
acc_train_list_all = list()
for p in np.arange(0.1, 1.0, 0.1):
    acc_val_list = list()
    acc_train_list = list()
    for i in range(nbr_repetitions):
        index = np.random.choice(np.arange(len(X_train)), int(len(X_train) * p), replace=False)
        knn = KNeighborsClassifier(n_neighbors=10)
        knn.fit(X_train[index], y_train[index])
        y_pred = knn.predict(X_val)
        y_pred_train = knn.predict(X_train[index])
        acc_val_list.append(accuracy_score(y_val, y_pred))
        acc_train_list.append(accuracy_score(y_train[index], y_pred_train))

    acc_val_list_all.append(acc_val_list)
    acc_train_list_all.append(acc_train_list)
```

In [171…
```
acc_val_list_all = np.array(acc_val_list_all)
acc_train_list_all = np.array(acc_train_list_all)
```

In [173…
```
plt.errorbar(x=np.arange(0.1, 1.0, 0.1),
             y=np.mean(acc_val_list_all, axis=1),
             yerr=np.std(acc_val_list_all, axis=1),
             label='Val', marker='.')

plt.errorbar(x=np.arange(0.1, 1.0, 0.1),
             y=np.mean(acc_train_list_all, axis=1),
             yerr=np.std(acc_train_list_all, axis=1),
             label='Train', marker='.')

plt.xlabel('Train size')
plt.ylabel('Accuracy')
plt.xticks(np.arange(0.1, 1.0, 0.1))
plt.grid()
plt.legend()
plt.show()
```



Emotion

Decision Tree on Emotion

Per la classificazione della classe Emotion tramite Decision Tree, ripeteremo i passaggi già utilizzati per la classificazione della classe Sex. In questo scenario, tuttavia, per la preparazione dei dati, possiamo evitare l'one-hot encoding per la variabile target 'emotion', optando invece per convertirla direttamente in etichette numeriche da 0 a 7. Questa scelta si giustifica dal fatto che 'emotion' sarà la nostra variabile target in un contesto di classificazione multiclasse, e l'uso di etichette numeriche semplifica il processo senza influire negativamente sull'apprendimento del modello. Le etichette saranno assegnate come segue: **0=angry, 1=calm, 2=disgust, 3=fearful, 4=happy, 5=neutral, 6=sad, 7=surprised.**

In [414…
```
dfe=pd.read_csv('A) Data Understanding & Preparation Output.csv', sep=',', skipinitialspace=True)
dfe.head()
```

Out[414…

	emotion	vocal_channel	emotional_intensity	statement	repetition	sex	length_ms	zero_crossings_sum	mfcc_mean	mfcc_st
0	fearful	0.0	0.0	0.0	1.0	0.0	3737.0	16995.0	-33.485947	134.65486
1	angry	0.0	0.0	0.0	0.0	0.0	3904.0	13906.0	-29.502108	130.48563
2	happy	0.0	1.0	0.0	1.0	0.0	4671.0	18723.0	-30.532463	126.57711
3	surprised	0.0	0.0	1.0	0.0	0.0	3637.0	11617.0	-36.059555	159.72516
4	happy	1.0	1.0	0.0	1.0	0.0	4404.0	15137.0	-31.405996	122.12582

In [417…
```
emotion = sorted(dfe['emotion'].unique())
emotion_mapping = dict(zip(emotion, range(0, len(emotion) + 1)))
dfe['Emotion_val'] = dfe['emotion'].map(emotion_mapping).astype(int)
```

In [419…
```
dfe.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2452 entries, 0 to 2451
Data columns (total 29 columns):
 #   Column               Non-Null Count  Dtype
---  ------               --------------  -----
 0   emotion              2452 non-null   object
 1   vocal_channel        2452 non-null   float64
 2   emotional_intensity  2452 non-null   float64
 3   statement            2452 non-null   float64
 4   repetition           2452 non-null   float64
 5   sex                  2452 non-null   float64
 6   length_ms            2452 non-null   float64
 7   zero_crossings_sum   2452 non-null   float64
 8   mfcc_mean            2452 non-null   float64
 9   mfcc_std             2452 non-null   float64
 10  mfcc_min             2452 non-null   float64
 11  mfcc_max             2452 non-null   float64
 12  sc_mean              2452 non-null   float64
 13  sc_std               2452 non-null   float64
 14  sc_min               2452 non-null   float64
 15  sc_max               2452 non-null   float64
 16  sc_kur               2452 non-null   float64
 17  sc_skew              2452 non-null   float64
 18  stft_mean            2452 non-null   float64
 19  stft_std             2452 non-null   float64
 20  stft_min             2452 non-null   float64
 21  stft_kur             2452 non-null   float64
 22  stft_skew            2452 non-null   float64
 23  std                  2452 non-null   float64
 24  min                  2452 non-null   float64
 25  max                  2452 non-null   float64
 26  kur                  2452 non-null   float64
 27  skew                 2452 non-null   float64
 28  Emotion_val          2452 non-null   int64
dtypes: float64(27), int64(1), object(1)
memory usage: 555.7+ KB
```

In [421…
```
dfe.dtypes[dfe.dtypes.map(lambda x: x == 'object')]
```

Out[421…
```
emotion    object
dtype: object
```

In [423…
```
dfe_noobj = dfe.drop(['emotion'], axis=1)
dfe_noobj.dtypes
```

Out[423…
```
vocal_channel          float64
emotional_intensity    float64
statement              float64
repetition             float64
sex                    float64
length_ms              float64
zero_crossings_sum     float64
mfcc_mean              float64
mfcc_std               float64
mfcc_min               float64
mfcc_max               float64
sc_mean                float64
sc_std                 float64
sc_min                 float64
sc_max                 float64
sc_kur                 float64
sc_skew                float64
stft_mean              float64
stft_std               float64
stft_min               float64
stft_kur               float64
stft_skew              float64
std                    float64
min                    float64
max                    float64
kur                    float64
skew                   float64
Emotion_val              int64
dtype: object
```

In [426…
```
dfe_noobj
```

Out[426…

	vocal_channel	emotional_intensity	statement	repetition	sex	length_ms	zero_crossings_sum	mfcc_mean	mfcc_std	mf
0	0.0	0.0	0.0	1.0	0.0	3737.0	16995.0	-33.485947	134.654860	-755
1	0.0	0.0	0.0	0.0	0.0	3904.0	13906.0	-29.502108	130.485630	-713
2	0.0	1.0	0.0	1.0	0.0	4671.0	18723.0	-30.532463	126.577110	-726
3	0.0	0.0	1.0	0.0	0.0	3637.0	11617.0	-36.059555	159.725160	-842
4	1.0	1.0	0.0	1.0	0.0	4404.0	15137.0	-31.405996	122.125824	-700
...	...	...	...	...	...	...	...	...	...	
2447	0.0	1.0	1.0	0.0	1.0	4605.0	9871.0	-30.225578	158.845500	-855
2448	0.0	0.0	0.0	0.0	1.0	4171.0	8963.0	-31.160332	157.499700	-825
2449	1.0	1.0	0.0	1.0	1.0	5239.0	9765.0	-26.135280	138.133210	-768
2450	0.0	0.0	1.0	0.0	1.0	3737.0	9716.0	-28.242815	159.943400	-868
2451	1.0	0.0	0.0	1.0	1.0	3837.0	9427.0	-29.019236	149.188950	-799

2452 rows × 28 columns

Scegliamo come variabile target l'attributo "Emotion_val" e prendiamo tutte le altre per addestrare il Decision Tree.

In [429…
```
target = 'Emotion_val'
columns = [c for c in dfe_noobj.columns if c not in target]
```

In [431…
```
columns
```

Out[431…
```
['vocal_channel',
 'emotional_intensity',
 'statement',
 'repetition',
 'sex',
 'length_ms',
 'zero_crossings_sum',
 'mfcc_mean',
 'mfcc_std',
 'mfcc_min',
 'mfcc_max',
 'sc_mean',
 'sc_std',
 'sc_min',
 'sc_max',
 'sc_kur',
 'sc_skew',
 'stft_mean',
 'stft_std',
 'stft_min',
 'stft_kur',
 'stft_skew',
 'std',
 'min',
 'max',
 'kur',
 'skew']
```

In [433…
```
X = dfe_noobj[columns].values
y = dfe_noobj[target].values
```

In [435…
```
from sklearn.model_selection import train_test_split
```

In [437…
```
X_train_val, X_test, y_train_val, y_test = train_test_split(X, y, test_size=0.2, stratify=y, random_state=0)
```

In [439…
```
X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val, test_size=0.2, stratify=y_train_val
```

In [441…
```
from sklearn.tree import DecisionTreeClassifier
```

In [443…
```
from sklearn.metrics import accuracy_score, confusion_matrix
```

In [445…
```
clf = DecisionTreeClassifier(
    criterion='gini',
    max_depth=None,
    min_samples_split=2,
    min_samples_leaf=1,
    ccp_alpha=0.0,
    random_state=0
)
clf.fit(X_train, y_train)
```

```
y_pred = clf.predict(X_val)
accuracy_score(y_val, y_pred)
```

Out[445… 0.4071246819338422

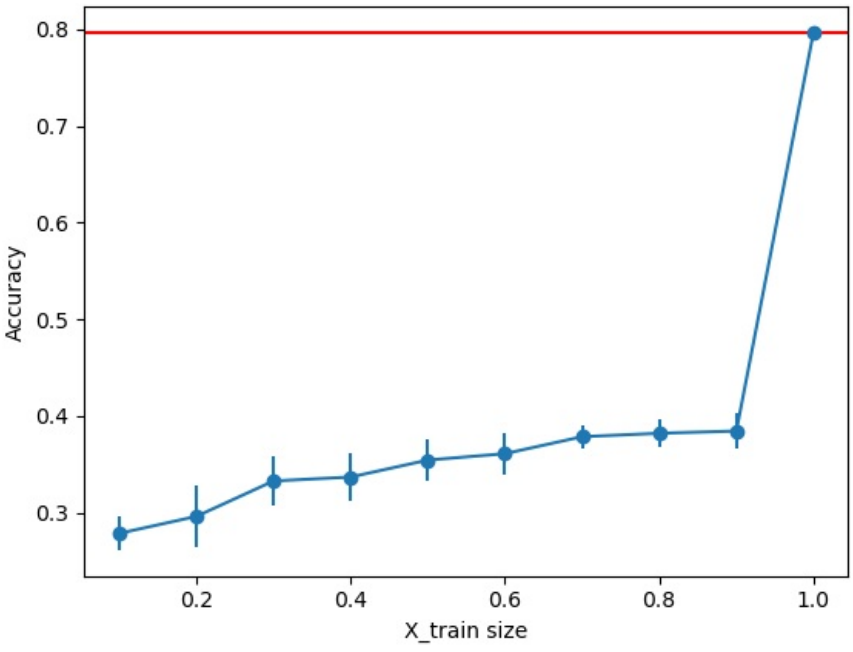
In [447…
```
confusion_matrix(y_val, y_pred)
```

Out[447…
```
array([[36,  0,  3, 10,  2,  2,  5,  2],
       [ 2, 28,  5,  3,  6,  1, 14,  1],
       [ 2,  5,  9,  4,  0,  3,  6,  2],
       [ 9,  4,  1, 20, 11,  4,  2,  9],
       [10,  4,  3,  6, 19,  2, 12,  4],
       [ 1,  5,  0,  5,  3, 10,  6,  0],
       [ 0, 11, 11,  2,  3,  6, 27,  1],
       [ 0,  6,  2,  4,  3,  2,  3, 11]])
```

Per valutare l'impatto della dimensione del Test Set sull'accuratezza del nostro modello, abbiamo plottato l'accuratezza in funzione della percentuale del dataset assegnata al Test Set. Dai risultati emerge che assegnare l'80% dell'intero dataset al Training Set offre le prestazioni migliori, caratterizzate da un'accuratezza elevata e da una minore variazione della stessa.

In [211…
```
accuracy_mean_list = list()
accuracy_std_list = list()
for p in np.arange(0.1, 1.0, 0.1):
    accuracy_list_p = list()
    for i in range(0, 10):
        index = np.random.choice(np.arange(0, len(X_train)), int(len(X_train) * p), replace=False)
        clf = DecisionTreeClassifier(
            criterion='gini',
            max_depth=None,
            min_samples_split=2,
            min_samples_leaf=1,
            ccp_alpha=0.0,
            random_state=0
        )
        clf.fit(X_train[index], y_train[index])

        y_pred = clf.predict(X_val)
        accuracy_list_p.append(accuracy_score(y_val, y_pred))
    accuracy_mean_list.append(np.mean(accuracy_list_p))
    accuracy_std_list.append(np.std(accuracy_list_p))
```

In [212…
```
accuracy_mean_list.append(0.7967914438502673)
accuracy_std_list.append(0.0)

plt.errorbar(x=np.arange(0.1, 1.1, 0.1), y=accuracy_mean_list,
             yerr=accuracy_std_list, marker='o')
plt.axhline(y=0.7967914438502673, color='r')
plt.ylabel('Accuracy')
plt.xlabel('X_train size')
plt.show()
```

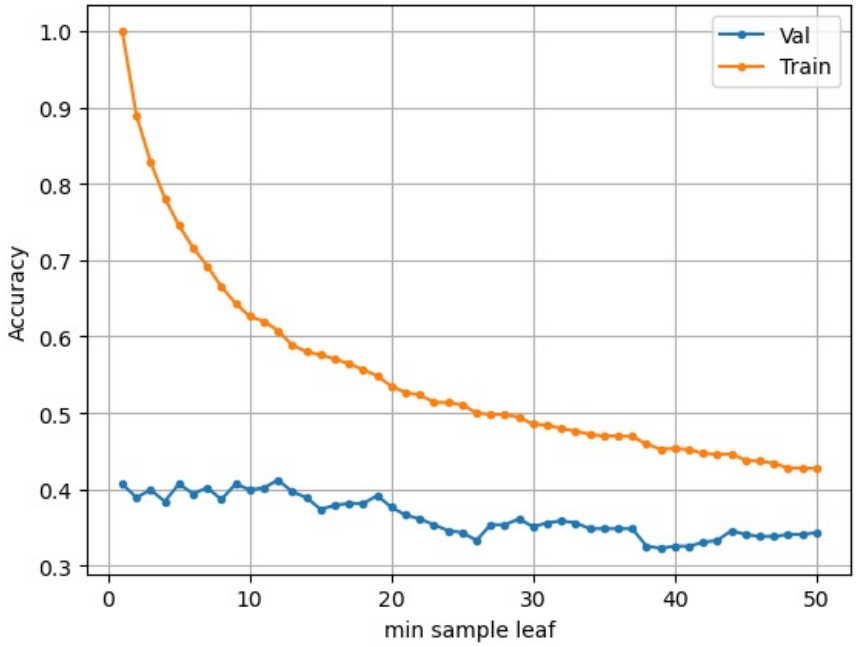


Come per l'analisi effettuata per la classificazione del sesso tramite Decision Tree, abbiamo proceduto a plottare l'effetto del parametro **min_samples_leaf** sull'accuratezza del modello. Questo parametro determina il numero minimo di campioni che devono essere presenti in una foglia. Dai nostri risultati, emerge che impostare min_samples_leaf a 12 ottimizza l'accuratezza del modello. Generalmente però

bisogna sottolineare che un numero più alto di questo valore produce un metodo più robusto.

In [451…
```
accuracy_val_list = list()
accuracy_train_list = list()
for min_sample_leaf in range(1,51):
    clf = DecisionTreeClassifier(
        criterion='gini',
        max_depth=None,
        min_samples_split=2,
        min_samples_leaf=min_sample_leaf,
        ccp_alpha=0.0,
        random_state=0
    )
    clf.fit(X_train, y_train)

    y_pred = clf.predict(X_val)
    accuracy_val_list.append(accuracy_score(y_val, y_pred))

    y_pred_train = clf.predict(X_train)
    accuracy_train_list.append(accuracy_score(y_train, y_pred_train))
```

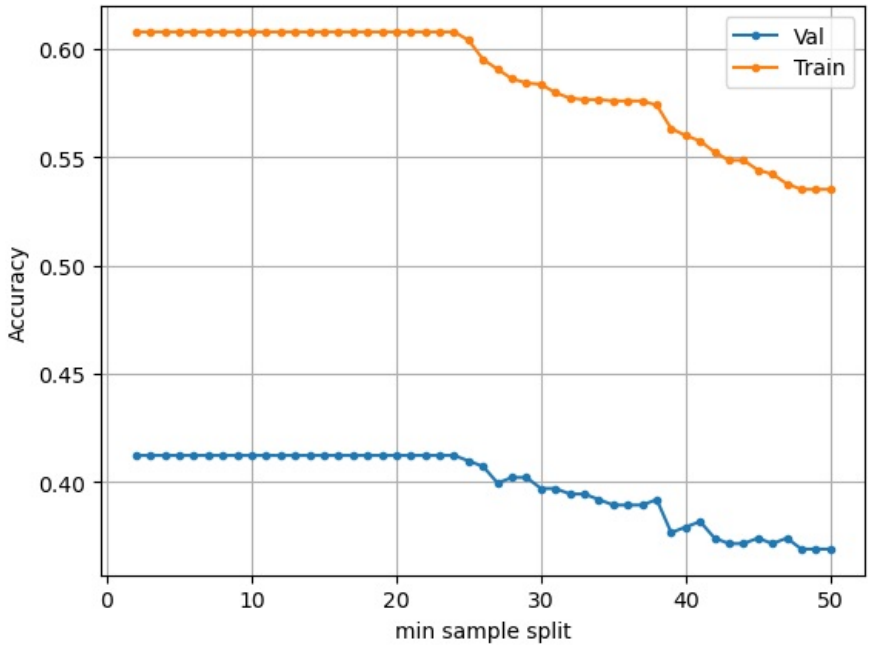
In [453…
```
plt.plot(min_sample_leaf_list, accuracy_val_list, label='Val', marker='.')
plt.plot(min_sample_leaf_list, accuracy_train_list, label='Train', marker='.')
plt.ylabel('Accuracy')
plt.xlabel('min sample leaf')
plt.grid('white')
plt.legend()
plt.show()
```



Abbiamo condotto un'ulteriore analisi per determinare come il parametro **min_samples_split** (il numero minimo di campioni che un nodo deve avere per essere considerato per la divisione) influisca sull'accuratezza del nostro modello. Dalla visualizzazione, possiamo osservare che i valori <=24 offrano le migliori prestazioni in termini di accuratezza.

In [456…
```
min_sample_split_list = np.arange(2, 50+1, 1)
min_sample_split_list

accuracy_val_list = list()
accuracy_train_list = list()
for min_sample_split in min_sample_split_list:
    clf = DecisionTreeClassifier(
        criterion='gini',
        max_depth=None,
        min_samples_split=min_sample_split,
        min_samples_leaf=12,
        ccp_alpha=0.0,
        random_state=0
    )
    clf.fit(X_train, y_train)

    y_pred = clf.predict(X_val)
    accuracy_val_list.append(accuracy_score(y_val, y_pred))

    y_pred_train = clf.predict(X_train)
    accuracy_train_list.append(accuracy_score(y_train, y_pred_train))
```

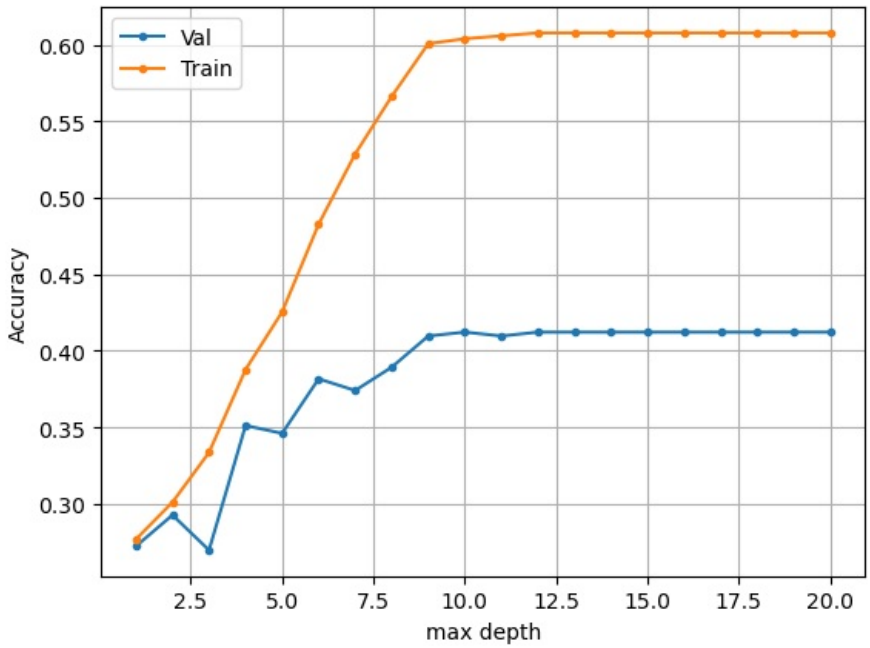
```
plt.plot(min_sample_split_list, accuracy_val_list, label='Val', marker='.')
plt.plot(min_sample_split_list, accuracy_train_list, label='Train', marker='.')
plt.ylabel('Accuracy')
plt.xlabel('min sample split')
plt.grid('white')
plt.legend()
plt.show()
```



Abbiamo ripetuto l'analisi precedente per valutare come la massima profondità dell'albero **(max_depth)** influisca sull'accuratezza del nostro modello di classificazione. Plottando l'accuratezza del modello in funzione di diverse profondità massime, abbiamo identificato che il valore di max_depth pari a 10 offre il miglior risultato in termini di accuratezza. Questo suggerisce che un albero con una profondità massima di 10 è sufficientemente complesso da catturare le relazioni significative nei dati senza andare in overfitting.

In [462…

```
max_depth_list = np.arange(1, 20+1, 1).tolist() + [None]

accuracy_val_list = list()
accuracy_train_list = list()
for max_depth in max_depth_list:
    clf = DecisionTreeClassifier(
        criterion='gini',
        max_depth=max_depth,
        min_samples_split=24,
        min_samples_leaf=12,
        ccp_alpha=0.0,
        random_state=0
    )
    clf.fit(X_train, y_train)

    y_pred = clf.predict(X_val)
    accuracy_val_list.append(accuracy_score(y_val, y_pred))

    y_pred_train = clf.predict(X_train)
    accuracy_train_list.append(accuracy_score(y_train, y_pred_train))

plt.plot(max_depth_list, accuracy_val_list, label='Val', marker='.')
plt.plot(max_depth_list, accuracy_train_list, label='Train', marker='.')
plt.ylabel('Accuracy')
plt.xlabel('max depth')
plt.grid('white')
plt.legend()
plt.show()
```



Procederemo ora all'utilizzo di **RandomizedSearchCV**, una tecnica di ottimizzazione automatica fornita da scikit-learn, per identificare la migliore combinazione di parametri per il nostro modello Decision Tree. Diversamente dalla ricerca esaustiva su griglia (GridSearchCV), RandomizedSearchCV riduce il tempo computazionale, ma permette comunque di esplorare un ampio spazio dei parametri. Questo approccio è particolarmente utile per ottimizzare modelli complessi su grandi set di dati, poiché consente di trovare una buona combinazione di parametri in modo più efficiente. Imposteremo un range di valori per ciascun parametro di interesse e lasceremo che RandomizedSearchCV determini la combinazione ottimale attraverso validazione incrociata.

In [227…
```
from sklearn.model_selection import RandomizedSearchCV
```

In [229…
```
clf = DecisionTreeClassifier(
        criterion='gini',
        max_depth=10,
        min_samples_split=10,
        min_samples_leaf=11,
        ccp_alpha=0.0,
        random_state=0)
```

In [232…
```
param_dict = {
    'max_depth': np.arange(1, 20+1, 1).tolist(),
    'min_samples_split': np.arange(2, 50+1, 1),
    'min_samples_leaf': np.arange(1, 50+1, 1),
    'ccp_alpha': np.arange(0.0,1, 0.1)
}
```

In [234…
```
random = RandomizedSearchCV(clf, param_dict, cv=5, scoring='accuracy', refit=True, n_iter=500, random_state=0)
random.fit(X_train_val, y_train_val)
random.best_params_
```

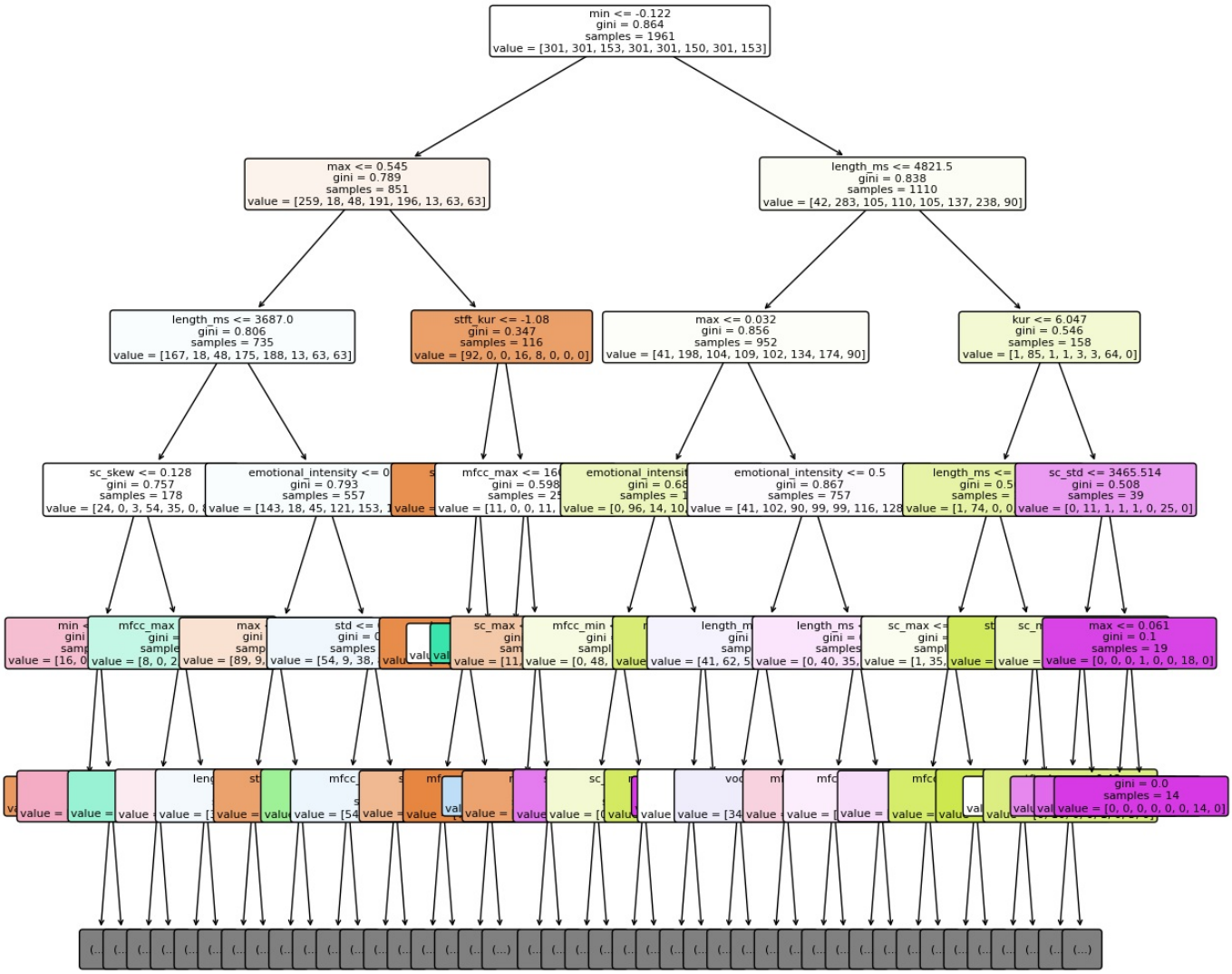
Out[234…
```
{'min_samples_split': 6,
 'min_samples_leaf': 5,
 'max_depth': 16,
 'ccp_alpha': 0.0}
```

Possiamo vedere che i valori dei parametri individuati tramite **RandomizedSearchCV** non corrispondono a quelli precedentemente identificati variando un parametro alla volta. Questa differenza sottolinea l'efficacia di RandomizedSearchCV nell'esplorare le varie combinazioni di parametri in modo più ampio e sistematico, considerando le interazioni tra più parametri contemporaneamente. Di conseguenza, abbiamo deciso di adottare i parametri suggeriti da RandomizedSearchCV per il nostro modello Decision Tree.

In [236…
```
clf = random.best_estimator_
```

In [237…
```
from sklearn.tree import plot_tree, export_text
```

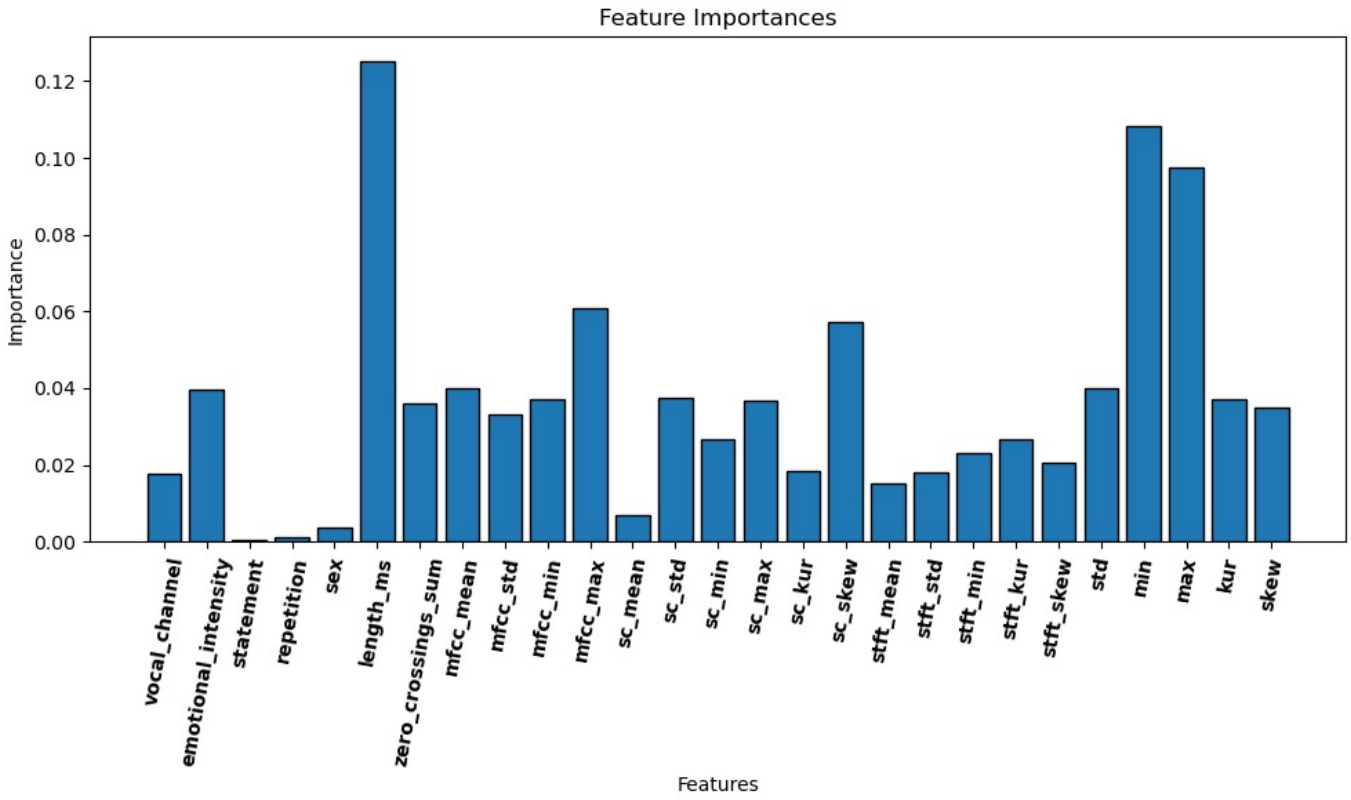
In [238…
```
plt.figure(figsize=(14, 14))
plot_tree(clf,
          feature_names=columns,
          filled=True,
          rounded=True,
          fontsize=8,
          max_depth=5
          )
plt.show()
```



Utilizzando una barchart, abbiamo visualizzato **l'importanza delle diverse features** del nostro dataset nell'influenzare la previsione della variabile 'Emotion' con il modello di Decision Tree. L'analisi rivela che la feature 'length_ms' è la più influente, seguita da 'min' e 'max'. Queste informazioni sono state ottenute analizzando quanto ogni feature sia efficace nel separare le diverse classi di 'Emotion' nel dataset.

In [240…

```
plt.figure(figsize=(10, 6))
plt.bar(columns, clf.feature_importances_, edgecolor='k')
plt.xticks(rotation=80, fontweight='bold')
plt.title('Feature Importances')
plt.xlabel('Features')
plt.ylabel('Importance')
plt.tight_layout()
plt.show()
```



Abbiamo valutato l'accuratezza del nostro modello di Decision Tree sul Test Set, ottenendo un risultato di 0.43. Oltre all'accuratezza, abbiamo calcolato anche Precision, Recall, F1-Score e generato la Confusion Matrix per avere una visione più completa delle prestazioni del modello. Per quanto riguarda il bilanciamento tra le classi predette correttamente e quelle erroneamente interpretate, è interessante notare che il modello preveda con maggiore accuratezza le emozioni 'angry' (0) e 'calm' (1), in linea con quanto osservato nella sezione di data understanding, dove queste emozioni risultavano essere ben separate dalle altre. Al contrario, le emozioni 'suprised' (7), fearful (3), disgust (2) e 'sad' (6) sono quelle che il modello trova più difficili da prevedere.

In [242…
```
y_pred = clf.predict(X_test)
y_pred_trainval=clf.predict(X_train_val)

accuracy_score(y_test, y_pred)
```

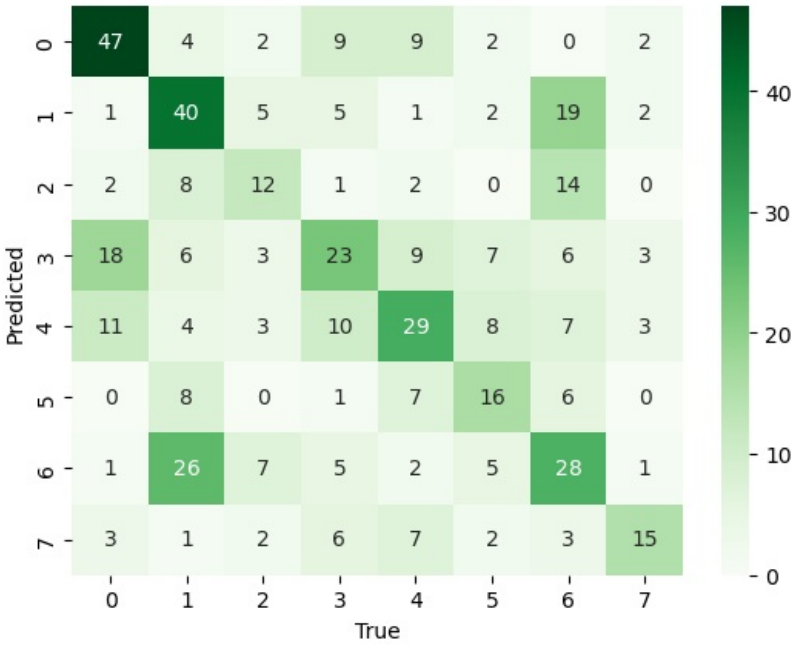
Out[242…
```
0.42769857433808556
```

In [243…
```
from sklearn.metrics import precision_score, recall_score, f1_score
precision = precision_score(y_test, y_pred,average='weighted')
recall = recall_score(y_test, y_pred,average='weighted')
f1 = f1_score(y_test, y_pred,average='weighted')
conf_matrix = confusion_matrix(y_test, y_pred)

print("Precision:", precision)
print("Recall:", recall)
print("F1 Score:", f1)
print("Confusion Matrix:", conf_matrix)
print(classification_report(y_pred, y_test))

print(classification_report(y_pred_trainval, y_train_val))
```

```
Precision: 0.43002924888387906
Recall: 0.42769857433808556
F1 Score: 0.4246737062512508
Confusion Matrix: [[47  4  2  9  9  2  0  2]
 [ 1 40  5  5  1  2 19  2]
 [ 2  8 12  1  2  0 14  0]
 [18  6  3 23  9  7  6  3]
 [11  4  3 10 29  8  7  3]
 [ 0  8  0  1  7 16  6  0]
 [ 1 26  7  5  2  5 28  1]
 [ 3  1  2  6  7  2  3 15]]
              precision    recall  f1-score   support

           0       0.63      0.57      0.59        83
           1       0.53      0.41      0.47        97
           2       0.31      0.35      0.33        34
           3       0.31      0.38      0.34        60
           4       0.39      0.44      0.41        66
           5       0.42      0.38      0.40        42
           6       0.37      0.34      0.35        83
           7       0.38      0.58      0.46        26

    accuracy                           0.43       491
   macro avg       0.42      0.43      0.42       491
weighted avg       0.44      0.43      0.43       491

              precision    recall  f1-score   support

           0       0.84      0.79      0.82       317
           1       0.86      0.75      0.80       347
           2       0.69      0.70      0.69       150
           3       0.71      0.77      0.74       279
           4       0.72      0.74      0.73       293
           5       0.70      0.75      0.72       140
           6       0.72      0.74      0.73       292
           7       0.70      0.75      0.72       143

    accuracy                           0.75      1961
   macro avg       0.74      0.75      0.74      1961
weighted avg       0.76      0.75      0.75      1961
```

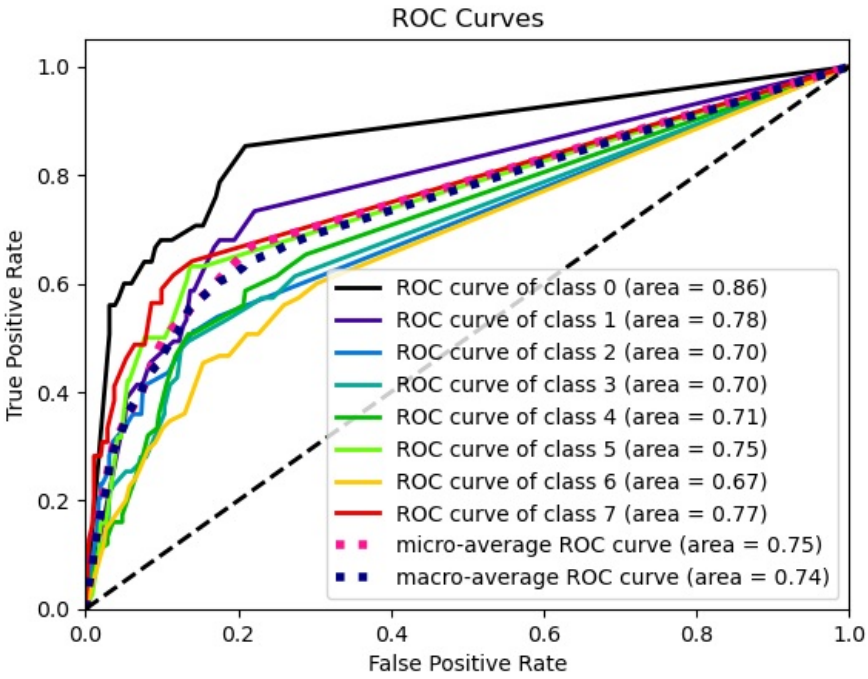
Visualizziamo anche la Confusion Matrix e plottiamo la ROC curve per questo modello predittivo

In [245…
```
import seaborn as sns
cf = confusion_matrix(y_test, y_pred)
sns.heatmap(cf, annot=True, cmap="Greens")
plt.xlabel("True")
plt.ylabel("Predicted")
plt.show()
```



In [247…
```
import scikitplot
y_pred_proba = clf.predict_proba(X_test)
scikitplot.metrics.plot_roc(y_test,y_pred_proba)
```

Out[247… <Axes: title={'center': 'ROC Curves'}, xlabel='False Positive Rate', ylabel='True Positive Rate'>



KNN on Emotion

Per l'analisi delle 'Emotion' con il modello KNN, applicheremo la stessa procedura e gli stessi passi precedentemente discussi e analizzati nella classificazione della variabile Sex.

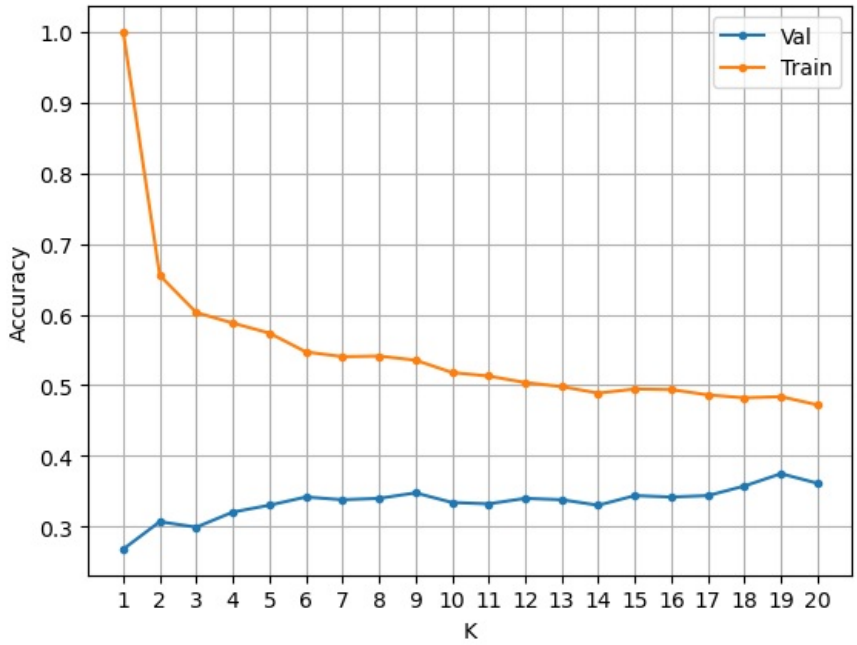
In [473…

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report

target = 'Emotion_val'
columns = [c for c in dfe_noobj.columns if c not in target]

X = dfe_noobj[columns].values
Y = dfe_noobj[target].values


scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

X_train_val, X_test, y_train_val, y_test = train_test_split(X_scaled, Y, test_size=0.3, stratify=Y, random_stat
X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val, test_size=0.3, stratify=y_train_val
print("Training set", X_train.shape, y_train.shape)
print("Validation set", X_val.shape, y_val.shape)
print("Test set", X_test.shape, y_test.shape)
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)
y_pred_val = knn.predict(X_val)
y_pred_train = knn.predict(X_train)
y_pred=knn.predict(X_test)


print("Test set:\n",classification_report(y_pred, y_test))
print(accuracy_score(y_val, y_pred_val), accuracy_score(y_train, y_pred_train))
print(classification_report(y_val, y_pred_val))
```

```
Training set (1201, 27) (1201,)
Validation set (515, 27) (515,)
Test set (736, 27) (736,)
Test set:
              precision    recall  f1-score   support

           0       0.58      0.40      0.47       167
           1       0.54      0.39      0.45       157
           2       0.22      0.25      0.24        52
           3       0.27      0.33      0.30        93
           4       0.25      0.26      0.25       108
           5       0.25      0.24      0.25        58
           6       0.17      0.35      0.23        54
           7       0.33      0.40      0.37        47

    accuracy                           0.34       736
   macro avg       0.33      0.33      0.32       736
weighted avg       0.39      0.34      0.35       736

0.3300970873786408 0.5736885928393006
              precision    recall  f1-score   support

           0       0.41      0.61      0.49        79
           1       0.41      0.48      0.44        79
           2       0.34      0.28      0.31        40
           3       0.25      0.23      0.24        79
           4       0.24      0.27      0.25        79
           5       0.34      0.25      0.29        40
           6       0.31      0.22      0.26        79
           7       0.24      0.17      0.20        40

    accuracy                           0.33       515
   macro avg       0.32      0.31      0.31       515
weighted avg       0.32      0.33      0.32       515
```

In [475…
```python
acc_val_list = list()
acc_train_list = list()
for k in np.arange(1, 20+1, 1):
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train, y_train)
    y_pred = knn.predict(X_val)
    y_pred_train = knn.predict(X_train)
    acc_val_list.append(accuracy_score(y_val, y_pred))
    acc_train_list.append(accuracy_score(y_train, y_pred_train))
```

In [477…
```python
plt.plot(np.arange(1, 20+1, 1), acc_val_list, label='Val', marker='.')
plt.plot(np.arange(1, 20+1, 1), acc_train_list, label='Train', marker='.')
plt.xlabel('K')
plt.ylabel('Accuracy')
plt.xticks(np.arange(1, 20+1, 1))
plt.grid()
plt.legend()
plt.show()
```

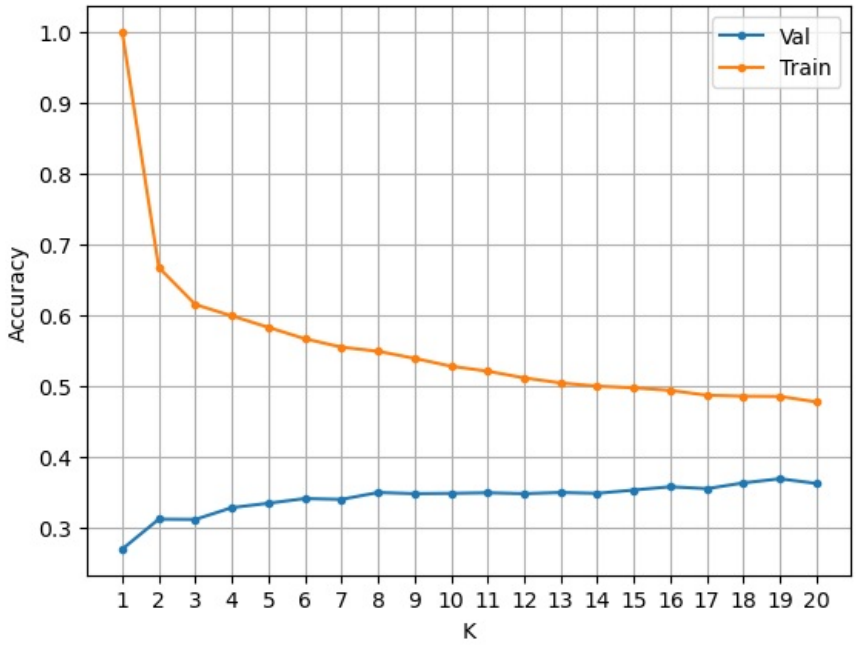


In [479…
```python
nbr_holdout = 5
acc_val_list_all = list()
```

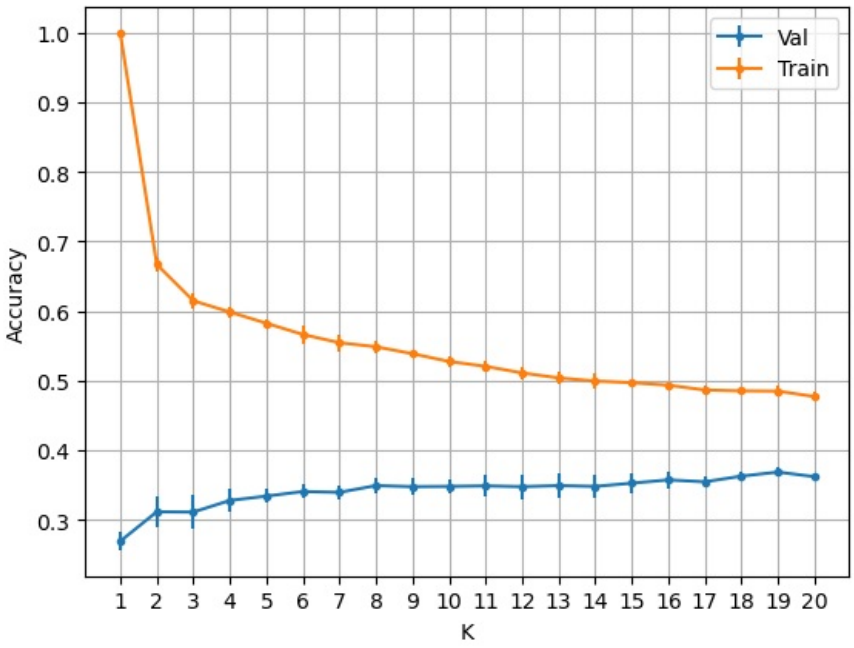
```
acc_train_list_all = list()
for i in range(nbr_holdout):

    X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val,
                                                      test_size=0.3,
                                                      stratify=y_train_val,
                                                      random_state=i)

    acc_val_list = list()
    acc_train_list = list()
    for k in np.arange(1, 20+1, 1):
        knn = KNeighborsClassifier(n_neighbors=k)
        knn.fit(X_train, y_train)
        y_pred = knn.predict(X_val)
        y_pred_train = knn.predict(X_train)
        acc_val_list.append(accuracy_score(y_val, y_pred))
        acc_train_list.append(accuracy_score(y_train, y_pred_train))

    acc_val_list_all.append(acc_val_list)
    acc_train_list_all.append(acc_train_list)
```

In [480…
```
acc_val_list_all = np.array(acc_val_list_all)
acc_train_list_all = np.array(acc_train_list_all)
```

In [483…
```
plt.plot(np.arange(1, 20+1, 1), np.mean(acc_val_list_all, axis=0), label='Val', marker='.')
plt.plot(np.arange(1, 20+1, 1), np.mean(acc_train_list_all, axis=0), label='Train', marker='.')
plt.xlabel('K')
plt.ylabel('Accuracy')
plt.xticks(np.arange(1, 20+1, 1))
plt.grid()
plt.legend()
plt.show()
```



In [485…
```
plt.errorbar(x=np.arange(1, 20+1, 1),
             y=np.mean(acc_val_list_all, axis=0),
             yerr=np.std(acc_val_list_all, axis=0),
             label='Val', marker='.')

plt.errorbar(x=np.arange(1, 20+1, 1),
             y=np.mean(acc_train_list_all, axis=0),
             yerr=np.std(acc_train_list_all, axis=0),
             label='Train', marker='.')

plt.xlabel('K')
plt.ylabel('Accuracy')
plt.xticks(np.arange(1, 20+1, 1))
plt.grid()
plt.legend()
plt.show()
```

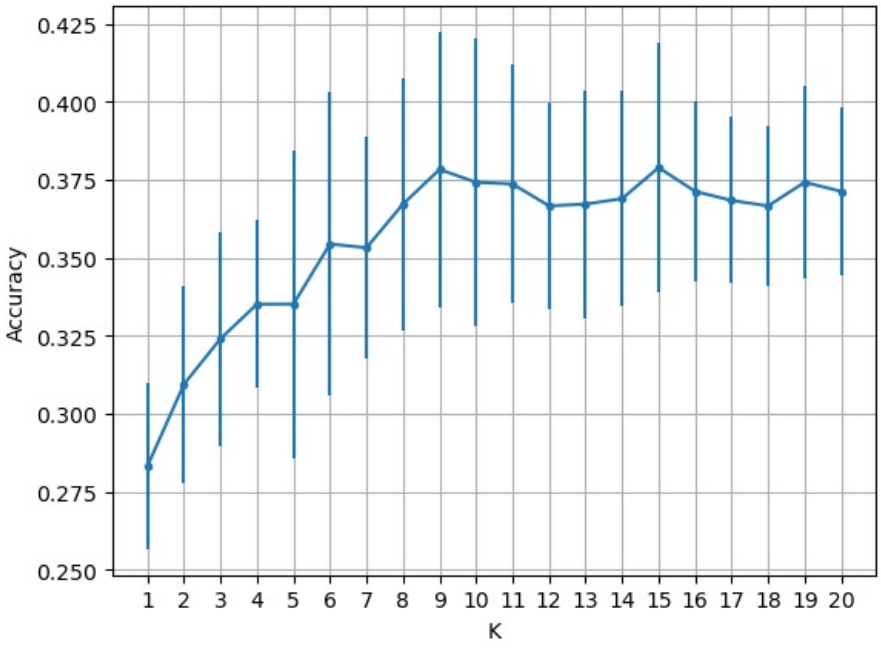


Cross Validation

In [488…
```
from sklearn.model_selection import cross_val_score
knn = KNeighborsClassifier(n_neighbors=5)
scores = cross_val_score(knn, X_train_val, y_train_val, cv=10)
```

In [490…
```
acc_list_mean = list()
acc_list_std = list()
for k in np.arange(1, 20+1, 1):
    knn = KNeighborsClassifier(n_neighbors=k)
    scores = cross_val_score(knn, X_train_val, y_train_val, cv=10)
    acc_list_mean.append(np.mean(scores))
    acc_list_std.append(np.std(scores))
```

In [491…
```
plt.errorbar(x=np.arange(1, 20+1, 1),
             y=acc_list_mean,
             yerr=acc_list_std,
             marker='.')

plt.xlabel('K')
plt.ylabel('Accuracy')
plt.xticks(np.arange(1, 20+1, 1))
plt.grid()
plt.show()
```



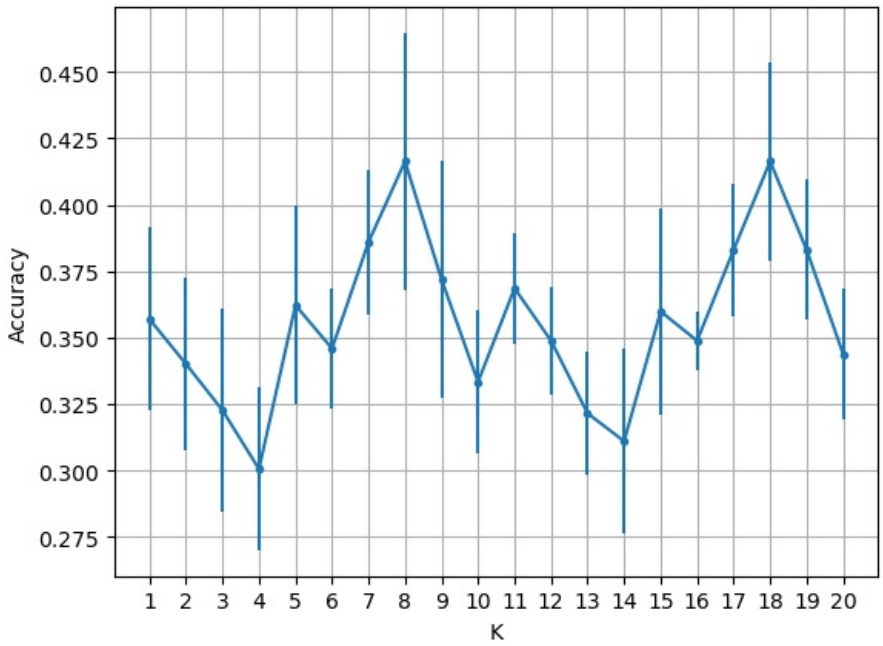
Dall'analisi del grafico plottato sotto, il numero migliore di K per il modello KNN su Emotion sembrano essere 8 e 18. Abbiamo deciso di scegliere K=18 per il nostro modello predittivo perché dal grafico sembra avere una variazione più bassa.

In [495…
```
nbr_repetitions = 5
acc_list_all = list()
for i in range(nbr_repetitions):
    acc_list = list()
    for k in np.arange(1, 20+1, 1):
        knn = KNeighborsClassifier(n_neighbors=k)
        scores = cross_val_score(knn, X_train_val, y_train_val, cv=10, #k-fold
                                 scoring='accuracy')
        acc_list.append(scores)

    acc_list_all.append(acc_list)
```

In [496…
```
acc_list_all = np.array(acc_list_all)
acc_list_all.shape
acc_list_all.reshape(50, 20)
np.mean(acc_list_all.reshape(50, 20), axis=0)
```

Out[496…
```
array([0.35697674, 0.34011628, 0.32267442, 0.3005814 , 0.3622093 ,
       0.34593023, 0.38596491, 0.41637427, 0.37192982, 0.33333333,
       0.36860465, 0.34883721, 0.32151163, 0.31104651, 0.35988372,
       0.34883721, 0.38304094, 0.41637427, 0.38304094, 0.34385965])
```

In [497…
```
plt.errorbar(x=np.arange(1, 20+1, 1),
             y=np.mean(acc_list_all.reshape(50, 20), axis=0),
             yerr=np.std(acc_list_all.reshape(50, 20), axis=0),
             marker='.')

plt.xlabel('K')
plt.ylabel('Accuracy')
plt.xticks(np.arange(1, 20+1, 1))
plt.grid()
plt.show()
```



Dall'analisi del report generato, osserviamo che il modello KNN ha un'accuratezza simile ma leggermente più bassa rispetto al Decision Tree. Inoltre, è possibile notare come anche per il modello KNN, le emozioni 'angry' e 'calm' sono quelle che vengono previste meglio. Notiamo invece che le precision per le emozioni 'sad', 'happy' sono basse e addirittura peggiori che nel caso del Decision Tree. Questo riflette l'analisi fatta nella data preparation che indicava che queste emozioni condividono caratteristiche simili e ciò rende più complessa la loro distinzione da parte dei modelli utilizzati. Un'altra cosa molto interessante da evidenziare per questo modello è come la precision per Surprised (class=7) sia significativamente più alta rispetto al Decision Tree.

In [501…
```
knn = KNeighborsClassifier(n_neighbors=18)
knn.fit(X_train_val, y_train_val)

y_pred = knn.predict(X_test)
print("Test set:\n",classification_report(y_pred, y_test))

y_pred_trainval = knn.predict(X_train_val)
print(classification_report(y_pred_trainval, y_train_val))
```

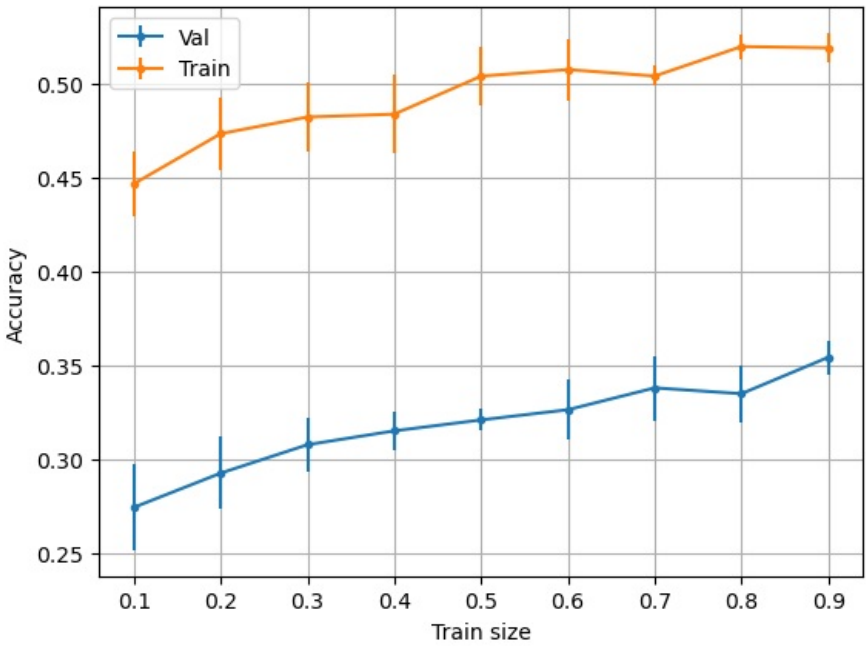
```
Test set:
              precision    recall  f1-score   support

           0       0.65      0.61      0.63       122
           1       0.50      0.41      0.45       135
           2       0.36      0.38      0.37        55
           3       0.38      0.42      0.40       102
           4       0.23      0.23      0.23       112
           5       0.34      0.28      0.31        68
           6       0.19      0.30      0.23        70
           7       0.54      0.43      0.48        72

    accuracy                           0.40       736
   macro avg       0.40      0.38      0.39       736
weighted avg       0.42      0.40      0.40       736

              precision    recall  f1-score   support

           0       0.74      0.63      0.68       308
           1       0.62      0.54      0.58       306
           2       0.35      0.42      0.38       112
           3       0.49      0.47      0.48       274
           4       0.47      0.49      0.48       255
           5       0.45      0.40      0.43       149
           6       0.31      0.46      0.37       176
           7       0.40      0.40      0.40       136

    accuracy                           0.50      1716
   macro avg       0.48      0.48      0.47      1716
weighted avg       0.52      0.50      0.50      1716
```

Attraverso l'analisi delle **Learning Curves**, abbiamo valutato come l'accuratezza del nostro modello varia al cambiare della dimensione del Training Set. Dai risultati ottenuti con il DataSet in esame, emerge che destinare l'70% dei dati al Training Set offre le migliori prestazioni in termini di accuratezza. Questo ci fornisce una linea guida su quale proporzione di dati utilizzare per massimizzare l'efficacia del modello. Lo stesso tipo di analisi potrebbe essere fatta per calcolare la migliore dimensione del Training Set riguardo agli altri parametri di valutazione come la precision o la recall.

In [505…
```python
nbr_repetitions = 5
acc_val_list_all = list()
acc_train_list_all = list()
for p in np.arange(0.1, 1.0, 0.1):
    acc_val_list = list()
    acc_train_list = list()
    for i in range(nbr_repetitions):
        index = np.random.choice(np.arange(len(X_train)), int(len(X_train) * p), replace=False)
        knn = KNeighborsClassifier(n_neighbors=10)
        knn.fit(X_train[index], y_train[index])
        y_pred = knn.predict(X_val)
        y_pred_train = knn.predict(X_train[index])
        acc_val_list.append(accuracy_score(y_val, y_pred))
        acc_train_list.append(accuracy_score(y_train[index], y_pred_train))

    acc_val_list_all.append(acc_val_list)
    acc_train_list_all.append(acc_train_list)
```

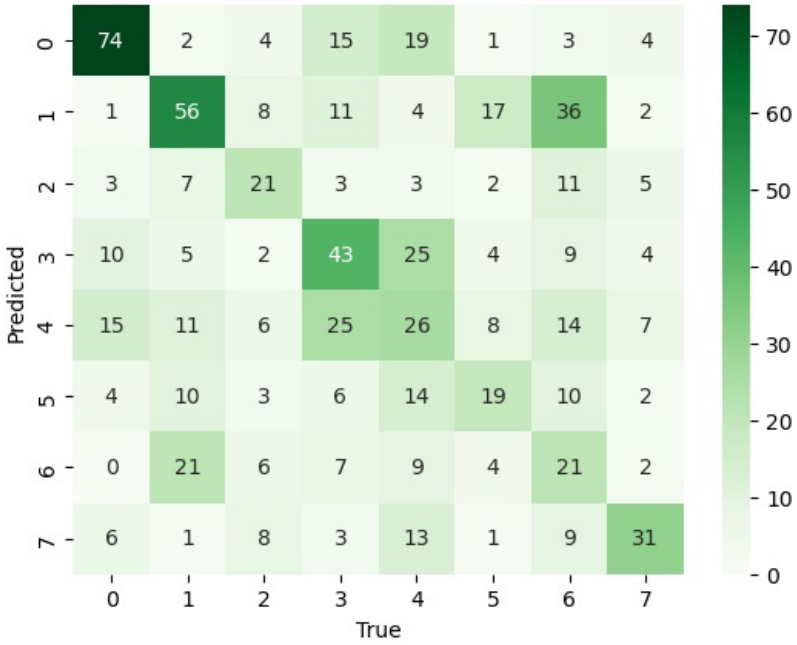
In [507…
```python
acc_val_list_all = np.array(acc_val_list_all)
acc_train_list_all = np.array(acc_train_list_all)
```

In [509…
```python
plt.errorbar(x=np.arange(0.1, 1.0, 0.1),
             y=np.mean(acc_val_list_all, axis=1),
             yerr=np.std(acc_val_list_all, axis=1),
             label='Val', marker='.')

plt.errorbar(x=np.arange(0.1, 1.0, 0.1),
             y=np.mean(acc_train_list_all, axis=1),
             yerr=np.std(acc_train_list_all, axis=1),
             label='Train', marker='.')

plt.xlabel('Train size')
plt.ylabel('Accuracy')
plt.xticks(np.arange(0.1, 1.0, 0.1))
plt.grid()
plt.legend()
plt.show()
```



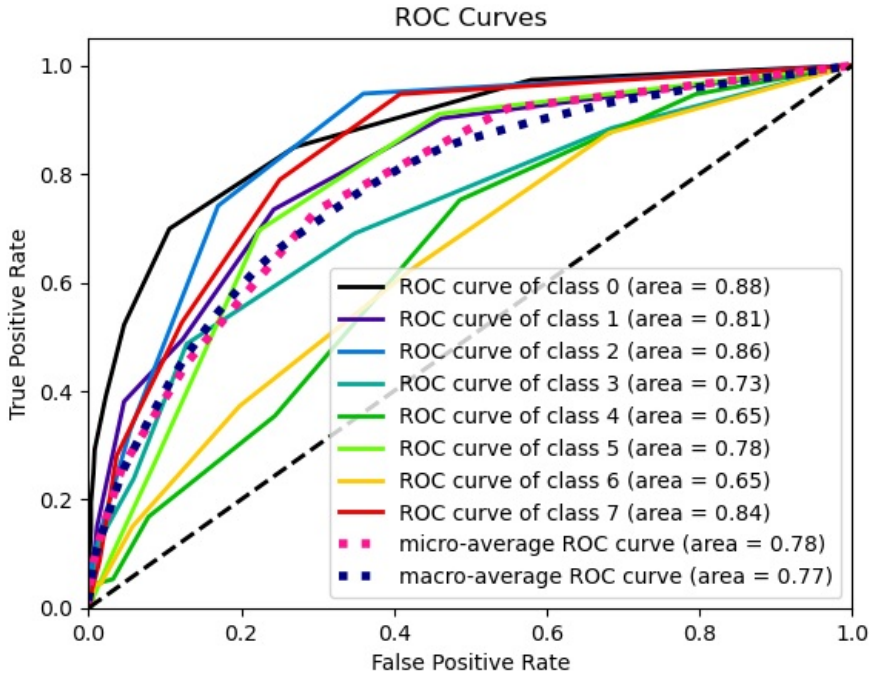
Nelle celle sotto abbiamo disegnato la confusion matrix per questo modello e la ROC curve

In [303…
```
import seaborn as sns
cf = confusion_matrix(y_pred, y_test)
sns.heatmap(cf, annot=True, cmap="Greens")
plt.xlabel("True")
plt.ylabel("Predicted")
plt.show()
```



In [299…
```
import scikitplot
y_pred_proba = knn.predict_proba(X_test)
scikitplot.metrics.plot_roc(y_test,y_pred_proba)
```

Out[299… `<Axes: title={'center': 'ROC Curves'}, xlabel='False Positive Rate', ylabel='True Positive Rate'>`



Random Forest Emotion

In questa sezione, abbiamo esplorato l'utilizzo dei **metodi ensemble**, che migliorano l'accuratezza delle predizioni aggregando i risultati di più classificatori, basandosi sul concetto di **"wisdom of the crowd"**. Uno di questi metodi ensemble è il **RandomForest**, che combina le previsioni di numerosi Decision Tree. Per ogni attributo, il RandomForest restituisce la moda dei risultati ottenuti dai vari alberi di decisione e questo tende a migliorare l'accuratezza e la robustezza del modello riducendo il rischio di overfitting. Questo metodo è particolarmente efficace perché sfrutta la diversità tra gli alberi di decisione, ognuno dei quali è addestrato su un sottoinsieme casuale dei dati (**bootstrap sampling**) e caratteristiche, selezionate anche loro in maniera casuale. Applicheremo lo stesso approccio che abbiamo utilizzato con il Decision Tree per prevedere 'Emotion'.

In [305…
```
from sklearn.ensemble import RandomForestClassifier
target = 'Emotion_val'
columns = [c for c in dfe_noobj.columns if c not in target]
```

In [307…
```
X = dfe_noobj[columns].values
Y = dfe_noobj[target].values
```

In [309…
```
X_train_val, X_test, y_train_val, y_test = train_test_split(X, y, test_size=0.2, stratify=y, random_state=0)
X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val, test_size=0.2, stratify=y_train_val
```

In [311…
```
clf = RandomForestClassifier (criterion='gini',
    max_depth=11,
    min_samples_split=11,
    min_samples_leaf=3,
    ccp_alpha=0.0,
    random_state=0
)

param_dict = {
    'max_depth': np.arange(1, 20+1, 1).tolist(),
    'min_samples_split': np.arange(2, 50+1, 1),
    'min_samples_leaf': np.arange(1, 50+1, 1),
    'ccp_alpha': np.arange(0.0,1, 0.1)
}
```

```
random = RandomizedSearchCV(clf, param_dict, cv=5, scoring='accuracy', refit=True, n_iter=500, random_state=0)
random.fit(X_train_val, y_train_val)
random.best_params_

best_params = random.best_params_

clf = random.best_estimator_

clf.fit(X_train, y_train)
```

Out[311…

```
RandomForestClassifier(max_depth=16, min_samples_leaf=5, min_samples_split=6,
                       random_state=0)
```

L'accuratezza calcolata per il modello RandomForest è **0.49** e quindi superiore rispetto ai modelli Decision Tree e KNN. Questo evidenzia la potenza dell'approccio ensemble, in particolare la capacità del RandomForest di aggregare le previsioni di molti alberi decisionali per ridurre il rischio di overfitting e gestire efficacemente la varianza. Possiamo notare che questo modello mostra un'alta accuratezza nella previsione delle emozioni 'angry' e 'calm' (similarmente ai modelli precedenti ma con un aumento significativo), e dimostra anche un notevole miglioramento per le emozioni 'fearful' e 'surprised' rispetto al KNN e al Decision Tree. Questo sembra dimostrare la robustezza del RandomForest nel catturare le differenze di dati che caratterizzano queste emozioni specifiche. Al contrario, le emozioni 'disgust', 'sad' e 'neutral' registrano una precisione più bassa.

In [313…

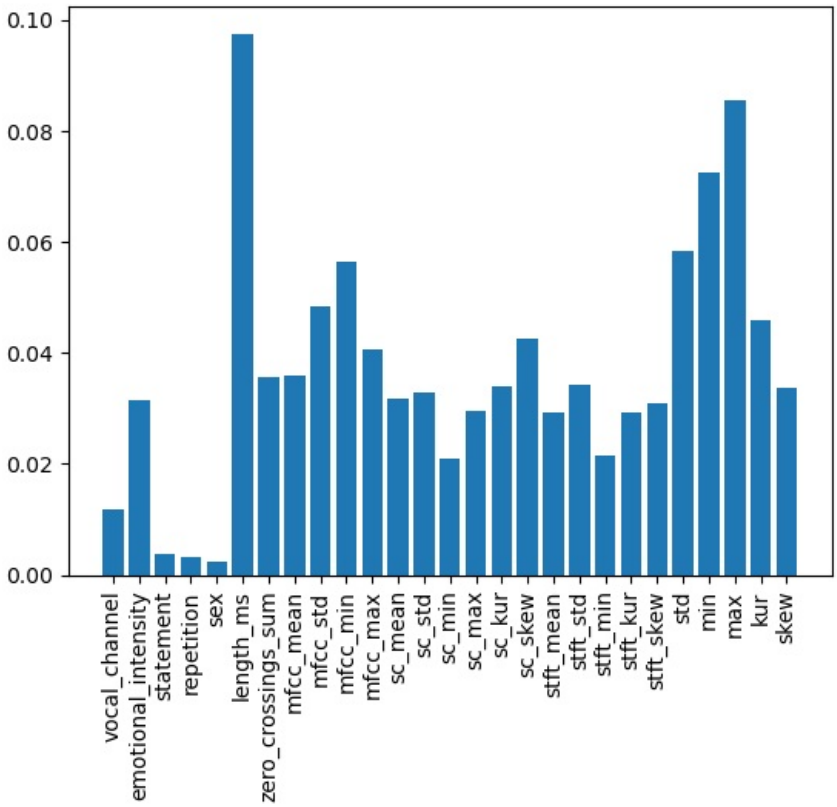
```
y_pred = clf.predict(X_test)
y_pred_trainval = clf.predict(X_train_val)

accuracy = accuracy_score(y_test, y_pred)
print(accuracy)

print(classification_report(y_pred, y_test))
print(classification_report(y_pred_trainval, y_train_val))
```

```
0.48676171079429736
              precision    recall  f1-score   support

           0       0.80      0.66      0.72        91
           1       0.83      0.50      0.62       124
           2       0.26      0.53      0.34        19
           3       0.32      0.56      0.41        43
           4       0.36      0.39      0.37        70
           5       0.29      0.23      0.26        48
           6       0.27      0.40      0.32        50
           7       0.64      0.54      0.59        46

    accuracy                           0.49       491
   macro avg       0.47      0.48      0.45       491
weighted avg       0.56      0.49      0.51       491

              precision    recall  f1-score   support

           0       0.93      0.87      0.90       319
           1       0.92      0.76      0.83       365
           2       0.75      0.89      0.82       129
           3       0.81      0.86      0.83       284
           4       0.82      0.80      0.81       310
           5       0.80      0.84      0.82       143
           6       0.76      0.87      0.81       262
           7       0.79      0.81      0.80       149

    accuracy                           0.83      1961
   macro avg       0.82      0.84      0.83      1961
weighted avg       0.84      0.83      0.83      1961
```

Dall'analisi della **Feature Importance** del Random Forest si può notare che le variabili *min,max,length_ms* sono le più importanti per prevedere le Emozioni.

In [315…

```
plt.bar(columns, clf.feature_importances_)
plt.xticks(rotation=90)
plt.show()
```

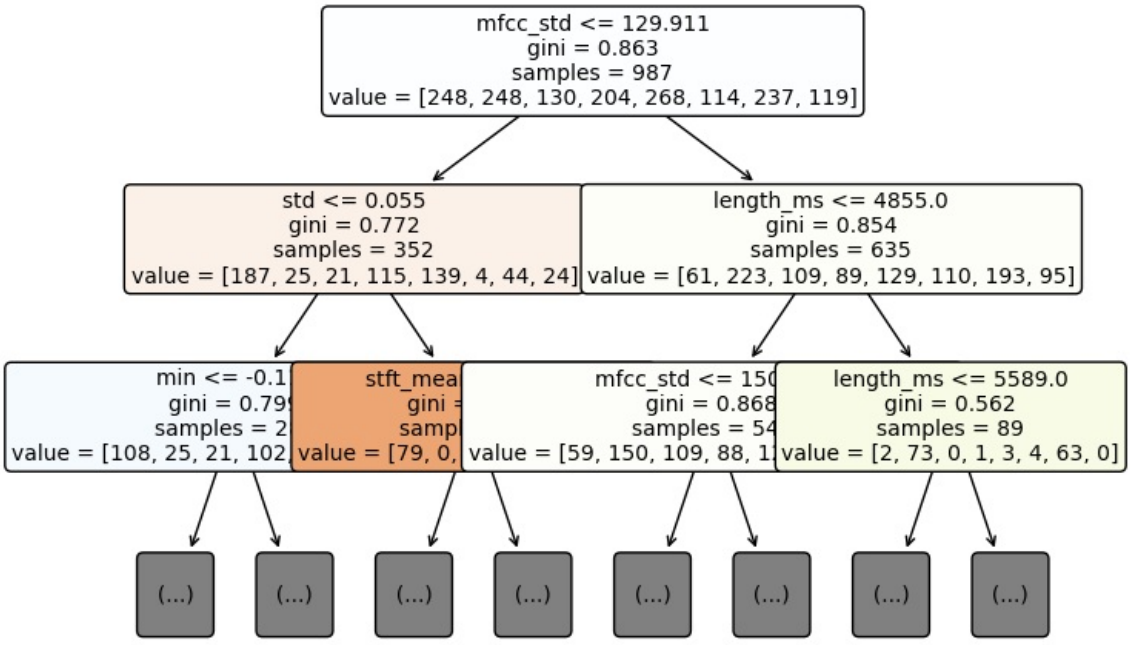


Vediamo adesso quanti sono i DecisionTrees usati per creare la RandomForest (100) e ne visualizziamo due (il primo e il secondo)

In [317…
```
len(clf.estimators_)
```

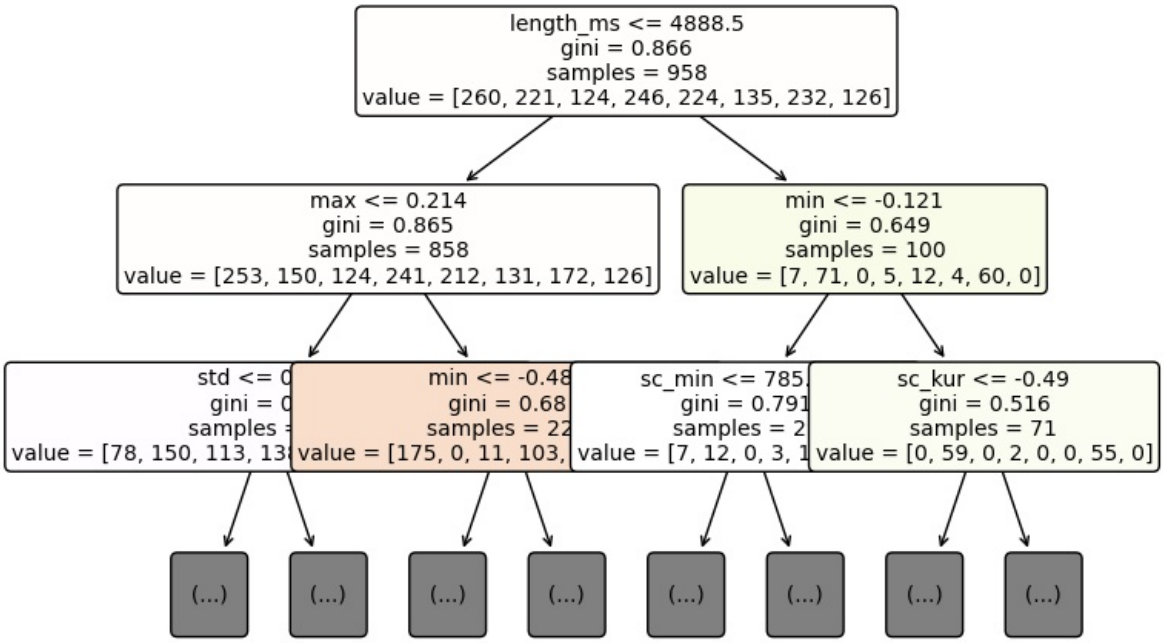
Out[317… 100

In [318…
```
plt.figure(figsize=(8, 6))
plot_tree(clf.estimators_[0],
          feature_names=columns,
          filled=True,
          rounded=True,
          fontsize=10,
          max_depth=2
          )
plt.show()
```



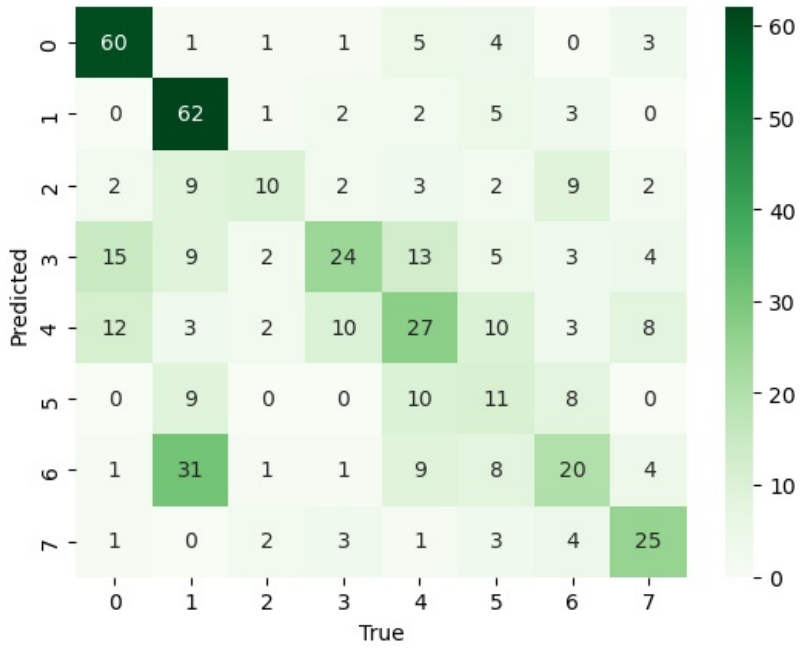
In [319…
```
plt.figure(figsize=(8, 6))
plot_tree(clf.estimators_[1],
          feature_names=columns,
          #class_names=clf.classes_,
          filled=True,
```

```
          rounded=True,
          fontsize=10,
          max_depth=2
          )
plt.show()
```



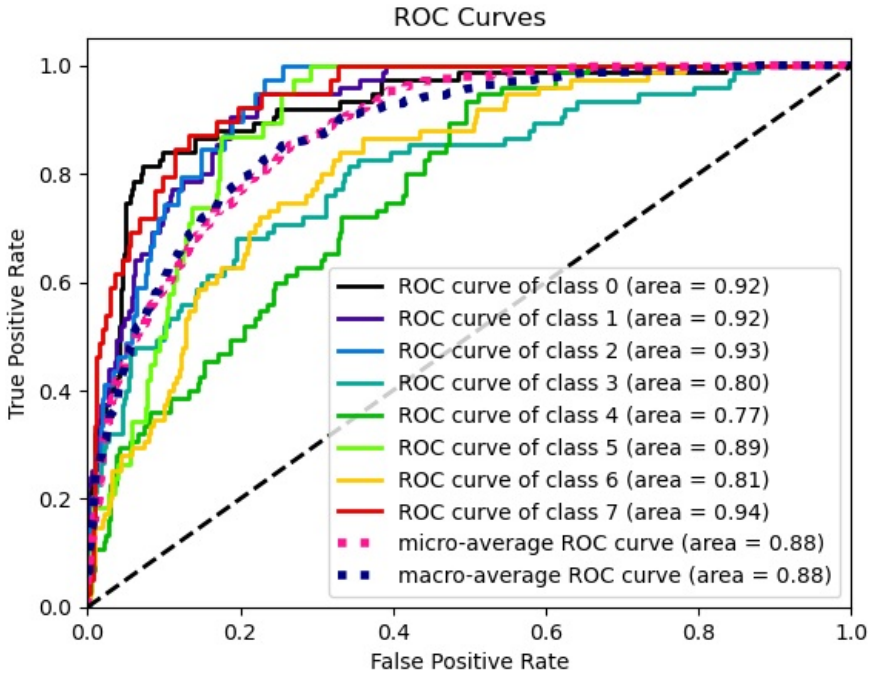
Nelle celle sotto abbiamo stampato la confusion matrix e plottato la ROC curve

In [321…
```
import seaborn as sns
cf = confusion_matrix(y_test, y_pred)
sns.heatmap(cf, annot=True, cmap="Greens")
plt.xlabel("True")
plt.ylabel("Predicted")
plt.show()
```



In [322…
```
import scikitplot
y_pred_proba = clf.predict_proba(X_test)
scikitplot.metrics.plot_roc(y_test,y_pred_proba)
```

Out[322… `<Axes: title={'center': 'ROC Curves'}, xlabel='False Positive Rate', ylabel='True Positive Rate'>`



Random Forest con class_weight

La variabile target *emotion* è sbilaciata nel nostro dataset nel senso che i campioni dove la emotion è surprised, disgust e neutral sono sotto rappresentati:

Emotion	Samples
fearful	376
angry	376
happy	376
calm	376
sad	376
surprised	192
disgust	192
neutral	188

0=angry,1=calm,2=disgust,3=Fearful,4=happy,5=Neutral,6=sad,7=surprised.

Questo potrebbe portare il modello predittivo a essere sbilanciato verso la previsione delle classi più frequenti, potenzialmente ignorando le classi minoritarie. Ciò può causare scarse prestazioni, specialmente in termini della capacità del modello di identificare correttamente le istanze delle classi meno rappresentate. Mentre nel caso di surprised (7) il problema non sembra sussistere usando il Random Forest, le altre due classi minoritarie (disgust e neutral) sono effettivamente quelle dove il Random Forest ha la precisione più bassa. Per affrontare questo problema, aggiungiamo quindi il parametro **class_weight** alla Randomized Search lo settiamo a *balanced* che imposta il peso di ogni classe in maniera inversamente proporzionale alla frequenza di quella classe.

In [338…
```
from sklearn.metrics import accuracy_score
from sklearn.model_selection import RandomizedSearchCV
```

In [340…
```
clf = RandomForestClassifier (criterion='gini',
    max_depth=11,
    min_samples_split=11,
    min_samples_leaf=3,
    ccp_alpha=0.0,
    random_state=0
)

param_dict = {
    'max_depth': np.arange(1, 20+1, 1).tolist(),
    'min_samples_split': np.arange(2, 50+1, 1),
```

```
    'min_samples_leaf': np.arange(1, 50+1, 1),
    'ccp_alpha': np.arange(0.0,1, 0.1),
    'class_weight':['balanced']
}

random = RandomizedSearchCV(clf, param_dict, cv=5, scoring='accuracy', refit=True, n_iter=500, random_state=0)
random.fit(X_train_val, y_train_val)
random.best_params_

best_params = random.best_params_

clf = random.best_estimator_

clf.fit(X_train, y_train)
```

Out[340…

RandomForestClassifier

```
RandomForestClassifier(class_weight='balanced', max_depth=16,
                       min_samples_leaf=5, min_samples_split=6, random_state=0)
```

Applicando il parametro class_weight al nostro classificatore Random Forest, abbiamo osservato un'accuratezza complessiva del 0.49, un risultato in linea con le prestazioni del modello precedente, che non teneva conto del peso delle classi. Tuttavia, l'aspetto più rilevante di questa implementazione si nota nell'analisi dettagliata della precisione relativa alle singole classi di emozioni. Abbiamo registrato un netto miglioramento nella capacità del modello di identificare con precisione le emozioni 'neutral','disgust', e 'surprised'. Queste classi, tra le meno rappresentate nel nostro dataset, hanno beneficiato in modo evidente dell'adozione di class_weight, che ha equilibrato efficacemente la loro rappresentazione nell'addestramento del modello. Questo approccio ha dimostrato la sua validità non solo nel migliorare la precisione di classi tradizionalmente problematiche o sotto-rappresentate ma ha anche confermato la capacità del parametro di ottimizzare le prestazioni complessive del modello, rendendolo più accurato e bilanciato nel riconoscimento delle diverse emozioni.

In [342…

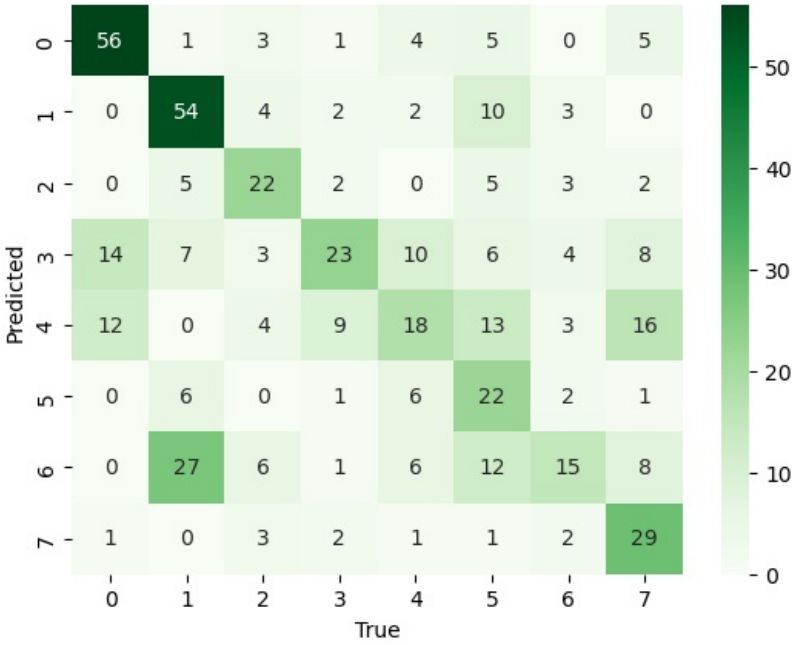
```
from sklearn.metrics import classification_report
y_pred = clf.predict(X_test)
y_pred_trainval = clf.predict(X_train_val)

accuracy = accuracy_score(y_test, y_pred)
print(accuracy)

print(classification_report(y_pred, y_test))
print(classification_report(y_pred_trainval, y_train_val))
```

```
0.48676171079429736
              precision    recall  f1-score   support

           0       0.75      0.67      0.71        83
           1       0.72      0.54      0.62       100
           2       0.56      0.49      0.52        45
           3       0.31      0.56      0.40        41
           4       0.24      0.38      0.30        47
           5       0.58      0.30      0.39        74
           6       0.20      0.47      0.28        32
           7       0.74      0.42      0.54        69

    accuracy                           0.49       491
   macro avg       0.51      0.48      0.47       491
weighted avg       0.58      0.49      0.51       491

              precision    recall  f1-score   support

           0       0.83      0.88      0.85       285
           1       0.89      0.84      0.87       318
           2       0.90      0.75      0.82       182
           3       0.75      0.86      0.80       263
           4       0.71      0.82      0.76       261
           5       0.93      0.65      0.76       214
           6       0.68      0.89      0.77       232
           7       0.88      0.65      0.75       206

    accuracy                           0.80      1961
   macro avg       0.82      0.79      0.80      1961
weighted avg       0.82      0.80      0.80      1961
```

Nelle celle seguenti abbiamo disegnato la confusion matrix e plottato la ROC curve per questo modello predittivo.

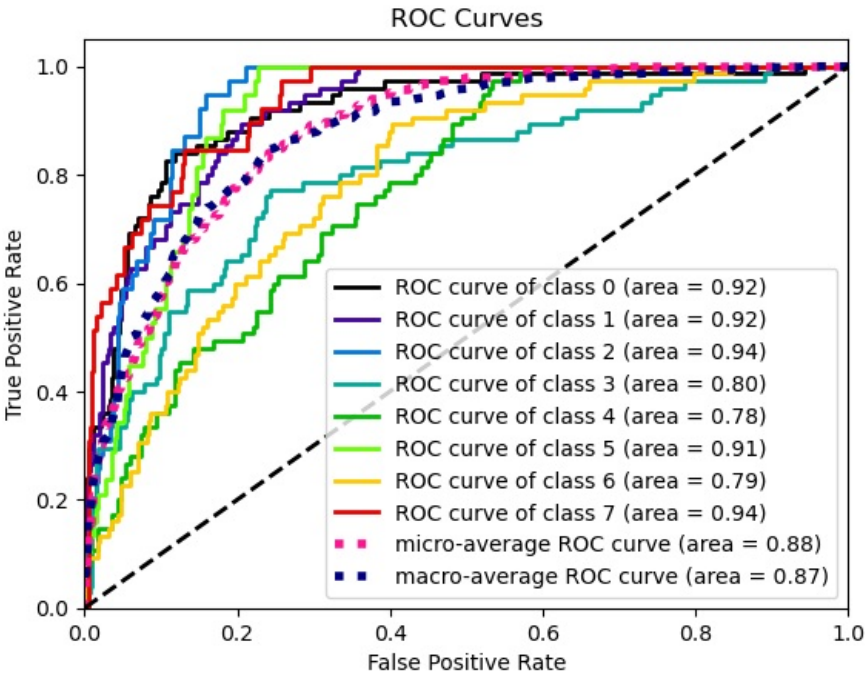
In [344…

```
import seaborn as sns
cf = confusion_matrix(y_test, y_pred)
sns.heatmap(cf, annot=True, cmap="Greens")
plt.xlabel("True")
plt.ylabel("Predicted")
plt.show()
```



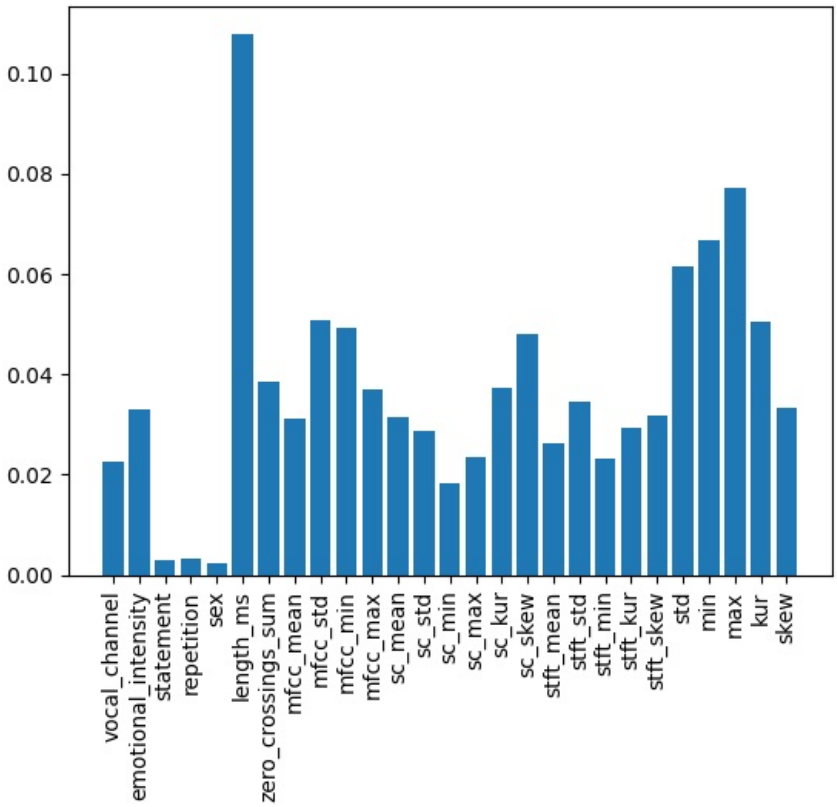
In [345…
```
import scikitplot
y_pred_proba = clf.predict_proba(X_test)
scikitplot.metrics.plot_roc(y_test,y_pred_proba)
```

Out[345… `<Axes: title={'center': 'ROC Curves'}, xlabel='False Positive Rate', ylabel='True Positive Rate'>`



Determiniamo le Feature Importance del **Random Forest**, in cui le più influenti risultano essere ancora *min,max,length_ms* anche se per questo modello length_ms diventa la feature più importante. Questo risultato suggerisce che probabilmente length_ms riveste un'importanza particolare nella predizione delle classi meno rappresentate, il cui equilibrio è stato ottimizzato attraverso l'applicazione del parametro class_weight.

In [347…
```
plt.bar(columns, clf.feature_importances_)
plt.xticks(rotation=90)
plt.show()
```

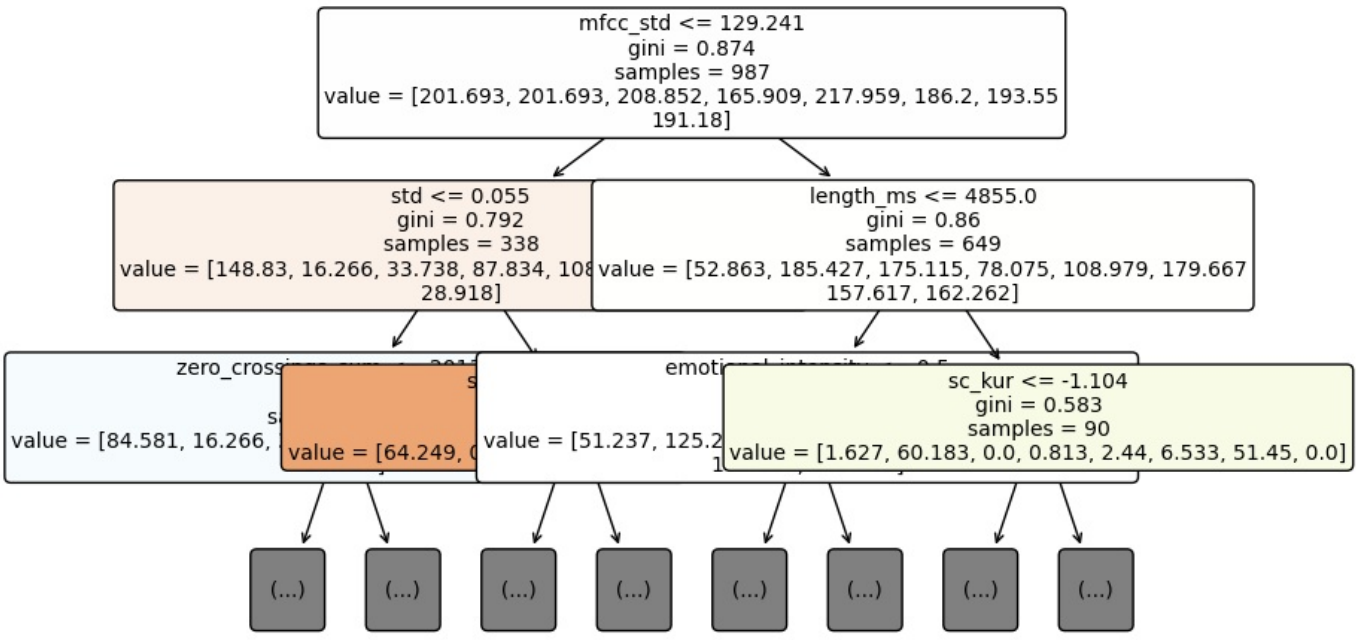


Vediamo adesso quanti DecisionTrees sono stati usati per creare la RandomForest (100) e ne visualizziamo due (il primo e il secondo)

In [349…
```
len(clf.estimators_)
```

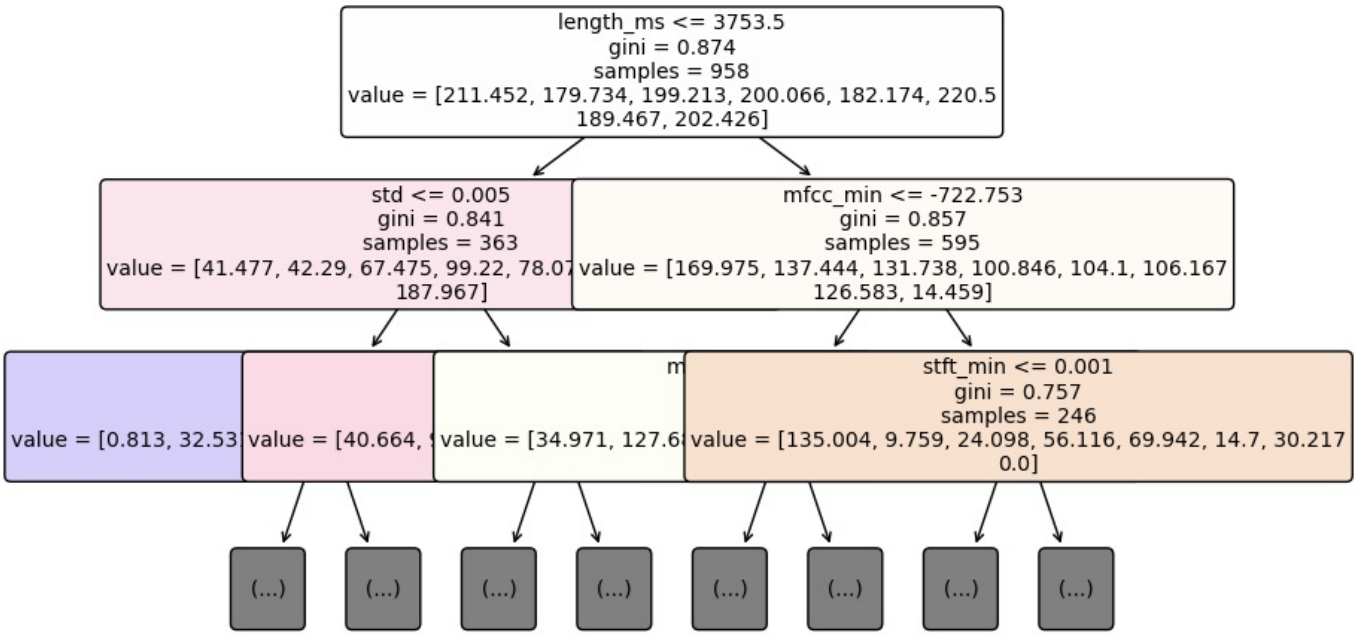
Out[349… 100

In [350…
```
plt.figure(figsize=(8, 6))
plot_tree(clf.estimators_[0],
          feature_names=columns,
          filled=True,
          rounded=True,
          fontsize=10,
          max_depth=2
          )
plt.show()
```



In [351…
```
plt.figure(figsize=(8, 6))
plot_tree(clf.estimators_[1],
          feature_names=columns,
          filled=True,
          rounded=True,
          fontsize=10,
```

```
          max_depth=2
          )
plt.show()
```



SVM

Procediamo ora ad utilizzare la Support Vector Machine (SVM) sul nostro dataset. La SVM è un metodo predittivo che cerca un iperpiano per separare i dati nello spazio delle feature. Per ridurre il carico computazionale, abbiamo deciso di ridurre la dimensionalità del Training Set.

Importiamo sia LinearSVC, che effettua una separazione lineare, sia SVC, che consente l'utilizzo del kernel trick per gestire dati non linearmente separabili.

In [262…
```
from sklearn.model_selection import train_test_split
from sklearn.svm import LinearSVC, SVC
from sklearn.metrics import accuracy_score
from sklearn.model_selection import RandomizedSearchCV
```

In [264…
```
target = 'Emotion_val'
columns = [c for c in dfe_noobj.columns if c not in ['Emotion_val','sex','emotional_intensity','statement','rep
```

In [266…
```
X = dfe_noobj[columns].values
y = dfe_noobj[target].values
```

In [268…
```
X_train_val, X_test, y_train_val, y_test = train_test_split(X, y, test_size=0.2, stratify=y, random_state=0)
X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val, test_size=0.2, stratify=y_train_val
```

In [284…
```
lin_svm = LinearSVC(C=1,max_iter=5000, dual=False)
lin_svm.fit(X_train_val, y_train_val)
y_pred = lin_svm.predict(X_test)
accuracy_score(y_test, y_pred)
```

Out[284… `0.37067209775967414`

In [294…
```
svm = SVC(C=100, kernel='poly')
svm.fit(X_train_val, y_train_val)
y_pred = svm.predict(X_test)
accuracy_score(y_test, y_pred)
```

Out[294… `0.3258655804480652`

Possiamo valutare la tipologia di Kernel e il parametro C attraverso l'**HYPERPARAMETER TUNING** durante la fase di validazione con una RandomizedSearch. Rispetto a un tentativo iniziale abbiamo ristretto il range dei parametri e il numero di iterazioni per ridurre il carico computazionale. Abbiamo anche settato il parametro n_jobs=-1 che permette di fare più calcoli in parallelo su più core potenzialmente aumentando la velocità di calcolo.

Purtroppo, nonostante i tentativi per ridurre i parametri, il numero di iterazioni e la dimensionalità, il sistema non è stato in grado di completare i calcoli entro un tempo accettabile, e abbiamo deciso di interompere l'esecuzione. Non è chiaro se sussiste un problema di velocità computazionale dei calcoli oppure se una delle combinazioni dei vari parametri crei un problema al sistema.

In [ ]:
```
param_dict = {
    'C': [0.1, 10, 100],
    'kernel': ['linear','poly','rbf'],
    'degree': [1, 2, 4, 5],
    'gamma': ['scale', 'auto'],
    'coef0': [0.0, 1, 10],
}

clfSCV = SVC()

rands = RandomizedSearchCV(clf, param_dict, cv=5, scoring='accuracy', refit=True, n_iter=50, random_state=0,n_j

rands.fit(X_train_val, y_train_val)

print("Best parameters:", rands.best_params_)

clf = rands.best_estimator_

clf.fit(X_train_val, y_train_val)

y_pred = clf.predict(X_test)

accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy:.4f}')
```

Classification - Discussion and Conclusions

In questa sezione abbiamo esplorato l'utilizzo di alcuni modelli di classificazione per prevedere due specifiche variabili del nostro dataset: "Sex" ed "Emotion". Ogni modello ha offerto spunti di riflessione e in particolare sulla performance in termini di accuratezza e precisione.

Nel caso della variabile target Sex, sia il Decision Tree che il KNN hanno dato ottimi risultati con accuratezze intorno al 90%. Nonostante questa similarità, il KNN sembra essere migliore del Decision Tree non solo perché la sua accuratezza è leggermente più alta, ma anche perché la precision sulle due classi è meno sbilanciata. Questo può essere attribuito alle sue caratteristiche intrinseche. Il KNN, essendo un algoritmo basato sulla distanza, tende a essere più flessibile e meno incline all'overfitting rispetto ai Decision Trees. Inoltre, la capacità del KNN di mantenere una precisione equilibrata tra le classi suggerisce che gestisce meglio il problema dello sbilanciamento delle classi senza necessità di tecniche aggiuntive.

Per quel che riguarda la classe Emotion, il modello che si è dimostrato migliore è stato il Random Forest. Questo non è sorprendente visto che è un modello ensemble che combina le previsioni di diversi alberi decisionali e quindi geralmente più robusto e accurato. Andando invece ad analizzare la precisione dei vari modelli per le singole emozioni, notiamo che questa varia in maniera evidente in tutti i modelli esaminati, con le emozioni come *angry* e *calm* che consistentemente mostrano una precision più alta per tutti i modelli esaminati e altre come *suprised*, *sad*, *disgust* e *neutral* con precision tendenzialmente più basse (anche se queste variano al variare del modello). La conclusione che possiamo trarre è quella che anche se il modello di Random Forest è quello migliore, questo non vale per tutte le classi della variabile target. Per ovviare a questo, abbiamo provato a introdurre il parametro class_weight durante l'addestramento del Random Forest e dato un peso alle varie classi inversamente proporziale alla loro rapprensentazione nel dataset. La precision per alcune classi è migliorata anche se per altre è peggiorata, così come è peggiorata l'accuracy generale del modello.

Se volessimo prevedere una classe specifica di Emotion, potremmo usare le seguenti strategie:

1. Scegliere il metodo con la migliore precisione per la classe d'interesse (o altre metriche di valutazione a seconda di quello su cui vogliamo focalizzarci).
2. Lavorare sulla class_weight delle classi durante la fase di allenamento del modello
3. Identificare e utilizzare le feature che esercitano maggiore influenza sulla predizione della classe di interesse, ottimizzando così le prestazioni del modello su quella specifica categoria.
4. Validazione incrociata e tuning degli iperparametri focalizzandoci su una classe specifica

In conclusione, la scelta del modello migliore dipende dall'obiettivo specifico, dalla natura dei dati e dalla distribuzione delle classi, richiedendo una valutazione per bilanciare accuratezza generale e distribuzione tra le classi.

In [ ]:
```
```