CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

scikit learn

# Machine Learning

## Machine Learning

Lecture: Dealing with Imbalanced Data

Ted Scully

# Data Pre-processing for Scikit Learn

- Dealing with Outliers (Optional)
- Dealing with Missing Values
- Handling Categorical Data
- Scaling Data
- **Handling Imbalance**
- Feature Selection
- Dimensionality Reduction

# Handling Imbalance

- **Class imbalance** in a classification task is a quite common problem when working with real-world data—this occurs when samples from one class or from multiple classes are over-represented in a dataset.

- There are several domains where this may occur, such as fraud detection, screening for diseases, network anomaly detection.

- While there are a wide range of approaches to dealing with this issue the most common is applying resampling techniques that attempts to equal the distribution.

# Imbalance – The Problem

- **In the example below we use a highly imbalanced dataset.**

```
from sklearn.model_selection import train_test_split
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix

df = pd.read_csv('train.csv')

targets = df.target.value_counts()
print (targets)

print ("Minority class represents just ",(targets[1]/len(df))*100, " % of the dataset")
```

```
0    573518
1     21694
Name: target, dtype: int64
Minority class represents just  3.6  % of the dataset
```

# Imbalance – The Problem

- Notice when we access the accuracy, the model appears to be doing well.
- But we will look at a little closer at the results over the next few slides.
- It would be helpful to **delve deeper** into these results. How many of each class did my model correctly classify and incorrectly classify.

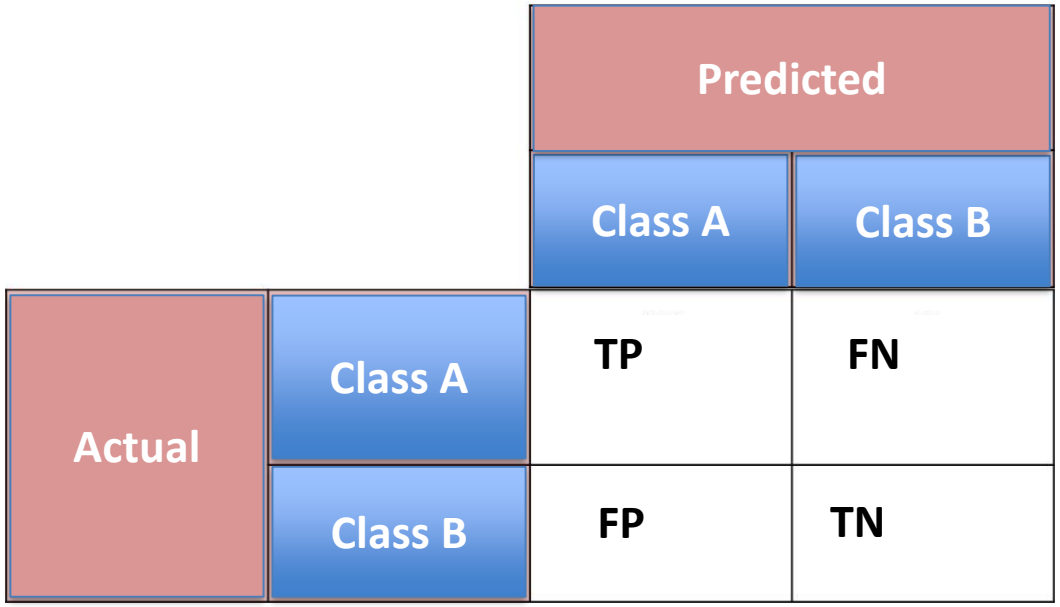
```
X = df.iloc[:, 2:]
y = df.iloc[:, 1]



X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=2)

model = RandomForestClassifier()
model.fit(X_train, y_train)
y_pred = model.predict(X_test)

accuracy = accuracy_score(y_test, y_pred)
print("Accuracy: ", accuracy )
```

Accuracy: 0.96

# Confusion Matrix

- A confusion matrix is a useful tool that contains information about the actual and predicted classifications of a classification algorithm. The confusion matrix in the example refers to class A as the positive class.

- **True positives (TP)** - Number of instances of Class A that were correctly classified as Class A.
- **False Positives (FP) -** Number of instances of Class B that were incorrectly classified as Class A.
- **False Negative (FN)** is the number of instances of Class A that were incorrectly classified as Class B.
- **True Negative (TN)** is the number of instances of Class B that were correctly classified as Class B.



# Measuring Accuracy

**Predicted**

| **Actual** | | Class A | Class B |
|---|---|---|---|
| | Class A | True Positives | False Negatives |
| | Class B | False Positives | True Negatives |

- Accuracy in machine learning algorithm is measured as:

$$\frac{True\ Positives\ + True\ Negatives}{True\ Positives + False\ Positives + True\ Negatives + False\ Negatives}$$

We can extract quite a lot of information from a confusion matrix. For example if we focus on the horse class we can see that there are 15 true positives, where a horse is correctly predicted to be a horse. There is only one instance of a horse being incorrectly classified, it is misclassified as a sheep.

Also notice that no goat is incorrectly classified as a horse and only a single sheep is classified as a horse.

| | | Predicted | | |
|---|---|---|---|---|
| | | Goat | Horse | Sheep |
| Actual | Goat | ***10*** | *0* | *6* |
| | Horse | *0* | ***15*** | *1* |
| | Sheep | *5* | *1* | ***9*** |

# Imbalance – The Problem

- **Now lets return to our example and observe what is returned if we generate a confusion matrix as shown in the previous slides.**

```
model = RandomForestClassifier()
model.fit(X_train, y_train)
y_pred = model.predict(X_test)

accuracy = accuracy_score(y_test, y_pred)
print("Accuracy: ", accuracy )


cf_mat = confusion_matrix(y_true=y_test, y_pred=y_pred)
print('Confusion matrix:\n', cf_mat)
```

```
Confusion matrix:
 [[114686      6]
 [  4350        1]]
```

# Imbalance – The Problem

- **Now lets return to our example and observe what is returned if we generate a confusion matrix as shown in the previous slides.**

```
model = RandomForestClassifier()
model.fit(X_train, y_train)
y_pred = model.predict(X_test)

accuracy = accuracy_score(y_test, y_pred)
print("Accuracy: ", accuracy )


cf_mat = confusion_matrix(y_true=y_test, y_pred=y_pred)
print('Confusion matrix:\n', cf_mat)
```

Despite the fact that the accuracy is very high we can see the performance of the model is really poor. There were 4351 instances of the minority class in the dataset. The model correctly classified just one of these. An accuracy of 0.02% on the minority class.

```
Confusion matrix:
 [[114686      6]
 [  4350        1]]
```

# Imbalance – The Problem

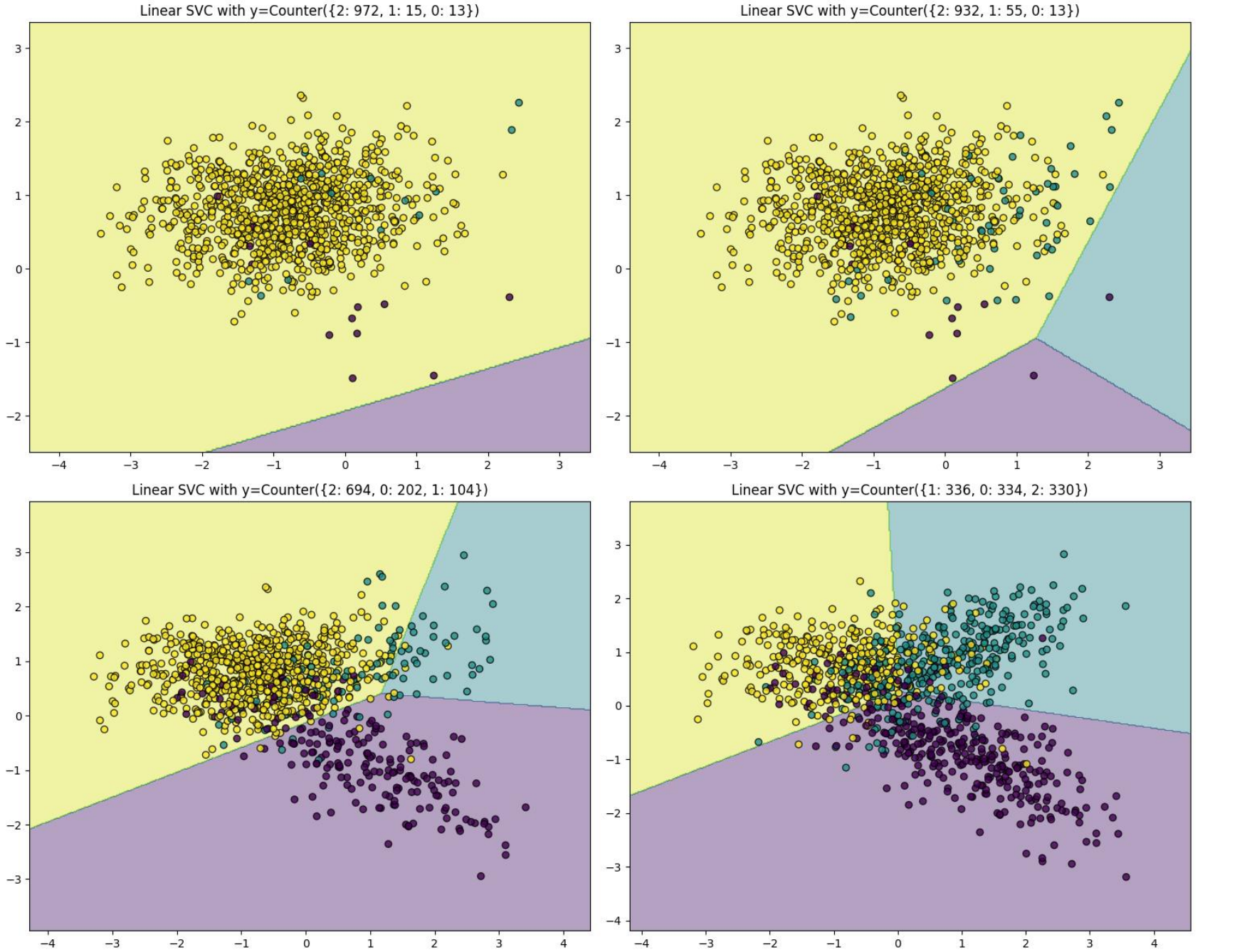
- **Now lets return to our example and observe what is returned if we generate a confusion matrix as shown in the previous slides.**

```
model = RandomForestClassifier()
model.fit(X_train, y_train)
y_pred = model.predict(X_test)

accuracy = accuracy_score(y_test, y_pred)
print("Accuracy: ", accuracy )


cf_mat = confusion_matrix(y_true=y_test, y_pred=y_pred)
print('Confusion matrix:\n', cf_mat)
```

In reality the machine learning model is fitting the majority class, it is fitting the data distribution.

Despite the fact that the accuracy is very high we can see the performance of the model is really poor. There were 4351 instances of the minority class in the dataset. The model correctly classified just one of these. An accuracy of 0.02% on the minority class.

```
Confusion matrix:
 [[114686      6]
 [  4350        1]]
```




https://imbalanced-learn.org/en/stable/auto_examples/over-sampling/plot_comparison_over_sampling.html

# Imbalanced Learn

- Imbalanced-learn is an excellent contribution package for Scikit-Learn and a collection of modern sampling techniques.

- The easiest way to install is using Conda. Go to your Anaconda folder and select your Anaconda Prompt and type the following command:

- *conda install -c conda-forge imbalanced-learn*

# Imbalance – The Problem

- There are a range of strategies that attempt to address the issue of imbalance. The most common techniques focus on sampling as a means of addressing the disparity.

- Common approaches include:
  - Random Undersampling
  - Random Oversampling
  - SMOTE
  - Tomek Links

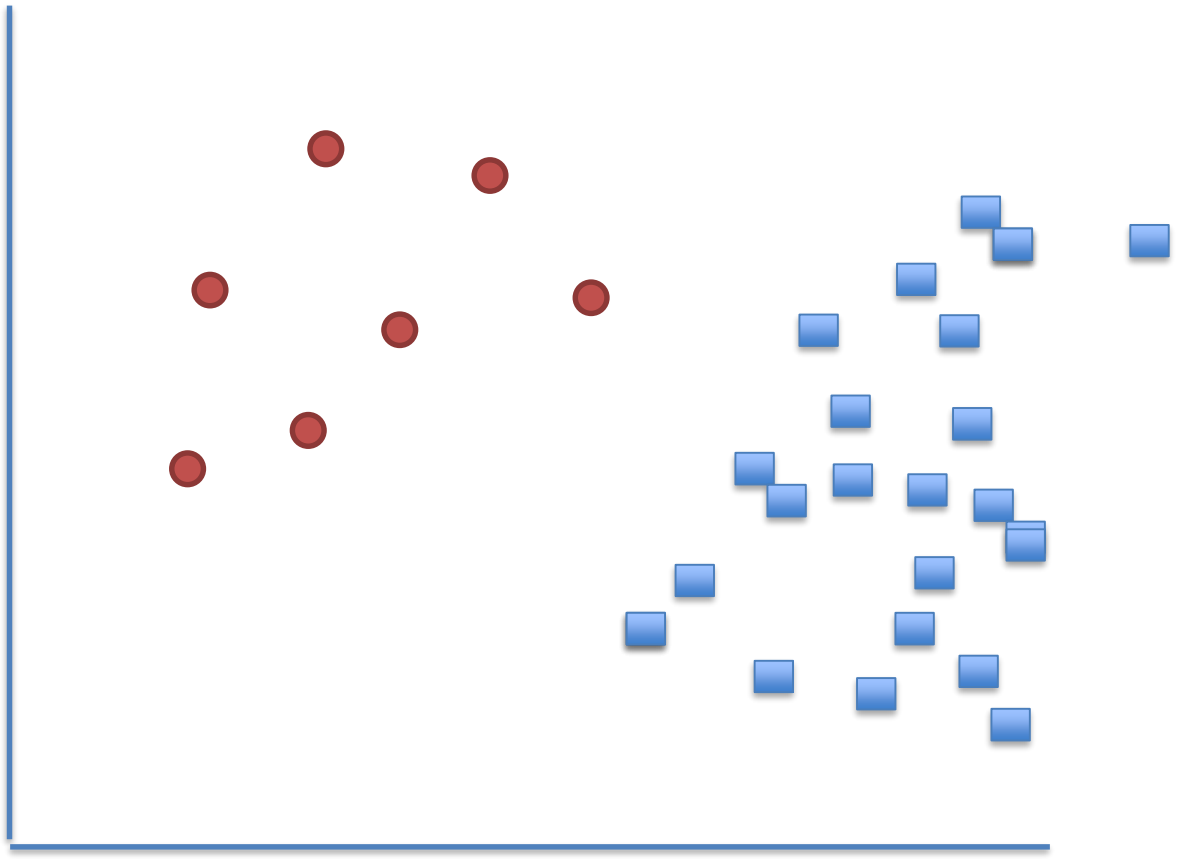
# Random Undersampling and Oversampling

- Both random under-sampling and oversampling are basic techniques that attempt to rebalance the data.
  - Random under-sampling will **randomly remove data** from the majority class until the two classes are balanced (imblearn.over_sampling.RandomOverSampler).
  - Random over-sampling in contrast will gradually increase the size of the minority class by **randomly duplicating instances** from the minority class. This is similar to observation weighting

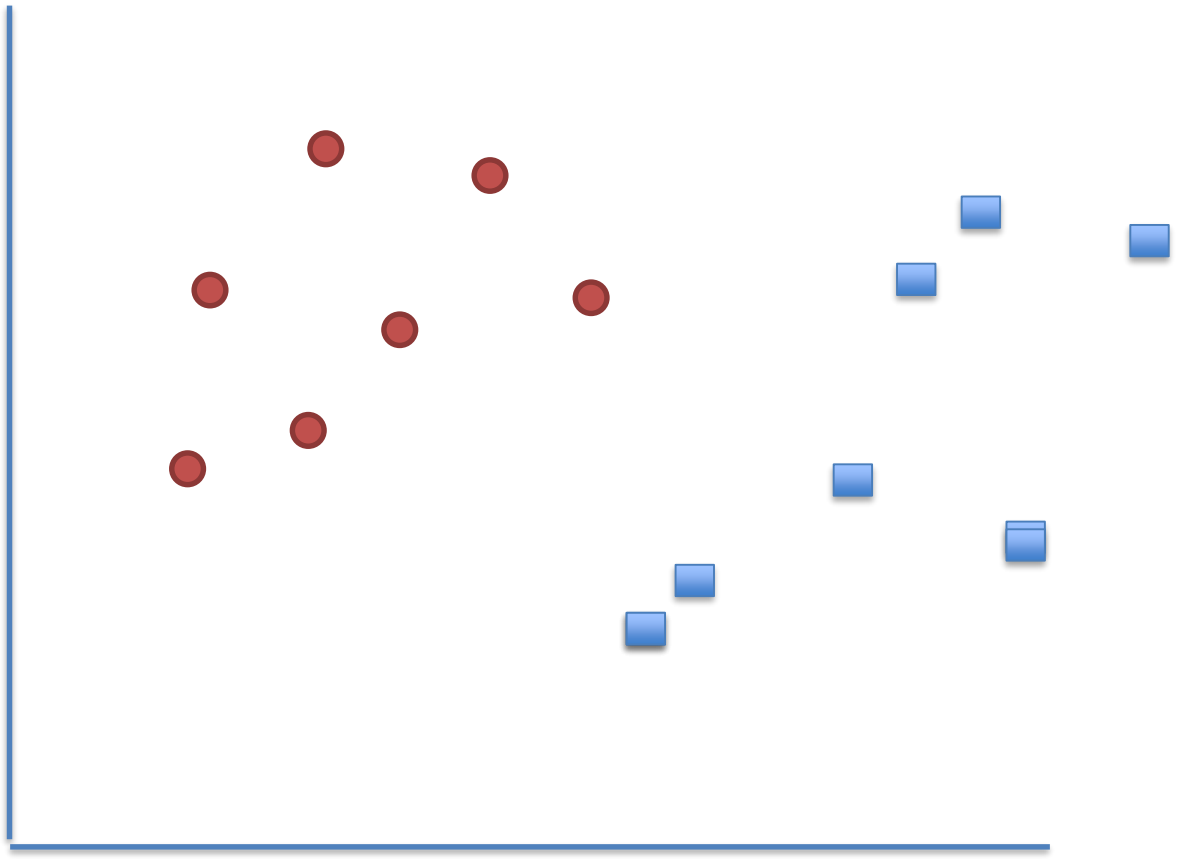
    (imblearn.under_sampling.RandomUnderSampler)

- The drawback of under-sampling is that you lose data. While oversampling may cause overfitting.

# Under-sampling (Before)



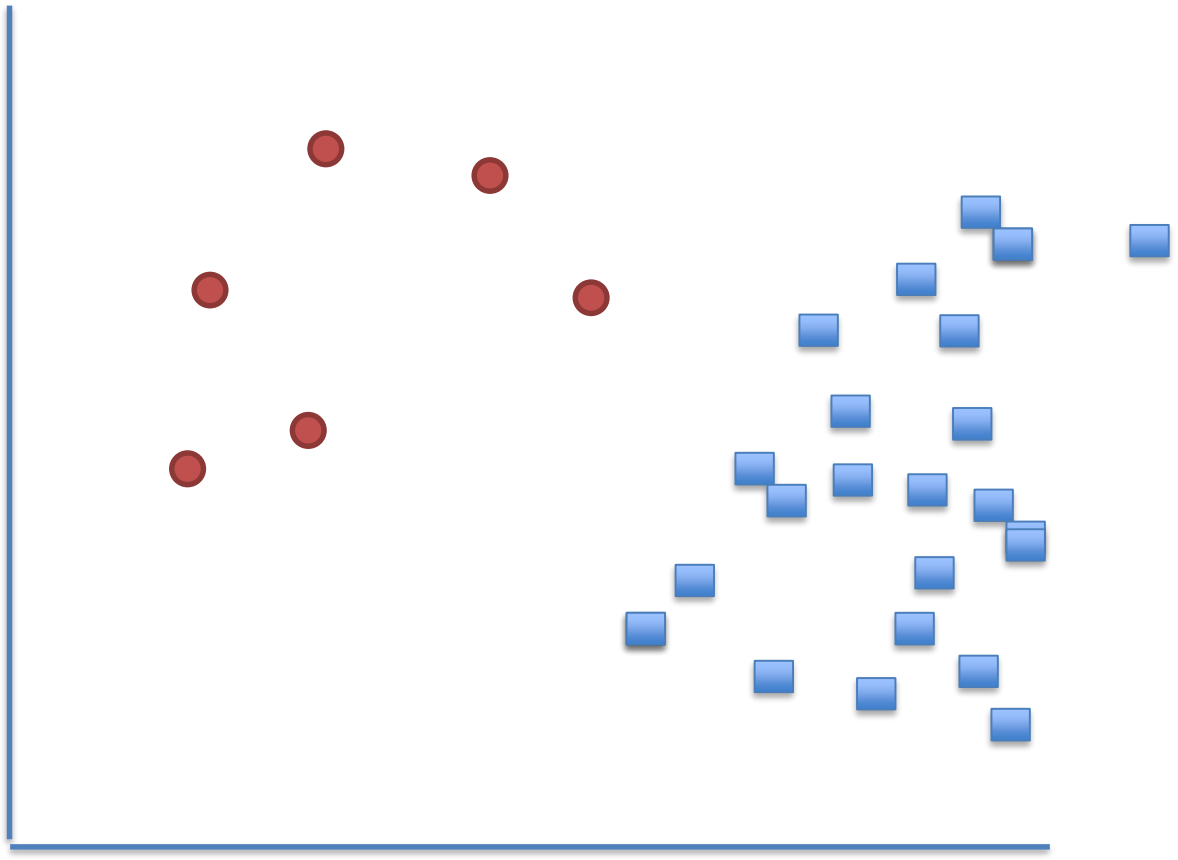
# Under-sampling (After)



# SMOTE

- SMOTE (Synthetic Minority Over-sampling TEchnique) is an oversampling technique that allows us to creates new artificial data points in our dataset (imblearn.over_sampling.SMOTE).

- We can interpret the operation of SMOTE as the iterative execution of the following steps:
  - Select a **data point from the minority class**
  - Identify it's **k nearest neighbours** in feature space.
  - **Randomly** select a neighbour
  - Get the difference between the original data point and the neighbour and multiplying it by a random number between 0 and 1.
  - Next a new **point is created** in feature space between the original data point and it's neighbour by adding the random number to the original data.

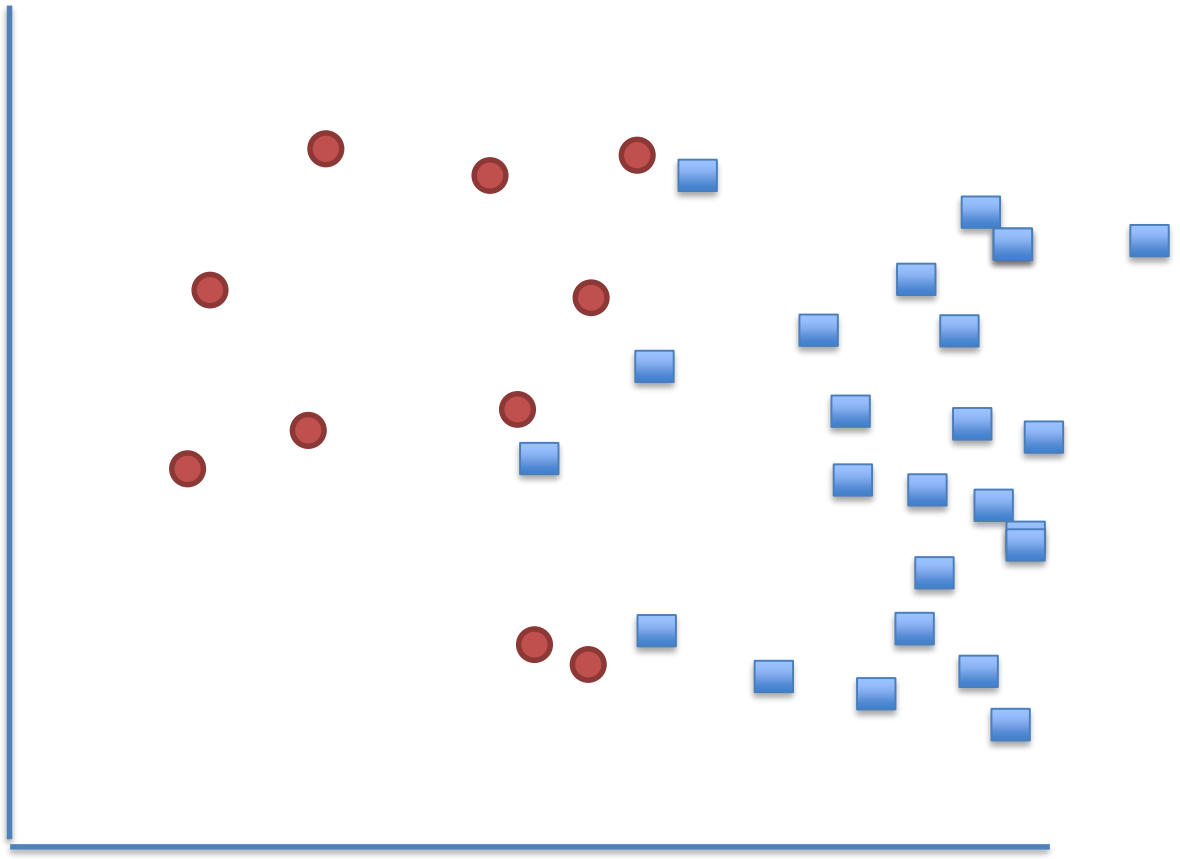


# Tomek Links

- Tomek links is a method of under-sampling and removes instances of the majority class in areas of significant overlap between the classes (imblearn.under_sampling.TomekLinks).

  - Let x be an instance of class A and y an instance of class B.
  - Let d(x, y) be the distance between x and y.
  - (x, y) is a **Tomek Link**, if for any instance z, $d(x, y) < d(x, z)$ or $d(x, y) < d(y, z)$

- This means that y is the nearest neighbour of x **or** that x is the nearest neighbour of y.
- Tomek links can be used to reduce the level of overlap between classes.
- Tomek Links can often be used in conjunction with an oversampling technique like SMOTE. For example, Imbalanced Learn provides a combination of both here.



```
from sklearn import datasets
from sklearn.svm import LinearSVC
from sklearn.model_selection import train_test_split
from imblearn.datasets import make_imbalance
from imblearn.over_sampling import SMOTE
from sklearn.metrics import confusion_matrix


RANDOM_STATE = 42

# Generate a balanced dataset
X, y = datasets.make_classification(n_classes=2, n_features=20, n_samples=15000,
                                    random_state=RANDOM_STATE)

# We use this initial dataset and make it imbalanced
X, y = make_imbalance(X, y, sampling_strategy={0: 7400, 1:200},
random_state=RANDOM_STATE)

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=RANDOM_STATE)
```

```
# Train the classifier
clf = LinearSVC(random_state=RANDOM_STATE)
clf.fit(X_train, y_train)

print (clf.score(X_test, y_test))

# Print confusion matrix
y_pred = clf.predict(X_test)

cf_mat = confusion_matrix(y_true=y_test, y_pred=y_pred)
print('Confusion matrix:\n', cf_mat)
```

```
0.975789473684
Confusion matrix:
 [[1837    9]
 [  37   17]]
```

Here we build and test a model on the imbalanced dataset. Notice that the performance on the minority class is very poor with an overall accuracy of the minority class only 31%.

If you were to print out the number of each class in the training data it would be {0: 5554, 1: 146}

```
# Use SMOTE to rebalance the dataset
sm = SMOTE(random_state=0)
X_train, y_train = sm.fit_sample(X_train, y_train)

# Train the classifier
clf = LinearSVC(random_state=RANDOM_STATE)
clf.fit(X_train, y_train)

print (clf.score(X_test, y_test))

# Print confusion matrix
y_pred = clf.predict(X_test)

cf_mat = confusion_matrix(y_true=y_test, y_pred=y_pred)
print('Confusion matrix:\n', cf_mat)
```

```
0.917894736842
Confusion matrix:
 [[1700  146]
 [  10   44]]
```

We rerun the same code but notice that this time we rebalance the dataset using SMOTE. Notice our overall accuracy drops but we have a much better overall performance on the minority class. Accuracy on the minority class goes from 31% to 81%.

If you were to print out the number of each class in the training data it would be {0: 5554, 1: 5554}

# Metrics for Imbalanced Dataset

- One issue you may have noticed in the previous slides is that while a confusion matrix is useful for gaining an insight into the performance of a model for each individual class, it can be difficult to quantify the specific impact of different sampling techniques on the performance of the model.

- It would be more useful if we had  specific metrics that could indicate the performance of the model.

- There are a range of metrics we can use such as precision, recall, specificity and geometric mean. We will cover these in more detail when we move on to the Evaluation section.

# Should I rebalance train and test set?

- A common mistake people make when dealing with an imbalanced dataset is they **rebalance the entire dataset** and then split into a training set and test set.
  - This is not good practice and will likely optimistically bias your results.

- You should apply the rebalancing technique to the training data only (not the test data).

- The model is developed by using the rebalanced training data and then evaluated on the test set (which has not been rebalanced).

- If you rebalance on the entire dataset and then perform the train/test split or cross fold validation then you risk optimistically biasing the final accuracy.
  - For example, you could have the same or similar data points appearing in both the training and test set.