Cost-Sensitive Decision Trees for Imbalanced Classification

by **Jason Brownlee** on August 21, 2020 in **Imbalanced Classification** **18**

The decision tree algorithm is effective for balanced classification, although it does not perform well on imbalanced datasets.

The split points of the tree are chosen to best separate examples into two groups with minimum mixing. When both groups are dominated by examples from one class, the criterion used to select a split point will see good separation, when in fact, the examples from the minority class are being ignored.

This problem can be overcome by modifying the criterion used to evaluate split points to take the importance of each class into account, referred to generally as the weighted split-point or weighted decision tree.

In this tutorial, you will discover the weighted decision tree for imbalanced classification.

After completing this tutorial, you will know:

- How the standard decision tree algorithm does not support imbalanced classification.
- How the decision tree algorithm can be modified to weight model error by class weight when selecting splits.
- How to configure class weight for the decision tree algorithm and how to grid search different class weight configurations.

Kick-start your project with my new book Imbalanced Classification with Python, including *step-by-step tutorials* and the *Python source code* files for all examples.

Let's get started.

How to Implement Weighted Decision Trees for Imbalanced Classification
Photo by Bonnie Moreland, some rights reserved.

Tutorial Overview

This tutorial is divided into four parts; they are:

1. Imbalanced Classification Dataset
2. Decision Trees for Imbalanced Classification
3. Weighted Decision Trees With Scikit-Learn
4. Grid Search Weighted Decision Trees

Imbalanced Classification Dataset

Before we dive into the modification of decision for imbalanced classification, let's first define an imbalanced classification dataset.

We can use the make_classification() function to define a synthetic imbalanced two-class classification dataset. We will generate 10,000 examples with an approximate 1:100 minority to majority class ratio.

```
...
# 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=3)
```

Once generated, we can summarize the class distribution to confirm that the dataset was created as we expected.

```
...
# summarize class distribution
counter = Counter(y)
print(counter)
```

Finally, we can create a scatter plot of the examples and color them by class label to help understand the challenge of classifying examples from this dataset.

```
...
# 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()
```

Tying this together, the complete example of generating the synthetic dataset and plotting the examples is listed below.

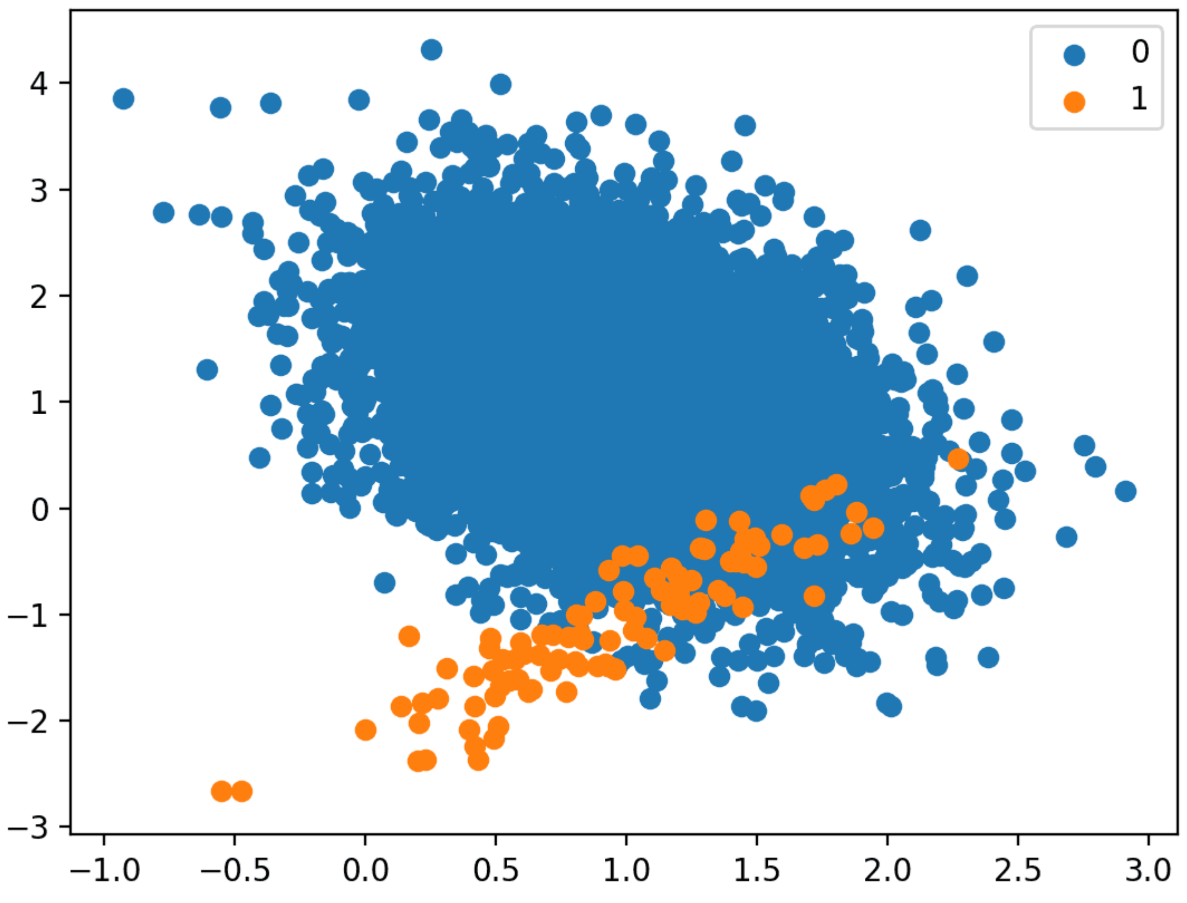
```
# Generate and plot a synthetic imbalanced classification dataset
from collections import Counter
from sklearn.datasets import make_classification
from matplotlib import pyplot
from numpy import where
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
 n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=3)
# 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()
```

Running the example first creates the dataset and summarizes the class distribution.

We can see that the dataset has an approximate 1:100 class distribution with a little less than 10,000 examples in the majority class and 100 in the minority class.

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

Next, a scatter plot of the dataset is created showing the large mass of examples for the majority class (blue) and a small number of examples for the minority class (orange), with some modest class overlap.



Scatter Plot of Binary Classification Dataset With 1 to 100 Class Imbalance

Next, we can fit a standard decision tree model on the dataset.

A decision tree can be defined using the DecisionTreeClassifier class in the scikit-learn library.

```
...
# define model
model = DecisionTreeClassifier()
```

We will use repeated cross-validation to evaluate the model, with three repeats of 10-fold cross-validation. The mode performance will be reported using the mean ROC area under curve (ROC AUC) averaged over repeats and all folds.

```
...
# 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))
```

Tying this together, the complete example of defining and evaluating a standard decision tree model on the imbalanced classification problem is listed below.

Decision trees are an effective model for binary classification tasks, although by default, they are not effective at imbalanced classification.

```
# fit a decision tree 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.tree import DecisionTreeClassifier
# 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=3)
# define model
model = DecisionTreeClassifier()
# 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))
```

Running the example evaluates the standard decision tree model on the imbalanced dataset and reports the mean ROC AUC.

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

We can see that the model has skill, achieving a ROC AUC above 0.5, in this case achieving a mean score of 0.746.

```
Mean ROC AUC: 0.746
```

This provides a baseline for comparison for any modifications performed to the standard decision tree algorithm.

Decision Trees for Imbalanced Classification

The decision tree algorithm is also known as Classification and Regression Trees (CART) and involves growing a tree to classify examples from the training dataset.

The tree can be thought to divide the training dataset, where examples progress down the decision points of the tree to arrive in the leaves of the tree and are assigned a class label.

The tree is constructed by splitting the training dataset using values for variables in the dataset. At each point, the split in the data that results in the purest (least mixed) groups of examples is chosen in a greedy manner.

Here, purity means a clean separation of examples into groups where a group of examples of all 0 or all 1 class is the purest, and a 50-50 mixture of both classes is the least pure. Purity is most commonly calculated using Gini impurity, although it can also be calculated using entropy.

The calculation of a purity measure involves calculating the probability of an example of a given class being misclassified by a split. Calculating these probabilities involves summing the number of examples in each class within each group.

The splitting criterion can be updated to not only take the purity of the split into account, but also be weighted by the importance of each class.

> *Our intuition for cost-sensitive tree induction is to modify the weight of an instance proportional to the cost of misclassifying the class to which the instance belonged …*

—An Instance-weighting Method To Induce Cost-sensitive Trees, 2002.

This can be achieved by replacing the count of examples in each group by a weighted sum, where the coefficient is provided to weight the sum.

Larger weight is assigned to the class with more importance, and a smaller weight is assigned to a class with less importance.

- **Small Weight**: Less importance, lower impact on node purity.
- **Large Weight**: More importance, higher impact on node purity.

A small weight can be assigned to the majority class, which has the effect of improving (lowering) the purity score of a node that may otherwise look less well sorted. In turn, this may allow more examples from the majority class to be classified for the minority class, better accommodating those examples in the minority class.

> *Higher weights [are] assigned to instances coming from the class with a higher value of misclassification cost.*

— Page 71, Learning from Imbalanced Data Sets, 2018.

As such, this modification of the decision tree algorithm is referred to as a weighted decision tree, a class-weighted decision tree, or a cost-sensitive decision tree.

Modification of the split point calculation is the most common, although there has been a lot of research into a range of other modifications of the decision tree construction algorithm to better accommodate a class imbalance.

Weighted Decision Tree With Scikit-Learn

The scikit-learn Python machine learning library provides an implementation of the decision tree algorithm that supports class weighting.

The DecisionTreeClassifier class provides the *class_weight* argument that can be specified as a model hyperparameter. The *class_weight* is a dictionary that defines each class label (e.g. 0 and 1) and the weighting to apply in the calculation of group purity for splits in the decision tree when fitting the model.

For example, a 1 to 1 weighting for each class 0 and 1 can be defined as follows:

```
...
# define model
weights = {0:1.0, 1:1.0}
model = DecisionTreeClassifier(class_weight=weights)
```

The class weighing can be defined multiple ways; for example:

- **Domain expertise**, determined by talking to subject matter experts.
- **Tuning**, determined by a hyperparameter search such as a grid search.
- **Heuristic**, specified using a general best practice.

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 invert 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 = DecisionTreeClassifier(class_weight=weights)
```

We might also define the same ratio using fractions and achieve the same result.

For example:

```
...
# define model
weights = {0:0.01, 1:1.0}
model = DecisionTreeClassifier(class_weight=weights)
```

This heuristic is available directly by setting the *class_weight* to '*balanced*.'

For example:

```
...
# define model
model = DecisionTreeClassifier(class_weight='balanced')
```

We can evaluate the decision tree algorithm with a class weighting using the same evaluation procedure defined in the previous section.

We would expect the class-weighted version of the decision tree to perform better than the standard version of the decision tree without any class weighting.

The complete example is listed below.

```
# decision tree 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.tree import DecisionTreeClassifier
# 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=3)
# define model
model = DecisionTreeClassifier(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))
```

Running the example prepares the synthetic imbalanced classification dataset, then evaluates the class-weighted version of the decision tree algorithm using repeated cross-validation.

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

The mean ROC AUC score is reported, in this case, showing a better score than the unweighted version of the decision tree algorithm: 0.759 as compared to 0.746.

```
Mean ROC AUC: 0.759
```

Grid Search Weighted Decision Tree

Using a class weighting that is the inverse ratio of the training data is just a heuristic.

It is possible that better performance can be achieved with a different class weighting, and this too will depend on the choice of performance metric used to evaluate the model.

In this section, we will grid search a range of different class weightings for the weighted decision tree and discover which results in the best ROC AUC score.

We will try the following weightings for class 0 and 1:

- Class 0:100, Class 1:1.
- Class 0:10, Class 1:1.
- Class 0:1, Class 1:1.
- Class 0:1, Class 1:10.
- Class 0:1, Class 1:100.

These can be defined as grid search parameters for the GridSearchCV class as follows:

```
...
# 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)
```

We can perform the grid search on these parameters using repeated cross-validation and estimate model performance using ROC AUC:

```
...
# 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')
```

Once executed, we can summarize the best configuration as well as all of the results as follows:

```
...
# 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))
```

Tying this together, the example below grid searches five different class weights for the decision tree algorithm on the imbalanced dataset.

We might expect that the heuristic class weighing is the best performing configuration.

```python
# grid search class weights with decision tree for imbalance classification
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.tree import DecisionTreeClassifier
# 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=3)
# define model
model = DecisionTreeClassifier()
# 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'
# execute the grid search
grid_result = grid.fit(X, y)
# 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))
```

Running the example evaluates each class weighting using repeated k-fold cross-validation and reports the best configuration and the associated mean ROC AUC score.

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

In this case, we can see that the 1:100 majority to minority class weighting achieved the best mean ROC score. This matches the configuration for the general heuristic.

It might be interesting to explore even more severe class weightings to see their effect on the mean ROC AUC score.

```
Best: 0.752643 using {'class_weight': {0: 1, 1: 100}}
0.737306 (0.080007) with: {'class_weight': {0: 100, 1: 1}}
0.747306 (0.075298) with: {'class_weight': {0: 10, 1: 1}}
0.740606 (0.074948) with: {'class_weight': {0: 1, 1: 1}}
0.747407 (0.068104) with: {'class_weight': {0: 1, 1: 10}}
0.752643 (0.073195) with: {'class_weight': {0: 1, 1: 100}}
```

Further Reading

This section provides more resources on the topic if you are looking to go deeper.

Papers

- An Instance-weighting Method To Induce Cost-sensitive Trees, 2002.

Books

- Learning from Imbalanced Data Sets, 2018.
- Imbalanced Learning: Foundations, Algorithms, and Applications, 2013.

APIs

- sklearn.utils.class_weight.compute_class_weight API.
- sklearn.tree.DecisionTreeClassifier API.
- sklearn.model_selection.GridSearchCV API.

Summary

In this tutorial, you discovered the weighted decision tree for imbalanced classification.

Specifically, you learned:

- How the standard decision tree algorithm does not support imbalanced classification.
- How the decision tree algorithm can be modified to weight model error by class weight when selecting splits.
- How to configure class weight for the decision tree algorithm and how to grid search different class weight configurations.

Do you have any questions?
Ask your questions in the comments below and I will do my best to answer.

More On This Topic

Cost-Sensitive Learning for Imbalanced Classification

Undersampling Algorithms for Imbalanced Classification

How to Develop an Extra Trees Ensemble with Python

SMOTE for Imbalanced Classification with Python

Best Resources for Imbalanced Classification

Step-By-Step Framework for Imbalanced Classification…

About Jason Brownlee

Jason Brownlee, PhD is a machine learning specialist who teaches developers how to get results with modern machine learning methods via hands-on tutorials.

View all posts by Jason Brownlee →

‹ Cost-Sensitive Logistic Regression for Imbalanced Classification

Cost-Sensitive SVM for Imbalanced Classification ›

18 Responses to *Cost-Sensitive Decision Trees for Imbalanced Classification*

marco January 31, 2020 at 1:19 am #

REPLY

Hello Jason,
I have a general question.
I'm studying M/L e D/L since last may and I'm wondering whether it worths to continue studying Machine Learning (i.e. Scikit-Learn) or it is better to study Deep Learning (Keras).
Or it is better to continue to have a look at both of them?
What are pros/ cons of each (in a nuthshell)?
Thanks

Jason Brownlee January 31, 2020 at 7:56 am #

REPLY

Both. They are different tools to solve different problems.

Most tabular data for classification and regression is best solved using sklearn. Most other data like images, text, etc. is best solved with neural nets.

Peter Marelas January 31, 2020 at 1:03 pm # REPLY

I am wondering is it enough to adjust the class weights to determine the best parameters. Reason is you have not calibrated the decision boundary (threshold). Without that I am wondering whether the results are conclusive?

Jason Brownlee January 31, 2020 at 2:06 pm # REPLY

No, adjusting the weights is typically one step in the modeling pipeline, but can hep a ton when the classes are imbalanced.

Varsha January 16, 2021 at 4:35 pm # REPLY

Hello,
How to create weighted decision tree for imbalanced multiclass dataset?

Jason Brownlee January 17, 2021 at 6:04 am # REPLY

Using exactly the same manner.

Specify the balance of the classes for your dataset.

Meriem Ferdjouni May 22, 2021 at 4:29 pm # REPLY

Hello,

I think some changes are required in the scoring if we passed 'roc_auc' into scoring parameters in cross_validation_score().

in multi_class is a parameter in roc_auc_score(), the default value is 'raise' which will raise an error if it was used for multi-class problem. We need to pass 'ovr' or 'ovo' into multi_class parameter if we have a multi-class problem.

Please correct me if I am wrong, because I am encountering an error 'ValueError: multiclass format is not supported' when running the same evaluation line of code on a multi-class data.

Jason Brownlee May 23, 2021 at 5:23 am # REPLY

ROC AUC is not supported for multi-class classification.

Meriem May 23, 2021 at 5:28 am # REPLY

what parameter should I use for ‘scoring’ parameter please for the case of imbalanced multi-class problem, please?

Thank you!

Jason Brownlee May 24, 2021 at 5:39 am # REPLY

Most of the metrics for binary problems can be used for multi-class as long as you specify which classes are a minority and which are the majority.

This will help choose:
https://machinelearningmastery.com/tour-of-evaluation-metrics-for-imbalanced-classification/

Meriem May 23, 2021 at 5:11 am # REPLY

Hello,

I am running the same standard model on an imbalanced multi-class data, I encounter an error when I evaluate it using “cross_val_score” with the parameter “scoring” = ‘roc_auc’. The error is “ValueError: multiclass format is not supported”.

Any idea how to solve it?

Please note that I tried different ways, I also tried to upgrade sklearn.

Thank you!

Jason Brownlee May 23, 2021 at 5:26 am # REPLY

You cannot use ROC AUC with more than two classes.

João May 26, 2021 at 9:12 pm # REPLY

Hi,

I am probably missing something, but how do you assign the misclassification costs in the DT root node, since in the first node I still don't know who are the FP and FN cases? I'm struggling with this initial step to implement my own cost-sensitive DT.

Thank you!

Jason Brownlee May 27, 2021 at 5:38 am # REPLY

You can set the "class_weight" variable and the weighting will be used for the evaluation every split.

Brian April 1, 2022 at 10:42 am # REPLY

Hi Jason, can you give insight on how class_weight is utilized to modify the entropy equation, exactly. The sklearn documentation is unclear. One possibility is the summation over all classes wi*p(X=i)log(wi*p(X=i)) where wi are the weights of each classes. Thank you.

James Carmichael April 2, 2022 at 12:29 pm # REPLY

Hi Brian…The following may be of interest:

https://machinelearningmastery.com/cross-entropy-for-machine-learning/

Brian April 4, 2022 at 8:13 am # REPLY

I'll take a look at this- thanks!

James Carmichael April 4, 2022 at 8:55 am # REPLY

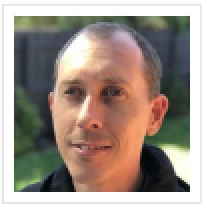
Keep up the great work Brian!

Leave a Reply

Name (required)

Email (will not be published) (required)

SUBMIT COMMENT

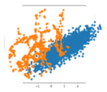


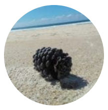
Welcome!
I'm *Jason Brownlee* PhD
and I **help developers** get results with **machine learning**.
Read more

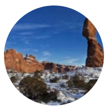
Never miss a tutorial:

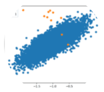
Picked for you:

SMOTE for Imbalanced Classification with Python




A Gentle Introduction to Threshold-Moving for Imbalanced Classification


Imbalanced Classification With Python (7-Day Mini-Course)


One-Class Classification Algorithms for Imbalanced Datasets

How to Fix k-Fold Cross-Validation for Imbalanced Classification

Loving the Tutorials?

The Imbalanced Classification EBook is
where you'll find the ***Really Good*** stuff.

>> SEE WHAT'S INSIDE