Working With Imbalanced Data Python Scikit-Learn

Ben Brock

Increasing successful detections using data resampling

- One approach to handle class imbalance
  - Undersampling
  - Oversampling
  - SMOTE
- Another approach to handle class imbalance
  - Scikit-learn models class weight option
- Find the optimal machine learning model
  - GridSearchCV

Imbalance-Learn

http://imbalanced-learn.org

```
pip install -U imbalanced-learn
```

Extends `sklearn` API

Example Sampler

To resample a data sets, each sampler implements:

```
data_resampled, targets_resampled = obj.sample(data, targets)
```

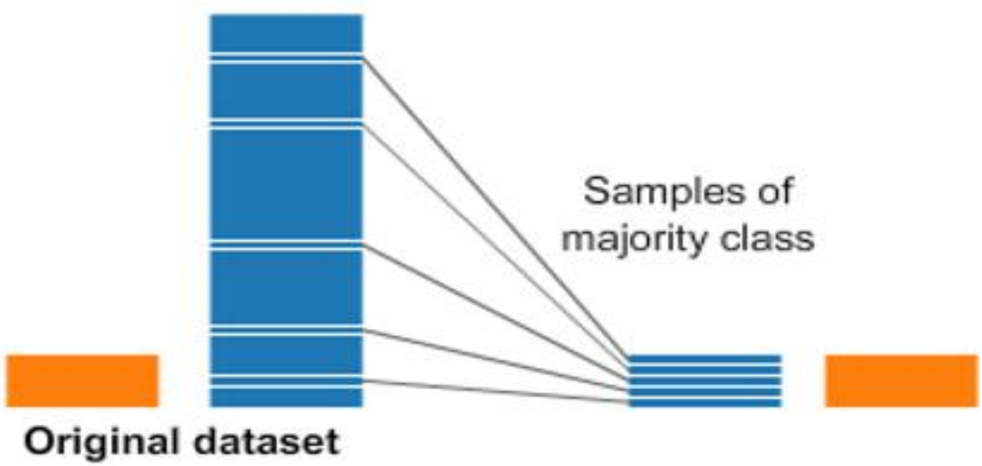
Fitting and sampling can also be done in one step:

```
data_resampled, targets_resampled = obj.fit_sample(data, targets)
```

In Pipelines: Sampling only done in `fit`!

Undersampling

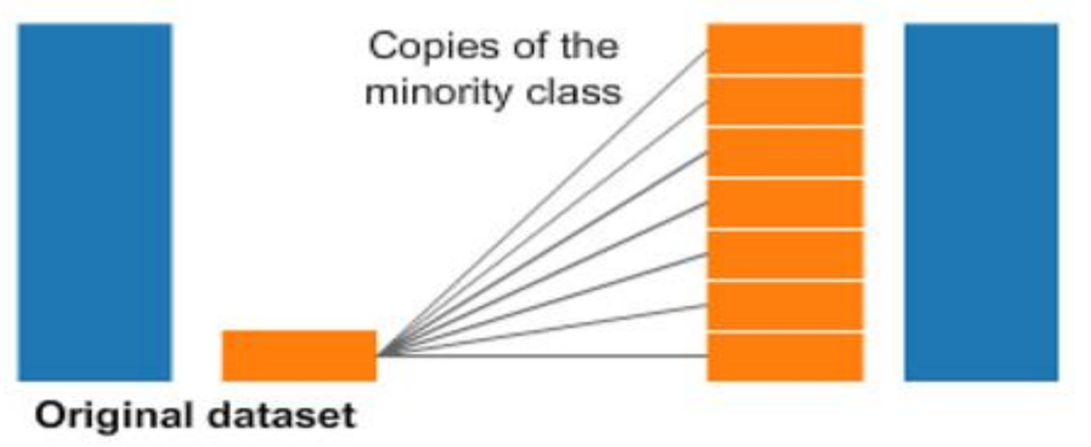
- Undersampling



- Drawback: Throwing away a lot of good data and information

Oversampling

- Oversampling

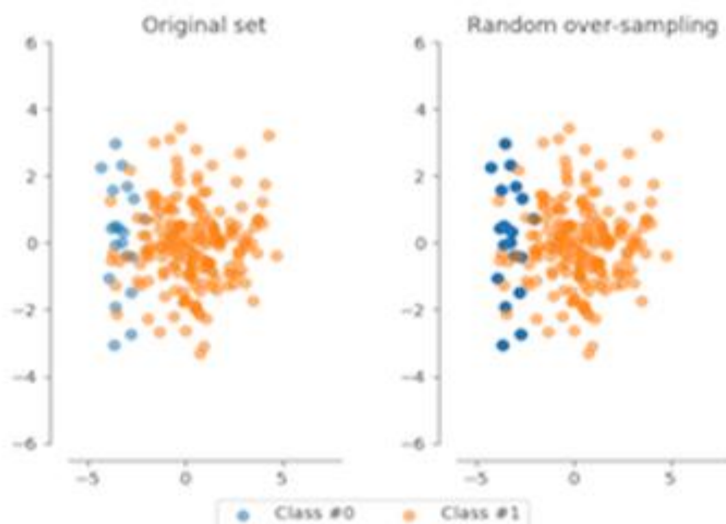


- Drawback: Training on duplicate data

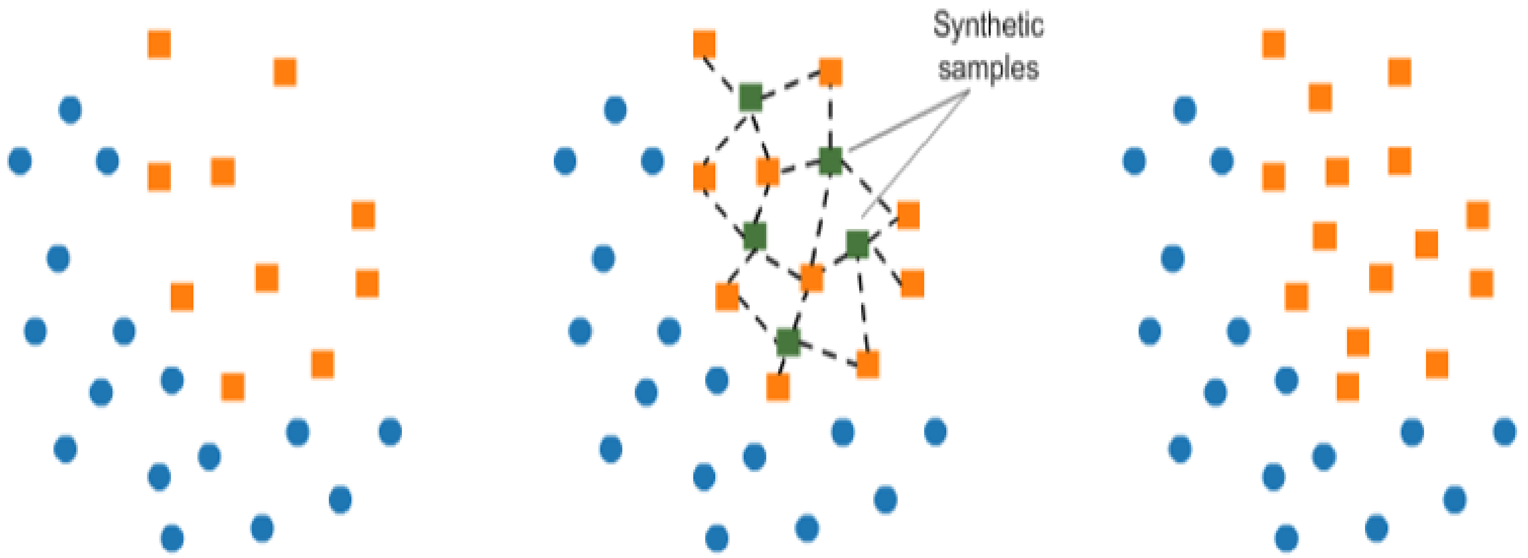
Oversampling in Python

- Compatible with scikit-learn

```
from imblearn.over_sampling import RandomOverSampler

method = RandomOverSampler()
X_resampled, y_resampled = method.fit_sample(X, y)

compare_plots(X_resampled, y_resampled, X, y)
```



Synthetic Minority Oversampling Technique (SMOTE)



Source: https://www.kaggle.com/rafjaa/resampling-strategies-for-imbalanced-datasets

Synthetic Minority Oversampling Technique (SMOTE)

- Oversampling the minority observations
  - Not just copying the minority classes, instead SMOTE uses characteristics of KNN nearest neighbors of old cases to create new synthetic cases, and thereby avoids creating duplications

Which resampling method to use?

- Random Under Sampling (RUS): throw away data, computationally efficient
  - If you have LARGE amount of data and many categorical type (i.e.., fraud) cases
- Random Over Sampling (ROS): straightforward and simple, but training your model on many duplicates
- Synthetic Minority Oversampling Technique (SMOTE): more sophisticated and realistic dataset, but you are training on "fake" data

When to use resampling methods

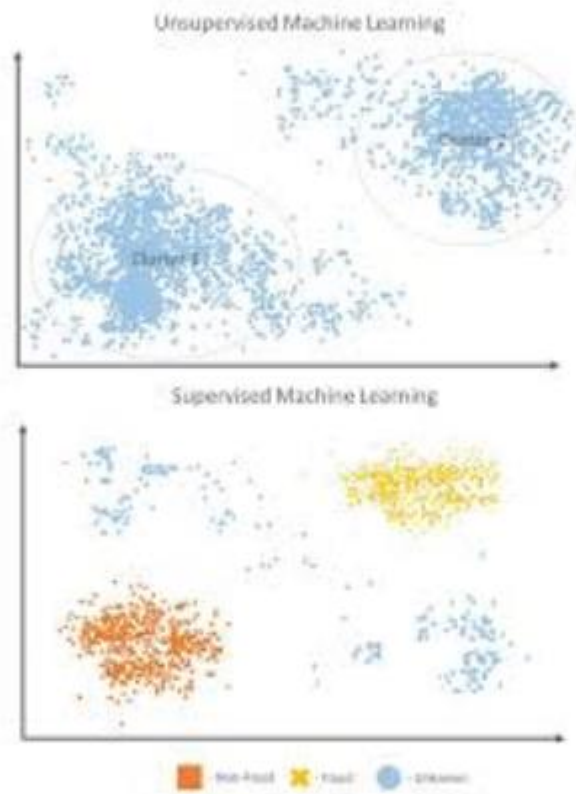
- Use resampling methods on your training set, never on your test set!

```
# Define resampling method and split into train and test
method = SMOTE(kind='borderline1')
X_train, X_test, y_train, y_test = train_test_split(X, y,
 train_size=0.8, random_state=0)

# Apply resampling to the training data only
X_resampled, y_resampled = method.fit_sample(X_train, y_train)

# Continue fitting the model and obtain predictions
model = LogisticRegression()
model.fit(X_resampled, y_resampled)

# Get your performance metrics
predicted = model.predict(X_test)
print (classification_report(y_test, predicted))
```

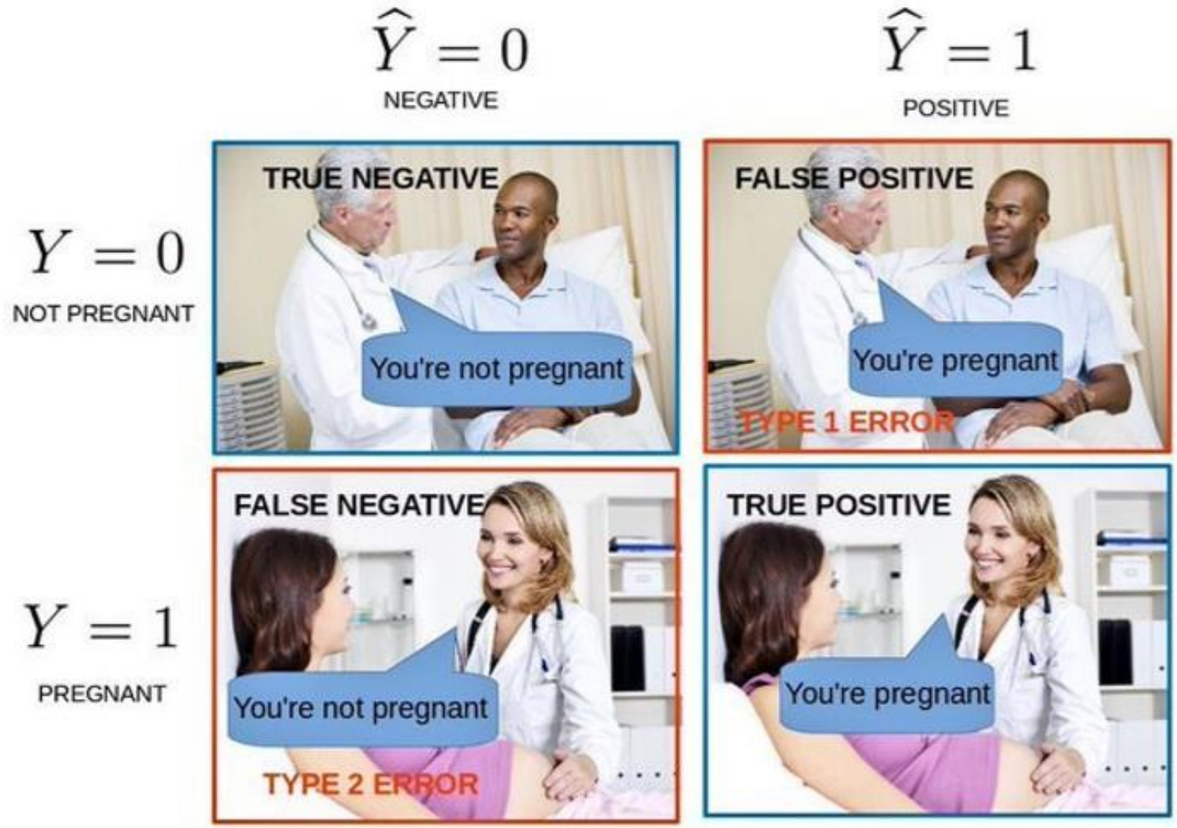
- Goal is to better train your model and give it more balanced data!!!

Why use machine learning

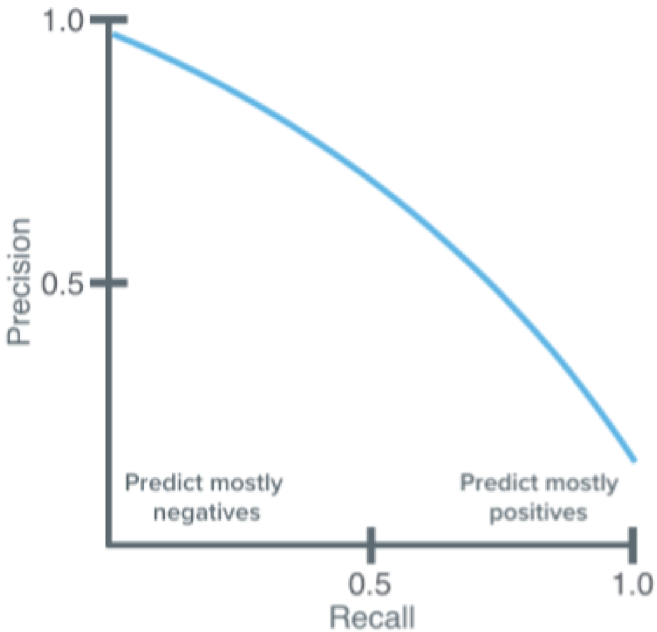
- Machine learning models adapt to the data, and thus can change over time
- Uses all the data combined rather than a threshold per feature
- Can give a score, rather than a yes/no
- Will typically have a better performance and can be combined with rule



False positives, False negatives



Precision Recall Trade-Off



$$Precision = \frac{\#True\ Positives}{\#True\ Positives + \#False\ Positives}$$

$$Recall = \frac{\#True\ Positives}{\#True\ Positives + \#False\ Negatives}$$

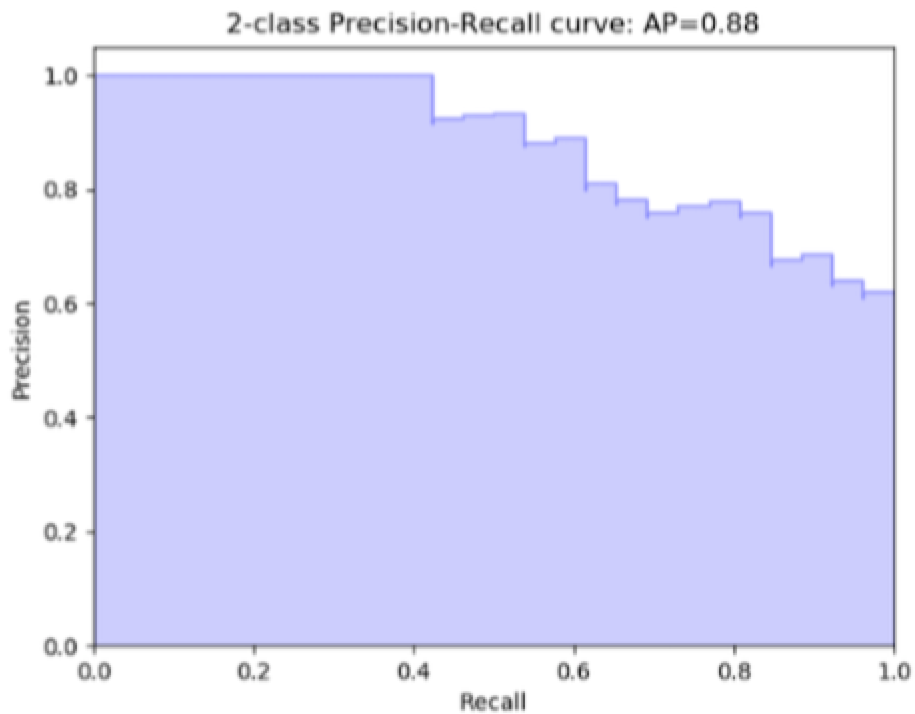
$$F-measure = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

$$= \frac{2 \times TP}{2 \times TP + FP + FN}$$

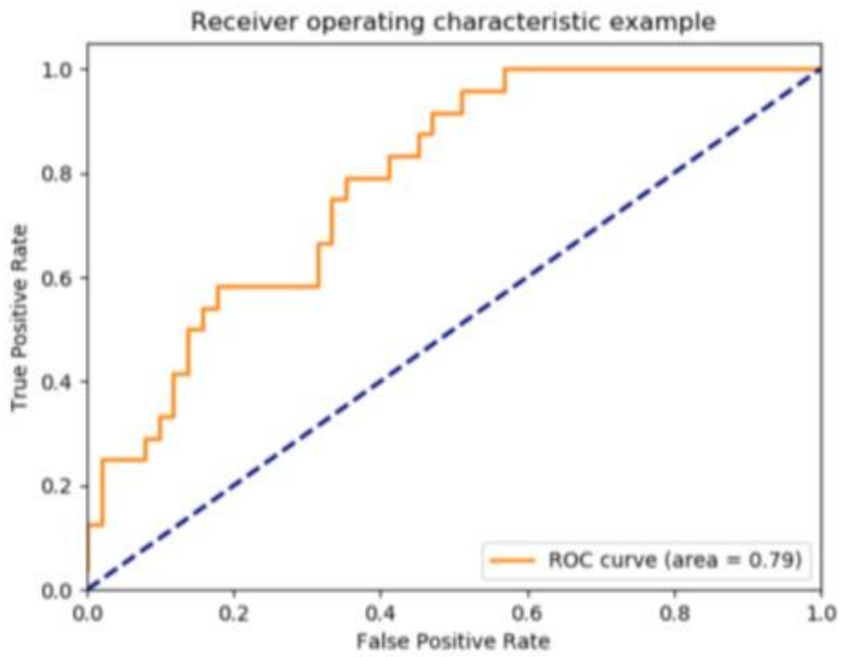
Obtaining performance metrics

```
# Import the packages
from sklearn.metrics import precision_recall_curve
from sklearn.metrics import average_precision_score

# Calculate average precision and the PR curve
average_precision = average_precision_score(y_test, predicted)

# Obtain precision and recall
precision, recall, _ = precision_recall_curve(y_test, predicted)
```

Precision-Recall Curve



ROC curve to compare algorithms




```
# Obtain model probabilities
probs = model.predict_proba(X_test)

# Print ROC_AUC score using probabilities
print(metrics.roc_auc_score(y_test, probs[:, 1]))
```

Confusion Matrix and Classification Report

```
from sklearn.metrics import classification_report, confusion_matrix
```

```
# Obtain predictions
predicted = model.predict(X_test)
```

```
# Print classification report using predictions
print(classification_report(y_test, predicted))

  precision    recall  f1-score   support

        0.0       0.99      1.00      1.00      2099
        1.0       0.96      0.80      0.87        91

avg / total       0.99      0.99      0.99      2190
```

```
# Print confusion matrix using predictions
print(confusion_matrix(y_test, predicted))

[[2096    3]
 [  18   73]]
```

Balance Weights

```
model = RandomForestClassifier(class_weight='balanced')
```

```
model = RandomForestClassifier(class_weight='balanced_subsample')
```

```
model = LogisticRegression(class_weight='balanced')
```

```
model = SVC(kernel='linear', class_weight='balanced', probability=True)
```

Hyperparameter tuning

```
model = RandomForestClassifier(class_weight={0:1,1:4},random_state=1)

model = LogisticRegression(class_weight={0:1,1:4}, random_state=1)
```

```
model = RandomForestClassifier(n_estimators=10,
criterion='gini',
max_depth=None,
min_samples_split=2,
min_samples_leaf=1,
max_features='auto',
n_jobs=-1, class_weight=None)
```

Using GridSearchCV

```
from sklearn.model_selection import GridSearchCV
```

```
# Create the parameter grid
param_grid = {
    'max_depth': [80, 90, 100, 110],
    'max_features': [2, 3],
    'min_samples_leaf': [3, 4, 5],
    'min_samples_split': [8, 10, 12],
    'n_estimators': [100, 200, 300, 1000]
}
```

```
# Define which model to use
model = RandomForestRegressor()
```

```
# Instantiate the grid search model
grid_search_model = GridSearchCV(estimator = model,
param_grid = param_grid, cv = 5,
n_jobs = -1, scoring='f1')
```

Finding the best model with GridSearchCV

```
# Fit the grid search to the data
grid_search_model.fit(X_train, y_train)
```

```
# Get the optimal parameters
grid_search_model.best_params_

{'bootstrap': True,
 'max_depth': 80,
 'max_features': 3,
 'min_samples_leaf': 5,
 'min_samples_split': 12,
 'n_estimators': 100}
```

```
# Get the best_estimator results
grid_search.best_estimator_
grid_search.best_score_
```

Working with Imbalanced Data

- Worked with highly imbalanced data
- Learned how to resample your data
- Learned about different resampling methods

Detailed

Changing Thresholds

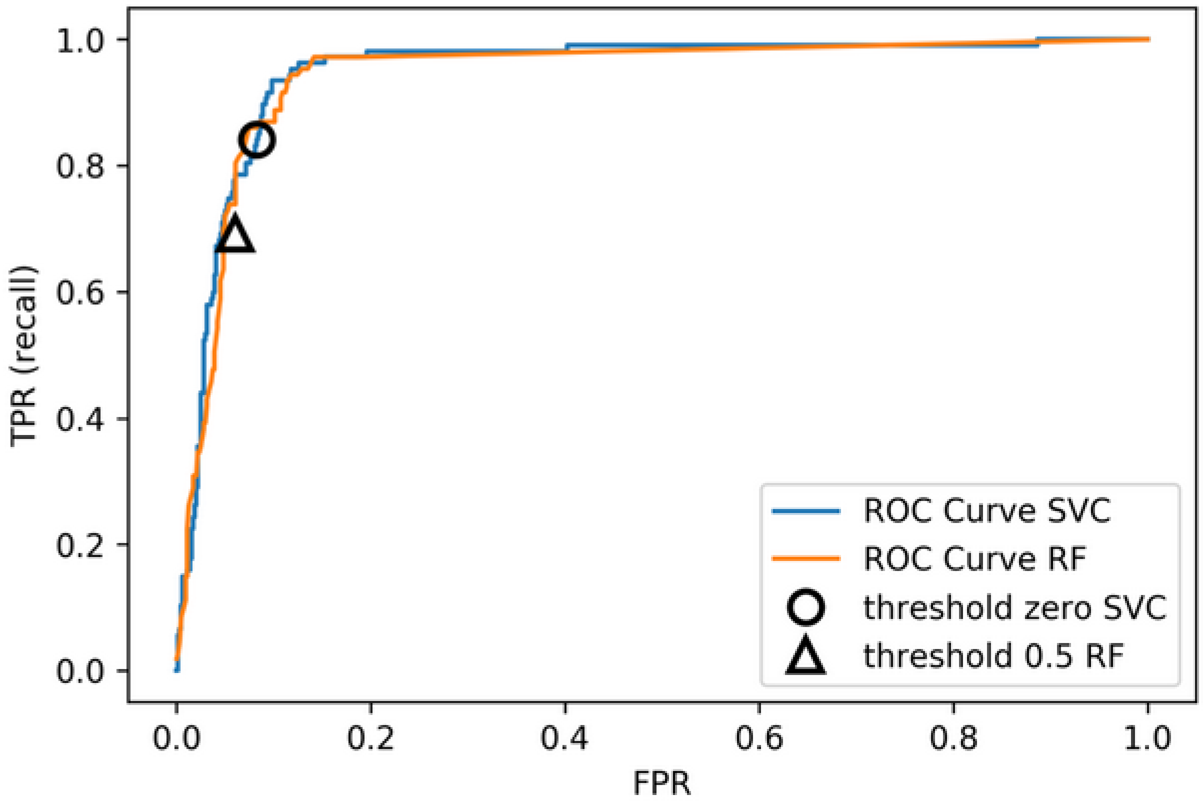
```
data = load_breast_cancer()

X_train, X_test, y_train, y_test = train_test_split(
    data.data, data.target, stratify=data.target, random_state=0)

lr = LogisticRegression().fit(X_train, y_train)
y_pred = lr.predict(X_test)

classification_report(y_test, y_pred)
```

```
          precision    recall  f1-score   support
0              0.91      0.92      0.92        53
1              0.96      0.94      0.95        90
avg/total      0.94      0.94      0.94       143
```

```
y_pred = lr.predict_proba(X_test)[:, 1] > .85

classification_report(y_test, y_pred)
```

```
          precision    recall  f1-score   support
0              0.84      1.00      0.91        53
1              1.00      0.89      0.94        90
avg/total      0.94      0.93      0.93       143
```

ROC Curve


1.0
0.8
0.6
0.4
0.2
0.0
TPR (recall)
0.0
0.2
0.4
0.6
0.8
1.0
FPR
ROC Curve SVC
ROC Curve RF
threshold zero SVC
threshold 0.5 RF

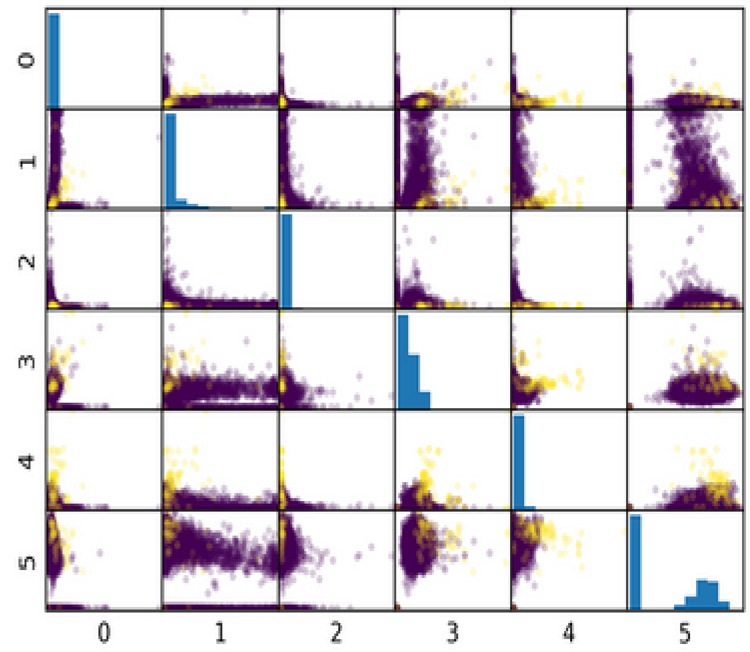
Mammography Data

```
from sklearn.datasets import fetch_openml
# mammography https://www.openml.org/d/310
data = fetch_openml('mammography')
X, y = data.data, data.target
y = (y.astype(np.int) + 1) // 2
X.shape
```

```
(11183, 6)
```

```
np.bincount(y)
```

```
array([10923,   260])
```



Mammography Data

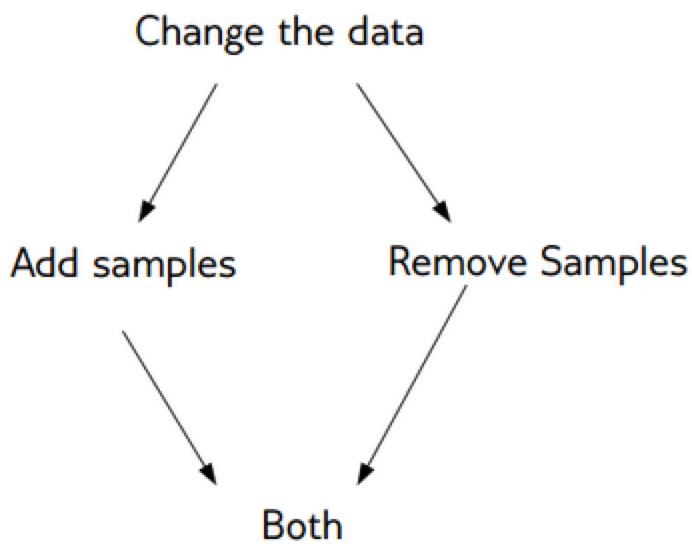
```
from sklearn.model_selection import cross_validate
from sklearn.linear_model import LogisticRegression

scores = cross_validate(LogisticRegression(),
                        X_train, y_train, cv=10,
                        scoring=('roc_auc', 'average_precision'))
scores['test_roc_auc'].mean(), scores['test_average_precision'].mean()
```

0.920, 0.630

```
from sklearn.ensemble import RandomForestClassifier
scores = cross_validate(RandomForestClassifier(n_estimators=100),
                        X_train, y_train, cv=10,
                        scoring=('roc_auc', 'average_precision'))
scores['test_roc_auc'].mean(), scores['test_average_precision'].mean()
```

0.939, 0.722

Basic Approaches



Change the training procedure

Sampler

To resample a data sets, each sampler implements:

```
data_resampled, targets_resampled = obj.sample(data, targets)
```

Fitting and sampling can also be done in one step:

```
data_resampled, targets_resampled = obj.fit_sample(data, targets)
```

In Pipelines: Sampling only done in `fit`!

Random Undersampling

```
from imblearn.under_sampling import RandomUnderSampler
rus = RandomUnderSampler(replacement=False)
X_train_subsample, y_train_subsample = rus.fit_sample(
    X_train, y_train)
print(X_train.shape)
print(X_train_subsample.shape)
print(np.bincount(y_train_subsample))
```

```
(8387, 6)
(390, 6)
[195 195]
```

Random Undersampling

```
from imblearn.pipeline import make_pipeline as make_imb_pipeline

undersample_pipe = make_imb_pipeline(RandomUnderSampler(), LogisticRegressionCV())
scores = cross_validate(undersample_pipe,
                        X_train, y_train, cv=10,
                        scoring=('roc_auc', 'average_precision'))
scores['test_roc_auc'].mean(), scores['test_average_precision'].mean()
# baseline was 0.920, 0.630
```

```
0.927, 0.527
```

```
undersample_pipe_rf = make_imb_pipeline(RandomUnderSampler(),
                                        RandomForestClassifier())
scores = cross_validate(undersample_pipe_rf,
                        X_train, y_train, cv=10,
                        scoring=('roc_auc', 'average_precision'))
scores['test_roc_auc'].mean(), scores['test_average_precision'].mean()
# baseline was 0.939, 0.722
```

Random Oversampling

```
from imblearn.over_sampling import RandomOverSampler
ros = RandomOverSampler()
X_train_oversample, y_train_oversample = ros.fit_sample(
    X_train, y_train)
print(X_train.shape)
print(X_train_oversample.shape)
print(np.bincount(y_train_oversample))
```

```
(8387, 6)
(16384, 6)
[8192 8192]
```

Random Oversampling

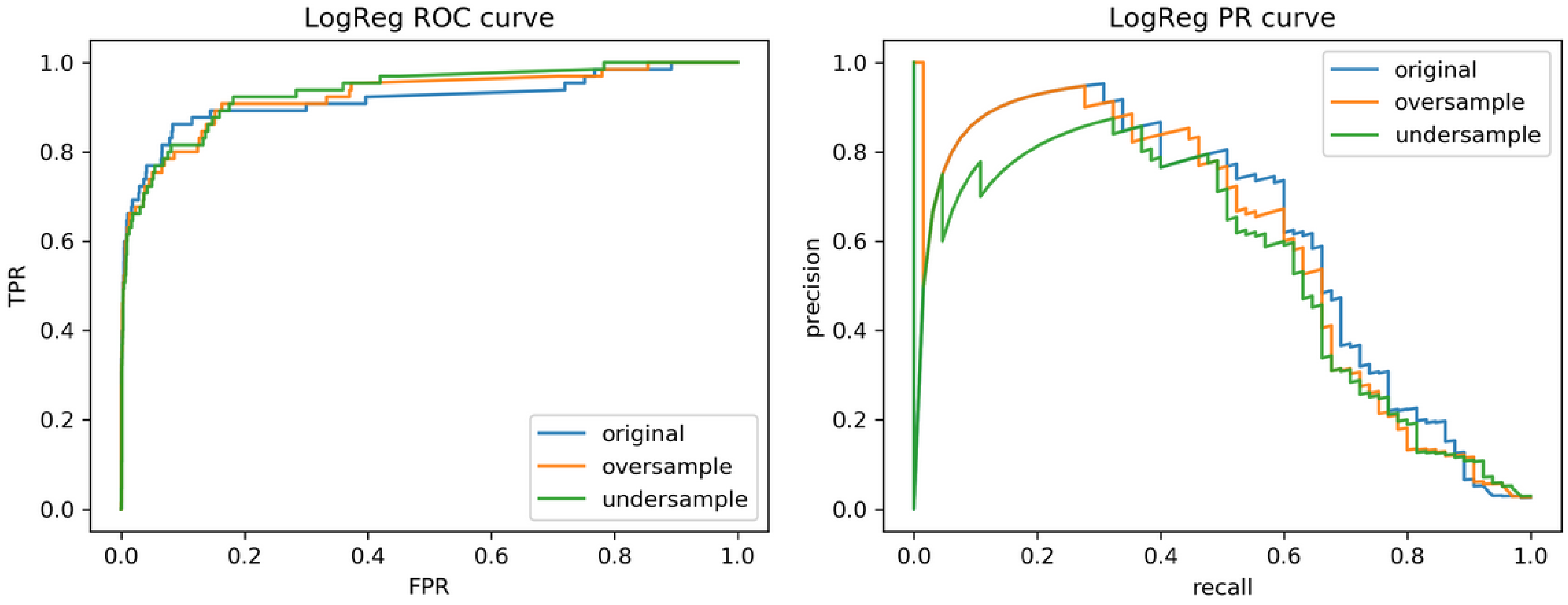
```
oversample_pipe = make_imb_pipeline(RandomOverSampler(), LogisticRegression())
scores = cross_validate(oversample_pipe,
                        X_train, y_train, cv=10,
                        scoring=('roc_auc', 'average_precision'))
scores['test_roc_auc'].mean(), scores['test_average_precision'].mean()
# baseline was 0.920, 0.630
```

0.917, 0.585

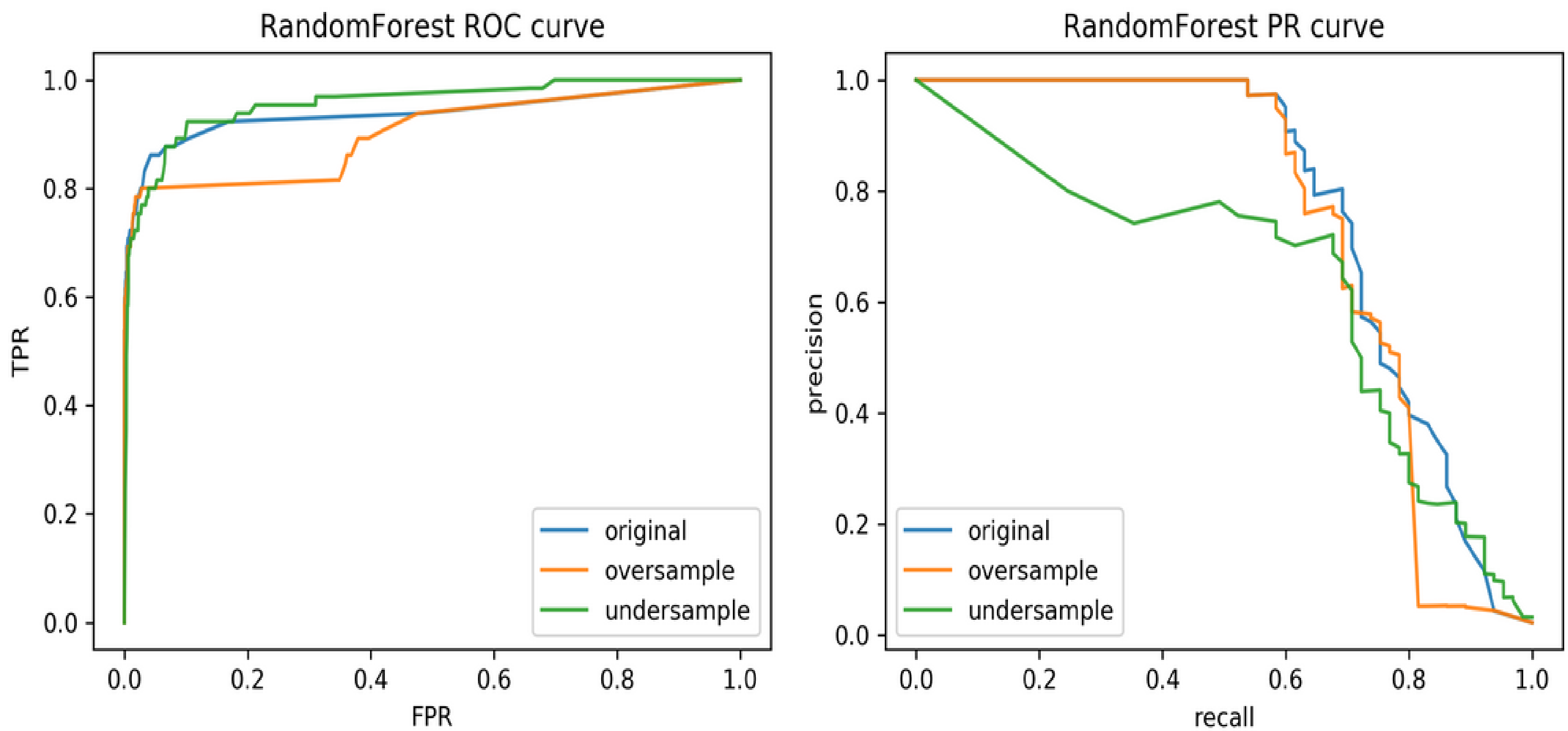
```
oversample_pipe_rf = make_imb_pipeline(RandomOverSampler(),
                                       RandomForestClassifier())
scores = cross_validate(oversample_pipe_rf,
                        X_train, y_train, cv=10,
                        scoring=('roc_auc', 'average_precision'))
scores['test_roc_auc'].mean(), scores['test_average_precision'].mean()
# baseline was 0.939, 0.722
```

0.926, 0.715

Curves for LogReg



Curves for Random Forest



ROC or PR?

FPR or Precision?

$$\mathrm{FPR} = \frac{\mathrm{FP}}{\mathrm{FP} + \mathrm{TN}}$$

$$\mathrm{Precision} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}}$$

Class-weights

- Instead of repeating samples, re-weight the loss function
- Works for most models!
- Same effect as over-sampling (though not random), but not as expensive (dataset size the same).

Class-weights in linear models

$$\min_{w \in \mathbb{R}^p, b \in \mathbb{R}} -C \sum_{i=1}^{n} \log(\exp(-y_i(w^T \mathbf{x}_i + b)) + 1) + ||w||_2^2$$

$$\min_{w \in \mathbb{R}^p, b \in \mathbb{R}} -C \sum_{i=1}^{n} c_{y_i} \log(\exp(-y_i(w^T \mathbf{x}_i + b)) + 1) + ||w||_2^2$$

Similar for linear and non-linear SVM

Class weights in trees

Gini Index:

$$H_{\text{gini}}(X_m) = \sum_{k \in \mathcal{Y}} p_{mk}(1 - p_{mk})$$

$$H_{\text{gini}}(X_m) = \sum_{k \in \mathcal{Y}} c_k p_{mk}(1 - p_{mk})$$

Prediction:

Weighted vote

Using Class-Weights

```
scores = cross_validate(LogisticRegression(class_weight='balanced'),
                        X_train, y_train, cv=10,
                        scoring=('roc_auc', 'average_precision'))
scores['test_roc_auc'].mean(), scores['test_average_precision'].mean()
# baseline was 0.920, 0.630
```

0.918, 0.587

```
scores = cross_validate(RandomForestClassifier(n_estimators=100,
                                               class_weight='balanced'),
                        X_train, y_train, cv=10,
                        scoring=('roc_auc', 'average_precision'))
scores['test_roc_auc'].mean(), scores['test_average_precision'].mean()
# baseline was 0.939, 0.722
```

0.917, 0.701

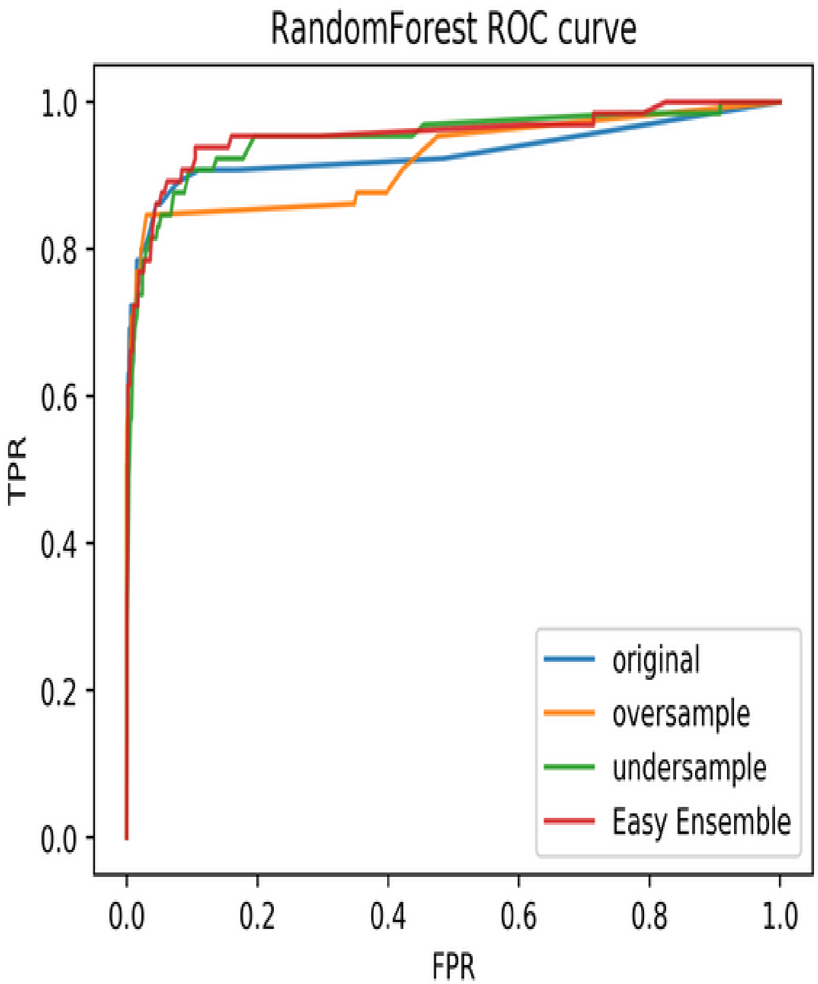
Ensemble Resampling

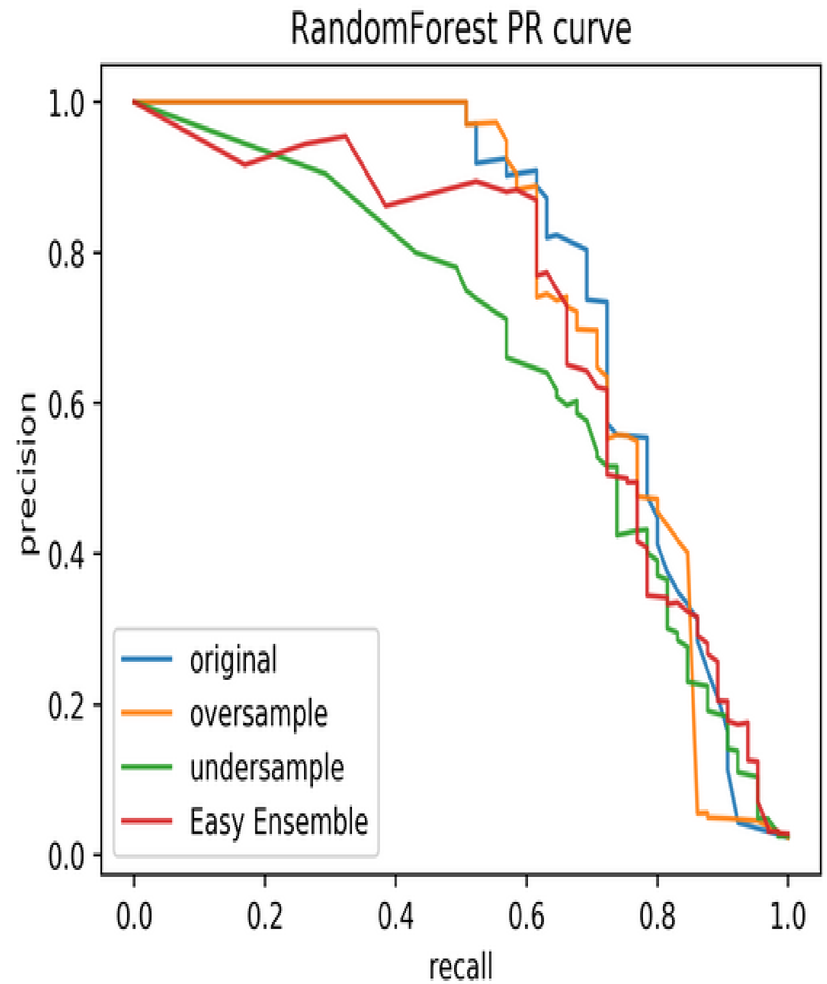
- Random resampling separate for each instance in an ensemble!
- Chen, Liaw, Breiman: “Using random forest to learn imbalanced data.”
- Paper: “Exploratory Undersampling for Class Imbalance Learning”
- Not in sklearn (yet)
- Easy with imblearn

Easy Ensemble with imblearn

```
from sklearn.tree import DecisionTreeClassifier
from imblearn.ensemble import BalancedBaggingClassifier

# from imblearn.ensemble import BalancedRandomForestClassifier
# resampled_rf = BalancedRandomForestClassifier()

tree = DecisionTreeClassifier(max_features='auto')
resampled_rf = BalancedBaggingClassifier(base_estimator=tree,
                                         n_estimators=100, random_state=0)
scores = cross_validate(resampled_rf,
                        X_train, y_train, cv=10,
                        scoring=('roc_auc', 'average_precision'))
scores['test_roc_auc'].mean(), scores['test_average_precision'].mean()
# baseline was 0.939, 0.722
```

0.957, 0.654

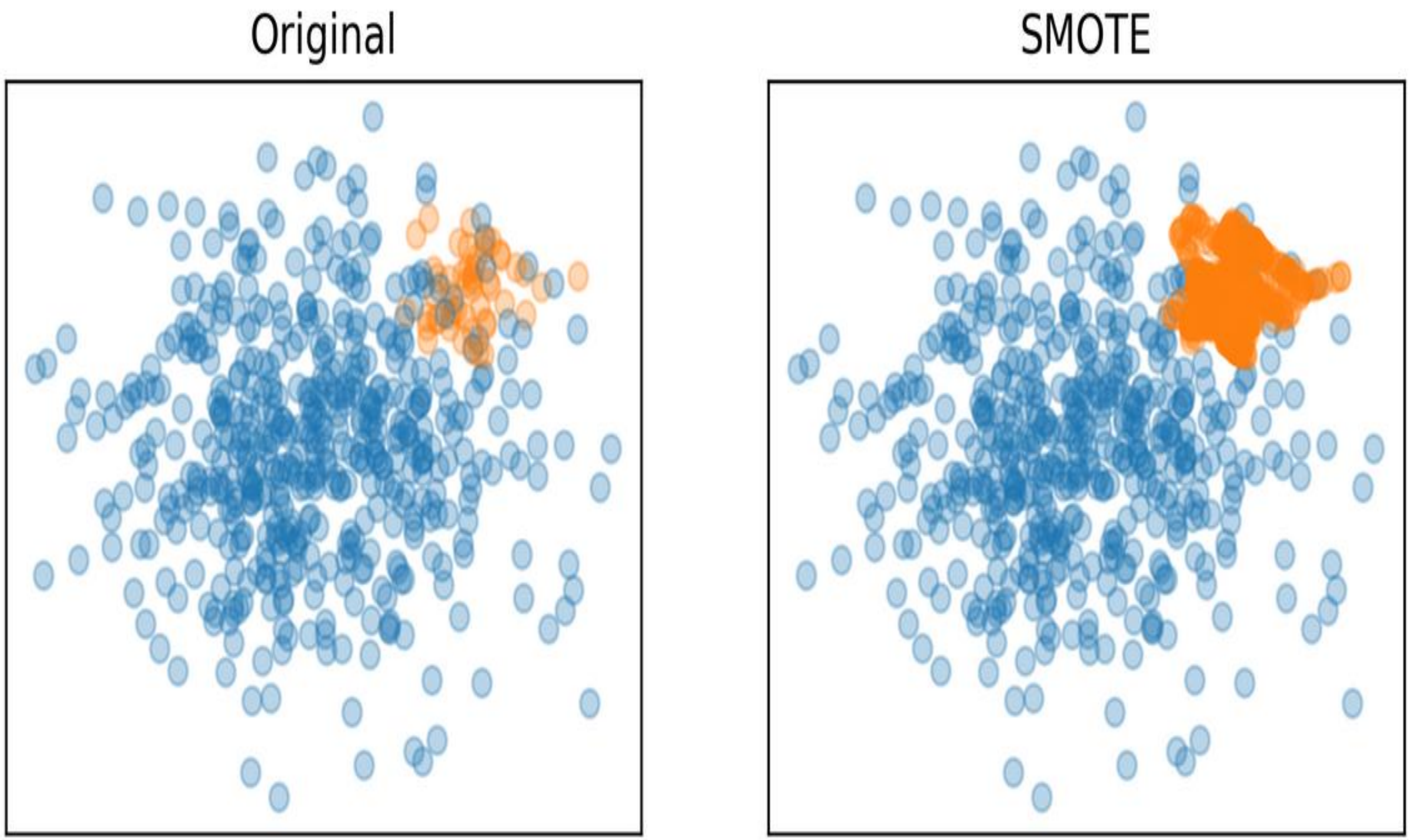

RandomForest ROC curve
TPR
FPR
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
original
oversample
undersample
Easy Ensemble


RandomForest PR curve
precision
recall
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
original
oversample
undersample
Easy Ensemble

Synthetic Sample Generation

Synthetic Minority Oversampling Technique (SMOTE)

- Adds synthetic interpolated data to smaller class
- For each sample in minority class:
  - Pick random neighbor from k neighbors.
  - Pick point on line connecting the two uniformly (or within rectangle)
  - Repeat


Original
SMOTE

SMOTE ...

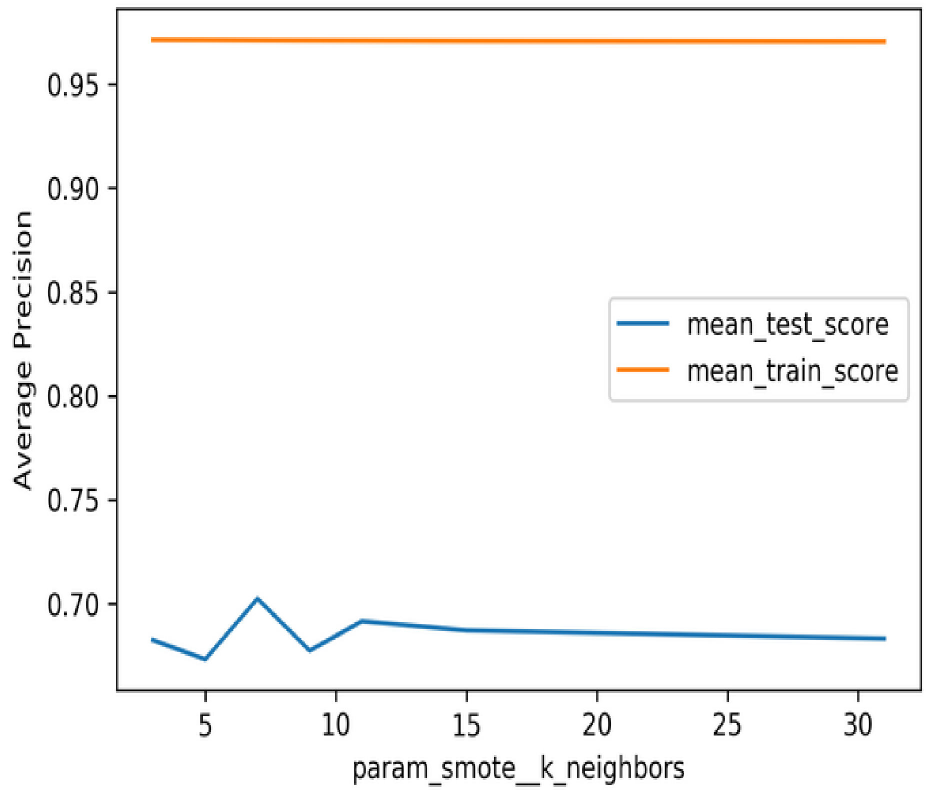
```
smote_pipe = make_imb_pipeline(SMOTE(), LogisticRegression())
scores = cross_validate(smote_pipe, X_train, y_train, cv=10,
                        scoring=('roc_auc', 'average_precision'))
pd.DataFrame(scores)[['test_roc_auc', 'test_average_precision']].mean()
# baseline was 0.920, 0.630
```

0.919, 0.585

```
smote_pipe_rf = make_imb_pipeline(SMOTE(),
                                  RandomForestClassifier(n_estimators=100))
scores = cross_validate(smote_pipe_rf, X_train, y_train, cv=10,
                        scoring=('roc_auc', 'average_precision'))
pd.DataFrame(scores)[['test_roc_auc', 'test_average_precision']].mean()
# baseline was 0.939, 0.722
```

0.946, 0.688

```python
param_grid = {'smote__k_neighbors': [3, 5, 7, 9, 11, 15, 31]}
search = GridSearchCV(smote_pipe_rf, param_grid, cv=10,
                      scoring="average_precision")
search.fit(X_train, y_train)
results = pd.DataFrame(search.cv_results_)
results.plot("param_smote__k_neighbors", ["mean_test_score", "mean_train_score"])
```



Summary

- Always check roc_auc an AP, look at curves
- Undersampling is very fast and can help!
- Undersampling + Ensembles worth a try.
- Many smart sampling strategies, mixed outcomes
- SMOTE allows adding new interpolated samples
- Mixed outcomes with SMOTE, also definition a bit unclear

References

- https://arxiv.org/pdf/1106.1813.pdf
- http://cs.nju.edu.cn/zhouzh/zhouzh.files/publication/tsmcb09.pdf

Miscellaneous