CMPS 396X TP Phase 3

MLSMOTE

Given our dataset, and after doing exploratory data analysis, we discovered that our dataset represents an imbalanced multi-label classification problem. This problem is hard to be contained or accurately predicted by any model without doing a very specific data preprocessing. To tackle our problem we used the work in “MLSMOTE: Approaching imbalanced multilabel learning through synthetic instance generation”. MLSMOTE, a data preprocessing and over sampling algorithm, stands for Multilabel Synthetic Minority Over-sampling Technique which is exactly what we need in our project to tackle the problem of imbalanced labels and minority cases. 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). Then applying any classification model on the new dataset will results in a better performance and higher accuracy. The original code of MLSMOTE is written in Python is available online through: https://gist.github.com/niteshsukhwani/6be61cf42d14eb78c52397f9ceacb569 Applying MLSMOTE as it is didn’t work. In the paper, the authors stated that it also works for categorical data, however, when applying the code on our data (which is a mix of categorical and numerical data) it didn’t work because of the constraints of needing numerical input. As a result, we added the code of doing hot encoding for the categorical features and generate a new dataset which will be passed then to MLSMOTE. The resulted dataset (after applying MLSMOTE) will therefore be passed to a classification model and the results is slightly better. And as we can noticed, the higher the number of samples generated the better accuracy we get.

```
#MLSMOTE

# -*- coding: utf-8 -*-
# Importing required Library
import numpy as np
import pandas as pd
import random
from sklearn.datasets import make_classification
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 identified

      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 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

    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

    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)
        # new_X = pd.concat([X, new_X], axis=0)
        # target = pd.concat([y, target], axis=0)
```

```
        # target = pd.concat([y, target], axis=0)
        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_200.csv')
    newY = pd.concat([y,y_res], ignore_index=True)
    newY.to_csv('train_targets_scored_mlsmote_200.csv')
    print(newX.shape)
    print(newY.shape)
```

Gradient Boost (on original data without feature selection)

In this notebook, we aim to solve our term project's problem using a Gradient Boost Trees model. For this, we tune and fit a Gradient Boost model to our initial dataset using techniques like Grid Search and Cross-Validation, assess the model's performance using various metrics, and finally plot and analyze its learning curve.

Then, to see whether feature selection increases our model performance, we repeat these same steps on our data after applying feature selection.

Imports

```
import pandas as pd
import numpy as np
from sklearn.multioutput import MultiOutputClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import multilabel_confusion_matrix, ConfusionMatrixDisplay
from sklearn.datasets import make_multilabel_classification
from sklearn.preprocessing import OneHotEncoder
from sklearn.utils import shuffle
from sklearn.metrics import accuracy_score
from sklearn.model_selection import RepeatedKFold
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import GridSearchCV
```

The Dataset

```
X = pd.read_csv("./Data/train_features.csv")
y = pd.read_csv("./Data/train_targets_scored.csv")
```

```
#making sure X and y are same length, plus that drugs are stored in same order in both X and
#indeed they are
print(all(X["sig_id"] == y["sig_id"]))

#drop sig_id col from both X and y
X = X.drop("sig_id", 1)
y = y.drop("sig_id", 1)

#encode categorical cols, X.cp_type and X.cp_dose, into new cols
X['cp_type_trt_cp'] = X['cp_type'].map( {'trt_cp':1, 'ctl_vehicle':0} )
X['cp_dose_D1'] = X['cp_dose'].map( {'D1':1, 'D2':0} )
```

```
#delete the original cols
X = X.drop("cp_type", 1)
X = X.drop("cp_dose", 1)
```

```
True
```

Train-Test Split

We shuffle the data and split it into 80% train and 20% test datasets. Also when splitting, we make sure that the training set contains at least one element from each possible output class, so that no class gets completely left out when fitting the model.

```
#shuffle X and y (together)
X_shuf, y_shuf = shuffle(X, y)

#trying a random split for now
X_train, X_test, y_train, y_test = train_test_split(X_shuf, y_shuf, test_size=0.2, random_sta

#making sure the train part has at least one instance of each class
while any(y_train.sum() == 0):
    X_train, X_test, y_train, y_test = train_test_split(X_shuf, y_shuf, test_size=0.2)
```

The Model, Grid Search and Cross-Validation

Now, we build our model, which is a `MultiOutputClassifier` using `GradientBoostingClassifier`'s as base estimators. So, our model is fitting about 200 Gradient Boost models, one for each possible output class. To tune hyperparameters of the base estimators, we use Grid Search to find the best paramaters out of a list of possible combinations. The parameters considered in Grid Search are:

loss (loss function):["deviance", "exponential"]

learning_rate: [0.005, 0.01, 0.1]

n_estimators: [50, 100, 200]

To evaluate performance of each combination and see which is best, we combine our grid search with repeated K-fold cross-validation (so we use `GridSearchCV`), where the number of repeats is five and the numner of repeats is two. The parameter values found to produce best performance are:

[Not found - took too long]

Sadly, the grid search is taking a very long time, and did not finish quickly enough for the deadline. So, for all code beyond this point, we did not manage to get results on time, but anyway we left the code we would have used below.

```
#gradient boost classifier
gboost = MultiOutputClassifier(GradientBoostingClassifier(random_state=0), n_jobs=1)
```

```
# lowered the number of splits and repeats, just to finish faster
#ideally we would want more repeats (say 5 or 10)
cv = RepeatedKFold(n_splits=5, n_repeats=2, random_state=1)
parameters = {'estimator__loss':["deviance", "exponential"], 'estimator__learning_rate': [0.0
              'estimator__n_estimators': [50, 100, 200]}
clf = GridSearchCV(gboost, parameters, cv=cv)

clf.fit(X_train, y_train)
```

The obtained best parameter values

```
#dumping the classifier for later use
import pickle

with open("gradient_boost_clf.pickle", "wb") as f:
    pickle.dump(clf, f)

#print the best params returned by grid search
print("Best params:", clf.best_params_)
```

Performance metrics

We now evaluate the performance of our model, with the best found hyperparameter values, using accuracy and f-score metrics. As our problem is multilabel, the AUC metric is not supported. Nonetheless, we did include code we would use to get AUC if we had a compatible problem. We also included the value of the log-loss metric, as this is the performance metric used in the corresponding Kaggle challenge for this dataset.

Again we did not manage to get any results on time.

```
from sklearn.metrics import f1_score
from sklearn.metrics import roc_curve
from sklearn.metrics import auc
from sklearn.metrics import log_loss

#predict
y_test_pred = clf.predict(X_test)

#Accuracy
print("Accuracy:", accuracy_score(y_true=y_test, y_pred=y_test_pred))

#F-score
print("F-score:", f1_score(y_true=y_test, y_pred=y_test_pred, average="micro"))
```

```
print("F_score: ", f1_score(y_true=y_test, y_pred=y_test_pred, average="micro"))

#AUC (will not work coz multilabel)
#fpr, tpr, thresholds = roc_curve(y_test, y_test_pred, pos_label=2)
#print("AUC:", auc(fpr, tpr))

#log-loss (metric used in the Kaggle competition). Lower is better
#Log-loss should give much better value when predictions are probabilistic
print("Log-loss:", log_loss(y_true=y_test, y_pred=y_test_pred))



print("F_score (macro):", f1_score(y_true=y_test, y_pred=y_test_pred, average="macro"))
```

Learning Curve

We now plot the a learning curve for our Gradient-Boost-based model. Looking at the plot below, we see that the learning curve obtained shows pretty high variance, as accuracy for the training set is much higher than the one for the validation set. So we conclude that our model suffers from high variance and is overfitting.

```
#plotting learning curve
#adapted from https://scikit-learn.org/stable/auto_examples/model_selection/plot_learning_cur
from sklearn.model_selection import learning_curve
from matplotlib import pyplot as plt

#Replace "..." with the optimal params we should obtain from grid search
best_model = GradientBoostingClassifier(random_state=0, ...)
best_clf = MultiOutputClassifier(best_model, n_jobs=2)

train_sizes, train_scores, test_scores = \
learning_curve(best_clf, X_train, y_train, cv=cv, n_jobs=2, return_times=False)



train_scores_mean = np.nanmean(train_scores, axis=1)
train_scores_std = np.nanstd(train_scores, axis=1)
test_scores_mean = np.nanmean(test_scores, axis=1)
test_scores_std = np.nanstd(test_scores, axis=1)

_, axes = plt.subplots(1, 1, figsize=(20, 5))

plt.grid()
plt.fill_between(train_sizes, train_scores_mean - train_scores_std,
                     train_scores_mean + train_scores_std, alpha=0.1,
                     color="r")
plt.fill_between(train_sizes, test_scores_mean - test_scores_std,
                     test_scores_mean + test_scores_std, alpha=0.1,
                     color="g")
plt.plot(train_sizes, train_scores_mean, 'o-', color="r",
```

```
                label="Training score")
    plt.plot(train_sizes, test_scores_mean, 'o-', color="g",
                label="Cross-validation score")
    plt.legend(loc="best")

    plt.show()
```

SVM (on original data without feature selection)

In this part of TP phase 3, we aim to use an SVM model to solve our project's problem. We first tune and fit the SVM to our initial dataset, assess performance and plot the resulting learning curve, and then re-apply the same process for our MLSMOTE-oversampled dataset (more details further below in theis notebook). We also repeat this same process to our dataset after feature selection (but without applying MLSMOTE) in a separate notebook.

Imports

```
import pandas as pd
import numpy as np
from sklearn.multioutput import MultiOutputClassifier
from sklearn.svm import LinearSVC
from sklearn.model_selection import train_test_split
from sklearn.metrics import multilabel_confusion_matrix, ConfusionMatrixDisplay
from sklearn.datasets import make_multilabel_classification
from sklearn.preprocessing import OneHotEncoder
from sklearn.utils import shuffle
from sklearn.metrics import accuracy_score
from sklearn.model_selection import RepeatedKFold
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import GridSearchCV
```

The Dataset

Again for this part, we are using our original dataset, before feature selection and MLSMOTE.

```
X = pd.read_csv("./Data/train_features.csv")
y = pd.read_csv("./Data/train_targets_scored.csv")
```

```
#making sure X and y are same length, plus that drugs are stored in same order in both X and
#indeed they are
print(all(X["sig_id"] == y["sig_id"]))

#drop sig_id col from both X and y
X = X.drop("sig_id", 1)
y = y.drop("sig_id", 1)

#encode categorical cols, X.cp_type and X.cp_dose, into new cols
X['cp_type_trt_cp'] = X['cp_type'].map( {'trt_cp':1, 'ctl_vehicle':0} )
X['cp_dose_D1'] = X['cp_dose'].map( {'D1':1, 'D2':0} )
```

```
#delete the original cols
X = X.drop("cp_type", 1)
X = X.drop("cp_dose", 1)
```

```
True
```

Train-Test Split

Shuffling the dataset before splitting, we use 80% of the data for training and 20% for test. We also make sure that for our training dataset, we have at least one sample from each possible output class.

```
#shuffle X and y (together)
X_shuf, y_shuf = shuffle(X, y)

#trying a random split for now
X_train, X_test, y_train, y_test = train_test_split(X_shuf, y_shuf, test_size=0.2, random_sta

#making sure the train part has at least one instance of each class
while any(y_train.sum() == 0):
    X_train, X_test, y_train, y_test = train_test_split(X_shuf, y_shuf, test_size=0.2)
```

The Model, Grid Search and Cross-Validation

Now, we build our model, which is a `MultiOutputClassifier` using `LinearSVC`'s as base estimator. So, our model is fitting about 200 SVM models, one for each possible output class. To tune hyperparameters of the base estimators, we use Grid Search to find the best paramaters out of a list of possible combinations. The parameters considered in Grid Search are:

C: {0.1, 1, 10}

loss: {"hinge", "square_hinge"}

To evaluate performance of each combination and see which is best, we combine our grid search with repeated K-fold cross-validation (so we use `GridSearchCV`), where the number of repeats is five and the numner of repeats is two. The parameter values found to produce best performance are:

C: 0.1

loss: "hinge"

```
#initializing the SVM, and making it multilabel using MultiOutputClassifier
multi_svm = MultiOutputClassifier(LinearSVC(random_state=42), n_jobs=1)

# lowered the number of splits and repeats, just to finish faster
cv = RepeatedKFold(n_splits=5, n_repeats=2, random_state=1)
parameters = {'estimator__C':[0.1, 1, 10], 'estimator__loss': ['hinge', 'squared_hinge']}
```

```
clf = GridSearchCV(multi_svm, parameters, cv=cv)

clf.fit(X_train, y_train)
```

```
C:\Users\Anis\Anaconda3\lib\site-packages\sklearn\svm\_base.py:977: ConvergenceWarnin
  "the number of iterations.", ConvergenceWarning)
C:\Users\Anis\Anaconda3\lib\site-packages\sklearn\svm\_base.py:977: ConvergenceWarnin
  "the number of iterations.", ConvergenceWarning)
C:\Users\Anis\Anaconda3\lib\site-packages\sklearn\svm\_base.py:977: ConvergenceWarnin
  "the number of iterations.", ConvergenceWarning)
C:\Users\Anis\Anaconda3\lib\site-packages\sklearn\svm\_base.py:977: ConvergenceWarnin
  "the number of iterations.", ConvergenceWarning)
C:\Users\Anis\Anaconda3\lib\site-packages\sklearn\svm\_base.py:977: ConvergenceWarnin
  "the number of iterations.", ConvergenceWarning)
C:\Users\Anis\Anaconda3\lib\site-packages\sklearn\svm\_base.py:977: ConvergenceWarnin
  "the number of iterations.", ConvergenceWarning)
C:\Users\Anis\Anaconda3\lib\site-packages\sklearn\svm\_base.py:977: ConvergenceWarnin
  "the number of iterations.", ConvergenceWarning)
C:\Users\Anis\Anaconda3\lib\site-packages\sklearn\svm\_base.py:977: ConvergenceWarnin
  "the number of iterations.", ConvergenceWarning)
C:\Users\Anis\Anaconda3\lib\site-packages\sklearn\svm\_base.py:977: ConvergenceWarnin
  "the number of iterations.", ConvergenceWarning)
C:\Users\Anis\Anaconda3\lib\site-packages\sklearn\svm\_base.py:977: ConvergenceWarnin
  "the number of iterations.", ConvergenceWarning)
C:\Users\Anis\Anaconda3\lib\site-packages\sklearn\svm\_base.py:977: ConvergenceWarnin
  "the number of iterations.", ConvergenceWarning)
C:\Users\Anis\Anaconda3\lib\site-packages\sklearn\svm\_base.py:977: ConvergenceWarnin
  "the number of iterations.", ConvergenceWarning)
C:\Users\Anis\Anaconda3\lib\site-packages\sklearn\svm\_base.py:977: ConvergenceWarnin
  "the number of iterations.", ConvergenceWarning)
C:\Users\Anis\Anaconda3\lib\site-packages\sklearn\svm\_base.py:977: ConvergenceWarnin
  "the number of iterations.", ConvergenceWarning)
C:\Users\Anis\Anaconda3\lib\site-packages\sklearn\svm\_base.py:977: ConvergenceWarnin
  "the number of iterations.", ConvergenceWarning)
C:\Users\Anis\Anaconda3\lib\site-packages\sklearn\svm\_base.py:977: ConvergenceWarnin
  "the number of iterations.", ConvergenceWarning)
C:\Users\Anis\Anaconda3\lib\site-packages\sklearn\svm\_base.py:977: ConvergenceWarnin
  "the number of iterations.", ConvergenceWarning)
C:\Users\Anis\Anaconda3\lib\site-packages\sklearn\svm\_base.py:977: ConvergenceWarnin
  "the number of iterations.", ConvergenceWarning)
C:\Users\Anis\Anaconda3\lib\site-packages\sklearn\svm\_base.py:977: ConvergenceWarnin
  "the number of iterations.", ConvergenceWarning)
C:\Users\Anis\Anaconda3\lib\site-packages\sklearn\svm\_base.py:977: ConvergenceWarnin
  "the number of iterations.", ConvergenceWarning)
C:\Users\Anis\Anaconda3\lib\site-packages\sklearn\svm\_base.py:977: ConvergenceWarnin
  "the number of iterations.", ConvergenceWarning)
C:\Users\Anis\Anaconda3\lib\site-packages\sklearn\svm\_base.py:977: ConvergenceWarnin
  "the number of iterations.", ConvergenceWarning)
C:\Users\Anis\Anaconda3\lib\site-packages\sklearn\svm\_base.py:977: ConvergenceWarnin
  "the number of iterations.", ConvergenceWarning)
C:\Users\Anis\Anaconda3\lib\site-packages\sklearn\svm\_base.py:977: ConvergenceWarnin
  "the number of iterations.", ConvergenceWarning)
C:\Users\Anis\Anaconda3\lib\site-packages\sklearn\svm\_base.py:977: ConvergenceWarnin
  "the number of iterations.", ConvergenceWarning)
C:\Users\Anis\Anaconda3\lib\site-packages\sklearn\svm\_base.py:977: ConvergenceWarnin
  "the number of iterations.", ConvergenceWarning)
C:\Users\Anis\Anaconda3\lib\site-packages\sklearn\svm\_base.py:977: ConvergenceWarnin
```

```
  "the number of iterations.", ConvergenceWarning)
C:\Users\Anis\Anaconda3\lib\site-packages\sklearn\svm\_base.py:977: ConvergenceWarning
  "the number of iterations.", ConvergenceWarning)
C:\Users\Anis\Anaconda3\lib\site-packages\sklearn\svm\_base.py:977: ConvergenceWarning
  "the number of iterations.", ConvergenceWarning)
```

A warning appearing frequently during this grid search run is the below:

`ConvergenceWarning: Liblinear failed to converge, increase the number of iterations.`

From these warnings, we see that our linear SVM-based model is strugling to converge quite often. A main possible reason for this is that some classes are not well represented. In our dataset, some classes have very few input rows that belong to them, with one class even having as low as one element in it. So, many folds of our cross-validation will only have elements that do not belong to those minority classes, negatively affecting model convergence and prediction quality.

We can confirm we do have this issue from the warning **`"This solver needs samples of at least 2 classes in the data, but the data contains only one class: 0"`** appearing multiple times during the grid search.

The obtained best parameter values

```
#dumping the classifier for later use
import pickle

with open("clf.pickle", "wb") as f:
    pickle.dump(clf, f)

#print the best params returned by grid search
print("Best params:", clf.best_params_)
```

```
 Best params: {'estimator__C': 0.1, 'estimator__loss': 'hinge'}
```

Performance metrics

We now evaluate the performance of our model, with the best found hyperparameter values, using accuracy and f-score metrics. As our problem is multilabel, the AUC metric is not supported. Nonetheless, we did include code we would use to get AUC if we had a compatible problem. We also included the value of the log-loss metric, as this is the performance metric used in the corresponding Kaggle challenge for this dataset.

With an accuracy of **`0.36`**, a micro f1-score of **`0.26`**, and a log-loss of **`7.91`**, our SVM-based classifier is performing quite poorly on the original dataset (remember, this is still without MLSMOTE and without feature selection).

```
from sklearn.metrics import f1_score
from sklearn.metrics import roc_curve
from sklearn.metrics import auc
from sklearn.metrics import log_loss

#predict
y_test_pred = clf.predict(X_test)

#Accuracy
print("Accuracy:", accuracy_score(y_true=y_test, y_pred=y_test_pred))

#F-score
print("F_score:", f1_score(y_true=y_test, y_pred=y_test_pred, average="micro"))

#AUC (will not work coz multilabel)
#fpr, tpr, thresholds = roc_curve(y_test, y_test_pred, pos_label=2)
#print("AUC:", auc(fpr, tpr))

#log-loss (metric used in the Kaggle competition). Lower is better
#Log-loss should give much better value when predictions are probabilistic
print("Log-loss:", log_loss(y_true=y_test, y_pred=y_test_pred))
```

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

```
print("F_score (macro):", f1_score(y_true=y_test, y_pred=y_test_pred, average="macro"))
```

```
F_score (macro): 0.11843347385344896
C:\Users\Anis\Anaconda3\lib\site-packages\sklearn\metrics\_classification.py:1465: Undef
  average, "true nor predicted", 'F-score is', len(true_sum)
```

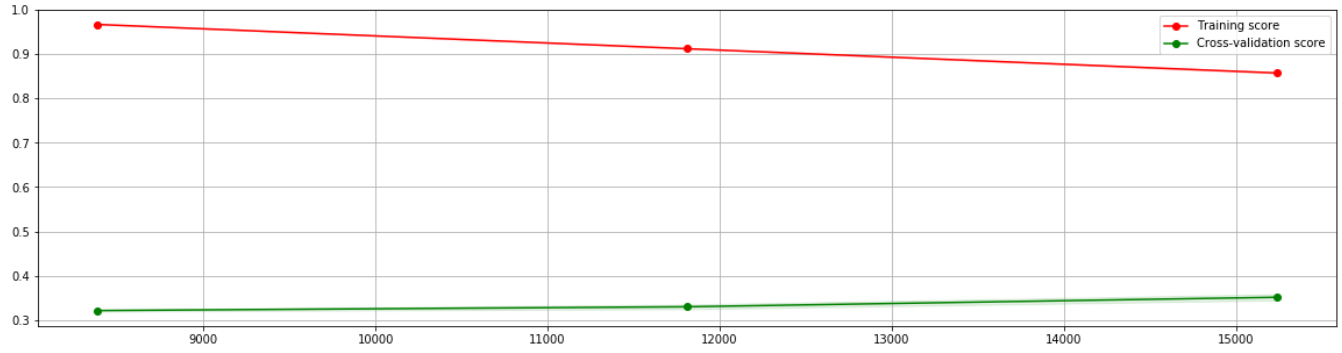
Learning Curve

We now plot the a learning curve for our SVM-based model. Looking at the plot below, we see that the learning curve obtained shows pretty high variance, as accuracy for the training set is much higher than the one for the validation set. So we conclude that our model suffers from high variance and is overfitting.

```
#plotting learning curve
#adapted from https://scikit-learn.org/stable/auto_examples/model_selection/plot_learning_cur
from sklearn.model_selection import learning_curve
from matplotlib import pyplot as plt

best_clf = MultiOutputClassifier(LinearSVC(random_state=42, C=0.1, loss="hinge"), n_jobs=2)
```

```
train_sizes, train_scores, test_scores = \
learning_curve(best_clf, X_train, y_train, cv=cv, n_jobs=2, return_times=False)



train_scores_mean = np.nanmean(train_scores, axis=1)
train_scores_std = np.nanstd(train_scores, axis=1)
test_scores_mean = np.nanmean(test_scores, axis=1)
test_scores_std = np.nanstd(test_scores, axis=1)

_, axes = plt.subplots(1, 1, figsize=(20, 5))

plt.grid()
plt.fill_between(train_sizes, train_scores_mean - train_scores_std,
                     train_scores_mean + train_scores_std, alpha=0.1,
                     color="r")
plt.fill_between(train_sizes, test_scores_mean - test_scores_std,
                     test_scores_mean + test_scores_std, alpha=0.1,
                     color="g")
plt.plot(train_sizes, train_scores_mean, 'o-', color="r",
             label="Training score")
plt.plot(train_sizes, test_scores_mean, 'o-', color="g",
             label="Cross-validation score")
plt.legend(loc="best")

plt.show()
```

```
C:\Users\Anis\Anaconda3\lib\site-packages\ipykernel_launcher.py:1: RuntimeWarning: Mean
  """Entry point for launching an IPython kernel.
C:\Users\Anis\Anaconda3\lib\site-packages\numpy\lib\nanfunctions.py:1665: RuntimeWarning
  keepdims=keepdims)
C:\Users\Anis\Anaconda3\lib\site-packages\ipykernel_launcher.py:3: RuntimeWarning: Mean
  This is separate from the ipykernel package so we can avoid doing imports until
```



SVM (without feature selection, but with MLSMOTE)

We now repeat the same process as above, but after applying MLSMOTE on our dataset. Using MLSMOTE, we added 2000 synthetic minority samples to our original dataset, and we try tuning and fitting an SVM model again to see if we get any performance gain.

Imports

```
import pandas as pd
import numpy as np
from sklearn.multioutput import MultiOutputClassifier
from sklearn.svm import LinearSVC
from sklearn.model_selection import train_test_split
from sklearn.metrics import multilabel_confusion_matrix, ConfusionMatrixDisplay
from sklearn.datasets import make_multilabel_classification
from sklearn.preprocessing import OneHotEncoder
from sklearn.utils import shuffle
from sklearn.metrics import accuracy_score
from sklearn.model_selection import RepeatedKFold
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import GridSearchCV
```

The Dataset (with 2000 synthetic rows added using MLSMOTE)

```
X = pd.read_csv("./Data/train_features_mlsmote_200.csv")
#X = pd.read_csv("./X_afterSteps_12.csv")
y = pd.read_csv("./Data/train_targets_scored_mlsmote_200.csv")
```

```
X = X.drop("Unnamed: 0", 1)
y = y.drop("Unnamed: 0", 1)
```

Splitting into train and test sets

```
#shuffle X and y (together)
X_shuf, y_shuf = shuffle(X, y)

#trying a random split for now
X_train, X_test, y_train, y_test = train_test_split(X_shuf, y_shuf, test_size=0.2, random_sta

#making sure the train part has at least one instance of each class
while any(y_train.sum() == 0):
    X_train, X_test, y_train, y_test = train_test_split(X_shuf, y_shuf, test_size=0.2)
```

Model, Grid Search, Cross-Validation

Now, we build our model, which is a `MultiOutputClassifier` using `LinearSVC`'s as base estimator. So, our model is fitting about 200 SVM models, one for each possible output class. To tune hyperparameters of the base estimators, we use Grid Search to find the best paramaters out of a list of possible combinations. The parameters considered in Grid Search are:

C: {0.1, 1, 10}

loss: {"hinge", "square_hinge"}

To evaluate performance of each combination and see which is best, we combine our grid search with repeated K-fold cross-validation (so we use `GridSearchCV`), where the number of repeats is five and the numner of repeats is two. The parameter values found to produce best performance are:

C: 0.1

loss: "hinge"

```
#initializing the SVM, and making it multilabel using MultiOutputClassifier
multi_svm = MultiOutputClassifier(LinearSVC(random_state=42), n_jobs=2)

cv = RepeatedKFold(n_splits=5, n_repeats=2, random_state=1)
parameters = {'estimator__C':[0.1, 1, 10], 'estimator__loss': ['hinge', 'squared_hinge']}
clf = GridSearchCV(multi_svm, parameters, cv=cv)

clf.fit(X_train, y_train)
```

```
/Users/anis/opt/anaconda3/lib/python3.8/site-packages/sklearn/model_selection/_valida
joblib.externals.loky.process_executor._RemoteTraceback:
"""
Traceback (most recent call last):
  File "/Users/anis/opt/anaconda3/lib/python3.8/site-packages/joblib/externals/loky/p
    r = call_item()
  File "/Users/anis/opt/anaconda3/lib/python3.8/site-packages/joblib/externals/loky/p
    return self.fn(*self.args, **self.kwargs)
  File "/Users/anis/opt/anaconda3/lib/python3.8/site-packages/joblib/_parallel_backen
    return self.func(*args, **kwargs)
  File "/Users/anis/opt/anaconda3/lib/python3.8/site-packages/joblib/parallel.py", li
    return [func(*args, **kwargs)
  File "/Users/anis/opt/anaconda3/lib/python3.8/site-packages/joblib/parallel.py", li
    return [func(*args, **kwargs)
  File "/Users/anis/opt/anaconda3/lib/python3.8/site-packages/sklearn/multioutput.py"
    estimator.fit(X, y, **fit_params)
  File "/Users/anis/opt/anaconda3/lib/python3.8/site-packages/sklearn/svm/_classes.py
    self.coef_, self.intercept_, self.n_iter_ = _fit_liblinear(
  File "/Users/anis/opt/anaconda3/lib/python3.8/site-packages/sklearn/svm/_base.py",
    raise ValueError("This solver needs samples of at least 2 classes"
ValueError: This solver needs samples of at least 2 classes in the data, but the data
"""
```

```
The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "/Users/anis/opt/anaconda3/lib/python3.8/site-packages/sklearn/model_selection
    estimator.fit(X_train, y_train, **fit_params)
  File "/Users/anis/opt/anaconda3/lib/python3.8/site-packages/sklearn/multioutput.py"
    super().fit(X, Y, sample_weight, **fit_params)
  File "/Users/anis/opt/anaconda3/lib/python3.8/site-packages/sklearn/multioutput.py"
    self.estimators_ = Parallel(n_jobs=self.n_jobs)(
  File "/Users/anis/opt/anaconda3/lib/python3.8/site-packages/joblib/parallel.py", li
    self.retrieve()
  File "/Users/anis/opt/anaconda3/lib/python3.8/site-packages/joblib/parallel.py", li
    self._output.extend(job.get(timeout=self.timeout))
  File "/Users/anis/opt/anaconda3/lib/python3.8/site-packages/joblib/_parallel_backen
    return future.result(timeout=timeout)
  File "/Users/anis/opt/anaconda3/lib/python3.8/concurrent/futures/_base.py", line 43
    return self.__get_result()
  File "/Users/anis/opt/anaconda3/lib/python3.8/concurrent/futures/_base.py", line 38
    raise self._exception
ValueError: This solver needs samples of at least 2 classes in the data, but the data

  warnings.warn("Estimator fit failed. The score on this train-test"
/Users/anis/opt/anaconda3/lib/python3.8/site-packages/sklearn/model_selection/_valida
joblib.externals.loky.process_executor._RemoteTraceback:
"""
Traceback (most recent call last):
  File "/Users/anis/opt/anaconda3/lib/python3.8/site-packages/joblib/externals/loky/p
    r = call_item()
  File "/Users/anis/opt/anaconda3/lib/python3.8/site-packages/joblib/externals/loky/p
    return self.fn(*self.args, **self.kwargs)
  File "/Users/anis/opt/anaconda3/lib/python3.8/site-packages/joblib/_parallel_backen
    return self.func(*args, **kwargs)
  File "/Users/anis/opt/anaconda3/lib/python3.8/site-packages/joblib/parallel.py", li
    return [func(*args, **kwargs)
  File "/Users/anis/opt/anaconda3/lib/python3.8/site-packages/joblib/parallel.py", li
```

The obtained best parameter combination

```
#dumping the classifier for later use
import pickle

with open("clf.pickle", "wb") as f:
    pickle.dump(clf, f)

#printing the best param values returned by grid search
print(clf.best_params_)
```

```
{'estimator__C': 0.1, 'estimator__loss': 'hinge'}
```

Performance metrics

We now evaluate the performance of our model, with the best found hyperparameter values, using accuracy and f-score metrics. As our problem is multilabel, the AUC metric is not supported. Nonetheless, we did include code we would use to get AUC if we had a compatible problem. We also included the value of the log-loss metric, as this is the performance metric used in the corresponding Kaggle challenge for this dataset.

With an accuracy of **`0.38`**, a micro f1-score of **`0.25`**, and a log-loss of **`6.64`**, our SVM-based classifier is performing slightly better on MLSMOTE'd data compared to our original dataset, but overall results are still poor.

```
from sklearn.metrics import f1_score
from sklearn.metrics import roc_curve
from sklearn.metrics import auc
from sklearn.metrics import log_loss

y_test_pred = clf.predict(X_test)

#accuracy
print("Accuracy:", accuracy_score(y_true=y_test, y_pred=y_test_pred))

#f-score
print("F-score micro:", f1_score(y_true=y_test, y_pred=y_test_pred, average="micro"))
print("F-score macro:", f1_score(y_true=y_test, y_pred=y_test_pred, average="macro"))
print("F-score weighted:", f1_score(y_true=y_test, y_pred=y_test_pred, average="weighted"))

#roc and auc
#but AUC is not supported for multi-label
#fpr, tpr, thresholds = roc_curve(y_test, y_test_pred, pos_label=2)
#print("AUC:", auc(fpr, tpr))

#log-loss (the metric used in the Kaggle challenge for this data)
#log-loss should give much better values when predictions are probabilistic
print("Log-loss:", log_loss(y_true=y_test, y_pred=y_test_pred))
```

```
Accuracy: 0.3881464264962231
F-score micro: 0.26546091015169193
/Users/anis/opt/anaconda3/lib/python3.8/site-packages/sklearn/metrics/_classification.py
  _warn_prf(
F-score macro: 0.13389851517631826
F-score weighted: 0.2574073194315111
Log-loss: 6.644603432093816
```

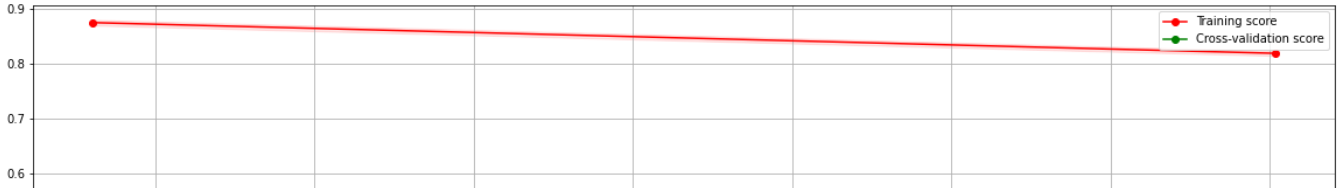
Learning Curve

We now plot the a learning curve for our SVM-based model. Looking at the plot below, we see that the learning curve obtained shows pretty high variance, as accuracy for the training set is much

higher than the one for the validation set. So we conclude that our model suffers from high variance and is overfitting.

```
#plotting learning curve
#adapted from https://scikit-learn.org/stable/auto_examples/model_selection/plot_learning_cur
from sklearn.model_selection import learning_curve
from matplotlib import pyplot as plt

#TODO: Replace "..." by the best values returned by grid search.
best_clf = MultiOutputClassifier(LinearSVC(random_state=42, C=0.1, loss="hinge"), n_jobs=2)

train_sizes, train_scores, test_scores = \
learning_curve(best_clf, X_train, y_train, cv=cv, n_jobs=2, return_times=False)



train_scores_mean = np.nanmean(train_scores, axis=1)
train_scores_std = np.nanstd(train_scores, axis=1)
test_scores_mean = np.nanmean(test_scores, axis=1)
test_scores_std = np.nanstd(test_scores, axis=1)

_, axes = plt.subplots(1, 1, figsize=(20, 5))

axes.grid()
axes.fill_between(train_sizes, train_scores_mean - train_scores_std,
                    train_scores_mean + train_scores_std, alpha=0.1,
                    color="r")
axes.fill_between(train_sizes, test_scores_mean - test_scores_std,
                    test_scores_mean + test_scores_std, alpha=0.1,
                    color="g")
axes.plot(train_sizes, train_scores_mean, 'o-', color="r",
             label="Training score")
axes.plot(train_sizes, test_scores_mean, 'o-', color="g",
             label="Cross-validation score")
axes.legend(loc="best")

plt.show()
```

```
<ipython-input-31-1619e089e11f>:1: RuntimeWarning: Mean of empty slice
  train_scores_mean = np.nanmean(train_scores, axis=1)
/Users/anis/opt/anaconda3/lib/python3.8/site-packages/numpy/lib/nanfunctions.py:1666: Ru
  var = nanvar(a, axis=axis, dtype=dtype, out=out, ddof=ddof,
<ipython-input-31-1619e089e11f>:3: RuntimeWarning: Mean of empty slice
  test_scores_mean = np.nanmean(test_scores, axis=1)
```



Conclusion: Did MLSMOTE improve our SVM model performance?

In conclusion, 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.

Random Forest on Original Dataset and Feature Selected Dataset

In thie notebook, we aim to use Random Forests to solve our term project problem. For this, we tune and fit a random forest classifier on our original dataset, assess the model's performance, before plotting and analyzing the model's learning curve.

We then repeat the same process on our data after applying feature selection, to see whether feature selection helps improve prediction performance.

Imports

```
import pandas as pd
import numpy as np
from sklearn.multioutput import MultiOutputClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import multilabel_confusion_matrix, ConfusionMatrixDisplay
from sklearn.datasets import make_multilabel_classification
from sklearn.preprocessing import OneHotEncoder
from sklearn.utils import shuffle
from sklearn.metrics import accuracy_score
from sklearn.model_selection import RepeatedKFold
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import GridSearchCV
```

Data (before feature selection)

```
X = pd.read_csv("./Data/train_features.csv")
y = pd.read_csv("./Data/train_targets_scored.csv")
```

```
#making sure X and y are same length, plus that drugs are stored in same order in both X and
#indeed they are
print(all(X["sig_id"] == y["sig_id"]))

#drop sig_id col from both X and y
X = X.drop("sig_id", 1)
y = y.drop("sig_id", 1)

# X = X.drop("Unnamed: 0", 1)
# y = y.drop("Unnamed: 0", 1)

#encode categorical cols, X.cp_type and X.cp_dose, into new cols
X['cp_type_trt_cp'] = X['cp_type'].map( {'trt_cp':1, 'ctl_vehicle':0} )
X['cp_dose_D1'] = X['cp_dose'].map( {'D1':1, 'D2':0} )
```

```
#delete the original cols
X = X.drop("cp_type", 1)
X = X.drop("cp_dose", 1)
```

```
True
```

Train-Test Split

Before fitting the model to our data, we shuffle the data and then split it into train (80%) and test (20%) datasets.

When splitting, we ensure that the training set has at least one instance of each possible class, so that no class is completely left out from the training process.

```
#shuffle X and y (together)
X_shuf, y_shuf = shuffle(X, y)

#trying a random split for now
X_train, X_test, y_train, y_test = train_test_split(X_shuf, y_shuf, test_size=0.2, random_sta

#making sure the train part has at least one instance of each class
while any(y_train.sum() == 0):
    X_train, X_test, y_train, y_test = train_test_split(X_shuf, y_shuf, test_size=0.2)
```

Random Forest: Basic Trial (no tuning)

This is an initial try at fitting a random forest with no tuning, keeping all parametera at their default, just to get a first impression of what we can achieve with this model.

a- On Original Dataset, Before Feature Selection

```
rf = RandomForestClassifier()
rf.fit(X_train, y_train)
```

```
RandomForestClassifier()
```

```
y_pred = rf.predict(X_test)
```

```
from sklearn.metrics import f1_score
from sklearn.metrics import roc_curve
from sklearn.metrics import auc
from sklearn.metrics import log_loss
```

```
#accuracy
```

```
print("Accuracy:", accuracy_score(y_true=y_test, y_pred=y_pred))

#f-score
print("F-score micro:", f1_score(y_true=y_test, y_pred=y_pred, average="micro"))
print("F-score macro:", f1_score(y_true=y_test, y_pred=y_pred, average="macro"))
print("F-score weighted:", f1_score(y_true=y_test, y_pred=y_pred, average="weighted"))
#roc and auc
#but AUC is not supported for multi-label
#fpr, tpr, thresholds = roc_curve(y_test, y_test_pred, pos_label=2)
#print("AUC:", auc(fpr, tpr))

#log-loss (the metric used in the Kaggle challenge for this data)
#log-loss should give much better values when predictions are probabilistic
print("Log-loss:", log_loss(y_true=y_test, y_pred=y_pred))
y_pred_proba = rf.predict_proba(X_test)
#print("Log-loss (probabilistic predictions):", log_loss(y_true=y_test, y_pred=y_pred_proba))
```

```
Accuracy: 0.4635733781230317
F-score micro: 0.2676560900716479
F-score macro: 0.04197341268771417
/Users/anis/opt/anaconda3/lib/python3.8/site-packages/sklearn/metrics/_classification.py
  _warn_prf(
F-score weighted: 0.18152000191463233
Log-loss: 3.3268664559846264
```

We assessed performance of this model using three metrics: accuracy, f-score, and log-loss (which is the metric used in the Kaggle competition of this dataset).

Looking at the performance of Random Forest without training, we see that its accuracy of `0.46` is better than the best SVM we could fit (at `0.38`), its log-loss of `3.3` is way better than that SVM (at about `6.5`), and its micro f-score is about the same as the SVM.

These results are obviously not good per se, but it's a relatively good sign nonetheless, as this shows than even a random forest without tuning is giving us better results than our SVM with tuning.

b- On Dataset After Feature Selection

```
X = pd.read_csv("./X_afterSteps_12.csv")
y = pd.read_csv("./Data/train_targets_scored.csv")


#making sure X and y are same length, plus that drugs are stored in same order in both X and
#indeed they are
print(all(X["sig_id"] == y["sig_id"]))

#drop sig_id col from both X and y
X = X.drop("sig_id", 1)
```

```
y = y.drop("sig_id", 1)

# X = X.drop("Unnamed: 0", 1)
# y = y.drop("Unnamed: 0", 1)

#encode categorical cols, X.cp_type and X.cp_dose, into new cols
X['cp_type_trt_cp'] = X['cp_type'].map( {'trt_cp':1, 'ctl_vehicle':0} )
X['cp_dose_D1'] = X['cp_dose'].map( {'D1':1, 'D2':0} )

#delete the original cols
X = X.drop("cp_type", 1)
X = X.drop("cp_dose", 1)
```

```
     True
```

```
#shuffle X and y (together)
X_shuf, y_shuf = shuffle(X, y)

#trying a random split for now
X_train, X_test, y_train, y_test = train_test_split(X_shuf, y_shuf, test_size=0.2, random_sta

#making sure the train part has at least one instance of each class
while any(y_train.sum() == 0):
    X_train, X_test, y_train, y_test = train_test_split(X_shuf, y_shuf, test_size=0.2)


rf = RandomForestClassifier()
rf.fit(X_train, y_train)

y_pred = rf.predict(X_test)


from sklearn.metrics import f1_score
from sklearn.metrics import roc_curve
from sklearn.metrics import auc
from sklearn.metrics import log_loss



#accuracy
print("Accuracy:", accuracy_score(y_true=y_test, y_pred=y_pred))

#f-score
print("F-score micro:", f1_score(y_true=y_test, y_pred=y_pred, average="micro"))
print("F-score macro:", f1_score(y_true=y_test, y_pred=y_pred, average="macro"))
print("F-score weighted:", f1_score(y_true=y_test, y_pred=y_pred, average="weighted"))
#roc and auc
#but AUC is not supported for multi-label
#fpr, tpr, thresholds = roc_curve(y_test, y_test_pred, pos_label=2)
#print("AUC:", auc(fpr, tpr))

#log-loss (the metric used in the Kaggle challenge for this data)
#log loss should give much better values when predictions are probabilistic
```

```
#log-loss should give much better values when predictions are probabilistic
print("Log-loss:", log_loss(y_true=y_test, y_pred=y_pred))
y_pred_proba = rf.predict_proba(X_test)
#print("Log-loss (probabilistic predictions):", log_loss(y_true=y_test, y_pred=y_pred_proba))
```

This sadly did not finish before the due date, so we have no results to show and interpret.

Hyperparameter tuning, using grid search and cross-validation

a- Before Feature Selection

We now build our model using sklearn's `RandomForestClassifier` class. This class does support multilabel problems, so there is no need to wrap our model inside a `MultiOutputClassifier`.

To get the most out of our model, we find the optimal paramaters (out pf a list of possible combinations) using Grid Search, coupled with Cross-Validation which scores should determine which parameter value combination results in best performance. The class `GridSearchCV` provides us exactly what we need, and this is what we will be using for this step.

The hyperparameter values we consider are:

1- The number of trees to have in the forest

`'n_estimators': [20, 50, 100]`

2- The maximum number of features to explore at each split of a tree

`'max_features': ["auto", "sqrt", "log2"]`

The best parameter combination we obtained is:

`'n_estimators': ...; 'max_features': ...`

Sadly this grid search did not finish before the due date, we left it almost two days and it did not finish on time. So unfortunately we do not have results to show and analyze.

```
rf_gridsearch = RandomForestClassifier()

cv = RepeatedKFold(n_splits=5, n_repeats=2, random_state=1)
parameters = {'n_estimators': [20, 50, 100], 'max_features': ["auto", "sqrt", "log2"]}
clf = GridSearchCV(rf_gridsearch, parameters, cv=cv)

clf.fit(X_train, y_train)
```

The best parameters found by Grid Search

```
print(clf.best_params_)
```

Performance Metrics

We now evaluate the performance of our model, with the best found hyperparameter values, using accuracy and f-score metrics. As our problem is multilabel, the AUC metric is not supported. Nonetheless, we did include code we would use to get AUC if we had a compatible problem. We also included the value of the log-loss metric, as this is the performance metric used in the corresponding Kaggle challenge for this dataset.

Again as model fitting did not finish in time, we have no results to show sadly.

```
y_pred = clf.predict(X_test)


from sklearn.metrics import f1_score
from sklearn.metrics import roc_curve
from sklearn.metrics import auc
from sklearn.metrics import log_loss


#accuracy
print("Accuracy:", accuracy_score(y_true=y_test, y_pred=y_pred))

#f-score
print("F-score micro:", f1_score(y_true=y_test, y_pred=y_pred, average="micro"))
print("F-score macro:", f1_score(y_true=y_test, y_pred=y_pred, average="macro"))
print("F-score weighted:", f1_score(y_true=y_test, y_pred=y_pred, average="weighted"))
#roc and auc
#but AUC is not supported for multi-label
#fpr, tpr, thresholds = roc_curve(y_test, y_test_pred, pos_label=2)
#print("AUC:", auc(fpr, tpr))

#log-loss (the metric used in the Kaggle challenge for this data)
#log-loss should give much better values when predictions are probabilistic
print("Log-loss:", log_loss(y_true=y_test, y_pred=y_pred))
y_pred_proba = rf.predict_proba(X_test)
#print("Log-loss (probabilistic predictions):", log_loss(y_true=y_test, y_pred=y_pred_proba))
```

Learning Curve

We now plot the a learning curve for our Random Forest model.

```
#plotting learning curve
#adapted from https://scikit-learn.org/stable/auto_examples/model_selection/plot_learning_cur
from sklearn.model_selection import learning_curve
from matplotlib import pyplot as plt

#TODO: Replace "..." by the best values returned by grid search.
```

```
best_clf = MultiOutputClassifier(LinearSVC(random_state=42, C=0.1, loss="hinge"), n_jobs=2)

train_sizes, train_scores, test_scores = \
learning_curve(best_clf, X_train, y_train, cv=cv, n_jobs=2, return_times=False)



train_scores_mean = np.nanmean(train_scores, axis=1)
train_scores_std = np.nanstd(train_scores, axis=1)
test_scores_mean = np.nanmean(test_scores, axis=1)
test_scores_std = np.nanstd(test_scores, axis=1)

_, axes = plt.subplots(1, 1, figsize=(20, 5))

axes.grid()
axes.fill_between(train_sizes, train_scores_mean - train_scores_std,
                     train_scores_mean + train_scores_std, alpha=0.1,
                     color="r")
axes.fill_between(train_sizes, test_scores_mean - test_scores_std,
                     test_scores_mean + test_scores_std, alpha=0.1,
                     color="g")
axes.plot(train_sizes, train_scores_mean, 'o-', color="r",
             label="Training score")
axes.plot(train_sizes, test_scores_mean, 'o-', color="g",
             label="Cross-validation score")
axes.legend(loc="best")

plt.show()
```

b- After Feature Selection

The code for this part would be very similar to the part before feature selection. Since the latter did not finish, we also could not start running this part.

Conclusion

Although we did not manage to get meaningful results in time, we do have some hopes for this model. Without any tuning, it performed better than our tuned SVM on the original data, with a significantly better log-loss (but still not good enough). So, we expect this model to improve further using hyperparameter tuning.

The more important question we did not manage to answer in time is whether Feature Selection improves (or affects in general) the performance of Random Forests.

Neural Network (on original data without feature selection)

```
from pandas import read_csv
import numpy as np
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import SGD
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, roc_auc_
from sklearn.model_selection import train_test_split, RepeatedKFold, learning_curve
import matplotlib.pyplot as plt
import tensorflow as tf


# X = read_csv('train_features_mlsmote_200.csv')
X = read_csv('train_features.csv')
X = np.asarray(X)[:,4:]

# y = read_csv('train_targets_scored_mlsmote_200.csv')
y = read_csv('train_targets_scored.csv')
y = np.asarray(y)[:,1:]

n_layers = [64, 128, 64]
X_shape, y_shape = X.shape[1], y.shape[1]
```

Creating model without feature selection

```
def get_model(learn_rate=0.0001, momentum=0.8, init_mode='glorot_normal', activation='relu'):
    n_input, n_classes = X_shape, y_shape
    model = Sequential()

    model.add(Dense(n_layers[0], input_dim=n_input, activation=activation, kernel_initializer
    for nb_neurons in n_layers[1:len(n_layers)]:
        model.add(Dense(nb_neurons, activation=activation, kernel_initializer=init_mode))
    model.add(Dense(n_classes, activation='sigmoid'))

    opt = SGD(lr=learn_rate, momentum=momentum)
    model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])

    return model

def print_metrics(y_test, yhat):
    acc = accuracy_score(y_test, yhat)
    prec = precision_score(y_test, yhat, average='micro')
    recall = recall_score(y_test, yhat, average='micro')
    f1 = f1_score(y_test, yhat, average='micro')
    roc = roc_auc_score(y_test, yhat, average='micro')
    print(acc)
```

```
    print(prec)
    print(recall)
    print(f1)
    print(roc)

def evaluate_model(model, X_train, X_test, y_train, y_test):
    history = model.fit(X_train, y_train, validation_split=0.33, epochs=500, batch_size=5, ve
    yhat = model.predict(X_test)
    yhat = yhat.round()
    print_metrics(y_test, yhat)
    return history


X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=123)
X_train, X_test, y_train, y_test = np.asarray(X_train, np.float64), np.asarray(X_test, np.flo

model = get_model()
history = evaluate_model(model, X_train, X_test, y_train, y_test)
```

Without MLSMOTE

precision = 0.06045370204106063

recall = 0.3013126491646778

f1-score = 0.10070292636721671

AUC = 0.642629845433697

After Applying MLSMOTE

precision = 0.3284118116520351

recall = 0.23407281001137656

f1-score = 0.2733311192294919

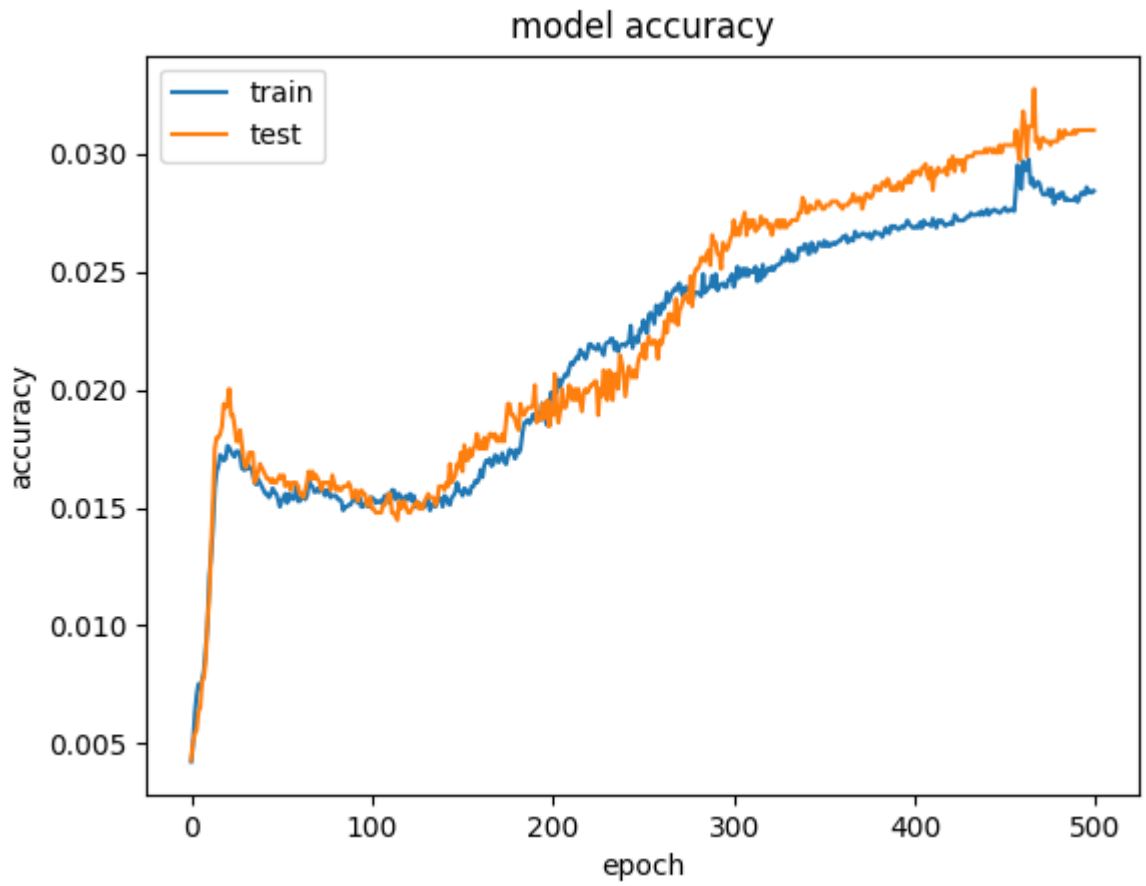
AUC = 0.6163926931866344

Plotting learning curves before MLSMOTE

```
def plotHistoryAccuracyVSEpoch(history):
    plt.plot(history.history['accuracy'])
    plt.plot(history.history['val_accuracy'])
    plt.title('model accuracy')
    plt.ylabel('accuracy')
    plt.xlabel('epoch')
    plt.legend(['train', 'test'], loc='upper left')
    plt.show()

def plotHistoryLossVSEpoch(history):
    plt.plot(history.history['loss'])
    plt.plot(history.history['val_loss'])
    plt.title('model loss')
    plt.ylabel('loss')
```

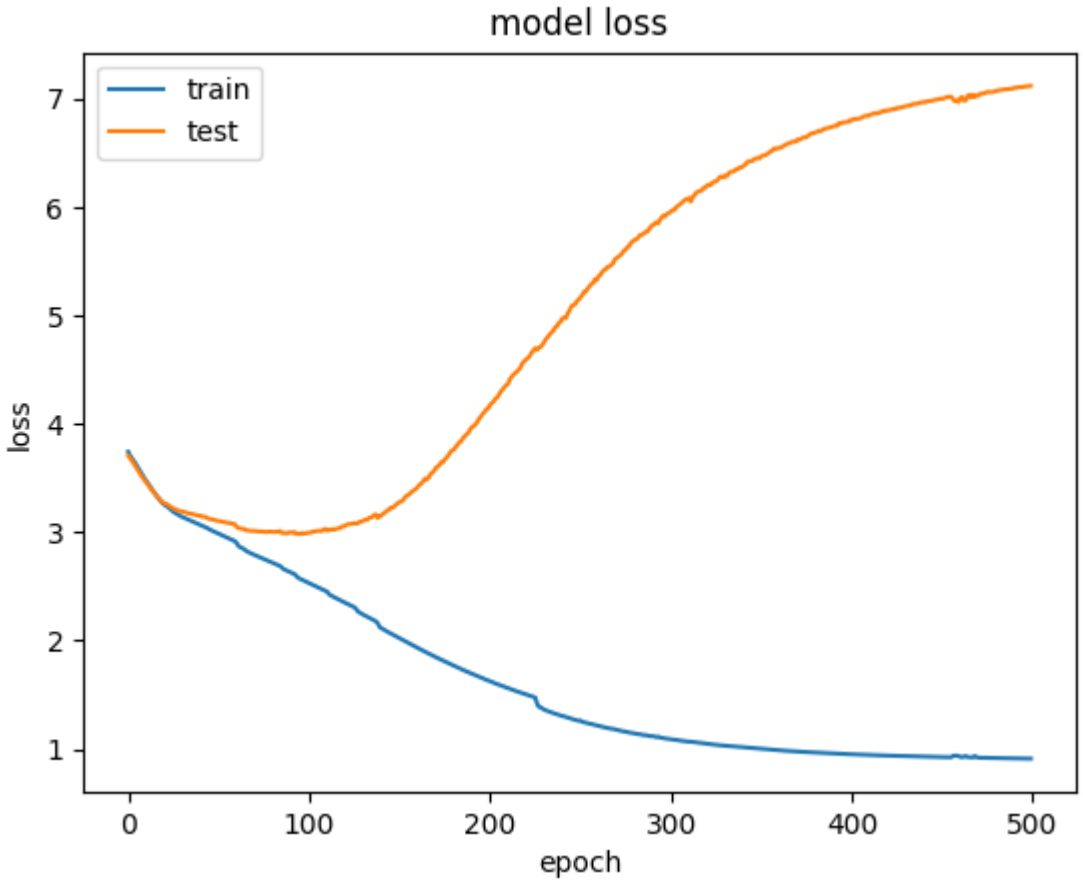
```
    plt.xlabel('epoch')
    plt.legend(['train', 'test'], loc='upper left')
    plt.show()
```

```
plotHistoryAccuracyVSEpoch(history)
```



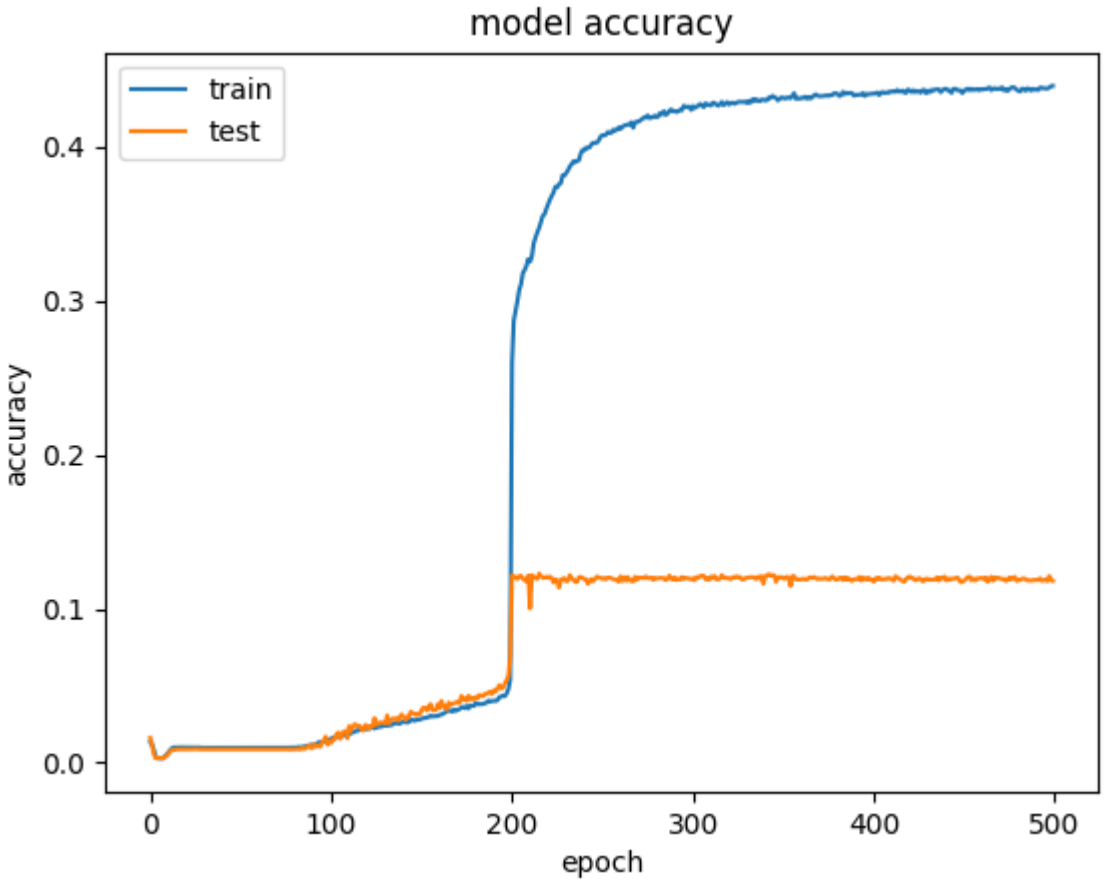
First, we got a model where the accuracy start increasing and after multiple epoch arround 40, the accuracy drops to 0. Therefore, we did some tuning parameters, and decreased the learning rate to 0.0001 and the batch size to 5, so we started getting higher accuracy and F-score. Here it was still clear that the accuracy was very low, and as the number of epoch increase the accuracy starts to become more constant.

```
plotHistoryLossVSEpoch(history)
```

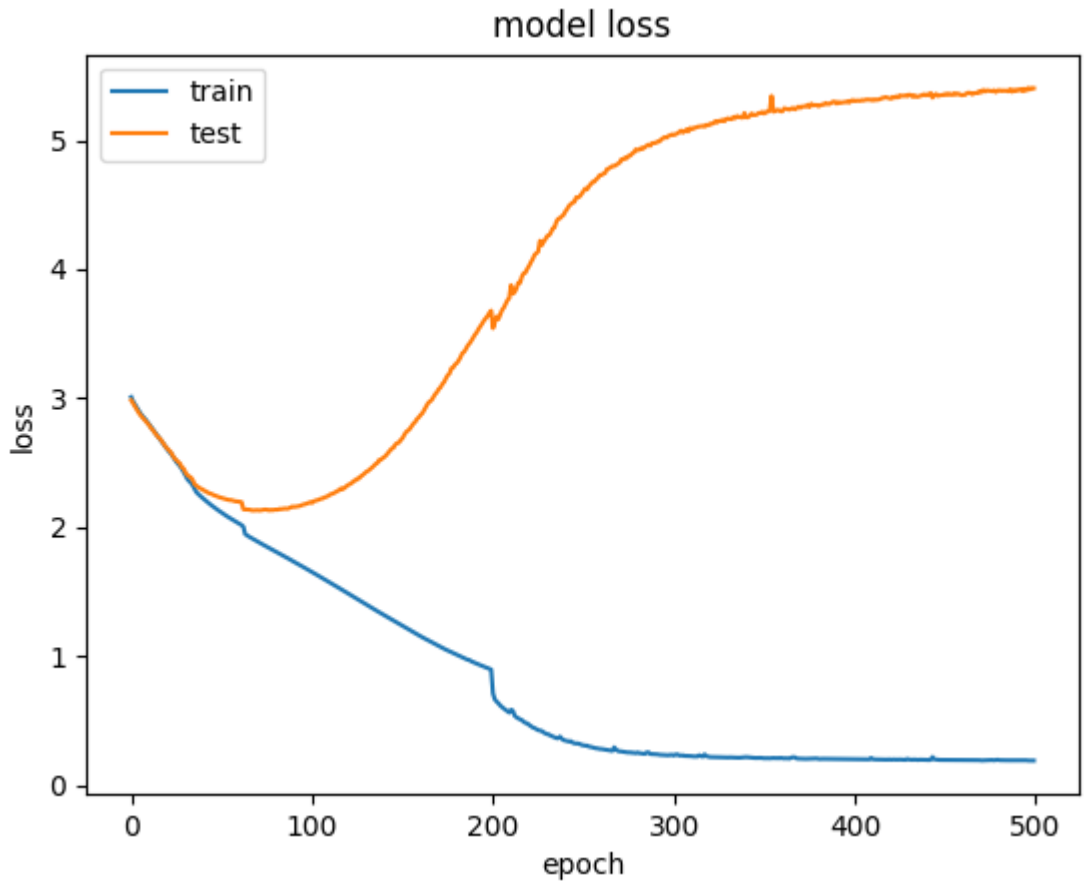


From this graph we can realize how the loss is decreasing for the training set while the loss of the test set is increasing and this will not allow the model to generalize but only give good result on the training set. It was clear that the number of rows in the dataset is very low and does not allow the Model to generalize and be able to predict.
After Applying, MLSMOTE on the dataset and creating arroung 8000 new instances for the minority classes, the accuracy remained low; however, it increased

Plotting learning curves after MLSMOTE
From the **Accuracy graph** below we can see that the accuracy increase for the training set for arround 0.4 as it was 0.03 which is a high improvement. However, you can realize that after increasing the number of epoch the function started to converge. Regarding the accuracy of the test set it is low and this show that not all feature are important in this multi-label classification
From the **Loss graph* below we can realize how the loss is decreasing for the training set while the loss of the test set is increasing and this will not allow the model to generalize. After adding more rows to the dataset, now it is better to do feature selection to find what are the most important features that are effecting the result.



Double-click (or enter) to edit



K-Fold Cross Validation

It was clear that the model is not very stable because of variance in the parameters after running the model multiple time, for example for a learning rate of 0.0001, batch size of 5 and epoch of 150, we were able to get the below metrics:
Accuracy = 0.4159143397018686
Precision = 0.32974504249291786
Recall = 0.17362768496420047
F1-score = 0.22747703732655852
AUC = 0.5862089291335085

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_test, yhat,
            'F1-score':f1_score(y_test, yhat, average='micro'), "ROC AUC":roc_auc_score(y_tes

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))
```

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}

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. Therefore we decided to use Grid Search to find a better Tuning for the Neural Network

Grid Search

This Grid Search have been applied on a hidden layers of [64, 128, 64], and have taken arround 30 hours to finalize a sample of the output is displayed below, and we found out that the best parameters were: 'activation': 'linear', 'init_mode': 'glorot_normal', 'learn_rate': 0.01, 'momentum': 0.9, and it got very low accuracy of 0.278490. However, it is possible sometimes with more tuning or taking into account an intermedite number of learning rate for example could improve the performance of model

```
X = np.asarray(X).astype('float64')
y = np.asarray(y).astype('float64')

model = KerasClassifier(build_fn=get_model, epochs=10)
learn_rate = [0.001, 0.01, 0.1, 0.2, 0.3]
momentum = [0.0, 0.2, 0.4, 0.6, 0.8, 0.9]
activation = ['softmax', 'softplus', 'softsign', 'relu', 'tanh', 'sigmoid', 'hard_sigmoid', '
init_mode = ['uniform', 'lecun_uniform', 'normal', 'zero', 'glorot_normal', 'glorot_uniform',
param_grid = dict(learn_rate=learn_rate, momentum=momentum, activation=activation, init_mode=
grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1, cv=10)
grid_result = grid.fit(X, y)

print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
```

```
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

Best: 0.278490 using {'activation': 'linear', 'init_mode': 'glorot_normal', 'learn_rate': 0.01, 'momentum': 0.9}

0.026330 (0.013417) with: {'activation': 'softmax', 'init_mode': 'uniform', 'learn_rate': 0.001, 'momentum': 0.0}
0.029564 (0.010454) with: {'activation': 'softmax', 'init_mode': 'lecun_uniform', 'learn_rate': 0.001, 'momentum': 0.0}
0.033426 (0.003896) with: {'activation': 'softmax', 'init_mode': 'lecun_uniform', 'learn_rate': 0.001, 'momentum': 0.2}
0.033426 (0.003896) with: {'activation': 'softmax', 'init_mode': 'normal', 'learn_rate': 0.001, 'momentum': 0.0}
0.027547 (0.014069) with: {'activation': 'softmax', 'init_mode': 'normal', 'learn_rate': 0.001, 'momentum': 0.2}
0.033426 (0.003896) with: {'activation': 'softmax', 'init_mode': 'zero', 'learn_rate': 0.001, 'momentum': 0.8}
0.033426 (0.003896) with: {'activation': 'softmax', 'init_mode': 'zero', 'learn_rate': 0.001, 'momentum': 0.9}
0.029564 (0.010454) with: {'activation': 'softmax', 'init_mode': 'glorot_normal', 'learn_rate': 0.001, 'momentum': 0.0}
0.023853 (0.015718) with: {'activation': 'softmax', 'init_mode': 'glorot_normal', 'learn_rate': 0.001, 'momentum': 0.2}

Neural Network after Data Exploration and analysis

Since our problem is not easy having many classes and imbalanced data, we decided to do another version of this phase with feature selection applied to our dataset, and this report is about some work that has been done again with feature selection considered. We believe we might obtain better results in some models and worse in others, that is why we decided to do with and without feature selection. In both cases, the result might not be the best since some advanced techniques should be applied to fix imbalanced issues (research part of this project) but we tried what we could for this phase. Below starts the work done on the problem after selecting features in the last phase

```
import numpy as np
import pandas as pd
import random as rd
import pickle
from sklearn.feature_selection import SelectKBest, chi2 ,f_classif
!pip install learn2learn
import learn2learn as l2l
from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import LinearSVC, SVC
from sklearn.model_selection import train_test_split, learning_curve, ShuffleSplit, GridSearc
from sklearn.metrics import accuracy_score
from torch import nn, optim
from sklearn.metrics import f1_score, log_loss, roc_auc_score
import matplotlib.pyplot as plt
from sklearn.pipeline import Pipeline
```

```
Requirement already satisfied: learn2learn in /usr/local/lib/python3.6/dist-packages (0
Requirement already satisfied: pandas in /usr/local/lib/python3.6/dist-packages (from l
Requirement already satisfied: numpy>=1.15.4 in /usr/local/lib/python3.6/dist-packages
Requirement already satisfied: torchvision>=0.3.0 in /usr/local/lib/python3.6/dist-pack
Requirement already satisfied: tqdm in /usr/local/lib/python3.6/dist-packages (from lea
Requirement already satisfied: gsutil in /usr/local/lib/python3.6/dist-packages (from l
Requirement already satisfied: torch>=1.1.0 in /usr/local/lib/python3.6/dist-packages (
Requirement already satisfied: gym>=0.14.0 in /usr/local/lib/python3.6/dist-packages (f
Requirement already satisfied: requests in /usr/local/lib/python3.6/dist-packages (from
Requirement already satisfied: python-dateutil>=2.7.3 in /usr/local/lib/python3.6/dist-
Requirement already satisfied: pytz>=2017.2 in /usr/local/lib/python3.6/dist-packages (
Requirement already satisfied: pillow>=4.1.1 in /usr/local/lib/python3.6/dist-packages
Requirement already satisfied: gcs-oauth2-boto-plugin>=2.7 in /usr/local/lib/python3.6/
Requirement already satisfied: six>=1.12.0 in /usr/local/lib/python3.6/dist-packages (f
Requirement already satisfied: argcomplete>=1.9.4 in /usr/local/lib/python3.6/dist-pack
Requirement already satisfied: mock==2.0.0 in /usr/local/lib/python3.6/dist-packages (f
Requirement already satisfied: fasteners>=0.14.1 in /usr/local/lib/python3.6/dist-packa
Requirement already satisfied: httplib2>=0.18 in /usr/local/lib/python3.6/dist-packages
Requirement already satisfied: pyOpenSSL>=0.13 in /usr/local/lib/python3.6/dist-package
Requirement already satisfied: google-reauth>=0.1.0 in /usr/local/lib/python3.6/dist-pa
Requirement already satisfied: retry-decorator>=1.0.0 in /usr/local/lib/python3.6/dist-
```

```
Requirement already satisfied: monotonic>=1.4 in /usr/local/lib/python3.6/dist-packages
Requirement already satisfied: crcmod>=1.7 in /usr/local/lib/python3.6/dist-packages (f
Requirement already satisfied: google-apitools>=0.5.30 in /usr/local/lib/python3.6/dist
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.6/dist-packa
Requirement already satisfied: future in /usr/local/lib/python3.6/dist-packages (from t
Requirement already satisfied: dataclasses in /usr/local/lib/python3.6/dist-packages (f
Requirement already satisfied: scipy in /usr/local/lib/python3.6/dist-packages (from gy
Requirement already satisfied: cloudpickle<1.7.0,>=1.2.0 in /usr/local/lib/python3.6/di
Requirement already satisfied: pyglet<=1.5.0,>=1.4.0 in /usr/local/lib/python3.6/dist-p
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.6/dist-packa
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.6/dist-pack
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /usr/local/li
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.6/dist-packages (
Requirement already satisfied: boto>=2.29.1 in /usr/local/lib/python3.6/dist-packages (
Requirement already satisfied: oauth2client>=2.2.0 in /usr/local/lib/python3.6/dist-pac
Requirement already satisfied: importlib-metadata<4,>=0.23; python_version == "3.6" in
Requirement already satisfied: pbr>=0.11 in /usr/local/lib/python3.6/dist-packages (fro
Requirement already satisfied: cryptography>=2.8 in /usr/local/lib/python3.6/dist-packa
Requirement already satisfied: pyu2f in /usr/local/lib/python3.6/dist-packages (from go
Requirement already satisfied: rsa>=3.1.4 in /usr/local/lib/python3.6/dist-packages (fr
Requirement already satisfied: pyasn1-modules>=0.0.5 in /usr/local/lib/python3.6/dist-p
Requirement already satisfied: pyasn1>=0.1.7 in /usr/local/lib/python3.6/dist-packages
Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.6/dist-packages (fro
Requirement already satisfied: cffi!=1.11.3,>=1.8 in /usr/local/lib/python3.6/dist-pack
Requirement already satisfied: pycparser in /usr/local/lib/python3.6/dist-packages (fro
```

```
# given features matrix and array with important feature indexes, this function returns the s
def get_selected_features(X, f_i):
  X_selected = []
  for x in X:
    selected = []
    for i in f_i:
      selected.append(x[i])
    X_selected.append(selected)
  return np.array(X_selected)

# same function but this is woring with dataframes instead of numpy arrays, returns a datafra
def get_selected_features_df(X_df, f_i):
  X_selected = X_df.copy()
  i = 0
  for col in X_df.iloc[:, 4:]:
    if i not in f_i: # numerical starts with 4
      X_selected.pop(col) # drop feature column at index i
    i += 1
  return X_selected



# load data - pre-processed before, we just need to load feature selection tools to use them
fi = []
train_data = []
with open('/content/drive/My Drive/Colab Notebooks/features_indexes.data', 'rb') as f:
  fi = pickle.load(f)
```

```
# train_data = pd.read_csv('/content/drive/My Drive/Colab Notebooks/X_afterSteps_12.csv')
train_data = pd.read_csv('/content/drive/My Drive/Colab Notebooks/train_features.csv')
df_y = pd.read_csv("/content/drive/My Drive/Colab Notebooks/train_targets_scored.csv")
train_labels = np.array(df_y.iloc[:, 1:])


train_data = get_selected_features_df(train_data, fi)


x = train_data.iloc[:, 2:]

def encode(df, col):
  new_df = df.copy()
  classes = list(pd.unique(new_df.iloc[:,col]))
  for i in range(len(df.iloc[:,col])):
    new_df.iloc[i, col] = classes.index(new_df.iloc[i, col])
  return new_df

x=encode(x,0)
x=encode(x,1)
x=encode(x,2)
print(x)
y = df_y.iloc[:, 1:]
```

```
       cp_time cp_dose      g-0     g-7  ...    c-96    c-97    c-98    c-99
0            0       0      0.0 -0.0326  ... -0.3981  0.2139  0.3801  0.4176
1            1       0      1.0  0.3372  ...  0.1522  0.1241  0.6077  0.7371
2            2       0      2.0  0.2155  ... -0.6417 -0.2187 -1.4080  0.6931
3            2       0      3.0  0.1792  ... -1.6210 -0.8784 -0.3876 -0.8154
4            1       1      4.0 -0.1498  ...  0.1094  0.2885 -0.3786  0.7125
...        ...     ...      ...     ...  ...     ...     ...     ...     ...
23809        0       1    527.0  0.3055  ...  0.0631  0.9171  0.5258  0.4680
23810        0       1   1138.0 -0.5565  ... -0.2084 -0.1224 -0.2715  0.3689
23811        2       1  14364.0  0.1745  ...  0.2256  0.7592  0.6656  0.3808
23812        0       0  14365.0  0.0463  ...  0.1732  0.7015 -0.6290  0.0740
23813        1       0  14366.0  0.9146  ... -3.5770 -0.4775 -2.1500 -4.2520

[23814 rows x 395 columns]
```

```
# save X and y after feature selection, encoding and other processing to use them later
with open("/content/drive/My Drive/Colab Notebooks/X.data", 'wb') as f:
  pickle.dump(x, f)
with open("/content/drive/My Drive/Colab Notebooks/y.data", 'wb') as f:
  pickle.dump(y, f)
X = x
```

SVM + Feature Selection

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=42)

model = OneVsRestClassifier(LinearSVC(random_state=0))
```

```
model = OneVsRestClassifier(LinearSVC(random_state=0))
model.fit(X_train, y_train)
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/svm/_base.py:947: ConvergenceWarning:
  "the number of iterations.", ConvergenceWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/svm/_base.py:947: ConvergenceWarning:
  "the number of iterations.", ConvergenceWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/svm/_base.py:947: ConvergenceWarning:
  "the number of iterations.", ConvergenceWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/svm/_base.py:947: ConvergenceWarning:
  "the number of iterations.", ConvergenceWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/svm/_base.py:947: ConvergenceWarning:
  "the number of iterations.", ConvergenceWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/svm/_base.py:947: ConvergenceWarning:
  "the number of iterations.", ConvergenceWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/svm/_base.py:947: ConvergenceWarning:
  "the number of iterations.", ConvergenceWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/svm/_base.py:947: ConvergenceWarning:
  "the number of iterations.", ConvergenceWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/svm/_base.py:947: ConvergenceWarning:
  "the number of iterations.", ConvergenceWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/svm/_base.py:947: ConvergenceWarning:
  "the number of iterations.", ConvergenceWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/svm/_base.py:947: ConvergenceWarning:
  "the number of iterations.", ConvergenceWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/svm/_base.py:947: ConvergenceWarning:
  "the number of iterations.", ConvergenceWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/svm/_base.py:947: ConvergenceWarning:
  "the number of iterations.", ConvergenceWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/svm/_base.py:947: ConvergenceWarning:
  "the number of iterations.", ConvergenceWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/svm/_base.py:947: ConvergenceWarning:
  "the number of iterations.", ConvergenceWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/svm/_base.py:947: ConvergenceWarning:
  "the number of iterations.", ConvergenceWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/svm/_base.py:947: ConvergenceWarning:
  "the number of iterations.", ConvergenceWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/svm/_base.py:947: ConvergenceWarning:
  "the number of iterations.", ConvergenceWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/svm/_base.py:947: ConvergenceWarning:
  "the number of iterations.", ConvergenceWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/svm/_base.py:947: ConvergenceWarning:
  "the number of iterations.", ConvergenceWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/svm/_base.py:947: ConvergenceWarning:
  "the number of iterations.", ConvergenceWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/svm/_base.py:947: ConvergenceWarning:
  "the number of iterations.", ConvergenceWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/svm/_base.py:947: ConvergenceWarning:
  "the number of iterations.", ConvergenceWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/svm/_base.py:947: ConvergenceWarning:
  "the number of iterations.", ConvergenceWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/svm/_base.py:947: ConvergenceWarning:
  "the number of iterations.", ConvergenceWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/svm/_base.py:947: ConvergenceWarning:
  "the number of iterations.", ConvergenceWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/svm/_base.py:947: ConvergenceWarning:
  "the number of iterations.", ConvergenceWarning)
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/svm/_base.py:947: ConvergenceWarning:
  "the number of iterations.", ConvergenceWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/svm/_base.py:947: ConvergenceWarning:
  "the number of iterations.", ConvergenceWarning)
```

```
predicted = model.predict(X_train)
print(accuracy_score(y_train, predicted))
print("f1-score: " + str(f1_score(y_train, predicted, average='weighted')))
```

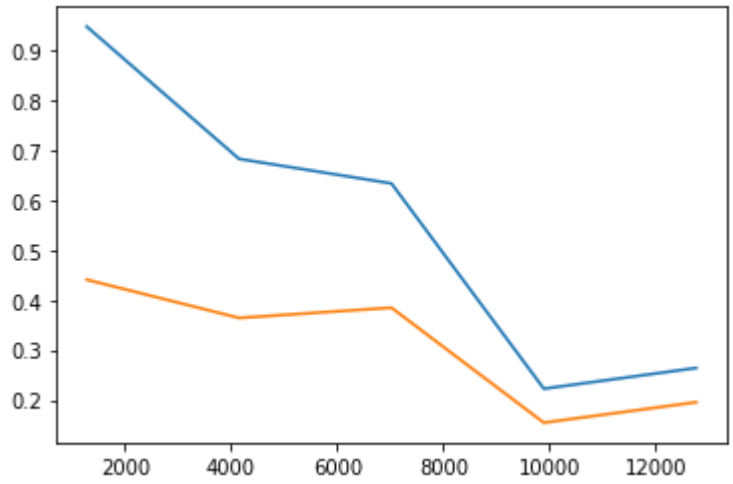
```
0.07953619554998433
f1-score: 0.4809028911550528
/usr/local/lib/python3.6/dist-packages/sklearn/metrics/_classification.py:1515: Undefin
  average, "true nor predicted", 'F-score is', len(true_sum)
```

As we can see, our accuracy is so bad. However, this problem is more complicated than just using multioutput classifier with SVM, but we had to try it for this phase because we're not seeking good predictions yet, we're still experimenting. Note that this is with feature selection, and the f-score is better here, but the accuracy isn't. This means that we're averaging good recall and precision compared to just accuracy.

```
# Learning curve

cv = ShuffleSplit(n_splits=5, test_size=0.2, random_state=0) # # 20% randomly selected test c
train_sizes, train_scores, test_scores, fit_times, _ = \
    learning_curve(model, X_train, y_train, cv=cv, n_jobs=-1, return_times=True)
```

```
# without grid search and optimal parameters
plt.plot(train_sizes, np.mean(train_scores, axis=1))
plt.plot(train_sizes, np.mean(test_scores, axis=1))
```

```
[<matplotlib.lines.Line2D at 0x7fe4654c0860>]
```



The learning curve for SVM with feature selection is weird, it gets worse with training and there is high bias, low variance. However, this is expected because the problem is multilabel and svm isn't really the choice to go for if we want good results, but it was nice to try and visualize. Now for this part (feature selection) I am still running grid search so I'm not sure if I will report the below results (if the code ever finishes :)). But we did report the results for the non-feature-selection part of this phase

```
# SVM with gridsearch CV
# X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=42)
parameters = {'estimator__C':[0.01, 0.1, 1, 10]}
svm = OneVsRestClassifier(SVC())
model = GridSearchCV(svm, parameters)
model.fit(X, y)
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/multiclass.py:75: UserWarning: Label not
  str(classes[c]))
/usr/local/lib/python3.6/dist-packages/sklearn/multiclass.py:75: UserWarning: Label not
  str(classes[c]))
/usr/local/lib/python3.6/dist-packages/sklearn/multiclass.py:75: UserWarning: Label not
  str(classes[c]))
/usr/local/lib/python3.6/dist-packages/sklearn/multiclass.py:75: UserWarning: Label not
  str(classes[c]))
/usr/local/lib/python3.6/dist-packages/sklearn/multiclass.py:75: UserWarning: Label not
  str(classes[c]))
```

```
predicted = model.predict(X_train)
print(accuracy_score(y_train, predicted))
print("f1-score: " + str(f1_score(y_train, predicted, average='weighted')))


# save - it took time to train :)
from joblib import dump, load
dump(svm, '/content/drive/My Drive/Colab Notebooks/svm_tuned_with_fs.joblib')


# learning curve
cv = ShuffleSplit(n_splits=5, test_size=0.2, random_state=0) # # 20% randomly selected test c
train_sizes, train_scores, test_scores, fit_times, _ = \
    learning_curve(model, X_train, y_train, cv=cv, n_jobs=-1, return_times=True)
```

Neural Networks + Meta Learning MAML - finn

```
import torch.nn as nn
import torch.nn.functional as F

# split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=42)
X_train = np.array(X_train)
y_train = np.array(y_train)
```

```
X_test = np.array(X_test)
y_test = np.array(y_test)

temp = []
for i in X_train:
    temp.append(i)
X_train_tensor = torch.Tensor(temp)

temp = []
for i in X_test:
  temp.append(i)
X_test_tensor = torch.Tensor(temp)
y_train_tensor = torch.Tensor(y_train)
y_test_tensor = torch.Tensor(y_test)


# deep neural network model: we tuned a little bit dropout probablities and number of layers
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(X.shape[1], 256)
        self.fc2 = nn.Linear(256, 1024)
        self.fc3 = nn.Linear(1024, 512)
        self.fc4 = nn.Linear(512, 206)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(nn.Dropout(p=0.2)(self.fc2(x)))
        x = F.relu(nn.Dropout(p=0.2)(self.fc3(x)))
        x = self.fc4(x)
        return torch.sigmoid(x)

net = Net()
print(net)
```

```
Net(
  (fc1): Linear(in_features=395, out_features=256, bias=True)
  (fc2): Linear(in_features=256, out_features=1024, bias=True)
  (fc3): Linear(in_features=1024, out_features=512, bias=True)
  (fc4): Linear(in_features=512, out_features=206, bias=True)
)
```

NN + Feature Selection (or not, uncomment loading corresponding data above)

Below we are trying a deep neural network with feature selection part of our dataset. We tuned it a little bit and tried different number of epochs, neuron number, layers number, activation functions, and loss functions. We believe the above defined network was the best we could come up with, taking into consideration the training time too.

```
# Pure deep NN

X_train = X_train_tensor
y_train = y_train_tensor
X_test = X_test_tensor
y_test = y_test_tensor

criterion = nn.BCELoss()
optimizer = optim.SGD(net.parameters(), lr=0.001)

def round_tensor(t, decimal_places=3):
  return round(t.item(), decimal_places)

def calculate_accuracy(y_true, y_pred):
  predicted = y_pred.ge(.5).view(-1) # convert <0.5 to 0 and >0.5 to 1 (this threshold we car
  y_true = y_true.reshape((y_true.shape[0]*y_true.shape[1],))
  return (y_true == predicted).sum().float() / len(y_true)

for epoch in range(1000):
  y_pred = net(X_train)
  y_pred = torch.squeeze(y_pred)
  train_loss = criterion(y_pred, y_train)
  if epoch % 100 == 0:
    train_acc = calculate_accuracy(y_train, y_pred)
    y_test_pred = net(X_test)
    y_test_pred = torch.squeeze(y_test_pred)
    test_loss = criterion(y_test_pred, y_test)
    test_acc = calculate_accuracy(y_test, y_test_pred)
    print(
f'''epoch {epoch}
Train set - loss: {round_tensor(train_loss)}, accuracy: {round_tensor(train_acc)}
Test  set - loss: {round_tensor(test_loss)}, accuracy: {round_tensor(test_acc)}
''')
  optimizer.zero_grad()
  train_loss.backward()
  optimizer.step()
```

```
epoch 0
Train set - loss: 15.99, accuracy: 0.509
Test  set - loss: 16.027, accuracy: 0.509

epoch 100
Train set - loss: 0.173, accuracy: 0.97
Test  set - loss: 0.176, accuracy: 0.97

epoch 200
Train set - loss: 0.123, accuracy: 0.985
Test  set - loss: 0.126, accuracy: 0.985

epoch 300
Train set - loss: 0.109, accuracy: 0.989
Test  set - loss: 0.112, accuracy: 0.989
```

```
epoch 400
Train set - loss: 0.103, accuracy: 0.991
Test  set - loss: 0.105, accuracy: 0.991

epoch 500
Train set - loss: 0.1, accuracy: 0.992
Test  set - loss: 0.102, accuracy: 0.992

epoch 600
Train set - loss: 0.098, accuracy: 0.993
Test  set - loss: 0.1, accuracy: 0.993

epoch 700
Train set - loss: 0.097, accuracy: 0.993
Test  set - loss: 0.099, accuracy: 0.993

epoch 800
Train set - loss: 0.095, accuracy: 0.994
Test  set - loss: 0.097, accuracy: 0.994

epoch 900
Train set - loss: 0.094, accuracy: 0.994
Test  set - loss: 0.097, accuracy: 0.994
```

```
outputs = net(X_train_tensor)
print("Log loss: " + str(log_loss(y_train_tensor.detach().numpy(), outputs.detach().numpy())))

outputs[outputs >= 0.5] = 1
outputs[outputs < 0.5] = 0
accuracy = (outputs == y_train_tensor).sum() / (y_train_tensor.shape[0]*y_train_tensor.shape[
print("Accuracy: " + str(accuracy) + "%")

f1 = f1_score(y_train_tensor.detach().numpy(), outputs.detach().numpy(), average='weighted')
print("f1-score: " + str(f1))

# auc = roc_auc_score(y_train_tensor.detach().numpy(), outputs.detach().numpy())
# print(auc)
```

```
Log loss: 9.987927913412836
Accuracy: tensor(99.4232)%
f1-score: 0.007800797356018251
```

As we can see above, the accuracy is very high but the f-score isn't good at all, meaning we're not hitting right targets most of the time. This problem seems to require more research and advanced techniques than what we tried with feature election and neural networks + MAML

MAML + Feature Selection

```
# MAML
```

```
def compute_loss(task_model):
  target = y_train_tensor
  output = task_model(X_train_tensor)
  return nn.BCELoss()(output, target)

model = net
maml = l2l.algorithms.MAML(model, lr=0.1)
opt = torch.optim.SGD(maml.parameters(), lr=0.001)
for iteration in range(100):
    opt.zero_grad()
    task_model = maml.clone()
    adaptation_loss = compute_loss(task_model)
    task_model.adapt(adaptation_loss)  # computes updated gradient, update task_model
    evaluation_loss = compute_loss(task_model)
    evaluation_loss.backward()  # gradients w.r.t. maml.parameters()
    opt.step()


outputs = model(X_train_tensor)
outputs[outputs >= 0.5] = 1
outputs[outputs < 0.5] = 0
accuracy = (outputs == y_train_tensor).sum() / (y_train_tensor.shape[0]*y_train_tensor.shape[
print("Accuracy: " + str(accuracy) + "%")

f1 = f1_score(y_train_tensor.detach().numpy(), outputs.detach().numpy(), average='weighted')
print("f1-score: " + str(f1))
```

```
Accuracy: tensor(99.4292)%
f1-score: 0.008755365607214816
```