The Effect of Class Imbalance -->>
Model Tuning - Alternate Cutoffs -
Adjusting Prior Probabilities -
Unequal Case Weights

Class imbalance occurs when the distribution of target classes in a dataset is highly skewed, which can negatively affect the performance of machine learning models, particularly classification models. To mitigate this issue, several techniques can be applied, including **model tuning, alternate cutoffs, adjusting prior probabilities, and unequal case weights**. Below, I will explain each technique with numerical examples.

---

1. Model Tuning

Model tuning involves adjusting hyperparameters or modifying the loss function to account for the imbalance.

Example

Consider a binary classification problem where:

- **Class 0 (Negative cases)**: 95% of the dataset (950 instances).
- **Class 1 (Positive cases - Minority Class)**: 5% of the dataset (50 instances).

If we train a logistic regression model without adjustments, it might predict everything as **Class 0** to achieve 95% accuracy, but this would be misleading.

Solution: Hyperparameter Tuning

One way to address this issue is by tuning hyperparameters such as **class weights** in logistic regression or modifying the loss function in deep learning models.

In **Scikit-learn**, we can apply `` `class_weight='balanced'` `` in logistic regression:

```
python


from sklearn.linear_model import LogisticRegression
model = LogisticRegression(class_weight='balanced')
```

The `` `class_weight='balanced'` `` setting assigns **higher weight** to the minority class (Class 1) by calculating:

$$\text{Class Weight} = \frac{\text{Total Samples}}{2 \times \text{Class Frequency}}$$

For Class 1:

$$\text{Weight} = \frac{1000}{2 \times 50} = 10$$

For Class 0:

$$\text{Weight} = \frac{1000}{2 \times 950} = 0.526$$

This forces the model to give more importance to Class 1, improving recall for the minority class.

---

2. Alternate Cutoffs

The decision threshold (default = 0.5 in logistic regression) can be adjusted to favor the minority class.

Example

Assume we train a model, and it outputs the following probabilities for five test cases:

Instance	True Class	Model Output Probability (Class 1)
A	0	0.3
B	1	0.4
C	1	0.6
D	0	0.7
E	1	0.8

If we use a **default threshold of 0.5**, then:

- **Predicted Class (0.5 threshold):** `[0, 0, 1, 1, 1]`
- **Accuracy:** 60%
- **Recall for Class 1: 2/3 = 66.7%**

Lowering the Threshold to 0.4

- **Predicted Class (0.4 threshold):** `[0, 1, 1, 1, 1]`
- **Accuracy:** 80%
- **Recall for Class 1: 100%**

By lowering the threshold, we increase sensitivity towards the minority class at the cost of more false positives.

Implementation in Python:

```
python


from sklearn.metrics import classification_report
y_pred = (model.predict_proba(X_test)[:,1] >= 0.4).astype(int)
print(classification_report(y_test, y_pred))
```

---

3. Adjusting Prior Probabilities

When training models like Naive Bayes, we can explicitly set class priors to reflect the importance of each class.

Example

Consider a dataset where:

- Class 0: 95% (950 instances)
- Class 1: 5% (50 instances)

Naive Bayes assumes equal priors:

$$P(C_0) = P(C_1) = 0.5$$

However, we can adjust them based on real class distribution:

$$P(C_0) = 0.95, \quad P(C_1) = 0.05$$

This adjustment increases the likelihood of Class 1 being chosen.

Implementation in Python (Using Naive Bayes in Scikit-learn):

```
python


from sklearn.naive_bayes import GaussianNB
model = GaussianNB(priors=[0.95, 0.05])
```

---

4. Unequal Case Weights

Assign different importance to misclassification errors using weighted loss functions.

Example

Suppose we use a **cost-sensitive approach** in a **confusion matrix**:

Actual / Predicted	Predicted Class 0	Predicted Class 1
Actual Class 0	**TN = 900**	**FP = 50**
Actual Class 1	**FN = 30**	**TP = 20**

If **False Negatives (FN)** are costlier than **False Positives (FP)** (e.g., in medical diagnosis), we assign different penalties:

- **Cost of FN (missing a positive case) = 5**
- **Cost of FP (wrongly classifying negative as positive) = 1**

Total misclassification cost:

$$\text{Cost} = (30 \times 5) + (50 \times 1) = 150 + 50 = 200$$

To minimize this cost, models can be trained with **weighted loss functions** like:

$$\text{Weighted Loss} = \sum w_i \cdot \text{Loss}_i$$

Implementation in Python (Random Forest with Class Weights):

```
python


from sklearn.ensemble import RandomForestClassifier
model = RandomForestClassifier(class_weight={0: 1, 1: 5})
```

This forces the model to **prioritize correctly classifying Class 1**, improving recall for the minority class.

Conclusion

Technique	Effect
Model Tuning	Adjusts class weights or modifies loss function
Alternate Cutoffs	Adjusts decision threshold to favor minority class
Adjusting Prior Probabilities	Alters priors to reflect real-world distributions
Unequal Case Weights	Assigns higher misclassification cost to the minority class

Each technique **helps mitigate class imbalance**, and a combination of these approaches is often most effective in real-world scenarios.

@ S S Roy. 23rd March,2025