glen

October 11, 2024

1 ASSIGNMENT 7

2 DA24C005

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.decomposition import PCA
from sklearn.tree import DecisionTreeClassifier
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import GridSearchCV, train_test_split
from sklearn.metrics import f1_score, precision_score, recall_score,␣
 ↪accuracy_score
from imblearn.over_sampling import SMOTE
```

2.1 TASK 1

[2]:
```
data=pd.read_csv('aps_failure_training_set.csv', skiprows=20)
data.head()
```

[2]:

	class	aa_000	ab_000	ac_000	ad_000	ae_000	af_000	ag_000	ag_001	ag_002	\
0	neg	76698	na	2130706438	280	0	0	0	0	0	
1	neg	33058	na	0	na	0	0	0	0	0	
2	neg	41040	na	228	100	0	0	0	0	0	
3	neg	12	0	70	66	0	10	0	0	0	
4	neg	60874	na	1368	458	0	0	0	0	0	

	…	ee_002	ee_003	ee_004	ee_005	ee_006	ee_007	ee_008	ee_009	ef_000	\
0	…	1240520	493384	721044	469792	339156	157956	73224	0	0	
1	…	421400	178064	293306	245416	133654	81140	97576	1500	0	
2	…	277378	159812	423992	409564	320746	158022	95128	514	0	
3	…	240	46	58	44	10	0	0	0	4	
4	…	622012	229790	405298	347188	286954	311560	433954	1218	0	

```
  eg_000
0      0
1      0
2      0
3     32
4      0

[5 rows x 171 columns]
```

[3]:
```python
data.replace('na',np.nan, inplace = True)
```

[4]:
```python
X = data.drop('class', axis=1)
y = data['class']
```

[5]:
```python
X.head()
```

[5]:
```
    aa_000 ab_000      ac_000 ad_000 ae_000 af_000 ag_000 ag_001 ag_002 ag_003  \
0    76698    NaN  2130706438    280      0      0      0      0      0      0
1    33058    NaN           0    NaN      0      0      0      0      0      0
2    41040    NaN         228    100      0      0      0      0      0      0
3       12      0          70     66      0     10      0      0      0    318
4    60874    NaN        1368    458      0      0      0      0      0      0

   …   ee_002  ee_003  ee_004  ee_005  ee_006  ee_007  ee_008 ee_009 ef_000  \
0  …  1240520  493384  721044  469792  339156  157956   73224      0      0
1  …   421400  178064  293306  245416  133654   81140   97576   1500      0
2  …   277378  159812  423992  409564  320746  158022   95128    514      0
3  …      240      46      58      44      10       0       0      0      4
4  …   622012  229790  405298  347188  286954  311560  433954   1218      0

  eg_000
0      0
1      0
2      0
3     32
4      0

[5 rows x 170 columns]
```

[6]:
```python
null = X.isna().mean() * 100
above_80 = (null> 80).sum()
_60_80= ((null > 60) & (null <= 80)).sum()
_40_60 = ((null > 40) & (null <= 60)).sum()
_20_40 = ((null > 20) & (null <= 40)).sum()
_0_20 = ((null > 0) & (null <= 20)).sum()
```

```
null_values = pd.DataFrame({
    'Percentage of Null values': ['Above 80%', '60-80%', '40-60%', '20-40%',␣
 ↪'0-20%'],
    'Number of Columns': [above_80, _60_80, _40_60, _20_40, _0_20]
})

null_values
```

[6]:

```
  Percentage of Null values  Number of Columns
0                 Above 80%                  2
1                    60-80%                  6
2                    40-60%                  1
3                    20-40%                 15
4                     0-20%                145
```

From this table, we infer that 169 columns of the dataset have null values. We have to do imputation to handle the missing data. We use a median imputer beacause median is not sensitive to outliers

[7]:
```
imputer = SimpleImputer(strategy='median')
X = pd.DataFrame(imputer.fit_transform(X), columns = X.columns)
```

[8]:
```
X.head()
```

[8]:

```
    aa_000  ab_000        ac_000  ad_000  ae_000  af_000  ag_000  ag_001  \
0  76698.0     0.0  2.130706e+09   280.0     0.0     0.0     0.0     0.0
1  33058.0     0.0  0.000000e+00   126.0     0.0     0.0     0.0     0.0
2  41040.0     0.0  2.280000e+02   100.0     0.0     0.0     0.0     0.0
3     12.0     0.0  7.000000e+01    66.0     0.0    10.0     0.0     0.0
4  60874.0     0.0  1.368000e+03   458.0     0.0     0.0     0.0     0.0

   ag_002  ag_003  …     ee_002    ee_003    ee_004    ee_005    ee_006  \
0     0.0     0.0  …  1240520.0  493384.0  721044.0  469792.0  339156.0
1     0.0     0.0  …   421400.0  178064.0  293306.0  245416.0  133654.0
2     0.0     0.0  …   277378.0  159812.0  423992.0  409564.0  320746.0
3     0.0   318.0  …      240.0      46.0      58.0      44.0      10.0
4     0.0     0.0  …   622012.0  229790.0  405298.0  347188.0  286954.0

     ee_007    ee_008  ee_009  ef_000  eg_000
0  157956.0   73224.0     0.0     0.0     0.0
1   81140.0   97576.0  1500.0     0.0     0.0
2  158022.0   95128.0   514.0     0.0     0.0
3       0.0       0.0     0.0     4.0    32.0
4  311560.0  433954.0  1218.0     0.0     0.0

[5 rows x 170 columns]
```

[9]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=42, stratify=y)
```

We can observe that the columns of the dataset have very different ranges. By applying scaling on the dataset, like `StandardScaler` we can ensure that different features are on the same scale. This helps to make sure that all features contribute equally towards the model learning process, irrespective of their scales. For SVC, scaling is necessary and for Logistic Regression, scaling helps to converge the model faster.

For a feature x, `StandardScaler` does the following:

$$z = \frac{x - \mu}{\sigma}$$

It normalises the feature to a 0 μ, 1 σ distribution

[10]:
```
scaler = StandardScaler()
X_train= scaler.fit_transform(X_train)
X_test= scaler.transform(X_test)
```

We store the parameters of the different classifiers in a dictionary. While doing hyperparameter tuning (Grid Search), we can access the parameter grid of the classifiers using this dictionary

[11]:
```
param_grid = {"svc": {'kernel': ['rbf', 'sigmoid'], "gamma":["scale", "auto"]},
"logreg": {'penalty': ['l1', 'l2'], 'C': [0.01,0.1,1], 'solver': ['liblinear']},
"dt": {'max_depth': [25, 30, 35], 'min_samples_leaf': [3,5,7]}}
```

This function is used to evaluate the performance of different classifier models. We calculate the performance metrics, F1 score, Accuracy, Recall, Precision using the macro-average.

[12]:
```
def model_performance(model, X, y):
    y_pred = model.predict(X)
    a= np.round(f1_score(y, y_pred, average='macro'), 4)
    print('F1 Score:', a)
    print('Accuracy score:', np.round(accuracy_score(y, y_pred),4))
    print('Recall score:', np.round(recall_score(y, y_pred, average='macro'),␣
↪4))
    print('Precision score:', np.round(precision_score(y, y_pred,␣
↪average='macro'), 4))
    return a
```

For Support Vector Classifier (SVC), we perform PCA by eliminating the redundant features. This reduces overfitting and improves the performance of the model. Also, since the number of features reduce, the computational time reduces.

[13]:
```
pca=PCA(n_components=0.85)
pca.fit(X_train)
```

[13]:
```
PCA(n_components=0.85)
```

[14]:
```
pca.n_components_
```

[14]:
```
np.int64(50)
```

[15]:
```
Xtrain_pca = pca.transform(X_train)
Xtest_pca = pca.transform(X_test)
```

2.1.1 SVC (baseline model)

[16]:
```
grid_svc = GridSearchCV(estimator=SVC(random_state=42), param_grid =
 ↪param_grid['svc'], cv=5, scoring='f1_macro', n_jobs=-1)
grid_svc.fit(Xtrain_pca, y_train)
```

[16]:
```
GridSearchCV(cv=5, estimator=SVC(random_state=42), n_jobs=-1,
             param_grid={'gamma': ['scale', 'auto'],
                         'kernel': ['rbf', 'sigmoid']},
             scoring='f1_macro')
```

[17]:
```
grid_svc.best_params_, grid_svc.best_score_
```

[17]:
```
({'gamma': 'scale', 'kernel': 'rbf'}, np.float64(0.7615560826908083))
```

[18]:
```
svc = grid_svc.best_estimator_
svc.fit(Xtrain_pca, y_train)
```

[18]:
```
SVC(random_state=42)
```

[19]:
```
print("Performance on Training set:")
model_performance(svc, Xtrain_pca, y_train)

print("\nPerformance on Test set:")
baseline_svc = model_performance(svc, Xtest_pca, y_test)
```

```
Performance on Training set:
F1 Score: 0.9408
Accuracy score: 0.9965
Recall score: 0.8975
Precision score: 0.9951

Performance on Test set:
F1 Score: 0.7796
Accuracy score: 0.9893
Recall score: 0.707
Precision score: 0.9366
```

2.1.2 Logistic Regression (baseline model)

[20]:
```
grid_logreg = GridSearchCV(LogisticRegression(random_state=42),␣
↪param_grid['logreg'], cv=5, n_jobs=-1)
grid_logreg.fit(X_train, y_train)
```

[20]:
```
GridSearchCV(cv=5, estimator=LogisticRegression(random_state=42), n_jobs=-1,
             param_grid={'C': [0.01, 0.1, 1], 'penalty': ['l1', 'l2'],
                         'solver': ['liblinear']})
```

[21]:
```
grid_logreg.best_params_, grid_logreg.best_score_
```

[21]:
```
({'C': 0.1, 'penalty': 'l1', 'solver': 'liblinear'},
 np.float64(0.9913541666666668))
```

[22]:
```
logreg = grid_logreg.best_estimator_
logreg.fit(X_train, y_train)
```

[22]:
```
LogisticRegression(C=0.1, penalty='l1', random_state=42, solver='liblinear')
```

[23]:
```
print("Performance on Training set:")
model_performance(logreg, X_train, y_train)

print("\nPerformance on Test set:")
baseline_logreg = model_performance(logreg, X_test, y_test)
```

```
Performance on Training set:
F1 Score: 0.8654
Accuracy score: 0.9922
Recall score: 0.821
Precision score: 0.9247

Performance on Test set:
F1 Score: 0.8391
Accuracy score: 0.9903
Recall score: 0.8108
Precision score: 0.8733
```

2.1.3 Decision tree (baseline model)

[24]:
```
dec_tree=DecisionTreeClassifier(random_state= 42)
grid_dectree=GridSearchCV(estimator=dec_tree, param_grid=param_grid["dt"],␣
↪cv=5, scoring='f1_macro', n_jobs=-1)
grid_dectree.fit(X_train, y_train)
```

[24]:
```
GridSearchCV(cv=5, estimator=DecisionTreeClassifier(random_state=42), n_jobs=-1,
             param_grid={'max_depth': [25, 30, 35],
```

```
                         'min_samples_leaf': [3, 5, 7]},
             scoring='f1_macro')
```

[25]:
```
grid_dectree.best_params_, grid_dectree.best_score_
```

[25]:
```
({'max_depth': 35, 'min_samples_leaf': 7}, np.float64(0.8457661216840245))
```

[26]:
```
dec_tree= grid_dectree.best_estimator_
dec_tree.fit(X_train, y_train)
```

[26]:
```
DecisionTreeClassifier(max_depth=35, min_samples_leaf=7, random_state=42)
```

[27]:
```
print("Performance on Training set:")
model_performance(dec_tree, X_train, y_train)

print("\nPerformance on Test set:")
baseline_dt = model_performance(dec_tree, X_test, y_test)
```

```
Performance on Training set:
F1 Score: 0.9265
Accuracy score: 0.9955
Recall score: 0.9018
Precision score: 0.9544

Performance on Test set:
F1 Score: 0.845
Accuracy score: 0.9904
Recall score: 0.8256
Precision score: 0.8671
```

2.2 TASK 2

2.2.1 Subtask (a):

2.2.2 OVERSAMPLING TECHNIQUE

We can use oversampling or undersampling techniques to address the issue of class imbalance. Undersampling can be useful when the dataset is extremely large. Since the class ratio is 1:59, we will not have sufficient data-points to train a good classifier model if we do undersampling. Undersampling can lead to loss of important information of majority class due to which we may not be able to generalise well. In our case, oversampling of the minority class is a better choice. Oversampling makes the dataset balanced so that a good classifier model can be learnt.

We use SMOTE technique for oversampling instead of the normal random oversampling. This is because random oversampling duplicates datapoints of minority class to balance the dataset. This might lead to overfitting. SMOTE (Synthetic Minority Over-sampling) technique creates synthetic samples for the minority class, rather than simply duplicating existing ones.

[28]:
```
smote = SMOTE(random_state=42, sampling_strategy=0.10)
Xtrain_os, ytrain_os = smote.fit_resample(X_train, y_train)
```

2.2.3 SVC

[29]:
```
Xtrain_pca=pca.transform(Xtrain_os)
Xtest_pca=pca.transform(X_test)
```

[30]:
```
grid_svc = GridSearchCV(estimator=SVC(random_state=42), param_grid =
  ↪param_grid['svc'], cv=5, scoring='f1_macro', n_jobs=-1, refit = False)
grid_svc.fit(Xtrain_pca, ytrain_os)
```

[30]:
```
GridSearchCV(cv=5, estimator=SVC(random_state=42), n_jobs=-1,
             param_grid={'gamma': ['scale', 'auto'],
                         'kernel': ['rbf', 'sigmoid']},
             refit=False, scoring='f1_macro')
```

[31]:
```
svc=SVC(**grid_svc.best_params_, random_state=42)
svc.fit(Xtrain_pca, ytrain_os)
```

[31]:
```
SVC(random_state=42)
```

[32]:
```
print("Performance on Training set:")
model_performance(svc, Xtrain_pca, ytrain_os)

print("\nPerformance on Test set:")
os_svc = model_performance(svc, Xtest_pca, y_test)
```

```
Performance on Training set:
F1 Score: 0.9554
Accuracy score: 0.9857
Recall score: 0.94
Precision score: 0.9721

Performance on Test set:
F1 Score: 0.855
Accuracy score: 0.9888
Recall score: 0.9206
Precision score: 0.8078
```

2.2.4 Logistic Regression

[33]:
```
smote = SMOTE(random_state=42, sampling_strategy=0.05)
Xtrain_os, ytrain_os = smote.fit_resample(X_train, y_train)
```

[34]:
```
grid_logreg=GridSearchCV(LogisticRegression(random_state=42), param_grid =
  ↪param_grid['logreg'], cv=5,
```

```
                                scoring='f1_macro',refit = False, n_jobs=-1)
grid_logreg.fit(Xtrain_os, ytrain_os)
```

[34]:
```
GridSearchCV(cv=5, estimator=LogisticRegression(random_state=42), n_jobs=-1,
             param_grid={'C': [0.01, 0.1, 1], 'penalty': ['l1', 'l2'],
                         'solver': ['liblinear']},
             refit=False, scoring='f1_macro')
```

[35]:
```
grid_logreg.best_params_, grid_logreg.best_score_
```

[35]:
```
({'C': 1, 'penalty': 'l1', 'solver': 'liblinear'},
 np.float64(0.9104233332689224))
```

[36]:
```
log_reg=LogisticRegression(**grid_logreg.best_params_, random_state = 42)
log_reg.fit(Xtrain_os, ytrain_os)
```

[36]:
```
LogisticRegression(C=1, penalty='l1', random_state=42, solver='liblinear')
```

[37]:
```
print("Performance on Training set:")
model_performance(log_reg, Xtrain_os, ytrain_os)

print("\nPerformance on Test set:")
os_logreg = model_performance(log_reg, X_test, y_test)
```

```
Performance on Training set:
F1 Score: 0.9197
Accuracy score: 0.9863
Recall score: 0.8938
Precision score: 0.9498

Performance on Test set:
F1 Score: 0.8493
Accuracy score: 0.9894
Recall score: 0.8742
Precision score: 0.8276
```

2.2.5 Decision Tree

[38]:
```
smote = SMOTE(random_state=42, sampling_strategy=0.03)
Xtrain_os, ytrain_os = smote.fit_resample(X_train, y_train)
```

[39]:
```
dec_tree = DecisionTreeClassifier(random_state=42)
grid_dectree = GridSearchCV(estimator=dec_tree, param_grid=param_grid["dt"],
 ↪cv=5, scoring='f1_macro',n_jobs=-1)
grid_dectree.fit(Xtrain_os, ytrain_os)
```

[39]:
```
GridSearchCV(cv=5, estimator=DecisionTreeClassifier(random_state=42), n_jobs=-1,
             param_grid={'max_depth': [25, 30, 35],
                         'min_samples_leaf': [3, 5, 7]},
             scoring='f1_macro')
```

[40]:
```
grid_dectree.best_params_, grid_dectree.best_score_
```

[40]:
```
({'max_depth': 30, 'min_samples_leaf': 5}, np.float64(0.8876397198725605))
```

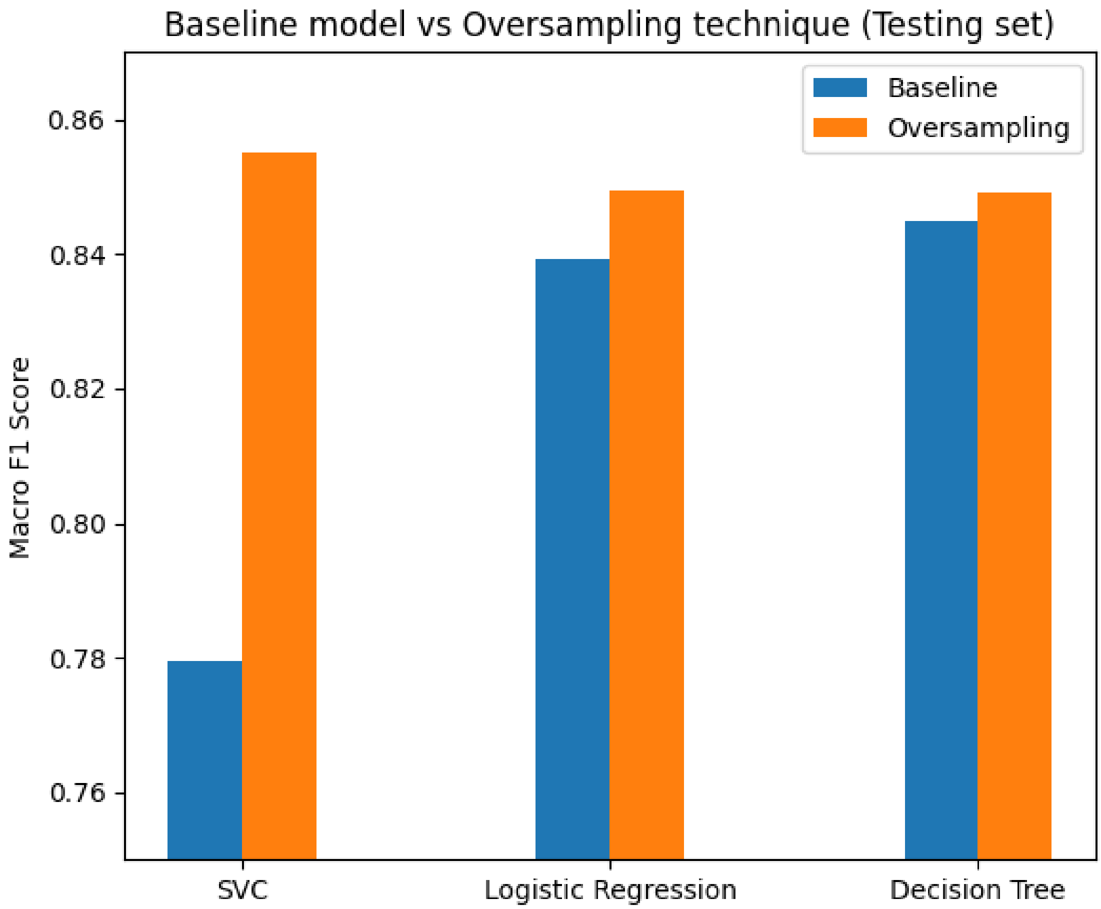
[41]:
```
dec_tree = grid_dectree.best_estimator_
```

[42]:
```
print("Performance on Training set:")
model_performance(dec_tree, Xtrain_os, ytrain_os)

print("\nPerformance on Test set:")
os_dt = model_performance(dec_tree, X_test, y_test)
```

```
Performance on Training set:
F1 Score: 0.9596
Accuracy score: 0.9955
Recall score: 0.9515
Precision score: 0.9681

Performance on Test set:
F1 Score: 0.8492
Accuracy score: 0.9898
Recall score: 0.8621
Precision score: 0.8372
```

2.2.6 Comparing Oversampling technique with the baseline models

[43]:
```
x, width = np.arange(3), 0.2

plt.figure(figsize=(6, 5))
plt.bar(x - width/2, [baseline_svc, baseline_logreg, baseline_dt], width,
  label='Baseline')
plt.bar(x + width/2, [os_svc, os_logreg, os_dt], width, label='Oversampling')

plt.ylabel('Macro F1 Score')
plt.title('Baseline model vs Oversampling technique (Testing set)')
plt.xticks(x,['SVC', 'Logistic Regression', 'Decision Tree'])
plt.ylim(0.75, 0.87)
plt.legend()
plt.tight_layout()
plt.show()
```



2.3 Subtask (b):

2.4 Class-weights technique

Class weights technique assigns different weights to each class during training. Minority class has fewer data-points, so class-weights ensure that the datapoints of minority class have more weightage while model is being learnt. Since minority class is under-represented during model training, we increase the penalty for misclassifying datapoints of minority class.

We can set class_weight as balanced or enter the weights manually.

2.4.1 SVC

[44]:
```
Xtrain_pca=pca.transform(X_train)
Xtest_pca=pca.transform(X_test)
```

[45]:
```
svc=SVC(kernel='rbf', class_weight="balanced", random_state=42)
svc.fit(Xtrain_pca, y_train)
```

[45]: SVC(class_weight='balanced', random_state=42)

[46]:
```
print("Performance on Training set:")
model_performance(svc, Xtrain_pca, y_train)

print("\nPerformance on Test set:")
cw_svc = model_performance(svc, Xtest_pca, y_test)
```

```
Performance on Training set:
F1 Score: 0.8725
Accuracy score: 0.9892
Recall score: 0.9798
Precision score: 0.8057

Performance on Test set:
F1 Score: 0.7867
Accuracy score: 0.9787
Recall score: 0.94
Precision score: 0.7155
```

2.4.2 Logistic Regression

[47]:
```
log_reg=LogisticRegression(solver='lbfgs', class_weight={"neg":2,"pos":5},␣
 ↪max_iter=500, random_state = 42)
log_reg.fit(X_train, y_train)
```

[47]:
```
LogisticRegression(class_weight={'neg': 2, 'pos': 5}, max_iter=500,
                   random_state=42)
```

[48]:
```
print("Performance on Training set:")
model_performance(log_reg, X_train, y_train)

print("\nPerformance on Test set:")
cw_logreg = model_performance(log_reg, X_test, y_test)
```

```
Performance on Training set:
F1 Score: 0.8848
Accuracy score: 0.9925
Recall score: 0.8843
Precision score: 0.8853

Performance on Test set:
F1 Score: 0.8454
Accuracy score: 0.9892
Recall score: 0.8692
Precision score: 0.8246
```

2.4.3 Decision tree

[49]:
```
dec_tree=DecisionTreeClassifier(random_state=42, class_weight={'neg': 3, 'pos':
 ↪10})
dec_tree.fit(X_train, y_train)
```

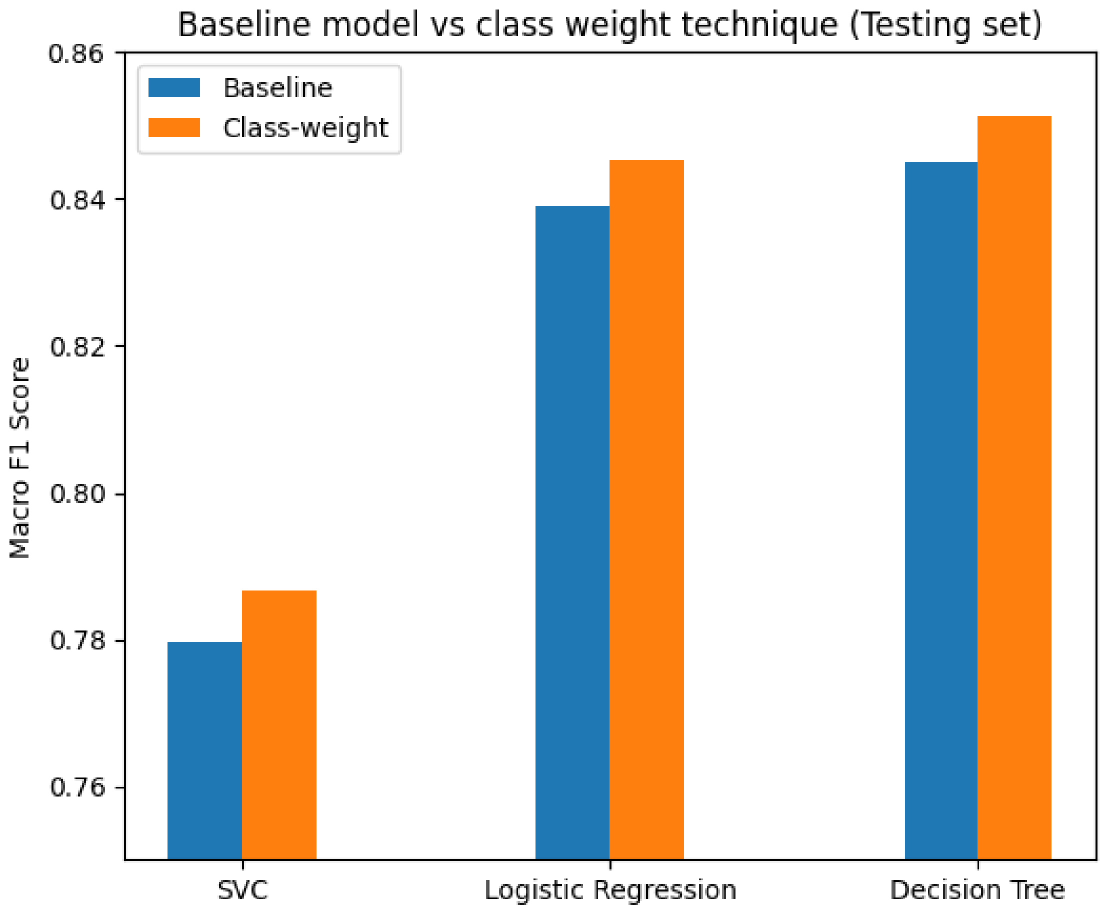
[49]:
```
DecisionTreeClassifier(class_weight={'neg': 3, 'pos': 10}, random_state=42)
```

[50]:
```
print("Performance on Training set:")
model_performance(dec_tree, X_train, y_train)

print("\nPerformance on Test set:")
cw_dt = model_performance(dec_tree, X_test, y_test)
```

```
Performance on Training set:
F1 Score: 1.0
Accuracy score: 1.0
Recall score: 1.0
Precision score: 1.0

Performance on Test set:
F1 Score: 0.8513
Accuracy score: 0.9901
Recall score: 0.8573
Precision score: 0.8455
```

2.4.4 Comparing class_weight technique with the baseline models

[51]:
```
x, width = np.arange(3), 0.2

plt.figure(figsize=(6, 5))
plt.bar(x - width/2, [baseline_svc, baseline_logreg, baseline_dt], width,␣
 ↪label='Baseline')
plt.bar(x + width/2, [cw_svc, cw_logreg, cw_dt], width, label='Class-weight')

plt.ylabel('Macro F1 Score')
plt.title('Baseline model vs class weight technique (Testing set)')
plt.xticks(x,['SVC', 'Logistic Regression', 'Decision Tree'])
plt.ylim(0.75, 0.86)
plt.legend()
plt.tight_layout()
plt.show()
```



2.4.5 Subtask (c):

2.4.6 `Sample_weights` technique

This technique allows us to assign a custom weight to each individual sample. Samples from the minority class are given higher weights to make them contribute more during training, because there are fewer samples of minority class. This helps to improve the performance of the model on imbalanced datasets by focusing more on the minority class during training.

2.4.7 SVC

[52]:
```
weights = {'neg': 2, 'pos': 5}

sample_weights = np.array([weights[class_] for class_ in y_train])
```

[53]:
```
svc = SVC(random_state=42)
svc.fit(Xtrain_pca, y_train, sample_weight=sample_weights)
```

[53]:
```
SVC(random_state=42)
```

[54]:
```python
print("Performance on Training set:")
model_performance(svc, Xtrain_pca, y_train)

print("\nPerformance on Test set:")
sw_svc = model_performance(svc, Xtest_pca, y_test)
```

```
Performance on Training set:
F1 Score: 0.9653
Accuracy score: 0.9978
Recall score: 0.9479
Precision score: 0.9842

Performance on Test set:
F1 Score: 0.813
Accuracy score: 0.9896
Recall score: 0.7661
Precision score: 0.881
```

2.4.8 Logistic Regression

[55]:
```python
weights = {'neg': 0.5, 'pos':1.2}
sample_weights = np.array([weights[class_] for class_ in y_train])
```

[56]:
```python
log_reg = LogisticRegression(solver='lbfgs', random_state=42, max_iter = 500)
log_reg.fit(X_train, y_train, sample_weight=sample_weights)
```

[56]:
```
LogisticRegression(max_iter=500, random_state=42)
```

[57]:
```python
print("Performance on Training set:")
model_performance(log_reg, X_train, y_train)

print("\nPerformance on Test set:")
sw_logreg = model_performance(log_reg, X_test, y_test)
```

```
Performance on Training set:
F1 Score: 0.8855
Accuracy score: 0.9926
Recall score: 0.8807
Precision score: 0.8903

Performance on Test set:
F1 Score: 0.8535
Accuracy score: 0.9898
Recall score: 0.8744
Precision score: 0.835
```

2.4.9 Decision tree

[58]:
```
weights = {'neg': 2, 'pos': 5}
sample_weights = np.array([weights[class_] for class_ in y_train])
```

[59]:
```
dec_tree = DecisionTreeClassifier(random_state=42)
dec_tree.fit(X_train, y_train, sample_weight=sample_weights)
```

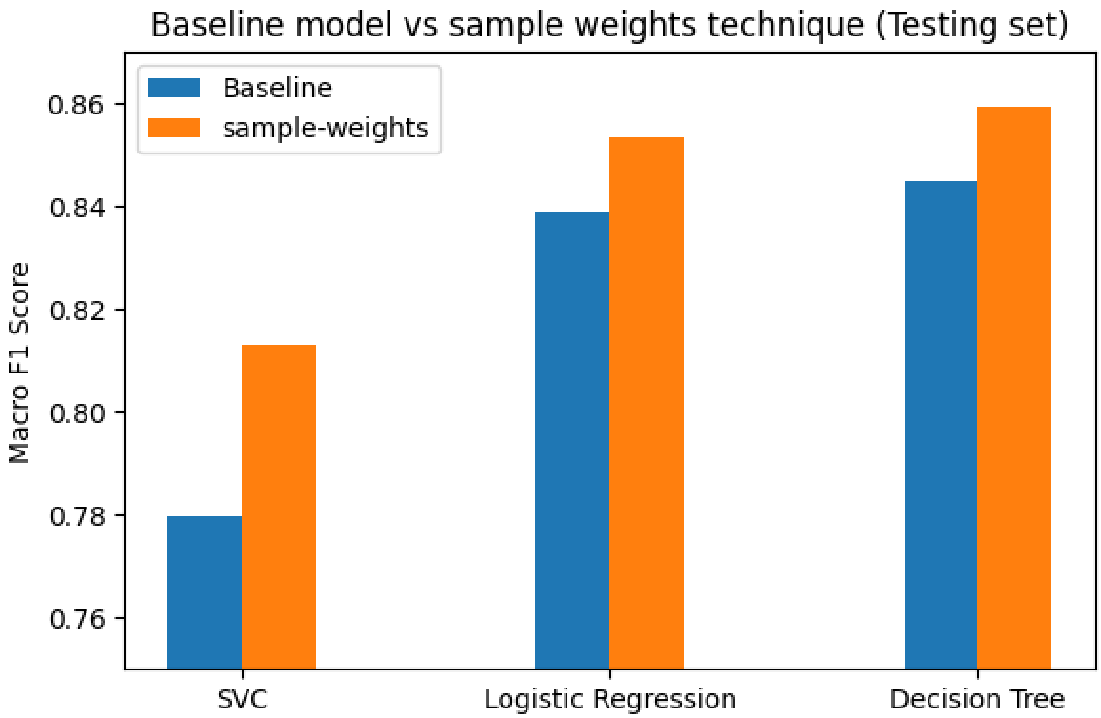
[59]:
```
DecisionTreeClassifier(random_state=42)
```

[60]:
```
print("Performance on Training set:")
model_performance(dec_tree, X_train, y_train)

print("\nPerformance on Test set:")
sw_dt = model_performance(dec_tree, X_test, y_test)
```

```
Performance on Training set:
F1 Score: 1.0
Accuracy score: 1.0
Recall score: 1.0
Precision score: 1.0

Performance on Test set:
F1 Score: 0.8595
Accuracy score: 0.9906
Recall score: 0.8674
Precision score: 0.8519
```

2.4.10 Comparing sample_weights technique with the baseline models

[61]:
```
x, width = np.arange(3), 0.2

plt.figure(figsize=(6, 4))
plt.bar(x - width/2, [baseline_svc, baseline_logreg, baseline_dt], width,␣
 ↪label='Baseline')
plt.bar(x + width/2, [sw_svc, sw_logreg, sw_dt], width, label='sample-weights')

plt.ylabel('Macro F1 Score')
plt.title('Baseline model vs sample weights technique (Testing set)')
plt.xticks(x,['SVC', 'Logistic Regression', 'Decision Tree'])
plt.ylim(0.75, 0.87)
plt.legend()
plt.tight_layout()
plt.show()
```



2.4.11 Subtask (d): Creative ways to address class-imbalance

2.4.12 Combining oversampling and `class_weight` technique

We try to combine oversampling and `class_weight` technique and try to improve the macro F1 score. We oversample the minority class using `SMOTE` and then apply custom `class_weight`, assigning higher weights to minority class, to improve model performance.

2.4.13 SVC

[62]:
```
smote = SMOTE(random_state=42, sampling_strategy=0.10)
Xtrain_os, ytrain_os = smote.fit_resample(X_train, y_train)
```

[63]:
```
Xtrain_pca=pca.transform(Xtrain_os)
Xtest_pca=pca.transform(X_test)
```

[64]:
```
svc = SVC(**grid_svc.best_params_,class_weight={'neg':2.5, 'pos':6},␣
 ↪random_state=42)
svc.fit(Xtrain_pca, ytrain_os)
```

[64]:
```
SVC(class_weight={'neg': 2.5, 'pos': 6}, random_state=42)
```

[65]:
```
print("Performance on Training set:")
model_performance(svc, Xtrain_pca, ytrain_os)
```

```
print("\nPerformance on Test set:")
comb_svc = model_performance(svc, Xtest_pca, y_test)
```

```
Performance on Training set:
F1 Score: 0.9687
Accuracy score: 0.9898
Recall score: 0.9646
Precision score: 0.9729

Performance on Test set:
F1 Score: 0.8256
Accuracy score: 0.9868
Recall score: 0.8753
Precision score: 0.788
```

2.4.14 Logistic Regression

[66]:
```
smote = SMOTE(random_state=42, sampling_strategy=0.04)
Xtrain_os, ytrain_os = smote.fit_resample(X_train, y_train)
```

[67]:
```
log_reg = LogisticRegression(solver = "liblinear", C = 0.02, penalty =␣
↪'l2',class_weight={'neg':0.7, 'pos':1.1}, random_state=42, max_iter = 500)
log_reg.fit(Xtrain_os, ytrain_os)
```

[67]:
```
LogisticRegression(C=0.02, class_weight={'neg': 0.7, 'pos': 1.1}, max_iter=500,
                   random_state=42, solver='liblinear')
```

[68]:
```
print("Performance on Training set:")
model_performance(log_reg, Xtrain_os, ytrain_os)

print("\nPerformance on Test set:")
comb_logreg = model_performance(log_reg, X_test, y_test)
```

```
Performance on Training set:
F1 Score: 0.9068
Accuracy score: 0.9868
Recall score: 0.8889
Precision score: 0.9266

Performance on Test set:
F1 Score: 0.8515
Accuracy score: 0.9892
Recall score: 0.8913
Precision score: 0.8194
```

2.4.15 Decision Tree

[69]:
```
smote = SMOTE(random_state=42, sampling_strategy=0.02)
Xtrain_os, ytrain_os = smote.fit_resample(X_train, y_train)
```

[70]:
```
dec_tree = DecisionTreeClassifier(**grid_dectree.
 ↪best_params_,class_weight={'neg':0.4, 'pos':0.6}, random_state=42)
dec_tree.fit(Xtrain_os, ytrain_os)
```

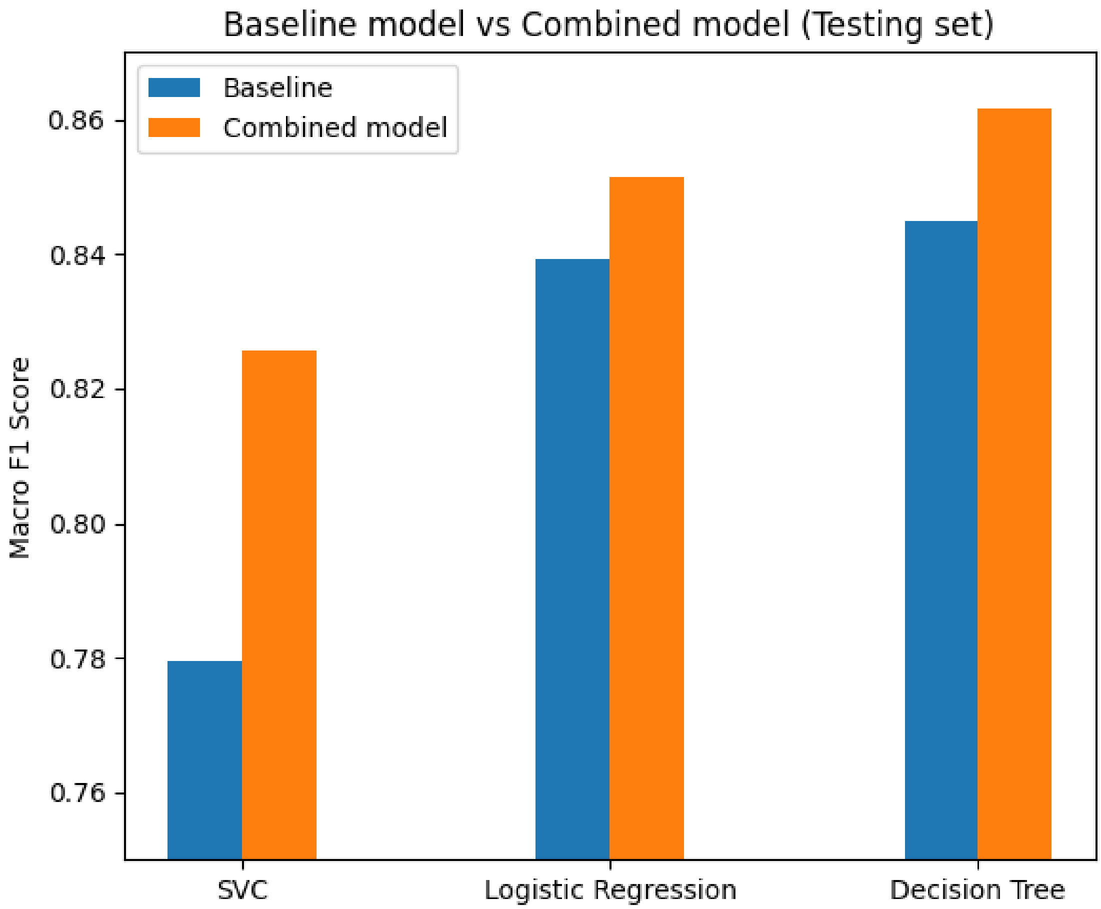
[70]:
```
DecisionTreeClassifier(class_weight={'neg': 0.4, 'pos': 0.6}, max_depth=30,
                       min_samples_leaf=5, random_state=42)
```

[71]:
```
print("Performance on Training set:")
model_performance(dec_tree, Xtrain_os, ytrain_os)

print("\nPerformance on Test set:")
comb_dt = model_performance(dec_tree, X_test, y_test)
```

```
Performance on Training set:
F1 Score: 0.9501
Accuracy score: 0.9962
Recall score: 0.9456
Precision score: 0.9546

Performance on Test set:
F1 Score: 0.8617
Accuracy score: 0.9907
Recall score: 0.8724
Precision score: 0.8517
```

2.4.16 Comparing the combination of oversampling and class_weight technique with baseline models

[72]:
```
x, width = np.arange(3), 0.2

plt.figure(figsize=(6, 5))
plt.bar(x - width/2, [baseline_svc, baseline_logreg, baseline_dt], width,␣
 ↪label='Baseline')
plt.bar(x + width/2, [comb_svc, comb_logreg, comb_dt], width, label='Combined␣
 ↪model')

plt.ylabel('Macro F1 Score')
plt.title('Baseline model vs Combined model (Testing set)')
plt.xticks(x,['SVC', 'Logistic Regression', 'Decision Tree'])
plt.ylim(0.75, 0.87)
plt.legend()
plt.tight_layout()
plt.show()
```



2.5 Summary

[73]:
```
data = {"Baseline": [baseline_svc, baseline_logreg,baseline_dt], 'Oversampling':
 ↪ [os_svc, os_logreg, os_dt], 'Class-weights': [cw_svc, cw_logreg, cw_dt],
    'Sample-weights': [sw_svc, sw_logreg, sw_dt], "Combined model": [comb_svc,␣
 ↪comb_logreg, comb_dt] }
f1scores = pd.DataFrame(data, index=["SVC", "Logistic Regression", "Decision␣
 ↪Tree"])
f1scores
```

[73]:
```
                     Baseline  Oversampling  Class-weights  Sample-weights  \
SVC                    0.7796        0.8550         0.7867          0.8130
Logistic Regression    0.8391        0.8493         0.8454          0.8535
Decision Tree          0.8450        0.8492         0.8513          0.8595

                     Combined model
SVC                          0.8256
Logistic Regression          0.8515
```

```
Decision Tree                  0.8617
```

From the above table, it is clear that the macro average F1 score of the hacked classifiers is better than the baseline classifier models