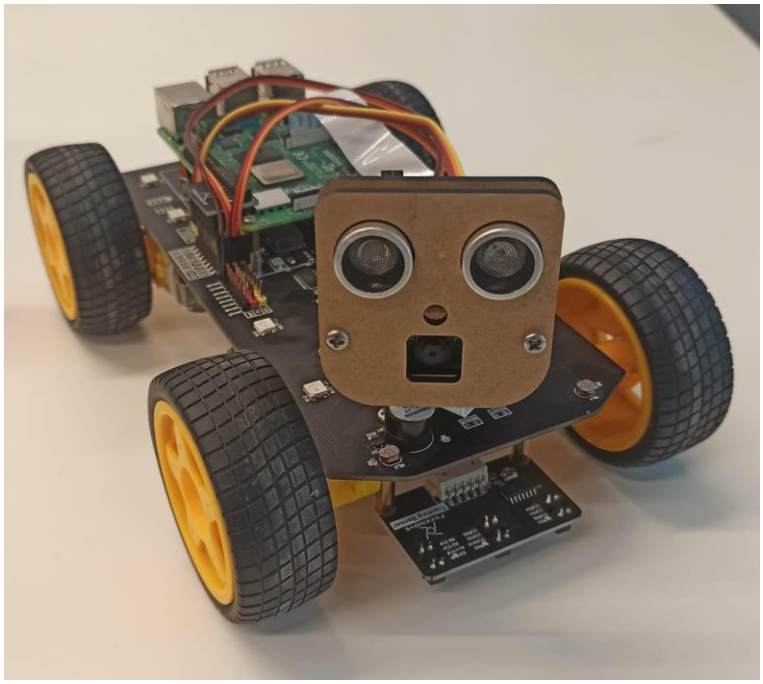
School of Mechanical Engineering
בית-הספר להנדסה מכנית

Iby and Aladar Fleischman
Faculty of Engineering
Tel Aviv University

הפקולטה להנדסה
ע"ש איבי ואלדר פליישמן
אוניברסיטת תל-אביב

חישה ותנועה בהשראת הטבע

שליטה ובקרה ברובוט על ידי זיהוי ריחות



איתמר משעני - 311337604
אלון מזרחי - 312284706
תאריך : 16.05.2022

1. מבוא

כחלק מפרויקט של הכנת חיישן לסיווג ריחות שונים המבוסס על תכונות החגב, עלה הצורך לבנות מודל קלסיפיקציה אשר מקבל קלט מידע, ומסווג את הריח המתאים לו. החיישן בנוי מאנטנה של חגב אשר מחוברת לאלקטרודות בשתי קצותיה כך שהאלקטרודות מודדות שינוי במתח לאורך זמן כאשר על האנטנה נמצאים רצפטורים אשר רגישים לריח ולמגע. למעשה, הקלט למודל הקלסיפיקציה הינו וקטור מתחים לאורך זמן. על מנת להמחיש את יכולות ושימוש המודל, בוצע שימוש ברובוט קטן עליו נמצא המודל המאומן, כך שכאשר הרובוט מקבל נתונים חדשים הוא חוזה באמצעות המודל את סוג הריח ומבצע פעולות שונות בהתאם (נסיעה ישר, שמאלה, ימינה או נשאר במקום).

2. מטרת הפרויקט

הרובוט יבדיל בין 3 ריחות שונים וינוע בהתאם לכל ריח. עבור ריח הדרים ינוע ישר, עבור ריח רוזמרין ינוע שמאלה ועבור ריח מרציפן ינוע ימינה. כאשר אין התאמה עבור אחד משלושת הריחות הרובוט ישאר במקום.

3. מהלך הפרויקט

2.1. עיבוד נתונים:

תחילה, מכיוון שהרצפטורים על האנטנה רגישים לסביבה ומושפעים ממגע יש לבצע תהליך "כיול" עבור החיישן. למעשה, מבצעים מדידה כאשר אין שום ריח שמופרש בסביבת החיישן כך שמדידה זו משמשת כמדידה המתארת את ההשפעות הסביבתיות. כעת, מדידת כיול זו מוחסרת בעת ביצוע שאר המדידות. סט הנתונים שהתקבל לאימון המודל מכיל 361 הקלטות ריח, (בשיטת EAG Response), אשר כל הקלטה מכילה וקטור באורך 150 דגימות מתח (שינוי מתח לאורך זמן) ועמודה אחרונה אשר משמת כתיוג ולמעשה מקשרת את הריח אל הוקטור המתאים (שם הריח). המתחים שהתקבלו נפרשו על תחום רחב, ולכן בוצע נירמול לנתונים כך שכלל ערכי הדגימות יפרשו בין 0 ל-1. באיור (1) ניתן לראות דגימה עבור כל אחד מהריחות. בנוסף שמות הריחות הומרו למספרים כך שריח הדרים סומן כ- '2' , ריח מרציפן כ- '1' וריח רוזמרין כ- '0'. ובהמשך עברו one hot encoding.

נתונים אלו חולקו לשני מערכים, מערך פיצ'רים (מכיל את הוקטורים המנורמלים) ומערך תיוגים (מכיל את הערכים הרצויים ביציאה מהמודל).

איור (1) - EAG Response עבור כל אחד מהריחות.

2.2. בנייה ואימון אלגוריתם למידת מכונה (אלגוריתם קלסיפיקציה):

לצורך בחירת מודל קלסיפיקציה מתאים נבחנו 6 מודלים מסוגים שונים:

`KNeighborsClassifier, GussianNaiveBayes, LogisticRegression, SupportVectorClassification, AdaBoostClassifier, RandomForestClassifier.`

כאשר על כל מודל בוצעו הפעולות הבאות:

1. על מנת למצוא את ההיפר-פרמטרים האופטימליים עבור כל מודל, בוצע חיפוש באמצעות `GridSearchCV` ומכיוון שגודל סט האימון קטן בוצע שימוש ב `KFold-CrossValidation` כאשר פונקציית המחיר שבאמצעותה מוערכת השגיאה של כל מודל הינה auc score ovr (Area Under Curve One VS Rest)
2. בחינת המודלים השונים ע"י גרפי ROC ו - Confusion Matrix.
3. אימון המודל הנבחר עם כל סט האימון.(`fit(x_train,y_train)`)

2.3. בדיקת האלגוריתם על ידי סט בדיקה.

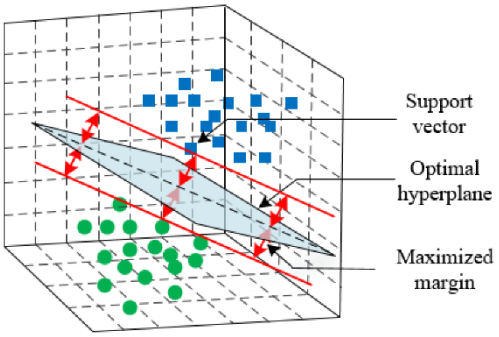
לאחר אימון המודל, בוצעה בחינה על סט הבדיקה כאשר סט הבדיקה הינו סט נפרד שהתקבל המכיל 15 דגימות. לבסוף נבחר המודל שהניב את התוצאות המיטביות.

2.4. כתיבת קוד אשר אחראי לפעולת הרובוט וממיר מזיהוי ריח לפקודה למנועי הרובוט. בסיום לאחר שנבחר המודל המיטבי, נכתב קוד אשר טוען את המודל המיטבי, מקבל קובץ בדיקה ועובר על כלל השורות אחת אחרי השנייה כאשר עבור כל שורה נבדק הריח, התוצאה המתקבלת יכולה להיות כל אחד מהריחות או כמה ריחות יחדיו אשר מעידות על אי זיהוי ריח. לפי התוצאה שמתקבלת נשלחת פקודה אל מנועי הרובוט אשר מניעה את הרובוט כנדרש.

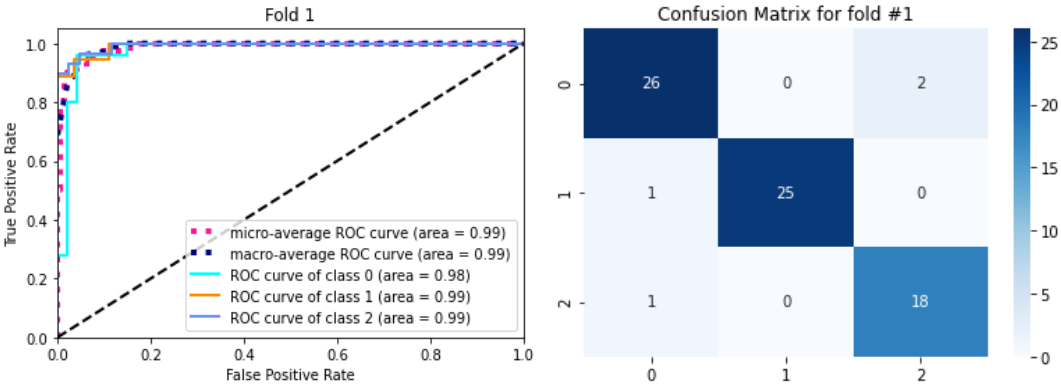
4. תוצאות

כלל המודלים הניבו תוצאות טובות על סט האימון כאשר ציון ה- AUC שלהם היה מעל 94 אחוזים. לכן על ידי בחינת עקומות roc auc וה- confusion matrix שהתקבלו[1], נבחר מודל ה-SVC (Support Vector Classifier) אשר תוצאותיו על סט האימון הינם Auc Score 99.15%, וה-Accuracy על סט הבדיקה הינו 92.86%.

אלגוריתם ה-SVC פועל כך שהוא יוצר מישורי הפרדה (לא בהכרח לינארים) בין קבוצות התיוגים השונות. בנוסף על ידי אימון על בסיס שיטה אחד מול השאר (One Vs Rest), המודל מאמן כל תיוג בנפרד אל מול כל שאר התיוגים. לבסוף המודל קובע סף מסוים, (אחוז מסוים), לכל תיוג. מעל סף זה המודל מחליט כי וקטור הכניסה מתאים לאותו תיוג. לפיכך כל וקטור כניסה יכול להתאים לכמה תיוגים. במערכת שלנו, התאמה ליותר מתיוג אחד או אי התאמה לכלל התיוגים משמעותה שאין זיהוי ריח. (נציין שאין ריחות מעורבבים בנתונים שקיבלנו).



איור (2) – המחשת עבודת Support Vector Classifier.



איור (3) – מימין דוגמת פלט confusion matrix, ומשמאל גרף ROC.

[1] כלל התוצאות, עקומות roc auc ו- confusion matrix מצורפות בנספח בסוף הדוח.

5. סיכום ומסקנות

בעזרת נתונים שנאספו על ידי חיישן ריחות בוצע מודל קלסיפיקציה אשר בעזרתו מתבצע זיהוי ריחות והנעת רובוט. התקבל מודל אשר לטעמנו, עונה על הציפיות עם דיוק גבוה מ-90 אחוזים. לפיכך, ניתן להסיק שבאמצעות מודלים פשוטים של למידת מכונה ניתן להבדיל בין עקומות EAG Response בצורה טובה.

במהלך הפרויקט נחשפנו לנושא מעניין אשר עוסק בפיתוח חיישן ריחות המבצע זיהוי ריחות על ידי עקומת תגובת אלקטרואנטנוגרם (EAG Response) המתבסס על חוש הריח אצל חגבים. בנוסף התנסנו בכלים רחבים בתחום למידת מכונה, ולבסוף התנסנו בהנעת רובוט באמצעות בקר Raspberry Pi.

6. מקורות

- Electroantenogram
- Support-vector machine
- Kfold - CrossValidation
- One vs One for classifier

7. נספחים

- קוד פיתון בפורמט 'Jupyter NoteBook'.

Bio-Inpired Sensing Movement

Final Project - Odor Identity Controls Robot Movements

Alon Mizrahi and Itamar Mishani

Importing packages:

In [1]:
```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
import scipy.stats as st
import seaborn as sns
from pandas.plotting import scatter_matrix
pd.options.mode.chained_assignment = None  # default='warn'
import time
from itertools import cycle

# Scikit-learn plots:
from sklearn.model_selection import train_test_split, learning_curve, KFold, Shuffle
from sklearn.svm import SVC
from sklearn.neural_network import MLPClassifier, MLPRegressor
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import roc_curve, auc, accuracy_score, plot_confusion_matrix, c
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import MinMaxScaler, StandardScaler, label_binarize
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier
from sklearn.metrics import plot_confusion_matrix
from sklearn.multiclass import OneVsRestClassifier

# pandas configuration
pd.set_option('display.max_columns', 999)
```

Reading data files:

In [2]:
```
data = pd.read_csv('./data/3_odors_bio_inspired_project.csv')
test = pd.read_excel('./data/Test_odors_and_control.xlsx')
```

Data exploration and preprocess:

In [3]:
```
train = data.iloc[:, :-4]
labels = data['label']
labels
```

Out[3]:
```
0      4-Rose
1      4-Rose
2      4-Rose
3      4-Rose
4      4-Rose
        ...
356    1-Benz
```

```
357    1-Benz
358    1-Benz
359    1-Benz
360    1-Benz
Name: label, Length: 361, dtype: object
```

Scaling:

In [4]:

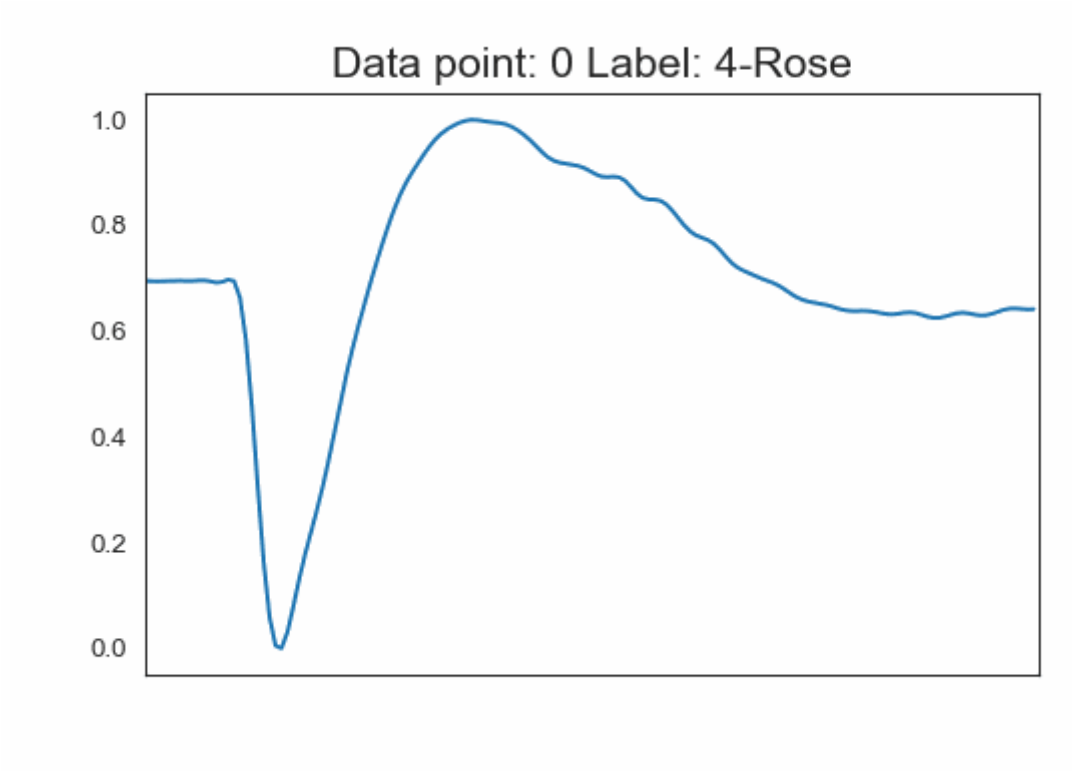
```
for index, row in train.iterrows():
    scaler = MinMaxScaler()
    scaler.fit(train.iloc[index,:].values.reshape(-1, 1))
    train.iloc[index,:] = scaler.transform(train.iloc[index,:].values.reshape(-1, 1)
```

In [5]:

```
# sns.set_style('white')
# for i in range(train.shape[0]):
#     plt.plot(train.iloc[i,:])
#     plt.tick_params(
#         axis='x',
#         which='both',
#         bottom=False,
#         top=False,
#         labelbottom=False)
#     plt.xlim(0, train.shape[1])
#     plt.title(f'Data point: {i} Label: {labels[i]}', fontsize=16)
#     filename= 'togif\Volcano_step'+str(i)+'.png'
#     plt.savefig(filename, dpi=96)
#     plt.gca()
#     plt.close()
```

In [6]:

```
# import glob
# from PIL import Image

# # Create the frames
# frames = []
# imgs = glob.glob("togif\*.png")
# for i in imgs:
#     new_frame = Image.open(i)
#     frames.append(new_frame)

#Save into a GIF file that loops forever

# frames[0].save('png_to_gif.gif', format='GIF',
#                append_images=frames[1:],
#                save_all=True,
#                duration=300, loop=0)
```



In [7]:
```
train
```

Out[7]:

	t0	t1	t2	t3	t4	t5	t6	t7	t8	
0	0.695099	0.694518	0.694297	0.694657	0.695378	0.695738	0.695494	0.695006	0.695134	0.695
1	0.679059	0.679423	0.679605	0.679540	0.679566	0.679969	0.680632	0.681140	0.681361	0.681
2	0.667398	0.667455	0.667398	0.667197	0.667011	0.667197	0.668100	0.669648	0.671483	0.671
3	0.618281	0.619193	0.619471	0.619034	0.618677	0.619431	0.620740	0.621335	0.619510	0.615
4	0.609366	0.611563	0.612731	0.612498	0.611236	0.610161	0.610161	0.611563	0.613339	0.614
...	...	...	...	...	...	...	...	...	...	
356	0.595104	0.591162	0.589114	0.590123	0.593729	0.597763	0.599749	0.599016	0.597152	0.595
357	0.622123	0.618815	0.618181	0.619584	0.622123	0.624929	0.627201	0.629406	0.631878	0.633
358	0.545254	0.543265	0.544464	0.547400	0.550084	0.551504	0.552136	0.554409	0.559302	0.565
359	0.707869	0.710065	0.713345	0.717280	0.721359	0.725266	0.728261	0.729715	0.727975	0.722
360	0.701064	0.704333	0.709385	0.713275	0.714220	0.713059	0.712113	0.712734	0.712059	0.706

361 rows × 151 columns

Number of unique labels:

In [8]:
```
labels.nunique()
```

Out[8]: 3

Tranforming each label from string to integer:

In [9]:

```
lab = []
for elem in labels:
    if elem == '4-Rose':
        lab.append(0)
    elif elem == '1-Benz':
        lab.append(1)
    elif elem == '6-Ger':
        lab.append(2)
    else:
        lab.append(None)
labels = pd.DataFrame(lab, columns=['Label'])
```

In [10]:

```
labels.head()
```

Out[10]:

	Label
0	0
1	0
2	0
3	0
4	0

In [11]:

```
y = labels.Label.values
X = train.values

X.shape, y.shape
```

Out[11]:

```
((361, 151), (361,))
```

In [12]:

```
models_running_time = []
```

There are many options for score matrics we can use. In our case, we used ROC AUC OVR metric (Reciever Operating Charactaristic, Area Under Curve, One vs Rest repectively)

In [13]:

```
from sklearn.metrics import SCORERS
sorted(SCORERS.keys())
```

Out[13]:

```
['accuracy',
 'adjusted_mutual_info_score',
 'adjusted_rand_score',
 'average_precision',
 'balanced_accuracy',
 'completeness_score',
 'explained_variance',
 'f1',
 'f1_macro',
 'f1_micro',
 'f1_samples',
 'f1_weighted',
 'fowlkes_mallows_score',
 'homogeneity_score',
 'jaccard',
```

```
'jaccard_macro',
'jaccard_micro',
'jaccard_samples',
'jaccard_weighted',
'max_error',
'mutual_info_score',
'neg_brier_score',
'neg_log_loss',
'neg_mean_absolute_error',
'neg_mean_absolute_percentage_error',
'neg_mean_gamma_deviance',
'neg_mean_poisson_deviance',
'neg_mean_squared_error',
'neg_mean_squared_log_error',
'neg_median_absolute_error',
'neg_root_mean_squared_error',
'normalized_mutual_info_score',
'precision',
'precision_macro',
'precision_micro',
'precision_samples',
'precision_weighted',
'r2',
'rand_score',
'recall',
'recall_macro',
'recall_micro',
'recall_samples',
'recall_weighted',
'roc_auc',
'roc_auc_ovo',
'roc_auc_ovo_weighted',
'roc_auc_ovr',
'roc_auc_ovr_weighted',
'top_k_accuracy',
'v_measure_score']
```

We now use gridsearch in order to find the best hyperparameters for all evaluated calssification models:

K-Nearest Neighbors:

In [14]:

```
start = time.time()
KNN_options = {'n_neighbors' : [10, 50, 100],
          'weights' : [ 'uniform', 'distance']
              }
# Setup classifier, and find using GridsearchCV the best hyper-parameters
kf = KFold(n_splits=5, shuffle = True, random_state=100)
KNN_best = GridSearchCV(KNeighborsClassifier() ,KNN_options, cv=kf, scoring='roc_auc
KNN_best.fit(X, y) #X_train, y_train

print ('KNN chosen parameters (recieved best AUC): {}'.format(KNN_best.best_params_)
print ("KNN AUC score with the chosen parameters: ", KNN_best.best_score_)
total_time = (time.time()-start)/60
print("Running time: %s minutes" % (round(total_time, 2)))
models_running_time.append(total_time)
```

```
KNN chosen parameters (recieved best AUC): {'n_neighbors': 10, 'weights': 'distanc
e'}
KNN AUC score with the chosen parameters:  0.9870097860782712
Running time: 0.08 minutes
```

In [15]:
```
neigh = KNeighborsClassifier(**KNN_best.best_params_)
```

Gaussian Naive Bayes:

In [16]:
```
start = time.time()
GNB_options = {'priors' : [None],
               'var_smoothing' : [ 1e-9, 1e-7, 1e-5, 1e-3, 0.1, 1, 10]
              }

# Setup classifier, and find using GridsearchCV the best hyper-parameters
skf = KFold(n_splits=5, shuffle = True, random_state=100)
GNB_best = GridSearchCV(GaussianNB(), GNB_options, cv=skf, scoring='roc_auc_ovr_weig
GNB_best.fit(X, y)

print ('GNB chosen parameters (recieved best AUC): {}'.format(GNB_best.best_params_)
print ("GNB AUC score with the chosen parameters: ", GNB_best.best_score_)
total_time = (time.time()-start)/60
print("Running time: %s minutes" % (round(total_time,2)))
models_running_time.append(total_time)
```

```
GNB chosen parameters (recieved best AUC): {'priors': None, 'var_smoothing': 0.1}
GNB AUC score with the chosen parameters:  0.9495980810563289
Running time: 0.0 minutes
```

In [17]:
```
GNB = GaussianNB(**GNB_best.best_params_)
```

Logistic Regression:

In [18]:
```
start = time.time()
LogisticRegression_options = {'penalty' : ['l1', 'l2'],
                              'C' : [ 0.001, 0.01, 0.1, 0.5, 1, 10, 100, 1000],
                              'tol' : [ 0.001, 1e-5 ],
                              'max_iter' : [100, 2000],
                              'solver' : ["liblinear"]}

kf = KFold(n_splits = 5, shuffle = True, random_state=100)
LR_best = GridSearchCV(LogisticRegression(), LogisticRegression_options, cv=kf, scor
LR_best.fit(X, y) #X_train, y_train

print ("Logistic Regresion best parameters: {}".format(LR_best.best_params_))
print ("Logistic Regresion AUC score with the chosen parameters: ", LR_best.best_sco
total_time = (time.time()-start)/60
print("Running time: %s minutes" % (round(total_time,2)))
models_running_time.append(total_time)
```

```
Logistic Regresion best parameters: {'C': 100, 'max_iter': 100, 'penalty': 'l2', 'so
lver': 'liblinear', 'tol': 1e-05}
Logistic Regresion AUC score with the chosen parameters:  0.9836368863151856
Running time: 2.42 minutes
```

In [19]:
```
LR = LogisticRegression(**LR_best.best_params_)
```

Support Vector Classification:

In [20]:
```
start = time.time()
parametersOptions = {'C':[1.,0.1],
                     'kernel': ['poly','rbf'],
                     'tol' : [1e-3, 1e-5],
```

```
                     'degree' : [1, 3, 5],
                     'probability': [True]
                    }

# Setup classifier, and find using GridsearchCV the best hyper-parameters with kfold
kfold = KFold(n_splits = 5, shuffle = True, random_state=100)
SVC_best = GridSearchCV(SVC(), parametersOptions, cv = kfold, scoring='roc_auc_ovr_w
SVC_best.fit(X, y) #X_train, y_train
print ('SVC chosen parameters (recieved best AUC): {}'.format(SVC_best.best_params_)
print ("SVC AUC score with the chosen parameters: ", SVC_best.best_score_)

total_time = (time.time()-start)/60
print("Running time: %s minutes" % (total_time))
models_running_time.append(total_time)
```

```
SVC chosen parameters (recieved best AUC): {'C': 0.1, 'degree': 5, 'kernel': 'poly',
'probability': True, 'tol': 1e-05}
SVC AUC score with the chosen parameters:  0.9915082219023091
Running time: 0.05128664175669352 minutes
```

In [21]:

```
svc = SVC(**SVC_best.best_params_)
```

Adaboost:

In [22]:

```
start = time.time()
parametersOptions = {'n_estimators':[500,1000],
                     'learning_rate': [0.01,0.1,0.3],
                    'random_state' :[100]}

# Setup classifier, and find using GridsearchCV the best hyper-parameters with kfold
kfold = KFold(n_splits = 5, shuffle = True, random_state=100)
ADB_best = GridSearchCV(AdaBoostClassifier(), parametersOptions, cv = kfold, scoring
ADB_best.fit(X, y)
print ('Adaptive Boosting chosen parameters (recieved best AUC): {}'.format(ADB_best
print ("Adaptive Boosting AUC score with the chosen parameters: ", ADB_best.best_sco


total_time = (time.time()-start)/60
print("Running time: %s minutes" % (total_time))
models_running_time.append(total_time)
```

```
Adaptive Boosting chosen parameters (recieved best AUC): {'learning_rate': 0.3, 'n_e
stimators': 1000, 'random_state': 100}
Adaptive Boosting AUC score with the chosen parameters:  0.9440826270561418
Running time: 0.8413517435391744 minutes
```

In [23]:

```
ADB = AdaBoostClassifier(**ADB_best.best_params_)
```

Random Forest:

In [24]:

```
# RFC algorithm
start = time.time()
parametersOptions = {'n_estimators':[200, 500, 1000],
                     'max_features': ['auto', 'sqrt', 'log2'],
                     'max_depth' : [4,5,6,7,8],
                     'criterion' :['gini', 'entropy']}

# Setup classifier, and find using GridsearchCV the best hyper-parameters with kfold
kfold = KFold(n_splits = 5, shuffle = True, random_state=100)
RFC_best = GridSearchCV(RandomForestClassifier(), parametersOptions, cv = kfold, sco
```

```
RFC_best.fit(X, y)
print ('RFC chosen parameters (recieved best AUC): {}'.format(RFC_best.best_params_)
print ("RFC AUC score with the chosen parameters: ", RFC_best.best_score_)
print ("RFC KFold parameters: ", kfold)

total_time = (time.time()-start)/60
print("Running time: %s minutes" % (total_time))
models_running_time.append(total_time)
```

```
RFC chosen parameters (recieved best AUC): {'criterion': 'entropy', 'max_depth': 8,
'max_features': 'log2', 'n_estimators': 1000}
RFC AUC score with the chosen parameters:  0.987644935492396
RFC KFold parameters:  KFold(n_splits=5, random_state=100, shuffle=True)
Running time: 3.0791301409403484 minutes
```

In [25]:

```
RFC = RandomForestClassifier(**RFC_best.best_params_)
RFC
```

Out[25]:

```
RandomForestClassifier(criterion='entropy', max_depth=8, max_features='log2',
                       n_estimators=1000)
```

For each Fold we will evaluate and plot both confusion matrices and ROC curves:

In [26]:

```
y = label_binarize(y, classes=[0, 1, 2])
y.shape
```

Out[26]:

```
(361, 3)
```

In [27]:

```
n_classes = y.shape[1]

n_samples, n_features = X.shape
```

In [28]:

```
def KfoldProcess(X, y, clf, k):
    """
    This function trains the model using the k-folds
    X - X_train, the data to train the model
    y - Y_train, the target data
    clf - The classifier to train
    k - Number of folds to process
    """

    # Set KFolds with a random state for consistent results
    kf = KFold(n_splits = k, shuffle = True, random_state=100)

    # we catch the tpr and fpr since we need to interpolate data
    # Validation set:
    cm = 0
    tpr_test, fpr_test, auc_test = [],[],[]
    for train_index, test_index in kf.split(X):
        #Splitting into train and validation, based on the current fold.
        X_train, X_test = X[train_index], X[test_index]
        y_train, y_test = y[train_index], y[test_index]

        # Learn to predict each class against the other
        classifier = OneVsRestClassifier(
            clf
        )
        trained = classifier.fit(X_train, y_train)#.decision_function(X_test)
```

```
        y_score = trained.predict_proba(X_test)
        # Compute ROC curve and ROC area for each class
        fpr = dict()
        tpr = dict()
        roc_auc = dict()
        for i in range(n_classes):
            fpr[i], tpr[i], _ = roc_curve(y_test[:, i], y_score[:, i])
            roc_auc[i] = auc(fpr[i], tpr[i])

        # Compute micro-average ROC curve and ROC area
        fpr["micro"], tpr["micro"], _ = roc_curve(y_test.ravel(), y_score.ravel())
        roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])

        auc_test.append(roc_auc)
        tpr_test.append(tpr)
        fpr_test.append(fpr)

        # plot confusion matrix for current fold
        sns.heatmap(confusion_matrix(y_test.argmax(axis=1), y_score.argmax(axis=1)),
        plt.title("Confusion Matrix for fold #%s"%(cm+1))
        plt.show()
        cm+=1
    plt.show()
    return [tpr_test, fpr_test, auc_test]
```

In [29]:

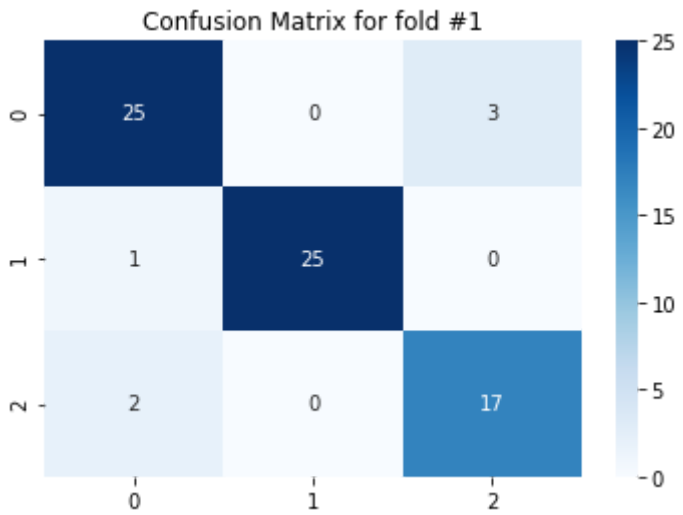
```
def plot_Kfold(tpr, fpr, roc_auc, fold, n_classes=n_classes):
    # First aggregate all false positive rates
    all_fpr = np.unique(np.concatenate([fpr[i] for i in range(n_classes)]))

    # Then interpolate all ROC curves at this points
    mean_tpr = np.zeros_like(all_fpr)
    for i in range(n_classes):
        mean_tpr += np.interp(all_fpr, fpr[i], tpr[i])

    # Finally average it and compute AUC
    mean_tpr /= n_classes

    fpr["macro"] = all_fpr
    tpr["macro"] = mean_tpr
    roc_auc["macro"] = auc(fpr["macro"], tpr["macro"])

    # Plot all ROC curves
    plt.figure()
    plt.plot(
        fpr["micro"],
        tpr["micro"],
        label="micro-average ROC curve (area = {0:0.2f})".format(roc_auc["micro"]),
        color="deeppink",
        linestyle=":",
        linewidth=4,
    )

    plt.plot(
        fpr["macro"],
        tpr["macro"],
        label="macro-average ROC curve (area = {0:0.2f})".format(roc_auc["macro"]),
        color="navy",
        linestyle=":",
        linewidth=4,
    )

    colors = cycle(["aqua", "darkorange", "cornflowerblue"])
    for i, color in zip(range(n_classes), colors):
```

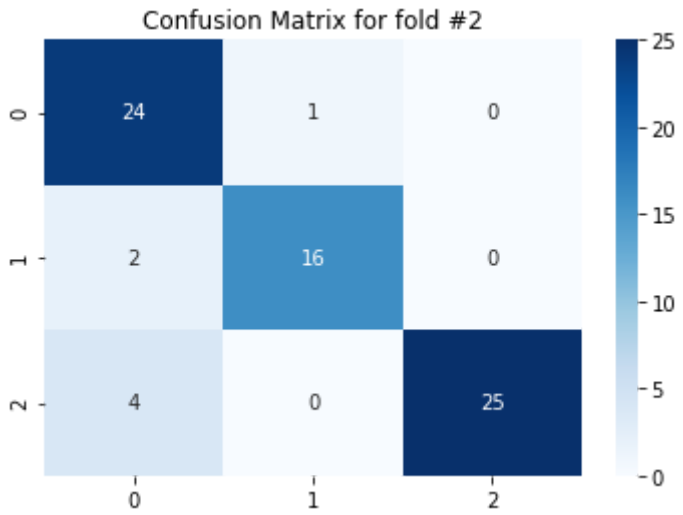
```
        plt.plot(
            fpr[i],
            tpr[i],
            color=color,
            lw=2,
            label="ROC curve of class {0} (area = {1:0.2f})".format(i, roc_auc[i]),
        )

    plt.plot([0, 1], [0, 1], "k--", lw=2)
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel("False Positive Rate")
    plt.ylabel("True Positive Rate")
    plt.title(f"Fold {fold}")
    plt.legend(loc="lower right")
    plt.show()
```

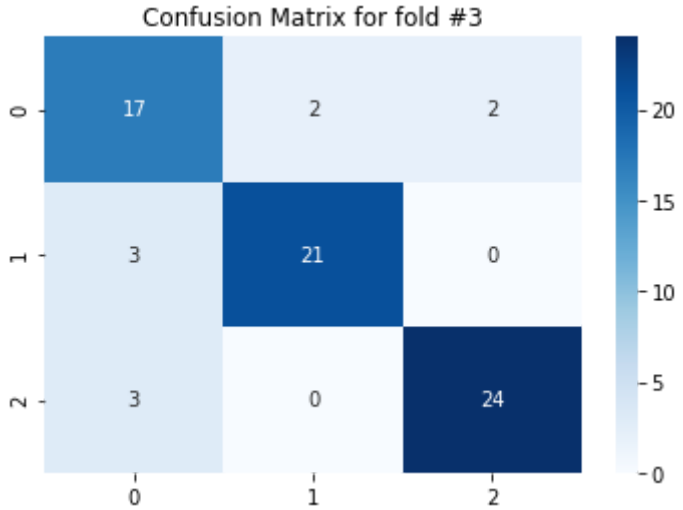
K-Nearest Neighbors:

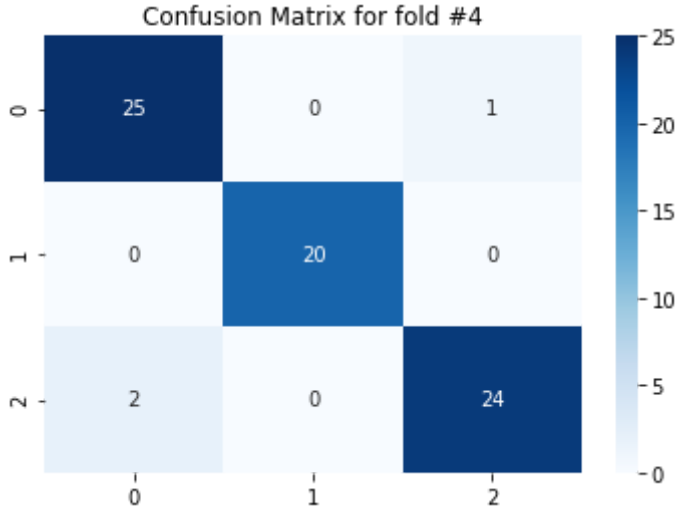
In [30]:

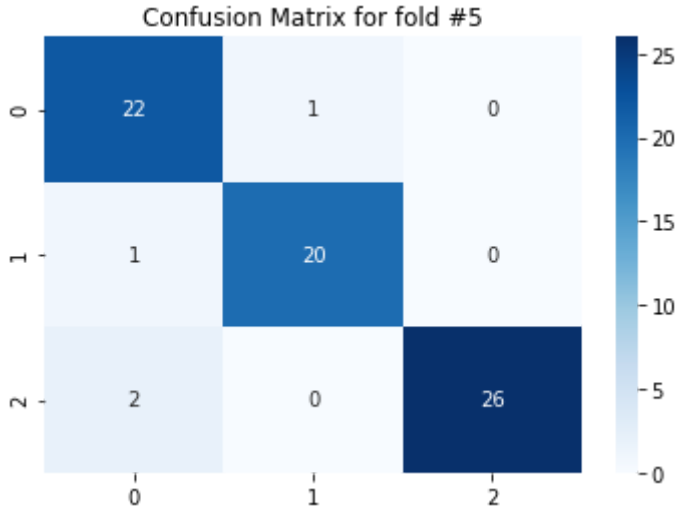
```
parameters = KfoldProcess(X, y, neigh, 5)
```





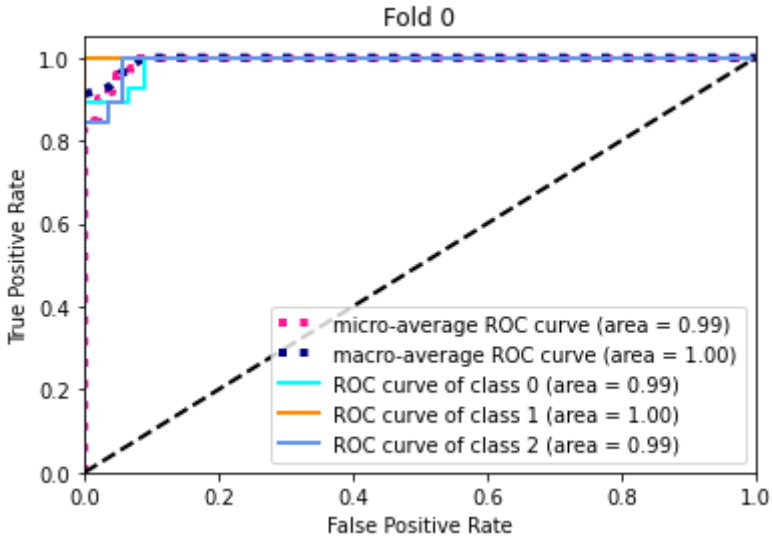


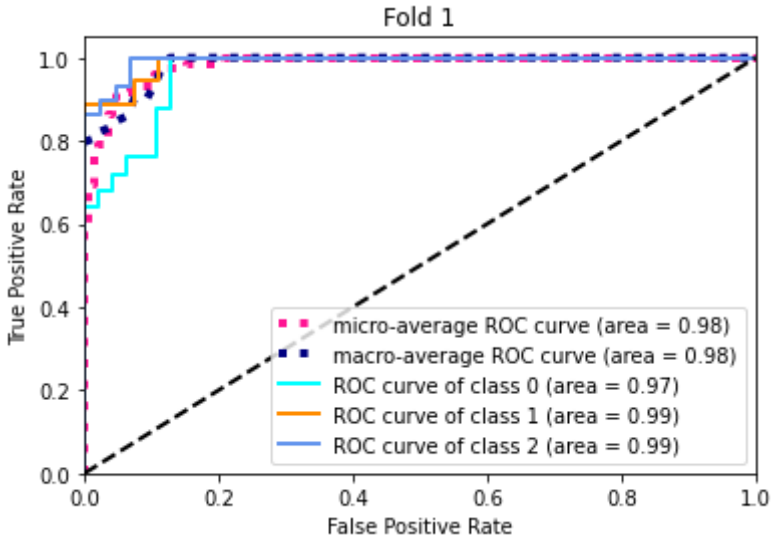


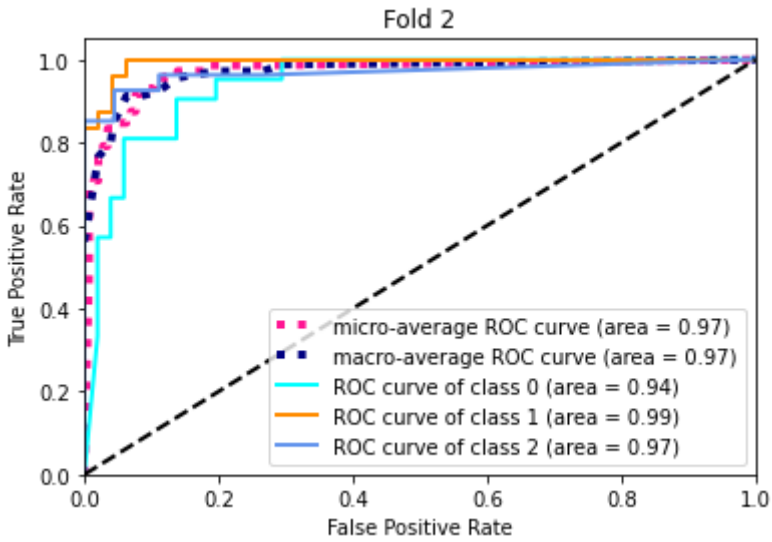


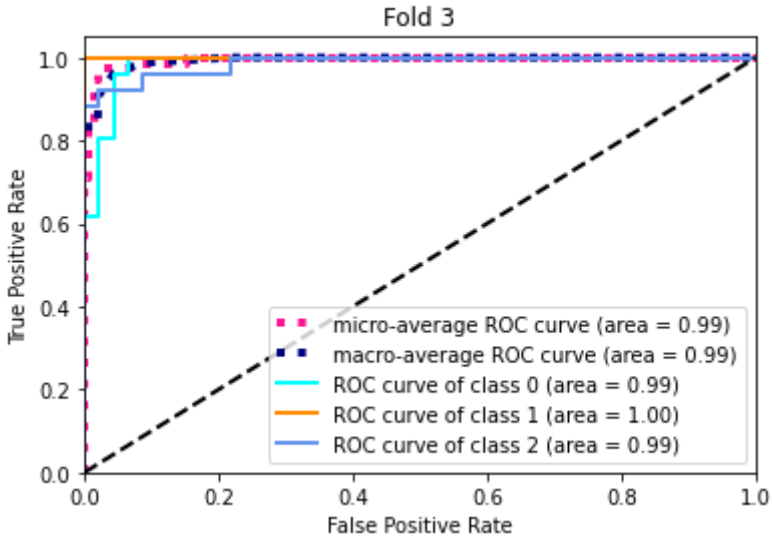
In [31]:

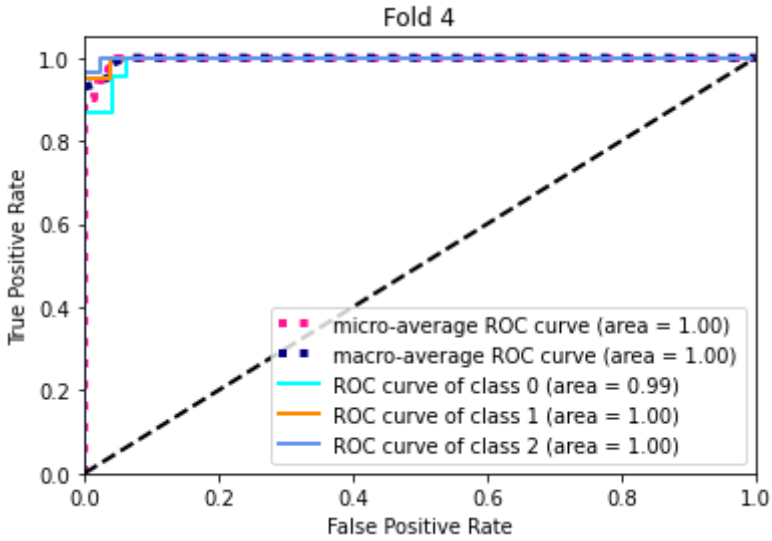
```
for i in range(len(parameters[0])):
    plot_Kfold(parameters[0][i], parameters[1][i], parameters[2][i], fold=i)
```


Fold 0
True Positive Rate
False Positive Rate
micro-average ROC curve (area = 0.99)
macro-average ROC curve (area = 1.00)
ROC curve of class 0 (area = 0.99)
ROC curve of class 1 (area = 1.00)
ROC curve of class 2 (area = 0.99)


Fold 1
True Positive Rate
False Positive Rate
micro-average ROC curve (area = 0.98)
macro-average ROC curve (area = 0.98)
ROC curve of class 0 (area = 0.97)
ROC curve of class 1 (area = 0.99)
ROC curve of class 2 (area = 0.99)


Fold 2
True Positive Rate
False Positive Rate
micro-average ROC curve (area = 0.97)
macro-average ROC curve (area = 0.97)
ROC curve of class 0 (area = 0.94)
ROC curve of class 1 (area = 0.99)
ROC curve of class 2 (area = 0.97)

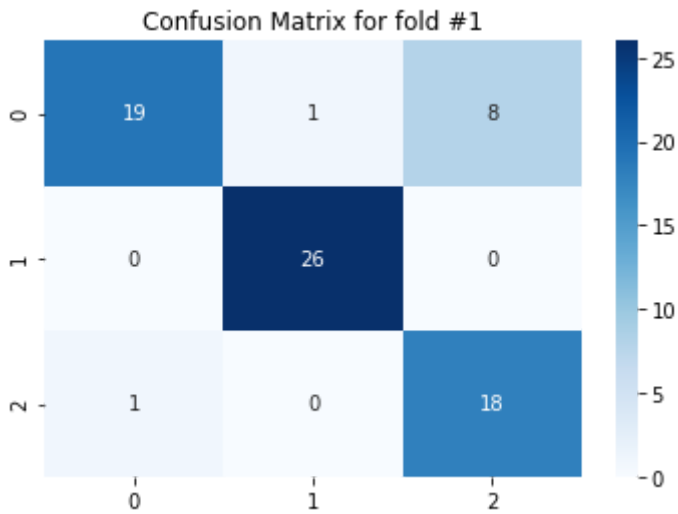


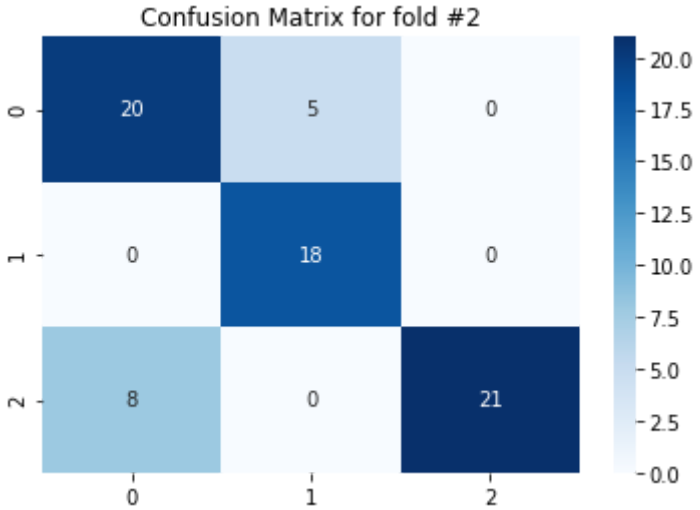


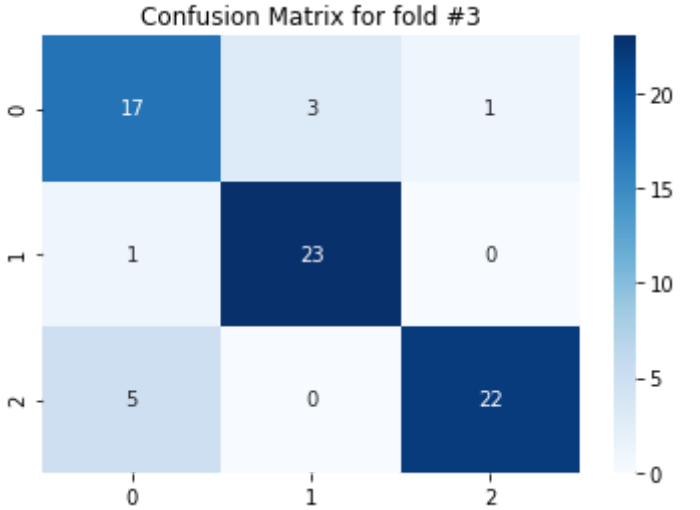
Gaussian Naive Bayes:

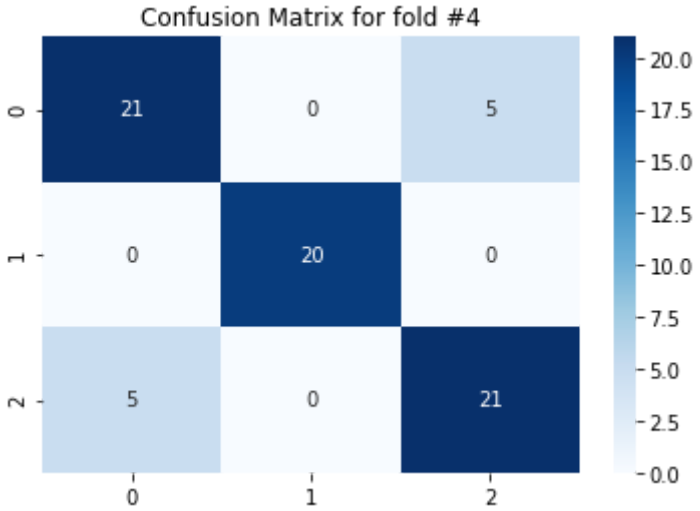
In [32]:

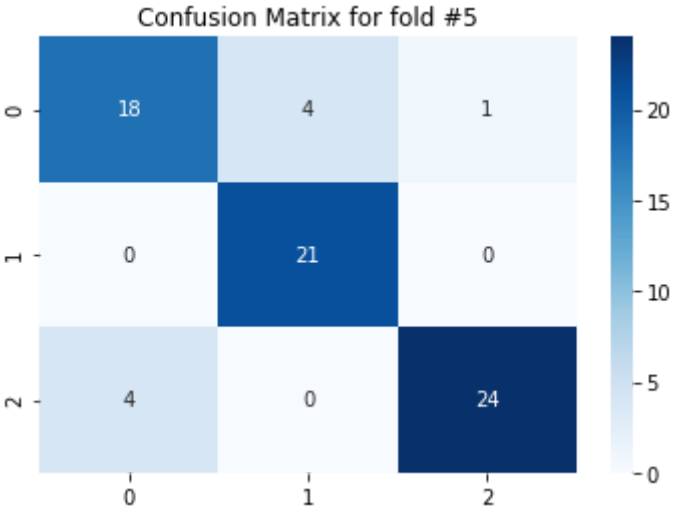
```
parameters = KfoldProcess(X, y, GNB, 5)
```




Confusion Matrix for fold #2
20
5
0
0
18
0
8
0
21
0
1
2
0
1
2
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0

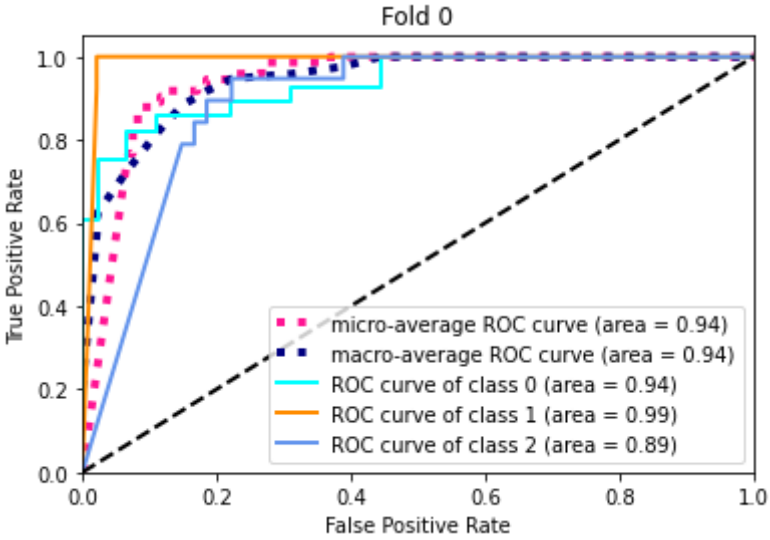

Confusion Matrix for fold #3
17
3
1
1
23
0
5
0
22
0
1
2
0
1
2
20
15
10
5
0

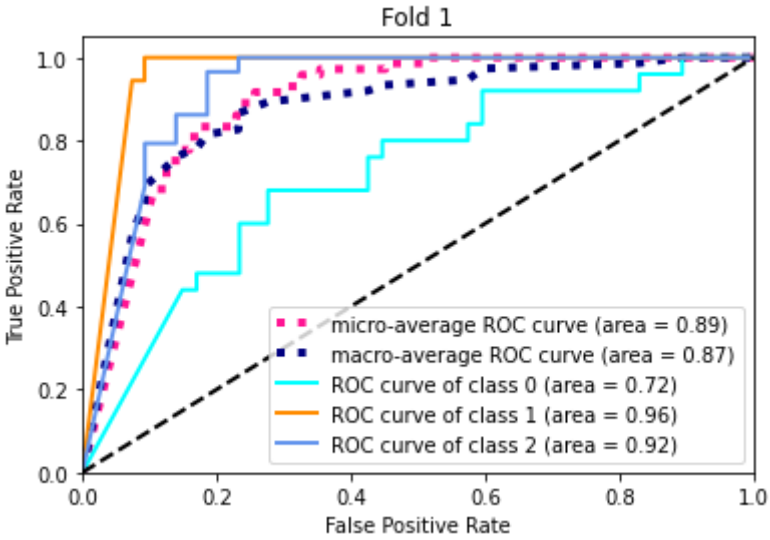

Confusion Matrix for fold #4
21
0
5
0
20
0
5
0
21
0
1
2
0
1
2
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0

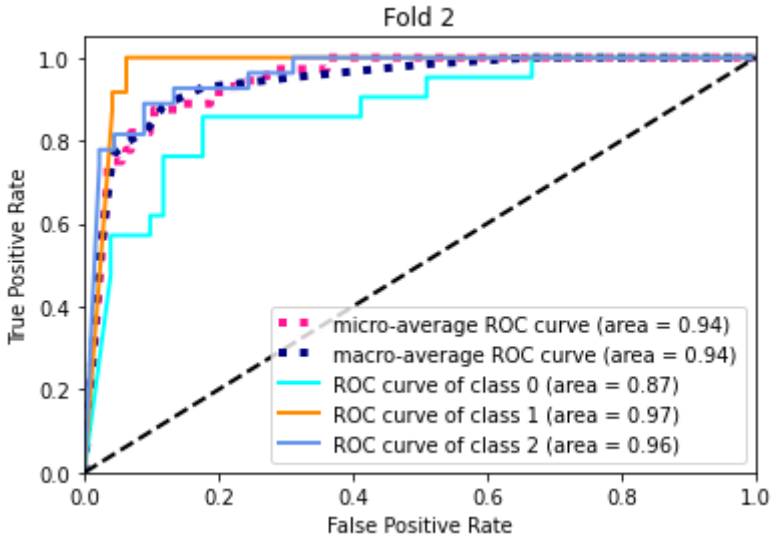


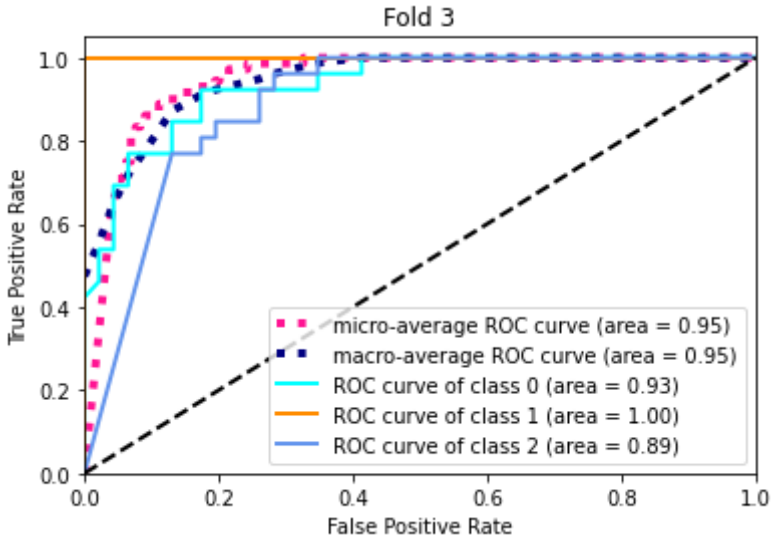
In [33]:

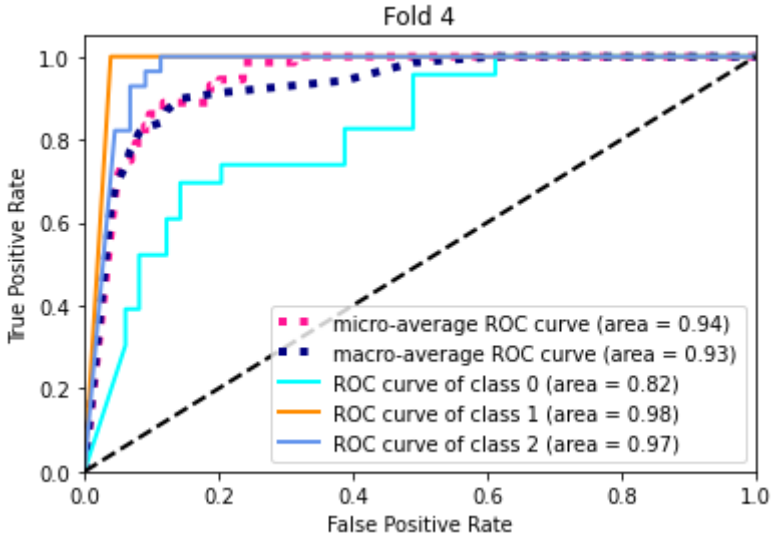
```
for i in range(len(parameters[0])):
    plot_Kfold(parameters[0][i], parameters[1][i], parameters[2][i], fold=i)
```







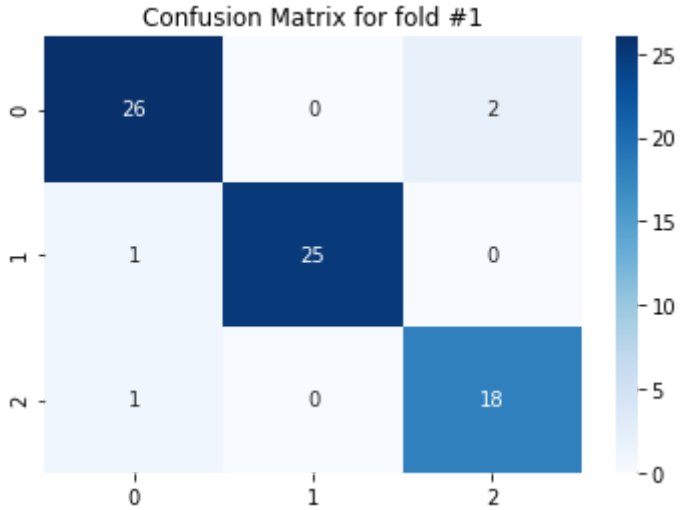


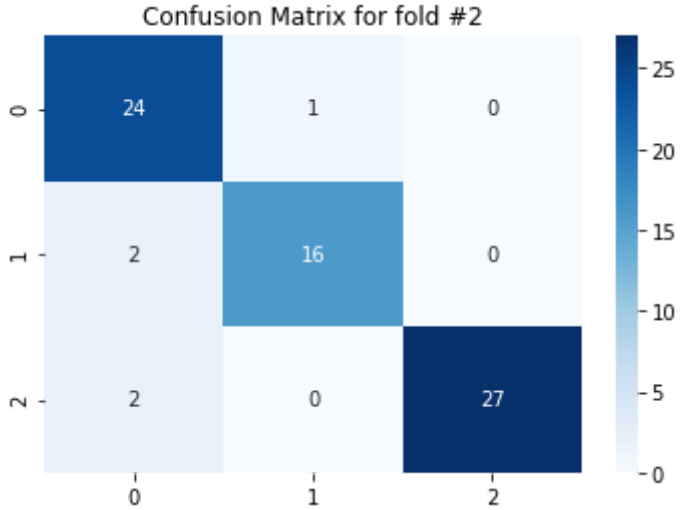


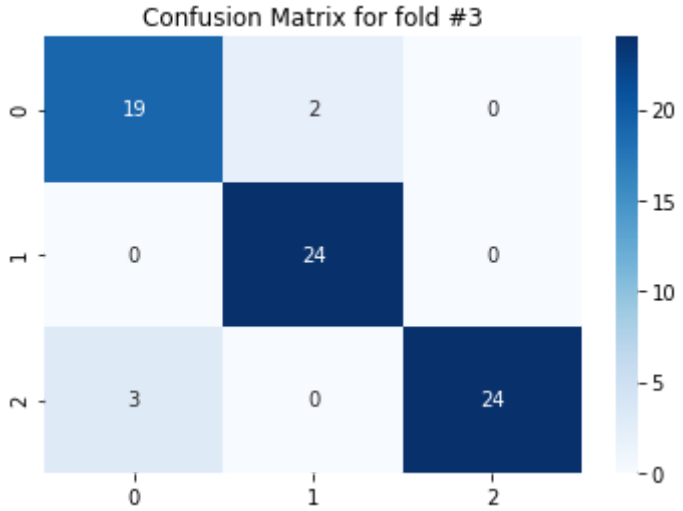
Logistic Regression:

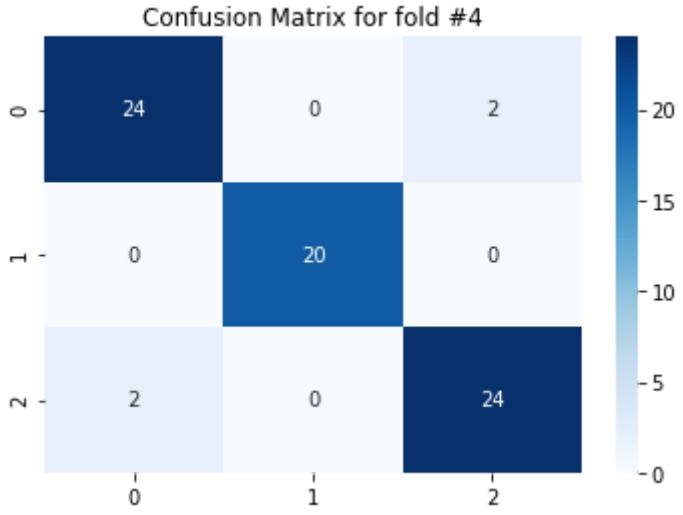
In [34]:

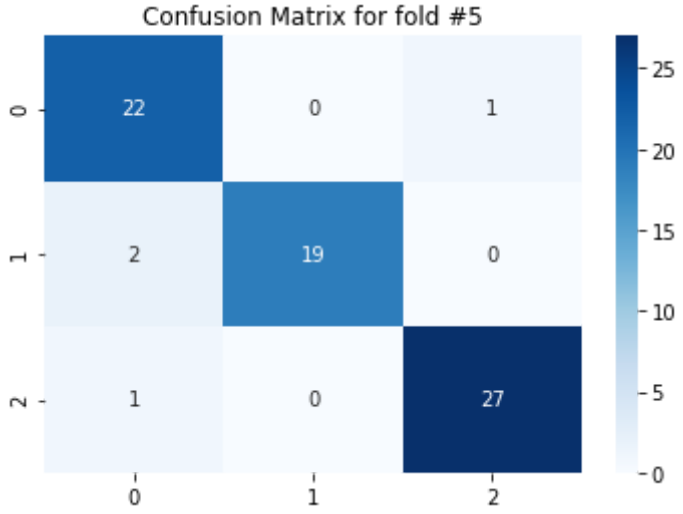
```
parameters = KfoldProcess(X, y, LR, 5)
```


Confusion Matrix for fold #1
26
0
2
1
25
0
1
0
18
0
1
2
25
20
15
10
5
0


Confusion Matrix for fold #2
24
1
0
2
16
0
2
0
27
0
1
2
25
20
15
10
5
0

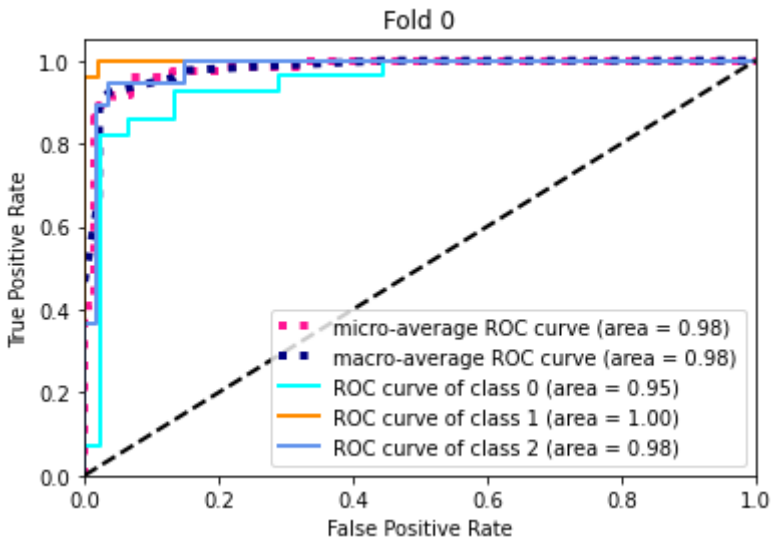

Confusion Matrix for fold #3
19
2
0
0
24
0
3
0
24
0
1
2
20
15
10
5
0

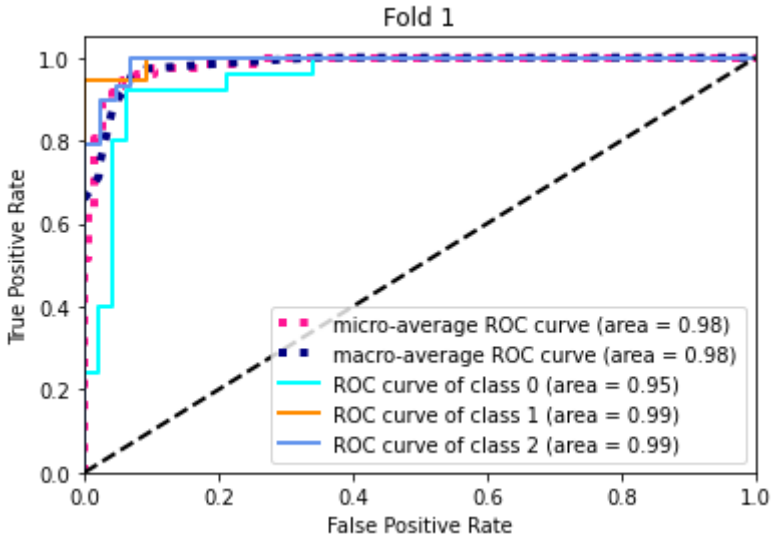


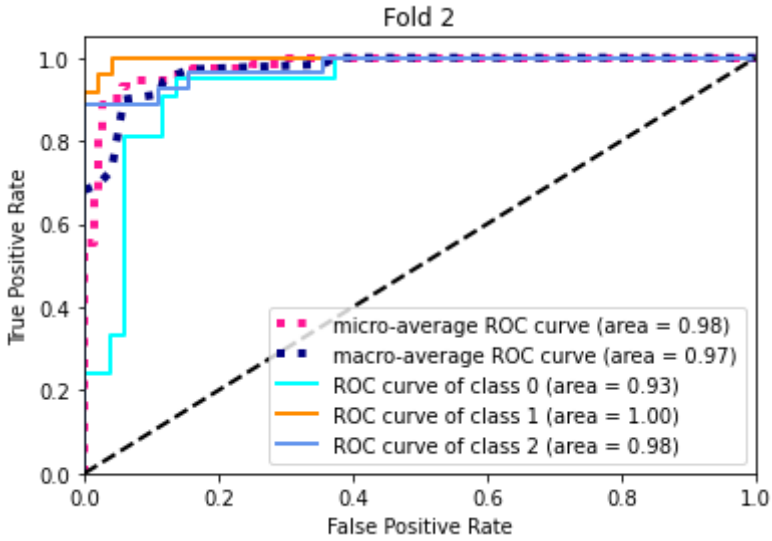


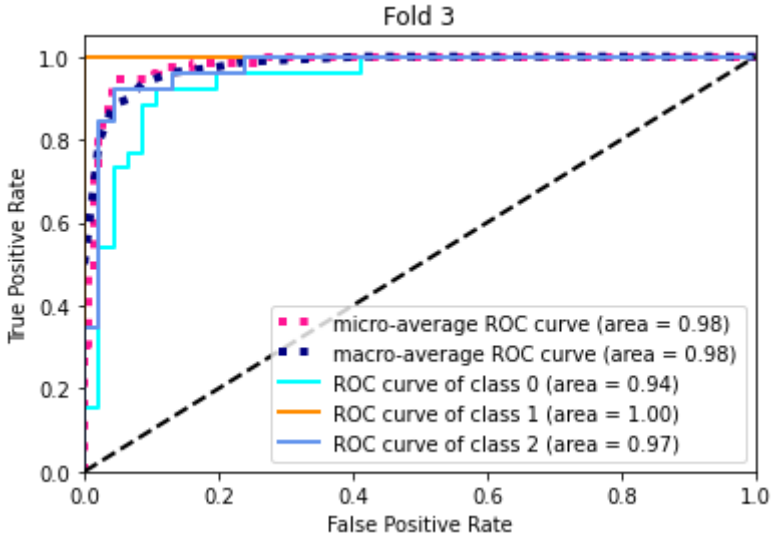
In [35]:

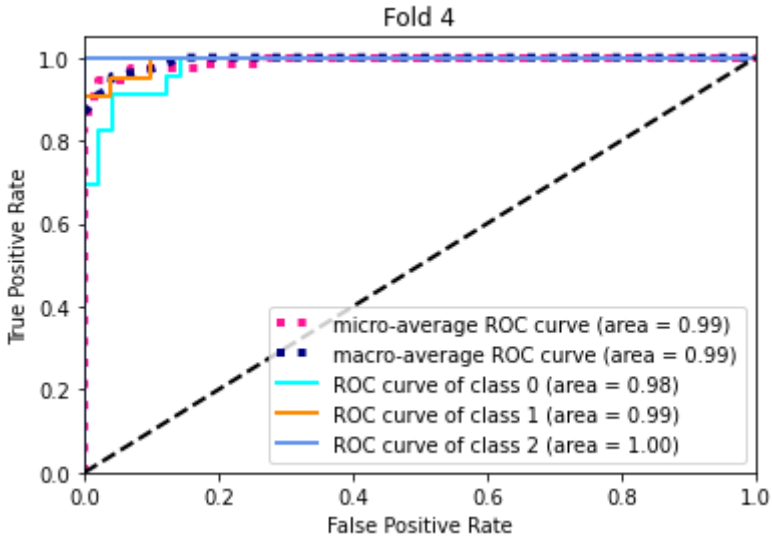
```
for i in range(len(parameters[0])):
    plot_Kfold(parameters[0][i], parameters[1][i], parameters[2][i], fold=i)
```




Fold 1
True Positive Rate
False Positive Rate
micro-average ROC curve (area = 0.98)
macro-average ROC curve (area = 0.98)
ROC curve of class 0 (area = 0.95)
ROC curve of class 1 (area = 0.99)
ROC curve of class 2 (area = 0.99)


Fold 2
True Positive Rate
False Positive Rate
micro-average ROC curve (area = 0.98)
macro-average ROC curve (area = 0.97)
ROC curve of class 0 (area = 0.93)
ROC curve of class 1 (area = 1.00)
ROC curve of class 2 (area = 0.98)

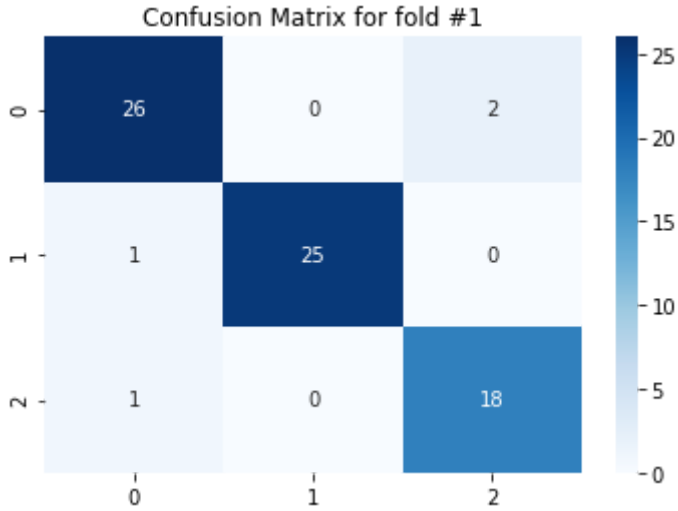

Fold 3
True Positive Rate
False Positive Rate
micro-average ROC curve (area = 0.98)
macro-average ROC curve (area = 0.98)
ROC curve of class 0 (area = 0.94)
ROC curve of class 1 (area = 1.00)
ROC curve of class 2 (area = 0.97)

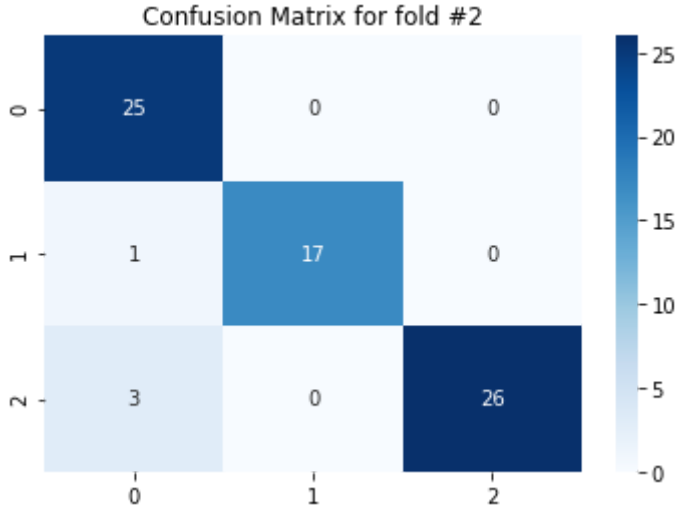


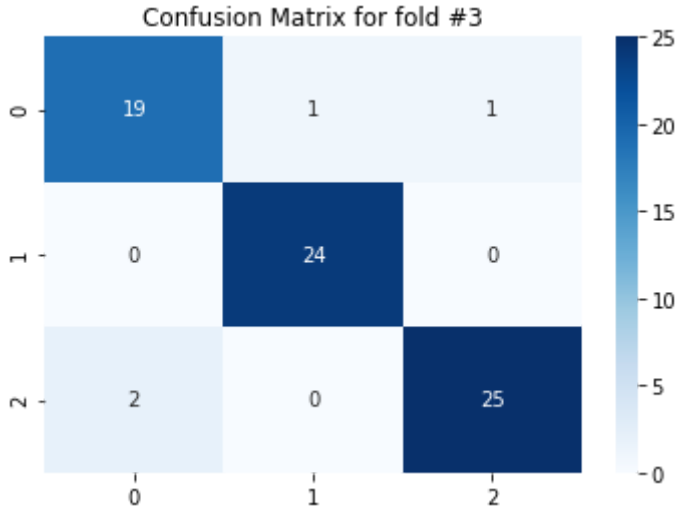
SVC:

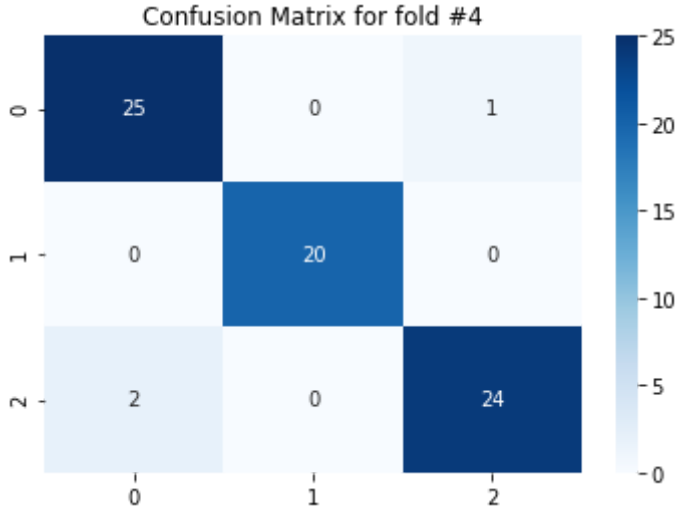
In [36]:

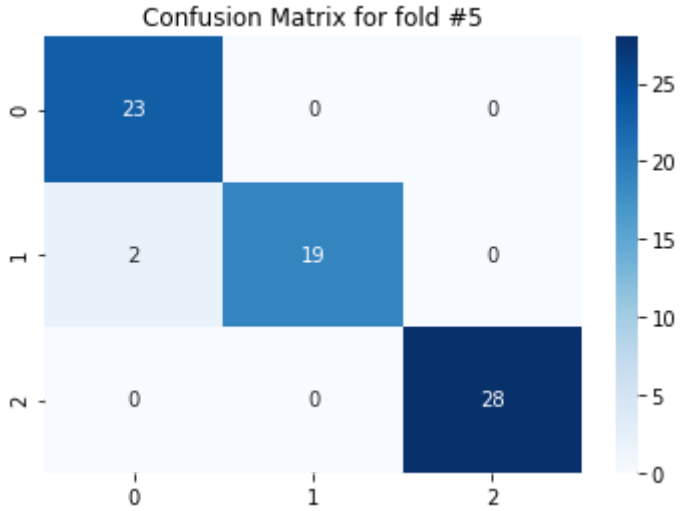
```
parameters = KfoldProcess(X, y, svc, 5)
```





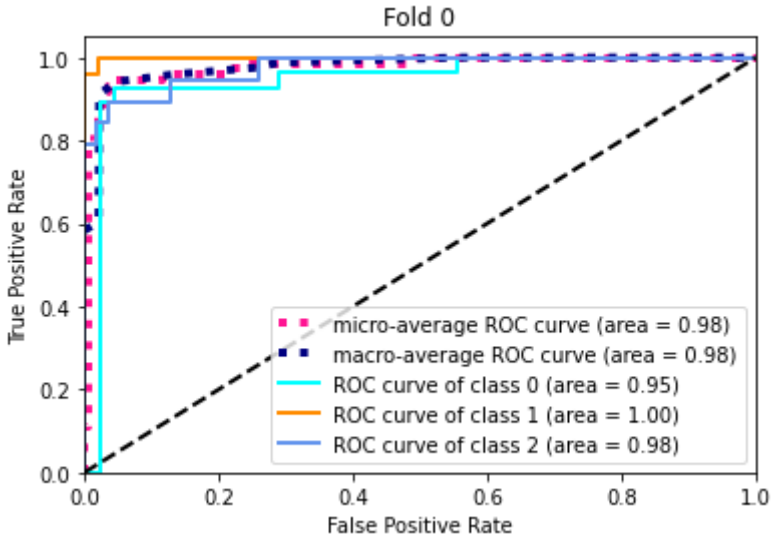


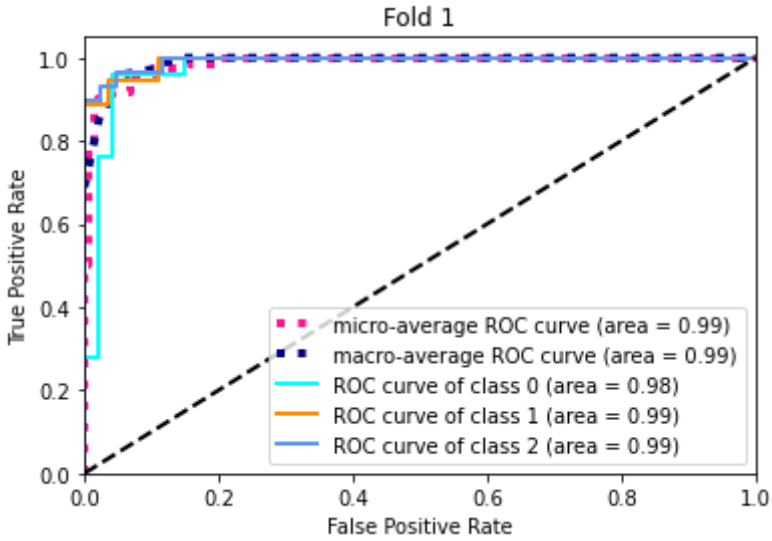


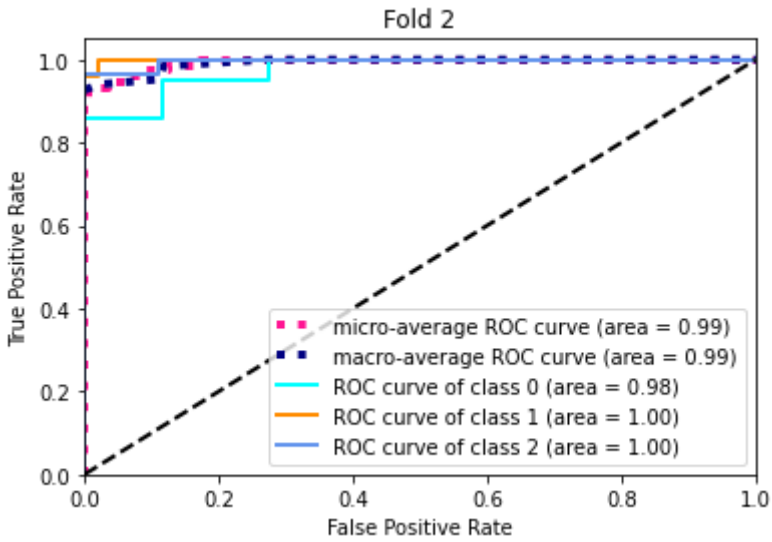


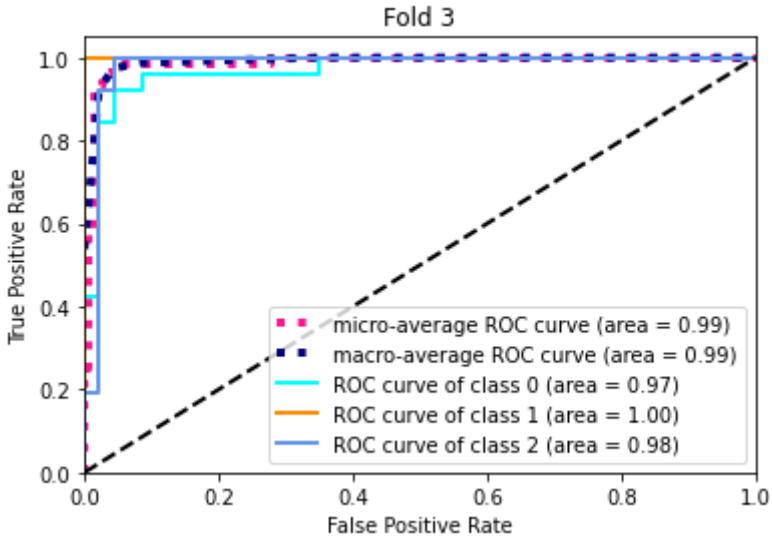
In [37]:

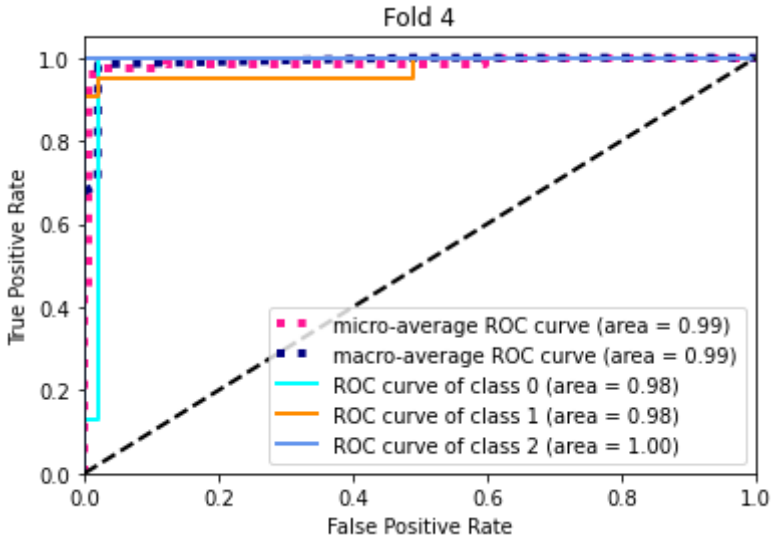
```
for i in range(len(parameters[0])):
    plot_Kfold(parameters[0][i], parameters[1][i], parameters[2][i], fold=i)
```


Fold 0
True Positive Rate
False Positive Rate
micro-average ROC curve (area = 0.98)
macro-average ROC curve (area = 0.98)
ROC curve of class 0 (area = 0.95)
ROC curve of class 1 (area = 1.00)
ROC curve of class 2 (area = 0.98)


Fold 1
True Positive Rate
False Positive Rate
micro-average ROC curve (area = 0.99)
macro-average ROC curve (area = 0.99)
ROC curve of class 0 (area = 0.98)
ROC curve of class 1 (area = 0.99)
ROC curve of class 2 (area = 0.99)


Fold 2
True Positive Rate
False Positive Rate
micro-average ROC curve (area = 0.99)
macro-average ROC curve (area = 0.99)
ROC curve of class 0 (area = 0.98)
ROC curve of class 1 (area = 1.00)
ROC curve of class 2 (area = 1.00)

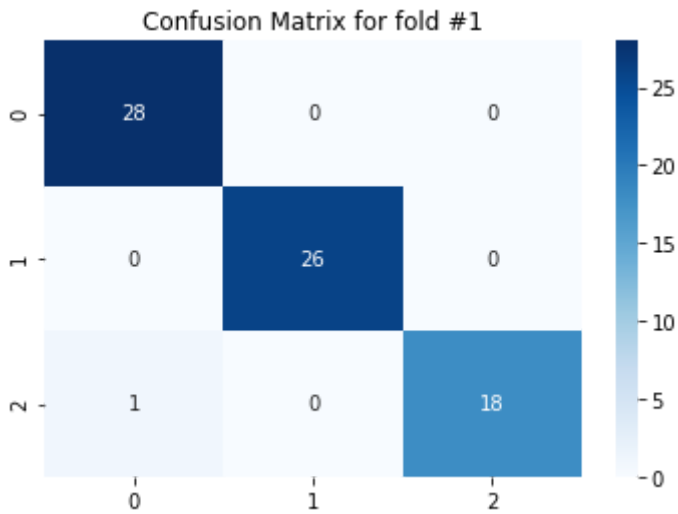


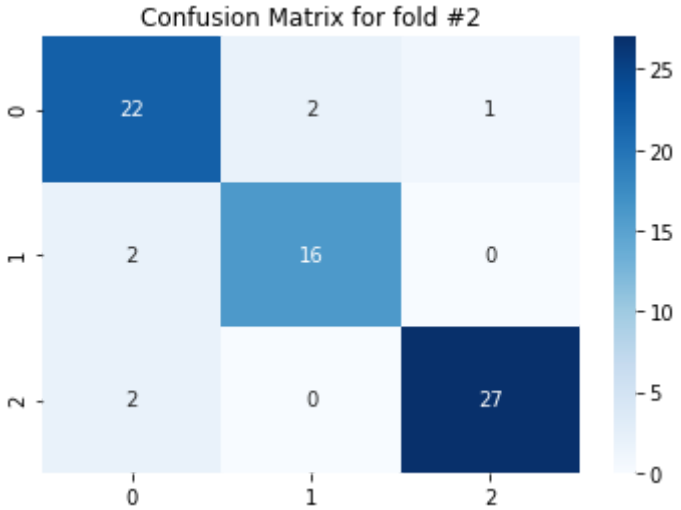


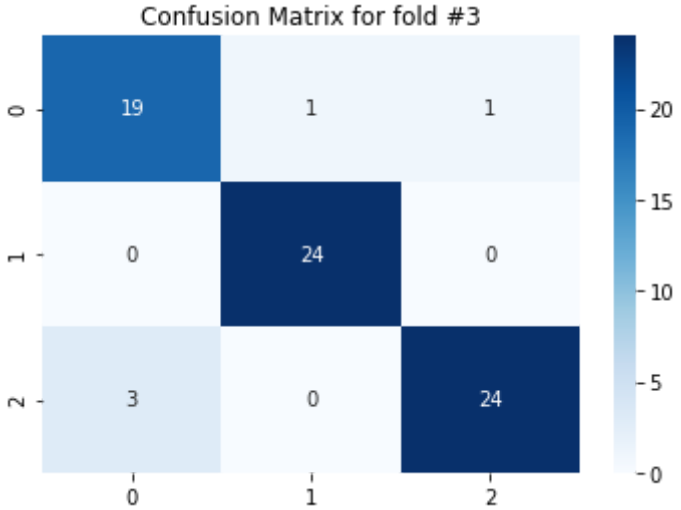
Adaboost:

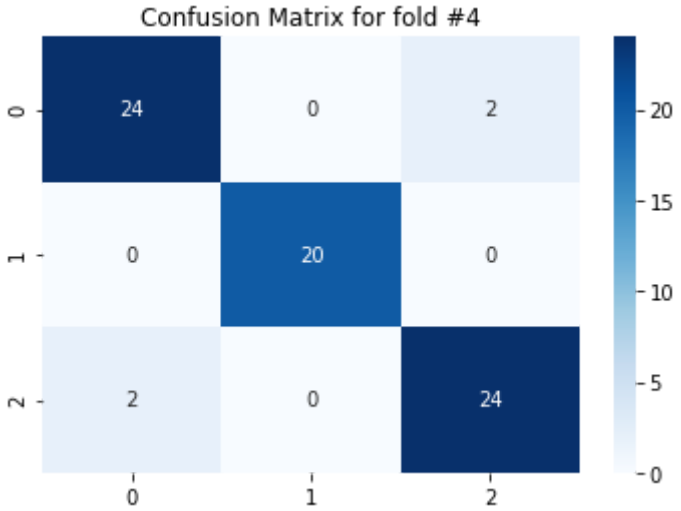
In [38]:

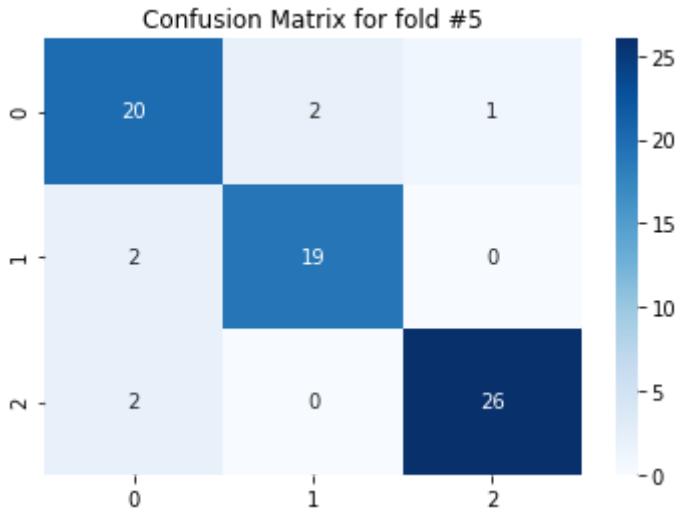
```
parameters = KfoldProcess(X, y, ADB, 5)
```




Confusion Matrix for fold #2
22
2
1
2
16
0
2
0
27
0
1
2
25
20
15
10
5
0

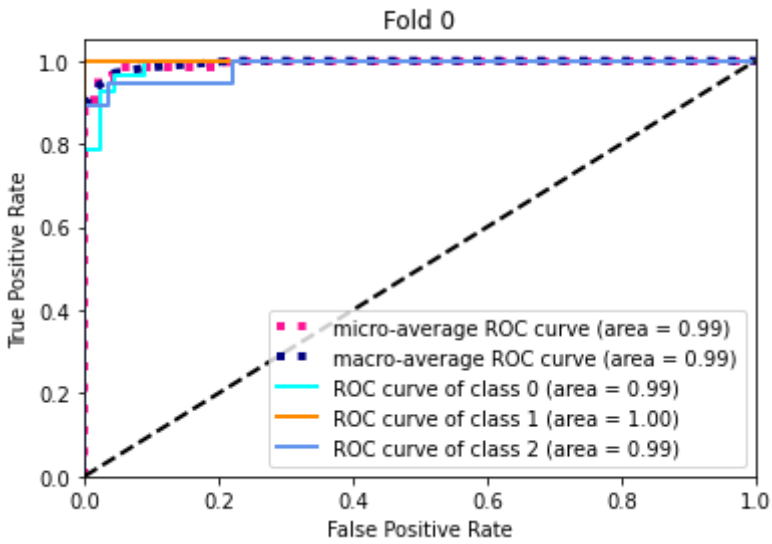

Confusion Matrix for fold #3
19
1
1
0
24
0
3
0
24
0
1
2
20
15
10
5
0

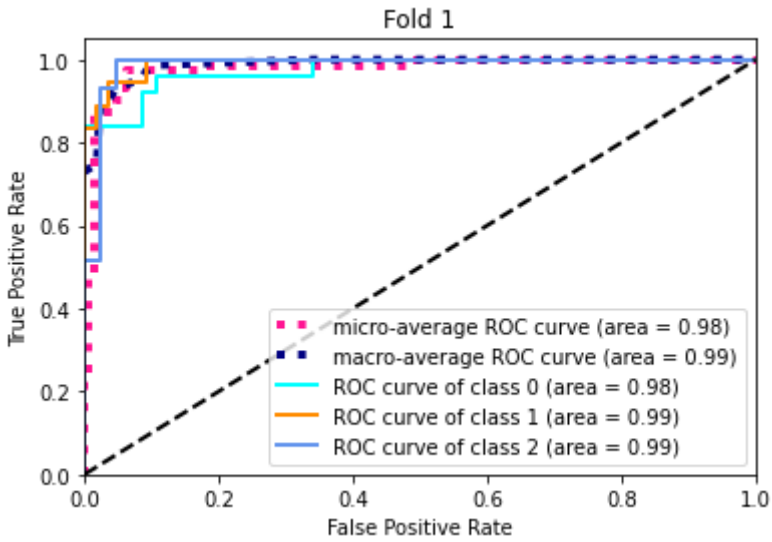

Confusion Matrix for fold #4
24
0
2
0
20
0
2
0
24
0
1
2
20
15
10
5
0

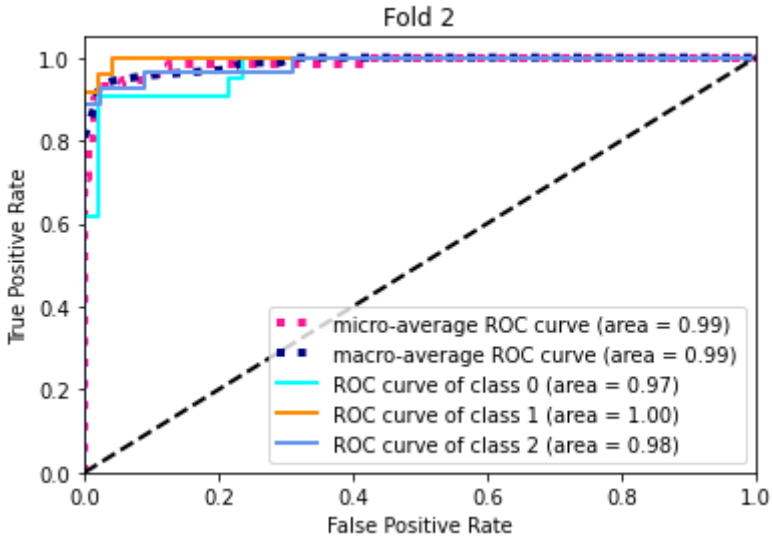


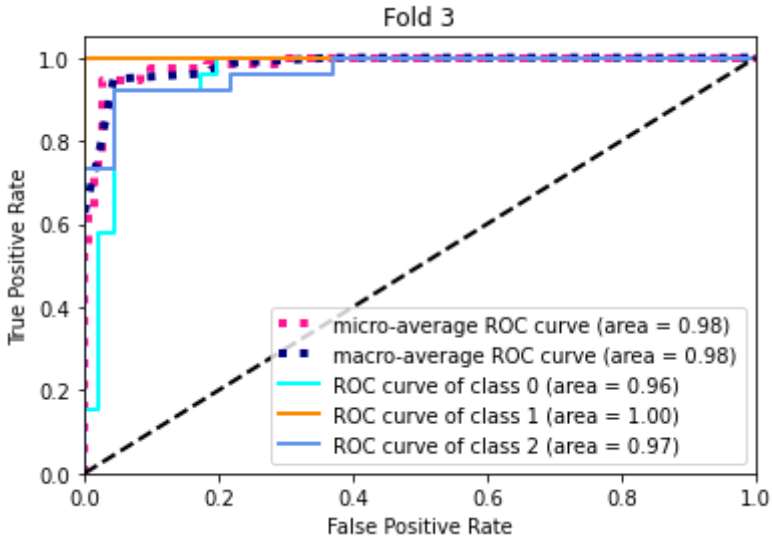
In [39]:

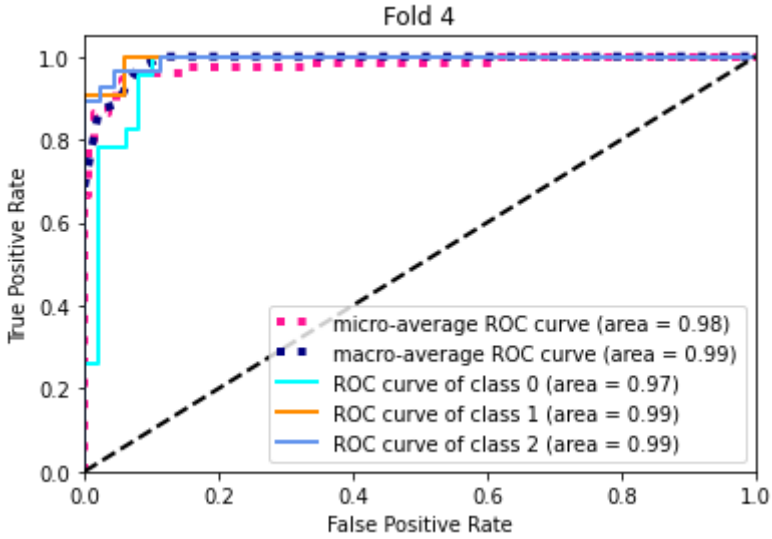
```
for i in range(len(parameters[0])):
    plot_Kfold(parameters[0][i], parameters[1][i], parameters[2][i], fold=i)
```







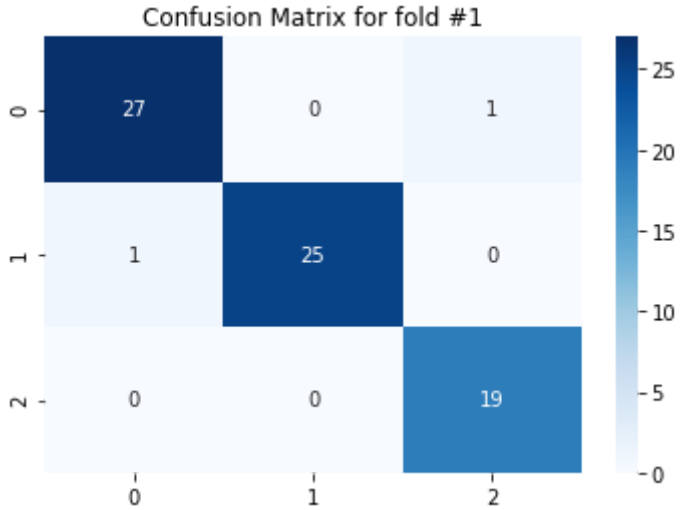


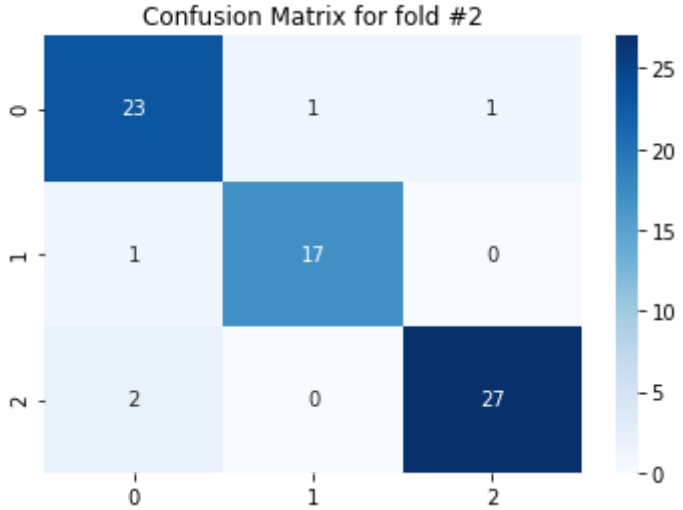


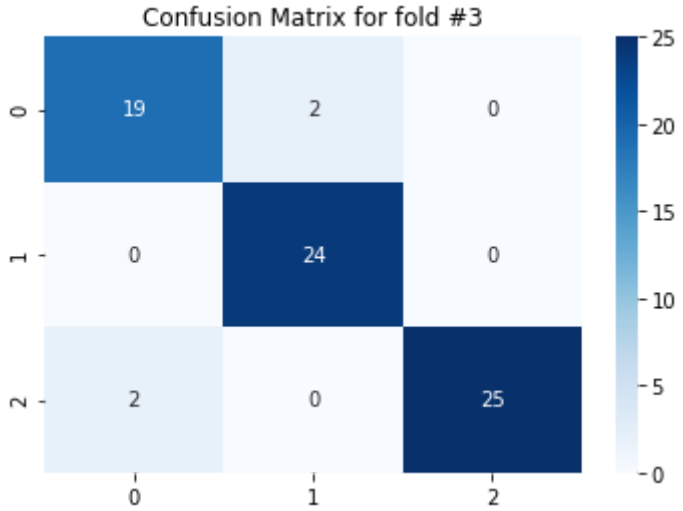
Random Forest:

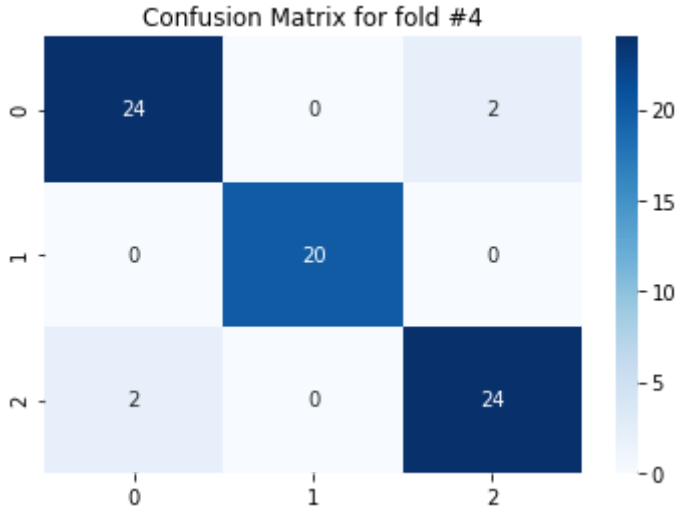
In [40]:

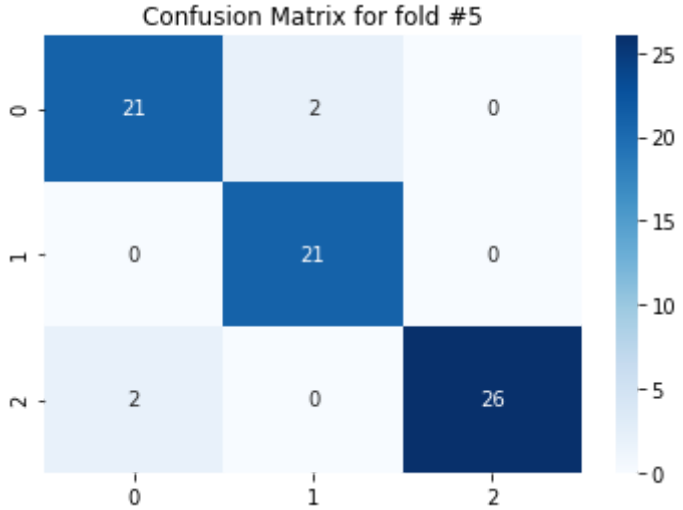
```
parameters = KfoldProcess(X, y, RFC, 5)
```


Confusion Matrix for fold #1
27
0
1
1
25
0
0
0
19
0
1
2
0
1
2
25
20
15
10
5
0


Confusion Matrix for fold #2
23
1
1
1
17
0
2
0
27
0
1
2
0
1
2
25
20
15
10
5
0

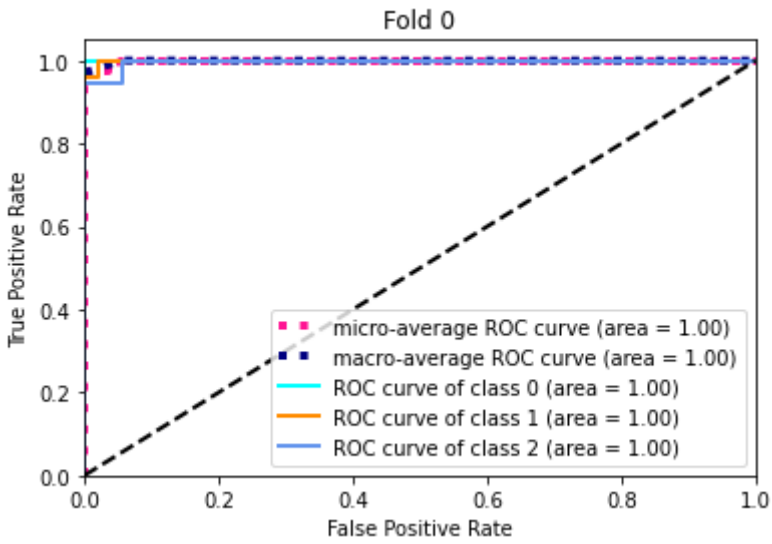

Confusion Matrix for fold #3
19
2
0
0
24
0
2
0
25
0
1
2
0
1
2
25
20
15
10
5
0

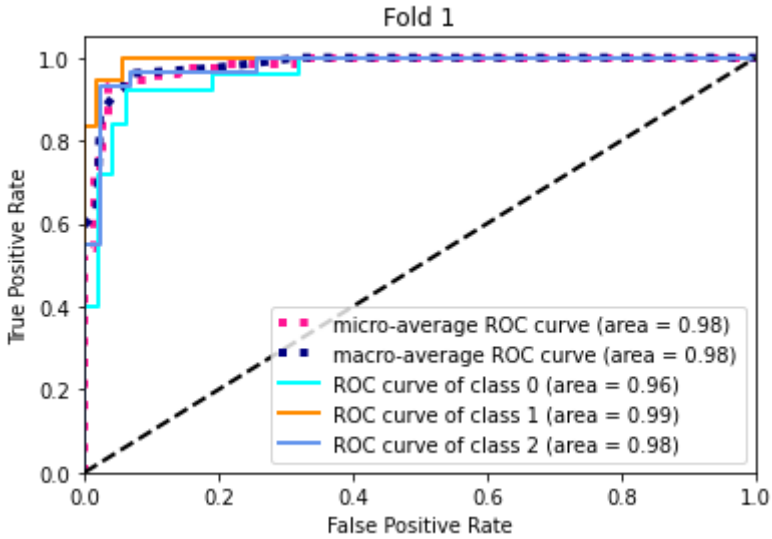


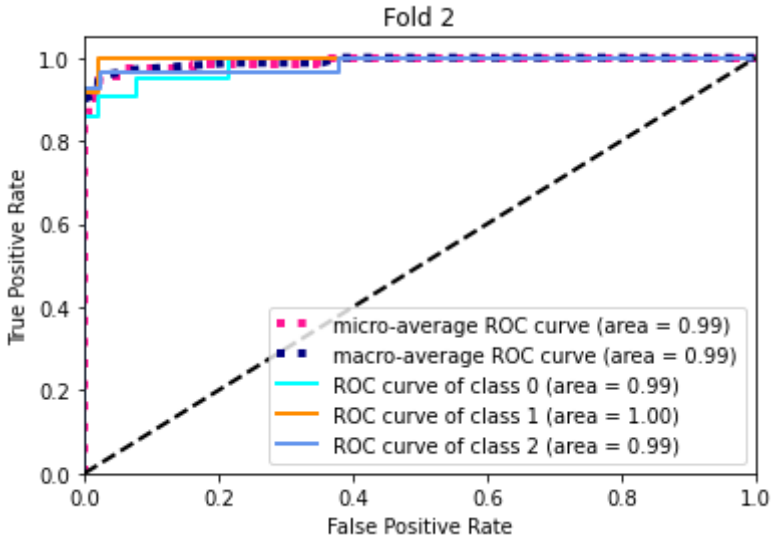


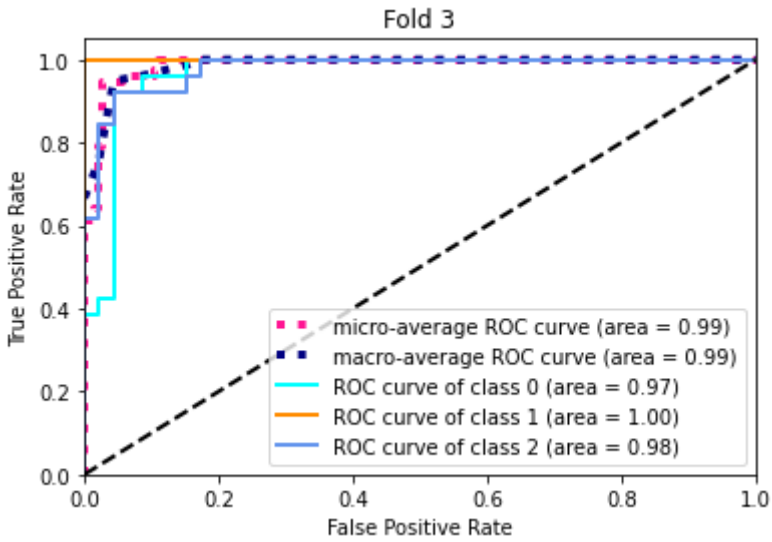
In [41]:

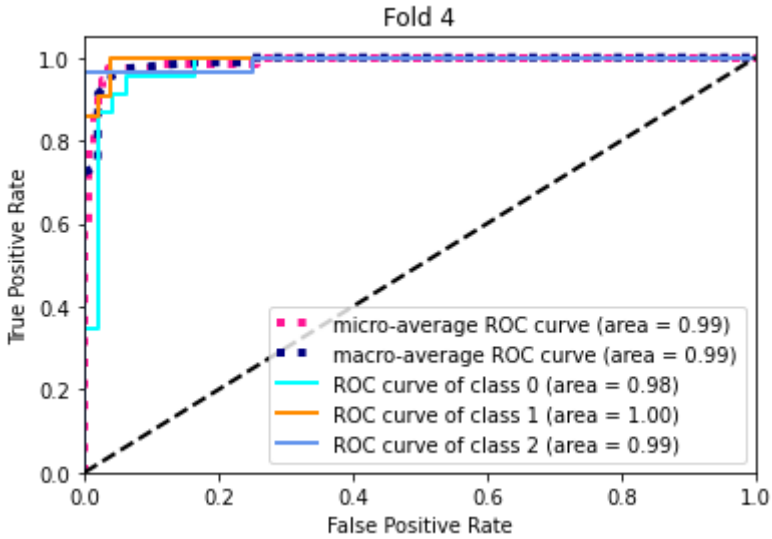
```
for i in range(len(parameters[0])):
    plot_Kfold(parameters[0][i], parameters[1][i], parameters[2][i], fold=i)
```




Fold 1
True Positive Rate
False Positive Rate
micro-average ROC curve (area = 0.98)
macro-average ROC curve (area = 0.98)
ROC curve of class 0 (area = 0.96)
ROC curve of class 1 (area = 0.99)
ROC curve of class 2 (area = 0.98)


Fold 2
True Positive Rate
False Positive Rate
micro-average ROC curve (area = 0.99)
macro-average ROC curve (area = 0.99)
ROC curve of class 0 (area = 0.99)
ROC curve of class 1 (area = 1.00)
ROC curve of class 2 (area = 0.99)


Fold 3
True Positive Rate
False Positive Rate
micro-average ROC curve (area = 0.99)
macro-average ROC curve (area = 0.99)
ROC curve of class 0 (area = 0.97)
ROC curve of class 1 (area = 1.00)
ROC curve of class 2 (area = 0.98)



Now, after choosing our model to be SVC, We will train it over the entire train dataset and explore the results on our test data:

In [42]:

```
classifier = OneVsRestClassifier(svc)
trained = classifier.fit(X, y)
```

In [43]:

```
test
```

Out[43]:

	t0	t1	t2	t3	t4	t5	t6	
0	9.500000	2.833333	8.166667	16.833333	16.833333	7.166667	5.166667	26.5
1	-112.000000	-67.666667	-37.666667	-23.666667	-13.333333	0.333333	19.000000	46.6
2	15.833333	-12.500000	-58.166667	-85.833333	-86.833333	-72.166667	-53.500000	-30.5
3	-254.833333	-258.833333	-254.833333	-248.833333	-243.166667	-242.166667	-243.500000	-226.1
4	26.000000	3.333333	6.000000	28.666667	52.666667	66.000000	72.666667	83.3
5	44.166667	63.833333	70.500000	68.166667	63.166667	56.833333	47.500000	34.8
6	20.666667	18.000000	23.333333	19.333333	6.000000	-15.333333	-23.333333	-11.3
7	25.000000	21.666667	16.666667	13.666667	11.333333	6.333333	0.666667	-5.0
8	-43.500000	-59.500000	-56.833333	-42.833333	-28.166667	-25.500000	-40.166667	-66.8
9	1.000000	5.000000	6.000000	3.333333	1.000000	4.666667	15.000000	29.3
10	-163.166667	-143.166667	-122.166667	-115.166667	-122.500000	-131.500000	-129.833333	-110.5
11	-640.000000	-636.000000	-629.000000	-624.000000	-621.000000	-622.000000	-628.000000	-633.0

	t0	t1	t2	t3	t4	t5	t6	
12	-319.000000	-312.000000	-310.000000	-315.000000	-323.000000	-329.000000	-332.000000	-331.0
13	-630.000000	-624.000000	-614.000000	-606.000000	-603.000000	-607.000000	-614.000000	-620.0

Preparing our test data:

In [44]:

```
data_test = test.iloc[:, :-4]
labels_test = test['label']

for index, row in data_test.iterrows():
    scaler = MinMaxScaler()
    scaler.fit(data_test.iloc[index,:].values.reshape(-1, 1))
    data_test.iloc[index,:] = scaler.transform(data_test.iloc[index,:].values.reshap

lab_ = []
for elem in labels_test:
    if elem == '4-Rose':
        lab_.append(0)
    elif elem == '1-Benz':
        lab_.append(1)
    elif elem == '6-Ger':
        lab_.append(2)
    else:
        lab_.append(-1)
```

In [45]:

```
lab_
```

Out[45]:

```
[1, 1, 1, 1, 0, 0, 0, 2, 2, 2, 2, -1, -1, -1]
```

In [46]:

```
labels_test = pd.DataFrame(lab_, columns=['Label']) # [:-3]

y_test = labels_test.Label.values
X_test = data_test.values
```

In [47]:

```
y_score = trained.predict_proba(X_test)
```

In [48]:

```
np.round(y_score*100, decimals=1)
```

Out[48]:

```
array([[  0. , 100. ,   0. ],
       [  1.5, 100. ,   0. ],
       [  0. ,  99.7,   0. ],
       [  0.2,  96.8,   0.6],
       [ 62.5,   2.9,  16.1],
       [  1.3,   0. ,  97.6],
       [ 97.9,   6.4,   0.4],
       [  0.6,   0.6, 100. ],
       [  0.1,   0. , 100. ],
       [  0.4,   0.4, 100. ],
```

```
       [  0.2,   0. ,  99.1],
       [  1.5,  58.3,  99. ],
       [ 15.7,  76.5,  43.3],
       [ 58.5,  95.2,  44.9]])
```

In [49]:

```
trained.predict(X_test)
```

Out[49]:

```
array([[0, 1, 0],
       [0, 1, 0],
       [0, 1, 0],
       [0, 1, 0],
       [1, 0, 0],
       [0, 0, 1],
       [1, 0, 0],
       [0, 0, 1],
       [0, 0, 1],
       [0, 0, 1],
       [0, 0, 1],
       [0, 1, 1],
       [0, 1, 1],
       [1, 1, 1]])
```

In [50]:

```
def pred(data):
    lst = []
    for row in data:
        if sum(row) != 1:
            lst.append(-1)
        else:
            lst.append(int(np.where(row == 1)[0]))
    return np.array(lst)
```

In [51]:

```
preds = pred(trained.predict(X_test))
```

In [52]:

```
y_test == preds
```

Out[52]:

```
array([ True,  True,  True,  True,  True, False,  True,  True,  True,
        True,  True,  True,  True,  True])
```

In [53]:

```
print(f'Model accuracy on test data is: {round(sum(y_test == preds)/len(y_test)*100,
```

```
Model accuracy on test data is: 92.86%
```

In [54]:

```
import pickle
with open('svc_model.pkl', 'wb') as h:
    pickle.dump(trained, h)
```

In [ ]:

In [ ]:

```
import pandas as pd
import numpy as np
import pickle
from sklearn.preprocessing import MinMaxScaler
from Motor import *


def Motor_Command(Classifier_output):

    if Classifier_output == 0:  # ('Ger')
        PWM.setMotorModel(1000, 1000, 1000, 1000)  # Forward
        print("The car is moving forward")
        time.sleep(1)
    elif Classifier_output == -1:  # ('Rose')
        PWM.setMotorModel(-1000, -1000, 1500, 1500)  # Left
        print("The car is turning left")
        time.sleep(1)
    elif Classifier_output == 1:  # ('Benz')
        PWM.setMotorModel(1500, 1500, -1000, -1000)  # Right
        print("The car is turning right")
        time.sleep(1)
    else:
        print('No match')

    PWM.setMotorModel(0, 0, 0, 0)  # Stop
    print("\nEnd session")

if __name__ == '__main__':

    test = pd.read_excel('../data/Test_odors_and_control.xlsx')
    data_test = test.iloc[:, :-4]
    labels_test = test['label']

    for index, row in data_test.iterrows():
        scaler = MinMaxScaler()
        scaler.fit(data_test.iloc[index,:].values.reshape(-1, 1))
        data_test.iloc[index,:] = scaler.transform(data_test.iloc[index,:].values.reshape(-1, 1

    lab_ = []
    for elem in labels_test:
        if elem == '4-Rose':
            lab_.append(-1)
        elif elem == '1-Benz':
            lab_.append(1)
        elif elem == '6-Ger':
            lab_.append(0)
        else:
            lab_.append(None)

    labels_test = pd.DataFrame(lab_[:-3], columns=['Label'])

    y_test = labels_test.Label.values
    # X_test = data_test.values[:-3, :]
    X_test = data_test.values

    with open('../data/svc_model.pkl', 'rb') as h:
        svc = pickle.load(h)

    pred = svc.predict(X_test)

    PWM=Motor()

    for prediction in pred:
        Motor_Command(prediction)
```