Module-2 Statistical Foundations

Dealing with Imbalanced data-set

Majority
0 → 9900

Minority
1 → 100

Under-sampling

↓ Majority $\xrightarrow{map}$ Minority

Over-sampling

Majority $\xleftarrow{map}$ Minority ↑

Scaling (↓) Majority to Minority ⇒ 0 → 200 1 → 100 } Under Sampling

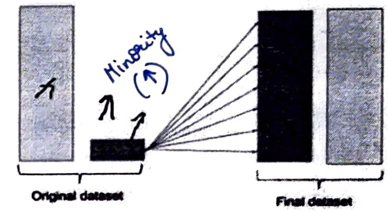
Scaling Minority (↑) to Majority ⇒ 0 → 9900 1 → 4950 } Over Sampling

(Don't)
Majority ⇒ Over-sampling

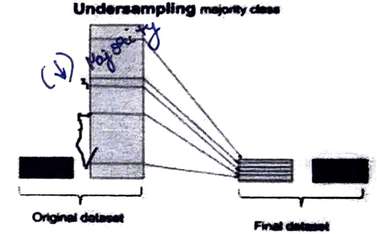
Minority ⇒ Under-sampling

Don't touch

Oversampling minority class



Undersampling majority class



In [1]:
```
from imblearn.under_sampling import RandomUnderSampler
from imblearn.over_sampling import RandomOverSampler
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

import seaborn as sns
import pandas as pd
import numpy as np

from collections import Counter

from numpy import mean
```

(Minority ↑)

Random OverSampling

In [2]:
```
## Defining an imbalance data-set We can demonstrate this on a simple synth
## problem with a 1:100 class imbalance.
X, y = list(make_classification(n_samples=10000, weights=[0.99], flip_y=0))

print("X-type : ",type(X))
print("y-type : ",type(y))

print("X-Dimension : ",X.ndim)
print("Y-Dimension : ",y.ndim)
```

```
X-type :  <class 'numpy.ndarray'>
y-type :  <class 'numpy.ndarray'>
X-Dimension :  2
Y-Dimension :  1
```

In [3]:
```
print(Counter(y))
np.array(np.unique(y, return_counts=True))
```

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

Out[3]:
```
array([[   0,    1],
       [9900,  100]], dtype=int64)
```

This means that 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.

In [4]:
```
# define oversampling strategy
oversample = RandomOverSampler(sampling_strategy='minority')
## we are going to scale the minority class to majority class
type(oversample)
```

Out[4]:
```
imblearn.over_sampling._random_over_sampler.RandomOverSampler
```

In [5]:
```
oversample = RandomOverSampler(sampling_strategy=0.5)
```

A floating point value can be specified to indicate the ratio of minority class majority examples in the transformed dataset.

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

This means that if the majority class had 1,000 examples and the minority class had 100, the transformed dataset would have 500 examples of the minority class.

In [6]:
```
# fit and apply the transform
X_over, y_over = oversample.fit_resample(X, y)
```

In [7]:
```
print(Counter(y_over))
np.array(np.unique(y_over, return_counts=True))
```

```
Counter({0: 9900, 1: 4950})
```

Out[7]:
```
array([[   0,    1],
       [9900, 4950]], dtype=int64)
```

In [8]:
```
# 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=
score  = mean(scores)
print('F1 Score: %.3f' % score)
```

```
F1 Score: 0.988
```

This is Evaluating a decision tree on an imbalanced dataset with a 1:100 class distribution.

The model is evaluated using repeated 10-fold cross-validation with three repeats, and the oversampling is performed on the training dataset within each fold separately, ensuring that there is no data leakage as might occur if the oversampling was performed prior to the cross-validation.

Running the example evaluates the decision tree model on the imbalanced dataset with oversampling.

The chosen model and resampling configuration are arbitrary, designed to provide a template that you can use to test undersampling with your dataset and learning algorithm, rather than optimally solve the synthetic dataset.

Random UnderSampling

(Majority ↓)

In [9]:
```
# define undersample strategy
undersample = RandomUnderSampler(sampling_strategy='majority')
## we are going to scale the majority class to minority class
type(oversample)

# define undersample strategy
undersample = RandomUnderSampler(sampling_strategy=0.5)
```

In [10]:
```
print(Counter(y))
np.array(np.unique(y, return_counts=True))
```

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

Out[10]:
```
array([[   0,    1],
       [9900,  100]], dtype=int64)
```

In [11]:
```
# fit and apply the transform
X_over, y_over = undersample.fit_resample(X, y)
```

In [12]:
```
print(Counter(y_over))
np.array(np.unique(y_over, return_counts=True))
```

```
Counter({0: 200, 1: 100})
```

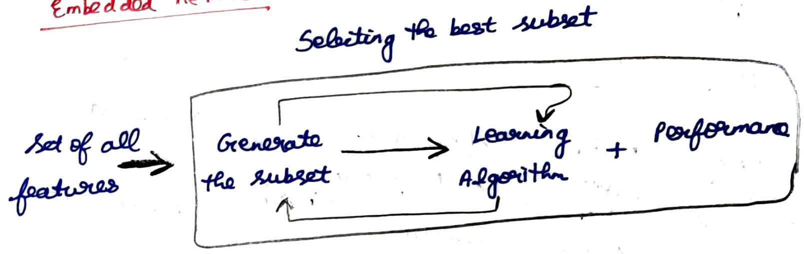
Out[12]:
```
array([[  0,   1],
       [200, 100]], dtype=int64)
```

In [13]:
```
# 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=
score  = mean(scores)
print('F1 Score: %.3f' % score)
```

```
F1 Score: 0.989
```

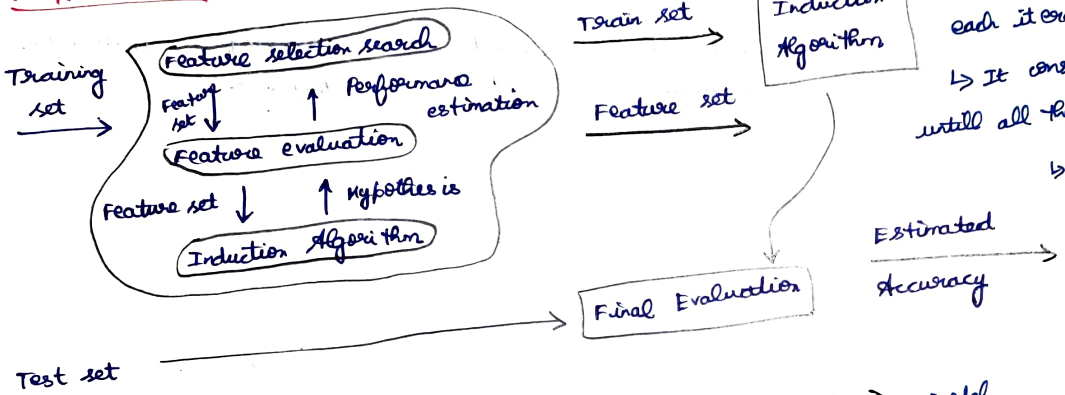
Embedded Method



Filter method

Set of all features → Selecting the best subset → Machine Learning Algorithms → Performance

Wrapper method



Recursive Feature Elimination

↳ Greedy optimization algorithm which aims to find best performing feature subset.

↳ It repeatedly creates models and keeps aside the best/worst performing feature at each iteration.

↳ It constructs the Next Model with the left features untill all the features are exhausted.

↳ It ~~takes the~~ then ranks the features based on the order of their elimination.

Forward selection

↳ iterative method in which we (start with having No feature) in model.

↳ In each iteration we keep on adding the feature which best improves of our model till an addition of a new variable does not improve the performance of the model.

Backward elimination:

↳ (Start with all the features) and removes the least significant feature at each iteration improves the performance of the model. This repeated untill No improvement is observed on removal of features.

Module-2 Statistical Foundations

1 Dimensionality Reduction

Data → ? Data Analysis Statistics Prediction

- Dimensionality reduction refers to techniques for reducing the number of input variables in training data.
- When dealing with high dimensional data, it is often useful to reduce the dimensionality by projecting the data to a lower dimensional subspace which captures the "essence" of the data. This is called dimensionality reduction.

multidimensio → not necessary

2

dimensional ← data modeling

- Dimensionality reduction is a data preparation technique performed on data prior to modeling. It might be performed after data cleaning and data scaling and before training a predictive model.

Techniques of Dimensionality reduction

Features → target

Feature selection methods
↳ wrapper method
↳ filter method.

3 Techniques for Dimensionality Reduction

extraction
already i have
some mapping
new feature

- **Feature Selection Methods**
- Perhaps the most common are so-called feature selection techniques that use scoring or statistical methods to select which features to keep and which features to delete.
- perform feature selection, to remove "irrelevant" features that do not help much with the classification problem
- Two main classes of feature selection techniques include wrapper methods and filter methods.

4

- Wrapper methods, as the name suggests, wrap a machine learning model, fitting and evaluating the model with different subsets of input features and selecting the subset the results in the best model performance. RFE is an example of a wrapper feature selection method.
- Filter methods use scoring methods, like correlation between the feature and the target variable, to select a subset of input features that are most predictive. Examples include Pearson's correlation and Chi-Squared test.

7 Autoencoder Methods

- Deep learning neural networks can be constructed to perform dimensionality reduction.
- A popular approach is called autoencoders. This involves framing a self-supervised learning problem where a model must reproduce the input correctly.

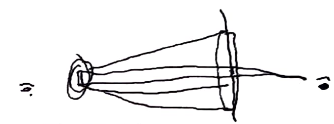
8

- An auto-encoder is a kind of unsupervised neural network that is used for dimensionality reduction and feature discovery. More precisely, an auto-encoder is a feedforward neural network that is trained to predict the input itself.

PCA
LDA

5 Matrix Factorization

- Techniques from linear algebra can be used for dimensionality reduction.
- Specifically, matrix factorization methods can be used to reduce a dataset matrix into its constituent parts.
- Examples include the eigendecomposition and singular value decomposition.
- The parts can then be ranked and a subset of those parts can be selected that best captures the salient structure of the matrix that can be used to represent the dataset.
- The most common method for ranking the components is principal components analysis, or PCA for short.



6 Manifold Learning

- Techniques from high-dimensionality statistics can also be used for dimensionality reduction.
- In mathematics, a projection is a kind of function or mapping that transforms data in some way.
- These techniques are sometimes referred to as "*manifold learning*" and are used to create a low-dimensional projection of high-dimensional data, often for the purposes of data visualization.
- The projection is designed to both create a low-dimensional representation of the dataset whilst best preserving the salient structure or relationships in the data.