Imbalanced Data Classification with Neural Networks and Classifiers

Andrea Boskovic

Submitted to the Department of Mathematics and Statistics in partial fulfillment of the requirements for the degree of Bachelor of Arts with honors.

Faculty Advisor: Professor Amy Wagaman

May 17, 2021

Acknowledgements

Writing this thesis would not have been possible without the support and mentorship of people in the Amherst College community. I appreciate everyone who helped me throughout this process.

First, I'd like to thank my thesis advisor, Amy Wagaman. She has supported me throughout my academic career at Amherst and through the many months of thesis work. Her thoughtfulness has shaped my approach to problem-solving, and her kindness has kept me balanced and focused in the research process. I would also like to thank Nick Horton for his dedicated, energetic mentorship and support throughout my time at Amherst.

Next, I'd like to express my gratitude toward the Amherst College Mathematics and Statistics Department and all the professors for their dedication to teaching and supporting students. Thanks to Amalia Culiuc and Ryan Alvarado for teaching me how to think critically in math. Thanks to Kat Correia and Kevin Donges for their caring and compassion.

Finally, I'd like to thank my friends from home for all the virtual laughs. Thanks to Alison Ortiz-Dimas and Jonah Botvinick-Greenhouse for helping to take my mind off of work with long conversations and juggling. Thanks to Justin Vernon, The Beatles, and Freddie Gibbs for providing a soundtrack to my thesis, and thank you to the producers of Young Sheldon for creating a thesis break show. Special thanks to Jamie Mazzola for his aesthetic consultations on my graphs, for keeping me energized with jumping jacks and midnight waffles, and for inspiring me throughout this process. Thank you to my parents for always making me smile and supporting me over the course of my academic career.

Abstract

In statistics, we often want to predict a response variable based on data. Binary classification is one example of this setting where the response variable takes on two possible values. Classification techniques then aim to classify this response, also known as the class, based on data in a way that maximizes accuracy. We are particularly interested in the classification of imbalanced data, a common data type in medical settings and fraud detection, where the number of instances in each class drastically differs. Canonical classification methods, such as classifiers and neural networks, often perform poorly on these imbalanced datasets. We show that lower imbalance levels, where the disparity between the number of instances in each class is large, affect the performance of harder classification tasks more than easier classification tasks. Through this investigation of how imbalance levels in both synthetic and real-world datasets affect classification performance, we can better understand how to mitigate this issue.

Table of Contents

1 Introduction **2**

2 Classification Techniques **5**

2.1 Classifiers . . . 5

2.1.1 Gaussian Naive Bayes . . . 6

2.1.2 Random Forests . . . 7

Algorithm . . . 8

2.1.3 Gradient Boosting . . . 9

Boosting . . . 9

Gradient Descent . . . 9

2.2 Neural Networks . . . 10

2.2.1 Passing Data . . . 12

Forward Pass . . . 12

Backpropagation . . . 13

Hyperparameters . . . 13

2.2.2 Regularization . . . 13

L_2 Regularization . . . 14

L_1 Regularization . . . 15

2.2.3 Convolutional Neural Networks . . . 15

Structure . . . 16

Implementation . . . 17

3 Class Imbalance **18**

3.1 Introduction . . . 18

3.2 Types of Imbalance: Binary and Multi-Class . . . 20

3.3 Techniques for Mitigating Class Imbalance . . . 21

3.3.1 Data Level Methods . . . 21

3.3.2 Algorithm Level Methods . . . 22

3.4 Metrics . . . 23

Sample Classifier Results . . . 25

3.5 Generating Imbalanced Data 26
3.5.1 Theoretical Background 26
3.5.2 Code Background 28
3.5.3 Function 29
3.5.4 Baseline Simulation Tests 30
Easy Classification Problem 31
Difficult Classification Problem 32
3.5.5 Investigating the Effects of Imbalance Levels 32
Classifiers 33
Neural Networks 34
Difficulty over All Imbalance Levels 36
Simulation Study 38
Conclusions 39

4 Imbalanced Data Classification **41**
4.1 Image Classification Background 41
4.1.1 Intel Images Dataset 42
4.2 Image Data Experiments 43
4.2.1 Experiment Structure 44
Preprocessing 44
Neural Network Structure 46
4.2.2 Results 46
Easy Classification Problem 47
Hard Classification Problem 48
Changing Imbalance Levels 48
4.2.3 Conclusions 50
4.3 Numerical Data Experiments 50
4.3.1 Experiment Structure 50
4.3.2 Results 51
Comparison to Generated Data Results 53
4.3.3 Conclusions 53

5 Loss Function Development **54**
5.1 Loss Functions 55
5.1.1 Focal Loss 55
5.1.2 Label-Distribution-Aware Margin (LDAM) Loss 55
5.1.3 Gradient Harmonizing Mechanism (GHM) Loss 56

5.1.4 Mean False Error (MFE) 57
5.2 Methods 58
5.2.1 Dataset 58
5.2.2 Neural Network 59
Dense Layer 59
Batch Normalization Layer 59
Dropout 59
Structure 60
5.2.3 Implementation 60
Loss Functions 61
5.3 Results 61
5.4 Conclusions 62

6 Conclusion **63**

Appendix A Synthetic data **69**
A.1 Synthetic Data Classification 69
A.1.1 Synthetic Data Test: Classifiers 69
Generating Classifier Results Figure 72
A.1.2 Synthetic Data Test: Neural Networks 74
Generating Neural Network Results Figures 79
A.2 Varying Imbalance levels 81
A.2.1 Single-Attempt Classification 81
Mathematical Process 81
Function to Test Imbalance 81
A.2.2 Simulation Study 84

Appendix B Intel Images **89**
B.1 Determine Difficulty of Classification Tasks 89
B.2 Neural Networks 91

Appendix C Kaggle Credit Card Fraud **97**
C.1 Comparison of Accuracy and Balanced Accuracy 97
C.2 Classification at Selected Values of ρ 98
C.3 Performance over All Imbalance Levels 101
C.3.1 Single-Attempt Classification 101
C.3.2 Simulation Study 103

Appendix D New Techniques 106

D.1 State of the Art Loss Functions 106

D.1.1 Focal Loss 106

D.1.2 Gradient Harmonizing Mechanism (GHM) Loss 107

D.1.3 Label-Distribution-Aware Margin (LDAM) Loss 109

D.2 Loss Function Experiments 110

List of Figures

2.1 Binary Confusion Matrix 6
2.2 Boosting 9
2.3 Gradient Descent Visualization 10
2.4 Inputs to a Perceptron 11
2.5 Three-layer Neural Network 11
2.6 Overfitting 14
2.7 Convolutional Neural Network Example 16
2.8 Convolution and Max Pooling Operations 17

3.1 Comparison of Multi-Class and Binary Target Variables 20
3.2 Easy Classification Task with Synthetically Generated Data 27
3.3 Difficult Classification Task with Synthetically Generated Data 28
3.4 Effects of Difficulty, Resampling, and Imbalance Level on Generated Data . 34
3.5 Neural Network Performance with and without Resampling 36
3.6 Classifier Performance over All Imbalance Levels 37
3.7 Classifier Performance Simulation over All Imbalance Levels 39

4.1 Sample Image from Intel Images Dataset 42
4.2 Principal Component Analysis (PCA) 43
4.3 Intel Images Imbalance Level Performance 49
4.4 Discrete Imbalance Levels Kaggle Credit Cards Dataset 52
4.5 All Imbalance Levels Kaggle Credit Cards Data 52
4.6 Simulation Study of Kaggle Credit Cards Data 53

5.1 Illustration of LDAM Loss Function 56
5.2 Neural Network Implementation Process 61

B.1 Example of Loss Function and Accuracy Convergence during Training . . . 92

List of Tables

3.1 Sample Binary Classification Cost Matrix 23
3.2 Performance Metric Summary 25
3.3 Classifier Accuracy and Balanced Accuracy Comparison 25
3.4 Generated Data Results with Easy Classification 31
3.5 Generated Data Results with Difficult Classification 32

4.1 Intel Images Classification Tasks with Predicted Difficulty 45
4.2 Training Proportions in Intel Images Premade Datasets 45
4.3 Structure of the CNN in the Intel Images Experiments 47
4.4 Balanced Accuracy in Intel Images Easy Classification Tasks 47
4.5 Balanced Accuracy in Intel Images Difficult Classification Tasks 48
4.6 Average Balanced Accuracy for Easy and Hard Experimental Settings . . . 49

5.1 Structure of Neural Network in Kaggle Credit Card Experiments 60
5.2 Neural Network Implementaion in Pytorch 61
5.3 ANN Performance Using State-of-the-Art Loss Functions 62

A.1 Experimental settings for the generated figure. 73

Chapter 1

Introduction

A primary goal in statistics involves predicting a response variable based on data. One example of this goal is in binary classification. In this setting, the response variable, often referred to as the target variable or class, takes on two possible values, and classification techniques aim to classify this response based on data in a way that maximizes accuracy. Classification techniques involve subsetting data into a training dataset, which is used to train the model with data instances for which the response is known, and a testing dataset, which contains instances where the response is treated as unknown.

We are particularly interested in classifying imbalanced data, a common data type in many real world datasets, where the number of instances in each class varies significantly. If we are examining a dataset containing information about incidence of a disease, we may be interested in predicting whether a patient is sick or healthy. This dataset will likely contain many more instances of healthy patients than sick patients, making the dataset highly imbalanced. Canonical classification methods often perform poorly on imbalanced datasets. In the case of the health dataset, a canonical classification method would likely predict almost exclusively healthy patients in order to optimize accuracy.

This thesis discusses classification of imbalanced datasets in the binary setting with neural networks and classifiers. Understanding imbalanced classificaton is critical in health applications. By developing techniques to improve classification in these datasets, we can more accurately determine incidences of cancer, for instance, based on tumor images. We investigate Convolutional Neural Networks (CNNs), a popular classification method for image datasets, and Artificial Neural Networks (ANNs), modifying loss functions to improve performance on imbalanced data.

In Chapter 2, we discuss classification, giving background on the topic with standard classifiers and neural networks. Chapter 3 reviews the class imbalance problem, providing motivation for the thesis and discussing current approaches for mitigating class imbalance. We provide a series of experiments in image classification with CNNs and classification using numerical data with ANNs in Chapter 4 and compare results to those of the generated data in Chapter 3. Finally, Chapter 5 tests the performance of state-of-the-art loss functions on neural networks. We conclude the thesis with a summary of our findings in Chapter 6.

Chapter 2

Classification Techniques

The goal of classification is to predict to which group a new data instance belongs. Classification techniques often involve subsetting data into a training dataset, which is used to train the model with data instances for which the class is known, and a testing dataset, which is used to test performance.

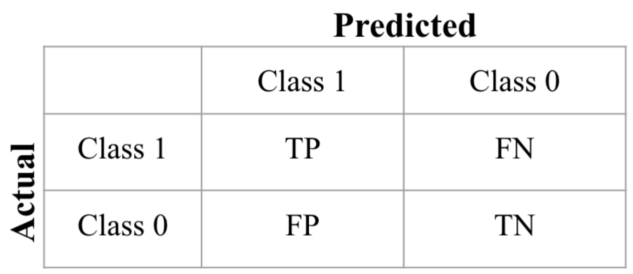
Two common classification methods are machine learning classifiers, discussed in Section 2.1, and neural networks, discussed in Section 2.2. Each method has advantages and disadvantages, and there are further subcategories of each method, i.e., there are many types of machine learning classifiers and many neural network structures.

2.1 Classifiers

Classifiers are tools used for classification tasks, where the goal is to predict a target, or class, based on data. Many different classifiers exist, and each is appropriate in different types of problems. This section discusses three classifiers used in this thesis: Gaussian Naive Bayes (Section 2.1.1), Random Forests (Section 2.1.2), and Gradient Boosting (Section 2.1.3).

Classification tasks can be binary or multi-class. In binary tasks, the class can only take on the values 0 or 1, as when classifying whether a credit card transaction is fraudulent (1) or not (0), for instance. In multi-class problems, there are more than two values the class can take on. One example of such a problem is classifying news articles from five sources back to the correct source.

A variety of techniques are used to evaluate the performance of classifiers. One common technique is the confusion matrix, which compares the number of instances in the actual dataset to the predicted dataset for each class, as shown in Figure 2.1.



		Predicted	
		Class 1	Class 0
Actual	Class 1	TP	FN
	Class 0	FP	TN

Figure 2.1: An example of a confusion matrix for a binary classification problem. Here, TP represents True Positive, FP represents False Positive, FN represents False Negative, and TN represents True Negative.

One of the most common metrics for evaluating classifier performance is accuracy, which measures how often the classifier correctly labels instances (Johnson and Khoshgoftaar, 2019). It is defined as

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{Total}}.$$

Classifiers can be easily implemented in Python using the `Scikit-Learn` library (Pedregosa et al., 2011). It contains many useful algorithms for machine learning tasks that can be implemented with the same general structure, including our three chosen classifiers: Gaussian Naive Bayes, Random Forests, and Gradient Boosting. To load this library, one can simply use the `import sklearn` command (Pedregosa et al., 2011). We discuss the theoretical background of some of these classifiers below. The discussion assumes the reader is familiar with decision trees.

2.1.1 Gaussian Naive Bayes

The Naive Bayes classifier is a probabilistic classifier that applies Bayes Theorem to make predictions, assuming independence and equal importance of predictors, hence the name "naive" (Gayathri and Sumathi, 2016). For some class y and a set of n features $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, Bayes Theorem states that

$$P(y|\mathbf{x}) = \frac{P(y)P(\mathbf{x}|y)}{P(\mathbf{x})}.$$

Since the classifier assumes independence among predictors, we have that

$$P(y|\mathbf{x}) = \frac{P(y)\Pi_{i=1}^{n}P(x_i|y)}{P(\mathbf{x})}.$$

Note that the denominator remains constant, so we can define a classifier that predicts y as

$$\hat{y} = \underset{y}{\operatorname{argmax}}\, P(y)\Pi_{i=1}^{n}P(x_i|y).$$

The Gaussian Naive Bayes classifier follows the same framework as general Naive Bayes classifiers, but they take on the additional assumption that every feature, each being a continuous variable, is distributed according to a Gaussian distribution (Gayathri and Sumathi, 2016). The classifier segments $\mathbf{x}$ by class and calculates the mean μ_j and variance σ_j^2 of $\mathbf{x}$ for each class level y_j. Using these calculations of mean and variance for each class, the classifier can then estimate probabilities with the equation for a normal distribution for some observed value v (Gayathri and Sumathi, 2016). This yields the following mathematical formulation:

$$P(\mathbf{x} = v|y) = \frac{1}{\sqrt{2\pi\sigma_j^2}} e^{-\frac{(v-\mu_j)^2}{2\sigma_j^2}}.$$

In general, Naive Bayes classifiers require small amounts of training data, making them advantageous for small datasets (Gayathri and Sumathi, 2016). The equal importance and independence assumptions do not hold for all datasets, making this technique inapplicable to some scenarios.

2.1.2 Random Forests

The Random Forest classifier uses boostrap aggregation, or bagging, to improve upon the popular Decision Tree classifier (Breiman et al., 2001). Bagging reduces the variance of estimated prediction functions and works particularly well for decision trees, which have high variance and low bias. Bagging averages noisy, unbiased models to reduce their variance. Decision trees often have low bias when grown sufficiently deep but are generally noisy, making these ideal candidates for bagging (Hastie, Tibshirani and Friedman, 2009).

Random forests essentially build a large collection of de-correlated trees and average them using bagging. Whereas decision trees split each node using the best split among all variables in the dataset using a majority vote, random forests split each node using the best among a subset of predictors randomly chosen at that node, constructing multiple trees, each on a bootstrapped dataset (Liaw, Wiener et al., 2002).

Algorithm

Our presentation of the random forest algorithm is based on Hastie, Tibshirani and Friedman, 2009 and Liaw, Wiener et al., 2002. The steps are explained below.

1. Draw B bootstrap samples from the training dataset.
2. Grow an unpruned decision tree T_k for each bootstrap sample, where $k = 1, \ldots, B$. Until the minimum node size s_{min} is reached, recursively repeat each of the following steps for each node:
   (a) Select a variables at random from the n predictors.
   (b) Select the best point to split among those variables.
   (c) Split the node into two parts.
3. Output the ensemble of trees $\{T_k\}^B$.
4. Create a prediction by aggregating the predictions of the B trees. Note that the prediction $\hat{y}$ is the majority vote: $\hat{y} = \text{majority vote}\{\hat{y}_k\}^B$, where $\hat{y}_k$ denotes the class prediction for tree k.

Random forests can be implemented in Python with the Scikit-learn library with the command:

```
from sklearn.ensemble import RandomForestClassifier.
```

Note that the parameters B and a have defaults in Scikit-learn, but can be altered by users in the call to the classifier.

2.1.3 Gradient Boosting

In contrast to random forests, which build an ensemble of deep independent trees, gradient boosting classifiers build an ensemble of shallow trees in a way where each tree improves upon the previous one (Greenwell, 2020). Gradient boosting combines boosting with gradient descent to achieve optimal results.

Boosting

Boosting adds new models to the ensemble sequentially. By starting with a weak learner, such as a decision tree, the model boosts performance by continually adding trees, where each new tree aims to fix errors in prediction from the previous one, as shown in Figure 2.2. Although most boosting algorithms, including gradient boosting, use decision trees as their base learners, boosting can improve upon predictions from any weak learner, which have error rates only slightly better than random guessing (Greenwell, 2020).

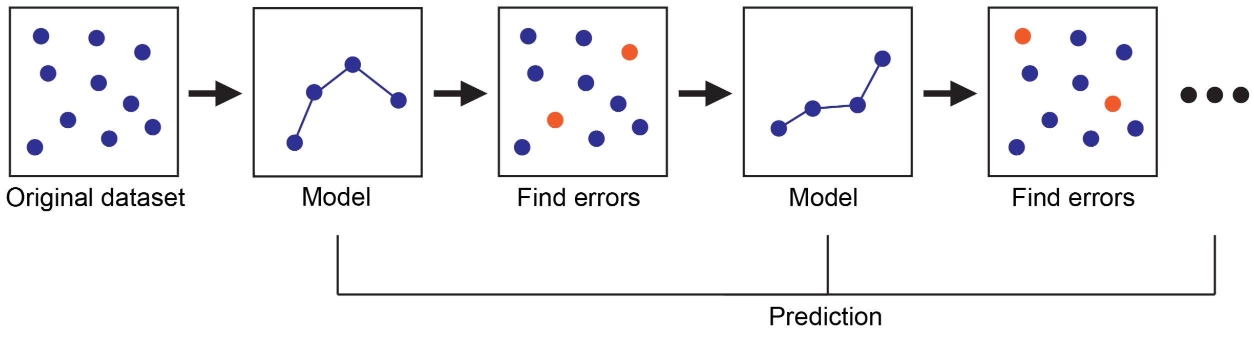


Figure 2.2: Boosting ensembles models sequentially by finding errors with each model, fixing those errors, and building a new model based on the new, learned information. Based on (Greenwell, 2020, Figure 12.1).

Gradient Descent

Gradient descent is a function minimization process. In the case of machine learning problems, the function we want to minimize is a loss, or cost, function in order to ensure the algorithm gives optimal results. Often, this function is given by Mean Squared Error

(MSE), Mean Absolute Error (MAE), or Binary Cross Entropy (BCE), but gradient descent can be performed on any differentiable loss function (Greenwell, 2020). We discuss loss functions further in Chapter 5.

The general idea of gradient descent is to take steps in the direction negative to the gradient at a point until it reaches a local minimum, as shown in Figure 2.3. The size of these steps is determined by the learning rate parameter γ. Too small of a learning rate increases computational time as the algorithm must run for more iterations, while too large of a learning rate means that the algorithm may miss the minimum (Greenwell, 2020).

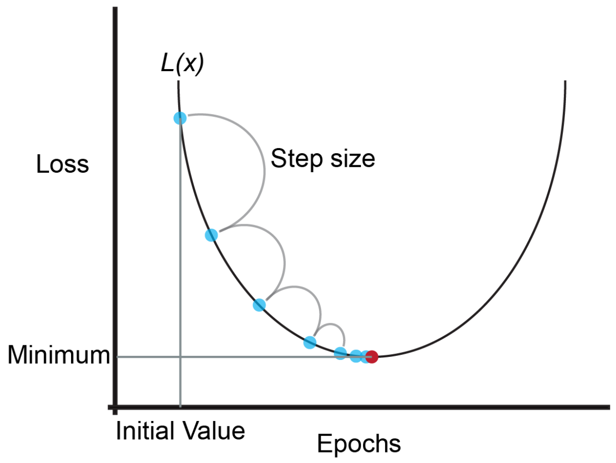


Figure 2.3: Gradient descent is used to minimize loss functions (Greenwell, 2020, Figure 12.4).

Gradient boosting can be implemented in Python with the Scikit-learn library with the command:

```
from sklearn.ensemble import GradientBoostingClassifier.
```

To alter the parameter γ, the user can change the hyperparameter for the learning rate within the Scikit-learn function.

2.2 Neural Networks

A neural network is a predictive model based on the architecture of the human brain. Neurons, or memory units, learn from the features in the dataset through supervised learning, a type of machine learning based on predicting a specific output from an input (Efron and Hastie, 2016). In order to understand how neurons and neural networks work, we consider the simplest example: a perceptron.

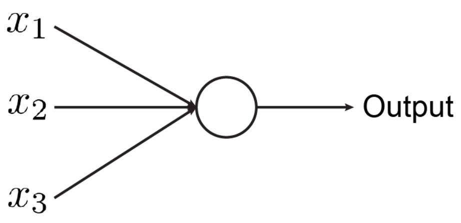


Figure 2.4: A perceptron with three inputs: x_1, x_2, x_3 (Nielsen, 2015, Figure 3).

A perceptron, illustrated in Figure 2.4, takes in n inputs $x_1, x_2, \ldots, x_n$ and produces a binary output, y (Nielsen, 2015). The perceptron creates this output by assigning a series of weights $w_1, w_2, \ldots, w_n$ to represent the relative importance of each input to the output. The binary output of the neuron, 0 or 1, is then determined by summing the weighted inputs and checking if they are above a specified threshold c (Nielsen, 2015). This can be represented mathematically as follows:

$$y = \begin{cases} 0 & \sum_i w_i x_i \leq c, \\ 1 & \sum_i w_i x_i > c. \end{cases}$$

The calculation of an output y for a perceptron can be generalized to a larger neural network, with multiple layers of perceptrons, as well. As in the single perceptron case, each perceptron in each layer of a more complex neural network produces a single output that then gets propagated to the next layer and ultimately produces a single output y, as evident in Figure 2.5.

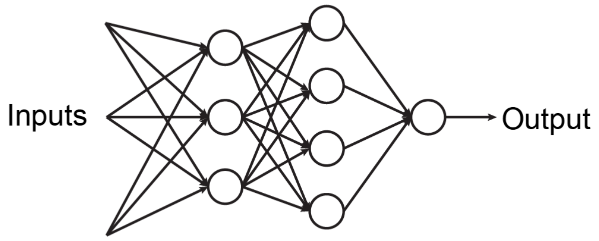


Figure 2.5: A neural network with three layers. Based on (Nielsen, 2015, Figure 4).

Now, we amend our definition of the output y for the sake of simplicity. Observe that

$\sum_i w_i x_i$ can be expressed equivalently as a dot product: $w \cdot x$ (Nielsen, 2015). We can also simplify the right side of the inequality by subtracting the threshold value c from both sides and defining the bias of the perceptron as $b := -c$ (Nielsen, 2015). Now we can define y as

$$y = \begin{cases} 0 & w \cdot x + b \leq 0, \\ 1 & w \cdot x + b > 0. \end{cases}$$

The output in this form is known as the logit. Many models, however, do not use the logit form because this form is relatively simple, only allowing for linearity. In order to introduce non-linearity into the neural network, namely to the output of a neuron, we can apply a nonlinear activation function. One such activation function is the logistic function, where the transformation of y is given by $g(y) = \frac{1}{1+e^y}$ (Skansi, 2018). In experiments discussed in Chapter 4 and Chapter 5, we use a Rectified Linear Unit (ReLU) activation function, which is given by $f(\mathbf{x}) = \max(0, \mathbf{x})$, representing the positive part of the function where $\mathbf{x}$ is a neuron (Nwankpa et al., 2018).

2.2.1 Passing Data

A fundamental part of neural networks involves passing data. This occurs in two ways: forward passes and backpropagation (Skansi, 2018).

Forward Pass

Now that we have established a basic structure for neural networks, the next step in understanding their architecture is by understanding how data is passed through the neurons. The passing of the input through the perceptrons in each layers is known as the forward pass. This is simply the sum of weighted inputs to each perceptron and can be represented as a composition of functions (Skansi, 2018). In Figure 2.5, let f_1, f_2, f_3 denote each layer of perceptrons, respectively. Then, for some input vector $\mathbf{x}$, the output vector $\mathbf{y}$, is given by:

$$\mathbf{y} = f_3(f_2(f_1(\mathbf{x}))).$$

However, in order to create a running neural network, we need to update weight values, which is done through backpropagation (Skansi, 2018).

Backpropagation

In backpropagation, the network measures error during classification and modifies the weights so as to minimize the error. In mathematical terms, backpropagation can be described as gradient descent (Skansi, 2018). Gradient descent, as discussed in Section 2.1.3, is a process by which we can minimize a function. In the case of backpropagation, we are interested in finding optimal values for w and b, weights and biases, respectively, in order to minimize some cost function $C(w, b)$ (Efron and Hastie, 2016). This can be written as

$$w_{\text{new}} = w_{\text{old}} - \gamma C.$$

Here, w_{new} represents the updated weight, w_{old} represents the weight before being updated, γ is the learning rate, and C is the cost function that we want to minimize (Skansi, 2018). We can then use gradient descent, discussed in detail in **??**, to minimize the cost, or loss, function.

Hyperparameters

To optimize the learning process, we must specify hyperparameters, or fixed parameters that control learning. Since these affect the learning process, namely the weight values that are updated, it is crucial to determine the best set of hyperparameters. One common method for this is a grid search, but this technique can be very computationally expensive when the algorithm must search over many combinations of hyperparameters.

2.2.2 Regularization

In order to understand regularization, we first must understand overfitting. Suppose we have two classes and two variables of interst. Some classifier could separate the two classes perfectly by simply drawing a line between the classes, leading to 100% training accuracy. Although this would classify the training data perfectly, it would not necessarily generalize to other data, such as the testing data, well. This problem, where we capture accidental properties in the data, is known as overfitting. Regularization adds a parameter R to the cost function so that it cannot pinpoint datapoints exactly: $C_{\text{new}} = C_{\text{original}} + R$ (Skansi, 2018). It does this by penalizing the cost function so that it becomes impossible to draw a line that simply separates the two classes, as seen in Figure 2.6.

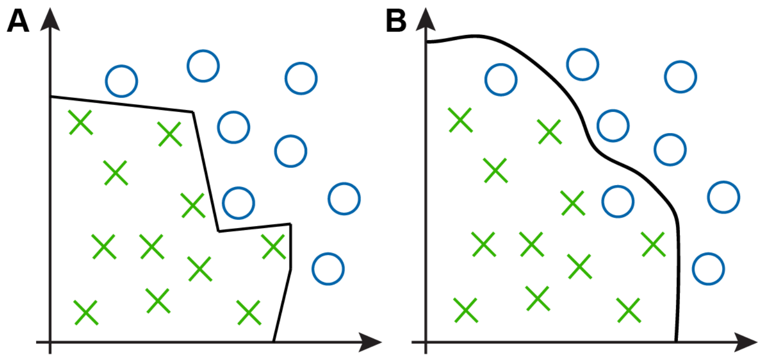


Figure 2.6: In (A), the dotted line shows overfitting where one class is O and the other class is X and these are completely separated. In (B), we see the effects of regularization. Based on (Skansi, 2018, Figure 5.1).

L_2 Regularization

One common type of regularization is L_2, otherwise known as weight decay (Skansi, 2018). In this case, the regularization parameter R that we add to the cost function simply becomes the L_2 norm of the vector of weights, $\mathbf{w}$. Recall that the L_2 norm of $\mathbf{w}$ is defined as

$$||\mathbf{w}||_2 := \sqrt{w_1^2 + w_2^2 + \ldots w_n^2}.$$

Often, however, we use the squared L_2 norm, i.e., $||\mathbf{w}||_2^2 = \sum_i w_i^2$, and we also add a regularization parameter λ to control the amount of regularization that is divided by m so that it is proportional to the number of observations (m), which reflects the number of instances observed before hyperparameters are modified (Skansi, 2018). Our regularized cost function then becomes

$$C_{\text{new}} = C_{\text{original}} + \frac{\lambda}{m} \sum_i w_i^2.$$

In order to figure out how to update the weights based on the updated cost function C_{new}, we take the partial derivative with respect to w on both sides, giving us

$$\mathbf{w}_{\text{new}} = \mathbf{w}_{\text{old}} - \gamma \left(\frac{\partial C_{\text{original}}}{\partial w} + \frac{\lambda}{m} w \right).$$

L_1 Regularization

L_1 regularization is best used for noisy data with many irrelevant instances (Skansi, 2018). It can be derived in the same way as L_2, shown in Section 2.2.2. The main difference between L_1 and L_2 is that L_1 uses an absolute value instead of squares (Skansi, 2018). The L_1 regularized cost function is given by

$$C_{\text{new}} = C_{\text{original}} + \frac{\lambda}{m} ||\mathbf{w}||_1 \Rightarrow C_{\text{new}} = C_{\text{original}} + \frac{\lambda}{m} \sum_i |w_i|.$$

By tuning the cost function, L_1 and L_2 regularization often improve the performance of neural networks.

2.2.3 Convolutional Neural Networks

Convolutional neural networks (CNNs) are a type of neural network architecture often used to process two-dimensional, grid-like data, such as images (W. Liu et al., 2017). As with neural networks in general, convolutional neural networks contain neurons with weights and biases that are updated in the learning process (F.-F. Li, 2018).

Traditional neural networks receive an input that they then transform in the hidden layers and output into the output layer. These neural networks, however, do not scale well to full images because the inherent full connectivity wastes computational time and can lead to overfitting (F.-F. Li, 2018).

The architecture of CNNs avoids the inefficiencies of traditional neural networks with a constrained structure that is optimized for image data (F.-F. Li, 2018). The main difference between regular neural networks and CNNs is that the layers of CNNs have neurons arranged in three dimensions to work for image data: width, height, and depth. Figure 2.7 illustrates this structure.

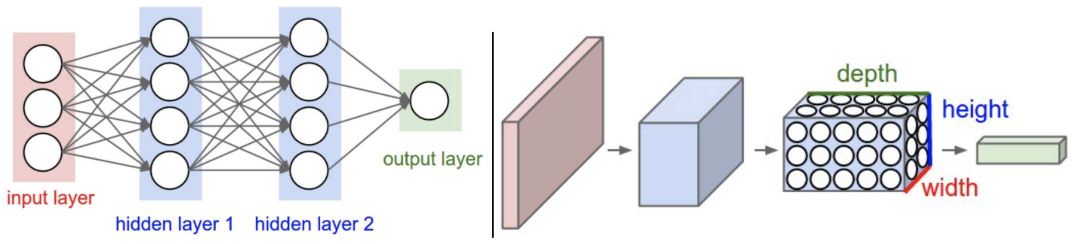


Figure 2.7: On the left, we see a standard neural network, where the input layer, hidden layers, and output layer are each one-dimensional. The right side illustrates a simple CNN architecture. Notice that in the CNN architecture, the neurons are arranged in three dimensions. This image was taken from (F.-F. Li, 2018, Figure 1).

Structure

CNNs have a fundamental structure that allows them to use images to predict class probabilities or class labels. This structure involves five types of layers: Convolutional layers, ReLU layers, Pooling layers, a Flattening layer, and a Fully Connected layer.

Convolutional layers apply a convolution operation, or a combined integration of two functions, to their input (Weaver, 2017). To be specific, we define a small, often 3×3 matrix, to represent the kernel K and overlay it in all possible ways to the image I (Velickovic, 2017). The convolution operator then records the sums of elementwise products as follows:

$$(I*K)_{xy} = \sum_{i=1}^{h}\sum_{j=1}^{d} K_{ij} \cdot I_{(x+i-1,y+j-1)}.$$

As such, the convolution operator allows us to exploit image structure, namely that neighboring pixels influence each other more than pixels on other sides of an image.

Convolutional layers, then, are responsible for extracting features with a feature detector and placing them into a feature map that preserves the spatial relationship of the pixels in the image (Weaver, 2017). We often apply a Rectified Linear Unit layer, or ReLU layer, to convolutional layers for regularization. These decrease the linearity of images, creating a rougher version of the image. This image is then fed into Pooling, or Max Pooling, layers, which remove unnecessary information while preserving features. We can think of Max

Pooling as downsampling our convolutional image to preserve maximally activated parts of the image, as shown in Figure 2.8 (Velickovic, 2017). Note that in Figure 2.8, matrices I and K are unrelated: I corresponds to the initial image, while the kernel K can be chosen by the user.

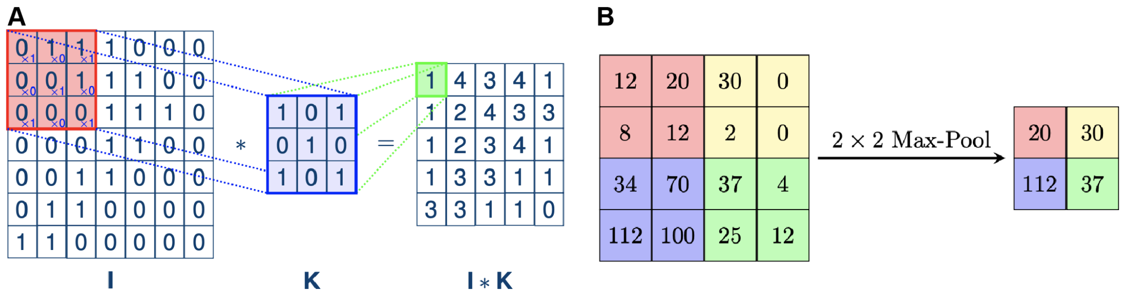


Figure 2.8: In (A), our kernel matrix K is applied to an image I, producing a convoluted image $I{*}K$. We can then apply Max Pooling, shown in (B), to this image to extract the most important features of the image (Velickovic, 2017).

The pooled feature map is then "flattened" with a Flattening layer for dimensionality reduction. The flattened feature map is then fed into the Fully Connected layer, where we concatenate our CNN to an artificial neural network. Making this connection allows us to train our neural network and ultimately get label outputs.

Implementation

We use two main methods to implement CNNs in Python: Keras and Pytorch. Each method has different advantages. Pytorch is more customizable for complex network structures but Keras is simpler to implement. We use Keras for the intial Intel Images experiments and Pytorch for experiments with unique loss functions.

In addition to implementing a loss function for all of our experiments and setting hyperparameters, we also define an optimizer. This helps improve the rate at which the loss function converges. Throughout this work, we use an Adaptive Motive Estimation (Adam) optimizer, an adaptive learning rate technique, which calculates learning rates for each model parameter separately (*Adam — latest trends in deep learning optimization* n.d.).

Chapter 3

Class Imbalance

Class imbalance is prevalent in many real-world datasets. This common data type often makes classification tasks very difficult because canonical machine learning methods struggle to classify data in this setting. This chapter discusses class imbalance in both binary and multi-class classification problems.

3.1 Introduction

Binary target variables often have an unequal representation of the two classes. In imbalanced data, common in health data and fraud data, it is common to have ten times as many instances of the negative class. Consider a dataset containing information about cancer incidence: we expect many more negative instances than positive instances, creating an inherently imbalanced dataset where the minority class, or positive class, contains instances where a patient has cancer and the majority class, or negative class, contains instances where a patient does not have cancer. In such datasets, we are often more interested in predicting the positive class correctly than predicting the negative class correctly. With the cancer dataset, predicting the positive class accurately means that we are detecting cancer effectively so that patients can get appropriate care quickly.

Despite the class imbalance problem, many common deep learning techniques are not attuned to this issue. In order to mitigate class imbalance and improve model performance, we must change metrics for evaluating models to get an accurate sense of model performance, implement sampling methods to balance class distributions, and modify existing algorithm structures to better capture minority class behavior. By doing so, we can more accurately predict the minority class and maximize model performance.

We refer to the Kaggle Credit Cards dataset, which displays extreme class imbalance, to contextualize class imbalance metrics and techniques throughout this chapter and later for classification with a CNN (*Credit Card Fraud Detection* n.d.). In addition to twenty-eight unnamed features obtained through Principal Component Analysis (PCA), a common dimensionality reduction technique, the dataset also contains information about time of a credit card transaction, the amount of the transaction, and whether or not the transaction was fraudulent, which we refer to as the class.

Out of all of the credit card transactions in the dataset, only 0.17% were fraudulent, and the remaining 99.83% were not fraudulent. Traditional machine learning techniques, however, often incorrectly classify positively labeled data because they are built to optimize for accuracy. In this example, standard techniques will tend to classify instances of fraudulent credit card transactions as non-fraudulent because there are many more instances of non-fraudulent transactions. Assuming the same class distribution in the training and testing datasets, by predicting all transactions to be non-fraudulent, classifiers can achieve 99.83% accuracy, i.e., the proportion of non-fraudulent credit card transactions in the dataset. This high accuracy is useless when our goal is to accurately classify positive instances, or fraudulent transactions.

As severely imbalanced datasets can have more extreme impacts on model performance, a metric called the imbalance ratio ρ is used to determine the level of class imbalance:

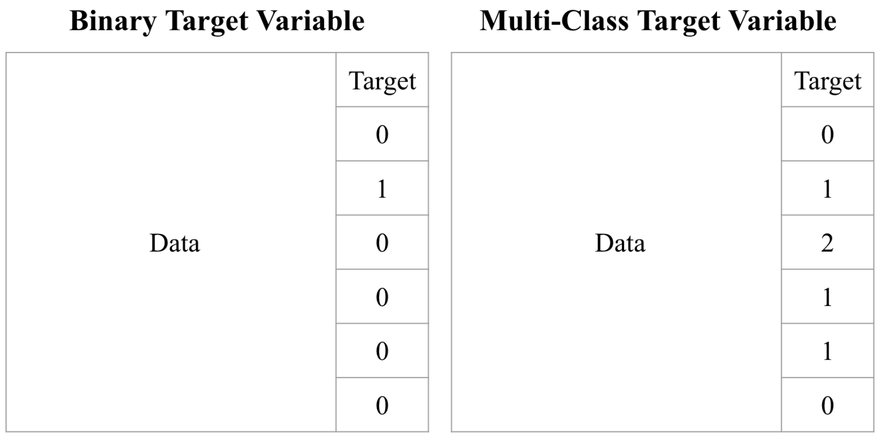
$$\rho = \frac{m_p}{m_n}.$$

Here, m_n represents the number of instances in the negative, or majority, class, and m_p represents the number of instances in the positive, or minority, class (Johnson and Khoshgoftaar, 2019). Higher values of ρ indicate that the dataset is more balanced, while lower values of ρ indicate that the the dataset is relatively imbalanced. For instance, for $m_n = 400$ and $m_p = 100$, we have that $\rho = \frac{100}{400} = 0.25$. This dataset with a total, T, of 500 observations is more balanced than one with $m_n = 450$, $m_p = 50$, which has an imbalance level of $\rho \approx 0.11$.

As per this definition of ρ, values below 1 indicate some level of imbalance, but $\rho < 0.2$ are likely to be particularly problematic. Note that there is no strict cutoff for a problematic imbalance level, but values of ρ closer to zero are more likely to make classification difficult.

3.2 Types of Imbalance: Binary and Multi-Class

Imbalanced data is prevalent in both binary and multi-class target variables. Binary classification, the most developed branch of imbalanced learning, can reflect predicting sick or healthy patients in health data, for instance (Krawczyk, 2016). In security tasks, a binary classification task can reflect detecting attacks on a system. Multi-class classification, on the other hand, involves a target variable that can take on more than two values, where one of these values occurs much more frequently than the others. Figure 3.1 illustrates a simple example of a binary and multi-class target variable.


Binary Target Variable

	Target
Data	0
	1
	0
	0
	0
	0

Multi-Class Target Variable

	Target
Data	0
	1
	2
	1
	1
	0

Figure 3.1: A sample dataset with a binary and multi-class target variable. On the left, the binary target variable is imbalanced with the 0 class being the majority class and the 1 class being the minority class. Though there is not much data shown, the right shows a small dataset with a multi-class target, and the 2 class is the minority class in this case.

Although the imbalance ratio largely dictates the difficulty of the imbalanced classification task, binary classification tasks change in difficulty given the proximity of the two class distributions. If the two distributions do not overlap significantly, the classification task becomes much easier for algorithms to solve. With overlapping class distributions, the task is generally more difficult, despite the imbalance ratio (Krawczyk, 2016). Section 3.5 below gives a detailed example of this idea.

3.3 Techniques for Mitigating Class Imbalance

There are many techniques that aim to improve classification results when dealing with imbalanced datasets. These techniques can be divided into two types: data level methods and algorithm level methods. Although each technique improves classification with imbalanced datasets, the two methods approach the problem differently.

3.3.1 Data Level Methods

One of the two main methods of mitigating class imbalance, data level methods directly alter the training dataset. The main goal of these methods is to balance the dataset by either:

1. Undersampling: Removing instances from the majority class to the level of the minority class or
2. Oversampling: Synthetically creating instances of the minority class to match the level of the majority class (Johnson and Khoshgoftaar, 2019).

Both undersampling and oversampling are generally referred to as resampling techniques, methods of reducing bias in machine learning and deep learning (Johnson and Khoshgoftaar, 2019). Resampling methods balance class distributions by creating or removing instances in data. In balancing positively and negatively labeled instances, classification tasks often become easier to solve.

The simplest forms of these sampling methods are random oversampling (ROS), which randomly duplicates instances from the minority class, and random undersampling (RUS), which randomly removes instances from the majority class (Johnson and Khoshgoftaar, 2019). Each technique has drawbacks. Because we remove examples in undersampling, we lose information by nature of the technique, and in cases with small datasets, we can lose valuable data points that the model can learn from. Oversampling techniques, on the other hand, are prone to overfitting because duplicating instances means a classifier is more likely to create a decision boundary separating the two classes exactly, making it difficult to generalize to other data. Although we want separation between the classes and seek to optimize number of correctly classified instances in the dataset, overfitting means the model is attuned to the intricacies of the training dataset. In other words, the model will likely fail to generalize to the testing dataset, ultimately worsening performance.

Researchers have created several advanced techniques to mitigate the problems with basic undersampling and oversampling techniques. One such method is Synthetic Minority Oversampling Technique (SMOTE), outlined in (Chawla et al., 2002), which creates synthetic instances by selecting k-nearest neighbors from the minority class. Variants of SMOTE, such as SMOTE Edited Nearest Neighbors (SMOTEENN) and SMOTETomek have been developed, which use a nearest neighbors approach and Tomek links, respectively, to oversample the minority class and undersample the majority class (Chawla et al., 2002).

3.3.2 Algorithm Level Methods

Unlike data level methods, algorithm level methods do not change the distribution of the training data. Instead, these methods adjust the algorithm's learning process to increase the importance of the minority class, often with weights and penalties (Johnson and Khoshgoftaar, 2019).

Cost-sensitive learning assigns costs, or penalties, to each class with a cost matrix. By increasing the cost of the majority class, we place more importance on correctly classifying the minority class than correctly classifying the majority class (Weiss, 2004). A cost matrix is illustrated in Table 3.1. Entry c_{ij} of the cost matrix represents the cost associated with predicting class i when the class is actually j, so consequently, we often set entries in the cost matrix to 0 when $i = j$ (Johnson and Khoshgoftaar, 2019). These cost-sensitive learning methods can be characterized into two groups: direct methods or meta-learning methods (Ling and Sheng, 2008).

In direct methods, classifiers are cost-sensitive in themselves, using misclassification costs in learning algorithms so that the optimization process minimizes total cost instead of total error (Ling and Sheng, 2008; Johnson and Khoshgoftaar, 2019). Meta-learning methods, on the other hand, are not cost-sensitive initially, but use pre-processing of the training data and post-processing of the output to create a cost-sensitive learning algorithm (Ling and Sheng, 2008). Thresholding methods redefine the decision threshold during classification using $p^* := \frac{c_{10}}{c_{10}+c_{01}}$, shown in Table 3.1 (Johnson and Khoshgoftaar, 2019). Threshold-moving is a meta-learning method, as it post-processes the output class.

	True Positive	True Negative
Predicted Positive	c_{11}	c_{10}
Predicted Negative	c_{01}	c_{00}

Table 3.1: A sample cost matrix for a binary classification problem.

One difficulty in implementing cost-sensitive learning into classifiers stems from choosing the optimal cost matrix. It is often defined empirically, based on prior knowledge, or by an expert in the field (Johnson and Khoshgoftaar, 2019). Other methods include using a validation set, but this can be computationally expensive, especially when the dataset is large.

3.4 Metrics

Traditional metrics for evaluating machine learning and deep learning models often do not capture model performance with imbalanced datasets. To illustrate this concept, we can consider the following example using a breast cancer dataset (Zwitter and Soklic, 1998). This dataset is moderately imbalanced, with the majority class containing 63% of total instances, and the minority class containing the remaining 37%. If a classifier simply classified all the data as positive, as many do with imbalanced data, we would achieve an accuracy of 63%, but this leaves the remaining 37% of the negatively-labeled observations classified incorrectly. This leads to a high false negative rate, meaning the model will not detect any patients who have cancer.

To combat the issue of incorrectly-classified positive instances, we outline several metrics for model evaluation of imbalanced data that have been developed and explain them in relation to the breast cancer example in Figure 3.2.

1. **Precision:** Proportion of all instances labeled positive that are truly positive, given by
$$\frac{\mathrm{TP}}{\mathrm{TP}+\mathrm{FP}}.$$

2. **Recall:** True positive rate, or the proportion of correctly labeled positive instances. This can be calculated as:
$$\frac{\mathrm{TP}}{\mathrm{TP}+\mathrm{FN}}.$$

3. **Balanced Accuracy:** This is an accuracy metric that is more sensitive to the minority class. It considers True Positive Rate (TPR), the proportion of correctly classified positives, and True Negative Rate (TNR), the proportion of correctly classified negatives, in its calculation:
$$\frac{1}{2}\cdot(\mathrm{TPR}+\mathrm{TNR}).$$

4. **G-Mean:** A performance metric that combines TPR and TNR as follows:
$$\sqrt{\mathrm{TPR}\cdot\mathrm{TNR}}.$$

5. **F-Measure:** Often referred to as the F_1 score, this metric is a combination of precision and recall with harmonic mean:
$$\frac{2\cdot\mathrm{Recall}\cdot\mathrm{Precision}}{\mathrm{Recall}+\mathrm{Precision}}.$$

Table 3.2 displays the performance of the Gradient Boosting classifier on the breast cancer dataset using metrics outlined in Section 3.4. We show accuracy and percent of the dataset that belongs to the majority class as baselines for performance comparison. As expected, accuracy slightly overestimates classifier performance at 95.3% compared to the balanced

Metric	**Value**
Accuracy	95.3%
Percent Majority Class	37%
Precision	50%
Recall	93.7%
Balanced Accuracy	94.9%
G-Mean	77.3%
F-Measure	65.2%

Table 3.2: Summary of performance of the Gradient Boosting classifier on the breast cancer dataset with respect to the metrics outlined in Section 3.4.

accuracy of 94.9%. Note that a lower percent of observations in the majority class in this dataset, i.e., a lower imbalance level, would exacerbate this difference.

Another popular metric for model performance with imbalanced data is the receiver operating characteristics (ROC) curve, which plots TPR over FPR to depict the tradeoff between classifying the minority class correctly and classifying the majority class incorrectly (Johnson and Khoshgoftaar, 2019). A summary of the ROC curve is the area under the curve (AUC). In classification problems, we aim to maximize AUC because a higher AUC means the model better distinguishes between the minority and majority classes. By implementing performance metrics that are less biased towards the negative class, we are better able to evaluate model performance.

Classifier	**Balanced Accuracy**	**Accuracy**
Gradient Boosting	0.562	0.999
Gaussian Naive Bayes	0.828	0.993
Random Forest	0.879	1.000

Table 3.3: A comparison of accuracy and balanced accuracy metrics of three classifiers of interest: Gradient Boosting, Gaussian Naive Bayes, and Random Forest on the Kaggle Credit Cards dataset.

Sample Classifier Results

We can also evaluate the performance of the three classifiers discussed in Chapter 2 on the Kaggle Credit Cards dataset. We are interested in further comparing accuracy and balanced accuracy of Gradient Boosting, Gaussian Naive Bayes, and Random Forest. Based on Table 3.3, we see that all three classifiers perform extremely well in terms of accuracy, illustrating the bias of the accuracy metric toward the majority class. There is much more variety in the balanced accuracy metric, with Gradient Boosting performing worst with balanced accuracy only 56.2% and Gaussian Naive Bayes performing best with balanced accuracy 99.3%. This difference in performance with respect to balanced accuracy is likely a product of the classifiers' decision-making process. Despite the disparity in balanced accuracy for the three classifiers, they all achieve relatively high accuracy.

3.5 Generating Imbalanced Data

In order to understand how deep learning techniques and machine learning classifiers interact with imbalanced data, we must first understand the structure of imbalanced data. This is not a trivial problem, as an imbalanced dataset can contain any number of features, only some of which will be informative in predicting the target. Taking this into account, we create a function that generates imbalanced data to experiment with. We outline a technique to generate imbalanced data given different parameters.

3.5.1 Theoretical Background

Our imbalanced data generation function takes in the following six inputs:

- `n_pos`: number of positive instances, or rows with a positive label
- `n_neg`: number of negative instances, or rows with a negative label
- `n_features`: number of features, or columns
- `n_inf_features`: number of features that inform the outcome in the label
- `sd`: standard deviation of the normal distribution for the minority class
- `class1_mean`: mean of the normal distribution for the minority class

In constructing an imbalanced dataset for our example, we split the problem into constructing the majority and minority class. For the sake of simplicity, assume that `n_features = 1` and `n_inf_features = 1`. We first construct the majority class. Let X represent the majority class. Then $X \sim N(0, 1)$, where the dimension of the array that these draws are stored in is `n_neg` $\times$ `n_features`. Let Y represent the informative subset of the minority class. Then $Y \sim N(\mu, \sigma)$, where μ, σ are defined as inputs to the function, `class1_mean` and `sd`, respectively. Note that all informative variables have the same distribution by this definition.

Note that the parameters μ and σ can be changed to alter the difficulty of the classification problem. $X \sim N(0, 1)$ by default, so suppose that we set `class1_mean = 5` and `sd = 1` such that $Y \sim N(5, 1)$. Then the two distributions X and Y are nearly completely separated, as evident in Figure 3.2, making it easy for classifiers to tackle the problem. To make the classification problem more difficult, set `class1_mean = 1` and `sd = 1` such that $Y \sim N(1, 1)$. Then the distributions of X and Y overlap significantly, making it more difficult for classifiers to distinguish between X and Y, or the majority and minority class, as shown in Figure 3.3. In other words, Figure 3.2 shows easily separable classes and Figure 3.3 shows classes that are difficult to separate.

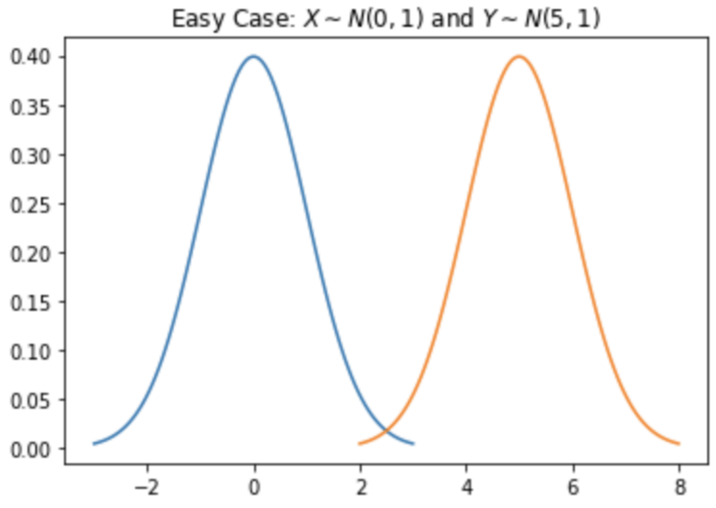


Figure 3.2: Example of an easy classification problem, where $X \sim N(0, 1)$ in blue and $Y \sim N(5, 1)$ in orange.

We can choose `n_pos` and `n_neg` to create the ratio of imbalanced data we want to test. Recall, the class imbalance ratio, ρ, is defined as

$$\rho = \frac{m_P}{m_N},$$

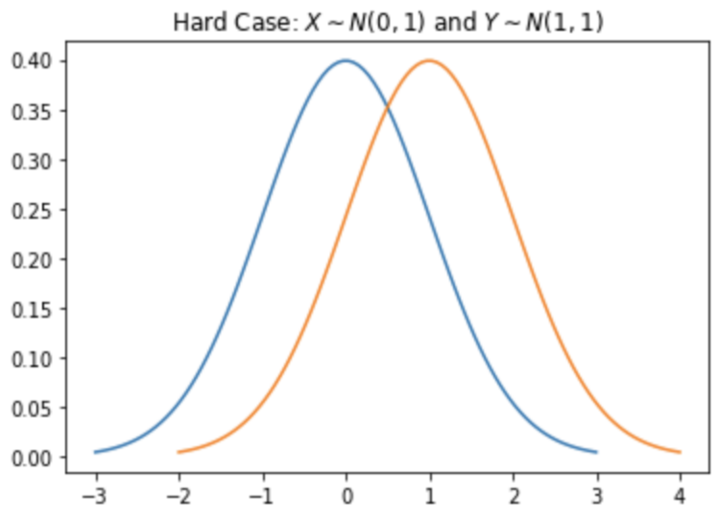


Figure 3.3: Example of a difficult classification problem, where $X \sim N(0, 1)$ in blue and $Y \sim N(1, 1)$ in orange.

where m_P represents the number of positive labels and m_N represents the number of negative labels (Luque et al., 2019). If a dataset with 2,000 instances has 1,900 negative instances and 100 positive instances, then $\rho = 0.05$. Fixing sample size to be 2,000, if the dataset is more balanced with 1,400 negative instances and 600 positive instances, then $\rho \approx 0.43$. In other words, if we fix the size of a dataset, more imbalanced datasets have lower values of ρ.

Finally, choosing `n_features` and `n_inf_features` requires the user to decide how many of the total features they would like and how many of them inform the class label.

3.5.2 Code Background

There are several packages needed to run the imbalanced data generation script. You can import them in Python as follows:

```
import random

import numpy as np

import pandas as pd
```

The random package generates draws from the specified normal distribution, the numpy package creates arrays to store the generated samples, and pandas converts the numpy arrays to dataframes to make them easier to manipulate.

3.5.3 Function

The function is split into two main parts: defining the classes and combining the datasets. We use two features in these simulations, and, for simplicity of notation, refer to the distributions as X and Y with usual univariate notation, though they are bivariate normal with orthogonal features.

The function first defines the negative class, X, where $X \sim N(0, 1)$. Then the function defines the informative part of the minority class, `class1`, i.e., Y, where $Y \sim N(\mu, \sigma)$ as outlined in Section 3.5.1. Finally the function defines the non-informative part of the minority class, `classr1`, which follows the same distribution as X. We combine these classes using the `pandas` package.

The imbalanced data generator function is given below:

```python
def imbalanced_data_generator(n_pos, n_neg, n_features,
                              n_inf_features, sd, class1_mean):

    ##### DEFINE CLASSES #####
    # Define negative class
    class0 = np.array(np.random.normal(loc = 0, scale = 1,
             size = (n_neg*n_features)))
    class0 = np.reshape(class0, (n_neg, n_features))
    class0 = pd.DataFrame(class0)

    # Define informative minority class with mean and std. dev.
    class1 = np.array(np.random.normal(loc = class1_mean, scale = sd,
                                        size = (n_pos*n_inf_features)))
    class1 = np.reshape(class1, (n_pos, n_inf_features))
    class1_cols = np.r_[(n_features - n_inf_features):
                  (n_features + n_inf_features - 2)]
    class1 = pd.DataFrame(class1, columns = class1_cols)

    # Define non-informative minority class (based on Std. Normal)
    classr1 = np.array(np.random.normal(loc = 0, scale = 1,
                                        size = (n_pos*(n_features-
                                    n_inf_features))))
    classr1 = np.reshape(classr1, (n_pos, (n_features-n_inf_features)))
    classr1 = pd.DataFrame(classr1)

    ##### COMBINE CLASSES INTO DATASET #####
    # Concatenate datasets
    c1_full_names = np.r_[0:(n_features)]
    class1_full = pd.concat([class1, classr1], axis=1)
    class1_full = pd.DataFrame(class1_full, columns = c1_full_names)
    data = pd.concat([class0, class1_full], axis = 0)

    # Add labels
    labels = ["0", "1"]
    full_labels = np.repeat(labels, [n_neg, n_pos])
    full_labels = pd.DataFrame(full_labels)
    full_labels = full_labels.rename(columns = {0:'target'})

    # Create full dataset
    full_data = np.append(data, full_labels.to_numpy(), axis = 1)
    full_data = pd.DataFrame(full_data)
    full_data.columns = [*full_data.columns[:-1], 'target']

    # Return dataset
    return(full_data)
```

Listing 3.1: Data Generation Function

3.5.4 Baseline Simulation Tests

Based on the theory outlined in Section 3.5.1, we test the imbalanced data generation function with a simple, easy case and a more difficult case by adjusting the mean of the Normal distribution we specify. In both cases, we hold the number of features, number of informative features, number of positive instances, number of negative instances, and the standard deviation of the informative minority class constant. We set these parameters:

- `n_pos = 100`
- `n_neg = 2000`
- `n_features = 2`
- `n_inf_features = 2`
- `sd = 1`

Note that by setting `n_pos = 100` and `n_neg = 2000`, we have an imbalance level $\rho = \frac{100}{2000} = 0.05$. In both the easy and hard cases, we test Gradient Boost, Random Forest, and Gaussian Naive Bayes with and without SMOTE resampling. Note that we use a 70:30 train test split, and our random seed throughout the simulation is set to 20. Each dataset is synthetically generated once with each parameter combination.

Easy Classification Problem

For the simple case, we have $X \sim N(0,1)$ in two dimensions for comparison and $Y \sim N(5,1)$ in two dimensions. Due to the separation of the classes, we expect the classifiers to do well with and without resampling. The results of the experiments are shown in Table 3.4.

Classifier	Resampling	Balanced Accuracy
Gradient Boost	None	1.000
Gradient Boost	SMOTE	1.000
Random Forest	None	0.985
Random Forest	SMOTE	1.000
Gaussian NB	None	1.000
Gaussian NB	SMOTE	1.000

Table 3.4: Results from the classification of artificially generated data for an easy problem comparing no resampling to SMOTE resampling.

Based on these results, the imbalanced data generator performs as expected, achieving 100% balanced accuracy in almost all cases regardless of classifier and resampling technique. Although the Random Forest does not achieve 100% accuracy without resampling, this is due to one incorrectly classified, false negative, instance.

Difficult Classification Problem

In the hard case, we again have $X \sim N(0,1)$ by default but $Y \sim N(1,1)$, both again in two dimensions. Recall that making μ_Y closer to μ_X makes the classification problem more difficult due to overlap of the classes. As a product of this overlap, we expect the classifiers to perform poorly, with balanced accuracy around 50%, on average. It is possible that we will see some improvement in balanced accuracy with resampling, but this will likely be negligible. The results of the experiments are shown in Table 3.5.

Classifier	Resampling	Balanced Accuracy
Gradient Boost	None	0.497
Gradient Boost	SMOTE	0.462
Random Forest	None	0.497
Random Forest	SMOTE	0.519
Gaussian NB	None	0.500
Gaussian NB	SMOTE	0.434

Table 3.5: Results from the classification of artificially generated data for an difficult problem comparing no resampling to SMOTE resampling.

As expected, all of the classifiers perform poorly on this harder classification problem, regardless of resampling. On average, the classifiers are able to achieve a balanced accuracy of 0.485, near what we expected. Unlike in the easy case, however, Random Forest performs best both with and without resampling, although it ties with Gradient Boost without resampling.

3.5.5 Investigating the Effects of Imbalance Levels

After considering both easy and difficult classification tasks by testing effects of classifiers, resampling, and data distributions for a fixed ρ, we investigate the effects of imbalance level on balanced accuracy in both classifiers and neural networks.

Classifiers

In the classifier experiments, we consider three levels of difficulty by changing the center and spread of our distributions of interest. We consider the following settings in two dimensions, ordered from hardest to easiest:

- $X \sim N(0,1), Y \sim N(2,3)$,
- $X \sim N(0,1), Y \sim N(3,3)$,
- $X \sim N(0,1), Y \sim N(5,3)$.

We test three imbalance levels in each of these settings, and we hold the number of generated observations constant at $2,100$. To change the imbalance levels, we simply change the `n_pos` and `n_neg` arguments to the `imbalanced_data_generator` function given in Section 3.5.3. The imbalance levels are as follows:

- High imbalance: `n_pos = 100` and `n_neg = 2000` $\Rightarrow \rho = \frac{100}{2000} = 5\%$,
- Medium imbalance: `n_pos = 190` and `n_neg = 1910` $\Rightarrow \rho = \frac{190}{1910} \approx 10\%$,
- Low imbalance: `n_pos = 420` and `n_neg = 1680` $\Rightarrow \rho = \frac{420}{1680} = 20\%$.

For each set of distributions and imbalance levels, we vary resampling, considering no resampling or SMOTE resampling, and the classification method: Gradient Boosting (experiments 1, 2, 7, and 8 in each panel), Random Forest (experiments 3, 4, 9, and 10 in each panel), and Gaussian Naive Bayes (experiments 5, 6, 11, and 12 in each panel). We report the results in Figure 3.4.

In Figure 3.4, we see that balanced accuracy is lower for the hard, control task, i.e., the setting $X \sim N(0,1), Y \sim N(2,3)$, than the easier tasks. Note that each data point in the figure represents a particular experimental setting for one classifier. We also observe that resampling with SMOTE significantly improves balanced accuracy for all classifiers but affects problems with lower initial balanced accuracy more prominently.

It's important to note that although we expect the classifiers to perform better on generated data with higher imbalance levels, i.e., higher values of ρ, the experiments don't show

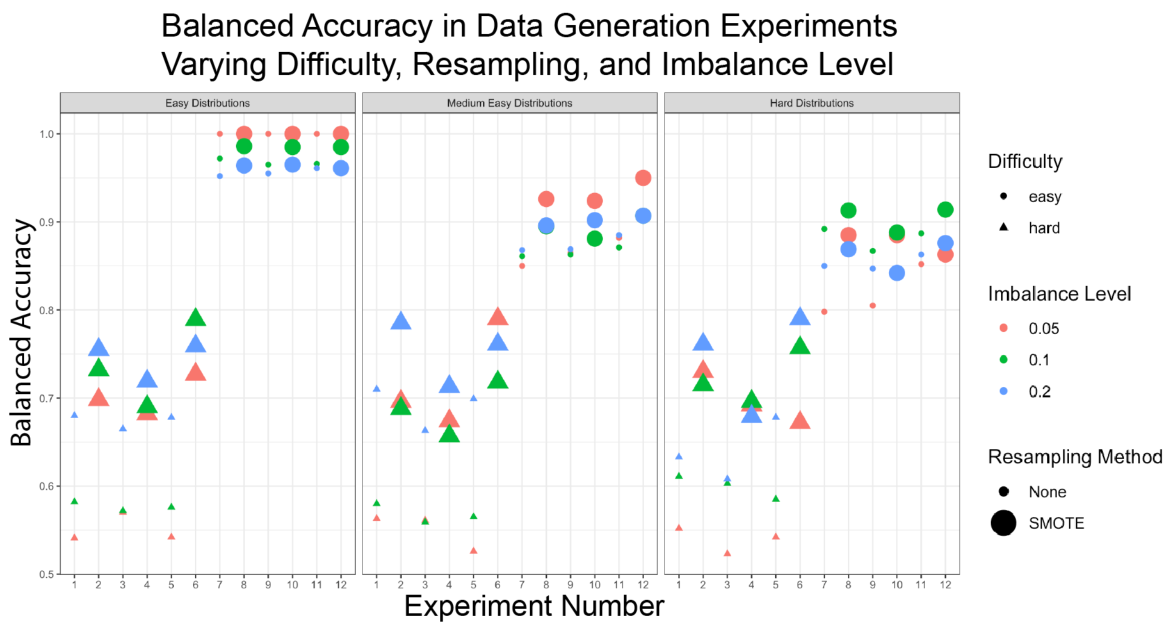


Figure 3.4: Comparison of three pairs of distributions that are used to alter the difficulty of classification. "Easy Distributions," "Medium Easy Distributions," and "Hard Distributions" compare $Y \sim N(5,3)$, $Y \sim N(3,3)$, and $Y \sim N(2,3)$, respectively, to our baseline: $X \sim N(0,1)$.

this trend. In both the easy distributions and medium easy distributions, the classifiers perform best on the most imbalanced data in the "easy" case, though this trend doesn't hold for the harder distributions in each set of experiments.

Neural Networks

The neural network experiments for generated data consider four levels of difficulty, each in two dimensions:

- Most Difficult: $X \sim N(0,1), Y \sim N(1,1)$,
- Medium Hard: $X \sim N(0,1), Y \sim N(2,1)$,
- Medium Easy: $X \sim N(0,1), Y \sim N(3,1)$,
- Easy: $X \sim N(0,1), Y \sim N(5,1)$.

These settings, in order from most to least difficult, hold the spread constant at $\sigma = 1$ but vary center. As in the classification experiments, we use $X \sim N(0,1)$ as a baseline for comparison. In each of these experimental settings, we also vary the imbalance levels:

- High imbalance: `n_pos = 100` and `n_neg = 2000` $\Rightarrow \rho = \frac{100}{2000} = 5\%$,
- Medium imbalance: `n_pos = 190` and `n_neg = 1910` $\Rightarrow \rho = \frac{190}{1910} \approx 10\%$,
- Low imbalance: `n_pos = 420` and `n_neg = 1680` $\Rightarrow \rho = \frac{420}{1680} = 20\%$.

Throughout these experiments, we keep a set of constant hyperparameters. Using a batch size of 16, 50 hidden nodes, 100 epochs and a learning rate of 10^{-4}, we see that the loss functions converge to 0, indicating that the neural networks have successfully optimized the loss functions.

We plot the balanced accuracy of the experiments for each level of difficulty and each imbalance level. As before, each data point indicates the performance of a single neural network.

In Figure 3.5A, we see that, unlike in the classifiers, the easiest tasks have the highest balanced accuracy and the most difficult tasks have lowest balanced accuracy in neural network experiments without resampling. For the "Easy" and "Medium Easy" tasks, we see that an increase in the imbalance level does not improve performance significantly, especially from $\rho = 0.1$ to $\rho = 0.2$. For both the "Medium Hard" and "Most Difficult" classification tasks, however, the increase in the imbalance level strongly correlates to better classification performance, given by balanced accuracy.

When we use SMOTE resampling in our neural network, this pattern changes. As shown in Figure 3.5B, instead of improving results with higher imbalance levels and for the "medium hard" and "most difficult" tasks, SMOTE resampling decreases balanced accuracy. For the "easy" and "medium easy" tasks, balanced accuracy improves significantly with higher imbalance levels with resampling. However, even the highest balanced accuracy we achieve with resampling at the $\rho = 0.2$ imbalance level is lower than the unresampled data at all values of ρ.

This suggests that resampling data with relatively separable classes, such as the "easy" and "medium easy" tasks, is detrimental to a neural network's performance, regardless of the imbalance level. For more difficult tasks, such as the "medium hard" and "most difficult" tasks, resampling may be beneficial for low imbalance levels, particularly $\rho \leq 0.1$.

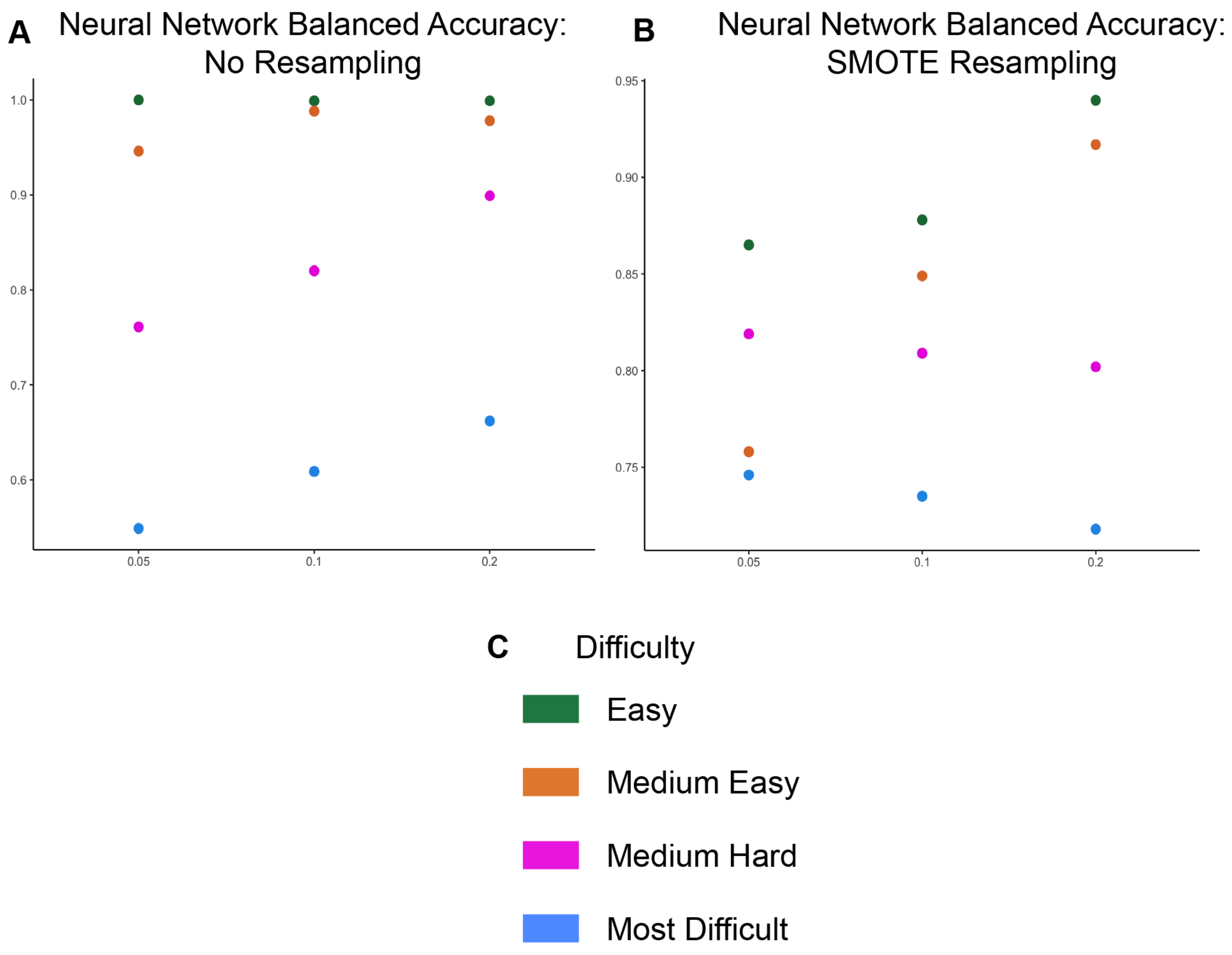


Figure 3.5: Plot of balanced accuracy at different imbalance levels (ρ) for the four difficulty settings with (A) no resampling and (B) SMOTE resampling. (C) provides a legend for the difficulty levels shown.

Difficulty over All Imbalance Levels

To determine how these classifiers perform at each imbalance level from 0.1 to 1, we create a function to calculate each classifier's performance at each imbalance level. As in the previous experiments, we use the same levels of difficulty, where each is two-dimensional:

- Most Difficult: $X \sim N(0,1), Y \sim N(1,1)$,
- Medium Hard: $X \sim N(0,1), Y \sim N(2,1)$,
- Medium Easy: $X \sim N(0,1), Y \sim N(3,1)$,
- Easy: $X \sim N(0,1), Y \sim N(5,1)$.

We show the results of this experiment in Figure 3.6. Note that each of Figure 3.6A-Figure 3.6D have different y-axis scales, so the visualizations are not directly comparable across the figure.

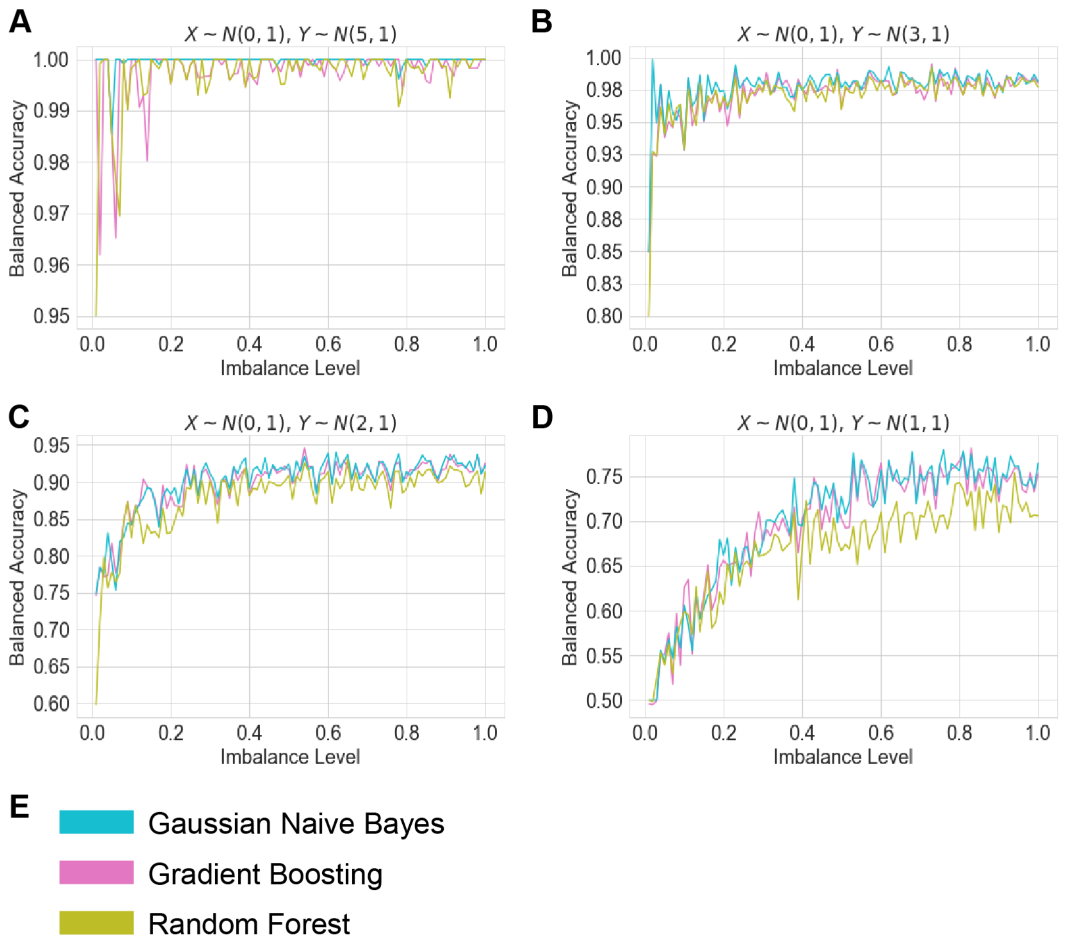


Figure 3.6: Visualizations of classifier performance for imbalance levels of 0.1 to 1 on different generated datasets. (A) shows an easy case, (B) shows a medium easy case, (C) shows a medium hard case, and (D) shows a hard case. (E) provides a legend for the classifiers used in the experiments.

Notice that with the easy and medium easy cases, low imbalance levels produce significant noise, indicating that classifiers are likely more sensitive to low imbalance levels on easily-classifiable data distributions. For the medium hard and hard cases, on the other hand, there is a little bit of noise initially using the Random Forest classifier on the medium hard case, while there is no noise at low imbalance levels on the hard case.

We also notice that it becomes easier to distinguish between classifier performance at higher imbalance levels for harder cases. In Figure 3.6C, Random Forest seems to consistently perform slightly worse than Gaussian Naive Bayes and Gradient Boosting for higher imbalance levels, i.e., $\rho > 0.4$. Similarly, in Figure 3.6D, the performance of Random Forest levels off at a lower balanced accuracy than Gaussian Naive Bayes and Gradient Boosting starting at approximately $\rho > 0.4$.

Simulation Study

We want to verify the results of the difficulty over all imbalance levels. To do so, we generate many imbalanced datasets and conduct Random Forest, Gradient Boosting, and Gaussian Naive Bayes Classification on each of them for each imbalance level. As in the previous experiments throughout this chapter, we test the same four difficulty levels, where each is two-dimensional:

- Most Difficult: $X \sim N(0, 1), Y \sim N(1, 1)$,
- Medium Hard: $X \sim N(0, 1), Y \sim N(2, 1)$,
- Medium Easy: $X \sim N(0, 1), Y \sim N(3, 1)$,
- Easy: $X \sim N(0, 1), Y \sim N(5, 1)$.

The results of this experiment are shown in Figure 3.7. Note that, as in Figure 3.6, each of Figure 3.7A-Figure 3.7D have different y-axis scales, so the visualizations are not directly comparable across the figure. We generate 100 datasets and use the three classifiers to perform classification on each dataset. The shaded region outside of the lines for each classifer in Figure 3.7 shows the balanced accuracy range for each of the 100 classification attempts.

The results mimic those of the single generated dataset almost exactly. In general, the classifiers all converge to the same balanced accuracy for each classification task difficulty. In Figure 3.7A, the classifiers converge to 1 in balanced accuracy, in Figure 3.7B, this value decreases to 0.975, both matching their respective parts in Figure 3.6. This trend holds for Figure 3.7C and Figure 3.7D when compared to Figure 3.6 as well. The balanced accuracy values to which each of these classifiers converge in each experimental setting

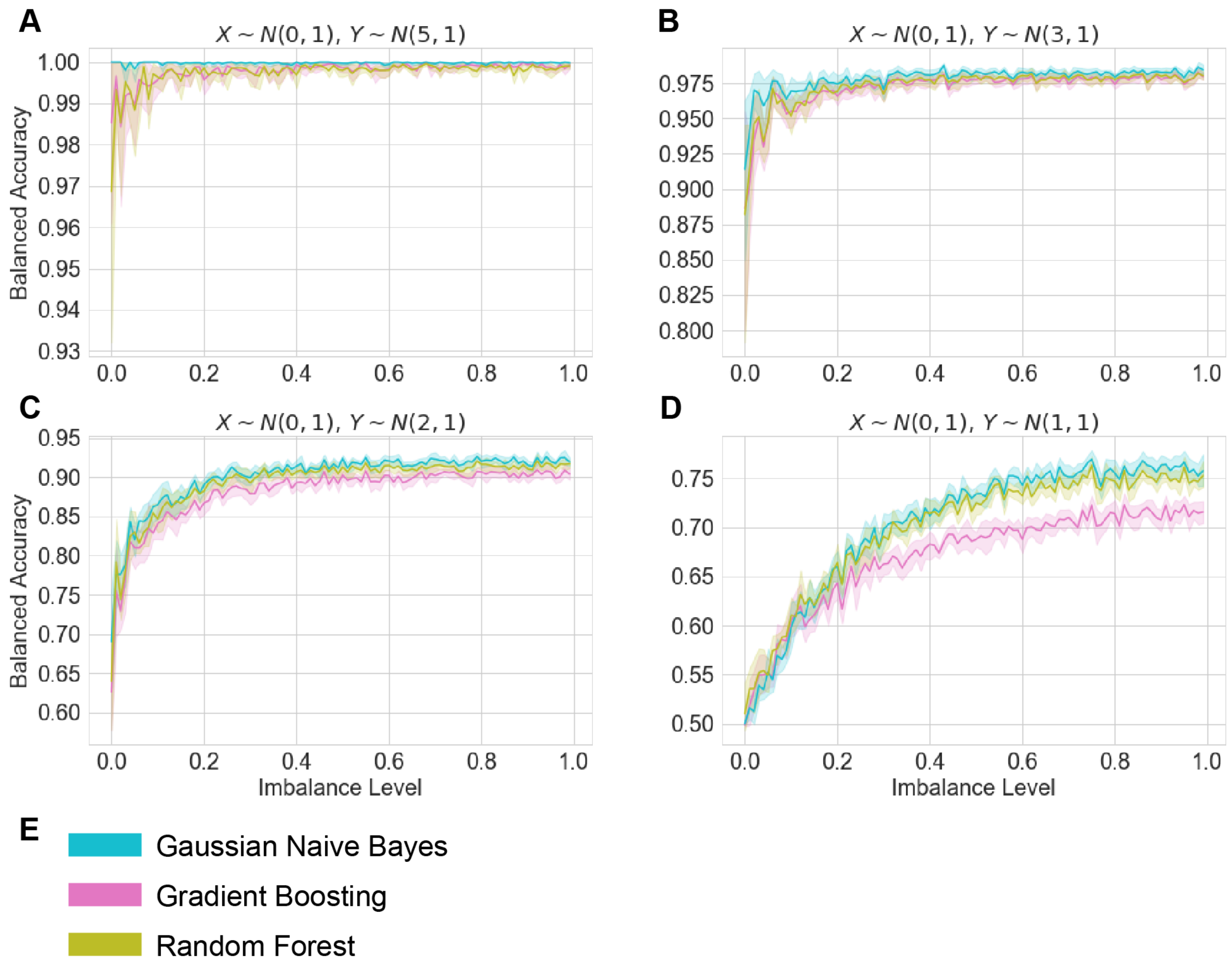


Figure 3.7: Visualizations of classifier performance for imbalance levels of 0 to 1 on 100 generated datasets at each imbalance levels. (A) shows an easy case, (B) shows a medium easy case, (C) shows a medium hard case, and (D) shows a hard case. (E) provides a legend for the classifiers used in the experiments.

decreases as the classification task becomes more difficult, as shown in Figure 3.6 as well. This also shows that Gradient Boosting performs worse than Random Forest, particularly for the more difficult classification tasks. As expected, the simulation shows more noise with lower imbalance levels, particularly those for which $0 < \rho < 0.2$, supporting the conclusion made from Figure 3.6.

Conclusions

Evidently, imbalance level highly affects both classifier and neural network performance in terms of balanced accuracy. The effects of this imbalance, however, depend on the distribution of the data. For easily separable classes, not only is balanced accuracy higher on average but it is also noisier for lower imbalance levels. For classes that are difficult to separate, classifiers generally perform worse and imbalance level significantly impacts performance. The simulation, where we generate 100 imbalanced datasets and perform classification, confirms these results.

Additionally, SMOTE resampling only marginally improves results for easily separable classes but improves results significantly for more difficult cases. Utilizing resampling techniques, therefore, proves to be more useful in highly imbalanced datasets with low separability.

Chapter 4

Imbalanced Data Classification

The goal of imbalanced data classification is to correctly identify labels based on features, or data. The data we generated in the previous chapter was numerical, but we can also predict labels using images as data. Recently, Convolutional Neural Networks (CNNs) have gained popularity for their strong performance and low time complexity in image classification tasks. This chapter first discusses the performance of a CNN on imbalanced image data from the Intel Images dataset, where one image occurs much more often than another, and then discusses classification of numerical data with the Kaggle Credit Cards dataset.

4.1 Image Classification Background

In image classification, as in all classification tasks, we want to assign correct labels to images. If we have a set of images containing cats and dogs, for instance, we want our classifier to recognize images of cats as cats and images that are not of cats as not of cats. With imbalanced data, there are many more images in one category than the other, and the more images that are classified correctly, the better the classifier performs. We examine this problem with the Intel Images dataset, described in Section 4.1.1, with experiments using neural networks and classifiers, discussed in Section 4.2.1. Finally, the results and analysis of the experiments are given and analyzed in Section 4.2.2, followed by a brief discussion of these results in Section 4.2.3.

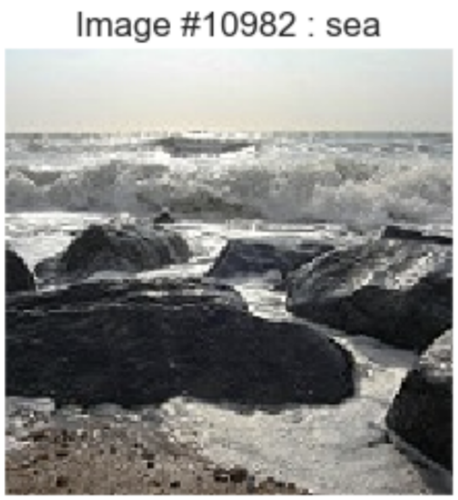


Figure 4.1: A random image from the Intel Images dataset that belongs to the "sea" class.

4.1.1 Intel Images Dataset

Throughout this chapter, we use the Intel Images dataset to test various CNN structures (Bansal, 2019). This dataset has six classes: mountains (3037 images), sea (2784 images), buildings (2628 images), glacier (2957 images), street (2883 images), and forest (2745 images). Figure 4.1 shows an example of what an image from the dataset may look like. To make this a binary classification problem, we consider each possible combination of these six classes, yielding fifteen possible combinations.

Some of these classification tasks, however, are more challenging than others due to persistent similarities in images. Using Principal Component Analysis (PCA), we attempt to determine which classification tasks are computationally difficult and which ones are "easy." By examining the PCA clusters, which roughly correspond to the image labels, we can determine the computational difficulty of pairing each combination of labels. For example, it is likely easier to achieve high balanced accuracy when classifying streets and mountains, as there is little overlap between them on the two-dimensional PCA. Classifying glaciers and mountains, however, is likely more difficult because the two-dimensional PCA shows significant overlap between these labels.

We examine the PCA in higher dimensions to get an even better sense of the difficulty of each problem. In Figure 4.2, we see that plotting different dimensions of the PCA in

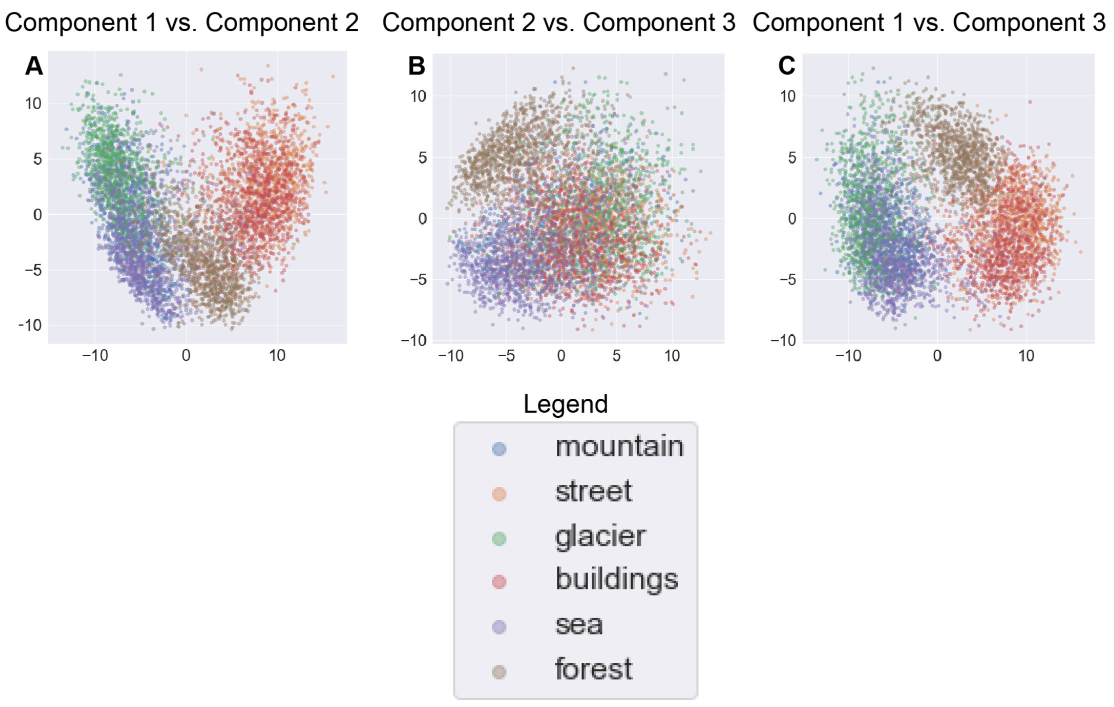


Figure 4.2: The three-dimensional Principal Component Analysis of the images in the Intel Images dataset can help determine whether classification will be easy or difficult.

three dimensions gives different impressions of the difficulty of each binary combination of labels.

In Figure 4.2A and Figure 4.2C, we see four relatively separate classes. There, blue points are mixed with the green and purple ones and orange points are mixed with red ones, indicating difficulty in distinguishing mountain images from glacier and sea images, and street images from building images, respectively. Figure Figure 4.2B shows separation between the brown points and the rest of the points, showing relative separation between forest images and the rest of the classes.

4.2 Image Data Experiments

We first discuss the results of the image data experiments using the Intel Images dataset, providing an overview of our neural network structure and implementation, followed by results and analysis of these experiments.

4.2.1 Experiment Structure

To test the performance of neural networks on the Intel Images dataset, we first need to preprocess the data. Then, we can develop an experimental structure to use to test a variety of techniques on this dataset.

Preprocessing

Although initially a multi-class, fairly balanced dataset, we modify the Intel Images dataset to make it a binary, imbalanced dataset by examining two classes images at a time and removing instances from one class to simulate imbalance.

As mentioned in Section 4.1.1, there are fifteen possible combinations of the six labels in the dataset, and we can categorize each task as easy or difficult by observing the label overlap in the PCA projection. Based on the PCA, however, we describe the predicted difficulty as "medium" if the PCA shows it is easy in some dimensions and hard in others. We omit these from our analysis, however, because we choose to examine only "easy" and "hard" cases for simplicity in analysis and conclusions. We outline the combinations and their predicted computational difficulty in Table 4.1.

We split the data into training and testing datasets using the premade splits in the Intel Images dataset. The split is near 80% in the training set and 20% in the testing dataset for each image class, but there is some variability between classes.

To prepare the images for training, we first convert the images from the BGR color space to the RBG color space, allowing us to visualize the images with the `matplotlib` package, a popular data and image visualization package. We then resize the images to a set size, 150×150 pixels.

Finally, we have to artificially make the Intel Images dataset imbalanced by removing some instances from one class to make it the minority class. We can tune the imbalance parameter, which determines which proportion of observations to keep in the positive class, to determine the level of imbalance in the dataset. Note that we have to remove observations from the desired class in both the training and testing datasets to simulate

Majority Class	**Minority Class**	**Predicted Difficulty**
Mountain	Street	Easy
Mountain	Building	Easy
Street	Glacier	Easy
Street	Sea	Easy
Glacier	Building	Easy
Mountain	Glacier	Hard
Mountain	Sea	Hard
Street	Building	Hard
Glacier	Sea	Hard
Mountain	Forest	Medium
Street	Forest	Medium
Glacier	Forest	Medium
Building	Sea	Medium
Building	Forest	Medium
Sea	Forest	Medium

Table 4.1: We use the PCA from Figure 4.2 to predict the classification difficulty of two given labels. We determine the classification difficulty for the fifteen possible combinations of the six labels in the Intel Images dataset.

Image Class	**Prop. in Training Set**
Mountain	0.83
Street	0.83
Glacier	0.81
Buildings	0.82
Sea	0.82
Forest	0.83

Table 4.2: The proportion of images in the training class is approximately 80% for each image, though there is some variation in each split.

a real-world imbalanced dataset that has few observations in the testing dataset. We use $\rho = 0.10$ as our imbalance level in the experiments discussed in Section 4.2.2 and Section 4.2.2 for each combination of image classes.

Theoretically, if we flip the majority and minority classes, the balanced accuracy we achieve should not change significantly. In other words, changing which is the minority class and which is the majority class in each binary combination of classes should not change a classifier's ability to separate classes. If the classes are easily separable, the classifier should be able to achieve high balanced accuracy whether it has many observations or few observations. Likewise, having many observations or few observations in two classes in a difficult classification problem does not change the predicted difficulty of the problem.

Neural Network Structure

We use the `keras` package, an easy and efficient framework for building neural networks, to create our models. In this process, we first define our model architecture, compile our model, train the model with the training dataset, and evaluate its performance on the testing dataset.

When we define our model architecture, we must define it as being `Sequential()`, or a linear stack of layers, and then add the layers we want. Our model will use convolutional layers, max pooling layers, a flattening layer, and fully-connected layers. Here, each layer can be added to the model with a call to `model.add()` that contains the desired layer type. If we want to add a convolutional layer, for instance, we write `model.add(Convolution2D)`. In each layer, there are parameters we can, or must, add, including dimensionality parameters and activation functions.

We use a sequential, seven-layer neural network structure: a convolutional layer, a max pooling layer, a convolutional layer, a max pooling layer, a flattening layer, and two fully-connected layers, motivated by model tuning and models in the literature. Table 4.3 shows the structure of the CNN used in these experiments. Using an Adam optimizer and a sparse categorical cross-entropy loss function, appropriate for integer target labels, we can evaluate our model's performance on the test dataset. Note that we run our model with a batch size of 128, use 20 epochs, and use a 0.2 validation split and that these hyperparameter choices are the result of hyperparameter tuning.

4.2.2 Results

In this section, we compare the results of easy and hard classification problems based on the three-dimensional PCA projection. Using the neural network structure described in Section 4.2.1, we classify the sets of images in our testing dataset after training our model.

Layer	**Width**	**Height**	**Depth**
Input	150	150	3
Convolution + ReLU	148	148	32
Max Pooling	74	74	32
Convolution + ReLU	72	72	32
Max Pooling	36	36	32
Flattening	1	1	41472
Fully-connected + ReLU	1	1	128
Fully-connected + Softmax	1	1	6

Table 4.3: Structure of the CNN in the Intel Images experiments.

Easy Classification Problem

Based on the three-dimensional PCA projection analysis, Table 4.1 shows that we have five classification tasks we predict will be easy.

As shown in Table 4.4, our neural network performs well on each of our predicted easy classification tasks. We achieve 89.2% balanced accuracy, on average. Classifying streets and sea seems to be the easiest, as our model achieves 92.2% balanced accuracy on that task. The model has trouble classifying glaciers and buildings, given that this is an easy task, however, as it achieves only 83.9% balanced accuracy here.

Labels	**Balanced Accuracy**
Mountain vs. Street*	0.919
Mountain vs. Building*	0.872
Street vs. Glacier*	0.908
Street vs. Sea*	0.922
Glacier vs. Building*	0.839

Table 4.4: The model performs as expected on easy classification tasks, achieving high balanced accuracy on each of our theoretically easy classifications. Note that the asterisk indicates that the label represents the minority class.

Intuitively, the model's strong performance on these tasks makes sense because pictures of these scenes lack overlap in each binary classification problem. In other words, there are likely no big buildings near glaciers, so it is easy for our model to learn the difference between these objects.

Hard Classification Problem

Table 4.1 shows four theoretically difficult classification tasks. The model should perform worse on these classification tasks that it did for the easy ones shown in Section 4.2.2.

Table 4.5 shows this is indeed true. We achieve an average balanced accuracy of only 68.3%, which is about 20% worse than that of the easy tasks. Here, the model is best at classifying streets and buildings with a balanced accuracy of 71.8% and worst at classifying glaciers and sea with a balanced accuracy of 66.5%.

Labels	**Balanced Accuracy**
Mountain vs. Glacier*	0.666
Mountain vs. Sea*	0.682
Glacier vs. Sea*	0.665
Street vs. Building*	0.718

Table 4.5: The model performs much worse on the theoretically difficult classification tasks.

Again, the difficulty of these tasks makes sense intuitively because of the overlap in these objects. Pictures of streets likely include buildings, for instance, so the model struggles to differentiate between these.

Changing Imbalance Levels

To better understand the effects of imbalance on performance on a real-world dataset, we change the imbalance levels on the Intel Images dataset. We change the imbalance level for both the easy and hard experiments using the neural network structure described in Section 4.2.1. By testing imbalance levels of $\rho = 0.05$ and $\rho = 0.2$, we can compare these values to our initial imbalance level of $\rho = 0.1$ used in experiments described in Section 4.2.2 and Section 4.2.2.

We show the average balanced accuracy of each experimental setting in Table 4.6 and the breakdown of the impact of each imbalance level in Figure 4.3.

Based on the results of these experiments, the average balanced accuracy of hard classific-

Abbreviation	**Experimental Setting**	**Difficulty**	**Average Balanced Acc.**
MO + ST	Mountain and Street*	Easy	0.906
MO + BU	Mountain and Building*	Easy	0.821
ST + GL	Street and Glacier*	Easy	0.894
ST + SE	Street and Sea*	Easy	0.907
GL + BU	Glacier and Building*	Easy	0.812
MO + GL	Mountain and Glacier*	Hard	0.654
MO + SE	Mountain and Sea*	Hard	0.667
GL + SE	Glacier and Sea*	Hard	0.702
ST + BU	Street and Building*	Hard	0.717

Table 4.6: Average balanced accuracy for both easy and hard experimental settings. We include the abbreviations for reference in Figure 4.3. As before, the class with the asterisk is the minority class.

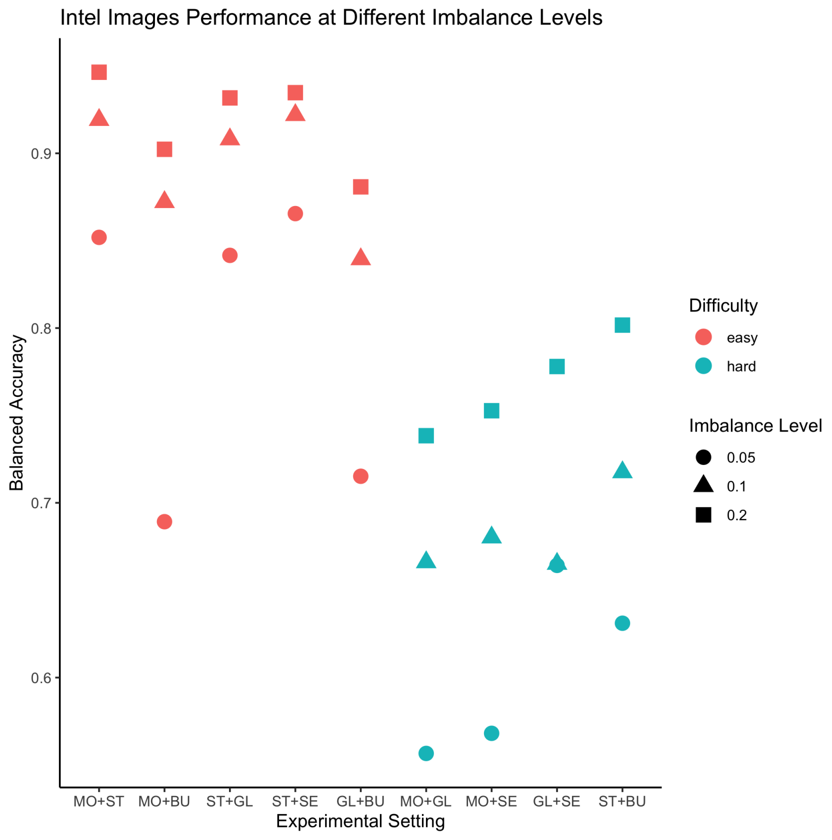


Figure 4.3: Intel Images binary classification performance for imbalance levels of $\rho = 0.05$, $\rho = 0.1$, $\rho = 0.2$.

ation tasks in the Intel Images dataset is 0.685, 26.7% lower than that of easy classification tasks, which have an average balanced accuracy of 0.868 for the dataset.

Specifically, according to Figure 4.3, we see that higher values of ρ correspond to higher balanced accuracy for all experimental settings. This means that more balanced datasets are easier for the neural network to classify. For the settings described as "easy," the gap between $\rho = 0.05$ and $\rho = 0.1$ was generally larger than that between $\rho = 0.1$ and $\rho = 0.2$, indicating that lower imbalance levels, i.e., lower values of ρ, have a stronger impact on classification performance. Due to the lack of a similar trend in the "hard" experiments, the difficulty of the classification task may offset the difficulty of lower imbalance levels.

4.2.3 Conclusions

The Intel Images dataset shows that CNNs perform worse on harder classification tasks than easy classification tasks, supporting the results from the neural network experiments with generated data from Chapter 3. Unlike the synthetic datasets, however, neural networks consistently perform better with higher imbalance levels on the Intel Images dataset, which supports the idea that higher imbalance levels make classification tasks easier.

4.3 Numerical Data Experiments

Now, we discuss classifier experiments with a real-world numerical dataset: the Kaggle Credit Card Fraud dataset. This dataset contains information about the validity of credit card transactions in Europe, where 492 out of 284,807 transactions, or 0.172% of transactions, are fraudulent (*Credit Card Fraud Detection* n.d.). The initial imbalance level in the dataset is $\rho \approx 0.002$, but we modify the imbalance level artificially in our experiments by removing instances from the negative class.

4.3.1 Experiment Structure

In these experiments, we use three classifiers to determine the validity of transactions: Gradient Boosting, Random Forest, and Gaussian Naive Bayes. As in previous experiments, we use a 70-30 train-test split for each experiment. Ultimately, our goal in these experiments is to compare the classification results of a real numerical dataset to those of our generated dataset from Chapter 3.

We perform three experiments to thoroughly compare these results. In all three experiments, we test the same three classifiers: Gradient Boosting, Random Forest, and Gaussian Naive Bayes. First, we test the performance of the classifiers with data at three imbalance levels (ρ = 0.05, 0.01, 0.2), both with SMOTE resampling and without any resampling. Next, we compare classifier performance over imbalance levels from 0.1 to 1. Finally, to test the robustness of these results, we perform classification with each classifier 100 times at each imbalance level.

Note that, because this is a real dataset, we don't know how "hard" or "easy" the classification task is. Using the results of these experiments, however, we may be able to infer the difficulty, which can inform which techniques may be appropriate to improve results.

4.3.2 Results

In our first experiment, we test classifier performance on the Kaggle Credit Cards dataset with imbalance levels of ρ = 0.05, 0.1, and 0.2 with no resampling and with SMOTE resampling. The results are shown in Figure 4.4.

As evident in Figure 4.4, resampling doesn't affect performance significantly, especially at higher imbalance levels. We also note that higher imbalance levels improve performance for all classifiers, both with and without resampling. Gradient Boosting and Random Forest classifiers consistently perform best, while Gaussian Naive Bayes performs worse at all imbalance levels, regardless of resampling.

When we test classifier performance over all imbalance levels, shown in Figure 4.5, Gradient Boosting better than both Random Forest and Gaussian Naive Bayes in general. Still, the balanced accuracy increases quickly for $0 < \rho < 0.2$, and even at these low imbalance levels, all classifiers achieve balanced accuracy of 95% or more. As the imbalance level approaches 1, the balanced accuracy also converges to 1 for all classifiers.

These results are reinforced in the simulation, where each point in Figure 4.6 represents

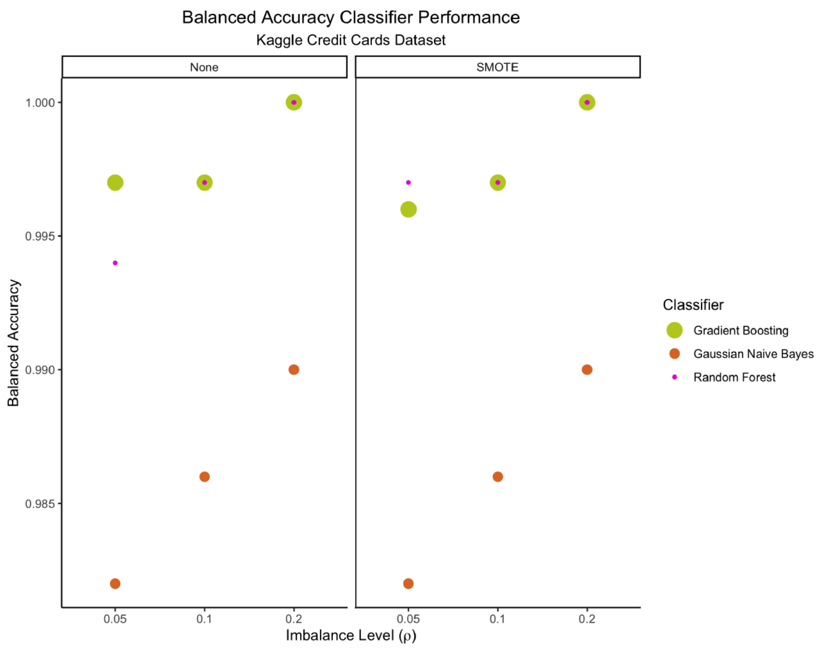


Figure 4.4: Classifier Performance on the Kaggle Credit Cards data with no resampling (left) and with SMOTE resampling (right) for imbalance levels of $\rho = 0.05, 0.1, 0.2$.

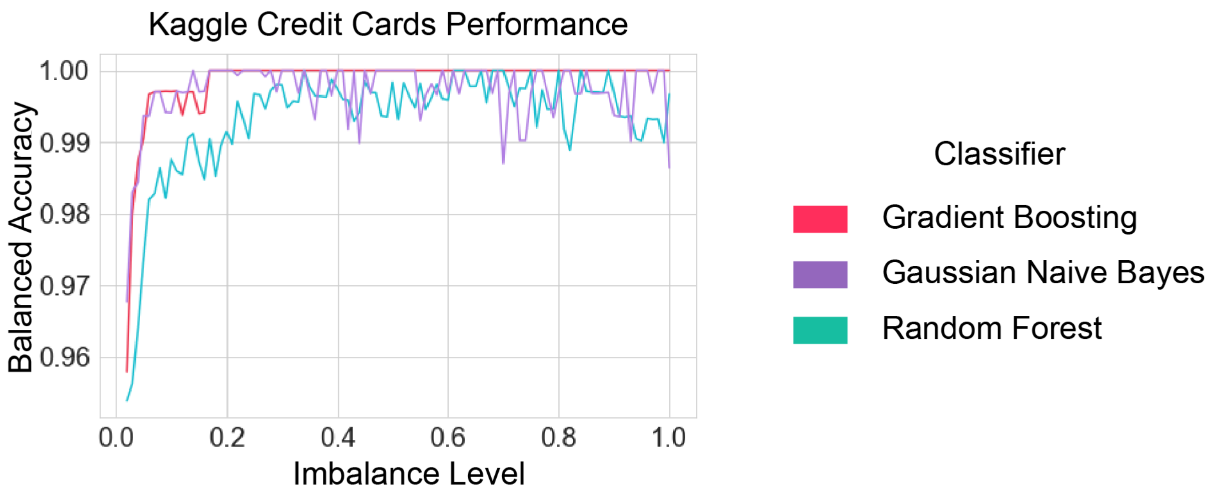


Figure 4.5: Classifier Performance on the Kaggle Credit Cards data over imbalance levels from 0 to 1. Note that no resampling is used.

classification attempted 100 times by the respective classifier. Again, balanced accuracy converges to 1 as the imbalance level approaches 1, and the worst performance for all classifiers occurs for imbalance levels such that $0 < \rho < 0.2$. Unlike in the single-attempt classification shown in Figure 4.5, Gaussian Naive Bayes performs best in the simulation.

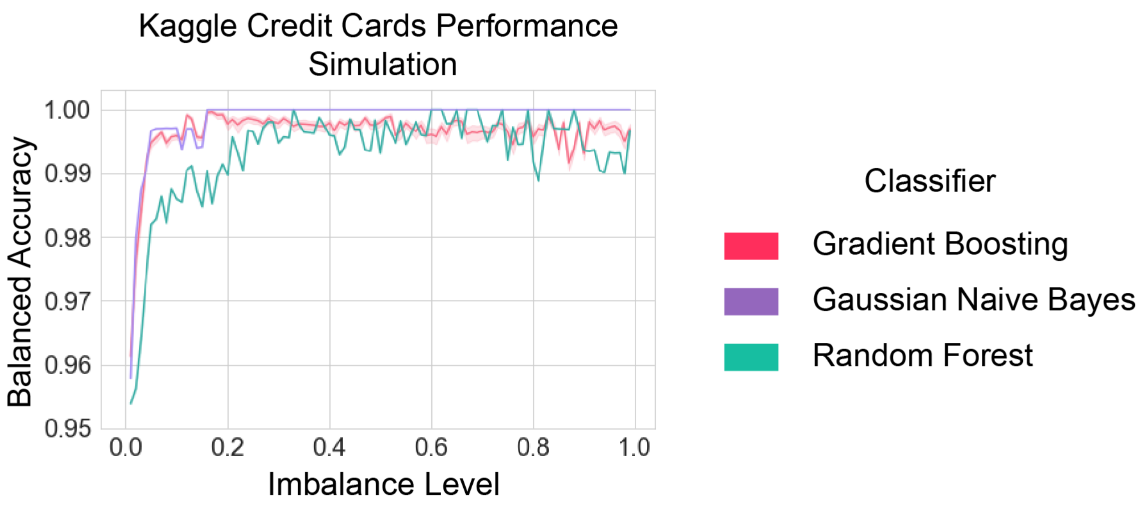


Figure 4.6: Classifier Performance on the Kaggle Credit Cards data over imbalance levels from 0 to 1 for $N = 100$ classification attempts for each classifier. Note that no resampling is used.

Comparison to Generated Data Results

The Kaggle Credit Cards experiments behave similarly to the generated data experiments, particularly for the easy case in the generated data, where $X \sim N(0, 1)$ and $Y \sim N(5, 1)$. In both, we see that balanced accuracy increases rapidly, converging to 1 after slightly lower balanced accuracy for imbalance levels where $0 < \rho < 0.2$. Even at these lower imbalance levels, however, both achieve over 95% balanced accuracy in both the individual classification experiments and the simulated experiment, indicating that the Kaggle Credit Cards data is likely relatively easy to classify based on the features provided.

4.3.3 Conclusions

The Kaggle Credit Cards data confirms the trends shown in the generated data experiments in Chapter 3. It seems that most classifiers perform well on easy classification tasks, like this one, even when the imbalance level is extremely low, making classification more difficult.

Chapter 5

Loss Function Development

One method of promoting better performance when training neural networks on imbalanced data involves modifying how the models determine loss. This is known as loss function development or loss function engineering (Johnson and Khoshgoftaar, 2019). This algorithm-level method uses theoretical ideas to modify canonical loss functions in ways that allow the minority class to contribute more to the loss, minimizing incorrectly-classified minority instances (Janocha and Czarnecki, 2017).

Traditionally, neural network loss functions are often cross-entropy loss functions. Generally, Cross-Entropy (CE) is defined as

$$\text{CE} = -\sum_{c=1}^{C} y_{i,c} \log(p_{i,c}),$$

where C represents the number of possible classes, $y_{i,c}$ is a binary variable to represent whether the label c is correct for observation i, and $p_{i,c}$ is the predicted probability that observation i is in class c. This form is often used for multi-class classification problems, but we are interested in binary classification problems.

For a binary classification problem, we can simplify the CE loss and define Binary Cross-Entropy (BCE) as follows:

$$\text{BCE} = -(y_i \log(p_i) + (1 - y_i) \log(1 - p_i)).$$

It is clear that the standard BCE loss function weighs both the majority and minority classes equally. Because the cost of misclassifying a minority class instance is much higher than misclassifying a majority class instance when dealing with imbalanced data, this loss function can be detrimental to model performance.

5.1 Loss Functions

Although relatively little research on loss function engineering exists, we discuss four loss functions aimed at improving classification of imbalanced datasets. We discuss the theoretical development of these loss functions and how they improve classification.

5.1.1 Focal Loss

The Focal Loss function is typically used in object detection problems, where negative background examples outnumber positive foreground examples. This loss function combats this issue by reshaping the standard CE loss function to reduce the burden of easily classified instances (Johnson and Khoshgoftaar, 2019; Lin et al., 2017).

This loss function multiplies the CE loss by a parameter $\alpha_i(1-p_i)^\gamma$, where $\alpha_i \geq 0$ is a weight to increase the importance of the minority class and γ adjusts the rate at which easily classified instances are downweighted (Johnson and Khoshgoftaar, 2019). The loss function is then given by

$$FL = -\alpha_i(1-p_i)^\gamma \log(p_i).$$

By focusing learning on difficult minority class instances, Focal Loss improves both speed and accuracy compared to the standard CE loss function it is built upon (Lin et al., 2017).

5.1.2 Label-Distribution-Aware Margin (LDAM) Loss

This loss function manipulates the margins of the training instances. The loss function pushes the boundary separating the classes towards the majority class, allowing more room for generalization error around the minority class, as shown in Figure 5.1 (Cao et al., 2019). In other words, by pushing the class separation boundary towards the majority class, LDAM allows more room for what would be misclassified minority class instances under a typical, equidistant margin.

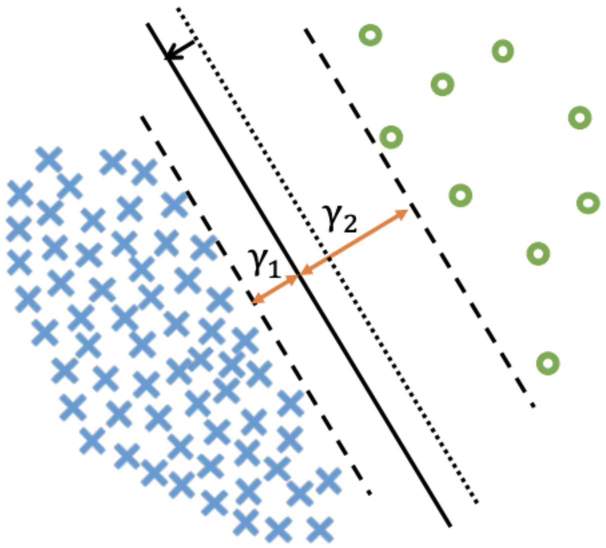


Figure 5.1: LDAM pushes the margin away from the minority class (green circles) and towards the majority class (blue X's) (Cao et al., 2019, Figure 1).

The loss is defined as

$$L = \max(\max_{j \neq y}\{z_j\} - z_y + \Delta_y, 0),$$

where $\Delta_j = \frac{C}{n_j^{1/4}}$, for $j \in \{1, \ldots, k\}$. Note that Δ_j represents the trade-off between class margins (Cao et al., 2019).

5.1.3 Gradient Harmonizing Mechanism (GHM) Loss

Classification methods often perform poorly when faced with a disparity in the number of positive and negative instances or in the number of easily-classified and hard to classify instances. The effects of these problems can be captured by the gradient norm distribution (B. Li, Y. Liu and X. Wang, 2019).

Suppose we have an imbalanced dataset. Then we know that the negative instances in the dataset are generally easy for a neural network or classifier to classify. These instances don't contribute significantly to the model because they are easily classifiable, meaning they produce a small magnitude of gradient. Positive instances, on the other hand, are much more difficult to classify, so they correspond to large magnitudes of gradient.

Although negative instances have small gradients that contribute little to the global gradient, the high amount of these easily classified points overwhelm neural networks. This disparity overwhelms the minority class, making the training process inefficient (B. Li, Y. Liu and X. Wang, 2019).

The Gradient Harmonizing Mechanism (GHM) loss improves the training process by calculating loss on instances with similar gradient densities and subsequently attaches a harmonizing parameter to the gradient of each instance according to its density (B. Li, Y. Liu and X. Wang, 2019). GHM loss is defined as follows:

$$L_{\mathrm{GHM}} = \frac{1}{N}\sum_{1}^{N} \beta_i L_{\mathrm{CE}}(p_i, p_i^*).$$

5.1.4 Mean False Error (MFE)

Instead of focusing on the cross-entropy loss function, the traditional Mean Squared Error (MSE) loss function performs poorly with imbalanced data (S. Wang et al., 2016). The MSE loss function compares true values, y, and predicted values, $\hat{y}$, as follows:

$$L(y, \hat{y}) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2.$$

Instead, Mean Squared False Error (MSFE) and Mean False Error (MFE) are both more sensitive to the minority class, improving the loss function's classification potential for imbalanced data (S. Wang et al., 2016). By splitting the MSE into two components, mean false positive error (FPE) and mean false negative error (FNE), and recombining them into the total loss (MFE), the loss function becomes particularly sensitive to the minority class. We define FPE and FNE below:

$$\mathrm{FPE} = \frac{1}{\mathbf{N}}\sum_{i=1}^{\mathbf{N}}\sum_{n}\frac{1}{2}(y - \hat{y}_n)^2$$

$$\mathrm{FNE} = \frac{1}{\mathbf{P}}\sum_{i=1}^{\mathbf{P}}\sum_{n}\frac{1}{2}(y - \hat{y}_n)^2$$

Note that **N** and **P** represent the number of instances in the negative and positive classes, respectively. To get the MFE, we simply sum these components,

$$MFE = FPE + FNE,$$

and take the sum of the squared values to get the MSFE:

$$MSFE = FPE^2 + FNE^2.$$

Note that the implementation of this function is not shown, but we use this to show that state-of-the-art loss functions are not only based on cross-entropy loss. This one, like some others, is based on Mean Square Error (MSE).

5.2 Methods

Our goal is to test these loss functions on the Kaggle Credit Cards dataset and compare their performance to that of the standard cross entropy loss function used in previous experiments. To do so, however, we use a different neural network module in Python called Pytorch to simplify this process (*Pytorch* n.d.). We discuss the data as well as the neural network structure and implementation of these loss functions using Pytorch.

5.2.1 Dataset

As described in earlier chapters, the Kaggle Credit Cards dataset contains information about the validity of credit card transactions. The dataset contains information from European card transactions, where 492 out of 284,807, or 0.172%, are fraudulent (*Credit Card Fraud Detection* n.d.). We use a 70-30 train-test split, and we use the `StandardScaler()` from the Sci-kit learn module to standardize the data (Pedregosa et al., 2011).

5.2.2 Neural Network

We create an Artificial Neural Network (ANN) to test the loss functions. To determine the optimal neural network structure and hyperparameters, we test several combinations of layers and hyperparameters.

As in previous experiments, we only use a ReLU activation function. The layers, however, include Dense, Batch Normalization, and Dropout layers, which have not been discussed. In each layer, we specify the number of inputs and outputs in the layer, and the number of outputs is then used as the input in the following layer. We elaborate on the use of the layers below.

Dense Layer

In dense layers, all the nodes are connected to the nodes of the previous layer. These layers each apply a linear transformation to the data such that $\mathbf{y} = \mathbf{x} \cdot \mathbf{w}^T + \mathbf{b}$. Here, the size of the vector $\mathbf{y}$ and $\mathbf{x}$ equate to the output dimension and input dimension, respectively (*Pytorch* n.d.).

Batch Normalization Layer

Batch normalization essentially allows us to normalize layer inputs. This both improves training speed of the ANN and its performance. Contrary to standard normalization layers, batch normalization refers to the grouping of data into batches, or smaller parts of the original dataset (Ioffe and Szegedy, 2015).

Dropout

Dropout prevents the ANN from considering certain nodes during training on a particular forward or backward pass. Specifically, nodes are excluded at a certain probability $1 - p$, where p is specified in the call to a dropout layer. These layers improve the ANN's performance by avoiding overfitting to the training data (Budhiraja, 2018).

Structure

Using these layers, we implement the structure below in the specified order. The number of hidden nodes is kept constant at 50, shown in the input and output dimensions for each layer in Table 5.1. We use a learning rate of 10^{-4}, run the ANN for 100 epochs, and use a batch size of 16.

Layer	Input Dimension	Output Dimension
Dense + ReLU	30	50
Batch Normalization (1D)	50	50
Dense + ReLU	50	50
Batch Normalization (1D)	50	50
Dropout ($p = 0.3$)	50	50
Dense	50	1

Table 5.1: Structure of the neural network in the Kaggle Credit Card experiments.

5.2.3 Implementation

Using Pytorch, we can embed our custom loss functions into the training function by changing our `criterion`. If we use a standard loss function, we can simply call it from the Pytorch neural network module, which we import with the `import torch.nn as nn` command, as follows: `criterion = nn.CrossEntropyLoss()`. For new, modified loss functions, we instead call our pre-defined function. For the Focal Loss Function, defined as `FocalLoss`, for instance, our criterion changes to `criterion = FocalLoss()`.

The basic process of implementing neural networks in Pytorch is shown in Figure 5.2. We first define our neural network with the layers specified in Table 5.1. Then, we define our loss function by modifying the `criterion`, as discussed above. After training our model over 100 epochs, we test our model and calculate its balanced accuracy.

We use Pytorch to implement the layers discussed in Table 5.1. The commands for each respective layer are shown in Table 5.2. Note that the dimension arguments in each layer depend on the amount of nodes in previous layers, or their dimension.

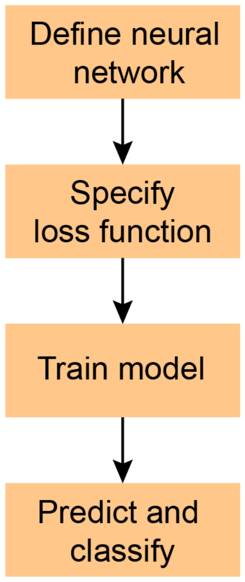


Figure 5.2: Process of creating a neural network structure for these experiments.

Layer or Transformation	**Implementation**
Dense	`nn.Linear(input_dim, output_dim)`
ReLU	`nn.ReLU()`
Batch Normalization	`nn.BatchNorm1d(hidden_nodes)`
Dropout	`nn.Dropout(p)`

Table 5.2: Pytorch implementation of the layers used in experiments.

Loss Functions

We also use Pytorch to implement each of our loss functions. In each loss function class, we have an `__init__()` and a `forward()` function. The former allows us to initialize necessary parameters and objects used to calculate loss, while the latter computes the loss for a pass of data.

5.3 Results

We see radically different performance in the different loss functions used in the ANNs. Binary Cross Entropy (BCE), our experimental control, performs similarly to Gradient Harmonized (GHM) Loss, as each achieve a relatively high balanced accuracy on the test set. BCE achieves the highest balanced accuracy of 0.831 and GHM achieves a balanced accuracy of 0.634. Focal Loss and Label-Distribution-Aware Margin (LDAM)

Loss, however, perform significantly worse, with balanced accuracy of 0.249 and 0.289 on the test set, respectively. The results of the experiments in the neural networks are shown in Table 5.3.

Loss Function	**Balanced Accuracy**
Binary Cross Entropy Loss	0.831
Focal Loss	0.249
Gradient Harmonized Loss	0.634
Label-Distribution-Aware Margin Loss	0.289

Table 5.3: ANN performance on the test set using different loss functions.

Although all the loss functions shown above are based on Binary Cross Entropy, the original loss function performs significantly better than its variations. Because both Focal Loss and LDAM Loss both use geometric-based approaches to improving classification in imbalanced data, it seems that these approaches may not be beneficial with the Kaggle Credit Cards data. GHM, on the other hand, uses the gradient densities of observations. It is possible that class separation is difficult for the two geometric methods to detect, making GHM perform favorably in comparison.

5.4 Conclusions

Although BCE fails to account for class imbalance, it performs better when in an ANN trained on the Kaggle Credit Cards dataset than our three state-of-the-art loss functions of interest: GHM, LDAM, and Focal Loss. It's possible that Focal Loss and LDAM perform significantly worse than BCE and GHM because they take a geometric approach to classifying imbalanced data, which may not work well for this data.

In the future, it may be interesting to test these loss functions on different datasets and manipulate the imbalance levels of the datasets to determine whether there are patterns in their performance.

Chapter 6

Conclusion

In recent years, classifiers and neural networks have been shown to perform well on classification tasks involving a wide variety of data types. When attempting to classify imbalanced data, where the number of instances in one class significantly exceeds that of the other, these techniques perform poorly, misclassifying the majority of positive class instances because they optimize for accuracy.

As a result, the accuracy performance metric can be misleading, so we consider balanced accuracy, a measure of accuracy weighting the positive and negative classes equally. To determine the amount of imbalance in a dataset and understand its effects, we use the imbalance parameter ρ, defined as $\rho = \frac{m_p}{m_n}$, or the ratio of positive to negative instances in the dataset.

To better understand the effects of imbalance levels on classification, we generate synthetic data with differing class separability, which impacts the difficulty of classification. When we test the performance of neural networks on these datasets at imbalance levels of $\rho = 0.05$, 0.1, and 0.2, we see that SMOTE resampling doesn't improve performance. Increasing the imbalance level improved performance for "medium hard" and "hard" classification tasks but not for "medium easy" and "easy" classification tasks.

Generalizing this to all imbalance levels and testing classifiers on this data, we see that balanced accuracy is lower and fluctuates most for $0 < \rho < 0.2$, after which all the classifiers converge to a certain balanced accuracy value. For harder classification tasks, this final balanced accuracy value was lower than that of easy tasks, and the convergence over imbalance levels occurred more slowly in classification for harder tasks. Simulated results for 100 classification attempts at each imbalance level support this conclusion.

In real-world datasets, we can use PCA to determine difficulty of classification tasks. When we use neural networks to classify combinations of labels, both easy and hard tasks, with the Intel Images dataset, we see that classifiers achieve a higher balanced accuracy on easy tasks than hard tasks on average. These CNNs consistently performed best on higher imbalance levels, as expected.

Our numerical data experiments using the Kaggle Credit Cards dataset also support this conclusion. Like in the generated data experiments, SMOTE resampling had no effect on classifier performance, and the simulation of 100 classification attempts at each imbalance level from 0.01 to 1 reinforce our conclusion.

Typically, neural networks minimize loss functions that encourage accuracy maximization, not weighting the minority class more heavily to account for the dearth of instances in the class. Some state-of-the-art loss functions attempt to account for this deficiency in standard loss functions, like BCE. When we classify the Kaggle Credit Cards data with these loss functions, however, none of them perform as well as BCE. The two loss functions with significantly worse performance, LDAM and FL, both use geometric interpretations of class imbalance, which may not be appropriate for this dataset.

We have shown that higher imbalance levels improve classifier and neural network performance on imbalanced data for harder classification tasks more than they do for easier classification tasks. For these harder tasks, SMOTE resampling will likely improve results by a large margin with classifiers, but resampling may not improve results with neural networks as significantly. Modifying loss functions in neural networks may improve results, but this likely depends on the dataset and the difficulty of the classification task.

Corrections

May 17, 2021

1. Fixed minor spelling and typographical errors throughout the thesis.
2. Changed the order of contents on the title page to follow the Amherst College Thesis Guidelines.
3. Added “and Artificial Neural Networks (ANNs)” to the third paragraph of the introduction.
4. Included “classification using numerical data with ANNs” in the last paragraph of the introduction.
5. Switched the order of “bagging” and “Bootstrap aggregation, or bagging” (Section 2.1.1, page 7).
6. Changed T_b to T_k in the description of the Random Forest Algorithm, changed “random forest tree” to “decision tree” in step two, and removed “terminal” from the description of node in step two (Section 2.1.2, page 8).
7. Changed subtraction to addition in the definition of a binary classifier threshold function (Section 2.2, page 12).
8. Removed the last four sentences of the Backpropogation section and added, “We can then use gradient descent, discussed in detail in Section 2.1.3, to minimize the cost, or loss, function” (Section 2.2.1, page 13).
9. Added the following sentence: “This also shows that Gradient Boosting performs worse than Random Forest, particularly for the more difficult classification tasks” (Section 3.5.5, page 40).

References

Adam — latest trends in deep learning optimization (n.d.). https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c. Accessed: 2021-04-08 (cit. on p. 17).

Bansal, Puneet (Jan. 2019). *Intel Image Classification*. URL: https://www.kaggle.com/puneet6060/intel-image-classification (cit. on p. 42).

Breiman, Leo et al. (2001). 'Statistical modeling: The two cultures (with comments and a rejoinder by the author)'. In: *Statistical science* 16.3, pp. 199–231 (cit. on p. 7).

Budhiraja, Amar (Mar. 2018). *Learning Less to Learn Better-Dropout in (Deep) Machine learning*. URL: https://medium.com/@amarbudhiraja/https-medium-com-amarbudhiraja-learning-less-to-learn-better-dropout-in-deep-machine-learning-74334da4bfc5 (cit. on p. 59).

Cao, Kaidi et al. (2019). 'Learning imbalanced datasets with label-distribution-aware margin loss'. In: *arXiv preprint arXiv:1906.07413* (cit. on pp. 55, 56, 109).

Chawla, Nitesh V et al. (2002). 'SMOTE: synthetic minority over-sampling technique'. In: *Journal of artificial intelligence research* 16, pp. 321–357 (cit. on p. 22).

Credit Card Fraud Detection (n.d.). URL: https://www.kaggle.com/mlg-ulb/creditcardfraud (cit. on pp. 19, 50, 58).

Efron, Bradley and Trevor Hastie (2016). *Computer Age Statistical Inference: Algorithms, Evidence, and Data Science*. 1st. USA: Cambridge University Press. ISBN: 1107149894 (cit. on pp. 11, 13).

Gayathri, BM and CP Sumathi (2016). 'An automated technique using Gaussian naıve Bayes classifier to classify breast cancer'. In: *International Journal of Computer Applications* 148.6, pp. 16–21 (cit. on pp. 6, 7).

Greenwell, Bradley Boehmke amp; Brandon (Feb. 2020). *Hands-On Machine Learning with R*. URL: https://bradleyboehmke.github.io/HOML/gbm.html (cit. on pp. 9, 10).

Hastie, Trevor, Robert Tibshirani and Jerome Friedman (2009). *The elements of statistical learning: data mining, inference and prediction*. 2nd ed. Springer. URL: http://www-stat.stanford.edu/~tibs/ElemStatLearn/ (cit. on p. 8).

Ioffe, Sergey and Christian Szegedy (2015). 'Batch normalization: Accelerating deep network training by reducing internal covariate shift'. In: *International conference on machine learning*. PMLR, pp. 448–456 (cit. on p. 59).

Janocha, Katarzyna and Wojciech Marian Czarnecki (2017). 'On loss functions for deep neural networks in classification'. In: *arXiv preprint arXiv:1702.05659* (cit. on p. 54).

Johnson, Justin M and Taghi M Khoshgoftaar (2019). 'Survey on deep learning with class imbalance'. In: *Journal of Big Data* 6.1, p. 27 (cit. on pp. 6, 19, 21–23, 25, 54, 55).

Krawczyk, Bartosz (2016). 'Learning from imbalanced data: open challenges and future directions'. In: *Progress in Artificial Intelligence* 5.4, pp. 221–232 (cit. on pp. 20, 21).

Li, Buyu, Yu Liu and Xiaogang Wang (2019). 'Gradient harmonized single-stage detector'. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 33. 01, pp. 8577–8584 (cit. on pp. 56, 57, 107).

Li, Fei-Fei (2018). *Convolutional Neural Networks (CNNs/ConvNets)*. URL: `%7Bhttps://cs231n.github.io/convolutional-networks/%7D` (cit. on pp. 15, 16).

Liaw, Andy, Matthew Wiener et al. (2002). 'Classification and regression by randomForest'. In: *R news* 2.3, pp. 18–22 (cit. on p. 8).

Lin, Tsung-Yi et al. (2017). 'Focal loss for dense object detection'. In: *Proceedings of the IEEE international conference on computer vision*, pp. 2980–2988 (cit. on pp. 55, 106).

Ling, Charles X and Victor S Sheng (2008). *Cost-sensitive learning and the class imbalance problem* (cit. on pp. 22, 23).

Liu, Weibo et al. (2017). 'A survey of deep neural network architectures and their applications'. In: *Neurocomputing* 234, pp. 11–26 (cit. on p. 15).

Luque, Amalia et al. (2019). 'The impact of class imbalance in classification performance metrics based on the binary confusion matrix'. In: *Pattern Recognition* 91, pp. 216–231 (cit. on p. 28).

Nielsen, Michael A (2015). *Neural networks and deep learning*. Vol. 2018. Determination press San Francisco, CA (cit. on pp. 11, 12).

Nwankpa, Chigozie et al. (2018). 'Activation functions: Comparison of trends in practice and research for deep learning'. In: *arXiv preprint arXiv:1811.03378* (cit. on p. 12).

Pedregosa, F. et al. (2011). 'Scikit-learn: Machine Learning in Python'. In: *Journal of Machine Learning Research* 12, pp. 2825–2830 (cit. on pp. 6, 58).

Pytorch (n.d.). `pytorch.org`. Accessed: 2021-03-23 (cit. on pp. 58, 59).

Skansi, Sandro (2018). *Introduction to Deep Learning - From Logical Calculus to Artificial Intelligence*. Undergraduate Topics in Computer Science. Springer. ISBN: 978-3-319-

73003-5. DOI: 10.1007/978-3-319-73004-2. URL: https://doi.org/10.1007/978-3-319-73004-2 (cit. on pp. 12–15).

Velickovic, Petar (2017). 'How to get a neural network to do what you want?' University Lecture (cit. on pp. 16, 17).

Wang, Shoujin et al. (2016). 'Training deep neural networks on imbalanced data sets'. In: *2016 international joint conference on neural networks (IJCNN)*. IEEE, pp. 4368–4374 (cit. on p. 57).

Weaver, Cliff (Aug. 2017). URL: http://rismyhammer.com/ml/keras.html (cit. on p. 16).

Weiss, Gary M (2004). 'Mining with rarity: a unifying framework'. In: *ACM Sigkdd Explorations Newsletter* 6.1, pp. 7–19 (cit. on p. 22).

Zwitter, M and M Soklic (1998). 'This breast cancer domain was obtained from the University Medical Centre'. In: *Institute of Oncology, Ljubljana, Yugoslavia* (cit. on p. 23).

Appendix A
Synthetic data

This section shows the code used for experiments involving synthetic data. Note that the code for all the experiments shown below, as well as all the code for other experiments, is available on Github at `https://github.com/Amherst-Statistics/Boskovic-Andrea-Thesis`.

A.1 Synthetic Data Classification

These scripts show both the synthetic data generator function code and sample experiments for classifiers (Section A.1.1) and neural networks (Section A.1.2). We also include scripts used to make figures to visualize results for both classifier and neural network experiments.

A.1.1 Synthetic Data Test: Classifiers

The data generation function is shown below in lines 22-66, followed by an example classification experiment where we use the following parameters:

- `n_pos = 100`
- `n_neg = 2000`
- `n_features = 2`
- `n_inf_features = 2`
- `sd = 1`
- `class1_mean = 5`.

Note that these values can be changed to create any type of imbalanced dataset. Using this dataset defined in line 69, we create a training and testing set both with and without SMOTE resampling. Finally, we test the performance of Gradient Boosting, Random Forest, and Gaussian Naive Bayes on each of these datasets. In line 122, we also display the code to create Figure 3.3 (the hard case) and Figure 3.2 (the easy case).

```
1 ##### Import packages #####
2 
3 # Data wrangling
4 import random
5 import math
```

```
import numpy as np
import pandas as pd

# Plotting
import seaborn as sns
import matplotlib.pyplot as plt
import scipy.stats as stats

# Testing function
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.naive_bayes import GaussianNB
from imblearn.over_sampling import SMOTE
from sklearn.metrics import balanced_accuracy_score,
    classification_report
from sklearn.model_selection import train_test_split

def imbalanced_data_generator(n_pos,
                              n_neg,
                              n_features,
                              n_inf_features,
                              sd,
                              class1_mean):
    """ Imbalanced Data Generator

    Inputs:
        n_pos (int): number of positive instances in dataset
        n_neg (int): number of negative instances in dataset
        n_features (int): number of features in dataset
        n_inf_features (int): number of informative features in
                              predicting target
        sd (float): standard deviation of the data
        class1_mean (float): desired mean for class 1
    """
    class0 = np.array(np.random.normal(loc = 0, scale = 1, size = (n_neg
    *n_features)))
    class0 = np.reshape(class0, (n_neg, n_features))
    class0 = pd.DataFrame(class0)

    class1 = np.array(np.random.normal(loc = class1_mean, scale = sd,
    size = (n_pos*n_inf_features)))
    class1 = np.reshape(class1, (n_pos, n_inf_features))
    class1_cols = np.r_[(n_features-n_inf_features):(n_features+
    n_inf_features-2)]
    class1 = pd.DataFrame(class1, columns = class1_cols)

    classr1 = np.array(np.random.normal(loc = 0, scale = 1, size = (
    n_pos*(n_features-n_inf_features))))
    classr1 = np.reshape(classr1, (n_pos, (n_features-n_inf_features)))
    classr1 = pd.DataFrame(classr1)

    c1_full_names = np.r_[0:(n_features)]
    class1_full = pd.concat([class1, classr1], axis=1)
    class1_full = pd.DataFrame(class1_full, columns = c1_full_names)
```

```python
    data = pd.concat([class0, class1_full], axis = 0)

    labels = ["0", "1"]
    full_labels = np.repeat(labels, [n_neg, n_pos])
    full_labels = pd.DataFrame(full_labels)
    full_labels = full_labels.rename(columns = {0:'target'})

    full_data = np.append(data, full_labels.to_numpy(), axis = 1)
    full_data = pd.DataFrame(full_data)
    full_data.columns = [*full_data.columns[:-1], 'target']

    return(full_data)

##### Sample values
df2 = imbalanced_data_generator(n_pos = 100,
                                n_neg = 2000,
                                n_features = 2,
                                n_inf_features = 2,
                                sd = 1,
                                class1_mean = 5)

# Create datasets, train/test splits, initialize SMOTE
X2 = df2.drop('target', axis=1)
y2 = df2[['target']]
X_train, X_test, y_train, y_test = train_test_split(X2,
                                                    y2,
                                                    test_size = 0.30,
                                                    random_state = 20)
smote = SMOTE(random_state = 20)
X_train_smote, y_train_smote = smote.fit_resample(X_train,y_train)

# GB: No resampling
gb = GradientBoostingClassifier(random_state = 20)
gb_pred = gb.fit(X_train, y_train).predict(X_test)
print(classification_report(y_test, gb_pred))
print(round(balanced_accuracy_score(y_test, gb_pred), 3))

# GB: SMOTE resampling
gb = GradientBoostingClassifier(random_state = 20)
gb_pred_smote = gb.fit(X_train_smote, y_train_smote).predict(X_test)
print(classification_report(y_test, gb_pred_smote))
print(round(balanced_accuracy_score(y_test, gb_pred_smote), 3))

# RF: No resampling
rf = RandomForestClassifier(random_state = 20)
rf_pred = rf.fit(X_train, y_train).predict(X_test)
print(classification_report(y_test, rf_pred))
print(round(balanced_accuracy_score(y_test, rf_pred), 3))

# RF: SMOTE resampling
rf = RandomForestClassifier(random_state = 20)
rf_pred_smote = rf.fit(X_train_smote, y_train_smote).predict(X_test)
print(classification_report(y_test, rf_pred_smote))
print(round(balanced_accuracy_score(y_test, rf_pred_smote), 3))
```

```

# GNB: No resampling
gnb = GaussianNB()
gnb_pred = gnb.fit(X_train, y_train).predict(X_test)
print(classification_report(y_test, gnb_pred))
print(balanced_accuracy_score(y_test, gnb_pred))

# GNB: SMOTE resampling
gnb = GaussianNB()
gnb_pred_smote = gnb.fit(X_train_smote, y_train_smote).predict(X_test)
print(classification_report(y_test, gnb_pred_smote))
print(round(balanced_accuracy_score(y_test, gnb_pred_smote), 3))

##### Figures #####

# Hard case: X ~ N(0,1) and Y ~ N(1,1)
mu_0 = 0
mu_1 = 1
variance = 1
sigma = math.sqrt(variance)
x = np.linspace(mu_0 - (3 * sigma), mu_0 + (3 * sigma), 100)
x_1 = np.linspace(mu_1 - (3 * sigma), mu_1 + (3 * sigma), 100)

plt.plot(x, stats.norm.pdf(x, mu_0, sigma))
plt.plot(x_1, stats.norm.pdf(x_1, mu_1, sigma))
plt.title('Hard Case: $X\sim N(0,1)$ and $Y\sim N(1,1)$')
plt.show()

# Easy case: X ~ N(0,1) and Y ~ N(5,1)
mu_0 = 0
mu_1 = 5
variance = 1
sigma = math.sqrt(variance)
x = np.linspace(mu_0 - 3*sigma, mu_0 + 3*sigma, 100)
x_1 = np.linspace(mu_1 - 3*sigma, mu_1 + 3*sigma, 100)
plt.plot(x, stats.norm.pdf(x, mu_0, sigma))
plt.plot(x_1, stats.norm.pdf(x_1, mu_1, sigma))
plt.title('Easy Case: $X\sim N(0,1)$ and $Y\sim N(5,1)$')
plt.show()
```

Generating Classifier Results Figure

After collating results of the classifier experiments, where we vary `sd` and `class1_mean` parameters to test different difficulty settings, into a csv file, we create a visualization of these results in R. The code to generate this visualization, shown in Figure 3.4, is given below.

We tidy each separate dataset, where `type1` represents the easiest case, `type2` represents a medium case, and `type3` represents the hardest case. Each setting is defined in Table A.1 for reference.

After combining these datasets in line 57, which we refer to as `full_df`, we plot the balanced accuracy of each experiment. In the plot, we use color to differentiate between imbalance levels, shape to differentiate between easy and hard difficulty levels in each experiment, and shape to show whether or not we use SMOTE resampling.

Experimental Setting	Code Reference	Distribution Comparison
Easy	`type1`	$X \sim N(0,1)$, $Y \sim N(5,3)$
Medium	`type2`	$X \sim N(0,1)$, $Y \sim N(3,3)$
Hard	`type3`	$X \sim N(0,1)$, $Y \sim N(2,3)$

Table A.1: Experimental settings for the generated figure.

```r
# Load packages
library(tidyverse)
library(latex2exp)
library(ggpubr)

# Load data
type1 <- read_csv("type1.csv") # Easiest
type2 <- read_csv("type2.csv") # Medium
type3 <- read_csv("type3.csv") # Hardest

# The difficulty comes from the original excel dataset
difficulty <- c(
  "hard", "hard", "hard", "hard", "hard", "hard",
  "easy", "easy", "easy", "easy", "easy", "easy"
)

# Tidy type1
type1 <- cbind(type1, difficulty) %>%
  mutate(id = row_number())
type1_tidy <- gather(type1, key = "imbalance_lvl",
                     value = "bal_acc", -c("difficulty", "id")) %>%
  mutate(imbalance_lvl = readr::parse_number(imbalance_lvl)) %>%
  mutate(
    imbalance_lvl = as.factor(imbalance_lvl),
    resample = ifelse(id %% 2 == 0, "SMOTE", "None"),
    id = as.factor(id),
    type = "Easy Distributions"
  )

# Tidy type2
type2 <- cbind(type2, difficulty) %>%
  mutate(id = row_number())
type2_tidy <- gather(type2, key = "imbalance_lvl",
                     value = "bal_acc", -c("difficulty", "id")) %>%
  mutate(imbalance_lvl = readr::parse_number(imbalance_lvl)) %>%
  mutate(
    imbalance_lvl = as.factor(imbalance_lvl),
    resample = ifelse(id %% 2 == 0, "SMOTE", "None"),
    id = as.factor(id),
    type = "Medium Easy Distributions"
  )

# Tidy type3
```

```r
type3 <- cbind(type3, difficulty) %>%
  mutate(id = row_number())
type3_tidy <- gather(type3, key = "imbalance_lvl",
                     value = "bal_acc", -c("difficulty", "id")) %>%
  mutate(imbalance_lvl = readr::parse_number(imbalance_lvl)) %>%
  mutate(
    imbalance_lvl = as.factor(imbalance_lvl),
    resample = ifelse(id %% 2 == 0, "SMOTE", "None"),
    id = as.factor(id),
    type = "Hard Distributions"
  )

# Combine datasets
full_df <- rbind(type1_tidy, type2_tidy, type3_tidy)
full_df$type <- factor(full_df$type,
                       levels = c("Easy Distributions",
                                  "Medium Easy Distributions",
                                  "Hard Distributions"))

# Plot
ggplot(full_df, aes(x = id, y = bal_acc, color = imbalance_lvl,
                    shape = difficulty, size = resample)) +
  geom_point() +
  facet_wrap(~ type) +
  labs(
    x = "Experiment Number",
    y = "Balanced Accuracy",
    shape = "Difficulty",
    color = "Imbalance Level",
    size = "Resampling Method"
  ) +
  theme_bw() +
  theme(
    legend.position = "none",
    strip.text.x = element_text(size = 13)
  )

# Save plot
ggsave("imbalance_diff.png")
```

A.1.2 Synthetic Data Test: Neural Networks

We also test classification performance of neural networks on synthetic datasets. We use the following parameters in this sample experiment, shown in line 66:

- `n_pos = 100`
- `n_neg = 2000`
- `n_features = 2`
- `n_inf_features = 2`
- `sd = 1`

- `class1_mean = 1`.

Since we define the mean of `class1` to be 1 and its standard deviation to be 1, this sample code shows the results of a hard case because mean and standard deviation of the other class, set by default in the `imbalanced_data_generator` function, are 0 and 1. Note that both `class1` and `class0` follow a Normal distribution with these specified parameters.

In this sample experiment, we set the following hyperparameters:

- `BATCH_SIZE = 16`
- `HIDDEN_NODES = 50`
- `EPOCHS = 100`
- `LEARNING_RATE = 0.0001`.

These hyperparameters can be changed to accommodate the desired experimental setting. The code for this experiment is shown below.

```python
# Import pacakges
import pandas as pd
import numpy as np
import random
import matplotlib.pyplot as plt
from itertools import chain
import imblearn
from imblearn.over_sampling import SMOTE

from sklearn.model_selection import train_test_split
from sklearn import preprocessing
from sklearn.preprocessing import StandardScaler
from sklearn import metrics
from sklearn.metrics import balanced_accuracy_score,
    classification_report

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader

# Function to generate data
def imbalanced_data_generator(n_pos,
                              n_neg,
                              n_features,
                              n_inf_features,
                              sd,
                              class1_mean):

    class0 = np.array(np.random.normal(loc = 0,
                                       scale = 1,
                                       size = (n_neg * n_features)))
    class0 = np.reshape(class0, (n_neg, n_features))
    class0 = pd.DataFrame(class0)

    class1 = np.array(np.random.normal(loc = class1_mean,
                                       scale = sd,
                                       size = (n_pos * n_inf_features)))
    class1 = np.reshape(class1, (n_pos, n_inf_features))
    class1_cols = np.r_[(n_features - n_inf_features):
                        (n_features + n_inf_features - 2)]
    class1 = pd.DataFrame(class1, columns = class1_cols)

    classr1 = np.array(np.random.normal(loc = 0,
                                        scale = 1,
                       size = (n_pos * (n_features - n_inf_features))))
    classr1 = np.reshape(classr1, (n_pos, (n_features - n_inf_features))
   )
    classr1 = pd.DataFrame(classr1)

    c1_full_names = np.r_[0:(n_features)]
    class1_full = pd.concat([class1, classr1], axis = 1)
    class1_full = pd.DataFrame(class1_full, columns = c1_full_names)
    data = pd.concat([class0, class1_full], axis = 0)

    labels = ["0", "1"]
    full_labels = np.repeat(labels, [n_neg, n_pos])
    full_labels = pd.DataFrame(full_labels)
    full_labels = full_labels.rename(columns = {0:'target'})

    full_data = np.append(data, full_labels.to_numpy(), axis = 1)
    full_data = pd.DataFrame(full_data)
    full_data.columns = [*full_data.columns[:-1], 'target']

    return(full_data)

# Run function to generate dataset
res = imbalanced_data_generator(n_pos = 100,
                                n_neg = 2000,
                                n_features = 2,
                                n_inf_features = 2,
                                sd = 1,
                                class1_mean = 1)

# Set hyperparameters
BATCH_SIZE = 16
HIDDEN_NODES = 50
EPOCHS = 100
LEARNING_RATE = 0.0001

# Define dataset, use 70:30 train:test split
X = res.drop('target', axis = 1)
y = res[['target']]
X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    test_size = 0.30,
                                                    random_state = 42)

# Standardize
```

```
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.fit_transform(X_test)

# Data Loader for training data
class trainData(Dataset):

    def __init__(self, X_data, y_data):
        self.X_data = X_data
        self.y_data = y_data

    def __getitem__(self, index):
        return self.X_data[index], self.y_data[index]

    def __len__ (self):
        return len(self.X_data)

y_train = y_train.astype(np.float32)
train_data = trainData(torch.FloatTensor(X_train),
                       torch.from_numpy(y_train.values))

# Data Loader for testing data
class testData(Dataset):

    def __init__(self, X_data):
        self.X_data = X_data

    def __getitem__(self, index):
        return self.X_data[index]

    def __len__ (self):
        return len(self.X_data)


test_data = testData(torch.FloatTensor(X_test))
train_loader = DataLoader(dataset = train_data, batch_size = BATCH_SIZE,
                          shuffle = True)
test_loader = DataLoader(dataset = test_data, batch_size = 1)

# Define neural network
class binaryClassification(nn.Module):
    def __init__(self):
        super(binaryClassification, self).__init__()
        # Number of input features is 2
        self.layer_1 = nn.Linear(2, HIDDEN_NODES)
        self.layer_2 = nn.Linear(HIDDEN_NODES, HIDDEN_NODES)
        self.layer_out = nn.Linear(HIDDEN_NODES, 1)

        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(p=0.3)
        self.batchnorm1 = nn.BatchNorm1d(HIDDEN_NODES)
        self.batchnorm2 = nn.BatchNorm1d(HIDDEN_NODES)

    def forward(self, inputs):
        x = self.relu(self.layer_1(inputs))
        x = self.batchnorm1(x)
        x = self.relu(self.layer_2(x))
        x = self.batchnorm2(x)
        x = self.dropout(x)
        x = self.layer_out(x)

        return x


# Define model, loss, optimizer
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model = binaryClassification()
criterion = nn.BCEWithLogitsLoss()
optimizer = optim.Adam(model.parameters(), lr = LEARNING_RATE)

# Train model
def binary_acc(y_pred, y_test):
    y_pred_tag = torch.round(torch.sigmoid(y_pred))
    correct_results_sum = (y_pred_tag == y_test).sum().float()
    acc = correct_results_sum/y_test.shape[0]
    acc = torch.round(acc * 100)
    return acc

model.train()

loss_vals=[]
for e in range(1, EPOCHS+1):
    epoch_loss = 0
    epoch_acc = 0

    for X_batch, y_batch in train_loader:
        X_batch, y_batch = X_batch.to(device), y_batch.to(device)

        optimizer.zero_grad()
        y_pred = model(X_batch)

        loss = criterion(y_pred, y_batch)
        acc = binary_acc(y_pred, y_batch)

        loss.backward()
        optimizer.step()

        epoch_loss += loss.item()
        epoch_acc += acc.item()

    loss_vals.append(epoch_loss/len(train_loader))

    if e % 10 == 0:
        print(f'Epoch {e+0:03}: | \
              Loss: {epoch_loss/len(train_loader):.5f} | \
              Acc: {epoch_acc/len(train_loader):.3f}')

plt.xlabel('Epochs')
```

```python
plt.ylabel('Loss')
plt.title('Loss per Epoch')
plt.plot(loss_vals)

# Test model
y_pred_list = []
model.eval()
with torch.no_grad():
    for X_batch in test_loader:
        X_batch = X_batch.to(device)
        y_test_pred = model(X_batch)
        y_test_pred = torch.sigmoid(y_test_pred)
        y_pred_tag = torch.round(y_test_pred)
        y_pred_list.append(y_pred_tag.cpu().numpy())
y_pred_list = [a.squeeze().tolist() for a in y_pred_list]
y_pred_list2 = [round(num) for num in y_pred_list]
y_test_new = list(chain.from_iterable(y_test.values.tolist()))
y_test_new = [int(value) for value in y_test_new]

# Determine results
print(classification_report(y_test_new, y_pred_list2))
print(balanced_accuracy_score(y_test_new,y_pred_list2))
```

Generating Neural Network Results Figures

The code below shows the R scripts for generating the results of the neural network experiments with and without SMOTE resampling using generated data, shown in Chapter 3.

This code generates Figure 3.5A, where we plot the balanced accuracy of the neural network on different difficulty levels at imbalance levels of $\rho = 0.05$, 0.1 and 0.2.

```r
# Load packages
library(tidyverse)
library(latex2exp)

# Load and tidy data
data <- read_csv("nn_datagen.csv")
data_tidy <- data %>%
  mutate(
    difficulty = as.factor(difficulty),
    imbalance_lvl = as.factor(imbalance_lvl),
    difficulty = fct_relevel(difficulty, "med_easy", after = 1),
    difficulty = fct_relevel(difficulty, "hard", after = 3)
  )

# Plot
ggplot(data_tidy, aes(x = imbalance_lvl, y = balanced_accuracy,
                      color = difficulty)) +
  geom_point(size = 3) +
  labs(
    title = "Balanced Accuracy in Data Generation Experiments
    \nUsing Neural Networks: No Resampling",
    x = TeX("Imbalance Level ($\\rho$)"),
    y = "Balanced Accuracy",
    color = "Difficulty"
  ) +
  theme_classic() +
  theme(
    plot.title = element_text(hjust = 0.5),
    plot.subtitle = element_text(hjust = 0.5)
  ) +
  scale_color_manual(
    labels = c("Easy", "Medium Easy", "Medium Hard", "Most Difficult"),
    values = c("#1e7640", "#de772d", "#e815dd", "4d88s4")
  ) +
  guides(color = guide_legend("Difficulty"))

# Save plot
ggsave("nn_imbalance_plt.png")
```

This code chunk generates Figure 3.5B, where we use the same neural network structure used for Figure 3.5A but resample the data with SMOTE.

```r
# Load packages
library(tidyverse)
library(latex2exp)

# Load data and clean data
data <- read_csv("nn_datagen_resample.csv")
data_tidy <- data %>%
  mutate(
    difficulty = as.factor(difficulty),
    imbalance_lvl = as.factor(imbalance_lvl),
    difficulty = fct_relevel(difficulty, "med_easy", after = 1),
    difficulty = fct_relevel(difficulty, "hard", after = 3)
  )

# Plot
ggplot(data_tidy, aes(x = imbalance_lvl, y = balanced_accuracy,
                      color = difficulty)) +
  geom_point(size = 3) +
  labs(
    title = "Balanced Accuracy in Data Generation Experiments
    \nUsing Neural Networks: SMOTE Resampling",
    x = TeX("Imbalance Level ($\\rho$)"),
    y = "Balanced Accuracy",
    color = "Difficulty"
  ) +
  theme_classic() +
  theme(
    plot.title = element_text(hjust = 0.5),
    plot.subtitle = element_text(hjust = 0.5)
  ) +
  scale_color_manual(
    labels = c("Easy", "Medium Easy", "Medium Hard", "Most Difficult"),
    values = c("#1e7640", "#de772d", "#e815dd", "4d88s4")
  ) +
  guides(color = guide_legend("Difficulty"))
```

```

# Save plot
ggsave("nn_SMOTE_imb_plt.png")
```

A.2 Varying Imbalance levels

In order to test how the synthetic data from our `imbalanced_data_generator()` function performs at all imbalance levels, we must create a function first to calculate how many positives are needed to achieve each imbalance level from 0.1 to 1.

A.2.1 Single-Attempt Classification

To generate Figure 3.6, we first outline the mathematical process to create this function, called `calc_pos_neg()`, and then show both this function and the `imb_sim()` function that tests all the desired imbalance levels.

Mathematical Process

Using a fixed T, total amount of observations in our generated dataset, we can calculate m_n and m_p, the number of negative and positive observations needed in the dataset, to achieve a desired imbalance level ρ.

Note that by definition, $\rho = \frac{m_p}{m_n}$ and $T = m_n + m_p$. We then have:

$$\begin{aligned}
&\frac{m_p}{m_n} = \rho \\
\iff &\frac{T - m_n}{m_n} = \rho \\
\iff &T - m_n = m_n \cdot \rho \\
\iff &T = m_n \cdot \rho + m_n \\
\iff &T = m_n(1 + \rho) \\
\iff &m_n = \frac{T}{1 + \rho}
\end{aligned}$$

Using this value of m_n and our fixed value of T, we can calculate m_p as $m_p = T - m_n$.

Function to Test Imbalance

Below, we show the code used to generate Figure 3.6 that tests classifier performance over all imbalance levels on the generated datasets.

```
# Import packages
import random
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

```python
import seaborn as sns

from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import balanced_accuracy_score
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import RandomForestClassifier

# Define imbalance levels of interest
imb_lvls = np.linspace(0,1,101) # Vector of imbalance levels 0.01 - 1

# Determine how many positive and negatives to achieve desired imbalance
def calc_pos_neg(imb_lvls):
    """Goal: Calculate n_pos and n_neg for data generation function
    based on imbalance levels

    Input:
        imb_lvls (vector): imbalance levels to test
    """

    # Initialize dataset size and arrays
    total_obs = 2100
    n_pos, n_neg = [], []

    # Append values for imbalance levels
    for i in range(0, len(imb_lvls)):
        denominator = 1 + imb_lvls[i]
        n_neg.append(int(round((total_obs / denominator))))
        n_pos.append(total_obs - n_neg[i])

    # Concatenate arrays into dataset
    data_class = {"n_pos": n_pos, "n_neg": n_neg}
    data_class = pd.DataFrame(data_class)
    data_class = data_class.iloc[1:]

    # Return dataset
    return data_class

# Get n_pos, n_neg based on desired imbalance levels
pos_neg = calc_pos_neg(imb_lvls)

# Function to test imbalance at all levels
def imb_sim(pos_neg, sd, mean):

    """Imbalance testing Function

    Inputs:
        pos_neg (dataframe): Dataset with num. pos./neg.
        sd (int): Standard deviation of normal distribution
        mean (int): Mean of normal distribution
    """

    # Get number of positive and negative based on calculation
    n_pos = pos_neg.loc[:,'n_pos'].values
    n_neg = pos_neg.loc[:,'n_neg'].values

    # Initialize array for results of classification
    rf_res = []
    nb_res = []
    gb_res = []

    for i in range(0, len(n_pos)):

        # Generate imbalanced dataset with parameters
        imb_df = imbalanced_data_generator(n_pos = n_pos[i],
                                           n_neg = n_neg[i],
                                           n_features = 2,
                                           n_inf_features = 2,
                                           sd = sd,
                                           class1_mean = mean)
        X = imb_df.drop('target', axis=1)
        y = imb_df[['target']]

        # Create train and test split (70:30 train:test)
        X_train, X_test, y_train, y_test = train_test_split(X, y,
                                           test_size = 0.30, \
                                           random_state = 20)

        # Classify Random Forest
        rf = RandomForestClassifier(random_state = 20)
        rf_pred = rf.fit(X_train, y_train).predict(X_test)
        rf_res.append(balanced_accuracy_score(y_test, rf_pred))

        # Classify Gaussian Naive Bayes
        nb = GaussianNB()
        nb_pred = nb.fit(X_train, y_train).predict(X_test)
        nb_res.append(balanced_accuracy_score(y_test, nb_pred))

        # Classify Gradient Boost
        gb = GradientBoostingClassifier(random_state = 20)
        gb_pred = gb.fit(X_train, y_train).predict(X_test)
        gb_res.append(balanced_accuracy_score(y_test, gb_pred))


    # Dataframe of results
    results = {"imb_lvl": imb_lvls[1:], "gb": gb_res, "nb": nb_res, "rf"
   : rf_res}
    results = pd.DataFrame(results)

    # Return results
    return(results)

# Run function to get results
SD = 1 # Choose sd
MEAN = 5 # Choose mean
easy1 = imb_sim(pos_neg = pos_neg, sd = SD, mean = MEAN)
easy1 = pd.melt(easy1, id_vars = 'imb_lvl',
                value_vars = ['gb', 'nb', 'rf'],
```

```python
                value_name = 'bal_acc', var_name = ['classifier'])

# Plot
sns.set_style('whitegrid')
sns.set_palette(['tab:pink', 'tab:cyan', 'tab:olive'])
plt.figure(figsize = (8, 5))
sns.lineplot(data = easy1, x = 'imb_lvl', y = 'bal_acc',
             hue = 'classifier',
             legend = None);
plt.xlabel('Imbalance Level', size = 20)
plt.ylabel('Balanced Accuracy', size = 20)
plt.xticks(fontsize = 20)
plt.yticks(fontsize = 20)
plt.title('$X\sim N(0,1),\, Y\sim N(5,1)$', size = 20);
plt.legend(title = 'Classifier', fontsize='large',
           title_fontsize='14',
           labels=['Gradient Boosting', 'Gaussian Naive Bayes',
           'Random Forest']);
plt.savefig('imb_sim_easy')
```

A.2.2 Simulation Study

To generate Figure 3.7, we perform the process shown in Section A.2.1 N times, where $N = 100$ in our experiment. The code is similar to that shown in Section A.2.1, but we have to add an extra loop to iterate from 1 to N.

The `imbalanced_data_generator()` function is defined first, which is discussed in detail in Chapter 3. We then define the `calc_pos_neg()` function which uses the mathematical process outlined in Section A.2.1 to determine the number of positive and negative instances needed in our dataset of a fixed size to acheive each imbalance level from 0.1-1.

Finally, the `imb_sim()` function iterates through all the imbalance levels and calculates the balanced accuracy of each classifier prediction N times. Note that we can change N by changing the parameter value set in line 94 below.

The results of the experiment are shown on an easy case below because we set `sd=1` and `mean=5` in line 151, meaning we are considering the classification of $X \sim N(0, 1)$ and $Y \sim N(5, 1)$.

We then plot our simulation results using the Seaborn and Matplotlib modules in lines 167-185.

```python
1  # Import packages
2  import random
3  import numpy as np
4  import pandas as pd
5  import matplotlib.pyplot as plt
6  import seaborn as sns
7
8  from sklearn.ensemble import GradientBoostingClassifier
9  from sklearn.ensemble import RandomForestClassifier
10 from sklearn.naive_bayes import GaussianNB
```

```python
from sklearn.metrics import balanced_accuracy_score
from sklearn.model_selection import train_test_split

# Function to generate data
def imbalanced_data_generator(n_pos,
                              n_neg,
                              n_features,
                              n_inf_features,
                              sd,
                              class1_mean):
    """
    Imbalanced Data Generator
    """
    class0 = np.array(np.random.normal(loc = 0, scale = 1, size = (n_neg
   *n_features)))
    class0 = np.reshape(class0, (n_neg, n_features))
    class0 = pd.DataFrame(class0)

    class1 = np.array(np.random.normal(loc = class1_mean, scale = sd,
   size = (n_pos*n_inf_features)))
    class1 = np.reshape(class1, (n_pos, n_inf_features))
    class1_cols = np.r_[(n_features-n_inf_features):(n_features+
   n_inf_features-2)]
    class1 = pd.DataFrame(class1, columns = class1_cols)

    classr1 = np.array(np.random.normal(loc = 0, scale = 1, size = (
   n_pos*(n_features-n_inf_features))))
    classr1 = np.reshape(classr1, (n_pos, (n_features-n_inf_features)))
    classr1 = pd.DataFrame(classr1)

    c1_full_names = np.r_[0:(n_features)]
    class1_full = pd.concat([class1, classr1], axis=1)
    class1_full = pd.DataFrame(class1_full, columns = c1_full_names)
    data = pd.concat([class0, class1_full], axis = 0)

    labels = ["0", "1"]
    full_labels = np.repeat(labels, [n_neg, n_pos])
    full_labels = pd.DataFrame(full_labels)
    full_labels = full_labels.rename(columns = {0:'target'})

    full_data = np.append(data, full_labels.to_numpy(), axis = 1)
    full_data = pd.DataFrame(full_data)
    full_data.columns = [*full_data.columns[:-1], 'target']

    return(full_data)

# Define vector of imbalance levels
imb_lvls = np.linspace(0,1,101) # Vector of imbalance levels 0.01 - 1

# Determine how many positive and negatives to achieve desired imbalance
def calc_pos_neg(imb_lvls):
    """Goal: Calculate n_pos and n_neg for data generation function
    based on imbalance levels

    Input:
        imb_lvls (vector): imbalance levels to test
    """

    # Initialize dataset size and arrays
    total_obs = 2100
    n_pos, n_neg = [], []

    # Append values for imbalance levels
    for i in range(0, len(imb_lvls)):
        denominator = 1 + imb_lvls[i]
        n_neg.append(int(round((total_obs / denominator))))
        n_pos.append(total_obs - n_neg[i])

    # Concatenate arrays into dataset
    data_class = {"n_pos": n_pos, "n_neg": n_neg}
    data_class = pd.DataFrame(data_class)
    data_class = data_class.iloc[1:]

    # Return dataset
    return data_class

# Get data based on desired imbalance levels
pos_neg = calc_pos_neg(imb_lvls)

# Imbalance simulation function
def imb_sim(pos_neg, sd, mean):

    # Get number of positive and negative based on calculation
    n_pos = pos_neg.loc[:,'n_pos'].values
    n_neg = pos_neg.loc[:,'n_neg'].values

    # Define number of iterations
    N = 100

    # Initialize arrays for results
    rf_res = np.zeros(shape = (len(n_pos), N + 2))
    nb_res = np.zeros(shape = (len(n_pos), N + 2))
    gb_res = np.zeros(shape = (len(n_pos), N + 2))

    for i in range(0, len(n_pos)):

        # Generate N imbalanced datasets with parameters
        for j in range(0, N):

            # Generate dataset with certain imbalance level
            imb_df = imbalanced_data_generator(n_pos = n_pos[i],
                                               n_neg = n_neg[i],
                                               n_features = 2,
                                               n_inf_features = 2,
                                               sd = sd,
                                               class1_mean = mean)
            X = imb_df.drop('target', axis=1)
            y = imb_df[['target']]

            # Create train and test split (70:30 train:test)
            X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                   test_size = 0.30)

            # Classify Random Forest
            rf = RandomForestClassifier()
            rf_pred = rf.fit(X_train, y_train).predict(X_test)
            rf_res[i][j] = balanced_accuracy_score(y_test, rf_pred)
            rf_res[i][10] = imb_lvls[i]
            rf_res[i][11] = 1 # Indicates RF

            # Classify Gaussian Naive Bayes
            nb = GaussianNB()
            nb_pred = nb.fit(X_train, y_train).predict(X_test)
            nb_res[i][j] = balanced_accuracy_score(y_test, nb_pred)
            nb_res[i][10] = imb_lvls[i]
            nb_res[i][11] = 2 # Indicates NB

            # Classify Gradient Boost
            gb = GradientBoostingClassifier()
            gb_pred = gb.fit(X_train, y_train).predict(X_test)
            gb_res[i][j] = balanced_accuracy_score(y_test, gb_pred)
            gb_res[i][10] = imb_lvls[i]
            gb_res[i][11] = 3 # Indicates GB

    # Make each result a dataframe and append them
    rf_res = pd.DataFrame(rf_res)
    nb_res = pd.DataFrame(nb_res)
    gb_res = pd.DataFrame(gb_res)

    # Create full dataset and return it
    full_res = rf_res.append([nb_res, gb_res])
    return(full_res)

# Run simulation and reformat data
easy1 = imb_sim(pos_neg = pos_neg, sd = 1, mean = 5)
easy1.rename(columns = {easy1.columns[10]: "imb_lvl" },
             inplace = True)
easy1.rename(columns = {easy1.columns[11]: "classifier" },
             inplace = True)
easy1['classifier'] = easy1['classifier'].replace([1.0], 'RF')
easy1['classifier'] = easy1['classifier'].replace([2.0], 'NB')
easy1['classifier'] = easy1['classifier'].replace([3.0], 'GB')
easy1 = pd.melt(easy1,
                id_vars = ['imb_lvl', 'classifier'],
                value_vars = [i for i in range(0, N)],
                value_name = 'bal_acc',
                var_name = ['sim_run'])


# Plot
sns.set_style('whitegrid')
sns.set_palette(['tab:pink', 'tab:cyan', 'tab:olive'])
```

```python
plt.figure(figsize=(8, 5))
sns.lineplot(data = easy1,
             x = 'imb_lvl',
             y = 'bal_acc',
             hue = 'classifier',
             legend = None);
plt.xlabel('Imbalance Level', size = 20)
plt.ylabel('Balanced Accuracy', size = 20)
plt.xticks(fontsize = 20)
plt.yticks(fontsize = 20)
plt.title('$X\sim N(0,1),\, Y\sim N(5,1)$', size = 20);
plt.legend(title = 'Classifier',
           fontsize='large',
           title_fontsize='14',
           labels=['Gradient Boosting',
                   'Gaussian Naive Bayes',
                   'Random Forest']);
plt.savefig('real_sim_figure')
```

Appendix B
Intel Images

The code below shows the Intel Images dataset code, including a PCA used to determine classification difficulty and neural network experiment structure and results analysis.

B.1 Determine Difficulty of Classification Tasks

As discussed in Chapter 4, we use PCA to determine the difficulty of each binary classification task in the Intel Images dataset. After defining the pca in line 86 of the code chunk below, we separate the components using the `pca.transform()` function from the Scikit-learn module. We define individual components, one in each of the three dimensions (specified in the argument of `decomposition.PCA()`) in lines 92-94. We can then plot one component against another and repeat this process for each combination of components.

```
1  # Import packages
2  import numpy as np
3  import os
4  from sklearn.metrics import confusion_matrix
5  from sklearn import decomposition
6  from sklearn.utils import shuffle
7  import matplotlib.pyplot as plt
8  import cv2
9  import tensorflow as tf
10 from tqdm import tqdm
11 
12 from keras.applications.vgg16 import VGG16
13 from keras.preprocessing import image
14 from keras.applications.vgg16 import preprocess_input
15 
16 # We need all the classes to conduct a PCA
17 class_names = ['mountain', 'street', 'glacier',
18                'buildings', 'sea', 'forest']
19 class_names_label = {class_name:i for i, class_name in enumerate(
       class_names)}
20 IMAGE_SIZE = (150, 150) # Set image size
21 
22 # Load images
23 def load_data():
24     """
25         Load the data
26     """
27 
```

```python
    datasets = ['seg_train', 'seg_test']
    output = []

    # Iterate through training and test sets
    for dataset in datasets:

        images = []
        labels = []

        print("Loading {}".format(dataset))

        # Iterate through each folder corresponding to a category
        for folder in os.listdir(dataset):
            label = class_names_label[folder]

            # Iterate through each image in our folder
            for file in tqdm(os.listdir(os.path.join(dataset, folder))):

                # Get the path name of the image
                img_path = os.path.join(os.path.join(dataset, folder),
                                                     file)

                # Open and resize the img
                image = cv2.imread(img_path)
                image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
                image = cv2.resize(image, IMAGE_SIZE)

                # Append the image and its corresponding label to the output
                images.append(image)
                labels.append(label)

        images = np.array(images, dtype = 'float32')
        labels = np.array(labels, dtype = 'int32')

        output.append((images, labels))

    return output

# Load and shuffle data
(train_images, train_labels), (test_images, test_labels) = load_data()
train_images, train_labels = shuffle(train_images, train_labels,
                                     random_state = 25)

# Get number of training
n_train = train_labels.shape[0]
n_test = test_labels.shape[0]
train_images = train_images / 255.0
test_images = test_images / 255.0

# Get features for PCA
model = VGG16(weights = 'imagenet', include_top = False)
train_features = model.predict(train_images)
test_features = model.predict(test_images)
```

```
n_train , x, y, z = train_features.shape # Train dimensions
n_test , x, y, z = test_features.shape # Test dimensions
numFeatures = x * y * z # Total number of features

# PCA
pca = decomposition.PCA(n_components = 3)
X = train_features.reshape((n_train , x * y * z))
pca.fit(X)
C = pca.transform(X)

# Individual PCA components (three dimensions)
C1 = C[:,0]
C2 = C[:,1]
C3 = C[:,2]

# Show C1 vs. C2
plt.subplots(figsize = (10, 10))
for i, class_name in enumerate(class_names):
    plt.scatter(C1[train_labels == i][:1000],
                C2[train_labels == i][:1000],
                label = class_name,
                alpha = 0.4)
plt.legend()
plt.title("C1 vs. C2 PCA Projection")
plt.show()

# Show C2 vs. C3
plt.subplots(figsize=(10,10))
for i, class_name in enumerate(class_names):
    plt.scatter(C2[train_labels == i][:1000],
                C3[train_labels == i][:1000],
                label = class_name,
                alpha = 0.4)
plt.legend()
plt.title("C2 vs. C3 PCA Projection")
plt.show()

# Show C1 vs. C3
plt.subplots(figsize=(10,10))
for i, class_name in enumerate(class_names):
    plt.scatter(C1[train_labels == i][:1000],
                C3[train_labels == i][:1000],
                label = class_name,
                alpha = 0.4)
legend = plt.legend()
plt.title("C1 vs. C3 PCA Projection")
plt.show()
```

B.2 Neural Networks

After determining the difficulty of each classification task using PCA, we classified each set of labels in the Intel Images dataset using a CNN, shown below. We define our model using Keras with specified layers and parameters in line 112. After compiling this model using the `model.compile()` function, we then fit our model using the `model.fit()` function. To test the model, we compare the true test labels to the predicted labels with the `model.evaluate()` function.

Note that when we define `history` in line 127, we can plot this to check the convergence of the model's loss and accuracy. Figure B.1 shows an example of convergence of the model's accuracy and loss during training for the Mountain vs. Street classification task. As expected, both model accuracy and the accuracy of the validation set converge to 1. Similarly, the loss of both the model and the validation set converge to 0 over the 20 epochs shown.

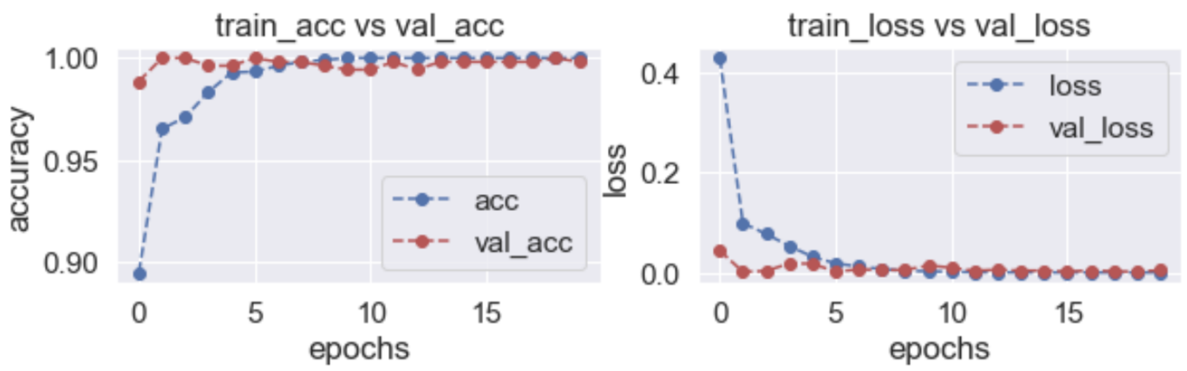


Figure B.1: Example of the convergence of the model's accuracy (left) and loss (right) during training.

```python
1  import numpy as np
2  import os
3  from sklearn.metrics import confusion_matrix
4  from sklearn.utils import shuffle
5  import matplotlib.pyplot as plt
6  import cv2
7  import tensorflow as tf
8  from tqdm import tqdm
9  import random
10 import sys
11 import sklearn
12 
13 # Choose labels and set image size
14 class_names = ['mountain', 'street']
15 class_names_label = {class_name:i for i, class_name in enumerate(
       class_names)}
16 IMAGE_SIZE = (150, 150)
17 
18 # Load data function
19 def load_data():
20     """
21         Data loading function
22     """
23 
24     datasets = ['seg_train', 'seg_test']
```

```
    output = []

    # Iterate through training and test sets
    for dataset in datasets:

        images = []
        labels = []

        print("Loading {}".format(dataset))

        # Iterate through each folder corresponding to a category
        for folder in os.listdir(dataset):

            # Check if folder is one of the ones we care about
            if folder not in class_names:
                continue

            label = class_names_label[folder]

            # Iterate through each image in our folder
            for file in tqdm(os.listdir(os.path.join(dataset, folder))):

                # Get the path name of the image
                img_path = os.path.join(os.path.join(dataset, folder),
                                        file)

                # Open and resize the img
                image = cv2.imread(img_path)
                image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
                image = cv2.resize(image, IMAGE_SIZE)

                # Append the image and label to the output
                images.append(image)
                labels.append(label)

        images = np.array(images, dtype = 'float32')
        labels = np.array(labels, dtype = 'int32')

        output.append((images, labels))

    return output

# Define training and testing datasets
(train_images, train_labels), (test_images, test_labels) = load_data()
train_images, train_labels = shuffle(train_images, train_labels,
                                     random_state = 25)


### Remove instances from positive class to create imbalance
pos_indices = [i for i, x in enumerate(train_labels) if x]
neg_indices = [i for i in range(len(train_labels)) if i not in
               pos_indices]
train_images_neg = train_images[neg_indices]
train_images_pos = train_images[pos_indices]
```

```


# Number of observations to remove (to create imbalance)
prop_keep = .10 # Prop. to keep in positive class
obs_remove = (1 - prop_keep) * (len(pos_indices))
obs_remove = int(obs_remove)
obs_keep = len(pos_indices) - obs_remove

# Take obs_remove examples from positive class and remove
indices_to_keep = np.arange(obs_keep)
train_images_pos = train_images_pos[indices_to_keep]
train_images = np.concatenate((train_images_pos, train_images_neg))
pos = [1] * len(train_images_pos)
neg = [0] * len(train_images_neg)
train_labels = np.concatenate((pos, neg))

### Visualize
n_train = train_labels.shape[0]
n_test = test_labels.shape[0]

_, train_counts = np.unique(train_labels, return_counts = True)
_, test_counts = np.unique(test_labels, return_counts = True)
pd.DataFrame({'train': train_counts,
              'test': test_counts},
              index=class_names
            ).plot.bar()
plt.show()

# Standardize images
train_images = train_images / 255.0
test_images = test_images / 255.0

# Create model
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(32, (3, 3), activation = 'relu',
                           input_shape = (150, 150, 3)),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Conv2D(32, (3, 3), activation = 'relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation = tf.nn.relu),
    tf.keras.layers.Dense(6, activation = tf.nn.softmax)
])
model.compile(optimizer = 'adam',
              loss = 'sparse_categorical_crossentropy',
              metrics = ['accuracy'])

# Fit model and train
history = model.fit(train_images, train_labels,
                    batch_size = 128,
                    epochs = 20,
                    validation_split = 0.2)

# Test model
```

```
test_loss = model.evaluate(test_images, test_labels)
```

To compare the performance of the CNNs on both the easy and hard classification tasks from the Intel Images dataset, we aggregate the balanced accuracy results in a dataset and plot the effects of changing the imbalance levels on each classification tasks. The code to generate Figure 4.3, which displays this comparison, is given below.

```
# Install packages
library(tidyverse)

# Load dataand clean
data <- read_csv("ii_imb.csv") %>%
  filter(difficulty != "medium")

# Separate data by imbalance levels
imb.05 <- data %>%
  filter(imb_lvl == 0.05) %>%
  mutate(id = row_number())
imb.1 <- data %>%
  filter(imb_lvl == 0.1) %>%
  mutate(id = row_number())
imb.2 <- data %>%
  filter(imb_lvl == 0.2) %>%
  mutate(id = row_number())

# Re-combine datasets
all_imb <- rbind(imb.05, imb.1, imb.2)

# Add experiment names
df <- all_imb %>%
  mutate(
    majority_class = toupper(stringr::str_extract(
      majority_class,
      "^.{2}"
    )),
    minority_class = toupper(stringr::str_extract(
      minority_class,
      "^.{2}"
    ))
  ) %>%
  mutate(
    exp_setting = paste0(majority_class, "+", minority_class),
    difficulty = ifelse(exp_setting == "ST+BU", "hard", difficulty)
  )

# Prepare data for plot
df_plot <- df %>%
  mutate(
    imb_lvl = as.factor(imb_lvl),
    difficulty = as.factor(difficulty)
  )
df_plot$exp_setting <- factor(df_plot$exp_setting, levels = c(
  "MO+ST", "MO+BU",
  "ST+GL", "ST+SE",
```

```r
  "GL+BU", "MO+GL",
  "MO+SE", "GL+SE",
  "ST+BU"
))
x
# Plot
ii_imb <- ggplot(df_plot, aes(
  x = exp_setting, y = bal_acc,
  color = difficulty,
  shape = imb_lvl
)) +
  geom_point(size = 4) +
  theme_classic() +
  labs(
    x = "Experimental Setting",
    y = "Balanced Accuracy",
    color = "Difficulty",
    shape = "Imbalance Level",
    title = "Intel Images Performance at Different Imbalance Levels"
  )

# Save plot
ggsave("ii_imb.png")

# Calculate mean performance for table
mean_df <- df %>%
  group_by(exp_setting) %>%
  summarise(mean_ba = mean(bal_acc))
```

Appendix C
Kaggle Credit Card Fraud

C.1 Comparison of Accuracy and Balanced Accuracy

This code compares the performance of Gradient Boosting, Gaussian Naive Bayes, and Random Forest on the Kaggle Credit Card Fraud dataset with accuracy and balanced accuracy. This code is used to generate the values in Table 3.3.

After reading in the data and defining the train and test sets with a 70:30 train-test split, we perform classification using each classifier and print the balanced accuracy and accuracy for each classification task.

Note that we do not make any modifications to the imbalance level of the Kaggle Credit Cards dataset for this analysis. The original dataset has an imbalance level of approximately $\rho = 0.002$, which is lower than any imbalance level we test in later experiments.

```python
# Import pacakges
import pandas as pd
import imblearn
import numpy as np
import seaborn as sns

from sklearn.model_selection import train_test_split
from sklearn import metrics
from sklearn.metrics import balanced_accuracy_score, accuracy_score
from sklearn.ensemble import GradientBoostingClassifier,
    RandomForestClassifier
from sklearn.naive_bayes import GaussianNB

# Load data
df = pd.read_csv('~/Desktop/thesis.nosync/large-files.nosync/credit_card
    /creditcard.csv')

# Create train and test set (70:30 train:test)
X = df.drop('Class', axis=1)
y = df[['Class']]
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                   test_size=0.30, random_state=42)

# Gradient Boosting
gb = GradientBoostingClassifier(random_state = 20)
gb_pred = gb.fit(X_train, y_train).predict(X_test)
print('bal_acc', balanced_accuracy_score(y_test, gb_pred))
```

```python
26 print('acc', accuracy_score(y_test, gb_pred))
27
28 # Gaussian Naive Bayes
29 gnb = GaussianNB()
30 gnb_pred = gnb.fit(X_train, y_train).predict(X_test)
31 print('bal_acc', balanced_accuracy_score(y_test, gnb_pred))
32 print('acc', accuracy_score(y_test, gnb_pred))
33
34 # Random Forest
35 rf = RandomForestClassifier(random_state = 20)
36 rf_pred = rf.fit(X_train, y_train).predict(X_test)
37 print('bal_acc', balanced_accuracy_score(y_test, rf_pred))
38 print('acc', accuracy_score(y_test, rf_pred))
```

C.2 Classification at Selected Values of ρ

We can conduct the same classification shown in Section C.1 but change the imbalance level in the Kaggle Credit Cards dataset to test the effects of different values of ρ on classifier performance with and without resampling.

To do so, we first define our desired value of ρ. Recall that ρ is defined as $\rho = \frac{m_p}{m_n}$, so we can first check the number of positive instances in the dataset and then determine the amount of negative instances we want to keep in the dataset because, rearranging the equation for ρ, we have that $m_n' = \frac{m_p}{\rho}$. Then, we remove $m_n - m_n'$ from the negative observations in the dataset to achieve the desired ρ. This process is shown in code in lines 22-34.

After completing this process, we separate the data into train and test sets using a 70:30 train-test split, as before. We also define resampled dataset with SMOTE, shown in lines 43-44.

Finally, we classify the data with each of our classifiers both with and without resampling.

```python
1 # Import packages
2 import numpy as np
3 import pandas as pd
4 import os
5
6 import sklearn
7 from sklearn.model_selection import train_test_split
8 from sklearn import metrics
9 from sklearn.metrics import balanced_accuracy_score,
    classification_report
10 from sklearn.ensemble import GradientBoostingClassifier,
    RandomForestClassifier
11 from sklearn.naive_bayes import GaussianNB
12
13 import imblearn
14 from imblearn.over_sampling import SMOTE, RandomOverSampler, ADASYN
15 from imblearn.under_sampling import NearMiss
16 from imblearn.combine import SMOTEENN, SMOTETomek
17
```

```
# Load Data
df = pd.read_csv('~/Desktop/thesis.nosync/large-files.nosync/credit_card
    /creditcard.csv')

##### Create imbalance #####
RHO = 0.05 # Set rho to some value
m_n_init = len(df[df['Class'] == 0].index) # Initial num. neg.
m_p = len(df[df['Class'] == 1].index) # Num. pos.
m_n_want = round(m_p / RHO) # Num. neg. for imbalance level
to_remove = round(m_n_init - m_n_want) # Num neg. instances to remove

# Separate negative and positive data
neg_data = df[df['Class'] == 0]
pos_data = df[df['Class'] == 1]
# Remove to_remove obs from neg_data
neg_data = neg_data.iloc[to_remove:]
# Concatenate datasets into imbalanced dataset
new_df = pos_data.append(pd.DataFrame(data = neg_data))

##### Separate data for classification #####
X = new_df.drop('Class', axis=1)
y = new_df[['Class']]
# Create train and test set (70:30 train:test)
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                    test_size=0.30, random_state=42)
# SMOTE datasets
smote = SMOTE(random_state = 20)
X_train_smote, y_train_smote = smote.fit_resample(X_train,y_train)

##### Classification #####

# Gradient Boost -  No Resampling
gb = GradientBoostingClassifier(random_state = 20)
gb_pred = gb.fit(X_train, y_train).predict(X_test)
print(classification_report(y_test, gb_pred))
print(balanced_accuracy_score(y_test, gb_pred))

# Gradient Boost - SMOTE Resampling
gb = GradientBoostingClassifier(random_state = 20)
gb_pred_smote = gb.fit(X_train_smote, y_train_smote).predict(X_test)
print(classification_report(y_test, gb_pred_smote))
print(balanced_accuracy_score(y_test, gb_pred_smote))

# Random Forest - No Resampling
rf = RandomForestClassifier(random_state = 20)
rf_pred = rf.fit(X_train, y_train).predict(X_test)
print(classification_report(y_test, rf_pred))
print(balanced_accuracy_score(y_test, rf_pred))

# Random Forest - SMOTE Resampling
rf = RandomForestClassifier(random_state = 20)
rf_pred_smote = rf.fit(X_train_smote, y_train_smote).predict(X_test)
print(classification_report(y_test, rf_pred_smote))
print(balanced_accuracy_score(y_test, rf_pred_smote))
```

```

# Gaussian NB - No Resampling
gnb = GaussianNB()
gnb_pred = gnb.fit(X_train, y_train).predict(X_test)
print(classification_report(y_test, gnb_pred))
print(balanced_accuracy_score(y_test, gnb_pred))

# Gaussian NB - SMOTE Resampling
gnb = GaussianNB()
gnb_pred_smote = gnb.fit(X_train_smote, y_train_smote).predict(X_test)
print(classification_report(y_test, gnb_pred_smote))
print(balanced_accuracy_score(y_test, gnb_pred_smote))
```

To visualize the results from the above experiments, we place our results in a csv file and import it into R for analysis. After cleaning the data, we can simply plot the balanced accuracy against each imbalance level we analyzed ($\rho = 0.05$, $\rho = 0.1$, $\rho = 0.2$). We display classifiers by both size and color because of overlapping points. After faceting for resampling and making minor aesthetic edits to the plot, we generate Figure 4.4.

```r
# Load packages
library(tidyverse)
library(latex2exp)

# Load data and clean
data <- read_csv('kaggle_resampling_imb.csv') %>%
  rename(Classifier = classifier) %>%
  mutate(imbalance_lvl = as.factor(imbalance_lvl))

# Create plot
ggplot(data = data, aes(
  x = imbalance_lvl, y = balanced_accuracy,
  color = Classifier, size = Classifier
)) +
  geom_point() +
  facet_wrap(~resampling) +
  labs(
    title = "Balanced Accuracy Classifier Performance",
    subtitle = "Kaggle Credit Cards Dataset",
    x = TeX("Imbalance Level ($\\rho$)"),
    y = "Balanced Accuracy"#,
    #color = "Classifier"
  ) +
  theme_classic() +
  theme(
    plot.title = element_text(hjust = 0.5),
    plot.subtitle = element_text(hjust = 0.5)
  ) +
  scale_color_manual(
    labels = c("Gradient Boosting", "Gaussian Naive Bayes", "Random
   Forest"),
    values = c("#BDCD25", "#de772d", "#e815dd")
  ) +
  scale_size_manual(
    labels = c("Gradient Boosting", "Gaussian Naive Bayes", "Random
   Forest"),
    values = c(5, 3, 1)
  )

# Save plot
ggsave('discrete_clf_kaggle.png')
```

C.3 Performance over All Imbalance Levels

In both of the single-attempt classification (Section C.3.1) and simulation study (Section C.3.2), we must remove negative instances from the Kaggle Credit Card Fraud dataset to achieve our desired level of ρ.

Recall that from Section C.2, this process involves several steps. We first define our desired value of ρ. Because ρ is defined as $\rho = \frac{m_p}{m_n}$, so we can first check the number of positive instances in the dataset and then determine the amount of negative instances we want to keep in the dataset because, rearranging the equation for ρ, we have that $m_n' = \frac{m_p}{\rho}$. Then, we remove $m_n - m_n'$ from the negative observations in the dataset to achieve the specified ρ.

C.3.1 Single-Attempt Classification

In single-attempt classification, like in Section A.2.1, we create a function to use classifiers to classify the Kaggle Credit Card Fraud dataset at each imbalance level from 0.1-1.

```
# Import packages
import random
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from matplotlib.ticker import StrMethodFormatter

from sklearn.metrics import balanced_accuracy_score
from sklearn.model_selection import train_test_split
from sklearn.ensemble import GradientBoostingClassifier,
   RandomForestClassifier
from sklearn.naive_bayes import GaussianNB

# Load data
df = pd.read_csv('~/Desktop/thesis.nosync/large-files.nosync/credit_card
   /creditcard.csv')

# Create vector of imbalance levels of interest
imb_lvls = np.linspace(0,1,101)
imb_lvls = imb_lvls[1:]

def imb_sim(df):

    """Goal: Get balanced accuracy for all imbalance levels from 0.1-1
    for RF, NB, GB

    Input: Dataset to classify
    """

    # Initialize array for results of classification
    rf_res = []
    nb_res = []
    gb_res = []

    for i in range(0, len(imb_lvls)-1):

        # Generate imbalanced dataset with parameters
        RHO = imb_lvls[i] # Set rho to some value
        m_n_init = len(df[df['Class'] == 0].index) # Initial num. neg.
        m_p = len(df[df['Class'] == 1].index) # Num. pos
        m_n_want = round(m_p / RHO) # Num. neg. to achieve rho
        to_remove = int(round(m_n_init - m_n_want)) # Neg. to remove

        # Remove observations, create new dataset
        neg_data = df[df['Class'] == 0]
        pos_data = df[df['Class'] == 1]
        neg_data = neg_data.iloc[to_remove:]
        new_df = pos_data.append(pd.DataFrame(data = neg_data))

        # Create train and test set (70:30 train:test)
        X = new_df.drop('Class', axis=1)
        y = new_df[['Class']]
        X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                test_size = 0.30,
   random_state = 42)

        # Classify Random Forest
        rf = RandomForestClassifier(random_state = 20)
        rf_pred = rf.fit(X_train, y_train).predict(X_test)
        rf_res.append(balanced_accuracy_score(y_test, rf_pred))

        # Classify Gaussian Naive Bayes
        nb = GaussianNB()
        nb_pred = nb.fit(X_train, y_train).predict(X_test)
        nb_res.append(balanced_accuracy_score(y_test, nb_pred))

        # Classify Gradient Boost
        gb = GradientBoostingClassifier(random_state = 20)
        gb_pred = gb.fit(X_train, y_train).predict(X_test)
        gb_res.append(balanced_accuracy_score(y_test, gb_pred))

    results = {"imb_lvl": imb_lvls[1:], "gb": gb_res, "nb": nb_res, "rf"
   : rf_res}
    results = pd.DataFrame(results)

    return(results)

# Run simulation on data
```

```python
res = imb_sim(df)

# Reformat data
data = pd.melt(res, id_vars = 'imb_lvl', value_vars = ['gb', 'nb', 'rf'
    ],
               value_name = 'bal_acc', var_name = ['classifier'])

# Plot
sns.set_style('whitegrid')
sns.set_palette(['#ED3550', 'tab:cyan', '#B17FE3'])
plt.figure(figsize = (8, 5))
sns.lineplot(data = data, x = 'imb_lvl', y = 'bal_acc',
             hue = 'classifier', legend = None);
plt.xlabel('Imbalance Level', size = 20)
plt.ylabel('Balanced Accuracy', size = 20)
plt.xticks(fontsize = 20)
plt.yticks(fontsize = 20)
plt.title('Kaggle Credit Cards Performance', size = 20);
plt.legend(title = 'Classifier', fontsize = 'large', title_fontsize = '
    14',
           labels = ['Gradient Boosting', 'Gaussian Naive Bayes',
                     'Random Forest']);
plt.savefig('imb_sim_kaggle')
```

C.3.2 Simulation Study

Similar to Section A.2.2, we create a function to use classifiers to classify the Kaggle Credit Card Fraud dataset N times at each imbalance level from 0.1-1. In our experiments, we choose to use $N = 100$, but this parameter can be changed. Increasing N will improve the robustness of the results, though it would also increase computational complexity, so this may be a good area to investigate in the future.

```python
# Import packages
import pandas as pd
import numpy as np
import random
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.ensemble import GradientBoostingClassifier,
    RandomForestClassifier
from sklearn.metrics import balanced_accuracy_score
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB

# Load dataset
df = pd.read_csv('~/Desktop/thesis.nosync/large-files.nosync/credit_card
    /creditcard.csv')

# Create vector of imbalance levels of interest
imb_lvls = np.linspace(0,1,101) # Vector of imbalance levels 0.01 - 1
imb_lvls = imb_lvls[1:]

```

```python
def imb_sim(df):

    """Goal: Get balanced accuracy for all imbalance levels from 0.1-1
    for RF, NB, GB over N classification attempts at each imbalance
   level

    Input: Dataset to classify
    """

    # Define number of iterations
    N = 100

    # Initialize arrays for results
    rf_res = np.zeros(shape=(len(imb_lvls), N+2))
    nb_res = np.zeros(shape=(len(imb_lvls), N+2))
    gb_res = np.zeros(shape=(len(imb_lvls), N+2))

    for i in range(0, len(imb_lvls)-1):

        # Generate N imbalanced datasets with parameters
        for j in range(0, N):

            # Generate imbalanced dataset with parameters
            RHO = imb_lvls[i] # Set rho to some value
            m_n_init = len(df[df['Class'] == 0].index) # Initial num.
   neg.
            m_p = len(df[df['Class'] == 1].index) # Num. pos
            m_n_want = round(m_p / RHO) # Num. neg. to achieve rho
            to_remove = int(round(m_n_init - m_n_want)) # Neg. to remove

            neg_data = df[df['Class'] == 0]
            pos_data = df[df['Class'] == 1]
            neg_data = neg_data.iloc[to_remove:]
            new_df = pos_data.append(pd.DataFrame(data = neg_data))

            X = new_df.drop('Class', axis = 1)
            y = new_df[['Class']]

            # Create train and test split (70:30 train:test)
            X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                test_size = 0.30,
                                                random_state = 42)

            # Classify Random Forest
            rf = RandomForestClassifier()
            rf_pred = rf.fit(X_train, y_train).predict(X_test)
            rf_res[i][j] = balanced_accuracy_score(y_test, rf_pred)
            rf_res[i][100] = imb_lvls[i]
            rf_res[i][101] = 1 # Indicates RF

            # Classify Gaussian Naive Bayes
            nb = GaussianNB()
            nb_pred = nb.fit(X_train, y_train).predict(X_test)
            nb_res[i][j] = balanced_accuracy_score(y_test, nb_pred)
            nb_res[i][100] = imb_lvls[i]
            nb_res[i][101] = 2 # Indicates NB

            # Classify Gradient Boost
            gb = GradientBoostingClassifier(random_state = 20)
            gb_pred = gb.fit(X_train, y_train).predict(X_test)
            gb_res[i][j] = balanced_accuracy_score(y_test, gb_pred)
            gb_res[i][100] = imb_lvls[i]
            gb_res[i][101] = 3 # Indicates GB

    # Make each result a dataframe
    rf_res = pd.DataFrame(rf_res)
    nb_res = pd.DataFrame(nb_res)
    gb_res = pd.DataFrame(gb_res)

    # Create full dataset and return
    full_res = rf_res.append([nb_res, gb_res])
    return(full_res)

# Run simulation and reformat data
data = imb_sim(df = df)
data.rename(columns={data.columns[100]: "imb_lvl" }, inplace = True)
data.rename(columns={data.columns[101]: "classifier" }, inplace = True)
data['classifier'] = data['classifier'].replace([1.0],'RF')
data['classifier'] = data['classifier'].replace([2.0],'NB')
data['classifier'] = data['classifier'].replace([3.0],'GB')
data = pd.melt(data, id_vars = ['imb_lvl', 'classifier'], v
                value_vars = [i for i in range(0,100)],
                value_name = 'bal_acc', var_name = ['sim_run'])

# Plot
sns.set_style('whitegrid')
sns.set_palette(['tab:pink', 'tab:cyan', 'tab:olive'])
plt.figure(figsize = (8, 5))
sns.lineplot(data = data, x = 'imb_lvl', y = 'bal_acc',
             hue = 'classifier', legend = None));
plt.xlabel('Imbalance Level', size = 20)
plt.ylabel('Balanced Accuracy', size = 20)
plt.xticks(fontsize = 20)
plt.yticks(fontsize = 20)
plt.title('Kaggle Credit Cards Performance Simulation', size = 20);
plt.legend(title = 'Classifier', fontsize = 'large',
           title_fontsize='14',
           labels = ['Gradient Boosting', 'Gaussian Naive Bayes',
                     'Random Forest']);
plt.ylim([0.95, 1.003])
plt.savefig('kaggle_sim')
```

Appendix D
New Techniques

The code below relates to Chapter 5, where we discuss state-of-the-art loss functions (Focal Loss, Gradient Harmonizing Mechanism Loss, and Label-Distribution-Aware Margin Loss) and test them on the Kaggle Credit Cards dataset. First, we show the code for each loss function in Section D.1, and we then give an example of a loss function experiment in Section D.2.

D.1 State of the Art Loss Functions

Each section below shows the code for a state-of-the-art loss function. Section D.1.1 displays Focal Loss, Section D.1.2 displays Gradient Harmonizing Mechanism (GHM) Loss, and Section D.1.3 displays Label-Distribution-Aware Margin (LDAM) Loss. Note that each of these functions were implemented with Pytorch, which we discuss in detail in Chapter 5.

To define these new loss functions, we need to define a new class for each. We can then call each loss function by classing its newly-defined class. Each class has an `__init__()` and a `forward()` function to initialize the function parameters and calculate loss, respectively.

D.1.1 Focal Loss

The `FocalLoss` class defines `gamma` and `alpha` in its `__init__()` function. Recall that γ adjusts the rate at which easily classified instances are downweighted and α acts as a weight to increase the importance of the minority class.

In the `forward()` function, we use the Functional module from Pytorch to calculate loss, which allows us to take logs and perform basic operations.

The code below for the Focal Loss function is also available on Github at `https://github.com/clcarwin/focal_loss_pytorch` (Lin et al., 2017).

```python
# Import packages
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable

# Define focal loss class
class FocalLoss(nn.Module):
    """ Focal Loss Function
    See "Focal Loss for Dense Object Detection"
    https://arxiv.org/pdf/1708.02002

    Inputs:
        gamma (float): tuneable focusing parameter
        alpha (float): class imbalance weight factor
        size_average (bool): return average or sum for loss
    """
    def __init__(self,
                 gamma = 0,
                 alpha = None,
                 size_average = True):
        super(FocalLoss, self).__init__()
        self.gamma = gamma
        self.alpha = alpha
        if isinstance(alpha,(float,int,long)): self.alpha = torch.Tensor
   ([alpha,1-alpha])
        if isinstance(alpha,list): self.alpha = torch.Tensor(alpha)
        self.size_average = size_average

    # Calculate loss
    def forward(self, input, target):
        if input.dim()>2:
            input = input.view(input.size(0),input.size(1),-1)
            input = input.transpose(1,2)
            input = input.contiguous().view(-1,input.size(2))
        target = target.view(-1,1)

        logpt = F.log_softmax(input)
        logpt = logpt.gather(1,target)
        logpt = logpt.view(-1)
        pt = Variable(logpt.data.exp())

        if self.alpha is not None:
            if self.alpha.type()!=input.data.type():
                self.alpha = self.alpha.type_as(input.data)
            at = self.alpha.gather(0,target.data.view(-1))
            logpt = logpt * Variable(at)

        loss = -1 * (1-pt)**self.gamma * logpt
        if self.size_average: return loss.mean()
        else: return loss.sum()
```

D.1.2 Gradient Harmonizing Mechanism (GHM) Loss

The GHM loss class, defined as `GHMC()`, implements the GHM loss function. We provide the GHM loss function below, but it is also available on Github at `https://github.com/libuyu/GHM_Detection` (B. Li, Y. Liu and X. Wang, 2019).

```
# Import packages
import torch
import torch.nn as nn
```

```python
import torch.nn.functional as F

# Resize class labels
def _expand_binary_labels(labels, label_weights, label_channels):
    bin_labels = labels.new_full((labels.size(0), label_channels), 0)
    inds = torch.nonzero(labels >= 1).squeeze()
    if inds.numel() > 0:
        bin_labels[inds, labels[inds] - 1] = 1
    bin_label_weights = label_weights.view(-1, 1).expand(
        label_weights.size(0), label_channels)
    return bin_labels, bin_label_weights

# Gradient Harmonizing Loss class
class GHMC(nn.Module):
    """ GHM Loss Function
    See "Gradient Harmonized Single-stage Detector
    https://arxiv.org/abs/1811.05181

    Inputs:
        bins (int): Number of the unit regions for distribution
  calculation.
        momentum (float): The parameter for moving average.
        use_sigmoid (bool): Can only be true for BCE based loss now.
        loss_weight (float): The weight of the total GHM-C loss.
    """
    def __init__(
            self,
            bins=10,
            momentum=0,
            use_sigmoid=True,
            loss_weight=1.0):
        super(GHMC, self).__init__()
        self.bins = bins
        self.momentum = momentum
        self.edges = torch.arange(bins + 1).float().cuda() / bins
        self.edges[-1] += 1e-6
        if momentum > 0:
            self.acc_sum = torch.zeros(bins).cuda()
        self.use_sigmoid = use_sigmoid
        if not self.use_sigmoid:
            raise NotImplementedError
        self.loss_weight = loss_weight

    def forward(self, pred, target, label_weight, *args, **kwargs):
        """Calculate the GHM-C loss.
        Args:
            pred (float tensor of size [batch_num, class_num]):
                The direct prediction of classification fc layer.
            target (float tensor of size [batch_num, class_num]):
                Binary class target for each sample.
            label_weight (float tensor of size [batch_num, class_num]):
                the value is 1 if the sample is valid and 0 if ignored.
        Returns:
            The gradient harmonized loss.
        """
        # Make target binary class label
        if pred.dim() != target.dim():
            target, label_weight = _expand_binary_labels(
                                    target, label_weight, pred.size(-1))
        target, label_weight = target.float(), label_weight.float()
        edges = self.edges
        mmt = self.momentum
        weights = torch.zeros_like(pred)

        # Gradient length
        g = torch.abs(pred.sigmoid().detach() - target)

        valid = label_weight > 0
        tot = max(valid.float().sum().item(), 1.0)
        n = 0  # n valid bins
        for i in range(self.bins):
            inds = (g >= edges[i]) & (g < edges[i+1]) & valid
            num_in_bin = inds.sum().item()
            if num_in_bin > 0:
                if mmt > 0:
                    self.acc_sum[i] = mmt * self.acc_sum[i] \
                        + (1 - mmt) * num_in_bin
                    weights[inds] = tot / self.acc_sum[i]
                else:
                    weights[inds] = tot / num_in_bin
                n += 1
        if n > 0:
            weights = weights / n

        loss = F.binary_cross_entropy_with_logits(
            pred, target, weights, reduction='sum') / tot
        return loss * self.loss_weight
```

D.1.3 Label-Distribution-Aware Margin (LDAM) Loss

The LDAM loss function, defined as the `LDAMLoss()` class, is shown below and can also be found on Github at `https://github.com/kaidic/LDAM-DRW` (Cao et al., 2019). LDAM returns a cross-entropy-based loss function, shown in lines 50-52 in the `forward()` function.

```
1  # Import packages
2  import math
3  import torch
4  import torch.nn as nn
5  import torch.nn.functional as F
6  import numpy as np
7  
8  # LDAM Loss class
9  class LDAMLoss(nn.Module):
10     """ LDAM Loss
11     See "Learning Imbalanced Datasets with Label-Distribution-Aware
12          Margin Loss"
13     https://arxiv.org/abs/1906.07413
```

```

    Inputs:
        cls_num_list (list): list of classes
        max_m (float): maximum value of n in denominator of delta
        weight (float): weight parameter for imbalance
        s (int): tuneable constant
    """
    def __init__(self,
                 cls_num_list,
                 max_m = 0.5,
                 weight = None,
                 s = 30):
        super(LDAMLoss, self).__init__()
        m_list = 1.0 / np.sqrt(np.sqrt(cls_num_list))
        m_list = m_list * (max_m / np.max(m_list))
        m_list = torch.cuda.FloatTensor(m_list)
        self.m_list = m_list
        assert s > 0
        self.s = s
        self.weight = weight

    def forward(self, x, target):
        """
        Calculate loss using LDAM
        """

        index = torch.zeros_like(x, dtype = torch.uint8)
        index.scatter_(1, target.data.view(-1, 1), 1)

        index_float = index.type(torch.cuda.FloatTensor)
        batch_m = torch.matmul(self.m_list[None, :],
                               index_float.transpose(0, 1))
        batch_m = batch_m.view((-1, 1))
        x_m = x - batch_m

        output = torch.where(index, x_m, x)
        return F.cross_entropy(self.s * output,
                               target,
                               weight = self.weight)
```

D.2 Loss Function Experiments

Below, we show a sample loss function experiment using a neural network to classify transactions in the Kaggle Credit Cards dataset. After loading all the required modules, we prepare the dataset by creating a 70:30 train-test split.

To use each loss function in the neural network, we must define them in a class. Note that each definition is taken from the respective code chunks given in Section D.1. Focal loss is defined in lines 48-90, GHM is defined in lines 93-146, and LDAM is defined in lines 148-180.

Then, we can define our neural network. When we define our hyperparameters, which are discussed in Chapter 5, we also define our loss function. Here, we just use BCE from Pytorch as an example, and we can call it as `nn.BCEWithLogitsLoss()`. To change the loss function used to run the neural network, we simply change the `LOSS_FUN` parameter in line 189 to the desired loss function. Each loss function class defined above is called with the class name followed by a set of parentheses. For example, to use Focal Loss, we set line 189 to the following: `LOSS_FUN = FocalLoss()`. This loss function is redefined as our criterion in line 259, and when we train our model, this criterion is used to compare the predicted label to the true label.

We then test the model, as shown in lines 297-305 and create a list of predicted labels. To evaluate the model, we use balanced accuracy as before.

```
1  # Import packages
2  import numpy as np
3  import pandas as pd
4  import os
5  import cv2
6  import tensorflow as tf
7  from tqdm import tqdm
8  import random
9  import sys
10 import pathlib
11 import shutil
12 import argparse
13
14 import torch
15 import torch.nn as nn
16 import torchvision
17 import torch.nn.functional as F
18 from torch.autograd import Variable
19 import torchvision.transforms as transforms
20 from torch.utils.data.sampler import SubsetRandomSampler
21 from torchvision import datasets
22 import torch.optim as optim
23 from torch.utils.data import Dataset, DataLoader
24 from torchvision import models
25
26 import sklearn
27 from sklearn.model_selection import train_test_split
28 from sklearn import preprocessing
29 from sklearn.preprocessing import StandardScaler, LabelEncoder
30 from sklearn import metrics
31 from sklearn.metrics import balanced_accuracy_score,
     classification_report
32 from sklearn.metrics import confusion_matrix
33 from sklearn.utils import shuffle
34
35 # Load data
36 df = pd.read_csv('~/Desktop/thesis.nosync/large-files.nosync/credit_card
     /creditcard.csv')
37
38 # Create train and test set (70:30 train:test)
39 X = df.drop('Class', axis=1)
```

```
y = df[['Class']]
X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    test_size = 0.30,
                                                    random_state = 42)

##### Define Loss Functions #####

class FocalLoss(nn.Module):
    """
    Focal Loss
    """
    def __init__(self, gamma = 0, alpha = None, size_average = True):
        super(FocalLoss, self).__init__()
        self.gamma = gamma
        self.alpha = alpha
        if isinstance(alpha,(float,int)): self.alpha = torch.Tensor([
   alpha,
                                                                    1 - alpha])
        if isinstance(alpha,list): self.alpha = torch.Tensor(alpha)
        self.size_average = size_average

    def forward(self, input, target):
        if input.dim()>2:
            input = input.view(input.size(0), input.size(1), -1)
            input = input.transpose(1, 2)   ]
            input = input.contiguous().view(-1, input.size(2))   ]
        target = target.view(-1, 1)

        logpt = F.log_softmax(input)
        logpt = logpt.view(-1)
        pt = Variable(logpt.data.exp())

        if self.alpha is not None:
            if self.alpha.type() != input.data.type():
                self.alpha = self.alpha.type_as(input.data)
            at = self.alpha.gather(0, target.data.view(-1))
            logpt = logpt * Variable(at)

        loss = -1 * (1 - pt)**self.gamma * logpt
        if self.size_average: return loss.mean()
        else: return loss.sum()


def _expand_binary_labels(labels, label_weights, label_channels):
    bin_labels = labels.new_full((labels.size(0), label_channels), 0)
    inds = torch.nonzero(labels >= 1).squeeze()
    if inds.numel() > 0:
        bin_labels[inds, labels[inds] - 1] = 1
    bin_label_weights = label_weights.view(-1, 1).expand(
        label_weights.size(0), label_channels)
    return bin_labels, bin_label_weights


```

```python
class GHMC(nn.Module):
    """
    GHM Classification Loss
    """
    def __init__(
            self,
            bins=10,
            momentum=0,
            use_sigmoid=True,
            loss_weight=1.0):
        super(GHMC, self).__init__()
        self.bins = bins
        self.momentum = momentum
        self.edges = torch.arange(bins + 1).float() / bins
        self.edges[-1] += 1e-6
        if momentum > 0:
            self.acc_sum = torch.zeros(bins).cuda()
        self.use_sigmoid = use_sigmoid
        if not self.use_sigmoid:
            raise NotImplementedError
        self.loss_weight = loss_weight

    def forward(self, pred, target, label_weight = 1, *args, **kwargs):

        if pred.dim() != target.dim():
            target, label_weight = _expand_binary_labels(
                                        target, label_weight, pred.size(-1))
        target, label_weight = target.float(), label_weight.float()
        edges = self.edges
        mmt = self.momentum
        weights = torch.zeros_like(pred)

        g = torch.abs(pred.sigmoid().detach() - target)

        valid = label_weight > 0
        tot = max(valid.float().sum().item(), 1.0)
        n = 0  # n valid bins
        for i in range(self.bins):
            inds = (g >= edges[i]) & (g < edges[i+1]) & valid
            num_in_bin = inds.sum().item()
            if num_in_bin > 0:
                if mmt > 0:
                    self.acc_sum[i] = mmt * self.acc_sum[i] \
                        + (1 - mmt) * num_in_bin
                    weights[inds] = tot / self.acc_sum[i]
                else:
                    weights[inds] = tot / num_in_bin
                n += 1
        if n > 0:
            weights = weights / n

        loss = F.binary_cross_entropy_with_logits(
            pred, target, weights, reduction='sum') / tot
        return loss * self.loss_weight
```

```

class LDAMLoss(nn.Module):
    """
    LDAM Loss
    """
    def __init__(self,
                 cls_num_list,
                 max_m = 0.5,
                 weight = None,
                 s = 30):
        super(LDAMLoss, self).__init__()
        m_list = 1.0 / np.sqrt(np.sqrt(cls_num_list))
        m_list = m_list * (max_m / np.max(m_list))
        m_list = torch.cuda.FloatTensor(m_list)
        self.m_list = m_list
        assert s > 0
        self.s = s
        self.weight = weight

    def forward(self, x, target):

        index = torch.zeros_like(x, dtype = torch.uint8)
        index.scatter_(1, target.data.view(-1, 1), 1)

        index_float = index.type(torch.cuda.FloatTensor)
        batch_m = torch.matmul(self.m_list[None, :],
                               index_float.transpose(0, 1))
        batch_m = batch_m.view((-1, 1))
        x_m = x - batch_m

        output = torch.where(index, x_m, x)
        return F.cross_entropy(self.s * output,
                               target,
                               weight = self.weight)

##### Neural Network #####

# Hyperparameters
BATCH_SIZE = 16
HIDDEN_NODES = 50
EPOCHS = 100
LEARNING_RATE = 0.0001
LOSS_FUN = nn.BCEWithLogitsLoss()

# Standardize data
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.fit_transform(X_test)

# Training data loader
class trainData(Dataset):

    def __init__(self, X_data, y_data):
        self.X_data = X_data
        self.y_data = y_data

    def __getitem__(self, index):
        return self.X_data[index], self.y_data[index]

    def __len__ (self):
        return len(self.X_data)

train_data = trainData(torch.FloatTensor(X_train),
                       torch.FloatTensor(y_train.values))

# Testing data loader
class testData(Dataset):

    def __init__(self, X_data):
        self.X_data = X_data

    def __getitem__(self, index):
        return self.X_data[index]

    def __len__ (self):
        return len(self.X_data)


test_data = testData(torch.FloatTensor(X_test))
train_loader = DataLoader(dataset = train_data,
                          batch_size = BATCH_SIZE,
                          shuffle = True)
test_loader = DataLoader(dataset = test_data, batch_size = 1)

# Define neural networks
class binaryClassification(nn.Module):
    def __init__(self):
        super(binaryClassification, self).__init__()
        # Number of input features is 30
        self.layer_1 = nn.Linear(30, HIDDEN_NODES)
        self.layer_2 = nn.Linear(HIDDEN_NODES, HIDDEN_NODES)
        self.layer_out = nn.Linear(HIDDEN_NODES, 1)

        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(p = 0.3)
        self.batchnorm1 = nn.BatchNorm1d(HIDDEN_NODES)
        self.batchnorm2 = nn.BatchNorm1d(HIDDEN_NODES)

    def forward(self, inputs):
        x = self.relu(self.layer_1(inputs))
        x = self.batchnorm1(x)
        x = self.relu(self.layer_2(x))
        x = self.batchnorm2(x)
        x = self.dropout(x)
        x = self.layer_out(x)

        return x

```

```python
# Define model, loss, learning rate
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model = binaryClassification()
model.to(device)
criterion = LOSS_FUN
optimizer = optim.Adam(model.parameters(), lr = LEARNING_RATE)

# Train model
def binary_acc(y_pred, y_test):
    y_pred_tag = torch.round(torch.sigmoid(y_pred))
    correct_results_sum = (y_pred_tag == y_test).sum().float()
    acc = correct_results_sum / y_test.shape[0]
    acc = torch.round(acc * 100)
    return acc

model.train()
loss_vals = []
for e in range(1, EPOCHS+1):
    epoch_loss = 0
    epoch_acc = 0

    for X_batch, y_batch in train_loader:
        X_batch, y_batch = X_batch.to(device), y_batch.to(device)

        optimizer.zero_grad()
        y_pred = model(X_batch)

        loss = criterion(y_pred, y_batch)
        acc = binary_acc(y_pred, y_batch)

        loss.backward()
        optimizer.step()

        epoch_loss += loss.item()
        epoch_acc += acc.item()

    loss_vals.append(epoch_loss/len(train_loader))
    print(f'Epoch {e+0:03}: \
          | Loss: {epoch_loss / len(train_loader):.5f} \
          | Acc: {epoch_acc / len(train_loader):.3f}')

# Test model
y_pred_list = []
model.eval()
with torch.no_grad():
    for X_batch in test_loader:
        X_batch = X_batch.to(device)
        y_test_pred = model(X_batch)
        y_test_pred = torch.sigmoid(y_test_pred)
        y_pred_tag = torch.round(y_test_pred)
        y_pred_list.append(y_pred_tag.cpu().numpy())

y_pred_list = [a.squeeze().tolist() for a in y_pred_list]

```

```python
# Determine results
print(classification_report(y_test, y_pred_list))
print(balanced_accuracy_score(y_test, y_pred_list))
```

List of Acronyms

ANN	Artificial Neural Network.
AUC	Area Under the Curve.
BCE	Binary Cross-Entropy.
BGR	Blue Green Red.
CE	Cross-Entropy.
CNN	Convolutional Neural Network.
FN	False Negative.
FNE	False Negative Error.
FNR	False Negative Rate.
FP	False Positive.
FPE	False Positive Error.
FPR	False Positive Rate.
GHM	Gradient Harmonizing Mechanism.
LDAM	Label-Distribution-Aware Margin.
MAE	Mean Absolute Error.
MFE	Mean False Error.
MSE	Mean Squared Error.
MSFE	Mean Squared False Error.
PCA	Principal Component Analysis.
ReLU	Rectified Linear Unit.
RGB	Red Green Blue.
ROC	Receiver Operating Characteristic.
ROS	Random Oversampling.

RUS	Random Undersampling.
SMOTE	Synthetic Minority Oversampling Technique.
SMOTEENN	Synthetic Minority Oversampling Technique Edited Nearest Neighbors.
SMOTETomek	Synthetic Minority Oversampling Technique Tomek-Based Resampling.
TN	True Negative.
TNR	True Negative Rate.
TP	True Positive.
TPR	True Positive Rate.

Symbols

i	indexing variable
j	indexing variable
k	indexing variable
x	input
$\mathbf{x}$	input vector
w	weights
$\mathbf{w}$	weight vector
y	output of a neural network
$\mathbf{y}$	output vector
$\hat{y}$	prediction output
h	height
d	depth
I	image
K	kernel
c	threshold
b	bias
s	hidden node size
μ	mean
v	observed value of a class
σ	standard deviation
$g(\cdot)$	logistic function
$f(\cdot)$	layer of a given perceptron
$C(\cdot)$	cost function
γ	learning rate
R	regularization term
λ	regularization parameter
N	number of batches
L_1	L_1 regularization
L_2	L_2 regularization
m	number of observations
n	number of variables
B	number of trees in a random forest
T	random forest trees