UNIVERSITAS INDONESIA

FAKULTAS ILMU KOMPUTER

Imbalanced Classification

Siti Aminah*, Adila A. Krisnadhi, Dina Chahyati, Aruni Y. Azizah, Fariz Darari

CSGE603130: Kecerdasan Artifisial dan Sains Data Dasar

Semester Gasal 2022/2023

Learning Objectives

- Mahasiswa dapat menjelaskan tantangan yang dihadapi dalam *imbalanced classification*
- Ketika diberikan masalah imbalanced classification, mahasiswa dapat
  - memilih *classification metric* yang sesuai
  - menerapkan *stratified cross validation* pada saat membagi data untuk training dan testing
  - menerapkan teknik oversampling, undersampling dan kombinasi keduanya.
  - menerapkan cost sensitive learning

UNIVERSITAS INDONESIA

FAKULTAS ILMU KOMPUTER

Outline

- What is Imbalanced Classification?
- Challenge of Imbalanced Classification
- Evaluation Metrics
- Cross Validation of Imbalanced Dataset
- Data Sampling
- Cost-Sensitive Learning

References

Main Reference:

- Jason Brownlee, Imbalanced Classification with Python, Machine Learning Mastery, V1.3, 2021

Additional References:

- https://www.sciencedirect.com/science/article/pii/S2215098621001233
- https://github.com/minoue-xx/Oversampling-Imbalanced-Data
- https://imbalanced-learn.org

UNIVERSITAS INDONESIA

FAKULTAS ILMU KOMPUTER

What is Imbalanced Classification

What is Imbalanced Classification?

- Imbalanced classification is the problem of classification when there is an unequal distribution of classes in the training dataset.
- The imbalance in the class distribution may vary, but a severe imbalance is more challenging to model and may require specialized techniques.
- Many real-world classification problems have an imbalanced class distribution, such as fraud detection, spam detection, and churn prediction.
- Other names:
  - Rare event prediction.
  - Extreme event prediction.
  - Severe class imbalance.

What is Imbalanced Classification?

- Slight Imbalance: the distribution of examples is uneven by a small amount in the training dataset (e.g. 4:6).
- Severe Imbalance: the distribution of examples is uneven by a large amount in the training dataset (e.g. 1:100 or more)

- Most of the contemporary works in class imbalance concentrate on imbalance ratios ranging from 1:4 up to 1:100.
- In real-life applications such as fraud detection or cheminformatics we may deal with problems with imbalance ratio ranging from 1:1000 up to 1:5000.

- A slight imbalance is often not a concern, and the problem can often be treated like a normal classification predictive modeling problem. A severe imbalance of the classes can be challenging to model and may require the use of specialized techniques.

Examples of Imbalanced Classification

- Fraud Detection.
- Claim Prediction
- Default Prediction.
- Churn Prediction.
- Spam Detection.
- Anomaly Detection.
- Outlier Detection.
- Intrusion Detection
- Conversion Prediction

Cause of Class Imbalance

- Data sampling
  - Biased sampling
  - Measurement errors
- Properties of the domain

UNIVERSITAS INDONESIA

FAKULTAS ILMU KOMPUTER

Challenge of Imbalanced Classification

Challenge of Imbalanced Classification

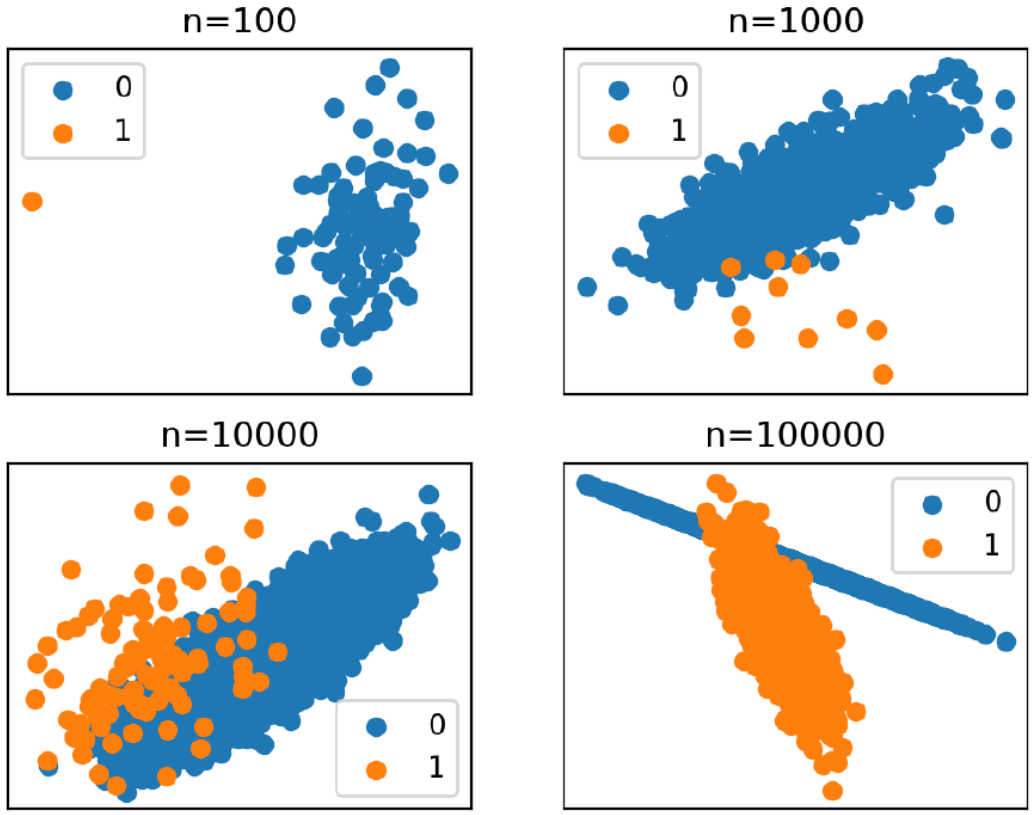
- Skewed Class Distribution
  - most machine learning algorithms will perform poorly and require modification to avoid simply predicting the majority class in all cases.
  - metrics like classification accuracy lose their meaning
- Cost Sensitivity of Misclassification Errors
  - Misclassifying an example from the majority class (normal case) as an example from the minority class (abnormal case) called a false positive is often not desired, but less critical than classifying an example from the minority class as belonging to the majority class, a so-called false negative

Why Imbalanced Classification Is Hard

- Class imbalance was widely acknowledged as a complicating factor for classification. However, some studies also argue that the imbalance ratio is not the only cause of performance degradation in learning from imbalanced data.
- Three of the most characteristics include:
  - Dataset Size.
  - Label Noise.
  - Data Distribution.
- Understanding these data intrinsic characteristics, as well as their relationship with class imbalance, is crucial for applying existing and developing new techniques to deal with imbalance data.

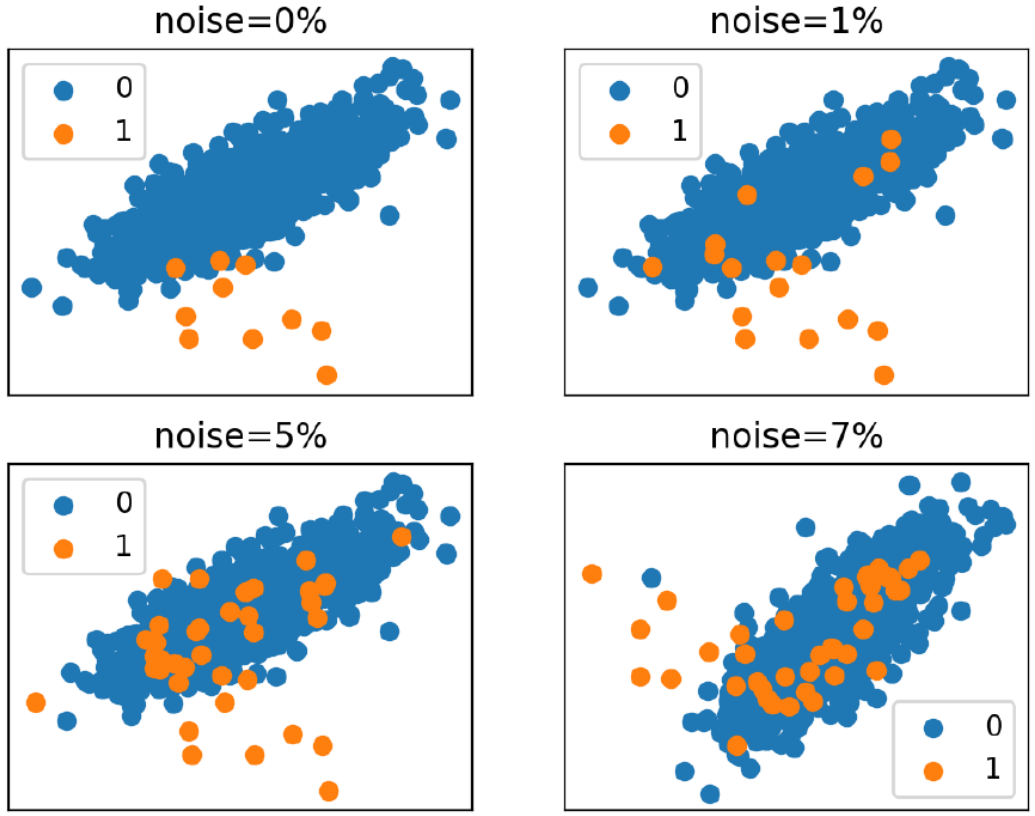
Compounding Effect of Dataset Size

- A problem that often arises in classification is the small number of training instances. This issue, often reported as data rarity or lack of data, is related to the lack of density or insufficiency of information.
- 1:100 examples with different size of dataset:



Compounding Effect of Label Noise

- Two types of noise are distinguished in the literature: feature (or attribute) and class noise.
- Class noise is generally assumed to be more harmful than attribute noise in ML. Class noise somehow affects the observed class values (e.g., by somehow flipping the label of a minority class instance to the majority class label).

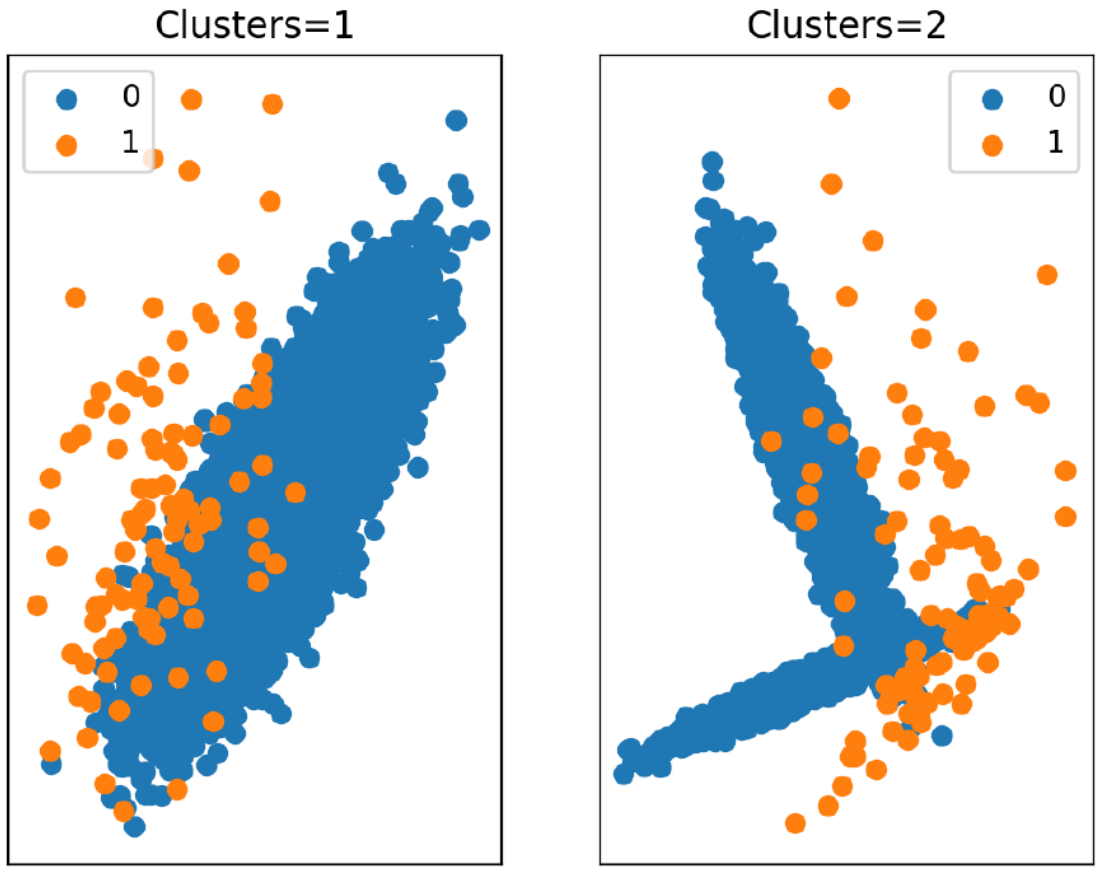


Compounding Effect of Data Distribution

- Another important consideration is the distribution of examples in feature space. If we think about feature space spatially, we might like all examples in one class to be located on one part of the space, and those from the other class to appear in another part of the space.
- If this is the case, we have good class separability and machine learning models can draw crisp class boundaries and achieve good classification performance. This holds on datasets with a balanced or imbalanced class distribution.
- This is rarely the case, and it is more likely that each class has multiple concepts resulting in multiple different groups or clusters of examples in feature space.
- For example, we might consider data that describes whether a patient is healthy (majority class) or sick (minority class). The data may capture many different types of illnesses, and there may be groups of similar illnesses, but if there are so few cases, then any grouping or concepts within the class may not be apparent and may look like a diffuse set mixed in with healthy cases.

Compounding Effect of Data Distribution

- Scatter Plots of an Imbalanced Classification Dataset With Different Numbers of Clusters.



UNIVERSITAS INDONESIA

FAKULTAS ILMU KOMPUTER

Evaluation Metric

How to Choose the Right Evaluation Metric

- Read the Data Science course materials “Evaluation Metric” (Dina Chahyati, Siti Aminah, Ari Wibisono, 2020)

UNIVERSITAS INDONESIA

FAKULTAS ILMU KOMPUTER

Cross Validation for Imbalanced Dataset

Cross Validation for Imbalanced Dataset

- k-fold cross-validation is not appropriate for evaluating imbalanced classifiers, because the data is split into k-folds with a uniform probability distribution.

```
# example of k-fold cross-validation with an imbalanced dataset
from sklearn.datasets import make_classification
from sklearn.model_selection import KFold
# generate 2 class dataset
X, y = make_classification(n_samples=1000, n_classes=2, weights=[0.99, 0.01], flip_y=0,
    random_state=1)
kfold = KFold(n_splits=5, shuffle=True, random_state=1)
# enumerate the splits and summarize the distributions
for train_ix, test_ix in kfold.split(X):
  # select rows
  train_X, test_X = X[train_ix], X[test_ix]
  train_y, test_y = y[train_ix], y[test_ix]
  # summarize train and test composition
  train_0, train_1 = len(train_y[train_y==0]), len(train_y[train_y==1])
  test_0, test_1 = len(test_y[test_y==0]), len(test_y[test_y==1])
  print('>Train: 0=%d, 1=%d, Test: 0=%d, 1=%d' % (train_0, train_1, test_0, test_1))
```

```
>Train: 0=791, 1=9, Test: 0=199, 1=1
>Train: 0=793, 1=7, Test: 0=197, 1=3
>Train: 0=794, 1=6, Test: 0=196, 1=4
>Train: 0=790, 1=10, Test: 0=200, 1=0
>Train: 0=792, 1=8, Test: 0=198, 1=2
```

disini maksudnya ada kelas minoritas training data, namun di testing gak ada

karena k fold jalanin proses training sebanyak k-1 (dapetin sampel contohnya 1000 untuk ditrain)

Cross Validation for Imbalanced Dataset

A similar issue exists if we use a simple train/test split of the dataset, although the issue is less severe.

```
# example of train/test split with an imbalanced dataset
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
# generate 2 class dataset
X, y = make_classification(n_samples=1000, n_classes=2, weights=[0.99, 0.01], flip_y=0,
    random_state=1)
# split into train/test sets with same class ratio
trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.5, random_state=2)
# summarize
train_0, train_1 = len(trainy[trainy==0]), len(trainy[trainy==1])
test_0, test_1 = len(testy[testy==0]), len(testy[testy==1])
print('>Train: 0=%d, 1=%d, Test: 0=%d, 1=%d' % (train_0, train_1, test_0, test_1))
```

```
>Train: 0=497, 1=3, Test: 0=493, 1=7
```

Cross Validation for Imbalanced Dataset

- Solution:
  - Don’t not split the data randomly when using k-fold cross-validation or a train-test split.
  - Use stratified sampling or stratification.

jadi pertahankan ciri-ciri proporsi (misal 1:100, maka pertahankan di testing dataset juga)

- The target variable (y), the class, is used to control the sampling process.
  - For example, we can use a version of k-fold cross-validation that preserves the imbalanced class distribution in each fold.
  - It is called stratified k-fold cross-validation and will enforce the class distribution in each split of the data to match the distribution in the complete training dataset

Cross Validation for Imbalanced Dataset

- Stratified k-fold Cross Validation:

```
# example of stratified k-fold cross-validation with an imbalanced dataset
from sklearn.datasets import make_classification
from sklearn.model_selection import StratifiedKFold
# generate 2 class dataset
X, y = make_classification(n_samples=1000, n_classes=2, weights=[0.99, 0.01], flip_y=0,
    random_state=1)
kfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=1)
# enumerate the splits and summarize the distributions
for train_ix, test_ix in kfold.split(X, y):
  # select rows
  train_X, test_X = X[train_ix], X[test_ix]
  train_y, test_y = y[train_ix], y[test_ix]
  # summarize train and test composition
  train_0, train_1 = len(train_y[train_y==0]), len(train_y[train_y==1])
  test_0, test_1 = len(test_y[test_y==0]), len(test_y[test_y==1])
  print('>Train: 0=%d, 1=%d, Test: 0=%d, 1=%d' % (train_0, train_1, test_0, test_1))
```

```
>Train: 0=792, 1=8, Test: 0=198, 1=2
>Train: 0=792, 1=8, Test: 0=198, 1=2
>Train: 0=792, 1=8, Test: 0=198, 1=2
>Train: 0=792, 1=8, Test: 0=198, 1=2
>Train: 0=792, 1=8, Test: 0=198, 1=2
```

karena di training ada 4 fold (karena dibuat 5 n_splits = 5 fold.

Maka karena training ada 80% data dan di testing ada 20% data

Nah karena yang minoritas cuman 1:100 dari data sesunguhnya berarti 1000/100 = 10,
Lalu akan terlihat bahwa 80% dari 10 ada di training dan 20% dari 10 ada di testing

Cross Validation for Imbalanced Dataset

- We can also use a stratified version of a train/test split.
- Set the stratify argument on the call to train test split() and setting it to the y variable containing the target variable from the dataset.
- The function will determine the desired class distribution and ensure that the train and test sets both have this distribution.

```
# example of stratified train/test split with an imbalanced dataset
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
# generate 2 class dataset
X, y = make_classification(n_samples=1000, n_classes=2, weights=[0.99, 0.01], flip_y=0,
    random_state=1)
# split into train/test sets with same class ratio
trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.5, random_state=2,
    stratify=y)
# summarize
train_0, train_1 = len(trainy[trainy==0]), len(trainy[trainy==1])
test_0, test_1 = len(testy[testy==0]), len(testy[testy==1])
print('>Train: 0=%d, 1=%d, Test: 0=%d, 1=%d' % (train_0, train_1, test_0, test_1))
```

Kita harus specify?

```
>Train: 0=495, 1=5, Test: 0=495, 1=5
```

UNIVERSITAS INDONESIA

FAKULTAS ILMU KOMPUTER

Data Sampling

Balance the Class Distribution With Sampling

- Many machine learning algorithms are designed to operate on classification data with an equal number of observations for each class.
- Data sampling provides a collection of techniques that transform a training dataset in order to balance or better balance the class distribution.
- Once balanced, standard machine learning algorithms can be trained directly on the transformed dataset without any modification.
- There are many different types of data sampling methods that can be used
  - no single best method to use on all classification problems and with all classification models.
  - Like choosing a predictive model, careful experimentation is required to discover what works best for your project.

Balance the Class Distribution With Sampling

- Machine learning algorithms like the Naive Bayes Classier learn the likelihood of observing examples from each class from the training dataset. By fitting these models on a sampled training dataset with an artificially more equal class distribution, it allows them to learn a less biased prior probability and instead focus on the specifics (or evidence) from each input variable to discriminate the classes.
- Sampling is only performed on the training dataset, the dataset used by an algorithm to learn a model.
  - It is not performed on the holdout test or validation dataset.
  - The reason is that the intent is not to remove the class bias from the model, but to continue to evaluate the resulting model on data that is both real and representative of the target problem domain

Popular Data Sampling Method

Oversampling Techniques

- Random Oversampling
- Synthetic Minority Oversampling Technique (SMOTE)
- Borderline-SMOTE
- Borderline Oversampling with SVM
- Adaptive Synthetic Sampling (ADASYN)

Undersampling Techniques

- Random Undersampling
- Condensed Nearest Neighbor Rule (CNN)
- Near Miss Undersampling
- Tomek Links Undersampling
- Edited Nearest Neighbors Rule (ENN)
- One-Sided Selection (OSS)
- Neighborhood Cleaning Rule (NCR)

Combinations of Techniques

- SMOTE and Random Undersampling
- SMOTE and Tomek Links
- SMOTE and Edited Nearest Neighbors Rule

Random Sampling

- Data sampling involves creating a new transformed version of the training dataset in which the selected examples have a different class distribution.
- The simplest strategy is to choose examples for the transformed dataset randomly. There are two main approaches:
  - Random Oversampling: Randomly duplicate examples in the minority class.
  - Random Undersampling: Randomly delete examples in the majority class.
- Both approaches can be repeated until the desired class distribution is achieved in the training dataset, such as an equal split across the classes. They are referred to as naive sampling methods because they assume nothing about the data and no heuristics are used. This makes them simple to implement and fast to execute, which is desirable for very large and complex datasets.
- Both techniques can be used for two-class (binary) classification problems and multiclass classification problems

Random Oversampling

- This technique can be effective for ML algorithms that are affected by a skewed distribution and where multiple duplicate examples for a given class can influence the fit of the model. This might include algorithms that iteratively learn coefficients, like artificial neural networks that use stochastic gradient descent. It can also affect models that seek good splits of the data, such as SVM and decision trees.
- It might be useful to tune the target class distribution. In some cases, seeking a balanced distribution for a severely imbalanced dataset can cause affected algorithms to overfit the minority class, leading to increased generalization error. The effect can be better performance on the training dataset, but worse performance on the holdout or test dataset
- To gain insight into the impact of the method, it is a good idea to monitor the performance on both train and test datasets after oversampling and compare the results to the same algorithm on the original dataset.

Random Oversampling with Python

```
...
# define oversampling strategy
oversample = RandomOverSampler(sampling_strategy='minority')
```

If the majority class had 1,000 examples and the minority class had 100, this strategy would oversampling the minority class so that it has 1,000 examples

```
...
# define oversampling strategy
oversample = RandomOverSampler(sampling_strategy=0.5)
```

This would ensure that the minority class was oversampled to have half the number of examples as the majority class, for binary classification problems.

Random Oversampling with Python

```
# example of random oversampling to balance the class distribution
from collections import Counter
from sklearn.datasets import make_classification
from imblearn.over_sampling import RandomOverSampler
# define dataset
X, y = make_classification(n_samples=10000, weights=[0.99], flip_y=0)
# summarize class distribution
print(Counter(y))
# define oversampling strategy
oversample = RandomOverSampler(sampling_strategy='minority')
# fit and apply the transform
X_over, y_over = oversample.fit_resample(X, y)
# summarize class distribution
print(Counter(y_over))
```

```
Counter({0: 9900, 1: 100})
Counter({0: 9900, 1: 9900})
```

Random Oversampling with Python

This transform can be used as part of a Pipeline to ensure that it is only applied to the training dataset as part of each split in a k-fold cross-validation. A traditional scikit-learn Pipeline cannot be used; instead, a Pipeline from the imbalanced-learn library can be used.

```
# example of evaluating a decision tree with random oversampling
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.tree import DecisionTreeClassifier
from imblearn.pipeline import Pipeline
from imblearn.over_sampling import RandomOverSampler
# define dataset
X, y = make_classification(n_samples=10000, weights=[0.99], flip_y=0)
# define pipeline
steps = [('over', RandomOverSampler()), ('model', DecisionTreeClassifier())]
pipeline = Pipeline(steps=steps)
# evaluate pipeline
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
scores = cross_val_score(pipeline, X, y, scoring='f1_micro', cv=cv, n_jobs=-1)
score = mean(scores)
print('F-measure: %.3f' % score)
```

```
F-measure: 0.990
```

Random Undersampling with Python

```
# example of random undersampling to balance the class distribution
from collections import Counter
from sklearn.datasets import make_classification
from imblearn.under_sampling import RandomUnderSampler
# define dataset
X, y = make_classification(n_samples=10000, weights=[0.99], flip_y=0)
# summarize class distribution
print(Counter(y))
# define undersample strategy
undersample = RandomUnderSampler(sampling_strategy='majority')
# fit and apply the transform
X_over, y_over = undersample.fit_resample(X, y)
# summarize class distribution
print(Counter(y_over))
```

```
Counter({0: 9900, 1: 100})
Counter({0: 100, 1: 100})
```

Random Undersampling with Python

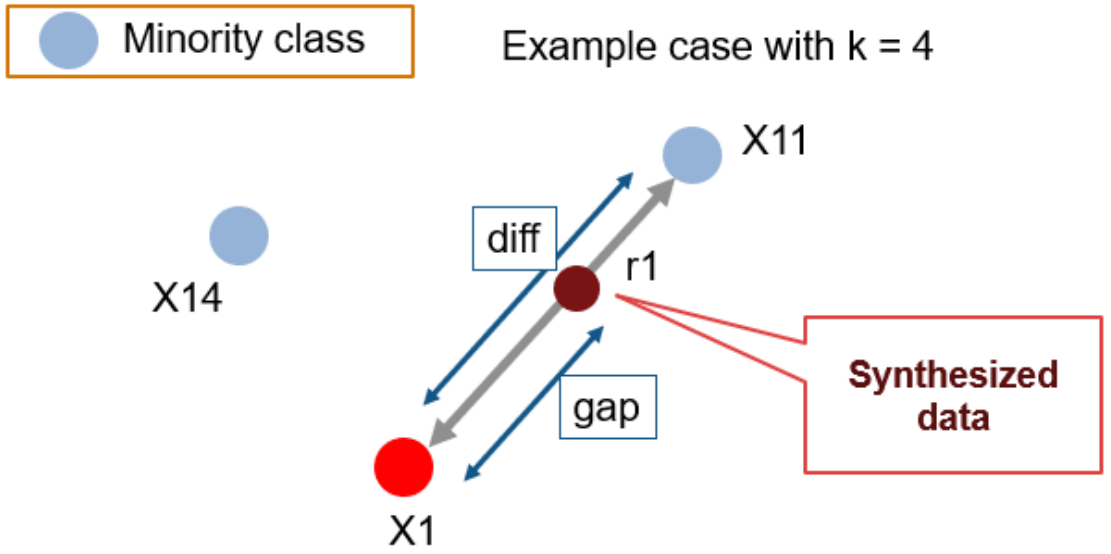
```
# example of evaluating a decision tree with random undersampling
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.tree import DecisionTreeClassifier
from imblearn.pipeline import Pipeline
from imblearn.under_sampling import RandomUnderSampler
# define dataset
X, y = make_classification(n_samples=10000, weights=[0.99], flip_y=0)
# define pipeline
steps = [('under', RandomUnderSampler()), ('model', DecisionTreeClassifier())]
pipeline = Pipeline(steps=steps)
# evaluate pipeline
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
scores = cross_val_score(pipeline, X, y, scoring='f1_micro', cv=cv, n_jobs=-1)
score = mean(scores)
print('F-measure: %.3f' % score)
```

```
F-measure: 0.889
```

Oversampling Method: SMOTE

- Duplicating examples in the minority class don't add any new information to the model.
- Instead, new examples can be synthesized from the existing examples
- This is a type of data augmentation for the minority class and is referred to as the Synthetic Minority Oversampling Technique (SMOTE)
- SMOTE first selects a minority class instance a at random and finds its k nearest minority class neighbors. The synthetic instance is then created by choosing one of the k nearest neighbors b at random and connecting a and b to form a line segment in the feature space. The synthetic instances are generated as a convex combination of the two chosen instances a and b.

SMOTE



basically

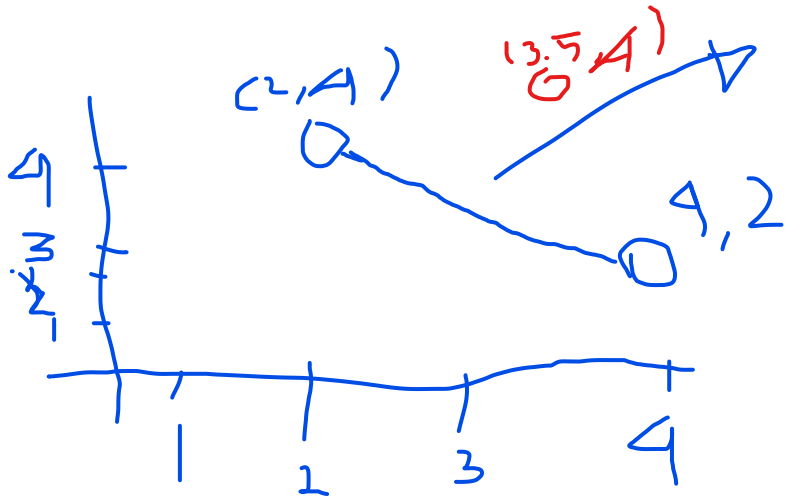
r1 itu ada di

S = a (x1) + b (x2)

dimana

a, b >= 0 v

a + b = 1 v



harus berada disini (artinya kalau 3.5, 4) gak mungkin karena jauh

Source: https://github.com/minoue-xx/Oversampling-Imbalanced-Data37

Oversampling Method: SMOTE

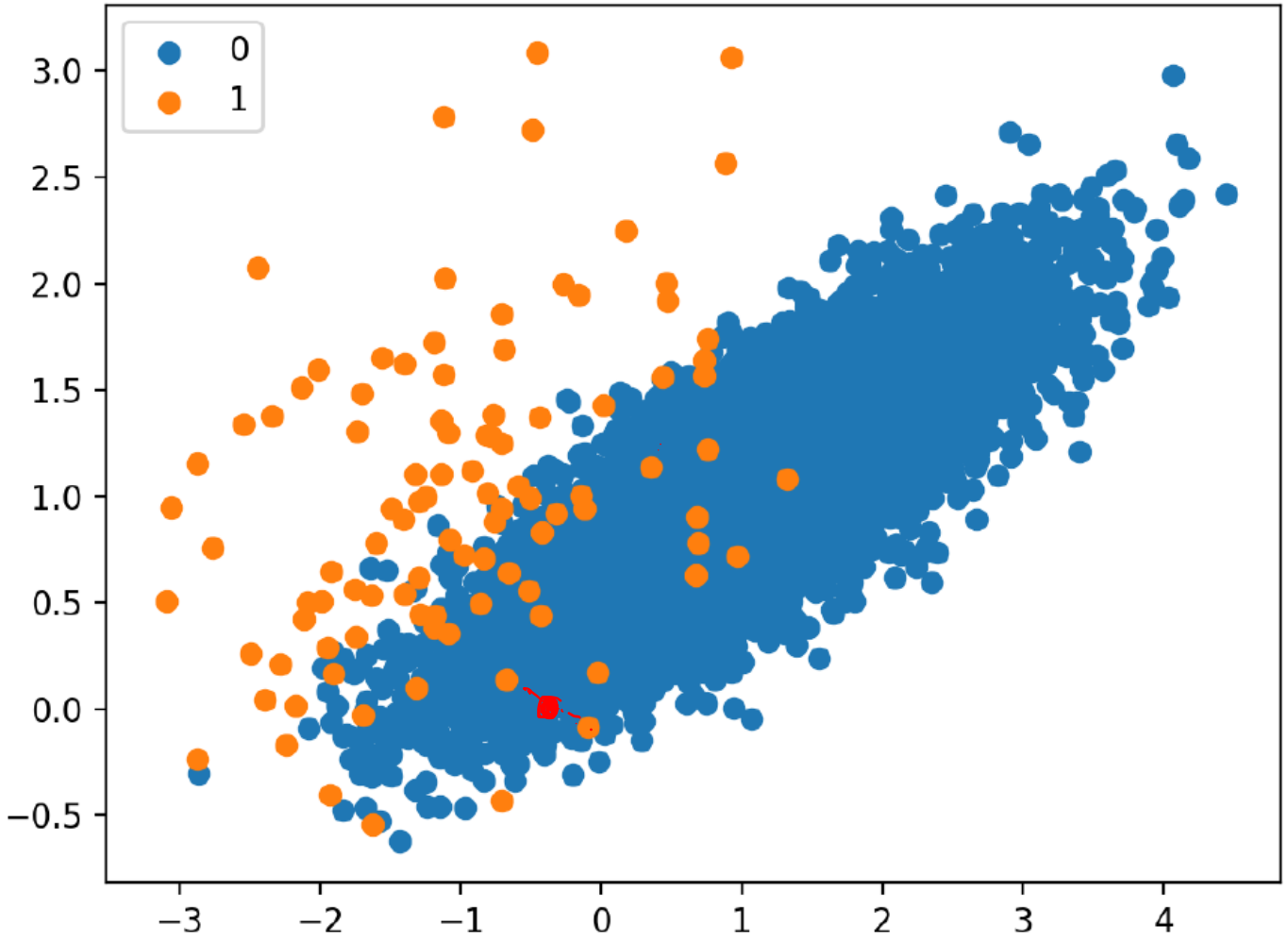
- This procedure can be used to create as many synthetic examples for the minority class as are required. It suggests first using random undersampling to trim the number of examples in the majority class, then use SMOTE to oversample the minority class to balance the class distribution.
- The approach is effective because new synthetic examples from the minority class are created that are plausible, that is, are relatively close in feature space to existing examples from the minority class.
- A general downside of the approach is that synthetic examples are created without considering the majority class, possibly resulting in ambiguous examples if there is a strong overlap for the classes.

SMOTE with Python

```
# Generate and plot a synthetic imbalanced classification dataset
from collections import Counter
from sklearn.datasets import make_classification
from matplotlib import pyplot
from numpy import where
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
  n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
# summarize class distribution
counter = Counter(y)
print(counter)
# scatter plot of examples by class label
for label, _ in counter.items():
  row_ix = where(y == label)[0]
  pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
pyplot.legend()
pyplot.show()
```

```
Counter({0: 9900, 1: 100})
```

SMOTE with Python



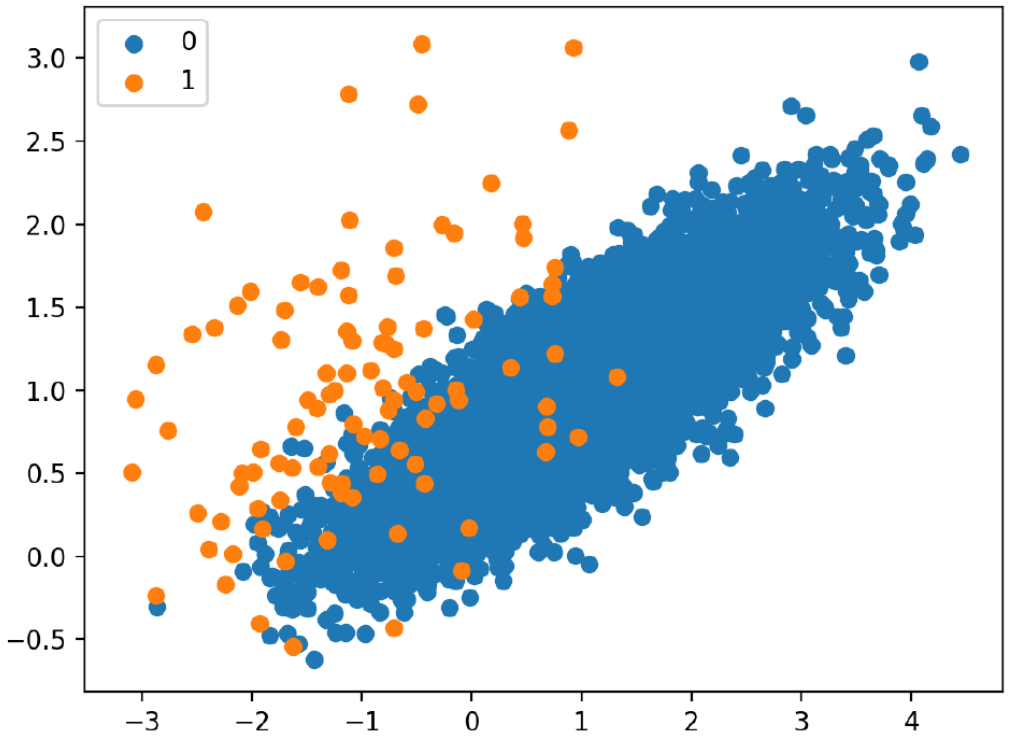
Scatter Plot of Imbalanced Binary Classification Problem

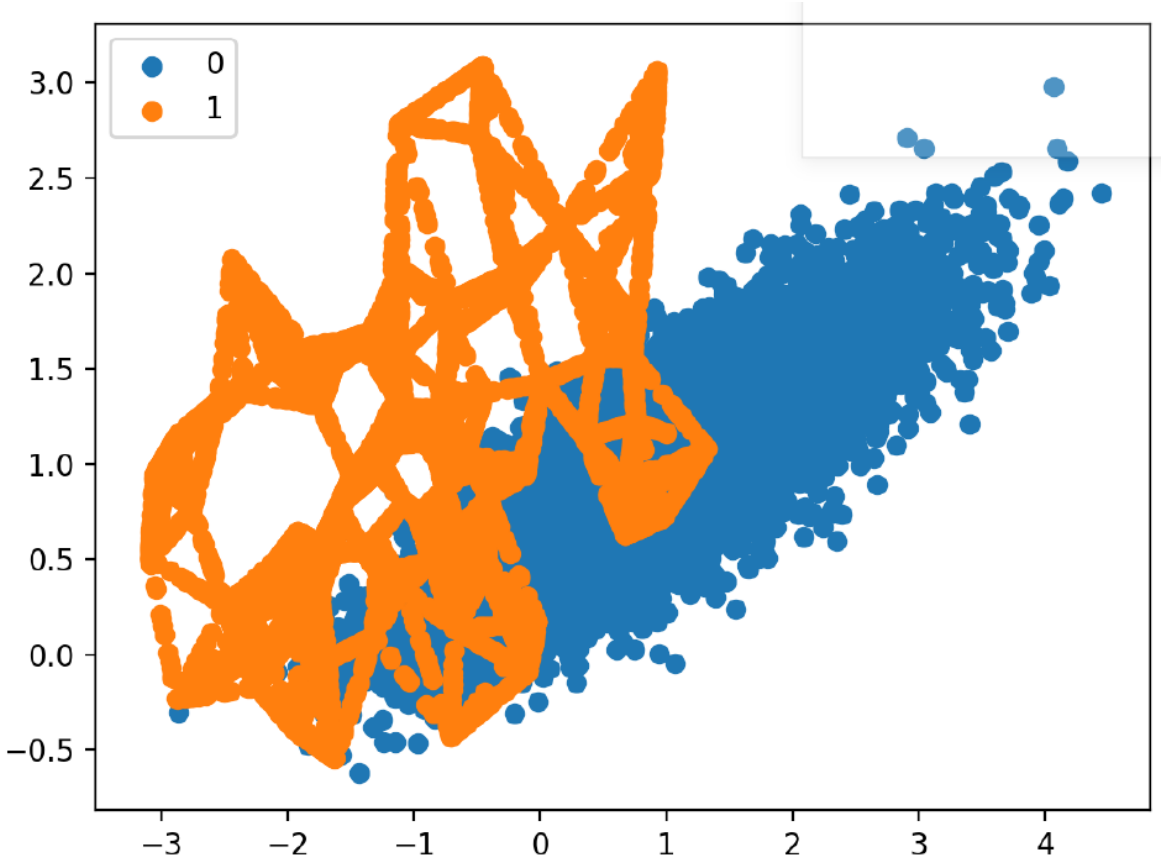
SMOTE with Python

```
# Oversample and plot imbalanced dataset with SMOTE
from collections import Counter
from sklearn.datasets import make_classification
from imblearn.over_sampling import SMOTE
from matplotlib import pyplot
from numpy import where
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
# summarize class distribution
counter = Counter(y)
print(counter)
# transform the dataset
oversample = SMOTE()
X, y = oversample.fit_resample(X, y)
# summarize the new class distribution
counter = Counter(y)
print(counter)
# scatter plot of examples by class label
for label, _ in counter.items():
  row_ix = where(y == label)[0]
  pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
pyplot.legend()
pyplot.show()
```

```
Counter({0: 9900, 1: 100})
Counter({0: 9900, 1: 9900})
```

SMOTE with Python





Scatter Plot of Imbalanced Binary Classification Problem Transformed by SMOTE.

SMOTE for Classification

```
# decision tree evaluated on imbalanced dataset
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.tree import DecisionTreeClassifier
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
# define model
model = DecisionTreeClassifier()
# evaluate pipeline
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
scores = cross_val_score(model, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
print('Mean ROC AUC: %.3f' % mean(scores))
```

```
Mean ROC AUC: 0.769
```

SMOTE for Classification

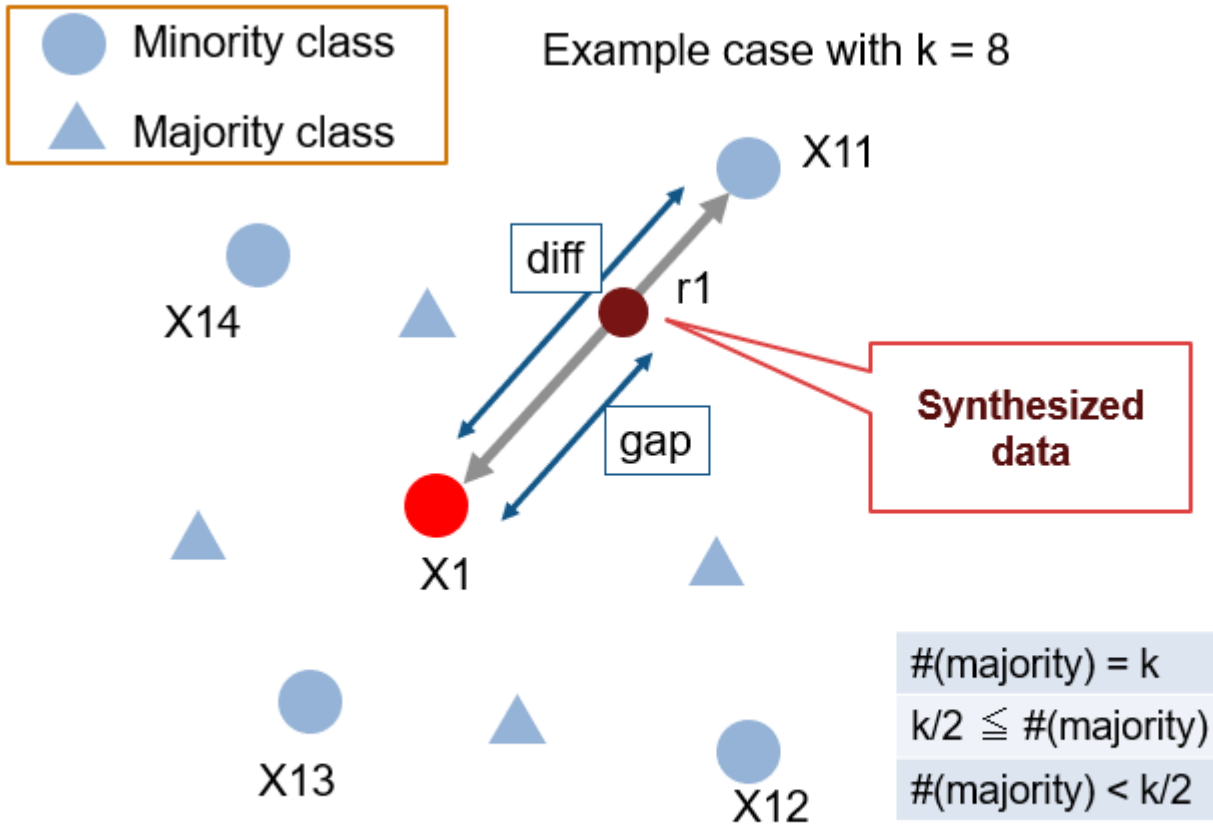
```
# decision tree evaluated on imbalanced dataset with SMOTE oversampling
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.tree import DecisionTreeClassifier
from imblearn.pipeline import Pipeline
from imblearn.over_sampling import SMOTE
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
# define pipeline
steps = [('over', SMOTE()), ('model', DecisionTreeClassifier())]
pipeline = Pipeline(steps=steps)
# evaluate pipeline
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
scores = cross_val_score(pipeline, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
print('Mean ROC AUC: %.3f' % mean(scores))
```

```
Mean ROC AUC: 0.826
```

SMOTE With Selective Sample Generation

- Borderline SMOTE
  - A popular extension to SMOTE involves selecting those instances of the minority class that are misclassified, such as with a k-nearest neighbor classification model. We can then oversample just those difficult instances, providing more resolution only where it may be required
- Borderline-SMOTE SVM
  - SVM algorithm is used instead of a KNN to identify misclassied examples on the decision boundary.
  - A SVM is used to locate the decision boundary defined by the support vectors and examples in the minority class that are close to the support vectors become the focus for generating synthetic examples.
- Adaptive Synthetic Sampling (ADASYN)
  - adaptively generating minority data samples according to their distributions: more synthetic data is generated for minority class samples that are harder to learn compared to those minority samples that are easier to learn

Borderline SMOTE



#(majority) = k	Noise
k/2 ≦ #(majority) < k	Borderline
#(majority) < k/2	Safe

Data generated from X1 only when it's within borderline

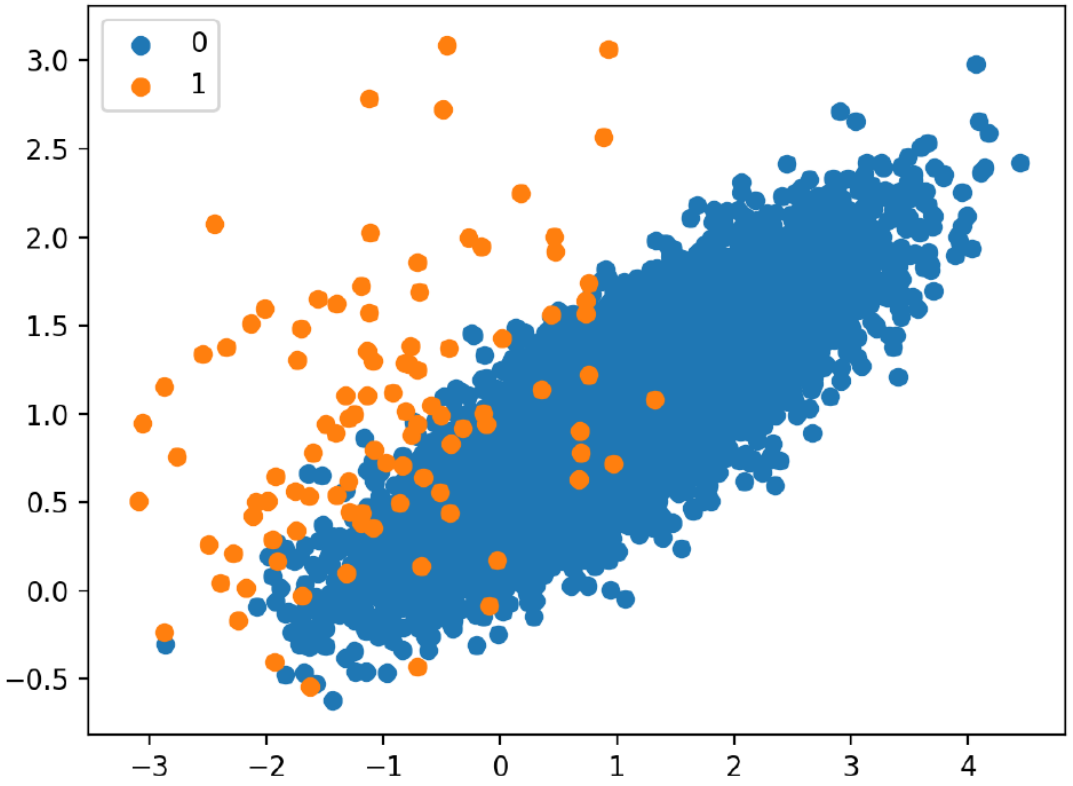
Source: https://github.com/minoue-xx/Oversampling-Imbalanced-Data

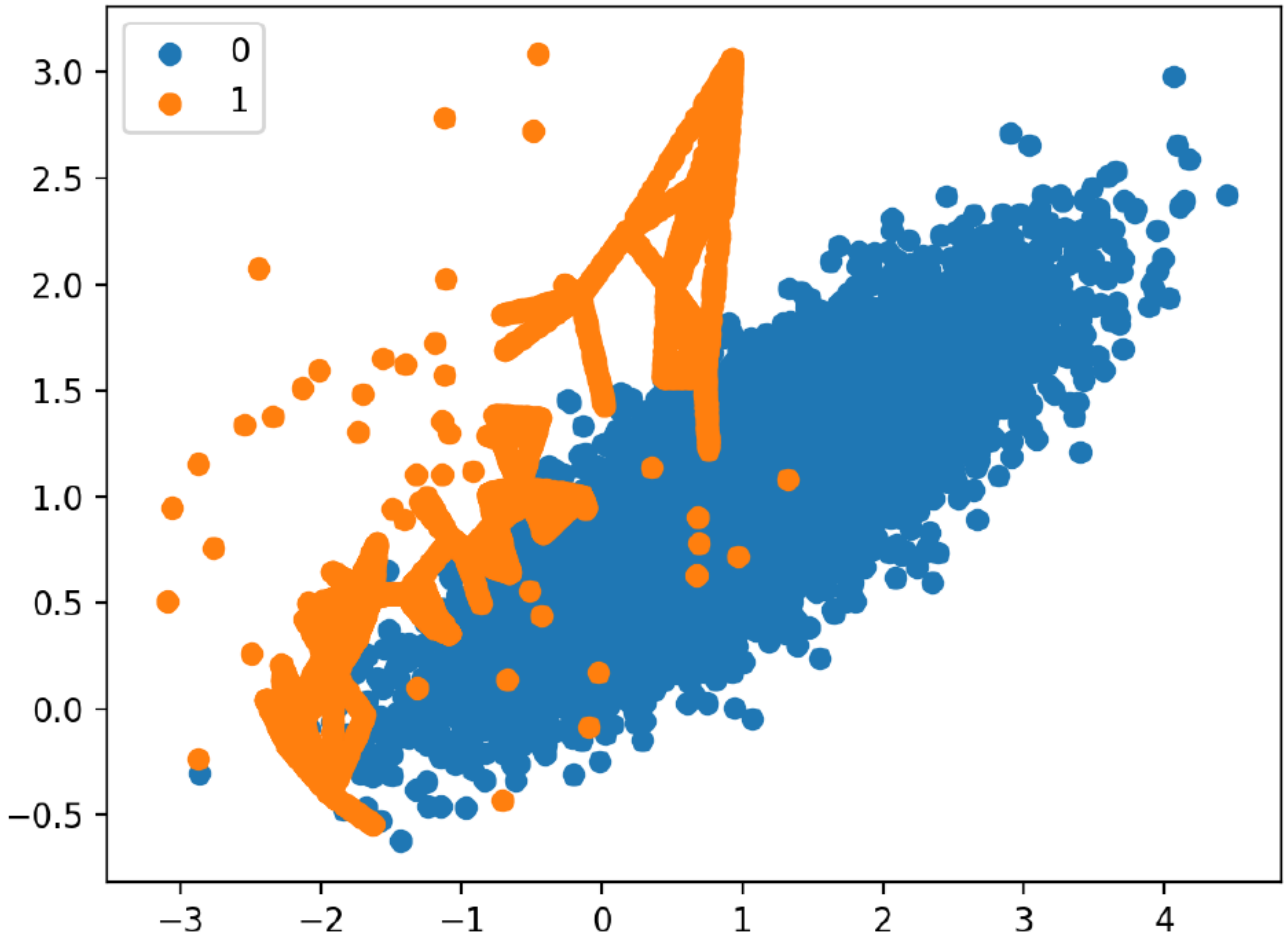
Borderline SMOTE

```
# borderline-SMOTE for imbalanced dataset
from collections import Counter
from sklearn.datasets import make_classification
from imblearn.over_sampling import BorderlineSMOTE
from matplotlib import pyplot
from numpy import where
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
# summarize class distribution
counter = Counter(y)
print(counter)
# transform the dataset
oversample = BorderlineSMOTE()
X, y = oversample.fit_resample(X, y)
# summarize the new class distribution
counter = Counter(y)
print(counter)
# scatter plot of examples by class label
for label, _ in counter.items():
  row_ix = where(y == label)[0]
  pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
pyplot.legend()
pyplot.show()
```

```
Counter({0: 9900, 1: 100})
Counter({0: 9900, 1: 9900})
```

Borderline SMOTE





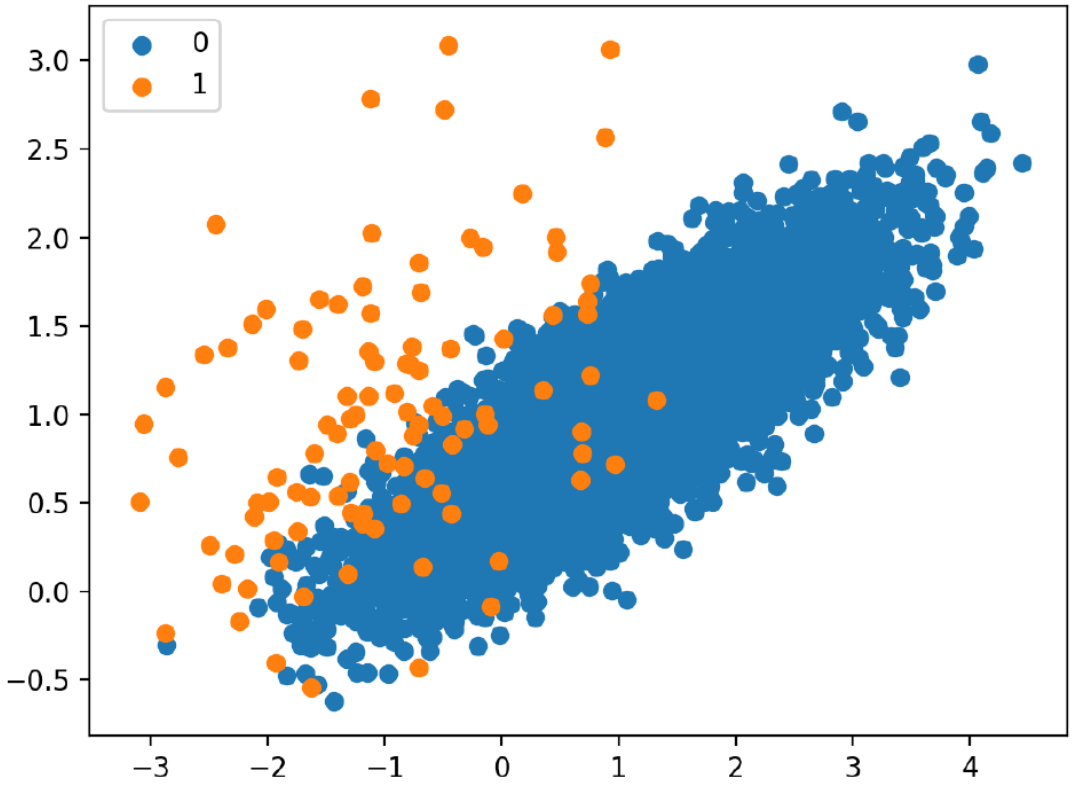
r = a X1 + b X2

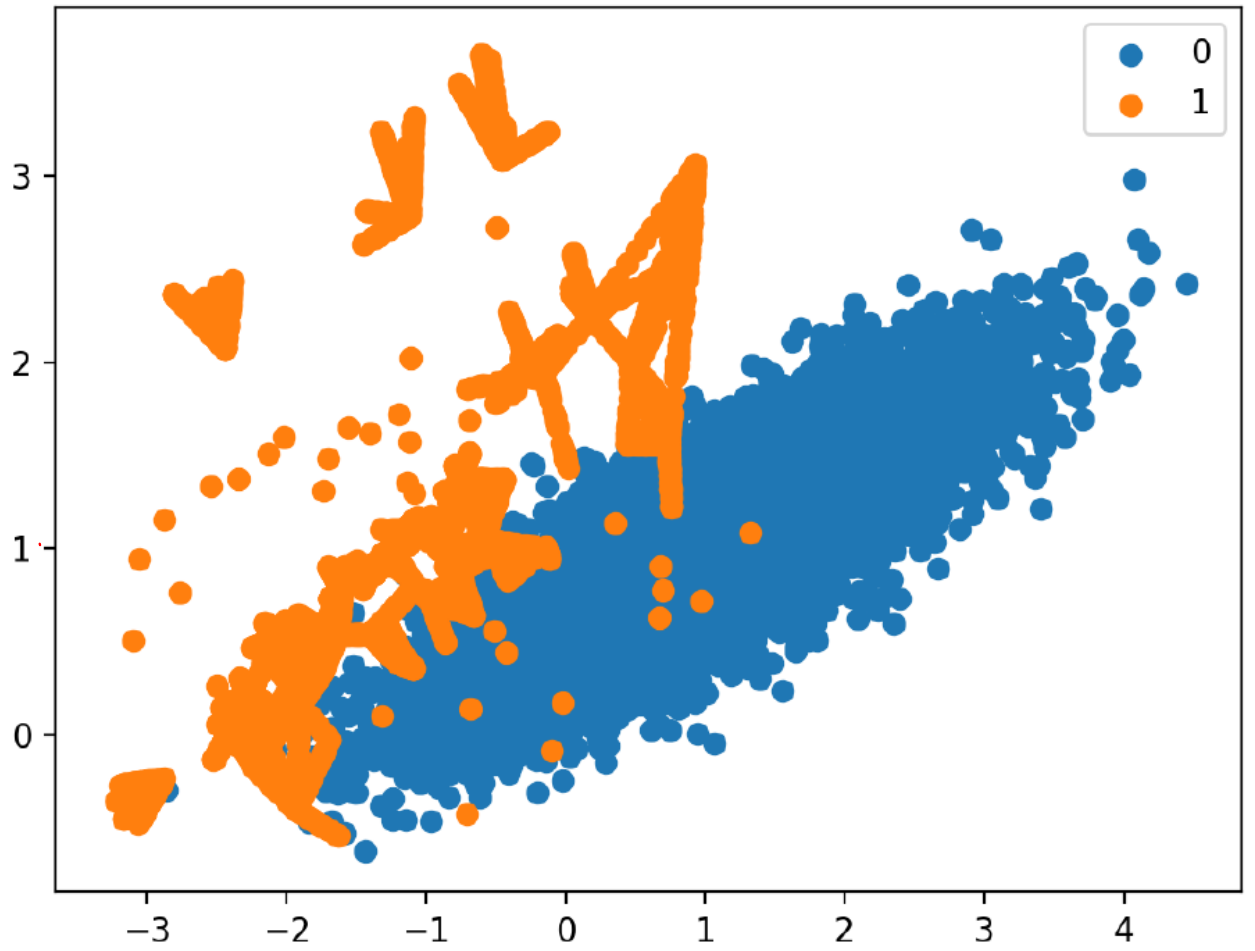
Borderline-SMOTE SVM

```
# borderline-SMOTE with SVM for imbalanced dataset
from collections import Counter
from sklearn.datasets import make_classification
from imblearn.over_sampling import SVMSMOTE
from matplotlib import pyplot
from numpy import where
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
# summarize class distribution
counter = Counter(y)
print(counter)
# transform the dataset
oversample = SVMSMOTE()
X, y = oversample.fit_resample(X, y)
# summarize the new class distribution
counter = Counter(y)
print(counter)
# scatter plot of examples by class label
for label, _ in counter.items():
  row_ix = where(y == label)[0]
  pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
pyplot.legend()
pyplot.show()
```

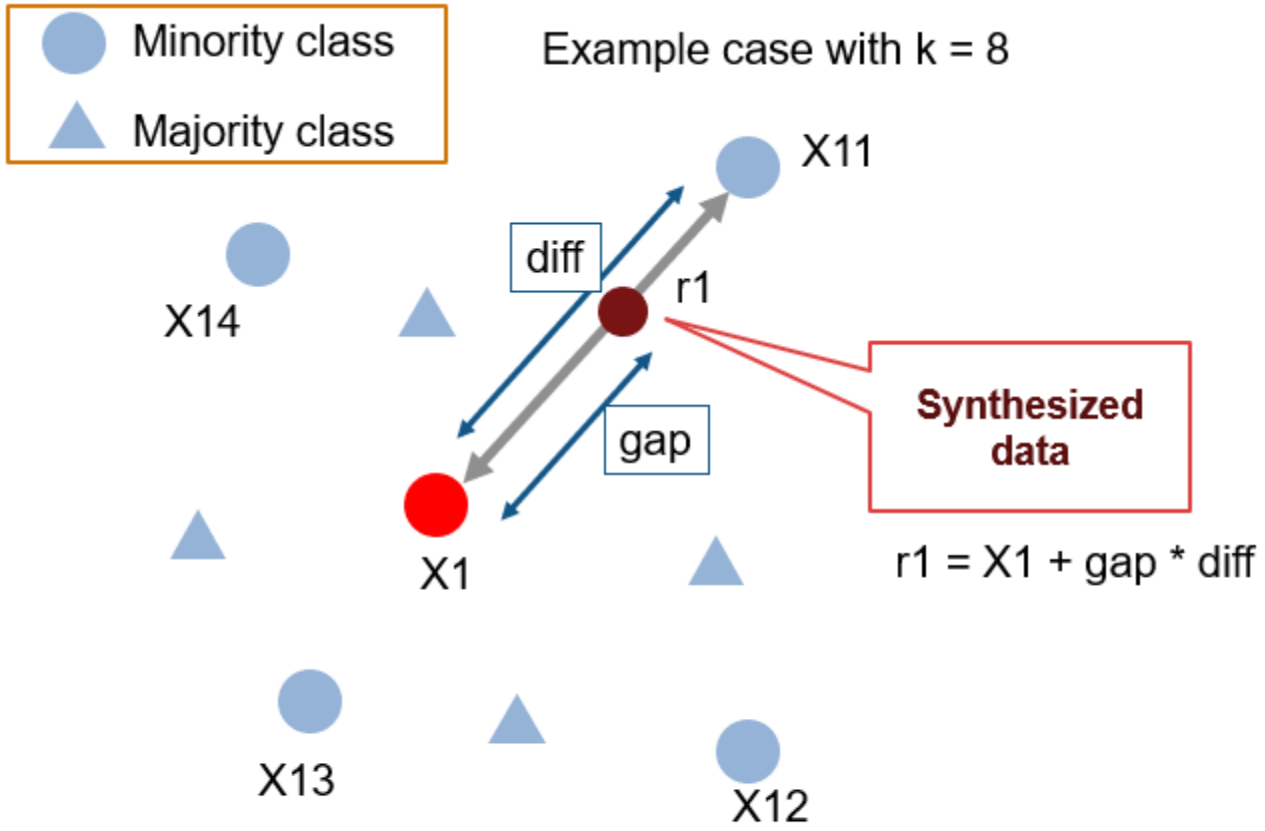
```
Counter({0: 9900, 1: 100})
Counter({0: 9900, 1: 9900})
```

Borderline-SMOTE SVM





ADASYN



Weight: w1 ∝ #(majority)/#(minority)
∝ # of synthesized data to generate

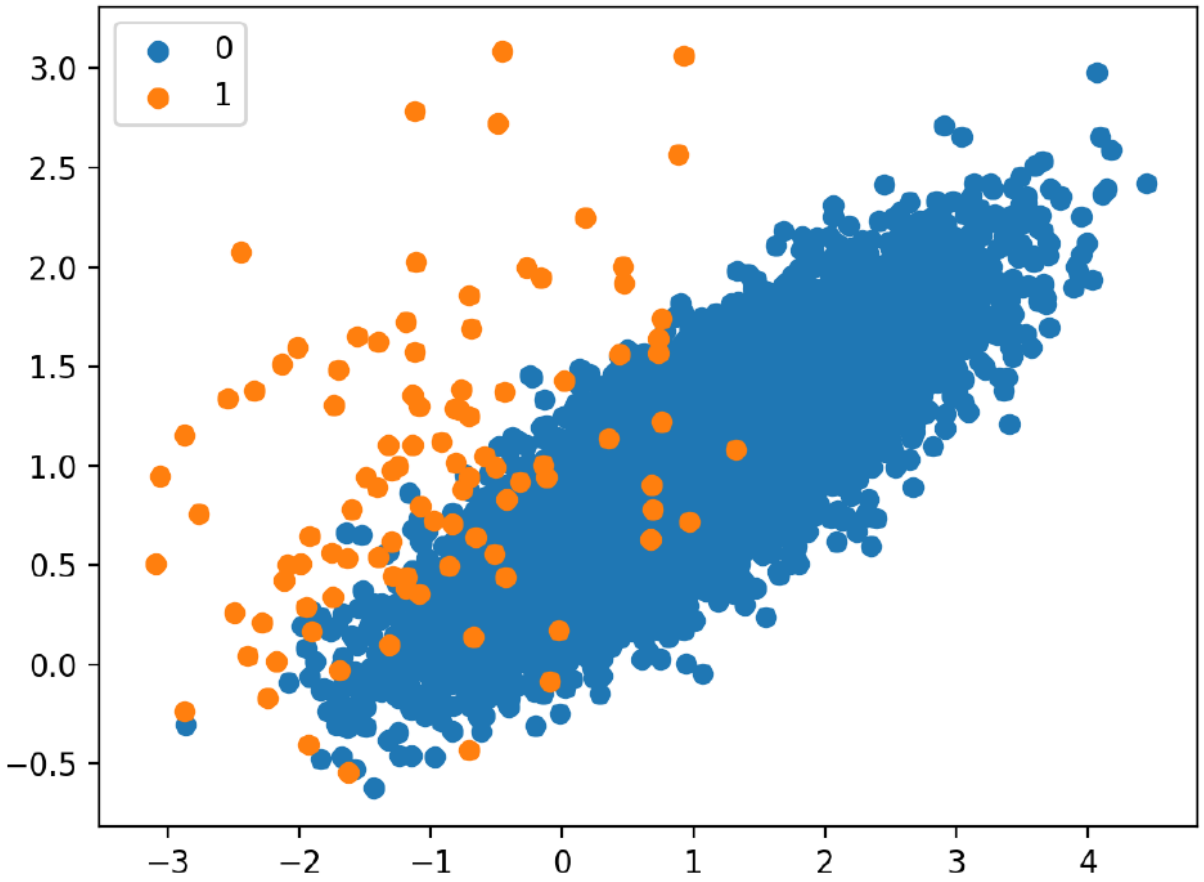
Source: https://github.com/minoue-xx/Oversampling-Imbalanced-Data

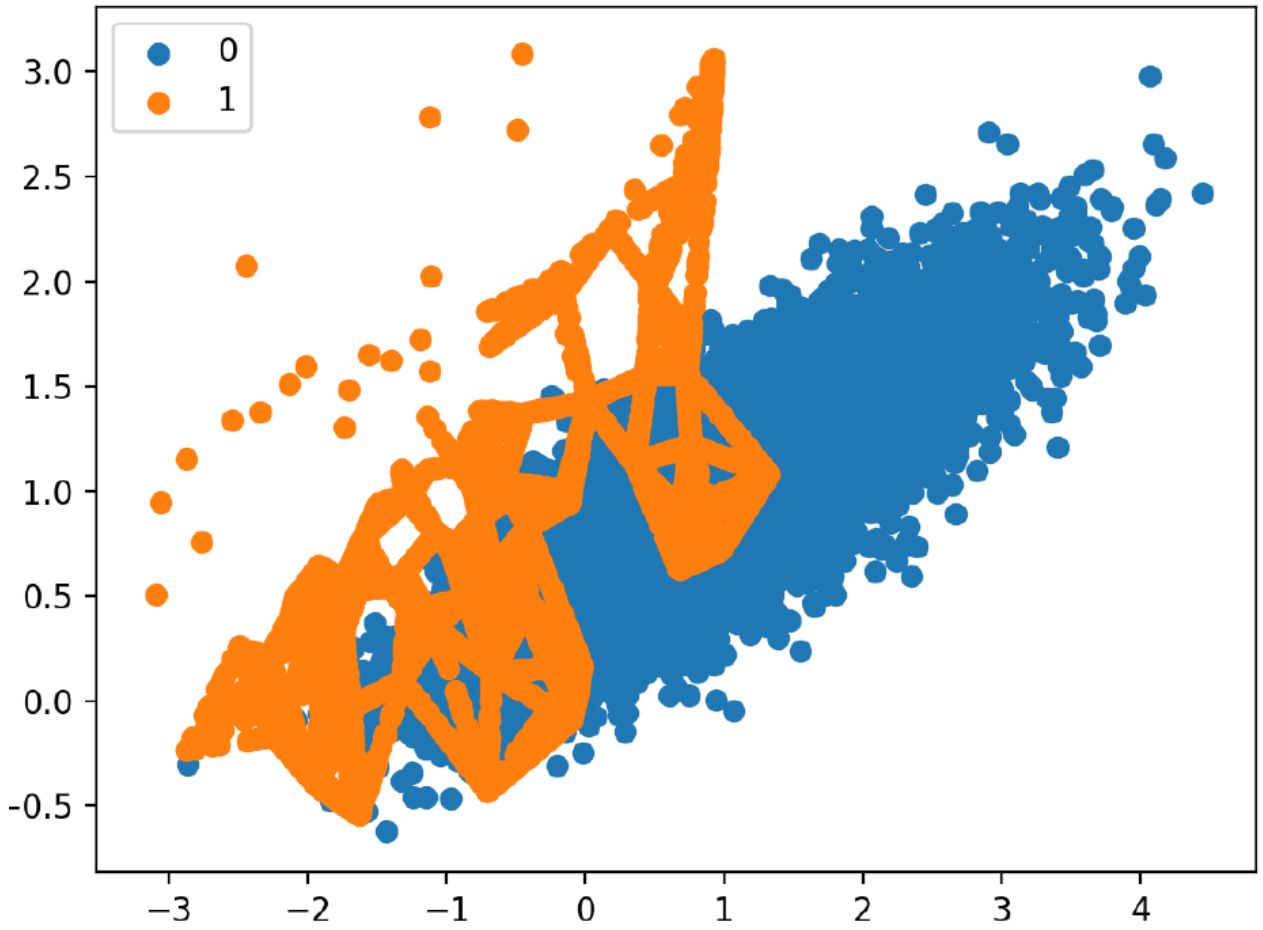
ADASYN

```
# Oversample and plot imbalanced dataset with ADASYN
from collections import Counter
from sklearn.datasets import make_classification
from imblearn.over_sampling import ADASYN
from matplotlib import pyplot
from numpy import where
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
# summarize class distribution
counter = Counter(y)
print(counter)
# transform the dataset
oversample = ADASYN()
X, y = oversample.fit_resample(X, y)
# summarize the new class distribution
counter = Counter(y)
print(counter)
# scatter plot of examples by class label
for label, _ in counter.items():
  row_ix = where(y == label)[0]
  pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
pyplot.legend()
pyplot.show()
```

```
Counter({0: 9900, 1: 100})
Counter({0: 9900, 1: 9899})
```

ADASYN





kelemahan adasyn:
- borderline smote ada kondisi dimana dia dapat mendeteksi data yang noise (#(majoirty) = k), dimana dia bisa tau kalau daerah mana yang tidak ada noise.
- Untuk adasyn, dia tidak memiliki kemampuan untuk mendeteksi data yang noise sehingga menghasilkan data noise

Jadi hilangkan dulu data minoritas yang merupakan data noisy apabila pakai adasyn (remove outlier).

ADASYN

- With Borderline-SMOTE, a discriminative model is not created. Instead, examples in the minority class are weighted according to their density, then those examples with the lowest density are the focus for the SMOTE synthetic example generation process.
- Like Borderline-SMOTE, we can see that synthetic sample generation is focused around the decision boundary as this region has the lowest density.
- Unlike Borderline-SMOTE, we can see that the examples that have the most class overlap have the most focus. On problems where these low density examples might be outliers, the ADASYN approach may put too much attention on these areas of the feature space, which may result in worse model performance.
- It may help to remove outliers prior to applying the oversampling procedure, and this might be a helpful heuristic to use more generally.

Undersampling for Imbalance Classification

- Undersampling techniques remove examples from the training dataset that belong to the majority class in order to better balance the class distribution, such as reducing the skew from a 1:100 to a 1:10, 1:2, or even a 1:1 class distribution.
- Undersampling Methods:
  - Methods that Select Examples to Keep
  - Methods that Select Examples to Delete
  - Combinations of Keep and Delete Methods

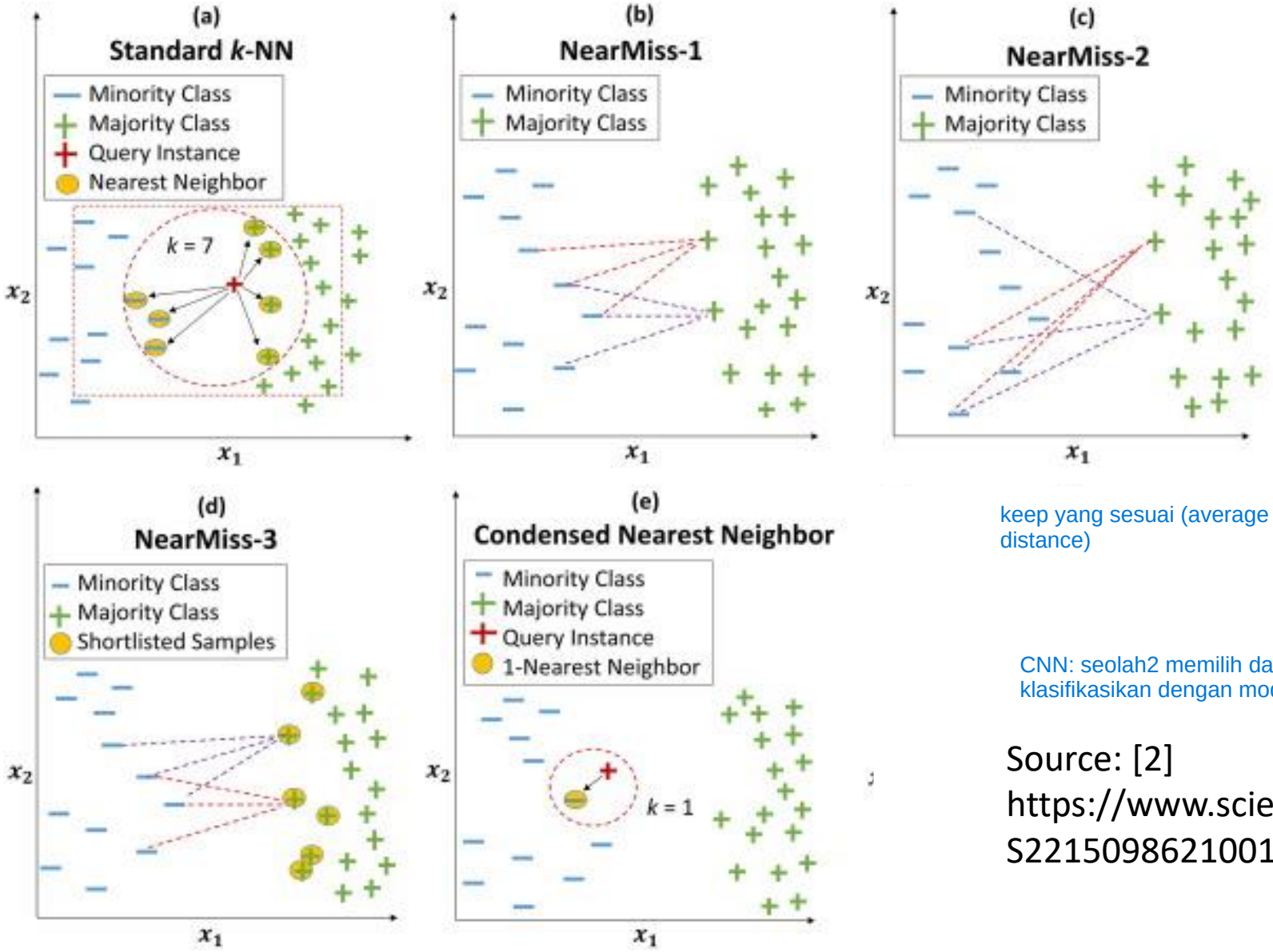
Methods that Select Examples to Keep

- Near Miss Undersampling
  - Near Miss refers to a collection of undersampling methods that select examples based on the distance of majority class examples to minority class examples.
  - NearMiss-1: Majority class examples with minimum average distance to three closest minority class examples.
  - NearMiss-2: Majority class examples with minimum average distance to three furthest minority class examples.
  - NearMiss-3: Majority class examples with minimum distance to each minority class example.
- Condensed Nearest Neighbor Rule Undersampling

Memilih data majoritas untuk disimpan

  - seeks a subset of a collection of samples that results in no loss in model performance, referred to as a minimal consistent set
  - CNN is designed for reducing the data samples towards classification using k-NN. It elects a set of prototypes p from the training samples in such a manner that k-NN with p can classify the instances exactly as k-NN does with the complete dataset [2].

Near Miss Undersampling, CNN



keep yang sesuai (average closest distance, average furthest distance, minimum distance)

CNN: seolah2 memilih data kelas mayoritas yang mungkin sulit untuk di klasifikasikan dengan model KNN

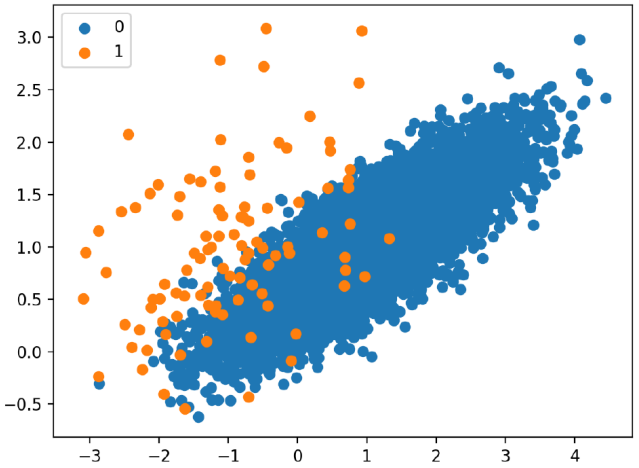
Source: [2] https://www.sciencedirect.com/science/article/pii/S2215098621001233

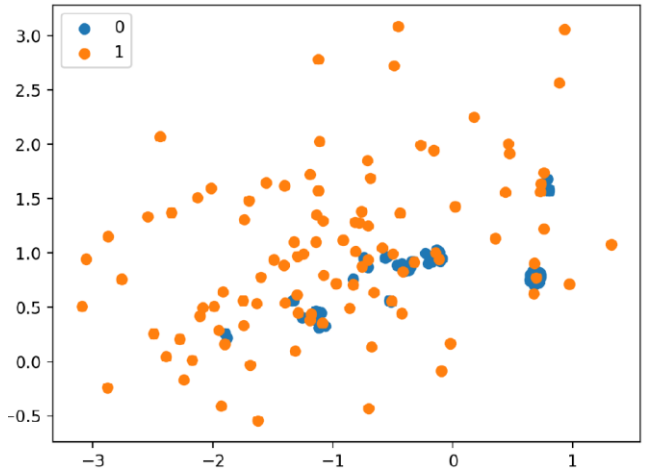
Methods that Select Examples to Keep

```
# Undersample imbalanced dataset with NearMiss-1
from collections import Counter
from sklearn.datasets import make_classification
from imblearn.under_sampling import NearMiss
from matplotlib import pyplot
from numpy import where
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
# summarize class distribution
counter = Counter(y)
print(counter)
# define the undersampling method
undersample = NearMiss(version=1, n_neighbors=3)
# transform the dataset
X, y = undersample.fit_resample(X, y)
```

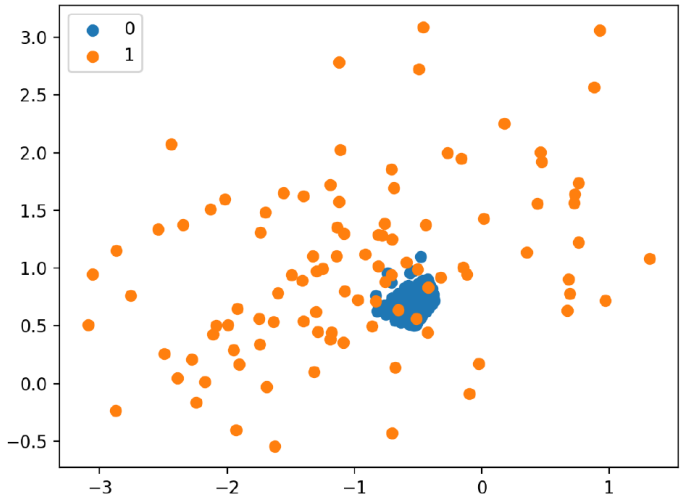
```
# summarize the new class distribution
counter = Counter(y)
print(counter)
# scatter plot of examples by class label
for label, _ in counter.items():
  row_ix = where(y == label)[0]
  pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
pyplot.legend()
pyplot.show()
```

Methods that Select Examples to Keep

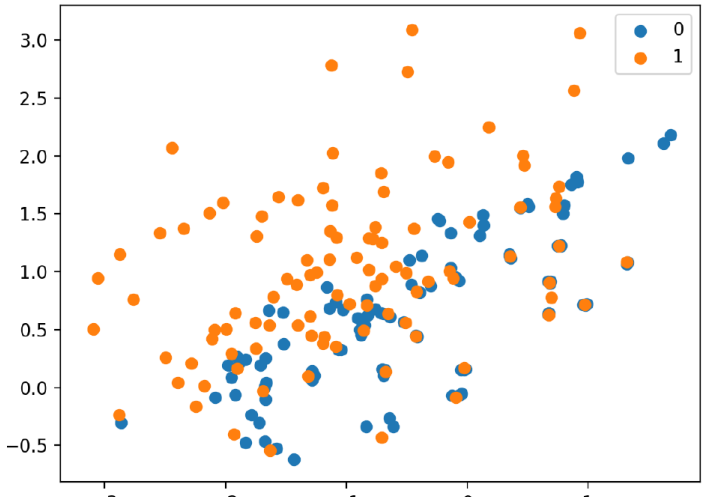




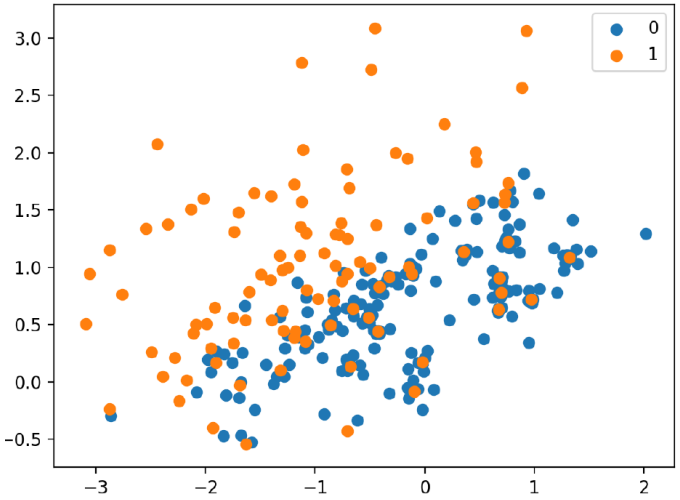
Scatter Plot of Imbalanced Dataset Undersampled with NearMiss-1.



Scatter Plot of Imbalanced Dataset Undersampled with NearMiss-2.



Scatter Plot of Imbalanced Dataset Undersampled with NearMiss-3.



Scatter Plot of Imbalanced Dataset Undersampled With the Condensed Nearest Neighbor Rule.

Methods that Select Examples to Delete

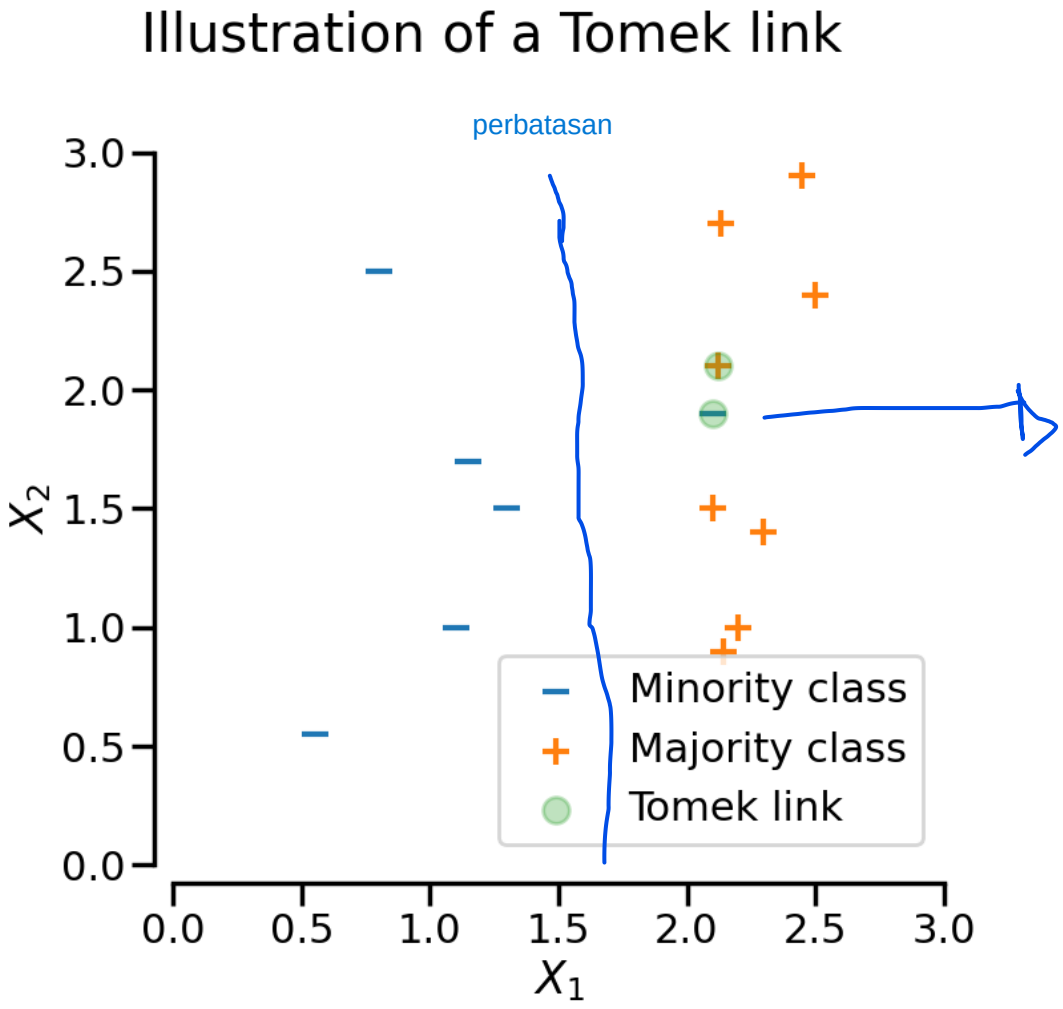
Bisa saja data yang kita keep di CNN, ternyata tidak perlu untuk di Keep.

Karena CNN ingin melakukan proses keep pada data majoritas walaupun berada dalam boundary safe.

- Tomek Link for Undersampling
  - A criticism of the Condensed Nearest Neighbor Rule is that examples are selected randomly, especially initially. This has the effect of allowing redundant examples into the store and in allowing examples that are internal to the mass of the distribution, rather than on the class boundary into the store.
  - Instances a and b define a Tomek Link if: (i) instance a's nearest neighbor is b, (ii) instance b's nearest neighbor is a, and (iii) instances a and b belong to different classes.
  - The procedure for finding Tomek Links can be used to locate all cross-class nearest neighbors.
  - If the examples in the minority class are held constant, the procedure can be used to find all of those examples in the majority class that are closest to the minority class, then removed.
  - From this definition, we see that instances that are in Tomek Links are either boundary instances or noisy instances.

Basically menghapus data2 majoritas yang memiliki tomek link dengan data minoritas

Tomek Link



nearest neighbor (2 kelas ini memiliki relasi tomek link).

Karena tanda + merupakan majoritas, maka yang + dihapus.

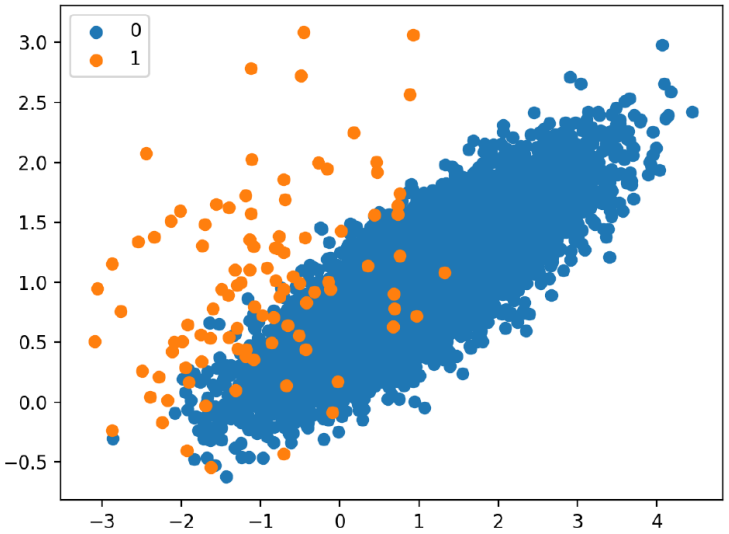
Namun tanda - aman.

https://imbalanced-learn.org/stable/auto_examples/under-sampling/plot_illustration_tomek_links.html

Methods that Select Examples to Delete

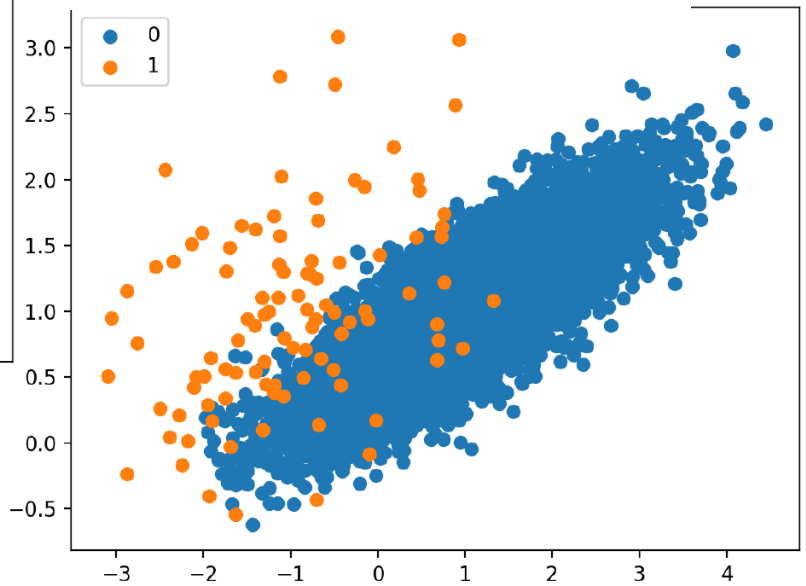
- Edited Nearest Neighbor Rule Undersampling
  - For each instance a in the dataset, its three nearest neighbors are computed. If a is a majority class instance and is misclassied by its three nearest neighbors, then a is removed from the dataset. Alternatively, if a is a minority class instance and is misclassied by its three nearest neighbors, then the majority class instances among a's neighbors are removed.

Methods that Select Examples to Delete



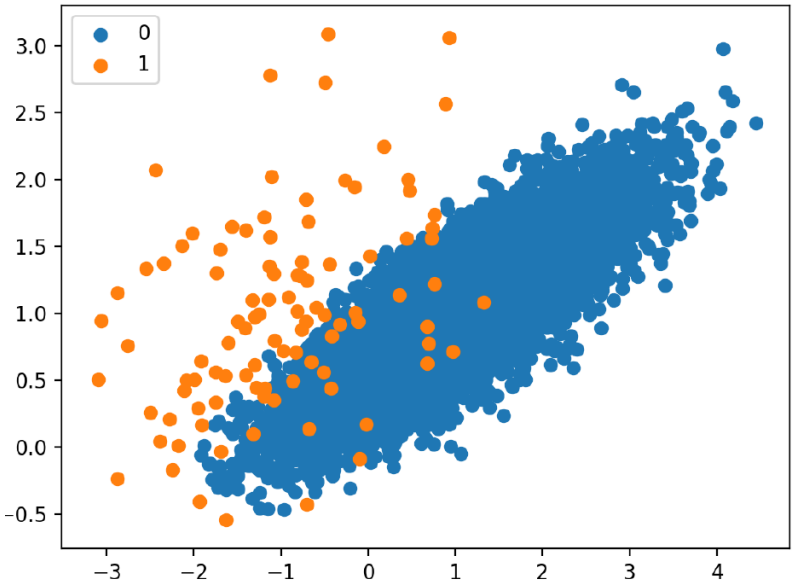

```
Counter({0: 9900, 1: 100})
Counter({0: 9874, 1: 100})
```

original vs dihapus



Scatter Plot of Imbalanced Dataset Undersampled with Tomek Links Method

```
Counter({0: 9900, 1: 100})
Counter({0: 9806, 1: 100})
```



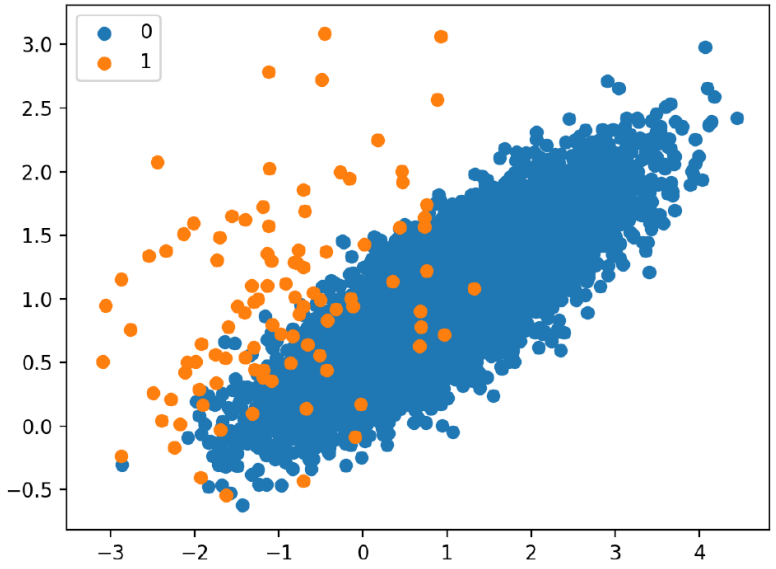
Scatter Plot of Imbalanced Dataset Undersampled with ENN Rule

- With Python, we can implement Tomek Links method for undersampling using the TomekLinks imbalanced-learn class.
- The Edited Nearest Neighbors rule can be implemented using the EditedNearestNeighbours imbalanced-learn class. The n neighbors argument controls the number of neighbors to use in the editing rule, which defaults to three.

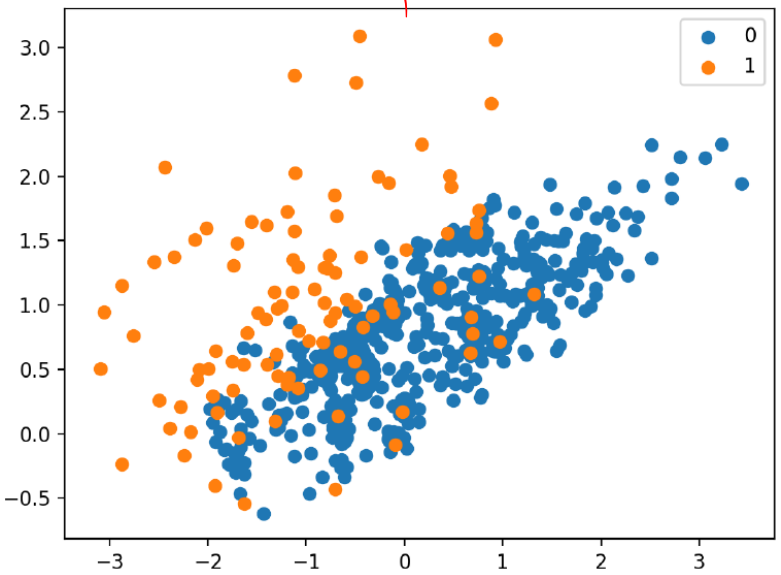
Combination of Keep & Delete Methods

- One Sided Selection (OSS) for Undersampling
  - combines Tomek Links and the Condensed Nearest Neighbor (CNN) Rule.
  - Tomek links are used as an undersampling method and removes noisy and borderline majority class examples.. The CNN method is then used to remove redundant examples from the majority class that are far from the decision boundary
  - We can implement the OSS with Python via the OneSidedSelection imbalanced-learn class.
- Neighborhood Cleaning Rule for Undersampling
  - combines both the Condensed Nearest Neighbor (CNN) Rule to remove redundant examples and the Edited Nearest Neighbors (ENN) Rule to remove noisy or ambiguous examples.
  - With Python, this technique can be implemented using the NeighbourhoodCleaningRule imbalanced-learn class.

Combination of Keep & Delete Methods



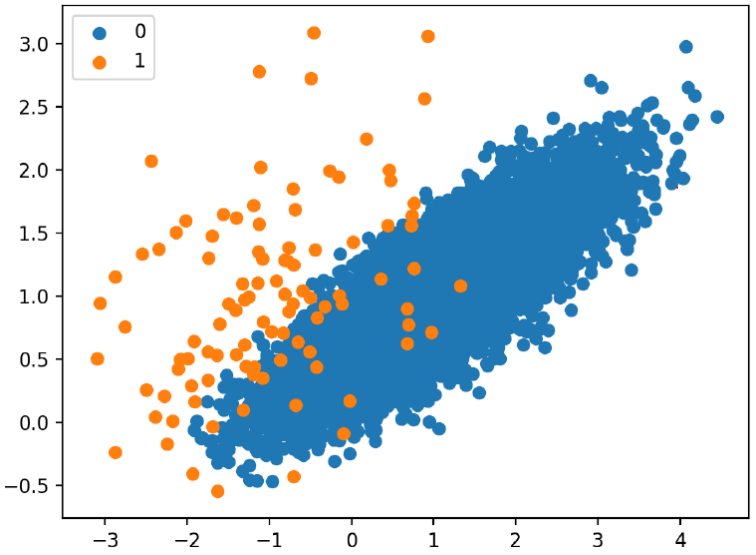

```
Counter({0: 9900, 1: 100})
Counter({0: 940, 1: 100})
```



Scatter Plot of Imbalanced Dataset Undersampled With One-Sided Selection.

CNN + Tomeklink

```
Counter({0: 9900, 1: 100})
Counter({0: 9786, 1: 100})
```



Scatter Plot of Imbalanced Dataset Undersampled With the Neighborhood Cleaning Rule.

Combination of Oversampling & Undersampling

Example: Manually Combine Data Sampling Method

```
...
# define sampling
over = ...
under = ...
# define pipeline
pipeline = Pipeline(steps=[('o', over), ('u', under)])
```

```
...
# fit and apply the pipeline
X_resampled, y_resampled = pipeline.fit_resample(X, y)
```

- This pipeline first applies an oversampling technique to a dataset, then applies undersampling to the output of the oversampling transform before returning the final outcome.
- It allows transforms to be stacked or applied in sequence on a dataset. The pipeline can then be used to transform a dataset

Combination of Oversampling & Undersampling

Alternately, a model can be added as the last step in the pipeline. This allows the pipeline to be treated as a model. When it is t on a training dataset, the transforms are first applied to the training dataset, then the transformed dataset is provided to the model so that it can develop a t.

```
...
# define model
model = ...
# define sampling
over = ...
under = ...
# define pipeline
pipeline = Pipeline(steps=[('o', over), ('u', under), ('m', model)])
```

Recall that the sampling is only applied to the training dataset, not the test dataset. When used in k-fold cross-validation, the entire sequence of transforms and fit is applied on each training dataset comprised of cross-validation folds. This is important as both the transforms and t are performed without knowledge of the holdout set

```
...
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
scores = cross_val_score(pipeline, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
```

Standard Combined Data Sampling Methods

- SMOTE and Tomek Links Undersampling

```
# combined SMOTE and Tomek Links sampling for imbalanced classification
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from imblearn.pipeline import Pipeline
from sklearn.tree import DecisionTreeClassifier
from imblearn.combine import SMOTETomek
from imblearn.under_sampling import TomekLinks
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
# define model
model = DecisionTreeClassifier()
# define sampling
resample = SMOTETomek(tomek=TomekLinks(sampling_strategy='majority'))
# define pipeline
pipeline = Pipeline(steps=[('r', resample), ('m', model)])
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
scores = cross_val_score(pipeline, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
# summarize performance
print('Mean ROC AUC: %.3f' % mean(scores))
```

```
Mean ROC AUC: 0.815
```

Standard Combined Data Sampling Methods

- SMOTE and ENN

```
# combined SMOTE and Edited Nearest Neighbors sampling for imbalanced classification
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from imblearn.pipeline import Pipeline
from sklearn.tree import DecisionTreeClassifier
from imblearn.combine import SMOTEENN
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
# define model
model = DecisionTreeClassifier()
# define sampling
resample = SMOTEENN()
# define pipeline
pipeline = Pipeline(steps=[('r', resample), ('m', model)])
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
scores = cross_val_score(pipeline, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
# summarize performance
print('Mean ROC AUC: %.3f' % mean(scores))
```

```
Mean ROC AUC: 0.856
```

UNIVERSITAS INDONESIA

FAKULTAS ILMU KOMPUTER

Cost Sensitive Learning

Not All Classification Errors Are Equal

- Real-world imbalanced binary classification problems typically have a different interpretation for each of the classification errors that can be made.
- Example: Bank Loan Problem
  - Consider a problem where a bank wants to determine whether to give a loan to a customer or not. Denying a loan to a good customer is not as bad as giving a loan to a bad customer that may never repay it.
- Example: Fraud Detection Problem
  - Consider the problem of an insurance company wants to determine whether a claim is fraudulent. Identifying good claims as fraudulent and following up with the customer is better than honoring fraudulent insurance claims.

Cost-Sensitive Learning

- Cost-sensitive learning is a subfield of machine learning that takes the costs of prediction errors (and potentially other costs) into account when training a machine learning model.
- It is a field of study that is closely related to the field of imbalanced learning that is concerned with classification on datasets with a skewed class distribution.
- Traditionally, machine learning algorithms are trained on a dataset and seek to minimize error.
- In cost-sensitive learning, a penalty associated with an incorrect prediction and is referred to as a cost. We could alternately refer to the inverse of the penalty as the benefit, although this framing is rarely used.
- The goal of cost-sensitive learning is to minimize the cost of a model on the training dataset, where it is assumed that different types of prediction errors have a different and known associated cost

Cost Sensitive Imbalanced Classification

- We might assign no cost to correct predictions in each class, a cost for False Positives and a cost for False Negatives.
- Example: Binary Cost Matrix

error false negative lebih parah karena kita lebih tidak mau misalnya fraud tapi malah dianggap bukan fraud.

	Predicted Positive	Predicted Negative
Actual Positive	0	88
Actual Negative	5	0

88 > 5

- The values of the cost matrix must be carefully defined. Like the choice of error function for traditional machine learning models, the choice of costs or cost function will determine the quality and utility of the model that is t on the training data.

Cost Sensitive Imbalanced Classification

- In some problem domains, defining the cost matrix might be obvious. In an insurance claim example, the costs for a false positive might be the monetary cost of follow-up with the customer to the company and the cost of a false negative might be the cost of the insurance claim.
- In other domains, defining the cost matrix might be challenging. For example, in a cancer diagnostic test example, the cost of a false positive might be the monetary cost of performing subsequent tests, whereas what is the equivalent dollar cost for letting a sick patient go home and get sicker? A cost matrix might be able to be defined by a domain expert or economist in such cases, or not.

Cost Sensitive Imbalanced Classification

- Further, the cost might be a complex multi-dimensional function, including monetary costs, reputation costs, and more. A good starting point for imbalanced classification tasks is to assign costs based on the inverse class distribution.
- For example, we may have a dataset with a 1 to 100 (1:100) ratio. This ratio can be inverted and used as the cost of misclassification errors, where the cost of a False Negative is 100 and the cost of a False Positive is 1.

	Predicted Positive	Predicted Negative
Actual Positive	0	100
Actual Negative	1	0

Cost-Sensitive Method

Cost-Sensitive Method that are most relevant for imbalanced learning:

1. Cost-Sensitive Resampling
   - Instead of sampling with a focus on balancing the skewed class distribution, the focus is on changing the composition of the training dataset to meet the expectations of the cost matrix.
   - This might involve directly sampling the data distribution or using a method to weight examples in the dataset. Such methods may be referred to as cost- proportionate weighing of the training dataset or cost-proportionate sampling.
2. Cost-Sensitive Algorithms
   - Many such algorithm-specific modification/augmentations have been proposed for popular algorithms, like decision trees and SVM.
3. Cost-Sensitive Ensembles

Cost Sensitive Logistic Regression

Standard Logistic Regression on imbalanced classification problem

```
# fit a logistic regression model on an imbalanced classification dataset
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.linear_model import LogisticRegression
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=2)
# define model
model = LogisticRegression(solver='lbfgs')
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
scores = cross_val_score(model, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
# summarize performance
print('Mean ROC AUC: %.3f' % mean(scores))
```

```
Mean ROC AUC: 0.985
```

Cost Sensitive Logistic Regression

- The coefficients of the logistic regression algorithm are fit using an optimization algorithm that minimizes the negative log likelihood (loss) for the model on the training dataset

$$\min \sum_{i=1}^{n} -(\log(yhat_i) \times y_i + \log(1 - yhat_i) \times (1 - y_i))$$

- This involves the repeated use of the model to make predictions followed by an adaptation of the coefficients in a direction that reduces the loss of the model. The calculation of the loss for a given set of coefficients can be modified to take the class balance into account. By default, the errors for each class may be considered to have the same weighting, say 1.0. These weightings can be adjusted based on the importance of each class.

$$\min \sum_{i=1}^{n} -(w0 \times \log(yhat_i) \times y_i + w1 \times \log(1 - yhat_i) \times (1 - y_i))$$

Cost Sensitive Logistic Regression

- The weighting is applied to the loss so that smaller weight values result in a smaller error value, and in turn, less update to the model coefficients.
  - Small Weight: Less importance, less update to the model coefficients.
  - Large Weight: More importance, more update to the model coefficients.
- As such, the modified version of logistic regression is referred to as Weighted Logistic Regression, Class-Weighted Logistic Regression or Cost-Sensitive Logistic Regression.

Cost Sensitive Logistic Regression

Example: evaluating a class-weighted logistic regression algorithm on the imbalanced classification dataset.

```
# weighted logistic regression model on an imbalanced classification dataset
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.linear_model import LogisticRegression
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=2)
# define model
weights = {0:0.01, 1:1.0}
model = LogisticRegression(solver='lbfgs', class_weight=weights)
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
scores = cross_val_score(model, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
# summarize performance
print('Mean ROC AUC: %.3f' % mean(scores))
```

```
Mean ROC AUC: 0.989
```

Cost Sensitive Logistic Regression

- The scikit-learn library provides an implementation of the best practice heuristic for the class weighting. It is implemented via the compute class weight() function and is calculated as:

$$\frac{\text{n\_samples}}{\text{n\_classes} \times \text{n\_samples\_with\_class}}$$

- We can test this calculation manually on our dataset. For example, we have 10,000 examples in the dataset, 9900 in class 0, and 100 in class 1. The weighting for class 0 is calculated as:

$$\begin{aligned}\text{weighting} &= \frac{\text{n\_samples}}{\text{n\_classes} \times \text{n\_samples\_with\_class}} \\ &= \frac{10000}{2 \times 9900} \\ &= \frac{10000}{19800}\end{aligned}$$

Cost Sensitive Logistic Regression

- The weighting for class 1 is calculated as:

$$\begin{aligned}\text{weighting} &= \frac{\text{n\_samples}}{\text{n\_classes} \times \text{n\_samples\_with\_class}} \\ &= \frac{10000}{2 \times 100} \\ &= \frac{10000}{200} \\ &= 50\end{aligned}$$

- We can confirm these calculations by calling the compute class weight() function and specifying the class weight as `balanced'.

```
# calculate heuristic class weighting
from sklearn.utils.class_weight import compute_class_weight
from sklearn.datasets import make_classification
# generate 2 class dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=2)
# calculate class weighting
weighting = compute_class_weight('balanced', [0,1], y)
print(weighting)
```

```
[ 0.50505051 50. ]
```

Cost Sensitive SVM

- SVMs are effective models for binary classification tasks, although by default, they are not effective at imbalanced classification.

```
# fit a svm on an imbalanced classification dataset
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.svm import SVC
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=4)
# define model
model = SVC(gamma='scale')
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
scores = cross_val_score(model, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
# summarize performance
print('Mean ROC AUC: %.3f' % mean(scores))
```

```
Mean ROC AUC: 0.804
```

Cost Sensitive SVM

- Perhaps the simplest and most common extension to SVM for imbalanced classification is to weight the *C* value in proportion to the importance of each class.
  - Small Weight: Smaller C value, larger penalty for misclassified examples.
  - Larger Weight: Larger C value, smaller penalty for misclassified examples.
- A best practice for using the class weighting is to use the inverse of the class distribution present in the training dataset. For example, the class distribution of the test dataset is a 1:100 ratio for the minority class to the majority class. The inverse of this ratio could be used with 1 for the majority class and 100 for the minority class; for example:

```
# define model
weights = {0:1.0, 1:100.0}
model = SVC(gamma='scale', class_weight=weights)
```

- This heuristic is available directly by setting the class weight to `balanced'.

```
...
# define model
model = SVC(gamma='scale', class_weight='balanced')
```

Cost Sensitive SVM

Example of evaluating a class weighted SVM on the imbalanced classification

```
# svm with class weight on an imbalanced classification dataset
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.svm import SVC
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=4)
# define model
model = SVC(gamma='scale', class_weight='balanced')
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
scores = cross_val_score(model, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
# summarize performance
print('Mean ROC AUC: %.3f' % mean(scores))
```

```
Mean ROC AUC: 0.964
```

Cost Sensitive SVM

- We can Grid Search a range of different class weightings for weighted SVM and discover which results in the best ROC AUC score.

```
...
# define grid
balance = [{0:100,1:1}, {0:10,1:1}, {0:1,1:1}, {0:1,1:10}, {0:1,1:100}]
param_grid = dict(class_weight=balance)
```

```
...
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# define grid search
grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1, cv=cv,
    scoring='roc_auc')
```

```
# report the best configuration
print('Best: %f using %s' % (grid_result.best_score_, grid_result.best_params_))
# report all configurations
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print('%f (%f) with: %r' % (mean, stdev, param))
```

Cost Sensitive Decision Tree

- The scikit-learn Python machine learning library provides examples of these cost-sensitive extensions via the `class_weight` argument on `DesicionTreeClassifier`.

- You can explore by yourself

UNIVERSITAS INDONESIA

FAKULTAS ILMU KOMPUTER

Latihan

Latihan 1

- Diberikan dataset berukuran 10.000 yang terdiri dari 2 kelas, dengan kelas A sebanyak 90% dan kelas B sebanyak 10%. Jika Anda melakukan **Stratified 5-fold Cross Validation**, berapa ukuran data training dan data testing pada setiap fold (percobaan)?

Latihan 2

Diberikan $x_1(2,4)$ dan $x_2(4,2)$ yang merupakan yang merupakan dua instances yang bertetangga dekat pada kelas minoritas. Dari data point di bawah ini, mana yang dapat merupakan data sintesis yang dihasilkan SMOTE dari x_1 dan x_2:

a) $s_1(2.5, 2)$

b) $s_2(2.5, 3.5)$

c) $s_3(3.5, 4)$

d) $s_3(4, 3.5)$

Latihan 3

- Diberikan dataset berukuran 100.000 samples dengan rincian:
  - 60.000 samples merupakan kelas A,
  - 30.000 samples merupakan kelas B,
  - 10.000 samples merupakan kelas C

Dalam konteks cost-sensitive learning, library scikit-learn mengimplementasikan heuristic pada fungsi compute_class_weight() untuk mengatur bobot cost tiap-tiap kelas. Berapa bobot dari masing-masing kelas untuk dataset tersebut?

Latihan 4

Anda diminta melakukan klasifikasi pada imbalanced dataset berukuran 10.000 yang terdiri dari 2 kelas: kelas A sebanyak 99% dan kelas B sebanyak 1%. Berikut potongan kode Python untuk menggenerate dataset ini.

```
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0, n
_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
```

Ada 3 model yang perlu Anda buat, yang terdiri dari:

- Model logistic regression standar
- Model logistic regression dengan teknik oversampling SMOTE
- Model cost sensitive logistic regression, jika diberikan informasi bahwa bahwa cost untuk kasus false negative adalah 50 x cost untuk false positive.

Lakukan analisis ketiga model yang Anda buat, dengan membandingkan nilai ROC AUC-nya.

UNIVERSITAS INDONESIA

FAKULTAS ILMU KOMPUTER

Wish you success ☺