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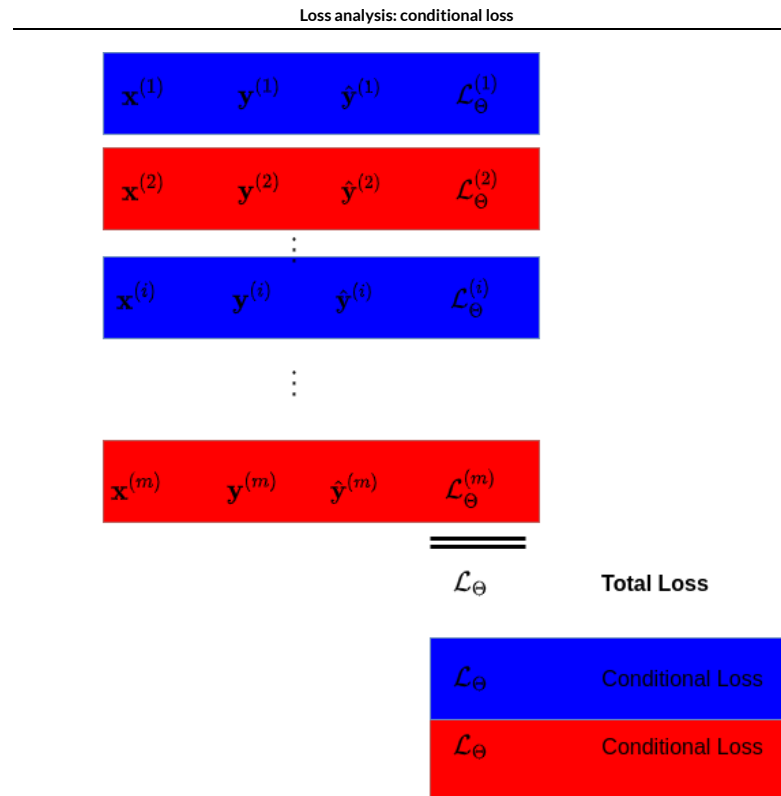
We revisit this in the case of a Binary Classification task, where one class dominates.

Training loss is usually given unconditionally:

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

$$\mathcal{L}_{\Theta} \qquad \textbf{Total Loss}$$

-

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

Loss analysis: conditional loss

$$\mathbf{x}^{(1)} \quad \mathbf{y}^{(1)} \quad \hat{\mathbf{y}}^{(1)} \quad \mathcal{L}_{\Theta}^{(1)}$$

$$\mathbf{x}^{(2)} \quad \mathbf{y}^{(2)} \quad \hat{\mathbf{y}}^{(2)} \quad \mathcal{L}_{\Theta}^{(2)}$$

$$\vdots$$

$$\mathbf{x}^{(i)} \quad \mathbf{y}^{(i)} \quad \hat{\mathbf{y}}^{(i)} \quad \mathcal{L}_{\Theta}^{(i)}$$

$$\vdots$$

$$\mathbf{x}^{(m)} \quad \mathbf{y}^{(m)} \quad \hat{\mathbf{y}}^{(m)} \quad \mathcal{L}_{\Theta}^{(m)}$$

$$\mathcal{L}_{\Theta}$$ **Total Loss**

$$\mathcal{L}_{\Theta}$$ Conditional Loss

$$\mathcal{L}_{\Theta}$$ Conditional Loss

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

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

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.

[The `imbalanced-learn` website (https://imbalanced-learn.org/stable/user_guide.html)](https://imbalanced-learn.org/stable/user_guide.html) documents approaches to this topic.

Conditional Loss

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

Choose a model that is not sensitive to imbalance

Not all models are affected by Imbalanced Data.

- 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

Most models in `sklearn` take an optional `class_weights` optional argument

Allows for different importance (w.r.t. Loss) of different classes

- Can be used to address imbalance
- Can be used to address increased importance (regardless of size) of a particular class

Data Augmentation

We will describe several methods for modifying the training examples.

They usually involve adding examples in order to achieve balance across classes.

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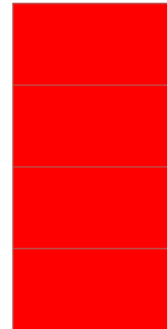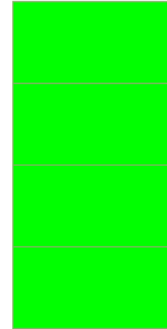- We can undersample (draw with lower probability) the Majority class

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

$\langle \mathbf{X}, \mathbf{y} \rangle$
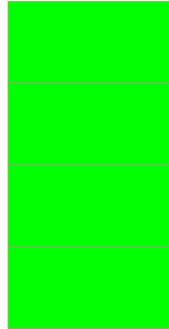
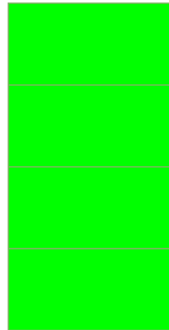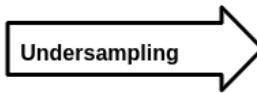$\langle \tilde{\mathbf{X}}, \tilde{\mathbf{y}} \rangle$

Oversampling

Imbalanced data: Undersample the majority

$\langle \mathbf{X}, \mathbf{y} \rangle$

$\langle \bar{\mathbf{X}}, \tilde{\mathbf{y}} \rangle$

Undersampling

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

Technical Issues with augmenting the training examples

We have described several techniques for augmenting the training examples.

- The goal is to make the Loss calculation more sensitive to the under-represented class

We could achieve the same effect by using a weighted Loss function (Loss sensitive training)

- Setting $C_{\text{Positive}} = C_{\text{Negative}}$ will give equal weight to both classes, regardless of the size of each

From a mathematical perspective (w.r.t., Loss) these are equivalent.

But there are important practical differences

Let

- p_{data} denote the true but potentially imbalanced distribution.
- p_{augment} denote the augmented distribution created by either
    - re-sampling from p_{data}
    - create synthetic examples

The following two methods of evaluating the Loss Function should be equivalent

- measured with respect to the augmented training distribution p_{augment}
- measured as a weighted Loss Function using the weights of Loss Sensitive Training

However: the Performance Metric should always be evaluated using the true distribution p_{data}

- assumed to be representative of the out of sample examples to been seen in the future

One therefore must be careful in using p_augment when fitting a model

- the out of sample test dataset must be created *before* augmentation
    - so that it comes from p_data rather than p_augment

- the multiple out of sample proxy datasets created by Cross Validation will be from

    - p_data if we train of p_data and use Loss Sensitive Training to compute a weighted Loss
    - p_augment if we train on p_augment
        - not representative of the true out of sample distribution against which the Performance Metric is computed

-

So, from a practical perspective

- Modifying the Loss function (loss-sensitive training)
- May avoid "side-effects" that can easily arise if we immediately augment

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

```python
# 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:3.3
f}".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) ]
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) )
```

```python
In [7]: print("Done")
```
Done