Fraud Detector

February 4, 2025

0.1 Fraud Detector - Dealing with Imbalanced Datasets

0.1.1 Goals:

- Understand the provided data
- Determine the Classifiers we are going to use and decide which one has a higher precision, recall and f1 scores.
- Apply Random-Undersampling and SMOTE to the dataset in order to deal with the heavy class imbalance.

Understanding our data

- The description of the dataset says that all features went through a PCA (Dimensionality Reduction Technique) transformation, except for "Time" and "Amount", meaning all "V" features are already scaled (Since that is need in order to implement PCA).

[1]:
```python
import pandas as pd

df = pd.read_csv("creditcard.csv")
df.head()
```

[1]:
```
   Time        V1        V2        V3        V4        V5        V6        V7  \
0   0.0 -1.359807 -0.072781  2.536347  1.378155 -0.338321  0.462388  0.239599
1   0.0  1.191857  0.266151  0.166480  0.448154  0.060018 -0.082361 -0.078803
2   1.0 -1.358354 -1.340163  1.773209  0.379780 -0.503198  1.800499  0.791461
3   1.0 -0.966272 -0.185226  1.792993 -0.863291 -0.010309  1.247203  0.237609
4   2.0 -1.158233  0.877737  1.548718  0.403034 -0.407193  0.095921  0.592941

         V8        V9  …       V21       V22       V23       V24       V25  \
0  0.098698  0.363787  … -0.018307  0.277838 -0.110474  0.066928  0.128539
1  0.085102 -0.255425  … -0.225775 -0.638672  0.101288 -0.339846  0.167170
2  0.247676 -1.514654  …  0.247998  0.771679  0.909412 -0.689281 -0.327642
3  0.377436 -1.387024  … -0.108300  0.005274 -0.190321 -1.175575  0.647376
4 -0.270533  0.817739  … -0.009431  0.798278 -0.137458  0.141267 -0.206010

        V26       V27       V28  Amount  Class
0 -0.189115  0.133558 -0.021053  149.62      0
1  0.125895 -0.008983  0.014724    2.69      0
2 -0.139097 -0.055353 -0.059752  378.66      0
```

```
3 -0.221929  0.062723  0.061458  123.50      0
4  0.502292  0.219422  0.215153   69.99      0

[5 rows x 31 columns]
```

[2]:
```python
df["Class"].value_counts()
```

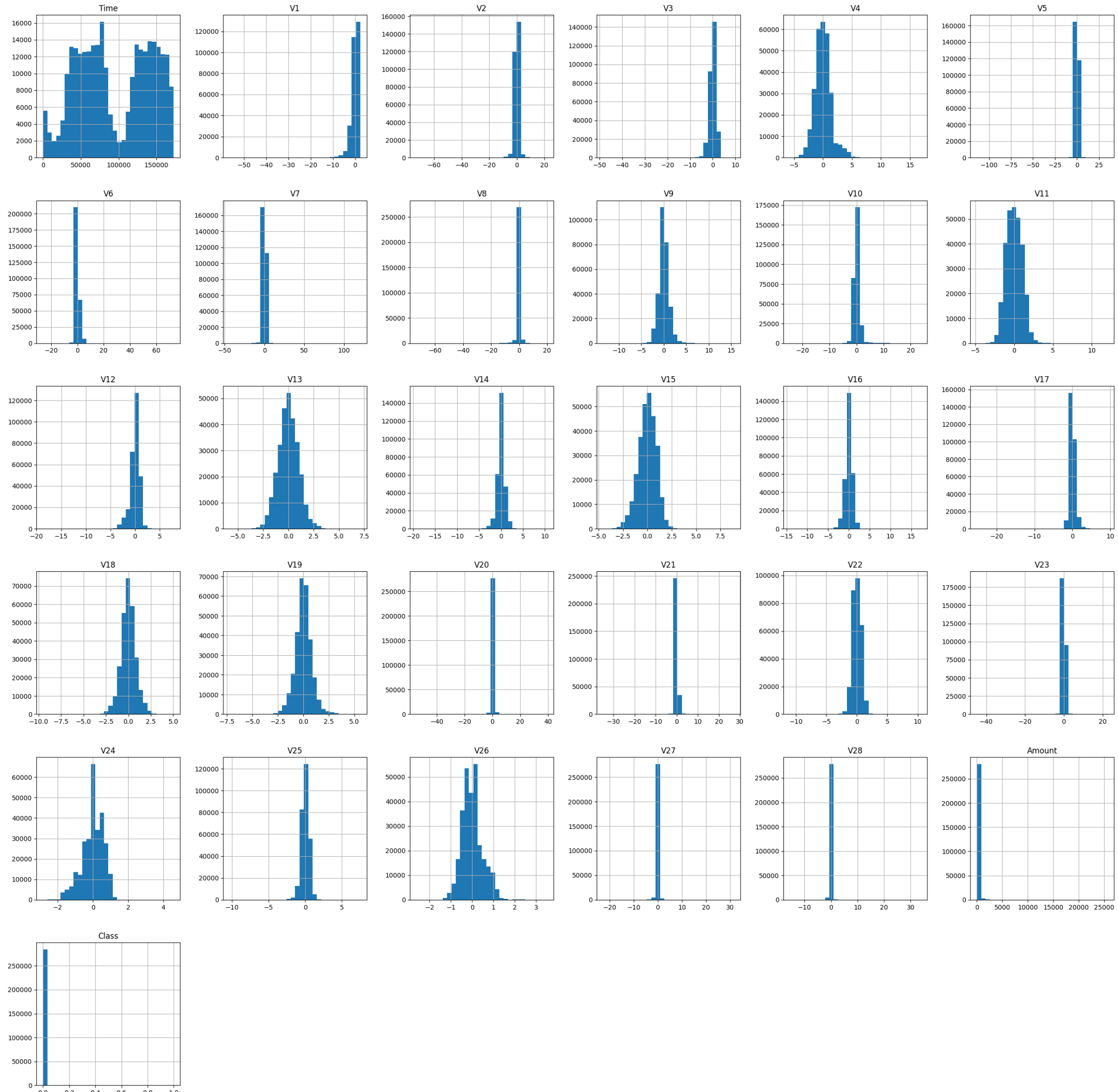
[2]:
```
Class
0    284315
1       492
Name: count, dtype: int64
```

[3]:
```python
print(f" No Frauds: {df["Class"].value_counts(normalize=True)[0] * 100:.2f} %␣
↪of the dataset")
print(f" Frauds: {df["Class"].value_counts(normalize=True)[1] * 100:.2f} % of␣
↪the dataset")
```

```
 No Frauds: 99.83 % of the dataset
 Frauds: 0.17 % of the dataset
```

Most of the transactions (99.83%) are non-fraud, meaning only 0.17% of the transactions are classified as fraud, making the dataset heavily imbalanced.

[4]:
```python
import matplotlib.pyplot as plt
df.hist(bins=30, figsize=(30,30))
plt.show()
```



[5]:
```
df.describe()
```

[5]:
```
                Time            V1            V2            V3            V4  \
count  284807.000000  2.848070e+05  2.848070e+05  2.848070e+05  2.848070e+05
mean    94813.859575  1.168375e-15  3.416908e-16 -1.379537e-15  2.074095e-15
std     47488.145955  1.958696e+00  1.651309e+00  1.516255e+00  1.415869e+00
min         0.000000 -5.640751e+01 -7.271573e+01 -4.832559e+01 -5.683171e+00
25%     54201.500000 -9.203734e-01 -5.985499e-01 -8.903648e-01 -8.486401e-01
50%     84692.000000  1.810880e-02  6.548556e-02  1.798463e-01 -1.984653e-02
75%    139320.500000  1.315642e+00  8.037239e-01  1.027196e+00  7.433413e-01
max    172792.000000  2.454930e+00  2.205773e+01  9.382558e+00  1.687534e+01

                 V5            V6            V7            V8            V9  \
```

```
count  2.848070e+05  2.848070e+05  2.848070e+05  2.848070e+05  2.848070e+05
mean   9.604066e-16  1.487313e-15 -5.556467e-16  1.213481e-16 -2.406331e-15
std    1.380247e+00  1.332271e+00  1.237094e+00  1.194353e+00  1.098632e+00
min   -1.137433e+02 -2.616051e+01 -4.355724e+01 -7.321672e+01 -1.343407e+01
25%   -6.915971e-01 -7.682956e-01 -5.540759e-01 -2.086297e-01 -6.430976e-01
50%   -5.433583e-02 -2.741871e-01  4.010308e-02  2.235804e-02 -5.142873e-02
75%    6.119264e-01  3.985649e-01  5.704361e-01  3.273459e-01  5.971390e-01
max    3.480167e+01  7.330163e+01  1.205895e+02  2.000721e+01  1.559499e+01

       …           V21           V22           V23           V24  \
count  …  2.848070e+05  2.848070e+05  2.848070e+05  2.848070e+05
mean   …  1.654067e-16 -3.568593e-16  2.578648e-16  4.473266e-15
std    …  7.345240e-01  7.257016e-01  6.244603e-01  6.056471e-01
min    … -3.483038e+01 -1.093314e+01 -4.480774e+01 -2.836627e+00
25%    … -2.283949e-01 -5.423504e-01 -1.618463e-01 -3.545861e-01
50%    … -2.945017e-02  6.781943e-03 -1.119293e-02  4.097606e-02
75%    …  1.863772e-01  5.285536e-01  1.476421e-01  4.395266e-01
max    …  2.720284e+01  1.050309e+01  2.252841e+01  4.584549e+00

                V25           V26           V27           V28         Amount  \
count  2.848070e+05  2.848070e+05  2.848070e+05  2.848070e+05  284807.000000
mean   5.340915e-16  1.683437e-15 -3.660091e-16 -1.227390e-16      88.349619
std    5.212781e-01  4.822270e-01  4.036325e-01  3.300833e-01     250.120109
min   -1.029540e+01 -2.604551e+00 -2.256568e+01 -1.543008e+01       0.000000
25%   -3.171451e-01 -3.269839e-01 -7.083953e-02 -5.295979e-02       5.600000
50%    1.659350e-02 -5.213911e-02  1.342146e-03  1.124383e-02      22.000000
75%    3.507156e-01  2.409522e-01  9.104512e-02  7.827995e-02      77.165000
max    7.519589e+00  3.517346e+00  3.161220e+01  3.384781e+01   25691.160000

               Class
count  284807.000000
mean        0.001727
std         0.041527
min         0.000000
25%         0.000000
50%         0.000000
75%         0.000000
max         1.000000

[8 rows x 31 columns]
```

- The amount column currently has values ranging from 0, to ~25000, which we should definetly scale. In this case a robust scaler is likely to work better because of it's robustness to outliers.
- The time column has no real outliers, therefore we will normalize it using the MinMaxScaler

Note: It is crucial to split our data **before** scaling to avoid data leakage. Data leakage occurs when information from outside the training dataset is used to create the model, leading to overly optimistic performance estimates.

[6]:
```python
from sklearn.model_selection import train_test_split

X = df.drop(columns="Class")
y = df["Class"]

X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3,␣
↪stratify=y, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5,␣
↪stratify=y_temp, random_state=42)

y_train.value_counts(), y_val.value_counts(), y_test.value_counts()
```

[6]:
```
(Class
 0    199020
 1       344
 Name: count, dtype: int64,
 Class
 0    42647
 1       74
 Name: count, dtype: int64,
 Class
 0    42648
 1       74
 Name: count, dtype: int64)
```

[7]:
```python
from sklearn.preprocessing import RobustScaler, MinMaxScaler

robust_scaler = RobustScaler()
minmax_scaler = MinMaxScaler()

# Fit train
X_train["Amount"] = robust_scaler.fit_transform(X_train["Amount"].to_numpy().
↪reshape(-1, 1))
X_train["Time"] = minmax_scaler.fit_transform(X_train["Time"].to_numpy().
↪reshape(-1, 1))

# Transform test and validation
X_test["Amount"] = robust_scaler.transform(X_test["Amount"].to_numpy().
↪reshape(-1, 1))
X_test["Time"] = minmax_scaler.transform(X_test["Time"].to_numpy().reshape(-1,␣
↪1))

X_val["Amount"] = robust_scaler.transform(X_val["Amount"].to_numpy().
↪reshape(-1, 1))
X_val["Time"] = minmax_scaler.transform(X_val["Time"].to_numpy().reshape(-1, 1))
```

[8]:
```python
X_train.describe()
```

[8]:

```
                Time             V1             V2             V3  \
count  199364.000000  199364.000000  199364.000000  199364.000000
mean        0.549205      -0.001137      -0.002024      -0.001333
std         0.274839       1.965794       1.658079       1.519820
min         0.000000     -56.407510     -72.715728     -48.325589
25%         0.313983      -0.919472      -0.600466      -0.890875
50%         0.491027       0.017529       0.064591       0.180371
75%         0.806548       1.315404       0.804932       1.026038
max         1.000000       2.451888      22.057729       9.382558

                  V4             V5             V6             V7  \
count  199364.000000  199364.000000  199364.000000  199364.000000
mean        0.000313       0.000202       0.000302      -0.000307
std         1.416731       1.387295       1.336558       1.248395
min        -5.683171    -113.743307     -26.160506     -43.557242
25%        -0.846902      -0.691963      -0.768846      -0.553719
50%        -0.020802      -0.054897      -0.273921       0.040482
75%         0.743833       0.611243       0.398847       0.571020
max        16.875344      34.801666      73.301626     120.589494

                  V8             V9  …            V20            V21  \
count  199364.000000  199364.000000  …  199364.000000  199364.000000
mean       -0.001291       0.001995  …      -0.001012       0.000459
std         1.198699       1.098649  …       0.776632       0.740233
min       -73.216718     -13.434066  …     -54.497720     -34.830382
25%        -0.208460      -0.640513  …      -0.211792      -0.228438
50%         0.022954      -0.050199  …      -0.062695      -0.029121
75%         0.326971       0.600147  …       0.133000       0.186562
max        20.007208      15.594995  …      39.420904      27.202839

                 V22            V23            V24            V25  \
count  199364.000000  199364.000000  199364.000000  199364.000000
mean        0.000360       0.000731      -0.000054      -0.000550
std         0.726146       0.625116       0.605084       0.521473
min       -10.933144     -44.807735      -2.836627     -10.295397
25%        -0.542054      -0.162021      -0.354888      -0.317041
50%         0.006539      -0.010594       0.041130       0.016344
75%         0.528738       0.147946       0.439173       0.350126
max        10.503090      22.083545       4.584549       6.070850

                 V26            V27            V28         Amount
count  199364.000000  199364.000000  199364.000000  199364.000000
mean        0.000072      -0.000405       0.000522       0.924193
std         0.482197       0.407727       0.329701       3.523125
min        -2.604551     -22.565679     -15.430084      -0.306279
25%        -0.326836      -0.070712      -0.052910      -0.227342
50%        -0.052065       0.001367       0.011266       0.000000
```

```
75%         0.240930       0.091088       0.078266       0.772658
max         3.517346      31.612198      33.847808     357.359877

[8 rows x 30 columns]
```

0.1.2 Random Undersampling

- Undersampling removes majority-class samples (non-fraud) to balance the dataset. The main issue with this is that our model might not be as accurate since we're losing a considerable amount of data(from 284,315 non-fraud to 492 non-fraud)
- We do **NOT** undersample test/validation sets, as they should reflect real-world class distribution.

[9]:

```python
from imblearn.under_sampling import RandomUnderSampler

undersampler = RandomUnderSampler(random_state=42)
X_train_under, y_train_under = undersampler.fit_resample(X_train, y_train)

print(f"Original Train Class Distributions: {y_train.value_counts()}")
print(f"Undersampled Train Class Distributions: {y_train_under.value_counts()}")
```

```
Original Train Class Distributions: Class
0    199020
1       344
Name: count, dtype: int64
Undersampled Train Class Distributions: Class
0    344
1    344
Name: count, dtype: int64
```

[10]:

```python
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier,␣
 ↪HistGradientBoostingClassifier

classifiers = {
    "Logistic Regression": LogisticRegression(random_state=42),
    "Linear SVC": SVC(kernel="linear", probability=True),
    "Random Forest": RandomForestClassifier(class_weight="balanced", n_jobs=2,␣
 ↪random_state=42),
    "Hist Gradient Boosting": HistGradientBoostingClassifier(max_iter=1000,␣
 ↪early_stopping=True, random_state=42)
}
```

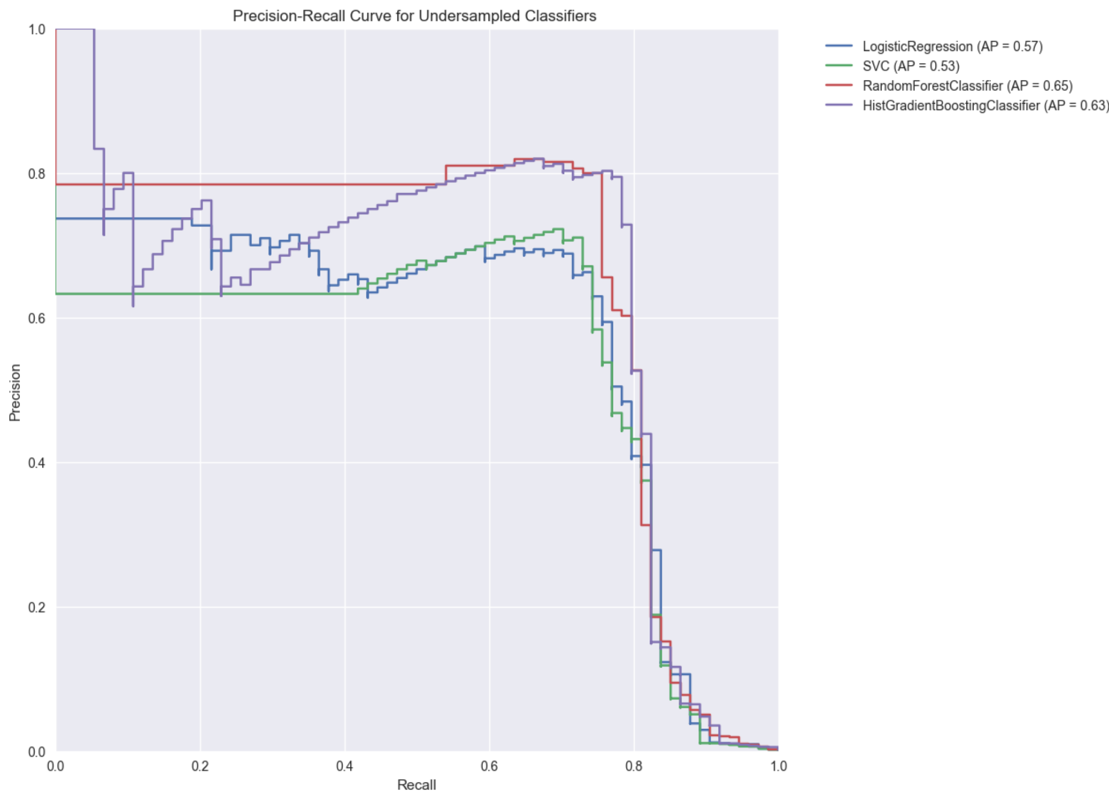
[11]:

```python
from sklearn.metrics import PrecisionRecallDisplay, classification_report,␣
 ↪average_precision_score

plt.style.use("seaborn-v0_8")
```

```python
plt.figure(figsize=(12,10))

for key, classifier in classifiers.items():

    classifier.fit(X_train_under, y_train_under)

    pred = classifier.predict(X_val)
    proba = classifier.predict_proba(X_val)[:, 1]
    print(f"{classifier.__class__.__name__} scores:\n"
          f"{classification_report(y_val, pred)}\n"
         f"{classifier.__class__.__name__} PR-AUC:
 ↪{average_precision_score(y_val, proba):.4f}\n")

    PrecisionRecallDisplay.from_estimator(
        classifier, X_val, y_val, ax=plt.gca(), marker="+", name=classifier.
 ↪__class__.__name__
    )

plt.title("Precision-Recall Curve for Undersampled Classifiers ")
plt.xlabel("Recall")
plt.ylabel("Precision")
plt.xlim(0, 1)
plt.ylim(0, 1)
plt.legend(bbox_to_anchor=(1.05, 1), loc="upper left")
plt.grid(True)
plt.show()
```

```
LogisticRegression scores:
              precision    recall  f1-score   support

           0       1.00      0.98      0.99     42647
           1       0.06      0.88      0.12        74

    accuracy                           0.98     42721
   macro avg       0.53      0.93      0.55     42721
weighted avg       1.00      0.98      0.99     42721

LogisticRegression PR-AUC:0.5687

SVC scores:
              precision    recall  f1-score   support

           0       1.00      0.97      0.99     42647
           1       0.05      0.88      0.10        74

    accuracy                           0.97     42721
```

```
   macro avg       0.53      0.93      0.54     42721
weighted avg       1.00      0.97      0.99     42721

SVC PR-AUC:0.5340

RandomForestClassifier scores:
              precision    recall  f1-score   support

           0       1.00      0.98      0.99     42647
           1       0.07      0.88      0.13        74

    accuracy                           0.98     42721
   macro avg       0.54      0.93      0.56     42721
weighted avg       1.00      0.98      0.99     42721

RandomForestClassifier PR-AUC:0.6455

HistGradientBoostingClassifier scores:
              precision    recall  f1-score   support

           0       1.00      0.97      0.99     42647
           1       0.05      0.89      0.10        74

    accuracy                           0.97     42721
   macro avg       0.53      0.93      0.54     42721
weighted avg       1.00      0.97      0.98     42721

HistGradientBoostingClassifier PR-AUC:0.6341
```



- As expected our tree ensembles have a better average precision than our simpler classifiers(Logisitic Regression and Linear SVC).
- Undersampling helped but did not achieve a great result handling the class imbalance.

0.1.3 SMOTE(Synthetic Minority Over-sampling Technique)

SMOTE creates new synthetic points in order to have an equal balance of the classes. This is another alternative for solving the "class imbalance problems".

Understanding SMOTE:

- Solving the Class Imbalance: SMOTE creates synthetic points from the minority class in order to reach an equal balance between the minority and majority class.
- Location of the synthetic points: SMOTE picks the distance between the closest neighbors of the minority class, in between these distances it creates synthetic points.
- Final Effect: More information is retained since we didn't have to delete any rows unlike in random undersampling.
- Accuracy - Time Tradeoff: Although it is likely that SMOTE will be more accurate than random under-sampling, it will take more time to train since no rows are eliminated as previously stated.

[12]:
```
from imblearn.over_sampling import SMOTE
from sklearn.ensemble import RandomForestClassifier
```

```
from sklearn.metrics import classification_report, roc_auc_score,␣
↪average_precision_score

smote = SMOTE(random_state=42)
X_train_res, y_train_res = smote.fit_resample(X_train, y_train)
```

[13]:
```
rf_model = RandomForestClassifier(random_state=42, n_jobs=2)
rf_model.fit(X_train_res, y_train_res)

y_pred = rf_model.predict(X_val)
y_pred_proba = rf_model.predict_proba(X_val)[:, 1]

print(classification_report(y_val, y_pred, target_names=['Not Fraud', 'Fraud']))
print(f"PR-AUC: {average_precision_score(y_val, y_pred_proba):.4f}")

#y_pred_test= rf_model.predict(X_test)
#y_pred_proba_test = rf_model.predict_proba(X_test)[:, 1]
#print(f"PR-AUC Test: {average_precision_score(y_test, y_pred_proba_test):.4f}")
```

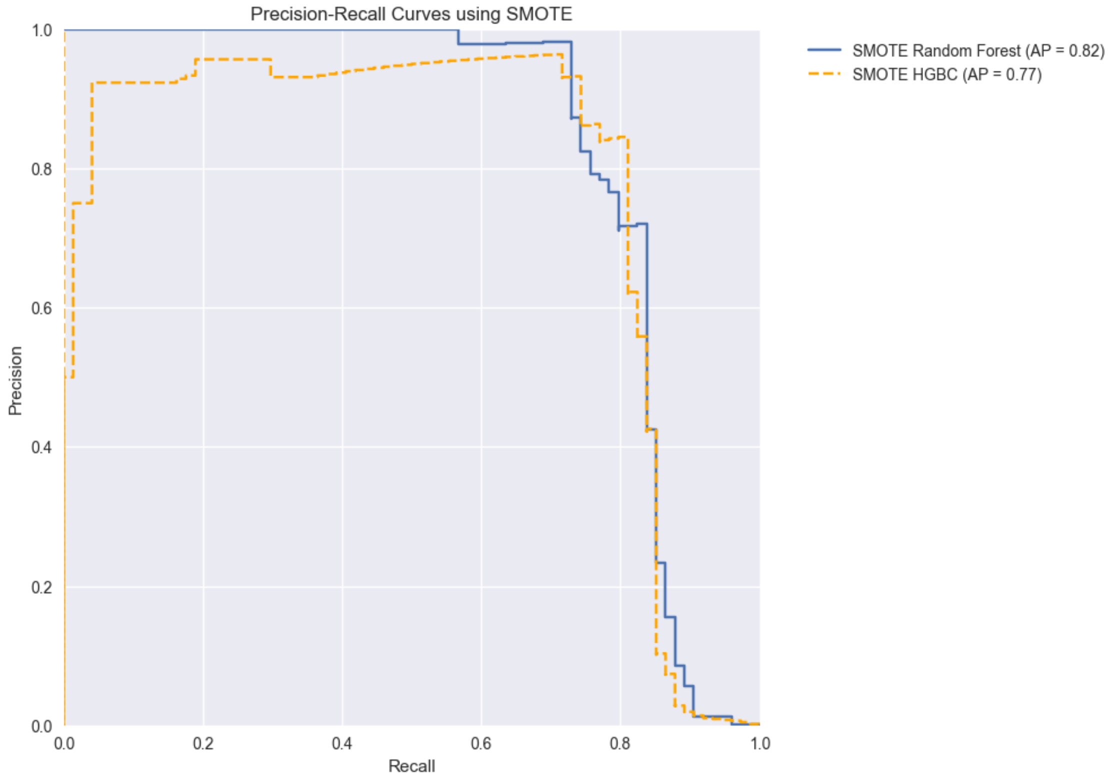
```
              precision    recall  f1-score   support

   Not Fraud       1.00      1.00      1.00     42647
       Fraud       0.86      0.74      0.80        74

    accuracy                           1.00     42721
   macro avg       0.93      0.87      0.90     42721
weighted avg       1.00      1.00      1.00     42721

PR-AUC: 0.8238
```

[14]:
```
boost_smote =  HistGradientBoostingClassifier(max_iter=1000,␣
↪early_stopping=True, random_state=42)
boost_smote.fit(X_train_res, y_train_res)

boost_smote_pred = boost_smote.predict(X_val)
boost_smote_pred_proba = boost_smote.predict_proba(X_val)[:, 1]

#boost_smote_pred_test = boost_smote.predict(X_test)
#boost_smote_pred_proba_test = boost_smote.predict_proba(X_test)[:, 1]

print(classification_report(y_val, boost_smote_pred, target_names=['Not Fraud',␣
↪'Fraud']))
print(f"PR-AUC: {average_precision_score(y_val, boost_smote_pred_proba):.4f}")
#print(f"PR-AUC Test: {average_precision_score(y_test,␣
↪boost_smote_pred_proba_test):.4f}")
```

```
              precision    recall  f1-score   support
```

```
   Not Fraud       1.00      1.00      1.00     42647
       Fraud       0.50      0.84      0.63        74

    accuracy                           1.00     42721
   macro avg       0.75      0.92      0.81     42721
weighted avg       1.00      1.00      1.00     42721

PR-AUC: 0.7732
```

[15]:
```
fig, ax = plt.subplots(figsize=(10, 8))

disp1 = PrecisionRecallDisplay.from_estimator(
    rf_model, X_val, y_val, pos_label=1, marker="+", ax=ax, name="SMOTE Random␣
↪Forest"
)

disp2 = PrecisionRecallDisplay.from_estimator(
    boost_smote, X_val, y_val, pos_label=1, marker="+", color="orange",␣
↪linestyle="--", ax=ax, name="SMOTE HGBC"
)

ax.set_xlabel("Recall")
ax.set_ylabel("Precision")
ax.set_xlim(0, 1)
ax.set_ylim(0, 1)
ax.legend(bbox_to_anchor=(1.05, 1), loc="upper left")
ax.set_title("Precision-Recall Curves using SMOTE")

plt.show()
```



Without any hyperparameter tuning, our Random Forest Classifier shows signs of overfitting because: - It maintains a long plateau at Precision = 1.0, suggesting it makes highly confident predictions for certain instances. - It has a sharp drop-off, indicating that when it misclassifies, it does so abruptly, which is a common sign of overfitting.

The HistGradientBoosting model has a slightly lower Average Precision (AP), but: - Its PR curve is smoother and doesn't have the same extreme plateau and drop-off. - This suggests it might generalize better by not over-optimizing for the validation set and potentially performing better on unseen data.

The ROC-AUC curve can be misleading on highly imbalanced datasets because it remains relatively high even when the model performs poorly on the minority class. That's why I'm evaluating only the PR Curve.

0.1.4 Hyperparameter tuning

It is a good idea to concatenate our validation and test sets into a single test set for hyperparameter tuning since: - Nested CV already handles validation internally. - Larger test set can lead to a more reliable evaluation.

[95]:
```
import numpy as np

X_test_full = np.concatenate([X_val, X_test])
y_test_full = np.concatenate([y_val, y_test])
```

```
X_test_full = pd.DataFrame(X_test_full, columns=X_train.columns)
```

[48]:

```
from imblearn.pipeline import Pipeline
from imblearn.over_sampling import SMOTE
from sklearn.ensemble import HistGradientBoostingClassifier
from sklearn.model_selection import RandomizedSearchCV, cross_validate,␣
↪StratifiedKFold
from sklearn.metrics import make_scorer, f1_score, average_precision_score
from scipy.stats import loguniform

#Hyperparemeter tuning for HGBC

inner_cv = StratifiedKFold(n_splits=3, shuffle=True, random_state=42)
outer_cv = StratifiedKFold(n_splits=2, shuffle=True, random_state=42)

scoring = {
    "f1": make_scorer(f1_score, pos_label=1),
    "average_precision": make_scorer(average_precision_score, pos_label=1)
}

param_distributions = {
    "classifier__max_leaf_nodes": [2, 5, 10, 50, 100],
    "classifier__learning_rate": loguniform(0.01, 1)
}

pipeline = Pipeline([
    ("smote", SMOTE(random_state=42)),
    ("classifier", HistGradientBoostingClassifier(max_iter=300,␣
↪early_stopping=True))
])

boost_random_search = RandomizedSearchCV(pipeline,␣
↪param_distributions=param_distributions, cv=inner_cv, n_iter=5, n_jobs=1)

cv_results = cross_validate(boost_random_search, X_train, y_train, cv=outer_cv,␣
↪scoring=scoring, return_estimator=True, n_jobs=1)

test_f1 = cv_results["test_f1"]
test_ap = cv_results["test_average_precision"]

print(f"Tuned HGBC mean f1: {test_f1.mean():.3f}")
print(f"Tuned HGBC mean AP: {test_ap.mean():.3f}")
```

```
Tuned HGBC mean f1: 0.827
Tuned HGBC mean AP: 0.685
```

[109]:
```
from imblearn.over_sampling import SMOTE
from sklearn.ensemble import HistGradientBoostingClassifier
from sklearn.metrics import classification_report, average_precision_score

#Extract best params

best_estimators = cv_results["estimator"]
best_params_list = [
   {k: tuple(v) if isinstance(v, list) else v for k, v in est.best_estimator_.
 ↪get_params().items()}
   for est in best_estimators
]
best_params = Counter(tuple(sorted(p.items())) for p in best_params_list).
 ↪most_common(1)[0][0]
best_params = dict(best_params)

classifier_params = {k.replace('classifier__', ''): v for k, v in best_params.
 ↪items()
                     if k.startswith('classifier__')}
```

[110]:
```
final_smote = SMOTE(random_state=42)
x_train_smote, y_train_smote = final_smote.fit_resample(X_train, y_train)

final_boost_model = HistGradientBoostingClassifier(**classifier_params)
final_boost_model.fit(x_train_smote, y_train_smote)
```

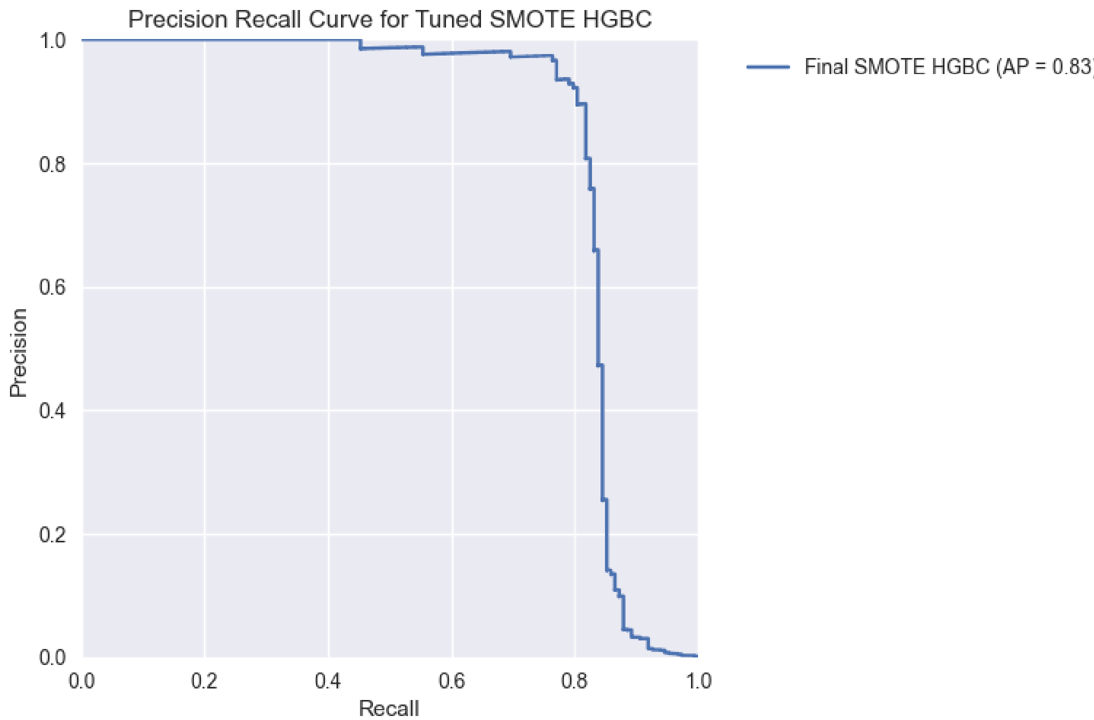
[110]:
```
HistGradientBoostingClassifier(early_stopping=True,
                               learning_rate=np.float64(0.10365439161475765),
                               max_iter=300, max_leaf_nodes=100)
```

[117]:
```
train_pred = final_boost_model.predict(X_val)
train_pred_proba = final_boost_model.predict_proba(X_val)[:, 1]

final_boost_prediction = final_boost_model.predict(X_test_full)
final_boost_prediction_proba = final_boost_model.predict_proba(X_test_full)[:,␣
 ↪1]

print("=== Training Set Performance ===")
print(classification_report(y_val, train_pred, target_names=['Not Fraud',␣
 ↪'Fraud']))
print(f"PR-AUC (Train): {average_precision_score(y_val, train_pred_proba):.4f}")
print("="*50)

print("=== Test Set Performance ===")
print(classification_report(y_test_full, final_boost_prediction,␣
 ↪target_names=['Not Fraud', 'Fraud']))
```

```
print(f"PR-AUC: {average_precision_score(y_test_full,␣
↪final_boost_prediction_proba):.4f}")
```

```
=== Training Set Performance ===
              precision    recall  f1-score   support

   Not Fraud       1.00      1.00      1.00     42647
       Fraud       0.85      0.82      0.84        74

    accuracy                           1.00     42721
   macro avg       0.92      0.91      0.92     42721
weighted avg       1.00      1.00      1.00     42721

PR-AUC (Train): 0.8336
==================================================
=== Test Set Performance ===
              precision    recall  f1-score   support

   Not Fraud       1.00      1.00      1.00     85295
       Fraud       0.86      0.82      0.84       148

    accuracy                           1.00     85443
   macro avg       0.93      0.91      0.92     85443
weighted avg       1.00      1.00      1.00     85443

PR-AUC: 0.8327
```

[114]:
```
disp = PrecisionRecallDisplay.from_estimator(
    final_boost_model, X_test_full, y_test_full, pos_label=1, marker="+",␣
↪name="Final SMOTE HGBC"
)

plt.xlabel("Recall")
plt.ylabel("Precision")
plt.xlim(0, 1)
plt.ylim(0, 1)
plt.legend(bbox_to_anchor=(1.05, 1), loc="upper left")
plt.title("Precision Recall Curve for Tuned SMOTE HGBC")

plt.show()
```



[119]:
```
from imblearn.pipeline import Pipeline
from imblearn.over_sampling import SMOTE
from sklearn.ensemble import HistGradientBoostingClassifier
from sklearn.model_selection import RandomizedSearchCV, cross_validate,␣
↪StratifiedKFold
from sklearn.metrics import make_scorer, f1_score, average_precision_score
from scipy.stats import loguniform

# Hyperparemeter tuning for Random Forest

inner_cv = StratifiedKFold(n_splits=3, shuffle=True, random_state=42)
outer_cv = StratifiedKFold(n_splits=2, shuffle=True, random_state=42)

scoring = {
    "f1": make_scorer(f1_score, pos_label=1),
    "average_precision": make_scorer(average_precision_score, pos_label=1)
}

rf_param_distributions = {
    "classifier__max_leaf_nodes": [10, 100, 200, None],
    "classifier__max_features": [1, 2, 3, 5, None],
    "classifier__min_samples_leaf": [1, 5, 10, 50, 100]
}
```

```
forest_pipeline = Pipeline([
    ("smote", SMOTE(random_state=42)),
    ("classifier", RandomForestClassifier())
])

forest_random_search = RandomizedSearchCV(forest_pipeline,␣
 ↪param_distributions=rf_param_distributions, cv=inner_cv, n_iter=5, n_jobs=1)

cv_results = cross_validate(forest_random_search, X_train, y_train,␣
 ↪cv=outer_cv, scoring=scoring, return_estimator=True, n_jobs=1)

rf_test_f1 = cv_results["test_f1"]
rf_test_ap = cv_results["test_average_precision"]

print(f"Tuned RF mean f1: {rf_test_f1.mean():.3f}")
print(f"Tuned RF mean AP: {rf_test_ap.mean():.3f}")
```

```
Tuned RF mean f1: 0.772
Tuned RF mean AP: 0.605
```

[120]:

```
#Extract Random Forest best params

best_estimators = cv_results["estimator"]
best_params_list = [
   {k: tuple(v) if isinstance(v, list) else v for k, v in est.best_estimator_.
 ↪get_params().items()}
   for est in best_estimators
]
best_params = Counter(tuple(sorted(p.items())) for p in best_params_list).
 ↪most_common(1)[0][0]
best_params = dict(best_params)

classifier_params = {k.replace('classifier__', ''): v for k, v in best_params.
 ↪items()
                      if k.startswith('classifier__')}
```

[122]:

```
final_rf_model = RandomForestClassifier(**classifier_params)
final_rf_model.fit(x_train_smote, y_train_smote)
```

[122]:

```
RandomForestClassifier(max_features=3, max_leaf_nodes=200, min_samples_leaf=5)
```

[124]:

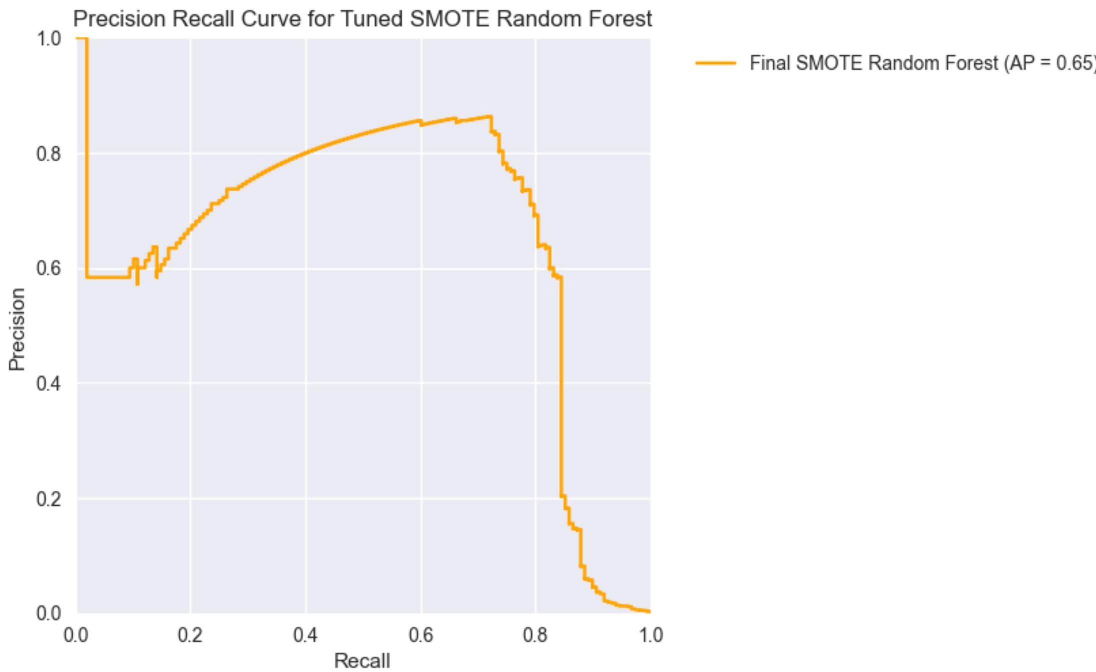
```
rf_train_pred = final_rf_model.predict(X_val)
rf_train_pred_proba = final_rf_model.predict_proba(X_val)[:,1]

final_rf_pred = final_rf_model.predict(X_test_full)
final_rf_pred_proba = final_rf_model.predict_proba(X_test_full)[:, 1]
```

```
print("=== Training Set Performance ===")
print(classification_report(y_val, rf_train_pred , target_names=['Not Fraud',␣
↪'Fraud']))
print(f"PR-AUC (Train): {average_precision_score(y_val, rf_train_pred_proba):.
↪4f}")
print("="*50)

print("=== Test Set Performance ===")
print(classification_report(y_test_full, final_rf_pred , target_names=['Not␣
↪Fraud', 'Fraud']))
print(f"PR-AUC: {average_precision_score(y_test_full, final_rf_pred_proba):.
↪4f}")
```

```
=== Training Set Performance ===
              precision    recall  f1-score   support

   Not Fraud       1.00      1.00      1.00     42647
       Fraud       0.56      0.84      0.67        74

    accuracy                           1.00     42721
   macro avg       0.78      0.92      0.84     42721
weighted avg       1.00      1.00      1.00     42721

PR-AUC (Train): 0.5997
==================================================
=== Test Set Performance ===
              precision    recall  f1-score   support

   Not Fraud       1.00      1.00      1.00     85295
       Fraud       0.58      0.84      0.69       148

    accuracy                           1.00     85443
   macro avg       0.79      0.92      0.84     85443
weighted avg       1.00      1.00      1.00     85443

PR-AUC: 0.6454
```

[125]:
```
disp = PrecisionRecallDisplay.from_estimator(
    final_rf_model, X_test_full, y_test_full, pos_label=1, marker="+",␣
↪name="Final SMOTE Random Forest", color="orange"
)

plt.xlabel("Recall")
plt.ylabel("Precision")
plt.xlim(0, 1)
plt.ylim(0, 1)
```

```
plt.legend(bbox_to_anchor=(1.05, 1), loc="upper left")
plt.title("Precision Recall Curve for Tuned SMOTE Random Forest")

plt.show()
```



Current hyperparemeter choices for the Random Forest Classifier might have caused it to underfit. While the hyperparemeter tuning has substantially improved the HistGradientBoostingClassifier scores.

```
[ ]:
```