A Study on the Effects of Representation Bias on AI Performance and Methods of Mitigating it

Sarah Abdelazim, Morris Chan and Julie Song

2023-06-26

Contents

1 Executive Summary

Representation bias has been a long-standing problem in machine learning. With imbalanced datasets, machine learning models can often only give accurate predictions in large classes but not in classes with smaller sample sizes. A Python package, `debiaser` was created to implement various methods to mitigate data imbalance issues in classification problems. The package includes a general pipeline for model training and hyperparameter optimization and supplementary functions for model comparison and visualization. Methods were tested against baseline models with a general dataset. Performances of each model are compared in terms of newly proposed class-balanced weighted metrics. Weighted random sampling and MaxNorm regularization were found to be the best methods with the dataset used.

2 Introduction

We are partnered with ALS GoldSpot Discoveries Ltd. in this capstone project. ALS Goldspot Discoveries Ltd. is a technology company that uses data science and machine learning techniques for better strategy planning in resource exploration and mining. One of the main tasks of resource exploration is to identify the nature of possible mining sites, for potential resources and risks. This can be done by drilling core samples for tests. In one example, there are samples of rock images with different levels of fracture severity. Accurately identifying the fracture severity will guide decision-making on whether the sites are safe for launching mining projects. Underestimating the rock fracture severity level could lead to serious injury and property damage, while overestimating it could result in opportunity losses. Any chosen model must be able to make accurate predictions on all classes in the dataset. However, accurately predicting images from all classes is challenging because of representation bias.

Representation bias arises when the observed data distribution is skewed dramatically in favor of some common classes. Traditional, cost-insensitive CNNs are susceptible to representation bias because they only optimize global accuracy and assume approximately equal sample sizes per group by assigning the same learning rate to each target class (Huang et al. 2022). Hence, these models will have difficulties classifying minority classes because they cannot effectively learn the decision rules for the rarer observations. This poses a problem when the specific error rates associated with each target class must be considered, along with the global accuracy of the model. Furthermore, traditional models also neglect the fact that misclassifying different classes may have different consequences in real life situations. The importance of accuracy in each individual class highlights the importance of mitigating representation bias.

In our project, a Python package has been built to allow users to train and compare PyTorch neural network models for a wide variety of classification tasks, with the objective of determining the best method(s) for effectively tackling imbalanced datasets. An initial literature review revealed two main approaches: data-level methods and algorithm-level methods. Data-level methods focus on resampling the original images or creating synthetic data for training, while algorithm-level methods focus on the architecture of classifiers, including training loss functions and parameter regularization. The best methods will be identified based on a mix of metrics including overall accuracy and error rates per class. The final deliverable will be a Python package called `debiaser` which will include functions and methods for:

- Augmentation and processing of data,
- Selection of pretrained models,
- Implementation of loss functions,
- Application of optimization algorithms and parameters regularization,
- Evaluation of models

3 Data Science Methods

3.1 Our Package `debiaser`

Our solution to mitigate the problem of data imbalance is to build a generalizable Python package that encompasses multiple debiasing methods and provides interpretable results for users to check and improve their models with.

The main functions for the package are included in the module `general`. Module `general` provides wrapper functions to preprocess the data using PyTorch's Transformer, load images using PyTorch's DataLoader, load pretrained models, load loss functions, load optimization algorithms, and train and evaluate models' performance. This module forms the basic pipeline of each experimental run: Users will pick the transforming procedures, sampling methods, the model to be used, the criterion and optimizer sequentially, and then train and evaluate the model (Table 1, Figure 1).

Table 1: Fundamental debiaser functions.

Function	Task	Options
transformer()	To load a PyTorch Transformer	Image size, normalization, flipping, rotation, cropping
load_image()	To load a PyTorch DataLoader	Batch size, sampling strategy
load_model()	To load a pretrained model	Pretrained model (ResNet18, MobileNet, EfficientNet)
load_loss()	To load a loss function	Loss function, beta (for class-balanced losses), gamma (for focal loss)
load_optimizer()	To load an optimizer	Learning rate, optimizer, parameters regularization strategy, weight_decay
train()	To train and evaluate models	Criterion, optimizer, learning rate scheduler, gradient clipping, patience for early stopping

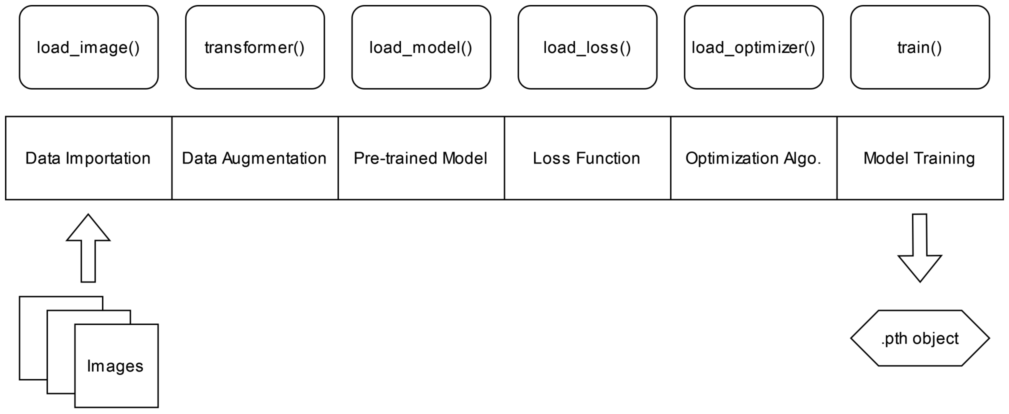

Figure 1: Basic functions in the package `debiaser`. Basic functions in `debiaser` include all steps of a general CNN training workflow: from loading and transforming the images to training the model with the selected methods and hyperparameters. The pipeline will take images as inputs and return a trained PyTorch model and the results as outputs.

3.2 Data Augmentation: `transformer()`

Before loading the data, it is necessary to decide the data augmentation steps. Data augmentation methods including normalization, rotation, and flipping should be included during model training to allow the models to be more robust to variations in the dataset.

```
load_transformer(image_size=224,
                 rotate=True,
                 degrees=30)
# All the codes shown in the report are just for demonstration.
# Refer to the package documentation for more comprehensive usages.
```

3.3 Data Importation: `load_image()`

When loading the data for model training, we can applying resampling schemes as the first level of debiasing methods.

```
load_image(batch_size=32,
           sampling_strategy='oversampling')
```

Resampling strategies will be used in during data importation. Three resampling strategies were proposed. These strategies aim to make datasets balanced before training the model (Figure 2. **`Weighted random sampling`** assigns probability weight to each sample based on their class's sample size. In other words, samples from the large classes will have lower probability to be drawn for the final training set. Samples from small classes will have higher probability to be included. **`Oversampling`** will create duplicates for samples from smaller classes so that all classes will have the same number of samples as the largest class in the final training set. **`Undersampling`** works similarly with all classes having the same number of samples as the smallest class in the final training set.

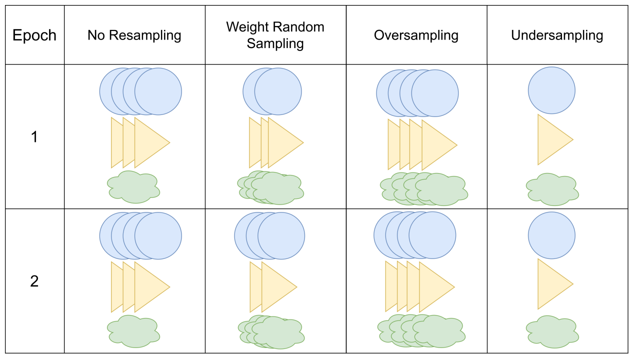

Figure 2: Resampling methods implemented in the package.

3.4 Transfer Learning: `load_model()`

Three pre-trained CNN models (ResNet-18, MobileNet, and EfficientNet) are proposed for the image classification tasks. Models will be fine-tuned with the given data set.

```
load_model(model_id='resnet18',
           n_class=4)
```

3.5 Loss Function Implementation: `load_loss()`

Loss functions are necessary for model training, in which the optimization of models' parameters are guided by the minimization of the loss scores. Inappropriate use of loss function is proposed to be one of the reason why models are biased towards samples in larger classes. Thus, we create a function for users to choose which loss functions to use.

```
load_loss(loss_family='CB',
          loss_type='CE')
```

Traditional loss functions, such as cross entropy loss for multiclass classification, are often biased towards to larger classes. The bias is because there are more entries in large classes and they have more contribution to the final loss scores. Class-balanced loss are proposed to address this issue (Alshammari et al. 2022). Class-balanced functions consider each class's effective number of samples (Cui et al. 2019). Effective number of samples is another representation of sample size based on the assumption that some observations will carry similar or overlapping information and can be calculated by the following formula: $\frac{1-\beta^n}{1-\beta}$, where $\beta\epsilon(0,1]$. This is because as sample size increases, the additional benefit of a new data point diminishes due to the higher chance of it carrying overlapping information. $\beta = 0$ corresponds to no re-weighting and $\beta = 1$ corresponds to re-weighing by inverse class frequency, allowing a smooth adjustment of the class-balanced term between no re-weighting and re-weighting by inverse class frequency.

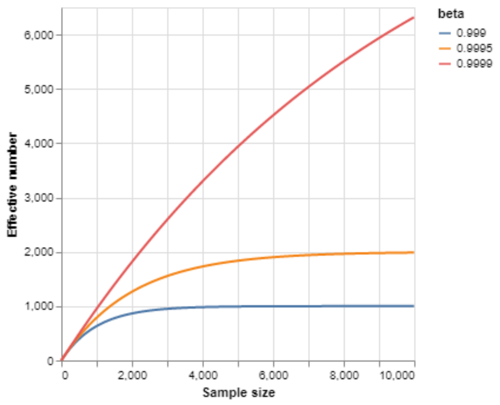

Figure 3: β controls how fast the effective number of samples grows. Given similar sample sizes, a smaller β results in a lower effective number of samples than a larger β.

This weighing term was then implemented in four kinds of loss functions: the `cross entropy with a softmax layer`, the `binary cross entropy with a softmax layer`, the `binary cross entropy with a sigmoid layer` and the `focal loss`. The `focus loss` is simply the sigmoid binary cross entropy function with a modulating factor. The modulating factor, which is controlled by the hyperparameter γ, reduces the relative loss for well-classified samples and focuses on difficult samples. Note that the use of binary cross entropy for multi-class classification was used to experiment with a one-vs-all implementation. The difference between the implementation of the softmax and the sigmoid layers is that the latter does not assume mutual exclusivity of classes and is more applicable to real life datasets (Cui et al. 2019).

A different kind of loss function, the `mean false error` (`MFE`) was adapted from Wang et al. (2016) and does not require any hyperparameter tuning. It has been derived from the `mean squared error` (`MSE`) loss which is calculated by summing up all the errors from the whole data set and then taking the average of that. However, this works well when the classes are balanced and if used on imbalanced datasets, will be biased towards the majority class. However, MFE loss calculates the average error in each class separately - by breaking them down into false positive error (FPE) and false negative error (FNE) - and then adds them together. This procedure takes into account the differences in sample sizes between the majority and minority classes. The mean squared false error (MSFE) loss further improves the performance of MFE by introducing more penalties to large errors (Wang et al. 2016).

3.6 Optimization Algorithm & Parameters Regularization: `load_optimizer()`

Optimization algorithms determine how a neural network can learn and adjust through each iteration of trial and error. For the package to effectively run on different datasets, we include multiple optimizers in the PyTorch ecosystem. Besides, we added parameters regularization methods into these optimizers for debiasing purposes.

```
load_optimizer(optimizer_name='Adam',
               strategy='weight_decay')
```

Parameters regularization is motivated by the observation that neural network models tend to have larger weights for classifying neurons corresponding to the large classes. One way to make the weights for all classes more balanced is to use parameters regularization. Three regularization strategies are proposed. `Weight Decay` penalizes larger weights more compared to smaller weights. This encourages more balanced weight growth as the model trains, and improves generalization of the model for minority classes. `L2 Normalization` directly imposes a restriction that forces each class's weight matrix to have a norm of one. This improves model performance for the minority class but sacrifices performance for the majority class. `MaxNorm` imposes a maximum threshold on the weight norm for each class. Weights are free to grow until their norm reaches this maximum threshold value. This threshold value also requires optimization, as it varies depending on the dataset (Srivastava et al. 2014). Literature shows that using both the `MaxNorm` and `Weight Decay` methods together have the best results, although this can vary for different datasets (Alshammari et al. 2022).

3.7 Model Training and Evaluation: `train()`

The final step of our general pipeline will take all the output of the above functions and return the model performance of the specified methods.

3.8 Class-balanced Weighted Metrics

To decide which the best methods are, a new formula is designed for summarizing model performance. Precision and recall are originally designed for individual classes. Existing averages of precision and recall include macro average and weighted average. Suppose m_i is the precision or recall score for the ith class and c is the number of classes. Macro average is calculated by dividing the sum of the scores by c as $\frac{\sum_i^c m_i}{c}$. Weighted average, on the other hand, considers sample size per class and weighs the scores by the class's sample size as $\frac{\sum_i^c m_i \times n_i}{\sum_i^c n_i}$. Unfortunately, both macro average and weighted average are biased towards larger classes. The performance of smaller classes will be masked as more attention is given to the larger classes.

Class-balanced weighted average weighs each class's score by the inverse of the sample size to place more importance on the smaller classes:

$$\frac{\sum_i^c m_i \times n_i^{-1}}{\sum_i^c n_i^{-1}}$$

.

This will help users to better perceive the performance of the tested models.

4 Data Product

4.1 General Pipeline

Since the workflow pipeline involves numerous variables and functions that will be run sequentially, all variables are stored in a dictionary. This dictionary is passed sequentially from the pipeline's first function `transformer` to the last function `train` and records the variables in each step. In the final step, `train` creates a JSON file that stores all performance metrics and variables used in the process for users to compare different methods and parameters.

4.2 Hyperparameter Optimization

Several of the debiasing methods related to the optimizer and the loss function involve hyperparameters that need to be optimized, as shown in Table 2. Parameters of these sections as well as of the `train()` function can be adjusted for the hyperparameter optimization process as well.

Table 2: Optimizable Parameters.

Field	Options
Optimizer	Learning rate, Optimizer, Regularization strategy and associated parameters
Critreion	Loss family, Loss type, Associated parameters
Training	Number of epochs, Learning rate scheduler, Gradient clipping value, Patience tolerance

Hyperparameter optimization can be performed with K-fold cross-validation using the `manual_opt()` function. This function requires the outputs of the `transformer()` and `load_model()` functions from the basic workflow, as well as a dictionary containing the parameters to be optimized. The specific syntax can be found in the documentation. The function directly returns a dictionary with all of the cross-validation results, which can be used for other workflows in the same working session. This information is also stored as a JSON file. The results from each cross-validation iteration are also stored together as JSON files, inside a separate folder each time the function is run.

Users can choose a metric to obtain the "best" cross-validation results by the mean validation accuracy, or the mean validation precision or recall of the minority class. They can also visualize the results of the cross-validation using the `plot_cv_results()` function.

4.3 Model Comparison & Visualizations

The package `debiaser` also encompasses a workflow for model comparison. `read_log` from the module `utils` take multiple JSON log files are return a `csv` file with all the variables and parameters stored. With the `CSV` file, the module `visualization` supports the creation of basic figures including bar charts and scatter plots. Scatter plot products are shown in the next section for demonstration.

5 Results

5.1 Data

Due to time constraints, we are not able to optimize the baseline model with the dataset shared by our industrial partner, To develop our package, we used a dataset from https://www.kaggle.com/datasets/kausthubkannan/5-flower-types-classification-dataset, which it's images are expected to have simpler features for the models to learn. We artificially made the dataset imbalance by removing images manually in each class (Table 3). All of the results presented in the report are yielded from this dataset.

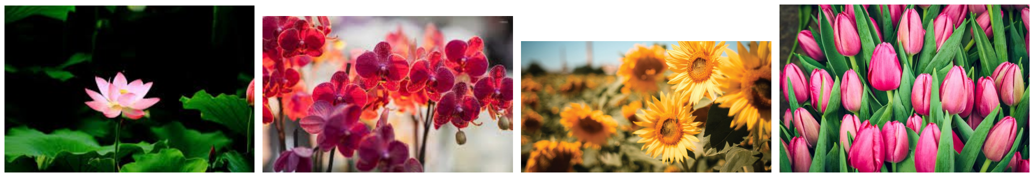

Figure 4: Example images of the dataset used in development. From left to right: Lotus, Orchid, Sunflower, Tulip

Table 3: Each Class's Sample Size for Training and Validation

Class	Training	Validation
Lotus	750	250
Orchid	425	150
Sunflower	250	75
Tulip	75	25

5.2 Baseline Model

A valid baseline model is necessary to provide a meaningful control for testing the proposed debiasing methods. ResNet-18 was chosen as the base model since it outperformed MobileNet and EfficientNet on the imbalanced datasets that were tested. The model was fine-tuned with the flowers dataset, using Adam as the optimization algorithm and a learning rate of $1e-5$ for 50 epochs. All images were resized into 224x224 pixels and normalized. Rotation with $\pm 30°$ is added randomly to prevent overfitting. No other augmentation was used. A batch size of 32 was used.

Table 4: Performance of the Baseline Model

Method	Accuracy	Weighted Precision	Weighted Recall
Baseline	95.9%	92.06%	81.41%

5.3 Resampling Methods

Performances of resampling methods are first compared (Table 5, Figure 5). Oversampling and weighted random sampling yielded lower validation loss and higher accuracy than the baseline model without any re-sampling. While the weighted recall scores are similar between oversampling and weighted random sampling, weighted random sampling has a higher weighted precision.

Table 5: Change in Performance with Different Resampling Methods

Method	Accuracy	Weighted Precision	Weighted Recall
Oversampling	1.76%	1.91%	10.45%
Weighted RS	1.05%	3.78%	10.12%
Baseline (no resampling)	0.00%	0.00%	0.00%
Undersampling	-6.29%	-28.30%	11.05%

Note:
All changes are shown in percentage point. Abbreviation: RS - random sampling.

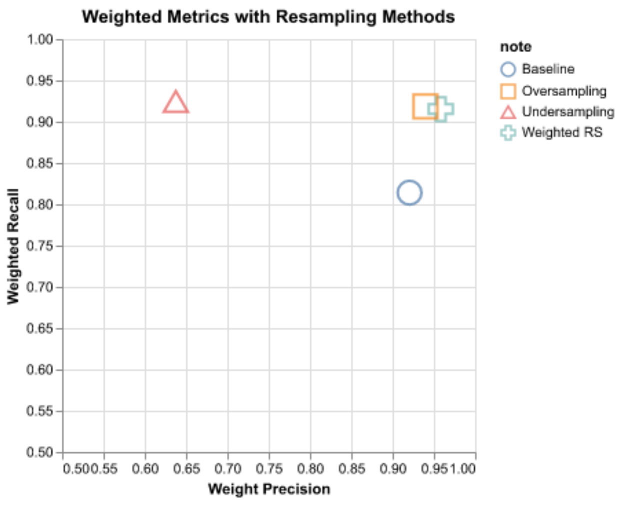

Figure 5: Weighted precision and recall with different resampling methods. All resampling methods work better than the baseline in terms of the weighted recall. Weighted random sampling and oversampling work the best with the highest weighted precision. The scatter plot is created using the function `viz_scatter()` in the `visualization` module from the package. Abbreviation: *RS* - random sampling.

5.4 Loss Functions

Performance of different loss functions are compared in Table 6 and Figure 6. Accuracy is similar across all loss functions. From Figure 6, the mean squared false error, class-balanced cross entropy loss, and binary cross entropy (sigmoid) appear to perform the best in this data set. Mean squared false error has the best recall but the worst precision. In contrast, binary cross entropy (sigmoid) has the best precision but the worst recall. It is difficult to conclude which is the best method based on the results as the decision should depend on the context of the business problem.

Table 6: Change in Performance with Different Loss Functions

Method	Accuracy	Weighted Precision	Weighted Recall
CB BCE (sigmoid)	0.58%	6.58%	2.64%
CB BCE (softmax)	0.39%	6.33%	2.43%
CB cross entropy	0.19%	-3.54%	7.33%
Baseline (cross entropy)	0.00%	0.00%	0.00%
Focal loss	0.00%	6.28%	-0.18%
MFE	0.00%	-4.24%	4.84%
MSFE	-1.95%	-12.05%	7.18%

Note:
All changes are shown in percentage point. Abbreviation: CB - class-balanced, BCE - binary cross entropy, MFE - mean false error, MSFE - mean squared false error.

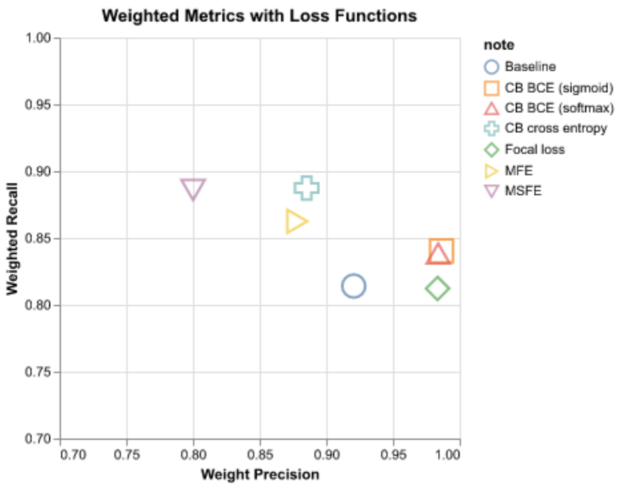

Figure 6: Weighted precision and recall with different loss functions specialized for imbalanced data. Mean squared false error yields the highest weighted recall score, while class-balanced cross entropy yields the highest weighted precision. The scatter plot is created using the function `viz_scatter()` in the `visualization` module from the package. Abbreviation: *CB* - class-balanced, *BCE* - binary cross entropy, *MFE* - mean false error, *MSFE* - mean squared false error.

5.5 Parameters Regularization

Performances of parameters regularization methods are compared in Table 7 and Figure 7. All methods have comparable accuracy. However, MaxNorm yields the best class-balanced weighted precision and recall scores.

Table 7: Change in Performance with Different Parameters Regularization Methods

Method	Accuracy	Weighted Precision	Weighted Recall
MaxNorm	0.82%	6.44%	7.83%
Weight decay	0.78%	6.27%	0.43%
Baseline (no regularization)	0.00%	0.00%	0.00%
L2 normalization	0.00%	0.00%	0.00%

Note:
All changes are shown in percentage point.

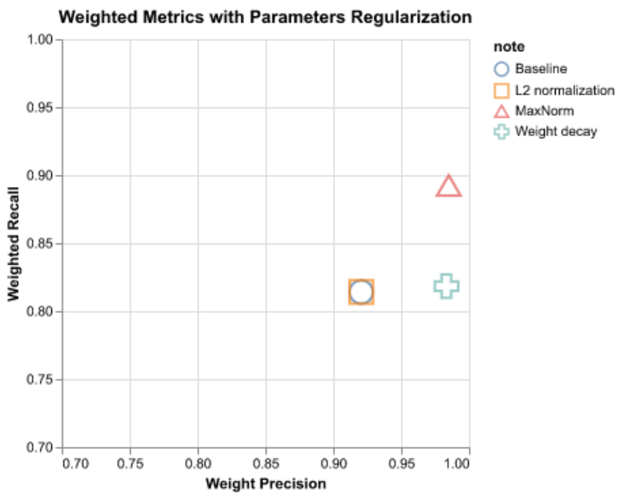

Figure 7: Weighted precision and recall with different parameters regularization methods. Both weight decay and MaxNorm yield better weighted precision than the baseline model, while MaxNorm also achives a better weighted recall score. The scatter plot is created using the function `viz_scatter()` in the `visualization` module from the package.

5.6 Selected Methods

To better identify the best methods for the data set, we further compared the performances of the best methods in each group of debiasing methods. In terms of both accuracy and class-balanced weighted metrics, weighted random sampling and MaxNorm performed the best with this data set (Table 8, Figure 8).

Table 8: Change in Performance with Different Debiasing Methods

Method	Accuracy	Weighted Precision	Weighted Recall
Weighted RS	1.05%	3.78%	10.12%
MaxNorm	0.82%	6.44%	7.83%
CB BCE (sigmoid)	0.58%	6.58%	2.64%
CB BCE (softmax)	0.39%	6.33%	2.43%
CB cross entropy	0.19%	-3.54%	7.33%
Baseline	0.00%	0.00%	0.00%
MSFE	-1.95%	-12.05%	7.18%

Note:
All changes are shown in percentage point. Abbreviation: RS - random sampling, CB - class-balanced, BCE - binary cross entropy, MFE - mean false error, MSFE - mean squared false error.

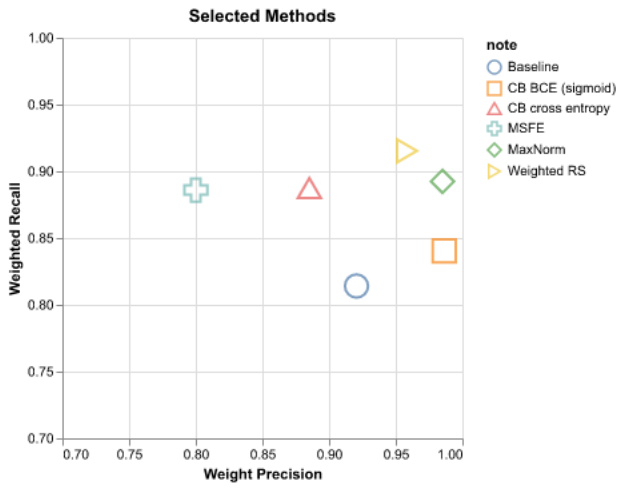

Figure 8: Weighted precision and recall with different parameters regularization methods. Combining methods from all levels, weighted random sampling and MaxNorm yield the best weighted precision and weighted recall scores. The scatter plot is created using the function `viz_scatter()` in the `visualization` module from the package. Abbreviation: *RS* - random sampling, *CB* - class-balanced, *BCE* - binary cross entropy, *MFE* - mean false error, *MSFE* - mean squared false error.

6 Conclusions and Recommendations

Methods to handle data imbalance have been demonstrated with this package. For the dataset that was used to develop the package, weighted random sampling was found to be the most effective debiasing method. Still, this conclusion is not universal as the performance of different methods will depend on the nature of the data sets.

Given the time constraint, it is impossible for the package to include every single model in the image classification literature. Thus, users are recommended to add any pretrained models or supplementary methods to the package for their specific data problems. Some projects have also used multi-stage learning for imbalanced data sets (e.g. Alshammari et al. 2022), while this package only enables single-stage learning by fine-tuning pretrained models with the data set once. More advanced training pipelines can be included to further expand the package's functionalities.

References

Alshammari, Shaden, Yu-Xiong Wang, Deva Ramanan, and Shu Kong. 2022. "Long-Tailed Recognition via Weight Balancing." https://arxiv.org/abs/2203.14197.

Cui, Yin, Menglin Jia, Tsung-Yi Lin, Yang Song, and Serge Belongie. 2019. "Class-Balanced Loss Based on Effective Number of Samples." https://arxiv.org/abs/1901.05555.

Huang, Zhan ao, Yongsheng Sang, Yanan Sun, and Jiancheng Lv. 2022. "A Neural Network Learning Algorithm for Highly Imbalanced Data Classification." *Information Sciences* 612: 496–513. https://doi.org/https://doi.org/10.1016/j.ins.2022.08.074.

Srivastava, Nitish, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting." *Journal of Machine Learning Research* 15 (56): 1929–58. http://jmlr.org/papers/v15/srivastava14a.html.

Wang, Shoujin, Wei Liu, Jia Wu, Longbing Cao, Qinxue Meng, and Paul J. Kennedy. 2016. "Training Deep Neural Networks on Imbalanced Data Sets." In *2016 International Joint Conference on Neural Networks (IJCNN)*, 4368–74. https://doi.org/10.1109/IJCNN.2016.7727770.