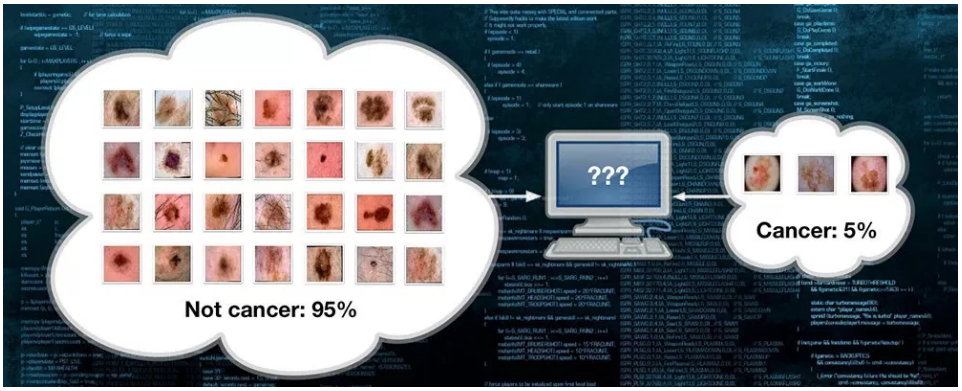

Dealing with Unbalanced Classes in Machine Learning

In many real-world classification problems, we stumble upon training data with unbalanced classes. This means that the individual classes do not contain the same number of elements. For example, if we want to build an image-based skin cancer detection system using convolutional neural networks, we might encounter a dataset with about 95% negatives and 5% positives. This is for good reasons: Images associated with a negative diagnosis are way more common than images with a positive diagnosis. Rather than regarding this as a flaw in the dataset, we should leverage the additional information that we get. This blog post will show you how.

Unbalanced classes create two problems:

1. The accuracy (i.e. ratio of test samples for which we predicted the correct class) is no longer a good measure of the model performance. A model that just predicts "not cancer" everytime will yield a 95% accuracy, even though it is a bad (and even dangerous) model that does not yield any insight or scientific advancement, despite the fact that "95% accuracy" sounds like something good. In addition, it's hard to get an intuition for how good a model with 96%, 97% or 98% accuracy really is.

not cancer, making it hard to further improve the model.

Fortunately, these problems are not so difficult to solve. Here are a few ways to tackle them.

1. Collect more data

If possible, you could collect more data for the underrepresented classes to match the number of samples in the overrepresented classes. This is probably the most rewarding approach, but it is also the hardest and most time-consuming, if not downright impossible. In the cancer example, there is a good reason that we have way more non-cancer samples than cancer samples: These are easier to obtain, since there are more people in the world who haven't developed cancer.

2. Create copies of training samples

Artificially increase the number of training samples for the underrepresented classes by creating copies. While this is the easiest solution, it wastes time and computing resources. In the cancer example, we would almost have to double the size of the dataset in order to achieve a 50:50 share between the classes, which also doubles training time without adding any new information.

3. Create augmented copies of training samples

Similar to 2, but create augmented copies of the underrepresented classes. For example, in the case of images, create slightly rotated, shifted or flipped versions of the original images. This has the positive side-effect of making the model more robust to unseen examples. However, it only does so for the underrepresented classes. Ideally, you would want to do this for all classes, but then the classes are unbalanced again and we're back where we started.

4. (Remove training samples)

Remove training samples from the overrepresented classes so that the number of training samples for all classes is the same. This solves our problem and reduces training time, but it makes our model worse. After all, we want to use as much labelled data as we possibly can, even if this causes unbalanced classes. I don't recommend this solution.

5. Train for sensitivity and specificity

The sensitivity tells us the probability that we detect cancer, given that the patient really has cancer. It is thus a measure of how good we are at correctly diagnosing

$$sensitivity = Pr(detect\,cancer \mid cancer) = \frac{\text{true positives}}{\text{positives}}$$

The specificity tells us the probability that we do not detect cancer, given that the patient doesn't have cancer. It measures how good we are at not causing people to believe that they have cancer if in fact they do not.

$$specificity = Pr(\neg\, detect\,cancer \mid \neg\, cancer) = \frac{\text{true negatives}}{\text{negatives}}$$

A model that always predicts cancer will have a sensitivity of 1 and a specificity of 0. A model that never predicts cancer will have a sensitivity of 0 and a specificity of 1. An ideal model should have both a sensitivity of 1 and a specificity of 1. In reality, however, this is unlikely to be achievable. Therefore, we should look for a model that achieves a good tradeoff between specificity and sensitivity. So which one of the two is more important? This can't be said in general. It highly depends on the application.

If you build a photo-based skin cancer detection app, then a high sensitivity is probably more important than a high specificity, since you want to cause people who might have cancer to get themselves checked by a doctor. Specificity is a little less important here, but still, if you detect cancer too often, people might stop using your app since they unnecessarily get annoyed and scared.

Now suppose that our desired tradeoff between sensitivity and specificity is given by a number $t \in [0, 1]$ where $t = 1$ means that we only pay attention to sensitivity, $t = 0$ means we only pay attention to specificity and $t = 0.5$ means that we regard both to be equally important. In order to incorporate the desired tradeoff into the training process, we need the samples of the different classes to have a different contribution to the loss. To achieve this, we can simply multiply the contribution of the cancer samples to the loss by

$$\frac{\text{number of non-cancer samples}}{\text{number of cancer samples}} \cdot t$$

Keras Implementation

In Keras, the class weights can easily be incorporated into the loss by adding the following parameter to the fit function (assuming that 1 is the cancer class):

```
class_weight={
    1: n_non_cancer_samples / n_cancer_samples * t
}
```

Now, while we train, we want to monitor the sensitivity and specificity. Here is how to do this in Keras. In other frameworks, the implementation should be similar (for instance, you could replace all the K calls by numpy calls).

```
def sensitivity(y_true, y_pred):
    true_positives = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
    possible_positives = K.sum(K.round(K.clip(y_true, 0, 1)))
    return true_positives / (possible_positives + K.epsilon())

def specificity(y_true, y_pred):
    true_negatives = K.sum(K.round(K.clip((1-y_true) * (1-y_pred), 0, 1)))
    possible_negatives = K.sum(K.round(K.clip(1-y_true, 0, 1)))
    return true_negatives / (possible_negatives + K.epsilon())

model.compile(
    loss='binary_crossentropy',
    optimizer=RMSprop(0.001),
    metrics=[sensitivity, specificity]
)
```

Generalizing to more than 2 classes

If we have more than two classes, we can generalize sensitivity and specificity to a "per-class accuracy":

$$perClassAccuracy(C) = Pr(detect\, C \mid C)$$

In order to train for maximum per-class accuracy, we have to specify class weights that are inversely proportional to the size of the class:

```
class_weight={
    0: 1.0/n_samples_0,
    1: 1.0/n_samples_1,
    2: 1.0/n_samples_2,
    ...
}
```

Here is a Keras implementation of the per-class accuracy, which I adopted from jdehesa at Stackoverflow.

```
INTERESTING_CLASS_ID = 0  # Choose the class of interest

def single_class_accuracy(y_true, y_pred):
    class_id_true = K.argmax(y_true, axis=-1)
    class_id_preds = K.argmax(y_pred, axis=-1)
    accuracy_mask = K.cast(K.equal(class_id_preds, INTERESTING_CLASS_ID
    class_acc_tensor = K.cast(K.equal(class_id_true, class_id_preds), '
    class_acc = K.sum(class_acc_tensor) / K.maximum(K.sum(accuracy_mask), 1)
    return class_acc
```

Share this:

Related

Deep Learning From Scratch V: Multi-Layer Perceptrons
August 26, 2017
In "Artificial Intelligence"

Deep Learning From Scratch II: Perceptrons
August 26, 2017
In "Artificial Intelligence"

Deep Learning From Scratch III: Training criterion
August 26, 2017
In "Artificial Intelligence"

By Daniel | September 16th, 2017 | Deep Learning, Keras, Machine Learning, Python | 0 Comments

ALSO ON DEEP IDEAS

Robot Localization III: The Kalman Filter
1 comment • 16 days ago
Pew Pew — These articles are great. I love that you include examples, that's really helpful.

Deep Learning From Scratch III: Training criterion
2 comments • a month ago
Daniel — If you follow the blog post, you will understand how to use TensorFlow as well. Just ignore the

Deep Learning From Scratch: Theory and Implementation
4 comments • a month ago
Daniel — Thanks for the tip, that's what I've done now :)

Deep Learning From Scratch I: Computational Graphs
7 comments • a month ago
thecity2 — Where is `dot()` coming from?