Multi-label Classification for abstract dataset

Pre experiment phase:

- 1) Preliminary analysis

Experiment phase:

- 2) Preprocessing
- 3) Metrics selection
- 4) Model training and evaluation
- 5) Hyperparameter tuning and best model selection

Post experiment phase:

- 6) Evaluation on test data

1) Preliminary data analysis:

- Print data
- Visualization
- Number of samples
- Number of features
- Number of classes
- Number of samples for each class

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn import preprocessing
```

```
#get raw data
raw_data = pd.read_pickle('/content/ass2.pickle')
X_train = raw_data['train'].iloc[:,0:42] # all features
y_train = raw_data['train']['target'] # target
label_encoder = preprocessing.LabelEncoder()
label_encoder.fit(y_train)
```

```
print(raw_data)
```

Visualization of data distribution for first impressions

Distribution of samples per class

```
# Count the number of samples for each class
class_counts = y_train.value_counts()

# Plot the bar chart
plt.bar(class_counts.index, class_counts.values)

# Set the title and labels
plt.title('Distribution of Samples by Class')
plt.xlabel('Class')
plt.ylabel('Number of Samples')

# Set the x-axis limits
plt.xlim(0, len(class_counts))

# Show the plot
plt.show()
```

Distribution of features

```
import matplotlib.pyplot as plt
# Iterate over each class
for label in y_train.unique():
    # Select data for the current class
    data = X_train[y_train == label].values.flatten()

    # Plot the histogram
    plt.hist(data, bins=20, alpha=0.5, label=f'Class {label}')

# Set the title and labels
plt.title('Distribution of Features by Class')
plt.xlabel('Feature Values')
plt.ylabel('Frequency')
plt.legend()

# Show the plot
plt.show()
```

We can clearly see that most of the samples belong to class 2, second most samples to class 1 and class 0 has less samples. Thus, from first impression we conclude that the data is imbalanced.

Now get the exact numbers of distribution:

```
print('Total number of (samples, features) in train-set is:', X_train.shape)
print('Labels of dataset are:', label_encoder.classes_ )
print('Number of samples of each lable is:')
print(y_train.value_counts(), '\n')

X_dev = raw_data['dev'].iloc[:,0:42]
y_dev = raw_data['dev']['target']
print('Total number of (samples, features) in dev-set is:', X_dev.shape)
print('Number of samples of each lable is:')
print(y_dev.value_counts(), '\n')

X_test = raw_data['test'].iloc[:,0:42]
y_test = raw_data['test']['target']
print('Total number of (samples, features) in test-set is:', X_test.shape)
print(y_test.value_counts(), '\n')
```

Preliminary data analysis shows that dataset is imbalanced.

2)Preprocessing

- balancing data
- Standard scaling

Explanation:

> Logistic Regression, KNN and SVM are sensitive to imbalanced and not standardized data thus, we balance and standardize X_train and X_dev.

> Tree classifiers aren't sensitive to imbalanced and not stadardized data.

Balancing the data

```
from imblearn.under_sampling import RandomUnderSampler
# Define the undersampler
undersampler = RandomUnderSampler(random_state=42)

# Fit and transform the training set
X_train_balanced, y_train_balanced = undersampler.fit_resample(X_train, y_train)
print('Now after balancing the number of samples of each lable in train-set is: \n')
print(y_train_balanced.value_counts(), '\n')
print('Total (samples, features) in train-set after balancing:',  X_train_balanced.shape)
```

Standard scaling

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(X_train)
X_train_transformed = scaler.transform(X_train)
scaler.fit(X_train_balanced)
```

```
X_train_transformed_balanced = scaler.transform(X_train_balanced)
scaler.fit(X_dev)
X_dev_transformed = scaler.transform(X_dev)
```

3) Metrics selection

We'll evaluate due to following metrices:

- accuracy
- weighted F1-score
- cross validation score
- classification report (precision/recall and F1-score)
- confusion matrix

Explanation:

> Since the data set is named only "generally" as feature names aren't named specifically but f1,f2,...,f42 and class labels are named as class 0,1,2 and not specific, we don't really know the datas contents and issues. Thus we don't know whether a high precision or a low recall is desirable in prediction. Hence, concluding that we'll use basically the harmonic F1-score of precision/recall for model evaluating. Specifically using F1-score weighted average because test data is imbalanced as well.

```
from sklearn.metrics import classification_report, f1_score,accuracy_score, confusion_matr
from sklearn.model_selection import cross_val_score,RepeatedStratifiedKFold
```

4) Model training and evaluation

We'll train the following models:

- Decision Tree
- Random Forest
- GBDT - Gradient Boost Decision Tree
- Logistic Regression
- K-nearest Neighbors

Hence the dataset is provided with labels/targets we won't train a clustering models.

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
```

```
non_sensitive_models = [
          ('Decision Tree',DecisionTreeClassifier(random_state=42)),
          ('Random Forrest', RandomForestClassifier(random_state=42)),
           ('Gradien Boost Decision Tree',GradientBoostingClassifier(learning_rate=1, rand
        ]

sensitive_models = [
          ('Logistic Regression', LogisticRegression(max_iter=1000)),
          ('KNN', KNeighborsClassifier())
        ]

target_names = ['0', '1', '2']

def eval_model(model,cv_scores, x_eval, y_eval, accuracies):
  y_dev_pred = clf.predict(x_eval)
  accuracy = accuracy_score(y_eval, y_dev_pred)
  if (accuracies.get('model') is None) or accuracy > accuracies.get('model'):
    accuracies[model] = accuracy
  #print results
  print(f'Classifier                       : {model}')
  print(f'Accuracy                         : {accuracy}')
  print(f'Weighted F1-score is             : {f1_score(y_eval, y_dev_pred ,average= "weigh
  print(f'cross-validation mean accuracies : {cv_scores.mean()}')
  print(f'classification report\n {classification_report(y_eval, y_dev_pred, target_names=
  print(f'confusion matrix\n {confusion_matrix(y_eval, y_dev_pred)}', '\n\n')
```

4.1) Training Decision Tree, Random Forrest and Gradient Boost with imbalanced data

```
accuracies_non_sensitive_models_imbalanced = {}
#train with 5-cross validation
print('Training and evaluating models with cross validation \n\n')
for _,model in non_sensitive_models:
        clf = model.fit(X_train, y_train)
        cv_scores = cross_val_score(clf, X_train, y_train,scoring='accuracy', cv=5, n_jobs
        eval_model(clf,cv_scores, X_dev, y_dev, accuracies_non_sensitive_models_imbalanced

print(f'Accuracies are: {accuracies_non_sensitive_models_imbalanced}\n')
```

4.2) Training Decision Tree, Random Forrest and Gradient Boost with balanced data

```
accuracies_non_sensitive_models_balanced={}
#train with undersampled balanced data and 5-cross validation
print('Training and evaluating models after undersampling and with cross validation \n\n')
for _, model in non_sensitive_models:
        clf = model.fit(X_train_balanced, y_train_balanced)
        cv_scores = cross_val_score(clf, X_train_balanced, y_train_balanced ,scoring='accu
        eval_model(clf, X_dev, y_dev,accuracies_non_sensitive_models_balanced)

print(f'Accuracies are: {accuracies_non_sensitive_models_balanced}\n')

#train with repeated stratified 5-fold cross validation
```

```
rep_strat_kfold = RepeatedStratifiedKFold(n_repeats=3, n_splits=5, random_state=42)
print('Training and evaluating models with repeated stratified 5-fold cross validation \n\
for _, model in sensitive_models:
        clf = model.fit(X_train, y_train)
        cv_scores = cross_val_score(clf, X_train, y_train ,scoring='accuracy', cv=rep_stra
        eval_model(clf, X_train, y_dev,accuracies_non_sensitive_models_balanced)
print(f'Accuracies are: {accuracies_non_sensitive_models_balanced}\n')
```

We see that the tree algorithms performed worse on the balanced data after undersampling, even when tree algorithms are generally not sensitive to imbalanced or balanced data. This may be a result of the "nature" of given data and altering the original data distribution may lead to decrease of accuracy which may be caused due to loss of information, class noise etc. after the undersampling.

4.3) Training separately classifiers sensitive to imbalanced and not stadardized data.

```
accuracies_sensitive_models_={}
#train with undersampled balanced data and 5-cross validation
print('Training and evaluating models after undersampling and with cross validation \n\n')
for _, model in sensitive_models:
        clf = model.fit(X_train_transformed_balanced, y_train_balanced)
        cv_scores = cross_val_score(clf, X_train_transformed_balanced, y_train_balanced ,s
        eval_model(clf, X_dev_transformed, y_dev,accuracies_sensitive_models_ , cv=5)
print(f'Accuracies are: {accuracies_sensitive_models_}\n')

#train with repeated stratified 5-fold cross validation
rep_strat_kfold = RepeatedStratifiedKFold(n_repeats=3, n_splits=5, random_state=42)
print('Training and evaluating models with repeated stratified 5-fold cross validation \n\
for _, model in sensitive_models:
        clf = model.fit(X_train_transformed, y_train)
        cv_scores = cross_val_score(clf, X_train_transformed_balanced, y_train_balanced ,s
        eval_model(clf, X_dev_transformed, y_dev,accuracies_sensitive_models_)
print(f'Accuracies are: {accuracies_sensitive_models_}\n')
```

We see that KNN and Logistic Regression both performed much better with repeated stratified 5-fold cross validation then with the undersampled data. This could be explained since repeated stratified k-fold cross validation:

- provides a more comprehensive evaluation by considering multiple iterations of training and testing on different subsets of the data.
- reduces the potential bias introduced by undersampling, where some information from the majority class may be lost.
- allows the models to learn from a wider range of samples, including both majority and minority class instances, which can lead to better performance.

5) Hyperparamter tuning and best model selection

5.1) Tuning hyperparameters of best 2 models of section 4:

Explanation:

> We found Random Forest Classifier and Gradien Boost Classifier performing best on dev-set. Thus, tuning hyperparameters of both, trying to improve performance on dev-set and finally choosing best classifier on dev-set.

```
overall_accuracies = [accuracies_non_sensitive_models_imbalanced, accuracies_non_sensitive
def get_best_classifiers(overall_accuracies):
    top_models = []
    top_accuracies = [0.0, 0.0]

    for dictionary in overall_accuracies:
        for model, accuracy in dictionary.items():
            # Update the top two models if the current model has a higher accuracy
            if accuracy > top_accuracies[0]:
                top_models.insert(0, model)
                top_accuracies.insert(0, accuracy)
            elif accuracy > top_accuracies[1]:
                top_models.insert(1, model)
                top_accuracies.insert(1, accuracy)

    return top_models[:2], top_accuracies[:2]


for accuracy in overall_accuracies:
  print(f'Accuracies of non sensitive models with imbalanced data: {accuracy}')

best_models, best_accuracies = get_best_classifiers(overall_accuracies)
# Print maximum accuracy and corresponding model
print("\n\n2 best accuracies  :", best_accuracies)
print("2 best classifiers :", best_models)
```

Tuning following hyperparamter with optuna:

- n_estimators
- max_depth
- max_features
- min_samples_split
- min_samples_leaf
- criterion

Explanation of hyperparameters and how they may affect performance:

- n_estimators:

  > Increasing number of Decision Trees can improve models perfomance by reducing overfitting, but can also increase training time.

- max_depth:

  > It controls the maximum depth of each decision tree. A deeper tree can capture more complex relationships in the data, but it may also lead to overfitting. Setting an appropriate max_depth value helps balance model complexity and generalization.

- max_features:

  > It determines the maximum number of features to consider when looking for the best split at each node. Limiting the number of features can improve the diversity and randomness of the trees, reducing the likelihood of overfitting.

- min_samples_split:

  > It specifies the minimum number of samples required to split an internal node during the construction of each decision tree. Increasing this value can prevent overfitting by enforcing a minimum number of samples required for a split.

- min_samples_leaf:

  > It sets the minimum number of samples required to be at a leaf node. Similar to min_samples_split, increasing this value helps prevent overfitting by requiring a minimum number of samples in each leaf.

- criterion:

  > It defines the function used to measure the quality of a split. Gini impurity measures the probability of misclassifying a randomly chosen element, while entropy measures the information gain in terms of the reduction in uncertainty.

- learning_rate:

  > Determines the contribution of each tree to the final ensemble. Smaller learning rates can lead to better generalization and more robust models, as they prevent overfitting by reducing the impact of individual trees. Higher

> learning rates can lead to faster convergence and improved training set performance, but they may also increase the risk of overfitting.

```
!pip install optuna
import optuna


best_model = None
best_params = None
best_accuracy = 0.0

def objective(trial):
    global best_model
    global best_params
    global best_accuracy

    # Define the hyperparameter search spaces for the first model (Random Forest)
    rf_params = {
        'n_estimators': trial.suggest_int('n_estimators', 100, 500, step=100),
        'max_depth': trial.suggest_int('max_depth', 5, 20),
        'min_samples_split': trial.suggest_int('min_samples_split', 2, 10),
        'min_samples_leaf': trial.suggest_int('min_samples_leaf', 1, 10),
        'max_features': trial.suggest_categorical('max_features', ['sqrt', 'log2']),
        'criterion': trial.suggest_categorical('criterion', ['gini', 'entropy'])
    }

    rf_model = best_models[0]
    rf_model.set_params(**rf_params)


    # Define the hyperparameter search spaces for the second model (Gradient Boosting)
    gb_params = {
    'n_estimators': trial.suggest_int('gb_n_estimators', 100, 500, step=100),
    'learning_rate': trial.suggest_loguniform('gb_learning_rate', 0.001, 0.1),
    'max_depth': trial.suggest_int('gb_max_depth', 3, 10),
    'min_samples_split': trial.suggest_int('gb_min_samples_split', 2, 10),
    'min_samples_leaf': trial.suggest_int('gb_min_samples_leaf', 1, 10),
    'max_features': trial.suggest_categorical('gb_max_features', ['sqrt', 'log2']),
}

    gb_model = best_models[1]
    gb_model.set_params(**gb_params)

    # Train and evaluate both models
    rf_model.fit(X_train, y_train)
    gb_model.fit(X_train, y_train)

    rf_accuracy = rf_model.score(X_dev, y_dev)
    gb_accuracy = gb_model.score(X_dev, y_dev)

    # Update the best model and parameters if necessary
    if rf_accuracy > best_accuracy:
        best_model = rf_model
        best_params = rf_params
```

```
            best_accuracy = rf_accuracy
        if gb_accuracy > best_accuracy:
            best_model = gb_model
            best_params = gb_params
            best_accuracy = gb_accuracy

        # Return the maximum accuracy of the two models
        return max(rf_accuracy, gb_accuracy)

study = optuna.create_study(direction='maximize')
study.optimize(objective, n_trials=100)
```

Visualization of hyperparamter tuning

```
from optuna.visualization import plot_optimization_history, plot_slice, plot_param_importa
plot_optimization_history(study)
# Visualize the hyperparameter slice
plot_slice(study)
# Visualize the importance of hyperparameters
plot_param_importances(study)
```

Explanation of model performance after hyperparamter tuning:

> We can clearly see that learning rate is the most important hyperparameter for increasing improvement of model performance, strongly followed by the second most important hyperparamter, max_depth.

5.2) Selecting best classifier

```
print(f'\nSelected best model is')
print(best_model)
```

Evaluation on dev-set of best model with best hyperparamters

```
# Refit best model with best hyperparameters
best_model.set_params(**best_params)
best_model.fit(X_train, y_train)
y_pred_dev = best_model.predict(X_dev)
cv_scores = cross_val_score(clf, X_train, y_train ,scoring='accuracy', cv=5, n_jobs=-1)
print(f'Best Hyperparameters  : {best_params}')
eval_model(best_model,cv_scores ,X_train, X_dev, y_dev,accuracies_sensitive_models_ , cv=r
```

6) Evaluation on test data

```
#evaluate best model on test set
y_test_pred = best_model.predict(X_test)
print(f'Final evaluation of {best_model} on test data\n')
print(f'Accuracy on test data              : {accuracy_score(y_test, y_test_pred)}')
print(f'Weighted F1-score on test data is  : {f1_score(y_test, y_test_pred ,average= "weig
print("classification report:\n", classification_report(y_test, y_test_pred, target_names=
```

Final analysis of performance on test-data:

> The best model, GradientBoostingClassifier(learning_rate=0.09759345772168124, max_depth=10,max_features='sqrt', min_samples_leaf=10,min_samples_split=6, n_estimators=500) , was evaluated on the test data. The test-set accuracy of the model was found to be 0.844, indicating that it performed well on real, unseen data. Not surprisingly the highest F1-score has been achieved for class 2, then class 1 and F1-score for class 0 is reasonable as well, which has least number of samples also in test-set Analyzing the classification report, we observe that the model exhibited varying performance for each class on the test data. The performance on the test data aligned with the performance on the dev-set, confirming the generalizability of our model.