DA24M011

October 13, 2024

##

DA5401 Data Analytics Labarotary

###

Assignment 7 - Submitted by: DA24M011 - Nandhakishore C S

Important:

Check the file output.txt for the verbose from gridsearch and the classification report for task 1. Check the file smote.txt for the verbose and classification for task 2. As the terminal output in a ipynb notebook was long and it affected exporting the notebook as a .pdf file, I saved the terminal outputs in separate file.

(From Question)

Let's learn to deal with class-imbalance this time! We will consider the IDA2016 Challengedataset for our experimentation. The dataset is a binary classification y = {'pos', 'neg'} problem with 170 features and 60,000 data points. The craziness here is that the class ratio is 1:59, that is, for every positive data point, there are 59 negative data points in the training data. The challenge dataset has a training file(aps_failure_training_set.csv) and a testing file (aps_failure_test_set.csv). We will consider only the training file for our experimentation

Task 1 [20 Points] (From Question)

Split the data file (aps_failure_training_set.csv) into train and test partitions. Build baseline classifiers {SVC, LogReg and DecisionTree} by cross-validating the best hyper-parameters of the respective models. For SVC, the hyperparametes are {kernel, kernel-params}; for LogReg {regularization choice L1/L2, regularization params}; and for DT {depth, leaf size}.Upon using GridSearchCV, the best parameters are to be found. Note that, GridSearchCV does 5-fold CV by default, which is sufficient for us. Once the parameters are fixed, you will learn the models on the train partition and report the performance metrics on the train and test partitions

Importing Libraries

[1]:
```
# Data handling, visualisation and matrix operation
import pandas as pd ␣
  ↪
  ↪type: ignore
```

```
import numpy as np␣
↪
↪type: ignore
import matplotlib.pyplot as␣
↪plt
↪type: ignore

# Data Preprocessing
from sklearn.preprocessing import LabelEncoder,␣
↪StandardScaler                                       # type: ignore
from sklearn.model_selection import train_test_split,␣
↪GridSearchCV                                    # type: ignore

# Machine learning algorithms - Support vector classifier, Logistic Rgeression␣
↪and Decision Trees
from sklearn.linear_model import LogisticRegression␣
↪                                                       # type:␣
↪ignore
from sklearn.tree import DecisionTreeClassifier␣
↪                                                           #␣
↪type: ignore
from sklearn.svm import␣
↪SVC
↪type: ignore

# Class imbalance learning
from imblearn.over_sampling import␣
↪SMOTE
↪type: ignore
from imblearn.under_sampling import␣
↪RandomUnderSampler                                                  #␣
↪type: ignore
from imblearn.pipeline import␣
↪Pipeline
↪type: ignore

from sklearn.utils import␣
↪class_weight
↪type: ignore
from sklearn.utils.class_weight import compute_sample_weight,␣
↪compute_class_weight        # type: ignore

# Ensemble Learning
from sklearn.ensemble import RandomForestClassifier,␣
↪GradientBoostingClassifier                # type: ignore
```

```
# Classification Metrics
from sklearn.metrics import f1_score, classification_report # type: ignore

import warnings
warnings.filterwarnings("ignore")
```

Importing Dataset

[2]:
```
df_path = '/Users/nandhakishorecs/Documents/IITM/Jul_2024/DA5401/Assignments/Assignment7/to_uci/aps_failure_training_set.csv'
df_raw = pd.read_csv(df_path, skiprows=20)
```

[3]:
```
df_raw
```

[3]:
```
      class   aa_000 ab_000       ac_000 ad_000 ae_000 af_000 ag_000 ag_001  \
0       neg    76698     na   2130706438    280      0      0      0      0
1       neg    33058     na            0     na      0      0      0      0
2       neg    41040     na          228    100      0      0      0      0
3       neg       12      0           70     66      0     10      0      0
4       neg    60874     na         1368    458      0      0      0      0
…       …       …      …           …      …      …      …      …      …
59995   neg   153002     na          664    186      0      0      0      0
59996   neg     2286     na   2130706538    224      0      0      0      0
59997   neg      112      0   2130706432     18      0      0      0      0
59998   neg    80292     na   2130706432    494      0      0      0      0
59999   neg    40222     na          698    628      0      0      0      0

      ag_002  …   ee_002  ee_003   ee_004   ee_005   ee_006  ee_007  ee_008  \
0          0  …  1240520  493384   721044   469792   339156  157956   73224
1          0  …   421400  178064   293306   245416   133654   81140   97576
2          0  …   277378  159812   423992   409564   320746  158022   95128
3          0  …      240      46       58       44       10       0       0
4          0  …   622012  229790   405298   347188   286954  311560  433954
…        …   …       …       …        …        …        …       …
59995      0  …   998500  566884  1290398  1218244  1019768  717762  898642
59996      0  …    10578    6760    21126    68424      136       0       0
59997      0  …      792     386      452      144      146    2622       0
59998      0  …   699352  222654   347378   225724   194440  165070  802280
59999      0  …   440066  183200   344546   254068   225148  158304  170384

      ee_009 ef_000 eg_000
0          0      0      0
1       1500      0      0
2        514      0      0
3          0      4     32
```

```
4        1218      0      0
…        …     …     …
59995   28588      0      0
59996       0      0      0
59997       0      0      0
59998  388422      0      0
59999     158      0      0

[60000 rows x 171 columns]
```

The dataset has 'na' as elements, which are basically numpy.nan values.

Deep Copy of dataframes to remove 'na' values

[5]:
```
df_np = df_raw
df = df_raw
df.columns
```

[5]:
```
Index(['class', 'aa_000', 'ab_000', 'ac_000', 'ad_000', 'ae_000', 'af_000',
       'ag_000', 'ag_001', 'ag_002',
       …
       'ee_002', 'ee_003', 'ee_004', 'ee_005', 'ee_006', 'ee_007', 'ee_008',
       'ee_009', 'ef_000', 'eg_000'],
      dtype='object', length=171)
```

[6]:
```
df.dtypes
```

[6]:
```
class     object
aa_000     int64
ab_000    object
ac_000    object
ad_000    object
           …
ee_007    object
ee_008    object
ee_009    object
ef_000    object
eg_000    object
Length: 171, dtype: object
```

Also, the elements in the dataframe are not numbers, rather they are objects with non - float / integer values. we need to convert them into numbers

[7]:
```
df = df.drop(columns = ['class'])
df_np = df_np.drop(columns = ['class'])
```

Converting dataframe into a numpy array to check for na values and replacing the 'na' values with the mean of the columns.

[8]:
```
df
```

[8]:
```
       aa_000 ab_000      ac_000 ad_000 ae_000 af_000 ag_000 ag_001 ag_002  \
0       76698     na  2130706438    280      0      0      0      0      0
1       33058     na           0     na      0      0      0      0      0
2       41040     na         228    100      0      0      0      0      0
3          12      0          70     66      0     10      0      0      0
4       60874     na        1368    458      0      0      0      0      0
…         …    …          …    …    …    …     …    …    …
59995  153002     na         664    186      0      0      0      0      0
59996    2286     na  2130706538    224      0      0      0      0      0
59997     112      0  2130706432     18      0      0      0      0      0
59998   80292     na  2130706432    494      0      0      0      0      0
59999   40222     na         698    628      0      0      0      0      0

      ag_003  …   ee_002  ee_003   ee_004   ee_005   ee_006  ee_007  ee_008  \
0          0  …  1240520  493384   721044   469792   339156  157956   73224
1          0  …   421400  178064   293306   245416   133654   81140   97576
2          0  …   277378  159812   423992   409564   320746  158022   95128
3        318  …      240      46       58       44       10       0       0
4          0  …   622012  229790   405298   347188   286954  311560  433954
…        …   …       …      …       …       …      …       …
59995   2564  …   998500  566884  1290398  1218244  1019768  717762  898642
59996      0  …    10578    6760    21126    68424      136       0       0
59997      0  …      792     386      452      144      146    2622       0
59998      0  …   699352  222654   347378   225724   194440  165070  802280
59999      0  …   440066  183200   344546   254068   225148  158304  170384

       ee_009 ef_000 eg_000
0           0      0      0
1        1500      0      0
2         514      0      0
3           0      4     32
4        1218      0      0
…         …    …     …
59995   28588      0      0
59996       0      0      0
59997       0      0      0
59998  388422      0      0
59999     158      0      0

[60000 rows x 170 columns]
```

[9]:
```
df_np = np.array(df_np)
```

[10]:
```
df_np
```

[10]: array([[76698, 'na', '2130706438', …, '0', '0', '0'],
       [33058, 'na', '0', …, '1500', '0', '0'],
       [41040, 'na', '228', …, '514', '0', '0'],
       …,
       [112, '0', '2130706432', …, '0', '0', '0'],
       [80292, 'na', '2130706432', …, '388422', '0', '0'],
       [40222, 'na', '698', …, '158', '0', '0']], dtype=object)

[11]:
```
for i in range(0, df_np.shape[0]):
    for j in range(0, df_np.shape[1]):
        if(df_np[i][j] == 'na'):
            df_np[i][j] = np.nan
        df_np[i][j] = float(df_np[i][j])
```

[12]:
```
df_np
```

[12]: array([[76698.0, nan, 2130706438.0, …, 0.0, 0.0, 0.0],
       [33058.0, nan, 0.0, …, 1500.0, 0.0, 0.0],
       [41040.0, nan, 228.0, …, 514.0, 0.0, 0.0],
       …,
       [112.0, 0.0, 2130706432.0, …, 0.0, 0.0, 0.0],
       [80292.0, nan, 2130706432.0, …, 388422.0, 0.0, 0.0],
       [40222.0, nan, 698.0, …, 158.0, 0.0, 0.0]], dtype=object)

[13]:
```
col_means = np.nanmean(df_np, axis = 0)
```

[14]:
```
df = np.array(df)
df
```

[14]: array([[76698, 'na', '2130706438', …, '0', '0', '0'],
       [33058, 'na', '0', …, '1500', '0', '0'],
       [41040, 'na', '228', …, '514', '0', '0'],
       …,
       [112, '0', '2130706432', …, '0', '0', '0'],
       [80292, 'na', '2130706432', …, '388422', '0', '0'],
       [40222, 'na', '698', …, '158', '0', '0']], dtype=object)

[15]:
```
for i in range(0, df.shape[0]):
    for j in range(0, df.shape[1]):
        if(df[i][j] == 'na'):
            df[i][j] = col_means[j]
        df[i][j] = float(df[i][j])
```

[16]:
```
class_column = df_raw['class']
print('len of classes:\t', len(class_column))
```

len of classes:	60000

[17]:
```
print('len of the header', len(df_raw.columns ))
```

```
len of the header 171
```

[18]:
```
df = np.c_[class_column, df]
#df = np.r_[df_raw.columns, df]
df
```

[18]:
```
array([['neg', 76698.0, 0.7131885012069343, …, 0.0, 0.0, 0.0],
       ['neg', 33058.0, 0.7131885012069343, …, 1500.0, 0.0, 0.0],
       ['neg', 41040.0, 0.7131885012069343, …, 514.0, 0.0, 0.0],
       …,
       ['neg', 112.0, 0.0, …, 0.0, 0.0, 0.0],
       ['neg', 80292.0, 0.7131885012069343, …, 388422.0, 0.0, 0.0],
       ['neg', 40222.0, 0.7131885012069343, …, 158.0, 0.0, 0.0]],
      dtype=object)
```

[19]:
```
header = np.array(df_raw.columns)
header
```

[19]:
```
array(['class', 'aa_000', 'ab_000', 'ac_000', 'ad_000', 'ae_000',
       'af_000', 'ag_000', 'ag_001', 'ag_002', 'ag_003', 'ag_004',
       'ag_005', 'ag_006', 'ag_007', 'ag_008', 'ag_009', 'ah_000',
       'ai_000', 'aj_000', 'ak_000', 'al_000', 'am_0', 'an_000', 'ao_000',
       'ap_000', 'aq_000', 'ar_000', 'as_000', 'at_000', 'au_000',
       'av_000', 'ax_000', 'ay_000', 'ay_001', 'ay_002', 'ay_003',
       'ay_004', 'ay_005', 'ay_006', 'ay_007', 'ay_008', 'ay_009',
       'az_000', 'az_001', 'az_002', 'az_003', 'az_004', 'az_005',
       'az_006', 'az_007', 'az_008', 'az_009', 'ba_000', 'ba_001',
       'ba_002', 'ba_003', 'ba_004', 'ba_005', 'ba_006', 'ba_007',
       'ba_008', 'ba_009', 'bb_000', 'bc_000', 'bd_000', 'be_000',
       'bf_000', 'bg_000', 'bh_000', 'bi_000', 'bj_000', 'bk_000',
       'bl_000', 'bm_000', 'bn_000', 'bo_000', 'bp_000', 'bq_000',
       'br_000', 'bs_000', 'bt_000', 'bu_000', 'bv_000', 'bx_000',
       'by_000', 'bz_000', 'ca_000', 'cb_000', 'cc_000', 'cd_000',
       'ce_000', 'cf_000', 'cg_000', 'ch_000', 'ci_000', 'cj_000',
       'ck_000', 'cl_000', 'cm_000', 'cn_000', 'cn_001', 'cn_002',
       'cn_003', 'cn_004', 'cn_005', 'cn_006', 'cn_007', 'cn_008',
       'cn_009', 'co_000', 'cp_000', 'cq_000', 'cr_000', 'cs_000',
       'cs_001', 'cs_002', 'cs_003', 'cs_004', 'cs_005', 'cs_006',
       'cs_007', 'cs_008', 'cs_009', 'ct_000', 'cu_000', 'cv_000',
       'cx_000', 'cy_000', 'cz_000', 'da_000', 'db_000', 'dc_000',
       'dd_000', 'de_000', 'df_000', 'dg_000', 'dh_000', 'di_000',
       'dj_000', 'dk_000', 'dl_000', 'dm_000', 'dn_000', 'do_000',
       'dp_000', 'dq_000', 'dr_000', 'ds_000', 'dt_000', 'du_000',
       'dv_000', 'dx_000', 'dy_000', 'dz_000', 'ea_000', 'eb_000',
       'ec_00', 'ed_000', 'ee_000', 'ee_001', 'ee_002', 'ee_003',
```

```
       'ee_004', 'ee_005', 'ee_006', 'ee_007', 'ee_008', 'ee_009',
       'ef_000', 'eg_000'], dtype=object)
```

[20]:
```
df = pd.DataFrame(df)
df.columns = df_raw.columns
df
```

[20]:
```
      class    aa_000    ab_000        ac_000         ad_000 ae_000 af_000  \
0       neg   76698.0  0.713189  2130706438.0          280.0    0.0    0.0
1       neg   33058.0  0.713189           0.0  190620.639314    0.0    0.0
2       neg   41040.0  0.713189         228.0          100.0    0.0    0.0
3       neg      12.0       0.0          70.0           66.0    0.0   10.0
4       neg   60874.0  0.713189        1368.0          458.0    0.0    0.0
…       …        …        …           …              …      …     …
59995   neg  153002.0  0.713189         664.0          186.0    0.0    0.0
59996   neg    2286.0  0.713189  2130706538.0          224.0    0.0    0.0
59997   neg     112.0       0.0  2130706432.0           18.0    0.0    0.0
59998   neg   80292.0  0.713189  2130706432.0          494.0    0.0    0.0
59999   neg   40222.0  0.713189         698.0          628.0    0.0    0.0

      ag_000 ag_001 ag_002  …     ee_002    ee_003     ee_004     ee_005  \
0        0.0    0.0    0.0  …  1240520.0  493384.0   721044.0   469792.0
1        0.0    0.0    0.0  …   421400.0  178064.0   293306.0   245416.0
2        0.0    0.0    0.0  …   277378.0  159812.0   423992.0   409564.0
3        0.0    0.0    0.0  …      240.0      46.0       58.0       44.0
4        0.0    0.0    0.0  …   622012.0  229790.0   405298.0   347188.0
…        …      …      …   …        …         …          …
59995    0.0    0.0    0.0  …   998500.0  566884.0  1290398.0  1218244.0
59996    0.0    0.0    0.0  …    10578.0    6760.0    21126.0    68424.0
59997    0.0    0.0    0.0  …      792.0     386.0      452.0      144.0
59998    0.0    0.0    0.0  …   699352.0  222654.0   347378.0   225724.0
59999    0.0    0.0    0.0  …   440066.0  183200.0   344546.0   254068.0

          ee_006    ee_007    ee_008    ee_009 ef_000 eg_000
0       339156.0  157956.0   73224.0       0.0    0.0    0.0
1       133654.0   81140.0   97576.0    1500.0    0.0    0.0
2       320746.0  158022.0   95128.0     514.0    0.0    0.0
3           10.0       0.0       0.0       0.0    4.0   32.0
4       286954.0  311560.0  433954.0    1218.0    0.0    0.0
…            …         …         …         …      …
59995  1019768.0  717762.0  898642.0   28588.0    0.0    0.0
59996      136.0       0.0       0.0       0.0    0.0    0.0
59997      146.0    2622.0       0.0       0.0    0.0    0.0
59998   194440.0  165070.0  802280.0  388422.0    0.0    0.0
59999   225148.0  158304.0  170384.0     158.0    0.0    0.0

[60000 rows x 171 columns]
```

The cells with 'na' values are converted into floating point elements and the header is added back for ease of handling data

[21]:
```
df.describe()
```

[21]:
```
        class   aa_000        ab_000   ac_000         ad_000   ae_000  \
count   60000  60000.0  60000.000000  60000.0   60000.000000  60000.0
unique      2  22095.0     30.000000   2062.0    1887.000000    334.0
top       neg      8.0      0.713189      0.0  190620.639314      0.0
freq    59000   1023.0  46329.000000   8752.0   14861.000000  55543.0

         af_000   ag_000   ag_001   ag_002  …   ee_002   ee_003   ee_004  \
count   60000.0  60000.0  60000.0  60000.0  …  60000.0  60000.0  60000.0
unique    419.0    155.0    618.0   2423.0  …  34489.0  31712.0  35189.0
top         0.0      0.0      0.0      0.0  …      0.0      0.0      0.0
freq    55476.0  59133.0  58587.0  56181.0  …   1364.0   1557.0   1797.0

         ee_005   ee_006   ee_007   ee_008   ee_009   ef_000   eg_000
count   60000.0  60000.0  60000.0  60000.0  60000.0  60000.0  60000.0
unique  36289.0  31796.0  30470.0  24214.0   9725.0     29.0     50.0
top         0.0      0.0      0.0      0.0      0.0      0.0      0.0
freq     2814.0   4458.0   7898.0  17280.0  31863.0  57021.0  56794.0

[4 rows x 171 columns]
```

[22]:
```
df.isnull().sum()
```

[22]:
```
class     0
aa_000    0
ab_000    0
ac_000    0
ad_000    0
         ..
ee_007    0
ee_008    0
ee_009    0
ef_000    0
eg_000    0
Length: 171, dtype: int64
```

No null / nan values after data cleaning

[23]:
```
X = np.array(df.drop(columns = ['class']))
y = np.array(df['class'])
```

Extracting features and classes from dataset for classification

[ ]:
```python
from sklearn.preprocessing import LabelEncoder

y = LabelEncoder().fit_transform(y)
plt.hist(y)
```

From the histogtram we can see that, there is skewness and the dataset is highly imbalanced with respect to classes (as from question)!

Also, we are lebel encoding the data for the ease of doing classification

[28]:
```python
train_X, test_X, train_y, test_y = train_test_split(
    X, y, test_size = 0.3, shuffle=True, random_state=42
)
```

Splitting the data into train test datasets

[29]:
```python
test_y = LabelEncoder().fit_transform(test_y)
```

[ ]:
```python
dt_clf = DecisionTreeClassifier()

dt_parameter_grid = {
        'max_depth' : [1, 10, 50, None],
        'max_leaf_nodes' : [2, 64, 128],
        'min_samples_leaf' : [8, 16, 32, 64],
}

grid_search = GridSearchCV(
    estimator = dt_clf,
        param_grid = dt_parameter_grid,
    cv = 5,
        n_jobs = -1,
        verbose = 3,
        scoring = 'f1_macro'
)

grid_search.fit(train_X, train_y)

best_tree_clf = grid_search.best_estimator_
print(best_tree_clf)
best_params_dt = grid_search.best_params_
pred_y = best_tree_clf.predict(test_X)
print(pred_y)
dt_test_f1 = f1_score(test_y, pred_y)
print(dt_test_f1)

dt_best = DecisionTreeClassifier(**best_params_dt).fit(train_X, train_y)

print("Decision Tree Performance on Train Set:")
```

```
print(classification_report(train_y, dt_best.predict(train_X),␣
 ↪target_names=['Negative', 'Positive']))

print("Decision Tree Performance on Test Set:")
print(classification_report(test_y, dt_best.predict(test_X),␣
 ↪target_names=['Negative', 'Positive']))
```

[ ]:
```
train_XX = StandardScaler().fit_transform(train_X)

logreg_clf =  LogisticRegression(max_iter = 1e+6)

logreg_parameter_grid = {
        'C': [1000, 100, 10, 1],
        'penalty': ['l1', 'l2']
}

grid_search = GridSearchCV(
    estimator = logreg_clf,
        param_grid = logreg_parameter_grid,
    cv = 5,
        n_jobs = -1,
        verbose = 3,
        scoring = 'f1_macro'
)

grid_search.fit(train_XX, train_y)

test_XX = StandardScaler().fit_transform(test_X)

best_logreg_clf = grid_search.best_estimator_
best_params_logreg = grid_search.best_params_
print(best_logreg_clf)

pred_y = best_logreg_clf.predict(test_XX)
logreg_test_f1 = f1_score(test_y, pred_y)
print(logreg_test_f1)

logreg_best = LogisticRegression(**best_params_logreg).fit(train_X, train_y)

print("Logistic Regression Performance on Train Set:")
print(classification_report(train_y, logreg_best.predict(train_XX),␣
 ↪target_names=['Negative', 'Positive']))

print("Logistic Regression Performance on Test Set:")
print(classification_report(test_y, logreg_best.predict(test_XX),␣
 ↪target_names=['Negative', 'Positive']))
```

[ ]:
```
svm_clf = SVC()

svm_parameter_grid = {
    'gamma': [0, 1, 10],
    'kernel': ['rbf', 'poly']
}

grid_search = GridSearchCV(
    estimator = svm_clf,
        param_grid = svm_parameter_grid,
    cv = 5,
        n_jobs = -1,
        verbose = 3,
        scoring = 'f1_macro'
)

grid_search.fit(train_X, train_y)
# %%
best_svm_clf = grid_search.best_estimator_
print(best_svm_clf)

best_params_svc = grid_search.best_params_
pred_y = best_svm_clf.predict(test_X)
svm_test_f1 = f1_score(test_y, pred_y)
print(svm_test_f1)

svc_best = SVC(**best_params_svc).fit(train_X, train_y)
print("SVC Performance on Train Set:")
print(classification_report(train_y, svc_best.predict(train_X),␣
 ↪target_names=['Negative', 'Positive']))

print("SVC Performance on Test Set:")
print(classification_report(test_y, svc_best.predict(test_X),␣
 ↪target_names=['Negative', 'Positive']))
```

Please refer 'output.txt' for the output of above cells with their respective f1 score(s) and classification report(s)

Task 2 [30 Points] (From Question)

Now, we want to address the class imbalance via multiple approaches. You are expected to apply the following in all the three families of classifiers.

Consider undersampling the majority class and/or oversampling the minority class.

Consider using class_weight which is inversely proportional to the class population.

Consider using sample_weights, where you may assign a penalty for misclassifying every data point depending on the class it falls in.

Consider any other creative ideas to address the class imbalance.

The goal here is the classification performance metric (macro average F__1) of the hacked classifiers should be better than the baseline classifiers

Please refer 'smote.txt' for the output of above cells with their respective f1 score(s) and classification report(s)

a. classification after undersampling the majority class / oversampling the minoruty class

[ ]:
```
# Oversampling the minority class
oversample = SMOTE()
X_train_oversampled, y_train_oversampled = oversample.fit_resample(train_X,␣
↪train_y)
X_test_oversampled, y_test_oversampled = oversample.fit_resample(test_X, test_y)
```

[ ]:
```
print('\nFit on oversampled data: \n')

sv = SVC().fit(X_train_oversampled, y_train_oversampled)
print("SVM Performance on Test Set:")
print(classification_report(y_test_oversampled, sv.predict(X_test_oversampled),␣
↪target_names=['Negative', 'Positive']))
print("SVM Performance on Train Set:")
print(classification_report(y_train_oversampled, sv.
↪predict(X_train_oversampled), target_names=['Negative', 'Positive']))

lg = LogisticRegression(max_iter = 1000000).fit(X_train_oversampled,␣
↪y_train_oversampled)
print("Logistic Regression Performance on Test Set:")
print(classification_report(y_test_oversampled, lg.predict(X_test_oversampled),␣
↪target_names=['Negative', 'Positive']))
print("Logistic Regression Performance on Train Set:")
print(classification_report(y_train_oversampled, lg.
↪predict(X_train_oversampled), target_names=['Negative', 'Positive']))

dt = DecisionTreeClassifier().fit(X_train_oversampled, y_train_oversampled)
print("Decision Tree Performance on Test Set:")
print(classification_report(y_test_oversampled, dt.predict(X_test_oversampled),␣
↪target_names=['Negative', 'Positive']))
print("Decision Tree Performance on Train Set:")
print(classification_report(y_train_oversampled, dt.
↪predict(X_train_oversampled), target_names=['Negative', 'Positive']))
```

[ ]:
```
# Undersampling the majority class
undersample = RandomUnderSampler()
X_train_undersampled, y_train_undersampled = undersample.fit_resample(train_X,␣
↪train_y)
```

```
X_test_undersampled, y_test_undersampled = undersample.fit_resample(test_X,␣
 ↪test_y)
```

[ ]:
```
sv = SVC().fit(X_train_undersampled, y_train_undersampled)
print("SVM Performance on Test Set:")
print(classification_report(y_test_undersampled, sv.
 ↪predict(X_test_undersampled), target_names=['Negative', 'Positive']))
print("SVM Performance on Train Set:")
print(classification_report(y_train_undersampled, sv.
 ↪predict(X_train_undersampled), target_names=['Negative', 'Positive']))

lg = LogisticRegression(max_iter = 1000000).fit(X_train_undersampled,␣
 ↪y_train_undersampled)
print("Logistic Regression Performance on Test Set:")
print(classification_report(y_test_undersampled, lg.
 ↪predict(X_test_undersampled), target_names=['Negative', 'Positive']))
print("Logistic Regression Performance on Train Set:")
print(classification_report(y_train_undersampled, lg.
 ↪predict(X_train_undersampled), target_names=['Negative', 'Positive']))

dt = DecisionTreeClassifier().fit(X_train_undersampled, y_train_undersampled)
print("Decision Tree Performance on Test Set:")
print(classification_report(y_test_undersampled, dt.
 ↪predict(X_test_undersampled), target_names=['Negative', 'Positive']))
print("Decision Tree Performance on Train Set:")
print(classification_report(y_train_undersampled, dt.
 ↪predict(X_train_undersampled), target_names=['Negative', 'Positive']))
```

b. classification using class weight argument in the classifiers

[ ]:
```
# Setting class weights inversely proportional to class frequencies
svc_weighted = SVC(class_weight='balanced')
logreg_weighted = LogisticRegression(class_weight='balanced',␣
 ↪solver='liblinear')
dt_weighted = DecisionTreeClassifier(class_weight='balanced')
```

c. classification using sample weights

[ ]:
```
# Assign sample weights
class_weights = dict(zip([0, 1], compute_class_weight('balanced',␣
 ↪classes=[1,0],  y=train_y)))
sample_weights =  compute_sample_weight(class_weight='balanced', y = train_y ) ␣
 ↪#train_y.map(class_weights)

# Fitting models with sample weights
svc_sample_weights = SVC().fit(train_X, train_y, sample_weight=sample_weights)
```

```
logreg_sample_weights = LogisticRegression(solver='liblinear').fit(train_X,␣
  ↪train_y, sample_weight=sample_weights)
dt_sample_weights = DecisionTreeClassifier().fit(train_X, train_y,␣
  ↪sample_weight=sample_weights)
```

d. usign ensemble methods to do classification

[ ]:
```
rf = RandomForestClassifier(class_weight='balanced')
gb = GradientBoostingClassifier()

rf.fit(train_X, train_y)
gb.fit(train_X, train_y)
```

Results of classification after doing hacks

[ ]:
```
print("Weighted SVC Performance on Test Set:")
print(classification_report(test_y, svc_weighted.predict(test_X),␣
  ↪target_names=['Negative', 'Positive']))

print("Weighted Logistic Regression Performance on Test Set:")
print(classification_report(test_y, logreg_weighted.predict(test_X),␣
  ↪target_names=['Negative', 'Positive']))

print("Weighted Decision Tree Performance on Test Set:")
print(classification_report(test_y, dt_weighted.predict(test_X),␣
  ↪target_names=['Negative', 'Positive']))

print("Random Forest Performance on Test Set:")
print(classification_report(test_y, rf.predict(test_X),␣
  ↪target_names=['Negative', 'Positive']))

print("Gradient Boosting Performance on Test Set:")
print(classification_report(test_y, gb.predict(test_X),␣
  ↪target_names=['Negative', 'Positive']))
```

###

Conclusion:

When classification is done witth class imbalance, the F1 score for classification for class 'neg' is very close to 1 and for class 'pos', the value is less than 0.7 and for svm it is to null. This is due to the fact that, the population of classes with 'pos' labels is very skewed to an extent such that, the model takes it up as noise!

When classification is done witth class oversampling / undersampling, the F1 score for classification for class 'neg' and class 'pos', are similar nearly stable at 0.9. But Oversampling / Undersampling might not be feasible because not all times a data point can be estimated into a probability distribution for sampling.

When classification is done with class weights or sample weights (with penalty), the model(s) perform well and the f1 score is 0.99 for all the three models. The best result recorded is decision trees with f1 for negative class as .99 and .60 for positive class

From the accuracy score during the hyper parameter tuning phase, we can see that, all three classifiers (logistic regression, svm and decision trees) have accuracies not more than 60%. Thus we can use different classifiers as an ensemble and random forest and gradientboosting are built and the F1 score is best for random forest at 0.91 for negative class and 0.68 for negative class!