Imbalanced datasets

What happens when our *training examples* are imbalanced

- some examples over-represented
- other examples under-represented

We already briefly covered this in [Loss Analysis (Training_Loss.ipynb#Conditional-loss)](Training_Loss.ipynb#Conditional-loss)

- Our motivation there was focusing on examples where errors occur
- Here our motivation is when the examples naturally partition into *imbalanced* subsets

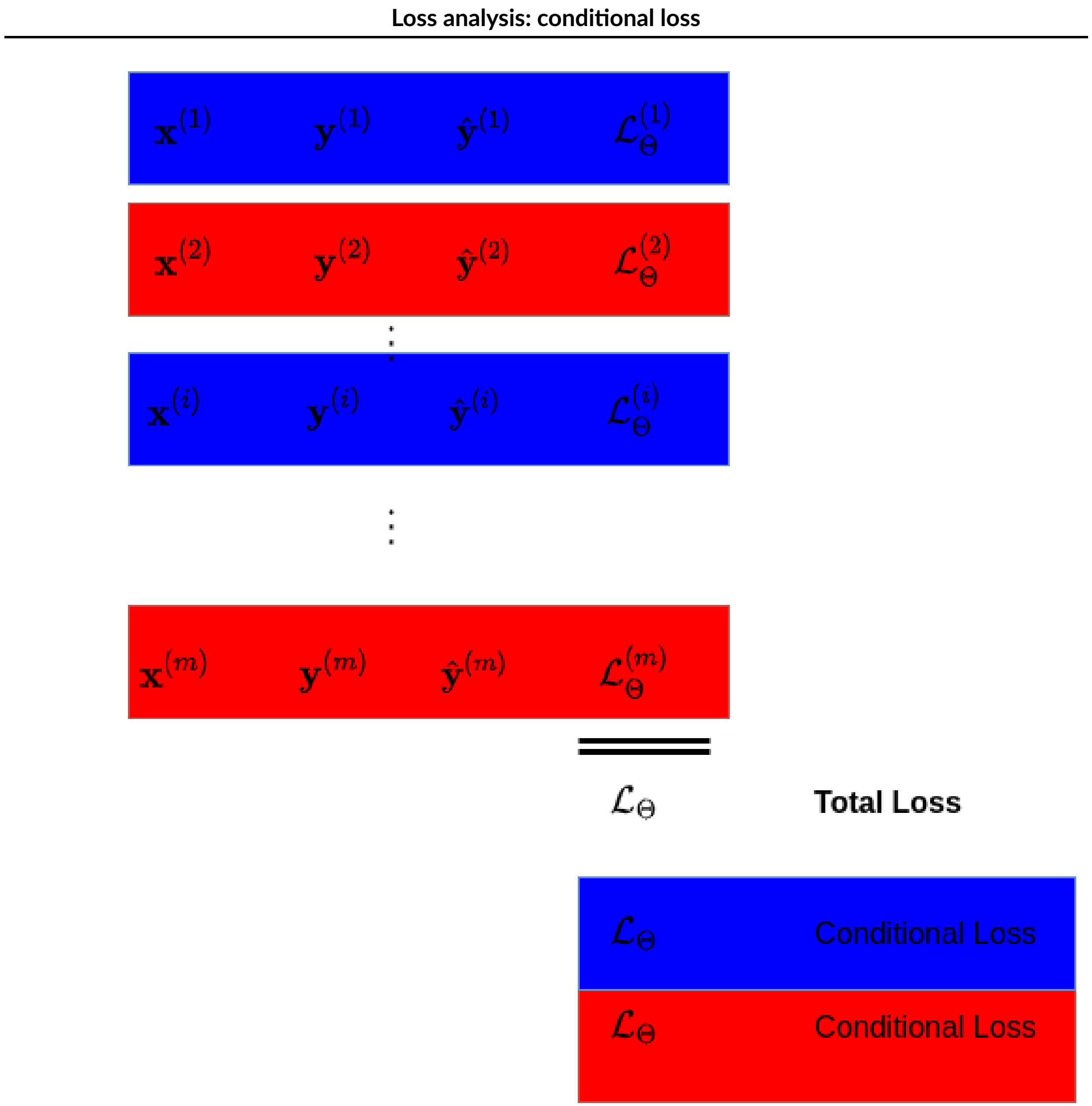
We revisit this in the case of a Binary Classification task, where one class dominates.

Training loss is usually given unconditionally:

Training Example				
$\mathbf{x}^{(1)}$	$\mathbf{y}^{(1)}$	$\hat{\mathbf{y}}^{(1)}$	$\mathcal{L}_{\Theta}^{(1)}$	
$\mathbf{x}^{(2)}$	$\mathbf{y}^{(2)}$	$\hat{\mathbf{y}}^{(2)}$	$\mathcal{L}_{\Theta}^{(2)}$	
	$\vdots$			
$\mathbf{x}^{(i)}$	$\mathbf{y}^{(i)}$	$\hat{\mathbf{y}}^{(i)}$	$\mathcal{L}_{\Theta}^{(i)}$	
	$\vdots$			
$\mathbf{x}^{(m)}$	$\mathbf{y}^{(m)}$	$\hat{\mathbf{y}}^{(m)}$	$\mathcal{L}_{\Theta}^{(m)}$	
			$\mathcal{L}_{\Theta}$	**Total Loss**

•

But we can also partition the examples and examine the loss in each partition



Suppose we partition the training examples into those whose class is Positive and those whose class is Negative:

$$\begin{aligned}\langle \mathbf{X}, \mathbf{y} \rangle &= [\mathbf{x}^{(\mathbf{i})}, \mathbf{y}^{(\mathbf{i})} | 1 \leq i \leq m] \\ &= [\mathbf{x}^{(\mathbf{i})}, \text{Positive} | 1 \leq i \leq m'] \;\cup\; [\mathbf{x}^{(\mathbf{i})}, \text{Negative} | 1 \leq i \leq m''] \\ &= \langle \mathbf{X}_{(\text{Positive})}, \mathbf{y}_{(\text{Positive})} \rangle \;\cup\; \langle \mathbf{X}_{(\text{Negative})}, \mathbf{y}_{(\text{Negative})} \rangle\end{aligned}$$

We can partition the training loss

$$\begin{aligned}\mathcal{L}_{\Theta} &= \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}_{\Theta}^{(\mathbf{i})}\\ &= \frac{m'}{m}\frac{1}{m'}\sum_{i' \in \mathbf{X}_{(\text{Positive})}}\mathcal{L}_{\Theta}^{(\mathbf{i})} + \frac{m''}{m}\frac{1}{m''}\sum_{i'' \in \mathbf{X}_{(\text{Negative})}}\mathcal{L}_{\Theta}^{(\mathbf{i})}\end{aligned}$$

That is, the Average loss is the weighted (with weights $\frac{m'}{m}, \frac{m''}{m}$) conditional losses

- $\frac{1}{m'}\sum_{i' \in \mathbf{X}_{(\text{Positive})}}\mathcal{L}_{\Theta}^{(\mathbf{i})}$
- $\frac{1}{m''}\sum_{i'' \in \mathbf{X}_{(\text{Negative})}}\mathcal{L}_{\Theta}^{(\mathbf{i})}$

As we've observed before

- As long as the majority class dominates in count (e.g., $m' \gg m''$)
- It is possible for Average Loss to be low
- Even if Conditional Loss for the minority class is high

This means that training is less likely to generalize well out of sample to the minority examples.

When the set of training examples $\langle \mathbf{X}, \mathbf{y} \rangle$ is such that

- $\mathbf{y}$ comes from set of categories C
- Where the distribution of $c \in C$ is *not* uniform

the dataset is called *imbalanced*.

This means that training is may be biased to not do as well on examples from under-represented classes.

For the Titanic survival:

- only 38% of the passengers survived, so the dataset is highly imbalanced
- a naive model that *always* predicted "Not survived" will
    - have 62% accuracy
        - be correct 100% of the time for 62% of the sample (those that didn't survive)
        - be incorrect 100% of the time for 38% of the sample (those that did survive)

The question is whether your use case requires high accuracy in *all* classes.

Approaches to imbalanced training data

There are a number of approaches to avoid a potential bias caused by imbalanced data.

There is even a website (https://imbalanced-learn.readthedocs.io/en/stable/user_guide.html) with software approaches to the topic.

Conditional Loss

- Use conditional metrics rather than unconditional metrics
  - Metric less influenced by size
  - Combination of Precision and Recall

Choose a model that is not sensitive to imbalance

- Decision Trees
    - branching structure can handle imbalance

Loss sensitive training

Modify the Loss function to weight conditional probabilities

Rather than

$$\begin{aligned}\mathcal{L}_\Theta &= \frac{1}{m}\sum_{i=1}^m \mathcal{L}_\Theta^{(\mathbf{i})} \\ &= \frac{m'}{m}\frac{1}{m'}\sum_{i' \in \mathbf{X}_{(\text{Positive})}} \mathcal{L}_\Theta^{(\mathbf{i})} + \frac{m''}{m}\frac{1}{m''}\sum_{i'' \in \mathbf{X}_{(\text{Negative})}} \mathcal{L}_\Theta^{(\mathbf{i})}\end{aligned}$$

adjust weights

$$\mathcal{L}_\Theta = C_{\text{Positive}} * \sum_{i' \in \mathbf{X}_{(\text{Positive})}} \mathcal{L}_\Theta^{(\mathbf{i})} + C_{\text{Negative}} * \sum_{i'' \in \mathbf{X}_{(\text{Negative})}} \mathcal{L}_\Theta^{(\mathbf{i})}$$

- Equally weighted across classes: $C_{\text{Positive}} = C_{\text{Negative}}$
- Relative importance
    - An error in one class may be more important than an error in the other

- `sklearn` inverse frequency weights
    - user-defined weights (`sklearn` optional `class_weights` argument to some classification models)

Resampling

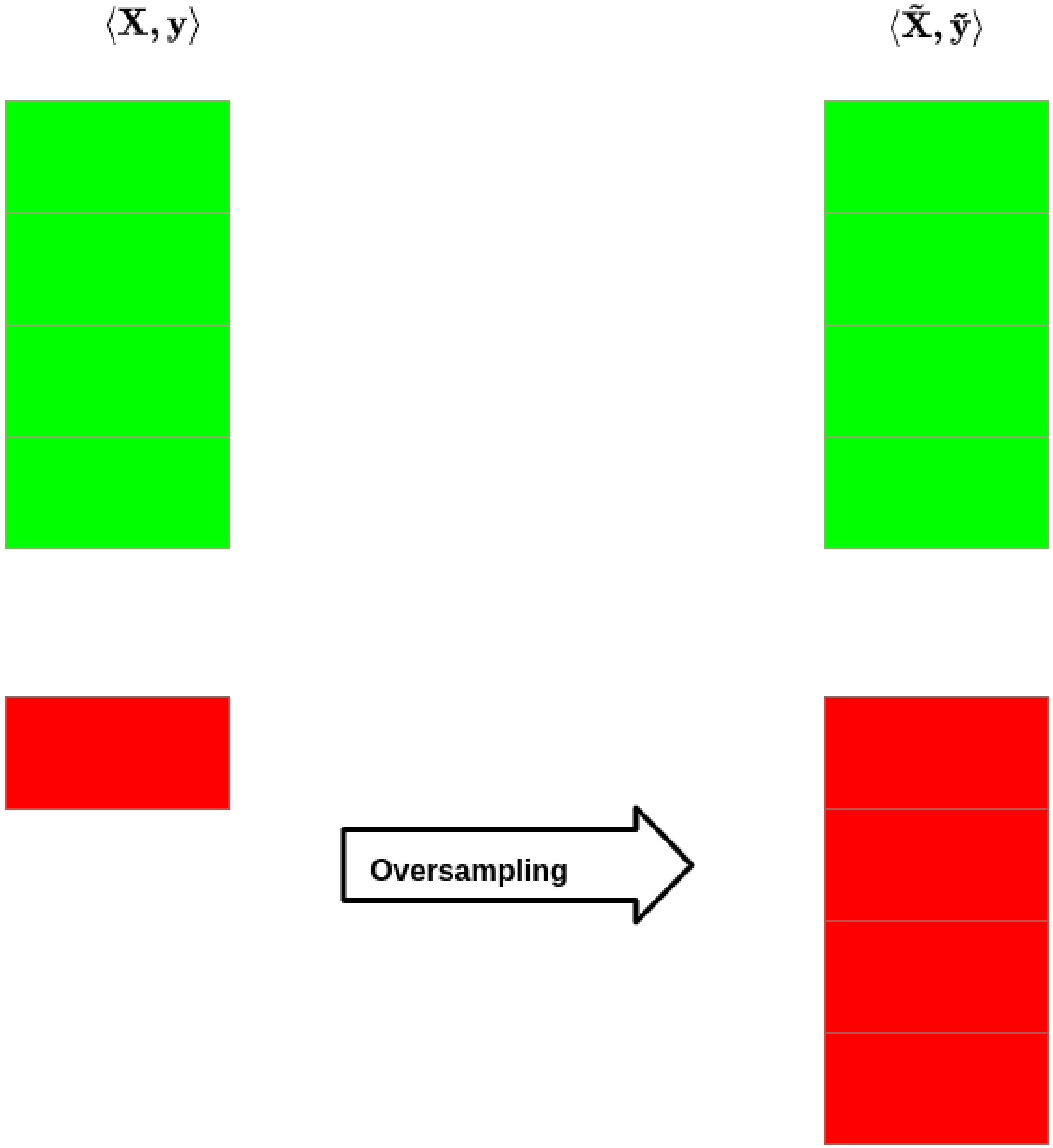
Re-sampling is a process of constructing a new set of training examples by drawing samples from the original.

The sampling doesn't have to be uniform and may be used such that the resampled examples are more balanced

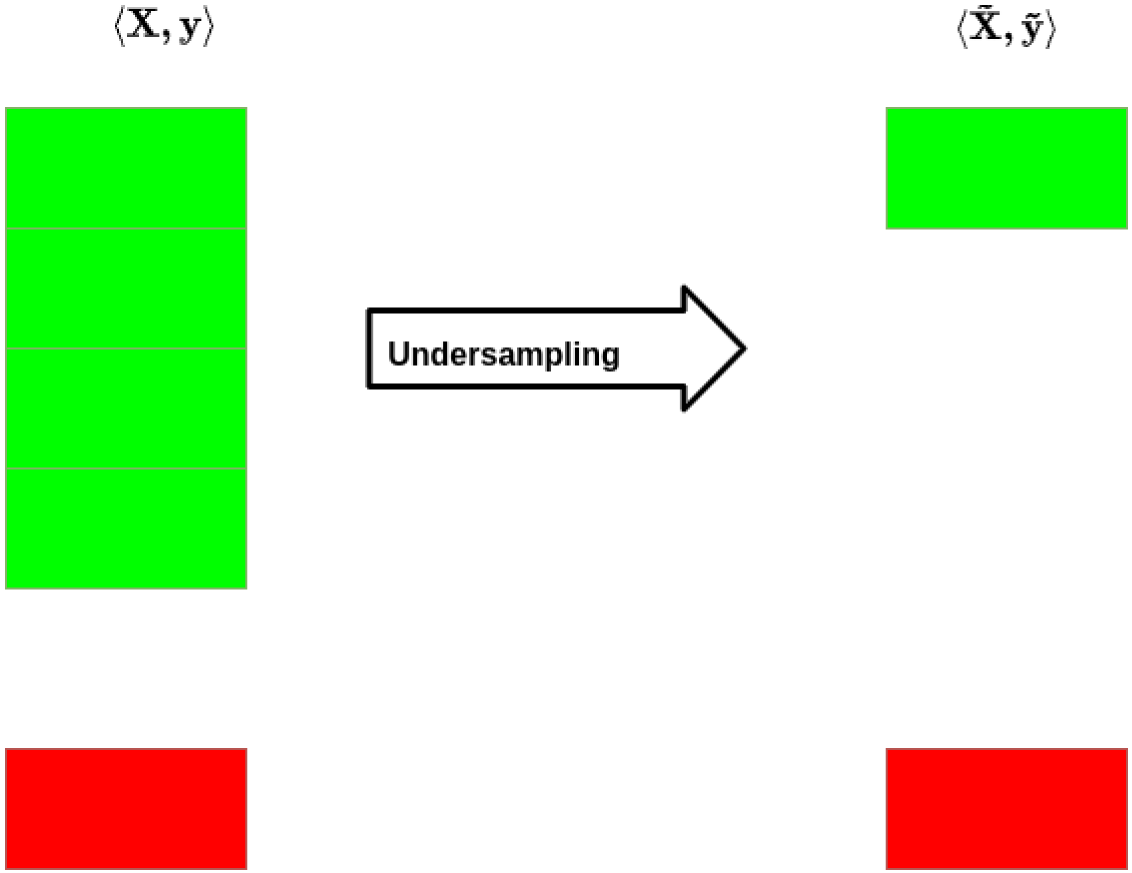
- We can oversample (draw with higher probability) the Minority class
- We can undersample (draw with lower probability) the Majority class

In the limit, weighted sampling is almost equivalent to using the original examples, but weighting the loss term for each class.

Imbalanced data: Oversample the minority



Imbalanced data: Undersample the majority



Caveat: No "cheating" in cross validation, again

Does it make a difference if

- We resample before cross validation ?
- We resample *during* cross validation ?

Yes !

- By resampling before cross validation
    - the fold that is out-of-sample comes from the more-balanced resampled distribution
- By resampling during cross validation
    - Create the folds, including the out of sample fold
    - Resample from the remaining folds to get a more balanced resampled distribution for this iteration

Synthetic examples

Oversampling merely duplicates existing examples from the Minority class.

There are techniques to generate *synthetic* examples

- SMOTE
- ADASYN

A rough description of creating a synthetic example

- Choose an example; find "close" neighboring examples of the same class (Minority)
    - Using a metric of distance between examples
- Create an example that blends/interpolates features from the neighbors into a new example

Imbalanced example

Here's an unbalanced dataset with 2 classes, and the associated predictions of some model.

In [4]:

```
y_true = np.array([0, 1, 0, 0, 1, 0])
y_pred = np.array([0, 1, 0, 0, 0, 1])
```

And the regular and class balanced accuracy

In [5]:

```
print("Acccuracy={a:3.3f}".format(a=accuracy_score(y_true,  y_pred)))
print("Class balanced Acccuracy={a:3.3f}".format(a=balanced_accuracy_score(y_tru
e,  y_pred)))
```

```
Acccuracy=0.667
Class balanced Acccuracy=0.625
```

Here's the math behind the two accuracy computations

- "Regular accuracy": per class conditional accuracy, weight by class fraction as percent of total
- "Balanced accuracy": simple average of per class conditional accuracy

In [6]:

```
# Enumerate the classes
classes = [0,1]

accs, weights = [], []

# Compute per class accuracy and fraction
for c in classes:
    # Filter examples and predictions, conditional on class == c
    cond = y_true == c
    y_true_cond, y_pred_cond = y_true[cond],  y_pred[cond]

    # Compute fraction of total examples in class c
    fraction = y_true_cond.shape[0]/y_true.shape[0]

    # Compute accuracy on this single class
    acc_cond = accuracy_score(y_true_cond, y_pred_cond)

    print("Accuracy conditional on class={c:d} ({p:.2%} of examples) = {a:3.3f}"
.format(c=c,

p=fraction,

a=acc_cond

)
          )

    accs.append(acc_cond)
    weights.append(fraction)

# Manual computation of accuracy, to show the math
acc = np.dot( np.array(accs), np.array(weights) )
acc_bal = np.average( np.array(accs) )

eqn_elts = [ "{p:.2%} * {a:3.3f}".format(p=fraction, a=acc_cond) for fraction, a
cc_cond in zip(weights, accs) ]
```

```python
eqn_bal = " + ".join( eqn_elts )

eqn = "average( {elts:s})".format(elts=", ".join([ str(a) for a in accs ]))
print("\n")
print("Computed Accuracy={a:3.3f} ( {e:s} )".format(a=acc, e=eqn_bal))
print("Computed Balanced Accuracy={a:3.3f} ( {e:s} )".format(a=acc_bal, e=eqn) )
```

```
Accuracy conditional on class=0 (66.67% of examples) = 0.750
Accuracy conditional on class=1 (33.33% of examples) = 0.500


Computed Accuracy=0.667 ( 66.67% * 0.750 + 33.33% * 0.500 )
Computed Balanced Accuracy=0.625 ( average( 0.75, 0.5) )
```

In [7]:

```
print("Done")
```

```
Done
```