Babeș-Bolyai University
Faculty of Mathematics and Computer Science

Curs opțional
Modele de inteligență artificială în schimbarea climatică

Abordări pentru seturile de date care nu au clasele echilibrate

Handling Imbalanced Classes

- In the real world, imbalanced classes are everywhere.
- Our best strategy is simply to collect more instances —especially instances from the minority class.
- However, this is often just not possible, so we have to resort to other options.
- A second strategy is to use a model evaluation metric better suited to imbalanced classes.
- A third strategy is to use the class weighing parameters included in implementations of some models.
- This allows us to have the algorithm adjust for imbalanced classes. Fortunately, many scikit-learn classifiers have a class_weight parameter, making it a good option.
- The fourth and fifth strategies are related: downsampling and upsampling.
  - In downsampling we create a random subset of the majority class of equal size to the minority class.
  - In upsampling we repeatedly sample with replacement from the minority class to make it of equal size as the majority class.
  - The decision between using downsampling and upsampling is context-specific, and in general we should try both to see which produces better results.

- We have a target vector with highly imbalanced classes.
- To demonstrate our solutions, we need to create some data with imbalanced classes.
- Fisher's Iris dataset contains three balanced classes of 50 instances, each indicating the species of flower (*Iris setosa*, *Iris virginica*, and *Iris versicolor*).
- To unbalance the dataset, we remove 40 of the 50 *Iris setosa* instances and then merge the *Iris virginica* and *Iris versicolor* classes.
- The end result is a binary target vector indicating if an instance is an *Iris setosa* flower or not.
- The result is 10 instances of *Iris setosa* (class 0) and 100 instances of not Iris setosa (class 1).

```python
# Load libraries
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import load_iris
# Load iris data
iris = load_iris()
# Create feature matrix
features = iris.data
# Create target vector
target = iris.target
# Remove first 40 observations
features = features[40:,:]
target = target[40:]
# Create binary target vector indicating if class 0
target = np.where((target == 0), 0, 1)
# Look at the imbalanced target vector
target
```

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
```

```python
# Create weights
weights = {0: .9, 1: 0.1}
# Create random forest classifier with weights
RandomForestClassifier(class_weight=weights)
```

```
RandomForestClassifier(bootstrap=True, class_weight={0: 0.9, 1: 0.1},
            criterion='gini', max_depth=None, max_features='auto',
            max_leaf_nodes=None, min_impurity_decrease=0.0,
            min_impurity_split=None, min_samples_leaf=1,
            min_samples_split=2, min_weight_fraction_leaf=0.0,
            n_estimators=10, n_jobs=1, oob_score=False, random_state=None,
            verbose=0, warm_start=False)
```

- Many algorithms in scikit-learn offer a parameter to weight classes during training to counteract the effect of their imbalance. RandomForestClassifier is a popular classification algorithm and includes a class_weightparameter.
- We can pass an argument specifying the desired class weights explicitly.

- Or we can pass balanced, which automatically creates weights inversely proportional to class frequencies.

```
# Train a random forest with balanced class weights
RandomForestClassifier(class_weight="balanced")
```

```
RandomForestClassifier(bootstrap=True, class_weight='balanced',
            criterion='gini', max_depth=None, max_features='auto',
            max_leaf_nodes=None, min_impurity_decrease=0.0,
            min_impurity_split=None, min_samples_leaf=1,
            min_samples_split=2, min_weight_fraction_leaf=0.0,
            n_estimators=10, n_jobs=1, oob_score=False, random_state=None,
            verbose=0, warm_start=False)
```

- Alternatively, we can downsample the majority class or upsample the minority class.
- In downsampling, we randomly sample without replacement from the majority class (i.e., the class with more observations) to create a new subset of instances equal in size to the minority class.

```python
# Indicies of each class' observations
i_class0 = np.where(target == 0)[0]
i_class1 = np.where(target == 1)[0]
# Number of observations in each class
n_class0 = len(i_class0)
n_class1 = len(i_class1)
# For every observation of class 0, randomly sample
# from class 1 without replacement
i_class1_downsampled = np.random.choice(i_class1, size=n_class0, replace=False)
# Join together class 0's target vector with the
# downsampled class 1's target vector
np.hstack((target[i_class0], target[i_class1_downsampled]))

# Join together class 0's feature matrix with the
# downsampled class 1's feature matrix
np.vstack((features[i_class0,:], features[i_class1_downsampled,:]))[0:5]
```

```
array([[5. , 3.5, 1.3, 0.3],
       [4.5, 2.3, 1.3, 0.3],
       [4.4, 3.2, 1.3, 0.2],
       [5. , 3.5, 1.6, 0.6],
       [5.1, 3.8, 1.9, 0.4]])
```

```python
# For every observation in class 1, randomly sample from class 0 with replacement
i_class0_upsampled = np.random.choice(i_class0, size=n_class1, replace=True)
# Join together class 0's upsampled target vector with class 1's target vector
np.concatenate((target[i_class0_upsampled], target[i_class1]))
```

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

- For example, if the minority class has 10 instances, we will randomly select 10 instances from the majority class and use those 20 instances as our data.
- Here we do exactly that using our unbalanced Iris data.
- Our other option is to upsample the minority class.
- In upsampling, for every observation in the majority class, we randomly select an observation from the minority class with replacement.
- The end result is the same number of instances from the minority and majority classes.
- Upsampling is implemented very similarly to downsampling, just in reverse.

Handling Imbalanced Classes - train a simple classifier model

Like many other learning algorithms in scikit-learn, LogisticRegression comes with a built-in method of handling imbalanced classes.

If we have highly imbalanced classes and have not addressed it during preprocessing, we have the option of using the class_weight parameter to weight the classes to make certain we have a balanced mix of each class (often a more useful argument is balanced, wherein classes are automatically weighted inversely proportional to how frequently they appear in the data).

Specifically, the balanced argument will automatically weigh classes inversely proportional to their frequency:

$$w_j = \frac{n}{kn_j}$$

where *wj* is the weight to class *j*, *n* is the number of observations, *nj* is the number of instances in class *j*, and *k* is the total number of classes.

```python
# Load libraries
import numpy as np
from sklearn.linear_model import LogisticRegression
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
# Load data
iris = datasets.load_iris()
features = iris.data
target = iris.target
# Make class highly imbalanced by removing first 40 observations
features = features[40:,:]
target = target[40:]
# Create target vector indicating if class 0, otherwise 1
target = np.where((target == 0), 0, 1)
# Standardize features
scaler = StandardScaler()
features_standardized = scaler.fit_transform(features)
# Create decision tree classifier object
logistic_regression = LogisticRegression(random_state=0, class_weight="balanced")
# Train model
model = logistic_regression.fit(features_standardized, target)
```

Handling Imbalanced Classes - train a support vector machine classifier

Solution: Increase the penalty for misclassifying the smaller class using class_weight: In support vector machines, C is a hyperparameter determining the penalty for misclassifying an observation.

One method for handling imbalanced classes in support vector machines is to weight C by classes, so that: $C_k = C * w_j$

where C is the penalty for misclassification, w_j is a weight inversely proportional to class j's frequency, and C_j is the C value for class j.

The general idea is to increase the penalty for misclassifying minority classes to prevent them from being "overwhelmed" by the majority class.

In scikit-learn, when using SVC we can set the values for C_j automatically by setting class_weight='balanced'.

The balanced argument automatically weighs classes such that:

where w_j is the weight to class j, n is the number of observations, n_j is the number of instances in class j, and k is the total number of classes.

```
from sklearn.svm import SVC
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
import numpy as np
#Load data with only two classes
iris = datasets.load_iris()
features = iris.data[:100,:]
target = iris.target[:100]
# Make class highly imbalanced by removing first 40 observations
features = features[40:,:]
target = target[40:]
# Create target vector indicating if class 0, otherwise 1
target = np.where((target == 0), 0, 1)
# Standardize features
scaler = StandardScaler()
features_standardized = scaler.fit_transform(features)
# Create support vector classifier
svc = SVC(kernel="linear", class_weight="balanced", C=1.0, random_state=0)
# Train classifier
model = svc.fit(features_standardized, target)
```

Handling Imbalanced Classes - Metrics

When in the presence of imbalanced classes, accuracy suffers from a paradox where a model is highly accurate but lacks predictive power.

For example, imagine we are trying to predict the presence of a very rare cancer that occurs in 0.1% of the population.

After training our model, we find the accuracy is at 95%. However, 99.9% of people do not have the cancer: if we simply created a model that "predicted" that nobody had that form of cancer, our naive model would be 4.9% more accurate, but clearly is not able to predict anything.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

For this reason, we are often motivated to use other metrics like precision, recall, and the F1 score.

Precision is the proportion of every observation predicted to be positive that is actually positive.

We can think about it as a measurement noise in our predictions—that is, when we predict something is positive, how likely we are to be right.

Models with high precision are pessimistic in that they only predict an observation is of the positive class when they are very certain about it.

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

Handling Imbalanced Classes – Metrics 2

Recall is the proportion of every positive observation that is truly positive. Recall measures the model's ability to identify an observation of the positive class. Models with high recall are optimistic in that they have a low bar for predicting that an observation is in the positive class:

$$\text{Recall} = \frac{TP}{TP + FN}$$

Almost always we want some kind of balance between precision and recall, and this role is filled by the F1 score. The F1 score is the harmonic mean (a kind of average used for ratios):

It is a measure of correctness achieved in positive prediction—that is, of instances labeled as positive, how many are actually positive:

$$F_1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

This is one of the downsides to accuracy; precision and recall are less intuitive, but more suitable for imbalanced classes.

Metrics - examples

As an evaluation metric, accuracy has some valuable properties, especially its simple intuition. However, better metrics often involve using some balance of precision and recall—that is, a trade-off between the optimism and pessimism of our model. *F1* represents a balance between the recall and precision, where the relative contributions of both are equal.

```
#Load libraries
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import make_classification
# Generate features matrix and target vector
X, y = make_classification(n_samples = 10000, n_features = 3,
n_informative = 3, n_redundant = 0, n_classes = 2, random_state = 1)
# Create logistic regression
logit = LogisticRegression()
# Cross-validate model using accuracy
cross_val_score(logit, X, y, scoring="accuracy")
```

```
array([0.95170966, 0.9580084 , 0.95558223])
```

```
# Cross-validate model using precision
cross_val_score(logit, X, y, scoring="precision")
```

```
array([0.95252404, 0.96583282, 0.95558223])
```

```
# Cross-validate model using recall
cross_val_score(logit, X, y, scoring="recall")
```

```
array([0.95080984, 0.94961008, 0.95558223])
```

```
# Cross-validate model using f1
cross_val_score(logit, X, y, scoring="f1")
```

```
array([0.95166617, 0.95765275, 0.95558223])
```

```
# Alternatively to using cross_val_score, if we already have
# the true y values and the predicted y values,
# we can calculate metrics like accuracy and recall directly:
# Load library
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
# Create training and test split
X_train, X_test, y_train, y_test = train_test_split(X,
y, test_size=0.1,
random_state=1)

# Predict values for training target vector
y_hat = logit.fit(X_train, y_train).predict(X_test)
# Calculate accuracy
accuracy_score(y_test, y_hat)
```

```
0.947
```