RP Phase 3: MLSMOTE

After performing MLSMOTE on a toy dataset that I've created using the make_classification function embedded in python. The new dataset created as mentioned in the video composed of the feature dataset has 20 features and 1000 rows, and the target dataset composed of 5 classes and 1000 rows as well. From the parameters I passed to make_classification, I made sure that the data created Is indeed an imbalanced multi-class, multi-label dataset that fit the MLSMOTE pre-processing technique. Using SVM model (LinearSVC) and OneVsRestClassifier classifier for multilabel dataset. The results are quiet promising and I achieved slight improvements in accuracy after performing MLSMOTE with 100 new samples on the toy dataset: before MLSMOTE: 0.822 and after MLSMOTE: 0.845.

Given that our TP dataset (MoA dataset) is an imbalanced multi-labeled dataset, a preprocessing technique such as MLSMOTE is indeed crucial to apply. The way that MLSMOTE works, is that it generates and adds new synthetic instances that belongs to minority classes to the original dataset, in way that it converts an imbalanced labeled problem to a somehow balanced one (based on the number of the new samples generated as user input). Moreover, given that our data is somehow big with 200 feature and about 23k samples, it is very important to adequately tune MLSMOTE by trying different number of samples generated and get the best accuracy possible. Additionally, since the code posted for MLSMOTE only works for numerical data (as opposed to the paper when it said it works for numerical and categorical data applying different approaches to each one), I made hot encoding for the categorical features (only 2) in the original dataset. Below is how I applied MLMSOTE to our data along with the dependent functions and the hot encoding as well as generating a new data set that will be passed then to a classifier algorithm:

```
import numpy as np
import pandas as pd
import random
from sklearn.neighbors import NearestNeighbors

def get_tail_label(df):
    """
    Give tail label colums of the given target dataframe

    args
    df: pandas.DataFrame, target label df whose tail label has to identified

    return
    tail_label: list, a list containing column name of all the tail label
    """
    columns = df.columns
    n = len(columns)
    irpl = np.zeros(n)
    for column in range(n):
        irpl[column] = df[columns[column]].value_counts()[1]
    irpl = max(irpl)/irpl
```

```
    mir = np.average(irpl)
    tail_label = []
    for i in range(n):
        if irpl[i] > mir:
            tail_label.append(columns[i])
    return tail_label

def get_index(df):
  """
  give the index of all tail_label rows
  args
  df: pandas.DataFrame, target label df from which index for tail label has to identif
ied

  return
  index: list, a list containing index number of all the tail label
  """
  tail_labels = get_tail_label(df)
  index = set()
  for tail_label in tail_labels:
    sub_index = set(df[df[tail_label]==1].index)
    index = index.union(sub_index)
  return list(index)

def get_minority_instace(X, y):
    """
    Give minority dataframe containing all the tail labels
    args
    X: pandas.DataFrame, the feature vector dataframe
    y: pandas.DataFrame, the target vector dataframe
    return
    X_sub: pandas.DataFrame, the feature vector minority dataframe
    y_sub: pandas.DataFrame, the target vector minority dataframe
    """
    index = get_index(y)
    X_sub = X[X.index.isin(index)].reset_index(drop = True)
    y_sub = y[y.index.isin(index)].reset_index(drop = True)
    return X_sub, y_sub
```

```python
def nearest_neighbour(X):
    """
    Give index of 5 nearest neighbor of all the instance
    args
    X: np.array, array whose nearest neighbor has to find
    return
    indices: list of list, index of 5 NN of each element in X
    """
    nbs=NearestNeighbors(n_neighbors=5,metric='euclidean',algorithm='kd_tree').fit(X)
    euclidean,indices= nbs.kneighbors(X)
    return indices

def MLSMOTE(X,y, n_sample):
    """
    Give the augmented data using MLSMOTE algorithm
    args
    X: pandas.DataFrame, input vector DataFrame
    y: pandas.DataFrame, feature vector dataframe
    n_sample: int, number of newly generated sample
    return
    new_X: pandas.DataFrame, augmented feature vector data
    target: pandas.DataFrame, augmented target vector data
    """
    indices2 = nearest_neighbour(X)
    n = len(indices2)
    new_X = np.zeros((n_sample, X.shape[1]))
    target = np.zeros((n_sample, y.shape[1]))
    for i in range(n_sample):
        reference = random.randint(0,n-1)
        neighbour = random.choice(indices2[reference,1:])
        all_point = indices2[reference]
        nn_df = y[y.index.isin(all_point)]
        ser = nn_df.sum(axis = 0, skipna = True)
        print(ser)
        target[i] = np.array([1 if val>2 else 0 for val in ser])
        ratio = random.random()
        gap = X.loc[reference,:] - X.loc[neighbour,:]
        new_X[i] = np.array(X.loc[reference,:] + ratio * gap)
    new_X = pd.DataFrame(new_X, columns=X.columns)
    target = pd.DataFrame(target, columns=y.columns)
    return new_X, target
def dummy_ecoding_2_class(dfc, col):
       new_dfc = dfc
       first_class=pd.unique(new_dfc.iloc[:,col])[0]
       for i in range(len(dfc.iloc[:,col])):
```

```
            if dfc.iloc[i,col]==first_class:
                  new_dfc.iloc[i,col]=1
            else:
                  new_dfc.iloc[i,col]=0
  return new_dfc

df = pd.read_csv("train_features.csv")
X=dummy_ecoding_2_class(df,1)
X=dummy_ecoding_2_class(df,3)
#df_ohe.drop(columns = ['sig_id'], axis=1)
del X['sig_id']
pd.DataFrame(X)
y = pd.read_csv("train_targets_scored.csv")
del y['sig_id']
X_sub, y_sub = get_minority_instace(X, y) #Getting minority instance of that datframe
X_res,y_res =MLSMOTE(X_sub, y_sub, 2000) #Applying MLSMOTE to augment the dataframe
newX = pd.concat([X,X_res], ignore_index=True)
newX.to_csv('train_features_mlsmote_2000.csv')
newY = pd.concat([y,y_res], ignore_index=True)
newY.to_csv('train_targets_scored_mlsmote_2000.csv')
print(newX.shape)
print(newY.shape)
```

As I said, this is not a classification algorithm but in fact a preprocessing technique. As mentioned in the above code, 2000 new synthetic samples are generated and then concatenated with the original data to create a new datasets train_targets_scored_mlsmote_2000.csv (for targets) and train_features_mlsmote_2000.csv (for features). These two files will then be passed to a classification technique. Below I will present to you some of the classification technique we applied on the new dataset along with some measures (slightly without performing tunings).

SVM:

Results without MLSMOTE

```
Accuracy: 0.3648960739030023
F_score: 0.2666263603385731
Log-loss: 7.918932332451279
```

Results with MLSMOTE

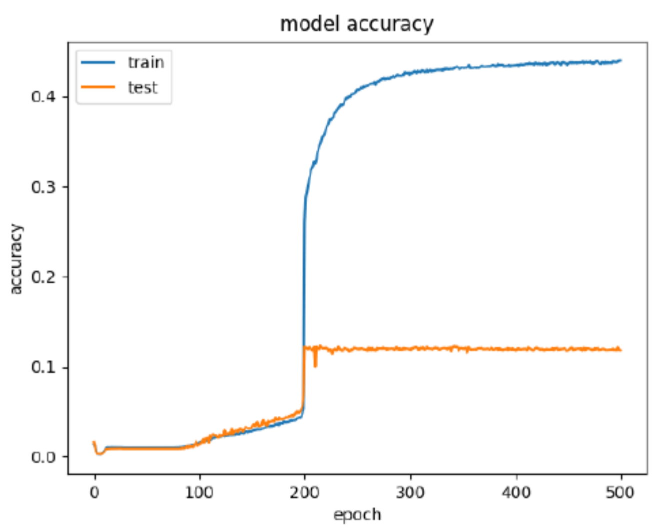
```
Accuracy: 0.3881464264962231
F-score micro: 0.26546091015169193
F-score macro: 0.13389851517631826
F-score weighted: 0.2574073194315111
Log-loss: 6.644603432093816
```

Adding 2000 minority samples to our data using MLSMOTE (alone, without feature selection) did not significantly improve SVM's performance for our problem. There is a slight improvement, but overall performance is still poor.

Neural Network:

Results without MLSMOTE
```
precision = 0.06045370204106063
recall = 0.3013126491646778
f1-score = 0.10070292636721671
AUC = 0.642629845433697
```
Results with MLSMOTE
```
precision = 0.3284118116520351
recall = 0.23407281001137656
f1-score = 0.2733311192294919
AUC = 0.6163926931866344
```

Plotting learning curves after MLSMOTE



From the **Accuracy graph** above we can see that the accuracy increases for the training set for around 0.4 as it was 0.03 which is a high improvement. However, you can realize that after increasing the number of epochs the function started to converge. Regarding the accuracy of the test set it is low and this show that not all features are important in this multi-label classification.

K-Fold Cross Validation:

In order to determine better the correct model performance, we applied the K-Fold Cross Validation method

```
def calculate_metrics(y_test, yhat):
      return
            {'accuracy':accuracy_score(y_test,yhat),'precision':precision_score(
            y_tes , yhat,'F1-score':f1_score(y_test, yhat, average='micro'),
            "ROC AUC":roc_auc_score(y_test)}
def evaluate_model_kfold(X, y, num_layers,X_shape, y_shape):
      results = pd.DataFrame(columns=['accuracy', 'precision', 'recall', 'F1
            score'])
      cv = RepeatedKFold(n_splits=10, n_repeats=1, random_state=1)
      for train_ix, test_ix in cv.split(X):
            X_train, X_test = np.asarray(X[train_ix], np.float64),
                  np.asarray(X[test_ix], np.floa
            y_train, y_test = np.asarray(y[train_ix], np.float64),
                  np.asarray(y[test_ix], np.floa
            model = get_model()
            history = model.fit(X_train, y_train, verbose=0, epochs=100)
            yhat = model.predict(X_test)
            yhat = yhat.round()
            metrics = calculate_metrics(y_test, yhat)
            print(metrics)
            results.append(metrics, ignore_index=True)
      return results
result = evaluate_model_kfold(X, y, layers_shape,X_shape, y_shape)
print(result.mean(axis=0))
```

Results Without MLSMOTE

{‘precision': 0.05142857142857143, 'recall': 0.38240574506283664, 'F1-score': 0.09066401816118048, 'ROC AUC': 0.6791522652977822}
{‘precision': 0.04487179487179487, 'recall': 0.33110976349302607, 'F1-score': 0.07903307519722082, 'ROC AUC': 0.6536724910364937}
{'precision': 0.07333333333333333, 'recall': 0.32660332541567694, 'F1-score': 0.11977351916376305, 'ROC AUC': 0.6561954394967663}
{'precision': 0.07359956385443642, 'recall': 0.31523642732049034, 'F1-score': 0.11933701657458565, 'ROC AUC': 0.6506680174350494}
{'precision': 0.0580566306783459, 'recall': 0.3413173652694611, 'F1-score': 0.09923398328690808, 'ROC AUC': 0.6611990904773544}
{'precision': 0.2117456896551724, 'recall': 0.22942206654991243, 'F1-score': 0.22022975623423927, 'ROC AUC': 0.6132144285115998}
{'precision': 0.07539353769676885, 'recall': 0.3230769230769231, 'F1-score': 0.12225705329153605, 'ROC AUC': 0.6546889783184678}
{'precision': 0.07528044330314908, 'recall': 0.3253504672897196, 'F1-score': 0.12226978377785094, 'ROC AUC': 0.6556760888458321}
{'precision': 0.07070173120126867, 'recall': 0.3197848176927675, 'F1-score': 0.11580086580086582, 'ROC AUC': 0.6526994741249259}
{'precision': 0.3207190160832545, 'recall': 0.20311563810665068, 'F1-score': 0.2487160674981658, 'ROC AUC': 0.6008233928774762}

Results With MLSMOTE

{'precision': 0.06350307113136516, 'recall': 0.36092342342342343, 'F1-score': 0.10800336983993258, 'ROC AUC': 0.6732315077699389}
{'precision': 0.03738830928967204, 'recall': 0.42292490118577075, 'F1-score': 0.06870299027701339, 'ROC AUC': 0.6967130391467179}
{'precision': 0.33148295003965106, 'recall': 0.22340994120791022, 'F1-score': 0.26692209450830146, 'ROC AUC': 0.611060101482553}
{'precision': 0.5037406483790524, 'recall': 0.22185612300933552, 'F1-score': 0.3080442241707968, 'ROC AUC': 0.6106236270108628}
{'precision': 0.0723649711588883, 'recall': 0.3758169934640523, 'F1-score': 0.1213613578401196, 'ROC AUC': 0.681140572636839}
{'precision': 0.06772986808426856, 'recall': 0.3765736179529283, 'F1-score': 0.1148101793909053, 'ROC AUC': 0.6810407537534127}
{'precision': 0.06839503682033693, 'recall': 0.3796192609182531, 'F1-score': 0.11590734250790666, 'ROC AUC': 0.6827438309181931}
{'precision': 0.04388467374810319, 'recall': 0.3968166849615807, 'F1-score': 0.07902934907361864, 'ROC AUC': 0.6863556509782744}
{'precision': 0.06806063774176686, 'recall': 0.3661417322834646, 'F1-score': 0.1147844485585824, 'ROC AUC': 0.6762507507032066}
{'precision': 0.23551401869158878, 'recall': 0.21224031443009544, 'F1-score': 0.22327229769639692, 'ROC AUC': 0.6051813732730733}

We can realize that even after using MLSMOTE and increasing the size of the input, the overall model still not well trained

The above results are applied before MLSMOTE and after MLSMOTE (with 2000 synthetic samples), without feature selection and without model tuning.

The codes of all the above operation can be found in:
https://mega.nz/file/0nASkDAC#CW7hw11mFSwHGsZQkMbyPisk22cwklBlahKdIPfBQas
Or simply refer to TP Phase 3 Report (our team submission for term project phase 3).

The file shared above describes how the models are applied along with specifying the choice of parameters for each. Also, it is meant to help the users actually refer to the codes and convince them of the choices. As it is known, SVM, Neural Network and K-fold cross validation are best fit for classification. However, since our problem is a bit complicated (multi-class, multi-labeled) and requires materials beyond this course, the accuracies for all the models fall low.

MLSMOTE comes to convert an imbalanced dataset to more balanced one, but it is not meant to highly increase the performance of any model applied then on the new dataset. This was the main reason why I applied MLSMOTE on our MoA original dataset. Given all the results above, the accuracy after performing MLSMOTE is slightly better although it is low. Applying feature selection and feature engineering will probably increase accuracy especially when performing MLSMOTE with adjusting the number of samples created (tuning MLSMOTE). Also grid search will help to improve accuracy.

Using my RP findings (MLSOMTE) on my TP dataset (MoA), the models perform to some extent better without applying feature selection and tuning the models. Additionally, it is preferably to perform feature engineering, feature crossing (to increase the number of features) since our problem requires more features (originally 200) given the number of samples (originally 23k). Then, the results will go better if feature selection and tuning the models are then applied to our data.