report_graphs

October 26, 2023

[ ]:
```python
import torch
import pandas as pd
import numpy as np
import torch.nn as nn


import  sklearn
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
from tqdm import tqdm
```

[ ]:
```python
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

[ ]:
```python
dataset_path = "imbalanced-benchmarking-set/datasets/MagicTelescope.csv"

df = pd.read_csv(dataset_path)


df = df.drop(df.columns[0], axis=1)

# Y - "TARGET"
# X - all other columns

Y = df["TARGET"]
X = df.drop(columns=["TARGET"])




unique_Y = Y.unique()
map_Y = dict(zip(unique_Y, range(len(unique_Y))))
print(map_Y)

Y = Y.map(map_Y)
```

```python
number_of_classes = len(unique_Y)
print("Number of classes: ", number_of_classes)

number_of_features = len(X.columns)
print("Number of features: ", number_of_features)

X_df = X.copy()
Y_df = Y.copy()

X = X.to_numpy()
Y = Y.to_numpy()

X_not_normalized = X.copy()
X = sklearn.preprocessing.normalize(X)

print("X shape: ", X.shape)
print("Y shape: ", Y.shape)

X_train, X_test, Y_train, Y_test = train_test_split(
    X, Y, test_size=0.33, random_state=42
)

print("X_train shape: ", X_train.shape)
print("X_test shape: ", X_test.shape)
print("Y_train shape: ", Y_train.shape)
print("Y_test shape: ", Y_test.shape)
```

```
{'g': 0, 'h': 1}
Number of classes:  2
Number of features:  10
X shape:  (19020, 10)
Y shape:  (19020,)
X_train shape:  (12743, 10)
X_test shape:  (6277, 10)
Y_train shape:  (12743,)
Y_test shape:  (6277,)
```

[ ]:

```python
# Logistic Regression
model_LR = sklearn.linear_model.LogisticRegression(
    penalty="l2", C=1.0, solver="liblinear", max_iter=1000
)

# Decision Tree
model_DT = sklearn.tree.DecisionTreeClassifier(
    criterion="gini", splitter="best", max_depth=None, min_samples_split=2
)
```

```
# Neural Network
internal_size = 128
model_NN_flat = nn.Sequential(
    nn.Linear(number_of_features,internal_size ),
    nn.LeakyReLU(),
    nn.Linear(internal_size, internal_size),
    nn.LeakyReLU(),
    nn.Linear(internal_size, number_of_classes),
    nn.Softmax(dim=1),
)



# Neural Network deep
internal_size_deep = 32
layers = 5

layers_list = []
layers_list.append(nn.Linear(number_of_features, internal_size_deep))
for i in range(layers):
    layers_list.append(nn.LeakyReLU())
    layers_list.append(nn.Linear(internal_size_deep, internal_size_deep))
layers_list.append(nn.LeakyReLU())
layers_list.append(nn.Linear(internal_size_deep, number_of_classes))
layers_list.append(nn.Softmax(dim=1))

model_NN_deep = nn.Sequential(*layers_list)

model_list_sklearn = [model_LR, model_DT]
model_names_sklearn = ["Logistic Regression", "Decision Tree"]
model_list_pytorch = [model_NN_flat, model_NN_deep]
model_names_pytorch = ["Neural Network flat", "Neural Network deep"]
```

[ ]:
```
for model in model_list_sklearn:
    model.fit(X_train, Y_train)

for model in model_list_pytorch:
    model.train()
    optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
    criterion = nn.CrossEntropyLoss()
    dataset = torch.utils.data.TensorDataset(
        torch.from_numpy(X_train).float(), torch.from_numpy(Y_train).long()
    )
    dataloader = torch.utils.data.DataLoader(dataset, batch_size=128,
 ↪shuffle=True)
    # allow for early stopping
```

```
    patience = 5
    patience_counter = 0
    best_loss = np.inf
    for epoch in tqdm(range(100)):
        epoch_loss = 0
        for batch in dataloader:
            optimizer.zero_grad()
            X_batch, Y_batch = batch
            output = model(X_batch)
            loss = criterion(output, Y_batch)
            loss.backward()
            optimizer.step()
            epoch_loss += loss.item()
        epoch_loss /= len(dataloader)
        if epoch_loss < best_loss:
            best_loss = epoch_loss
            patience_counter = 0
        else:
            patience_counter += 1
        if patience_counter >= patience:
            print("Early stopping")
            break
    print("Best loss: ", best_loss)
```

```
 40%|          | 40/100 [00:19<00:29,  2.03it/s]

Early stopping
Best loss:  0.4774926269054413

 59%|          | 59/100 [00:41<00:28,  1.41it/s]

Early stopping
Best loss:  0.47452698677778243
```

[ ]:
```
for model,model_name in zip(model_list_sklearn[:-1], model_names_sklearn[:-1]):
    Y_pred = model.predict(X_test)
    print("Model: ", model_name)
    print("Accuracy: ", accuracy_score(Y_test, Y_pred))
    print(classification_report(Y_test, Y_pred))
    print("Confusion matrix: ")
    print(confusion_matrix(Y_test, Y_pred))

dataset_test = torch.utils.data.TensorDataset(
    torch.from_numpy(X_test).float(), torch.from_numpy(Y_test).long()
)
dataloader_test = torch.utils.data.DataLoader(dataset_test, batch_size=128,␣
 ↪shuffle=False)
```

```
for model,model_name in zip(model_list_pytorch[:-1], model_names_pytorch[:-1]):
    model.eval()
    Y_pred = []

    for batch in dataloader_test:
        X_batch, Y_batch = batch
        output = model(X_batch)
        Y_pred.append(torch.argmax(output, dim=1).numpy())
    Y_pred = np.concatenate(Y_pred)
    print("Model: ", model_name)
    print("Accuracy: ", accuracy_score(Y_test, Y_pred))
    print(classification_report(Y_test, Y_pred))
    print("Confusion matrix: ")
    print(confusion_matrix(Y_test, Y_pred))
```

```
Model:  Logistic Regression
Accuracy:  0.7270989326111199
              precision    recall  f1-score   support

           0       0.73      0.92      0.81      4071
           1       0.71      0.38      0.49      2206

    accuracy                           0.73      6277
   macro avg       0.72      0.65      0.65      6277
weighted avg       0.72      0.73      0.70      6277

Confusion matrix:
[[3728  343]
 [1370  836]]
Model:  Neural Network flat
Accuracy:  0.822685996495141
              precision    recall  f1-score   support

           0       0.83      0.92      0.87      4071
           1       0.82      0.64      0.72      2206

    accuracy                           0.82      6277
   macro avg       0.82      0.78      0.79      6277
weighted avg       0.82      0.82      0.82      6277

Confusion matrix:
[[3752  319]
 [ 794 1412]]
```

[ ]:
```
# compute class distribution
class_distribution = {}
```

```
for y in Y:
    if y not in class_distribution:
        class_distribution[y] = 0
    class_distribution[y] += 1

# normalize class distribution
class_distribution_normalized = {}
for y in class_distribution:
    class_distribution_normalized[y] = class_distribution[y] / len(Y)
print(class_distribution_normalized)
```

```
{0: 0.6483701366982124, 1: 0.3516298633017876}
```

[ ]:
```
import lime
import lime.lime_tabular

explainer = lime.lime_tabular.LimeTabularExplainer(
    X_train,
    feature_names=df.columns[:-1],
    class_names=unique_Y,
    discretize_continuous=False,
)
```

[ ]:
```
for i in range(5):
    i = np.random.randint(0, len(X_test))

    exp = explainer.explain_instance(
        X_test[i],
        model_list_sklearn[0].predict_proba,
        num_features=10,
        top_labels=1,
        num_samples=1000,
    )

    exp.show_in_notebook(show_table=True, show_all=False)
```

```
<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>
```

[ ]:
```
import random
sum_weights_map = {}
#samples = len(X_test)
```

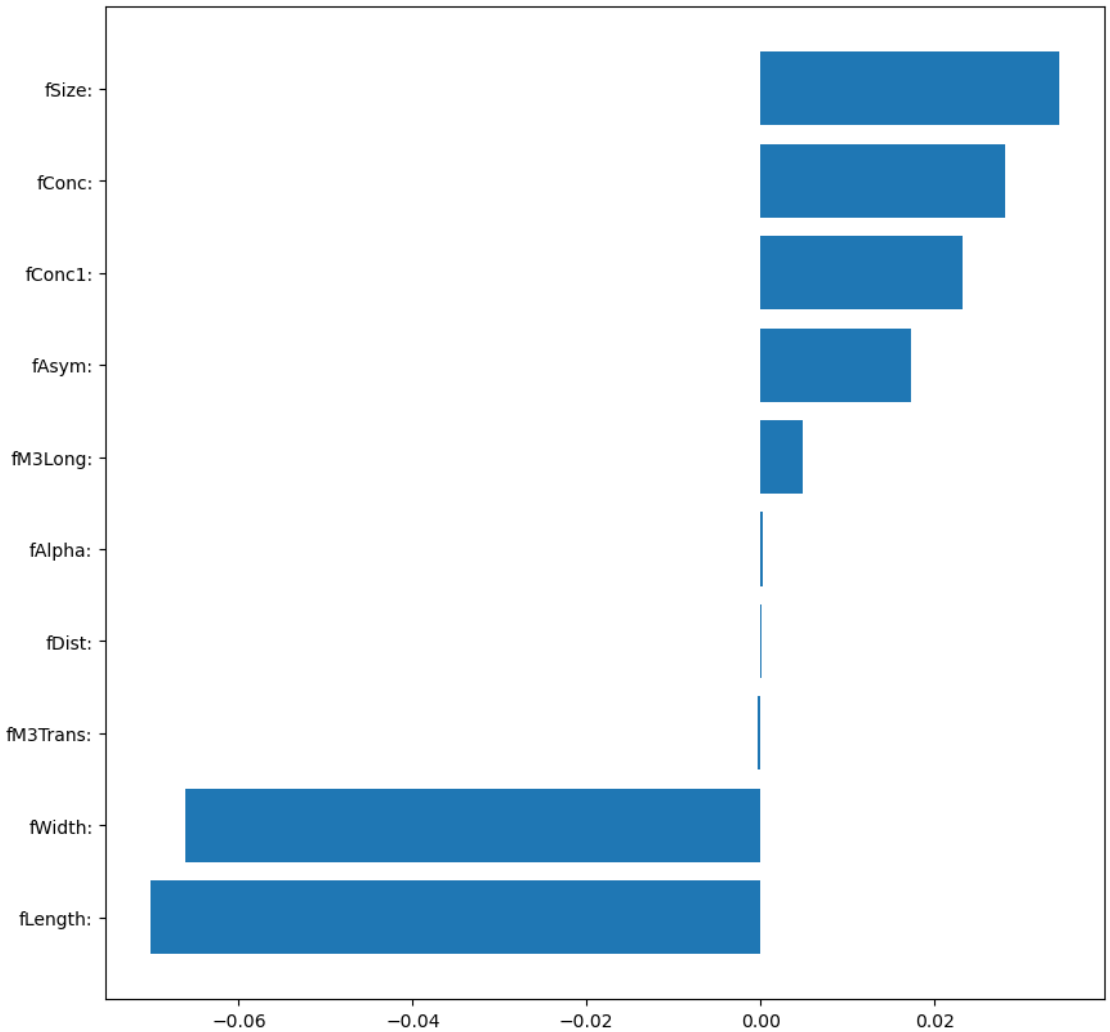
```
samples = 1000
for i in tqdm(range(samples)):
    sample = random.randint(0, len(X_test))
    exp = explainer.explain_instance(
        X_test[sample],
        model_list_sklearn[0].predict_proba,
        num_features=10,
        top_labels=1,
        num_samples=1000,
    )
    for feature, weight in exp.as_map()[list(exp.as_map().keys())[0]]:
        if feature not in sum_weights_map:
            sum_weights_map[feature] = 0

        # add weight to sum multiplied by class distribution

        sum_weights_map[feature] += weight


for feature in sum_weights_map:
    sum_weights_map[feature] /= samples
print(sum_weights_map)
```

```
100%|      | 1000/1000 [00:03<00:00, 278.98it/s]
```

```
{0: -0.07009810946835772, 8: -0.06613695622175278, 6: 0.03433966440320149, 1:
0.0281569052842955, 5: 0.023281229738779635, 9: 0.017333970138579333, 2:
0.00479825715110375, 7: -0.0003588962264027233, 3: 0.00027542229701660016, 4:
0.00015022972185573582}
```

[ ]:
```
import matplotlib.pyplot as plt
# bar plot using labels and weights
weights_with_labels = list(zip(sum_weights_map.values(), df.columns[:-1]))
weights_with_labels.sort(key=lambda x: x[0])

plt.figure(figsize=(10, 10))
plt.barh([x[1] for x in weights_with_labels], [x[0] for x in␣
 ↪weights_with_labels])
```

[ ]:
```
<BarContainer object of 10 artists>
```



[ ]:
```
explainer = lime.lime_tabular.LimeTabularExplainer(
    X_train,
    feature_names=df.columns[:-1],
    class_names=unique_Y,
    discretize_continuous=True,
)

for i in range(5):
    i = np.random.randint(0, len(X_test))

    exp = explainer.explain_instance(
        X_test[i],
        model_list_sklearn[0].predict_proba,
        num_features=5,
        top_labels=1,
```

```
        num_samples=1000,
    )

    exp.show_in_notebook(show_table=True, show_all=False)
```

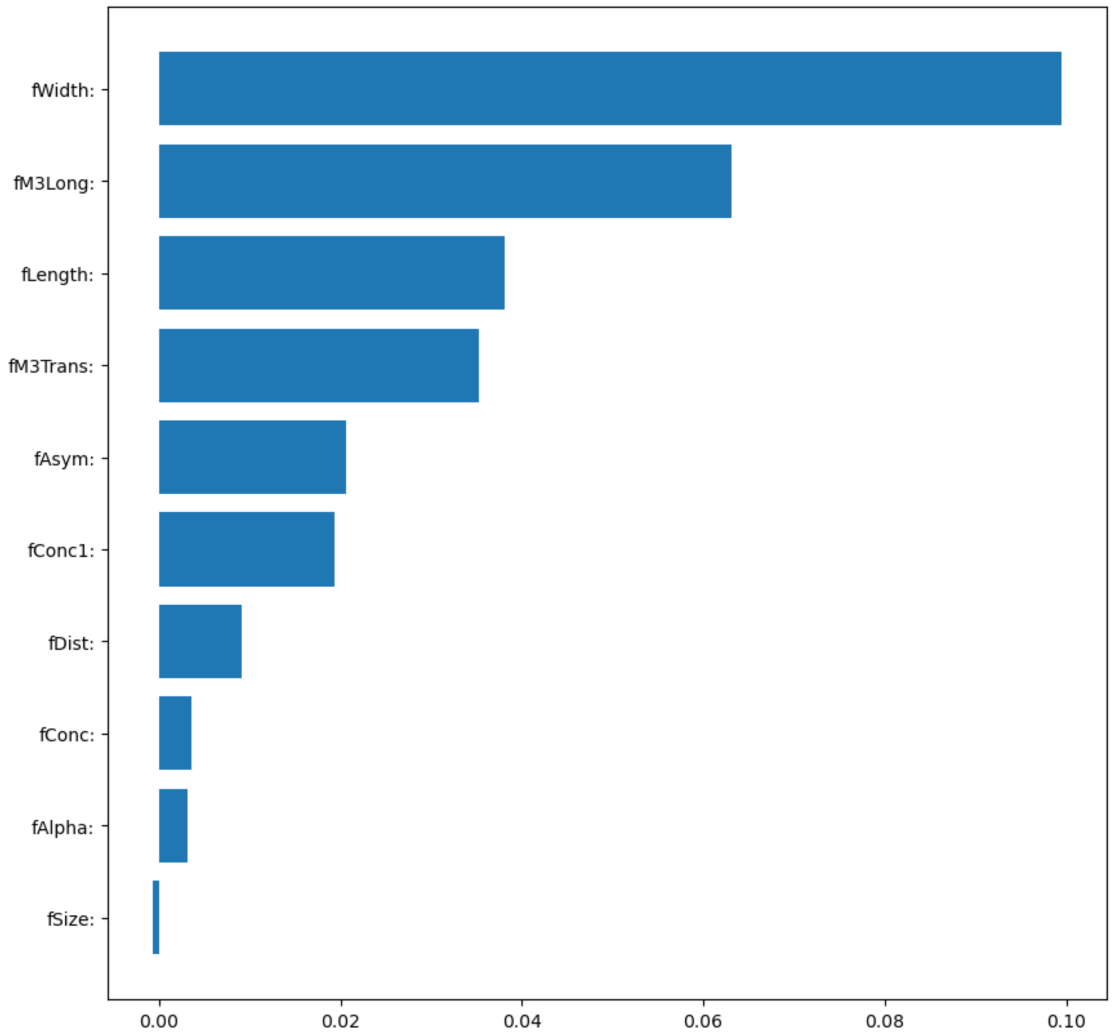
```
<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>
```

[ ]:
```
sum_weights_map = {}
#samples = len(X_test)
samples = 1000
for i in tqdm(range(samples)):
    exp = explainer.explain_instance(
        X_test[i],
        # lambda that converts to tensor
        lambda x: model_list_pytorch[0](torch.from_numpy(x).type(torch.
 ↪float32)).detach().numpy(),
        num_features=10,
        top_labels=1,
        num_samples=1000,
    )
    for feature, weight in exp.as_map()[list(exp.as_map().keys())[0]]:
        if feature not in sum_weights_map:
            sum_weights_map[feature] = 0
        sum_weights_map[feature] += weight

for feature in sum_weights_map:
    sum_weights_map[feature] /= samples
```

```
100%|          | 1000/1000 [00:12<00:00, 82.36it/s]
```

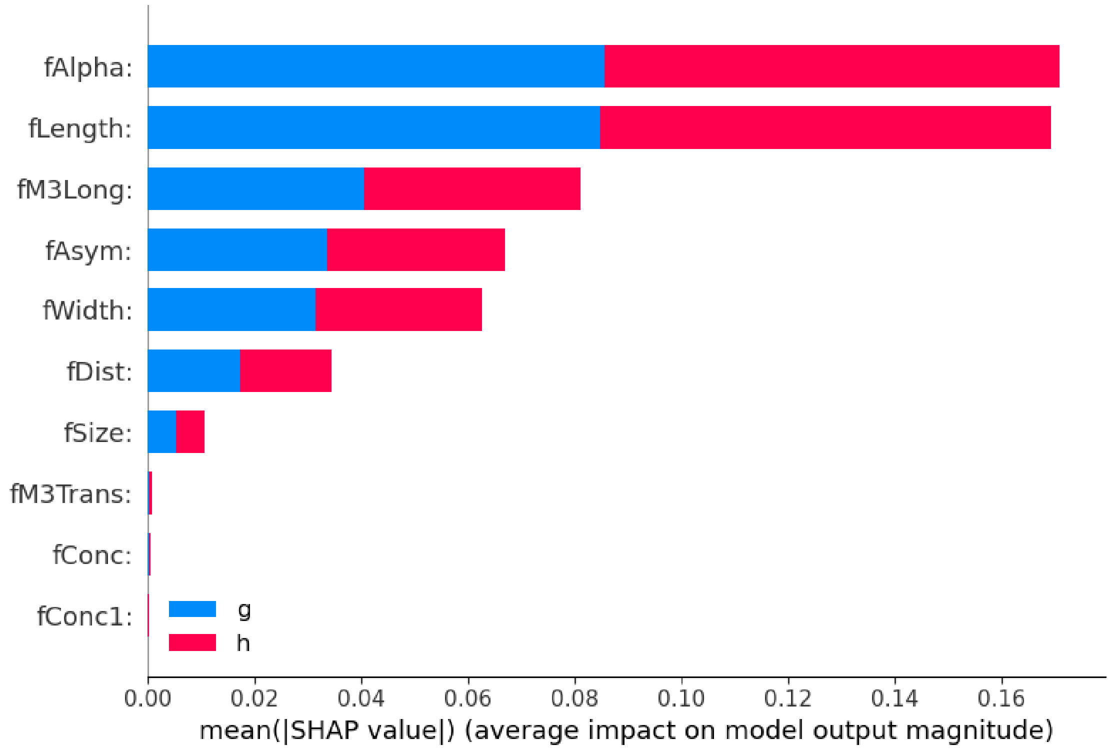
[ ]:
```
weights_with_labels = list(zip(sum_weights_map.values(), df.columns[:-1]))
weights_with_labels.sort(key=lambda x: x[0])

plt.figure(figsize=(10, 10))
plt.barh([x[1] for x in weights_with_labels], [x[0] for x in␣
 ↪weights_with_labels])
```

[ ]:
```
<BarContainer object of 10 artists>
```



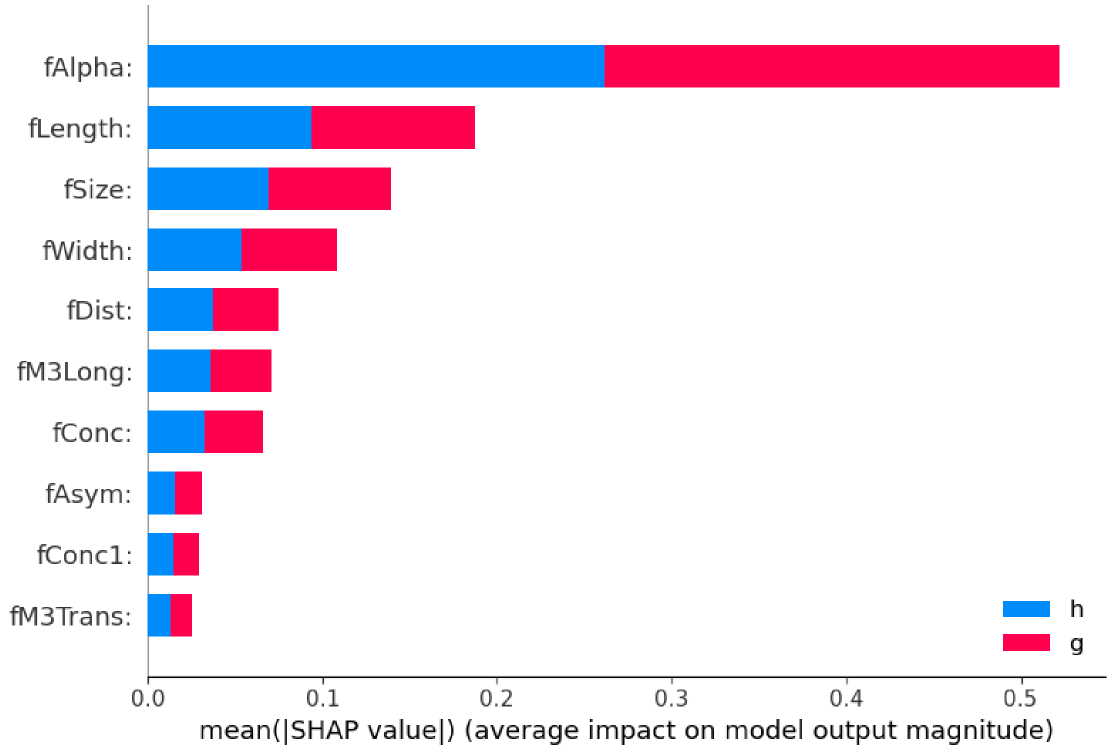
[ ]:
```
import shap

shap.initjs()
pred = model_list_sklearn[0].predict(X_test)
explainer = shap.KernelExplainer(model_list_sklearn[0].predict_proba, shap.
 ↪sample(X_test, 100))
shap_values = explainer.shap_values(shap.sample(X_test, 100))
shap.summary_plot(shap_values, X_test, feature_names=df.columns[:-1],␣
 ↪class_names=unique_Y)
```

```
<IPython.core.display.HTML object>

  0%|          | 0/100 [00:00<?, ?it/s]
```



[ ]:
```
# SHAP for PyTorch
pred = model_list_pytorch[0](torch.from_numpy(X_test).type(torch.float32)).
 ↪detach().numpy()
explainer = shap.KernelExplainer(lambda x: model_list_pytorch[0](torch.
 ↪from_numpy(x).type(torch.float32)).detach().numpy(), shap.sample(X_test,␣
 ↪100))
shap_values = explainer.shap_values(shap.sample(X_test, 100))
shap.summary_plot(shap_values, X_test, feature_names=df.columns[:-1],␣
 ↪class_names=unique_Y)
```

```
0%|          | 0/100 [00:00<?, ?it/s]
```


fAlpha:
fLength:
fSize:
fWidth:
fDist:
fM3Long:
fConc:
fAsym:
fConc1:
fM3Trans:
h
g
0.0
0.1
0.2
0.3
0.4
0.5
mean(|SHAP value|) (average impact on model output magnitude)