Final Project for CSDS440

Imbalanced Learning

Fabian Ardeljan, Prafful Chowdhary, Tucker Guen,
Yuqi Hu, Cathy Tao, Mei Wong

December 4, 2020

Contents

1 Introduction

1.1 The Class Imbalance Problem

Considering binary classification, the class imbalance problem occurs when there are significantly fewer examples of one class than the other. In training classifiers to learn to predict these classes, typical performance metrics and loss functions used can produce classifiers that perform well simply by classifying all or most examples as the majority class. For many imbalanced situations, however, the minority class is often the class of interest and the classifier cannot afford to misclassify the minority class to such a degree. This notion of class imbalance is also commonly intertwined with cost sensitivity, where the classification problem being solved gives more weight to the error on one class than another. Cost-sensitive problems are not necessarily imbalanced. However, it is often that class imbalance solutions and cost-sensitive solutions can be used to solve the same problem, since the two result from the same issue of disparity in user preference.

1.2 Survey of Solutions

In the literature, there are typically three distinct methods: data pre-processing, prediction post-processing, and modified algorithms. There are also algorithms that are a combination of the three [3].

1.2.1 Data Pre-Processing

The goal of data pre-processing is to equalize the distribution of the classes in the dataset. One of the most common ways is to perform oversampling of the minority class or undersampling of the majority class. In general there are two ways to do this: selectively duplicating/deleting samples or generating new, synthetic samples. Selectively duplicating samples in oversampling is beneficial because there is no uncertainty about whether or not the new points model the distribution of the points the original dataset. However, duplicating examples leads to lots of redundant data points and can cause overfitting. To combat these downfalls, synthetic techniques are used to generate new examples that are not copies, but instead are mathematically similar to existing points in the minority class. While this may reduce overfitting and avoid redundancy, the method used to generate the synthetic examples will have a big impact on the generality of the classifier trained on these datasets. Methods for undersampling have similar implementations, where some randomly remove points and others heuristically remove them. There are also other methods that combine these listed ones, where some degree of both undersampling and oversampling are used to equalize the class distributions.

The benefits of these types of data pre-processing techniques are that they can be readily applied to any classifier, since they only modify the data. It is also easy for the user to modify the degree to which the distribution is equalized, meaning that the user can specifically give preference to one class over another. However, determining this degree of influence is harder than it sounds, and often the correct distribution can be difficult to determine and requires trial and error. Sections 2, 3, 4, and 6 detail various pre-processing techniques.

1.2.2 Prediction Post-Processing

The prediction post-processing methods consist of adjusting the classifications outputted by the classifier. The area consists of two methods: threshold modification and cost-sensitivity. Threshold modification simply adjusts the threshold that determines what range of outputs of the classifier are considered a positive class prediction or a negative class prediction. There are a variety of methods for selecting the correct threshold for a given classifier and dataset imbalance. It was shown that this method can perform just as well as over and under sampling as well as modified algorithm techniques. Cost-sensitive post-processing is a relatively under-studied area for classification problems but has been studied in regression. It consists of changing the predictions of the model to be cost sensitive. Although this hasn't been applied directly to imbalanced classification, the close relationship between cost-sensitivity and imbalance means that it is likely applicable and could be studied in future work.

Considering that prediction post-processing is not widely studied, particularly in imbalanced, binary classification, it is hard to say empirically what the pros and cons are. However, conceptually, prediction post-processing is clearly beneficial in that it can be applied to any existing, trained learning algorithm. On the downside, the methods don't explicitly bake the data imbalance problem into the learning method, meaning that the loss function that is optimized isn't really in accordance with any specific imbalance in the data or cost preference of the user.

1.2.3 Modified Algorithms

Modified algorithms are formulations of specific learning algorithms where the importance of the error on one class is built into the learning/optimization method. Often this will mean that the algorithm considers error on the minority class more strongly than errors on the majority class, in order to balance out the influence of each on the optimization. For most well known learning algorithms such as SVM's, MLP's, Decision Trees, k-NN's and Naive Bayes there exist a modified version for cost-sensitivity and/or imbalanced learning.

Modified learning algorithms can be greatly beneficial since the learning process is minimizing a loss that is directly related to the imbalance or cost preference of the user. However, the fact that the method restricts the user to a single learning algorithm is a massive disadvantage as compared to the other two methods. Additionally, modifying the loss function and incorporating it into a learning algorithm is a non-trivial task and any change in the loss function requires a new formulation of the algorithm. Sections 4 and 7 detail two modified learning algorithms.

1.3 Shared Datasets

All team members used the Vehicle1 and SolarFlareF datasets from the Machine Learning Repository from the University of California Irvine. Neither dataset contains missing data or noise. The Vehicle1 dataset contains only continuous integer data and a single binary target attribute. The SolarFlareF dataset contains nominal data and integer data that can be treated as nominal and originally had three target attributes, but was simplified to a single binary target attribute.

2 Fabian Ardeljan - ADASYN

2.1 Papers Read

ADASYN The first paper topic selected for Imbalanced Learning was ADASYN, for which two papers were read and analyzed. The first paper, titled "ADASYN: Adaptive synthetic sampling approach for imbalanced learning" [7] is the paper that first proposed and laid the groundwork for the algorithm. It proposes the ADASYN algorithm as a method for creating synthetic data for minority examples by using a weighted distribution that prioritizes difficult to learn examples. Then, it creates a number of synthetic examples that is proportional to the difficulty of each minority example. This method proves to be highly effective at reducing the bias introduced when tested on imbalanced data, and successfully shifts the classification boundary towards difficult to learn examples.

The second paper related to ADASYN, titled "Adaptive Synthetic–Nominal (ADASYN–N) and Adaptive Synthetic–KNN (ADASYN-KNN) for Multiclass Imbalance Learning on Laboratory Test Data" [8], takes the ADASYN algorithm a step further by introducing two variations to handle nominal data. The two algorithms, ADASYN-N and ADASYN-KNN, used different approaches to handle multiclass nominal neighbors. In ADASYN-N, the nearest neighbors were found using a modified version of the Value Difference Metric (VDM) as is done in SMOTE-N. In ADASYN-KNN, the traditional K-nearest neighbor algorithm was used instead. Using these techniques, the team was able to produce synthetic examples for cervical cancer tests in Indonesia with nearly identical results to the SMOTE-N algorithm used for comparison.

Boosting Weighted ELM The second paper topic revolves around how Imbalanced Learning can be applied to Extreme Machine Learning (ELM), an innovative approach to machine learning that uses a single-hidden-layer feedforward neural network. One paper was selected for this topic titled "Boosting weighted ELM for imbalanced learning" [9]. The paper discusses the successful use of a modified AdaBoost algorithm to handle imbalanced data in an ELM. This was done by assigning an extra weight to each training example that gets updated with each iteration. The AdaBoost algorithm was modified such that the initial distribution weights are asymmetrical for faster convergence, which were then updated separately to maintain the asymmetry. The results successfully proved that the proposed method could achieve more balanced results than weighted ELMs without boosting, providing a useful technique for future ELM research.

2.2 Algorithms

Due to its fundamental importance in Imbalanced Learning research, the ADASYN algorithm was chosen for implementation. Two versions of this algorithm were created and compared: the original version as discussed in the proposal paper [7], and a modified version known as Bidirectional ADASYN, which first eliminates redundant examples from the majority class and then creates synthetic examples for minority classes.

2.2.1 ADASYN

The original ADABOOST algorithm as described in the proposal paper [7] is as follows:

(1) Calculate the degree of class imbalance:

$$d = m_s/m_l \qquad (1)$$

where $d \in (0,1]$.

(2) If d<d_{th} (d_{th} is a preset threshold for the maximum tolerated degree of class imbalance ratio), then:

(a) Calculate the synthetic data examples that need to be generated for the minority class:

$$G = (m_s/m_l) \times \beta \qquad (2)$$

where $\beta \in [0,1]$ is a parameter used to specify the desired balance level after generation of the synthetic data. $\beta = 1$ means a fully balanced dataset is created after the generalization process.

(b) For each example $x_i \in minorityclass$, find K nearest neighbors based on the Euclidean distance in n dimensional space, and calculate the ratio r_i defined as:

$$r_i = \Delta_i/K_i, i = 1, ..., m_s \qquad (3)$$

where Δ_i is the number of examples in the K nearest neighbors of x_i that belong to the majority class, therefore $r_i \in [0,1]$;

(c) Normalize r_i according to $\hat{r}_i = r_i/\sum_{i=1}^{m_s} r_i$, so that $\hat{r}_i$ is a density distribution $(\sum_i \hat{r}_i = 1)$

(d) Calculate the number of synthetic data examples that need to be generated for each minority example x_i:

$$g_i = \hat{r}_i \times G \qquad (4)$$

where G is the total number of synthetic data examples that need to be generated for the minority class as defined in Equation (2).

(e) For each minority class data example x_i, generate g_i synthetic data examples according to the following steps:

Do the **Loop** from 1 to g_i:

(i) Randomly choose one minority data example, x_{zi}, from the K nearest neighbors for data x_i.

(ii) Generate the synthetic data example:

$$s_i = x_i + (x_{zi} - x_i) \times \lambda \qquad (5)$$

where $(x_{zi} - x_i)$ is the difference vector in n dimensional spaces, and λ is a random number: $\lambda \in [0,1]$.

End **Loop**

As there are many public open-source implementations of ADASYN already available, this paper's coded implementation focused on expanding the usefulness of ADASYN rather than merely rewriting it. The open-source implementation from Stavrianos Skalidis [14], which precisely follows the steps listed above, was used for inspiration, but modified in two significant ways. First, the entire code was rewritten to fit within a greater algorithm. This included changing not only the functions, but also how they operate, such as appending new examples to the end rather than to the beginning of the array of examples, such that the original indexing could be retained.

The second and most significant addition to the classic ADASYN algorithm was adding pre- and post-processing capabilities to the algorithm such that an ADASYN resampled copy of the original dataset is created that can be read by any other algorithm. This was done for both of the common .CSV and C4.5 formats, such that the new dataset could be read with the same Name file as the original. Indexing for the new file was also kept consistent with the old one, with new examples continuing the original index, such that examples could be easily cross-referenced between files. Other index features such as "ID" retained their value for the old examples and used a "-1" value to signify a synthetic example, which could be easily found and modified if needed. The new data file contains a suffix to its name to be distinguished from the original, such as "flare_a.data" or "flare_b.data"

2.2.2 Bidirectional ADASYN

The proposed modified version of ADASYN, named Bidirectional ADASYN, seeks to reduce the runtime of algorithms using lengthy resampled ADASYN data while maintaining all the benefits of using ADASYN. This is done by first removing redundant majority class data and then creating ADASYN synthetic examples to match a smaller majority data size. Although other sensitivity-based undersampling algorithms such as DDUS exist that do that in their own way, Bidirectional ADASYN undersamples data using a novel algorithm to test for redundancy in majority examples.

The algorithm for testing redundancy for a set of examples X is done as follows:

(1) Create an empty set of non-redundant values X_{new}.

(2) Add the first value in X to X_{new}.

(3) Using a sensitivity parameter s, for each remaining example x in X, test that for each example x_{new} in X_{new}, all examples are varied enough such that at least one feature satisfies the inequality

$$\frac{min(x_{\mathrm{i}}, x_{\mathrm{newi}}) + 1}{max(x_{\mathrm{i}}, x_{\mathrm{newi}}) + 1} \geq s$$

where x_i represents the value of x for feature i.

(4) If the inequality is satisfied, the example x is added to X_{new}, otherwise it is discarded.

(5) Repeat steps 1-4 with a different sensitivity value s until a suitable percentage of the dataset is removed (if too much data is removed, increase s, and vice versa).

The value of s may need to be adjusted several times on a subset of X until a desirable fractional size is determined for X_{new}, before running the algorithm on the whole set X. The runtime of the redundancy algorithm varies between O(n) and O(n^2) depending on the value of s. Therefore, Bidirectional ADASYN is recommended only for creating datasets that will be used repeatedly rather than for single use. The benefit of using Bidirectional ADASYN is directly related to how redundant the original dataset is as well as how much runtime is aimed to be saved.

2.3 Implementation

The above algorithms were implemented exclusively in Python using the JupyterLab program through the Anaconda software package. The pandas and numpy libraries were the primary source for data handling. The matplotlib library was used for visualization. The Counter function from collections was used for counting. Several functions from sklearn libraries were used to simplify the algorithm, such as NearestNeighbors, check_array, check_random_state, datasets, and PCA. The sklearn LabelEncoder and LabelBinarizer functions for handling nominal data. Lastly, the mldata library was used for handling C4.5 format datasets.

For programming and testing, the SolarFlareF and Vehicle1 datasets from the University of California Irvine that the team shared were used.

2.4 Experiments and Results

The results of the ADASYN and Bidirectional ADASYN are presented in two ways. For the first part, each dataset is graphically represented before and after each algorithm was used, with the value of each example as the average of each feature and the class represented by a labeled color. For the second part, experiments were performed using a machine learning algorithm on each dataset before modification, after modification with ADASYN, and after modification with Bidrectional ADASYN

Preliminary Results The following outputs were produced by each algorithm for the Vehicle1 and SloarFlareF datasets:

```
Majority class is 0 and total number of classes is 2
Class 0 (647/846) is within imbalance threshold
Class 1 (199/846) will be oversampled
The new distribution for Class 1 is 626/1273
```

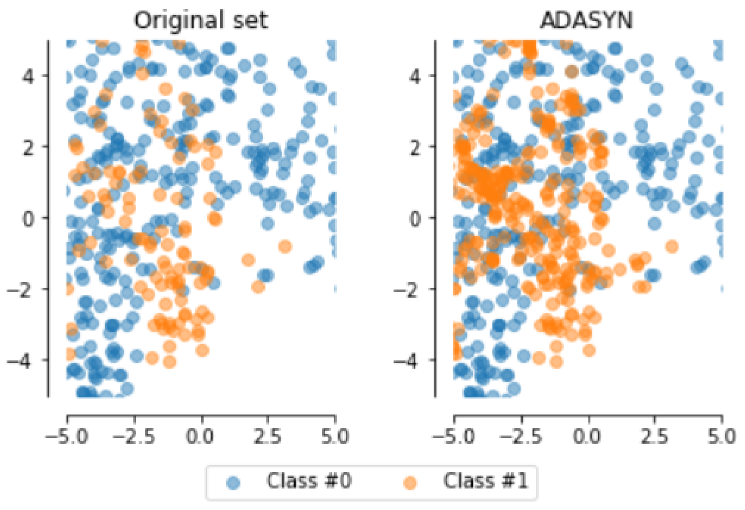


Figure 1: Graphical representation of ADASYN algorithms using Vehicle1 dataset

```
Majority class is 0 and total number of classes is 2
There are 647 instances of the majority class
The majority class has been downsampled to 608 instances
Class 0 (608/807) is within imbalance threshold
Class 1 (199/807) will be oversampled
The new distribution for Class 1 is 602/1210
```

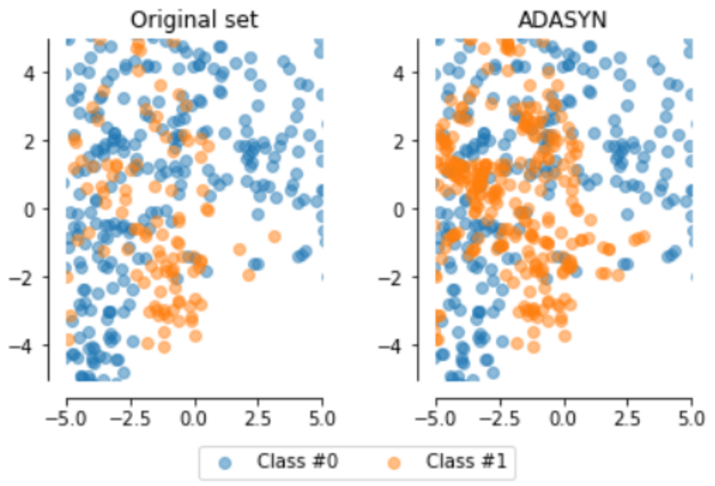


Figure 2: Graphical representation of Bidirectional ADASYN algorithms using Vehicle1 dataset

```
Majority class is 0 and total number of classes is 2
Class 0 (1130/1389) is within imbalance threshold
Class 1 (259/1389) will be oversampled
The new distribution for Class 1 is 1006/2136
```

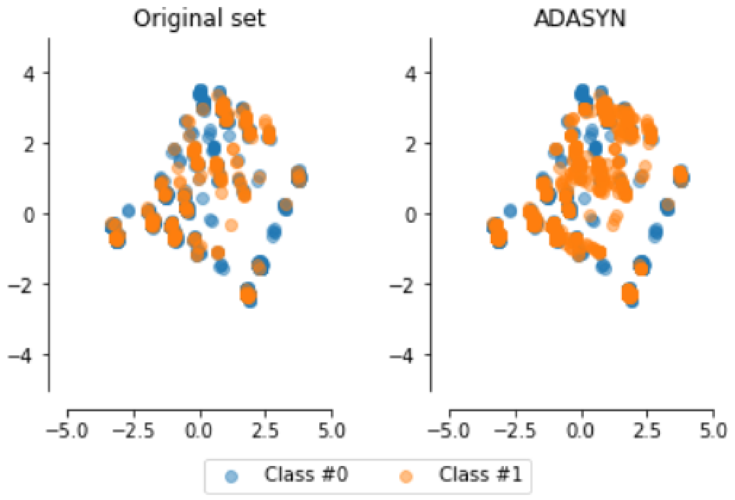
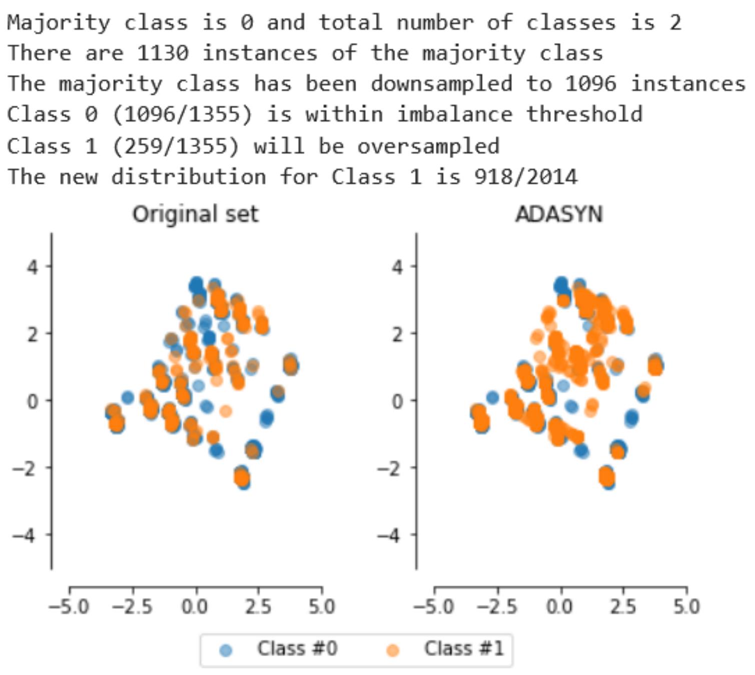


Figure 3: Graphical representation of ADASYN algorithms using SolarFlareF dataset

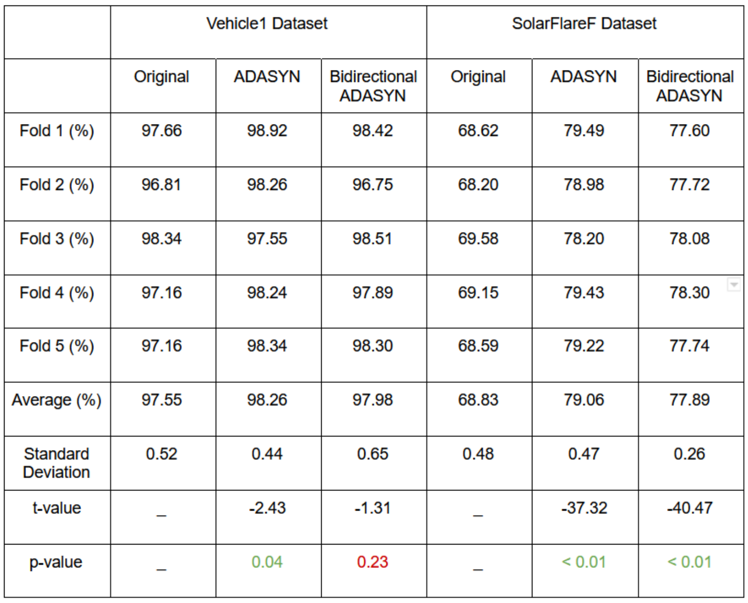


```
Majority class is 0 and total number of classes is 2
There are 1130 instances of the majority class
The majority class has been downsampled to 1096 instances
Class 0 (1096/1355) is within imbalance threshold
Class 1 (259/1355) will be oversampled
The new distribution for Class 1 is 918/2014
```

Figure 4: Graphical representation of Bidirectional ADASYN algorithms using SolarFlareF dataset

Experiments The graphical representation provides an intuitive understanding of the datasets before and after running each algorithm. However, to test whether the results truly help improve accuracy in machine learning algorithms, experiments must be run with the original and modified datasets in a real machine learning scenario.

In order to do this, a simple Decision Tree algorithm using C4.5 data was used to test both datasets. In different experiments, the tree used stratified 5-fold cross validation using the original vs resampled datasets to learn, but was always tested with the original datasets. There was no growth limit or pruning provided to any tree, and the tree only returned a weighted accuracy score average and standard deviation. The weighted accuracy results of all datasets as produced by the Decision Tree are as follows:



	Vehicle1 Dataset			SolarFlareF Dataset		
	Original	ADASYN	Bidirectional ADASYN	Original	ADASYN	Bidirectional ADASYN
Fold 1 (%)	97.66	98.92	98.42	68.62	79.49	77.60
Fold 2 (%)	96.81	98.26	96.75	68.20	78.98	77.72
Fold 3 (%)	98.34	97.55	98.51	69.58	78.20	78.08
Fold 4 (%)	97.16	98.24	97.89	69.15	79.43	78.30
Fold 5 (%)	97.16	98.34	98.30	68.59	79.22	77.74
Average (%)	97.55	98.26	97.98	68.83	79.06	77.89
Standard Deviation	0.52	0.44	0.65	0.48	0.47	0.26
t-value	_	-2.43	-1.31	_	-37.32	-40.47
p-value	_	0.04	0.23	_	< 0.01	< 0.01

Figure 5: Weighted accuracy of Decision Tree algorithm trained by each dataset

Performing a T-test on the resulting data determined that the SolarFlareF data for both algorithms showed a significant improvement at a $p < 0.01$ level. However, for the Vehicle1 dataset, only the ADASYN algorithm showed significant improvement, with a borderline satisfactory p value of 0.04. The reason why the two datasets perfomed so differently was because the Vehicle1 dataset was already producing near-perfect results on the Decision Tree algorithm without any ADASYN support, despite the data being imbalanced. The improvement therefore could only be minuscule, so the dataset cannot properly represent the potential of the ADASYN or Bidirectional ADASYN datasets. From the SolarFlareF dataset, however, it is evident that both ADASYN and Bidirectional ADASYN provide a significant boost to imbalanced data learning.

2.5 Conclusion

The results of the experiment showed that both ADASYN and Bidirectional ADASYN are able to signficantly boost a machine learning algorithm's ability to learn an imbalanced dataset. Although the ADASYN algorithm produced a higher weighted accuracy score for both datasets, the aim of the Bidirectional ADASYN algorithm is to reduce the overall size of the data produced by ADASYN without massively altering the results. As such, Bidirectional ADASYN has a sensitivity parameter that allows the user to determine just how much redundant data to discard form a dataset, which provides a useful tool when managing a tradeoff between results and computation time. Overall, both algorithms have potential value for a variety of situations.

3 Prafful Chowdhary - DSUS

3.1 Papers Read

Sensitivity-Based Undersampling The first paper selected was for Sensitivity-Based Undersampling for Imbalanced Data set, the paper selected for this is titled "Diversified Sensitivity-Based Undersampling for Imbalance Classification Problems" [12]. This paper explains how undersampling is a methods used for pattern classification for imbalanced data-set. The methods used back in the day for imbalanced data classification were Random-based Undersampling and resampling, but as discussed in the paper both the methods had to ignore the distribution information present in the training dataset. The paper proposes a diversified sensitivity-based undersampling method. Alogrithms made and improved by me are based on this paper.

The problem mentioned in the paper is in classification problems if in the dataset data points for all classes are not equal or balanced then the model trained with it may not be generalized properly. Model's output would be more biased towards the class for which more data is available. To avoid that during the processing of the training dataset, the dataset is balanced so that the number of samples in each class remains the same or nearner.

There are several techniques like random oversampling (ROS) where data is minority class is increased by adding dummy data points to match the sample size. And random undersampling (RUS) by reducing datasets from the majority classes. There is a problem with these techniques that is the loss of distribution information. Author had specifically targeted this problem.

Training Deep Neural Networks The Second paper selected was for Training Deep Neural Network for imbalanced datasets, the paper selected for this is titled "Training deep neural networks on imbalanced data sets" [15]. This paper explains how a Neural network can be trained with imbalanced dataset because back in the day neural networks were trained on balanced class labels. But in real life the presence of imbalanced data is everywhere and that had been a real challenge to do classification on imbalanced data. The paper speaks about a method mean false error and mean squared false error which is an improved version of the first one, these methods are used to classify errors in the classes both majority and minority. The experiments done in this paper show that the proposed methods are a lot better than the conventional methods.

3.2 Algorithms

As i mentioned earlier for the algorithm the first paper was used, which is "Diversified Sensitivity-Based Undersampling for Imbalance Classification Problems" [12]. The problem is discussed in the Papers Read section. The solution to that problem as proposed by author is Diversified sensitivity undersampling(DSUS) techniques to select the useful data. The proposed method data is first grouped in majority and minority class data. And then using the k-means clustering data were clustered and data points near to centroid from each class's dataset is taken and a Radial basis function network (RBFNN) is trained. Then further Sensitivity Measure is calculated using the provided equation and RBFNN is trained with selected data points and repeated. Number of clusters was calculated from the size of the data set.

The proposed algorithm is implemented for binary classification problems. And Author mentioned that time complexity is much worse than that of RUS due to that SM calculation. But able to get much meaningful data. Author tested the proposed method on various datasets and compared it with multiple methods. Author concluded that the proposed algorithm DSUS is robust to different imbalanced ratios. And the SM calculator for oversampling proposed as future work.

The proposed algorithm required 3 task 1) Clustering of Datasets, 2) SM calculation 3) Radial Basis Function Neural Network for calculating one of the terms in SM calculation equation.

Following libraries were used in the program

No.	Library	Reason
1	pandas	To read dataset
2	numpy	For mathematical operations
3	scipy	For calculation of error function
4	matploitib	For plotting the histogram
5	sklearn	For k-means clustering and MLP Classifier
6	scipy	To get the data points which are near to centroids

Author used RBFNN for SM calculation for undersampling the majority class to leave meaning full dataset for training. RBFNN class is coded. With SM calculation implementation. Mathematically output of RBFNN is defined by following equation

$$f(\mathbf{x}) = \sum_{j=1}^{M} w_j \exp\left(\frac{\| \mathbf{x} - \mathbf{u}_j \|^2}{-2v_j^2}\right)$$

When a class is initialized it Stores the Training data and initializes the empty centroids (uj), weight matrix (wj) and width parameter (Vj). For training of RBFNN, 2 Step processes need to be followed. 1) Cluster the training data and find the centroid (uj) and 2) Then using the training data (X,y) weight matrix is calculated (wj) using least squares estimation equations.

Number of Cluster or neurons (M) in RBFNN is selected as the square root of training dataset size. First step of training is implemented using the k-means clustering and implemented in "generate_centroid" method of the RBFNN class. That method finds the centroids (uj) and weight parameters for each (Vj) neuron. Second step of training is implemented using the LSE equation which is as follows W = (B'B)-1B'y.

Before calculating the W, singularity of (B'B) is calculated and if found singular the some random values were added to avoid the inversion errors. There is "adapt" method also implemented to train with new data which basically repeats the above procedure with new data.

For calculation of SM following equations were implemented as methods in code.

$$\begin{aligned} SM &= \int\limits_{S_Q(\mathbf{x})} (f(\mathbf{x}_b) - f(\mathbf{x}))^2 p(\mathbf{x}) d\mathbf{x} \\ &= \int\limits_{S_Q(\mathbf{x})} \left(f(\mathbf{x}_b)^2 - 2f(\mathbf{x}_b)f(\mathbf{x}) + f(\mathbf{x})^2\right) p(\mathbf{x}) d\mathbf{x} \\ &= f(\mathbf{x}_b)^2 - 2f(\mathbf{x}_b) \int\limits_{S_Q(\mathbf{x})} f(\mathbf{x}) p(\mathbf{x}) d\mathbf{x} + \int\limits_{S_Q(\mathbf{x})} f(\mathbf{x})^2 p(\mathbf{x}) d\mathbf{x} \\ &= f(\mathbf{x}_b)^2 - 2f(\mathbf{x}_b) I_1 + I_2. \end{aligned}$$

Where

$$
\begin{aligned}
I_1 &= \int\limits_{S_Q(\mathbf{x})} f(\mathbf{x})p(\mathbf{x})d\mathbf{x} \\
&= \frac{1}{(2Q)^n} \int\limits_{S_Q(\mathbf{x})} f(\mathbf{x}_b + \Delta\mathbf{x})d\mathbf{x} \\
&= \frac{1}{(2Q)^n} \int\limits_{S_Q(\mathbf{x})} \sum_{j=1}^{M} w_j \exp\left(\frac{\sum_{i=1}^{n}\left(\Delta x_i + x_{bi} - u_{ji}\right)^2}{-2v_j^2}\right)dx \\
&= \left(\frac{\sqrt{\pi}}{2\sqrt{2}Q}\right)^n \sum_{j=1}^{M} w_j \prod_{i=1}^{n} \left(v_j\left(\operatorname{erf}\left(\frac{x_{bi} - u_{ji} + Q}{\sqrt{2v_j^2}}\right)\right.\right. \\
&\qquad\qquad \left.\left. - \operatorname{erf}\left(\frac{x_{bi} - u_{ji} - Q}{\sqrt{2v_j^2}}\right)\right)\right)
\end{aligned}
$$

And

$$
\begin{aligned}
I_2 &= \int\limits_{S_Q(\mathbf{x})} f(\mathbf{x})^2 p(\mathbf{x})d\mathbf{x} \\
&= \frac{1}{(2Q)^n} \int\limits_{S_Q(\mathbf{x})} f(\mathbf{x}_b + \Delta\mathbf{x})^2 d\mathbf{x} \\
&= \frac{1}{(2Q)^n} \int\limits_{S_Q(\mathbf{x})} \left(\sum_{j=1}^{M} w_j \exp\left(\frac{\sum_{i=1}^{n}\left(\Delta x_i + x_{bi} - u_{ji}\right)^2}{-2v_j^2}\right)\right)^2 d\mathbf{x} \\
&= \left(\frac{\sqrt{\pi}}{4Q}\right)^n \sum_{j,k=1}^{M} \left(\left(\frac{\sqrt{2v_j^2 v_k^2\left(v_k^2 + v_j^2\right)}}{v_k^2 + v_j^2}\right)^n\right. \\
&\quad \prod_{i=1}^{n}\left(\left(\operatorname{erf}\left(\frac{\left(v_k^2 + v_j^2\right)\left(x_{bi} + Q\right) - \left(v_k^2 u_{ji} + v_j^2 u_{ki}\right)}{\sqrt{2v_j^2 v_k^2\left(v_k^2 + v_j^2\right)}}\right)\right.\right. \\
&\qquad \left.\left.\left. - \operatorname{erf}\left(\frac{\left(v_k^2 + v_j^2\right)\left(x_{bi} - Q\right) - \left(v_k^2 u_{ji} + v_j^2 u_{ki}\right)}{\sqrt{2v_j^2 v_k^2\left(v_k^2 + v_j^2\right)}}\right)\right)\right)\right).
\end{aligned}
$$

3.3 Implementation

Equations were implemented using the for loops and numpy arrays. In “SM”, “I1” and “I2” methods of RBFNN class. Rest of the program follows the algorithm provided in literature as follows,

1. Training the initial RBFNN
   (a) Cluster both U_n and U_p into k = Np clusters each.
   (b) Set both P_0 and R_0to be empty sets.
   (c) For each of k clusters of the minority class, add the sample located closest to its center to P_0.
   (d) For each of k clusters of the majority class, add the sample located closest to its center to R_0.
   (e) U_p = U_p P_0 , U_n = U_n R_0, S = P_0 R_0 and b=0
2. Train a RBFNN using S
   while n_p (greator or equal) to k do

```
        (a) Find representative samples in the majority class(C).
        (b) Set C,R_b and P_b to be empty sets
        (c) n_p= size(Up) and b=b+1
        (d) Cluster U_n into n_p clusters
        (e) For each of n_p clusters of U_n, add the sample located closest to its center to C
        (f) Sample Selection using the Sensitivity Measure
        (g) Compute the SM value for each sample in both C and U_P with the current RBFNN
        (h) Add k samples from C yielding the largest SM values to R_b
        (i) Add k samples from U_p yielding the largest SM values to P_b
        (j) Up = Up CP_b and U_n = U_n CR_b , S = S  P_b  R_b
        (k) Train a RBFNN usingS
    end while
```

To measure the results of Sampling method F1 Score is used which can be calculated using the following equation. F1 = 2 * (PR)/(P+R) Where P is precision and R is recall that can be calculated from the confusion matrix. However here f1 score calculating tool from scikit learn library is used.

3.4 Experimental results and Improvement

Experimental Results Author used those data to classify using the RBFNN but here, 3- MLP Classifier models are developed and each trained with differently treated data of the same dataset.
Model - 1: MLP Classifier trained with the given dataset as it is.
Model - 2: MLP Classifier trained with the Balanced dataset, achieved with above mentioned algorithm.
Model - 3: MLP Classifier trained with the Balanced Dataset, achieved with random undersampling method.

Two dataset were used, those dataset have higher imbalance ratio. For Comparison k-fold validation for k = 10 is considered and the average value of F1 for each method is calculated. Following are the results

	F1 Scores			Data Selection Time (sec)	
Dataset	Model 1	Model 2 (DSUS)	Model 3 (RUS)	DSUS	RUS
Flare Dataset	0.242489	0.339667	0.359285	47.759	0.022
Vehicle Dataset	0.952592	0.979324	0.988098	21.4624	0.009898

The DSUS method takes much more time compared to the RUS method. Also the F1 Score of RUS is more than the DSUS method. From the DSUS algorithm by removing the SM calculation and RBFNN and just using the k-means clustering method for sample selection from the majority class. That will reduce the time also.

Improvement Instead using Sensitivity Measure to select the data points, we can just use the data points near to centroids of clusters of datasets created by k-means clustering, to reduce the time and with little bit compromise in classification accuracy.

Following are the algorithm Steps,

```
1.  (a) Separate Dataset in Majority and minority class in U_n and U_p variables.
```

```
    (b) Take k = sqrt(N_p); where N_p is size or total number of samples of minority class
    (c) Cluster Data in U_n and U_p separately for k number of clusters. And find the centroid
    (d) Select the data points which are nearest to centroid from each k cluster of each data group.
        And Store in P and R variables which are from U_p and U_n respectively
    (e) Remove P and R from U_n and U_p sets and take S = P U R
2. While (1) do
    (a) Take n_p = size(U_p)
    (b) Check If n_p < k:
        Break Loop
    (c) Cluster datas from U_n into n_p clusters.
    (d) Find the points from U_n nearest to the centroids and Store in C
    (e) Take first k elements from C and store in Rb and remove those elements from U_n
    (f) Take first k elements from U_p and store in Pb and remove those elements from U_p
    (g) Take S = S U Rb U Pb
```

S is an undersampled data set.

The SM and RBFNN calculation is removed from the previous algorithm. Same takes is conducted to evaluate the results and yield the following results.

	F1 Scores			Data Selection Time (sec)	
Dataset	Model 1	Model 2 (Improved)	Model 3 (RUS)	Improved	RUS
Flare Dataset	0.242482	0.353597	0.348302	5.7	0.024796
Vehicle Dataset	0.956004	0.986815	0.988535	1.667	0.009852

This new updated algorithm is taking around 80% less time than the DSUS algorithm. And Generating up to 5% higher F1 score than the DSUS. However compared to RUS, the difference in F1 score is not much. Major changes in DSUS algorithm as removal of RBFNN and SM calculation.

3.5 Conclusion

As mentioned in the paper the proposed DSUS method preserves the distribution information of the majority class and select information from both majority and minority class. The results show that the DSUS method takes much more time compared to RUS method. But as in method instead of using Sensitivity Measures to select the data points, we can just use the data points near to centroids of clusters of datasets created by k-means clustering, to reduce the time and with little bit compromise in classification accuracy. In the Improvement done by me for the existing algorithm (same as above mentioned) which was also their future work it seems to make the algorithm better it generated 5% higher F1 score and also reduced the time by roughly 80%. Back in the day the work done on imbalanced dataset was less but now we have more and means to make the algorithm better as demonstrated above.

4 Tucker Guen - Cost Sensitive Multilayer Perceptron

4.1 Papers Read

Cost Sensitive Multilayer Perceptron (CSMLP)
The authors describe a cost-sensitive multilayer perceptron detailed in [4] that proposes a modified loss function that uses a hyperparameter λ to weight the effects of the positive class and negative class errors during the optimization of the multilayer perceptron (MLP) weights. It also proposes a modification of the Levenberg-marquardt (LM) weight optimization algorithm to account for the new, weighted cost function. The paper goes on to detail the theoretically optimal value for λ, and tests their theories on both synthetic and real-world datasets.

A related paper entitled "Levenberg-marquardt algorithm with adaptive momentum for the efficient training of feedforward networks" [1] presents the LM algorithm in detail and describes its trade-offs. Particularly, it recognizes the problem of LM converging to sub-optimal, local minima, which forces training to be re-rerun repeatedly until the algorithm converges to another minimum. To correct this, the authors introduce a "momentum term" which modifies the weight update rule to account for the value of the second order derivative at each step. We considered this paper for help in implementing LM as well as in designing the extension to help avoid local minima.

Synthetic Minority Oversampling Technique (SMOTE)
This paper details a method for introducing new, synthetic samples of the minority class to an imbalanced dataset. It loops over every point in the minority class and randomly picks one of the *k* nearest neighbors to that point. It then randomly picks a point on the line between the original point and the random neighbor to be a new synthetic sample of the minority class. It repeats this neighbor selection and interpolation step a number of times depending on the percent of oversampling desired. The authors demonstrate that the technique works well for a decision tree classifier, Ripper, and a Naive Bayes Classifier as compared to the base methods and undersampling methods with those classifiers.

Dynamic sampling approach to training neural networks for multiclass imbalance classification
To help synthesize the effects of SMOTE and oversampling techniques on MLP's, we also looked at [10], which specifies a method that dynamically modifies the influence that any one example has on the weight update. Although this paper discusses multiclass imbalance, many of the comments that they make about SMOTE and MLP's are applicable to binary classification. Primarily, the authors' comments on the problem of overlapping and overfitting with oversampling techniques as well as the dynamic importance of each class during weight updates are of interest. First, they imply that SMOTE is an ineffective algorithm for training MLP's in certain datasets, particularly multiclass. Second, their comments imply that CSMLP which works by weighting the minority class by a constant amount during training may not perform well, particularly on multiclass problems. Interestingly though, the authors of [4] state that CSMLP with slight modifications should be effective for multiclass problems.

4.2 Implemented Algorithms

Implementation Details
The following MLP/CSMLP and SMOTE algorithms are written in Python with array and matrix oper-

ations done using Numpy. SMOTE and the extension for SMOTE makes use of the scikit-learn Python packages for k-nearest-neighbors and k-means clustering. The matplotlib library was used for plotting datasets and graphs. All other algorithms and implementations were done from scratch.

4.2.1 Cost Sensitive Multilayer Perceptron

Algorithm Overview

The CSMLP algorithm is a modification of the MLP algorithm for binary classification. The specific topology studied is a fully connected MLP with one input layer of n units, one hidden layer of h units, and an output layer of a single unit. For this formulation, the output of the MLP given an example vector $\mathbf{x} = \{x_1, x_2, ..., x_n\}$, is given by [4](Equation 9), or equivalently in matrix form

$$\hat{f} = \phi(v) = \phi\Big(W_s \cdot \phi\big(W_{sr} \cdot x\big)\Big) \tag{1}$$

Where W_{sr} is the $n \times h$ matrix containing each weight w_{sr} from input unit r to hidden unit s, and W_s is the column vector of length h containing each weight from the hidden unit s to the output unit. This is the formulation used in the implementation.

The novel loss function presented in [4](Equations 10-12) is a weighted sum function of the form

$$\lambda x + (1-\lambda)y,\ 0 \le \lambda \le 1 \tag{2}$$

The training set T is broken down into two subsets such that $T = \{T_1 \cup T_2\}$ where T_1 is of length N_1 and contains all example vectors $\mathbf{x_i}$ of the positive class and T_2 is of length N_2 containing all vectors of the negative class. The loss function is in the form of (2), where x and$y$ are J_k =sum of squared errors over T_k, $k = 1, 2$. Note that $\lambda = 0.5$ is equivalent to the standard, unweighted sum squared errors function.

The study in [4] then presents a modified weight update based on the Levenberg-Marquardt (LM) algorithm for optimizing nonlinear least squares problems. The derivation of the modified weight update rule [4](Equations 13-24) follows the form of (2), where the Hessian matrix and gradient vector required by LM are independently computed for sets T_1, T_2, and then weighted. The authors provide a formulation of the Jacobian [4](Equation 19) used in the computation of the Hessian and gradient. However, they do not specify the equation for $\partial e(i)/\partial w_l$, the partial derivative terms of the Jacobian, where $e(i) = y(i) - \hat{f}(i)$, $y(i)$ = true class label $\in \{-1, 1\}$, and i is the i^{th} training example. The next subsection derives the the computation of the Jacobian for the input to hidden layer weights and the hidden to output layer weights.

Derivation of the Jacobian

The Jacobian for example set T_k, $k = 1, 2$ is given by [4](Equation 19). Each row is an example in the example set T_k, and each column a weight in the network. For simplicity, the Jacobian for the weights in W_s and W_{sr}, and subsequently the Hessian, gradient, and weight update term in LM are computed separately. This still yields the same weight update as computing them as one. The Jacobian for each can then be respectively written as

$$Z_{k,s} = \begin{bmatrix} \frac{\partial e(1)}{\partial w_0} & \frac{\partial e(1)}{\partial w_1} & \cdots & \frac{\partial e(1)}{\partial w_h} \\ \frac{\partial e(2)}{\partial w_0} & \frac{\partial e(2)}{\partial w_0} & \cdots & \frac{\partial e(2)}{\partial w_h} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial e(N_k)}{\partial w_0} & \frac{\partial e(N_k)}{\partial w_0} & \cdots & \frac{\partial e(N_k)}{\partial w_h} \end{bmatrix} \tag{3}$$

$$Z_{k,sr} = \begin{bmatrix} \frac{\partial e(1)}{\partial w_{0,0}} & \frac{\partial e(1)}{\partial w_{0,1}} & \cdots & \frac{\partial e(1)}{\partial w_{0,h}} & \frac{\partial e(1)}{\partial w_{1,0}} & \cdots & \frac{\partial e(1)}{\partial w_{n,h}} \\ \frac{\partial e(2)}{\partial w_{0,0}} & \frac{\partial e(2)}{\partial w_{0,1}} & \cdots & \frac{\partial e(2)}{\partial w_{0,h}} & \frac{\partial e(2)}{\partial w_{1,0}} & \cdots & \frac{\partial e(2)}{\partial w_{n,h}} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \cdots \\ \frac{\partial e(N_k)}{\partial w_{0,0}} & \frac{\partial e(N_k)}{\partial w_{0,1}} & \cdots & \frac{\partial e(2)}{\partial w_{0,h}} & \frac{\partial e(2)}{\partial w_{1,0}} & \cdots & \frac{\partial e(N_k)}{\partial w_{n,h}} \end{bmatrix} \tag{4}$$

The hidden layer to output layer weight partials can be computed roughly following the backpropagation derivation in Lecture 10, slides 20-23. Note that the activation function used in [4] and this implementation is the hyperbolic tangent function $\tanh(x)$. Also note that $\tanh'(x) = 1 - \tanh^2(x)$.

$$e(i; w_s) = y(i) - \tanh(n_s) \tag{5}$$

$$\frac{\partial e(i)}{\partial n_s} = (1 - \tanh^2(n_s)) \tag{6}$$

$$\frac{\partial e(i)}{\partial w_s} = (1 - \tanh^2(n_s))x_s \tag{7}$$

$$\tag{8}$$

The same derivation can be done for the input to hidden weights following Lecture 10 slides 24-26, yielding

$$\frac{\partial e(i)}{\partial n_s} = (1 - \tanh^2(n_s)) \left(\sum_{k \in Downstream(s)} \frac{\partial e(i)}{\partial n_k} w_{ks} \right) \tag{9}$$

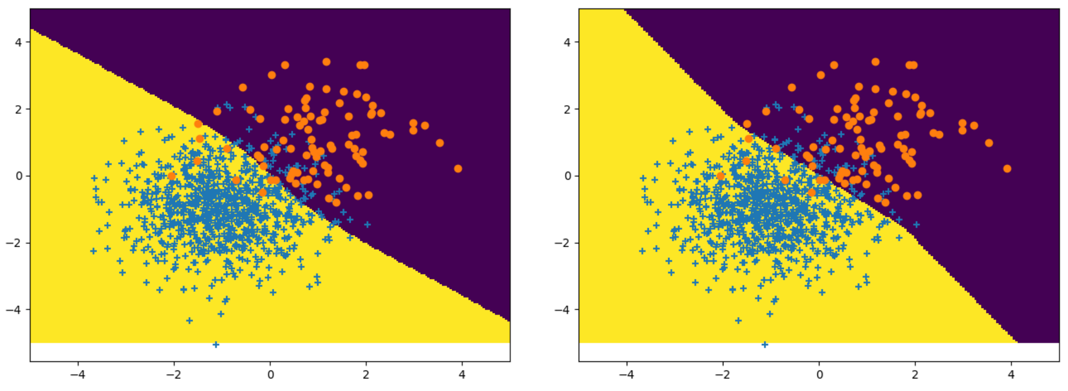
$$\frac{\partial e(i)}{\partial w_{sr}} = (1 - \tanh^2(n_s))x_{sr} \left(\sum_{k \in Downstream(s)} \frac{\partial e(i)}{\partial w_{ks}} \frac{w_{ks}}{x_{ks}} \right) \tag{10}$$

For the specific case of only a single output neuron, and a single hidden layer, this reduces to

$$\frac{\partial e(i)}{\partial w_{sr}} = (1 - \tanh^2(n_s))x_{sr} \frac{\partial e(i)}{\partial w_s} \frac{w_s}{x_s} \tag{11}$$

Hyperparameter λ

Because the hyperparameter λ is what makes CSMLP cost sensitive, a large subsection of [4] is dedicated to the analysis of the parameter and its optimal value. The authors concluded analytically that the best classification occurs when $\lambda = \frac{N_2}{N_1+N_2}$. This is the same as the intuitive value, implying that the best classification is one that weights the loss inversely proportional to the fraction of examples that the class takes up in the overall dataset. The following figure 6 show an example of the effect of λ on the decision boundary of a synthetic dataset.



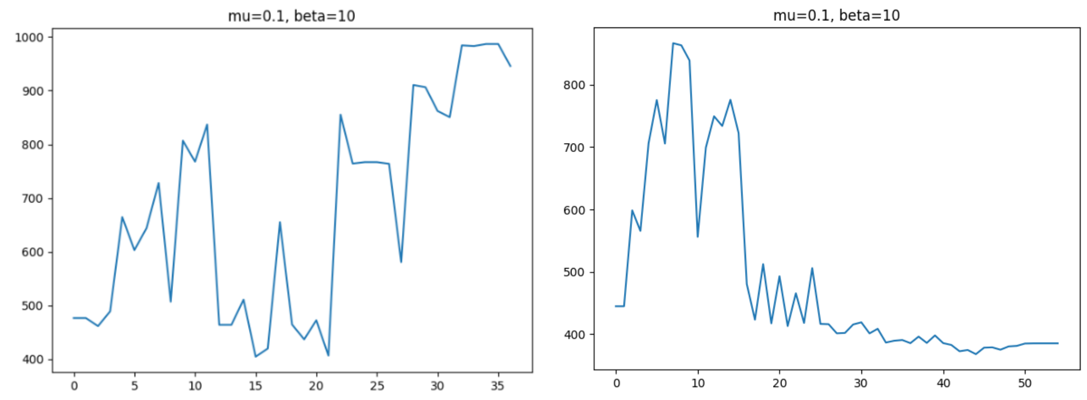
(a) $\lambda = 0.5$ (b) $\lambda = 0.05$

Figure 6: Effect of λ on the decision boundary of a synthetic dataset with 19:1 imbalance. Generated from normal distribution given positive class mean $\mu_1 = (-1, -1)$ and negative class mean $\mu_2 = (1, 1)$, with an identity covariance matrix.

Extension

Although the LM optimization algorithm converges to solutions quickly compared to gradient descent and Newton's method (since it is a combination of both), convergence to sub-optimal local minima is a known problem [1]. Prior to convergence, when μ is relatively large, the value of the loss is often very volatile. While this helps avoid convergence to the first visited minimum, it is often that the algorithm visits small loss values, begins to converge, and then departs and later converges at a less optimal minimum. This was particularly apparent during training for the flare dataset.

Figure 7 is a good example of a training run that visited many smaller minima but eventually converged to a very large loss value. This inconsistency in optimization leads to a long and tedious training process. During testing, it was found that in a 10 trial sample of identical trainings on the flare dataset, the original LM algorithm converged to the more-optimal minimum at a loss of approximately 400 only 1 out of 10 trials. The other 9 trials converged to a loss of approximately 850. The accuracies were collected for the 400 loss classifier versus the 850 loss classifier both trained on the same training subset and validated on the same validation subset. Comparing the two, the 850 loss classifier had worse accuracy of 0.491 compared to 0.749, worse weighted accuracy of 0.488 to compared to 0.752, and worse minority class accuracy 0.484 compared to 0.757. Similar behavior is also observed on the vehicle dataset.

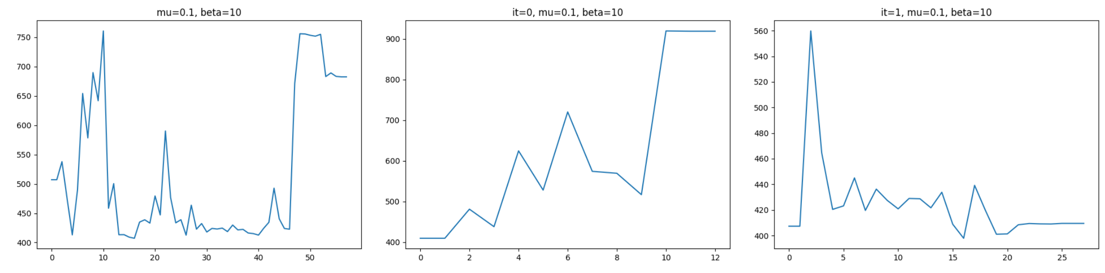


(a) Sub-optimal, volatile convergence (b) More optimal, stable convergence

Figure 7: Loss during CSMLP Levenberg Marquardt weight optimization, $\mu = 0.1, \beta = 10$

The proposed extension will be refered to as Repeat-CSMLP (R-CSMLP) which first tracks the minimum losses $L_{min} = \{L_1, L_2, .., L_n\}$, $n =$ maximum number of losses tracked, and their corresponding weights, $M_{sr} = \{W_{sr,1}, W_{sr,2}, ...W_{sr,n}\}$ and $M_s = \{W_{s,1}, W_{s,2}, ...W_{s,n}\}$. For our experiments, $n = 3$. After completing the initial optimization, for each $(L_i, W_{sr,i}, W_{s,i})$, the optimization is rerun with small $\mu = 0.001$ and small $\beta = 5$. Small μ, β are chosen to increase the chance of convergence around the new initialization minima. Although this process increases runtime on average up to 4 fold, in the long run it minimizes the number of training runs by a greater factor, assuming that the more optimal loss is desired. The theory is that instead of re-training initializing weights randomly over and over until the algorithm converges to the desired loss, the knowledge of M_{sr}, M_s found during the initial optimization allows the algorithm to initialize itself near potential minima, increasing its chance of convergence at smaller minima than the originally found minimum. And if the algorithm doesn't converge to one of these smaller minima, then it's likely that losses in L_{min} are bad minima that are a product of noise, meaning that they wouldn't generalize well on unseen examples.

After implementing this extension, it was found that in the same 10 trial scenario, the algorithm converged to the more optimal minimum 7 out of 10 times, with an average number of total iterations of roughly 200. Figure 8 shows a graph of the loss over time for the initial iteration and the subsequent iterations initialized around each $(L_i, W_{sr,i}, W_{s,i})$.



(a) Sub-optimal, first round convergence (b) Sub-optimal, second round convergence (c) More optimal, third round convergence

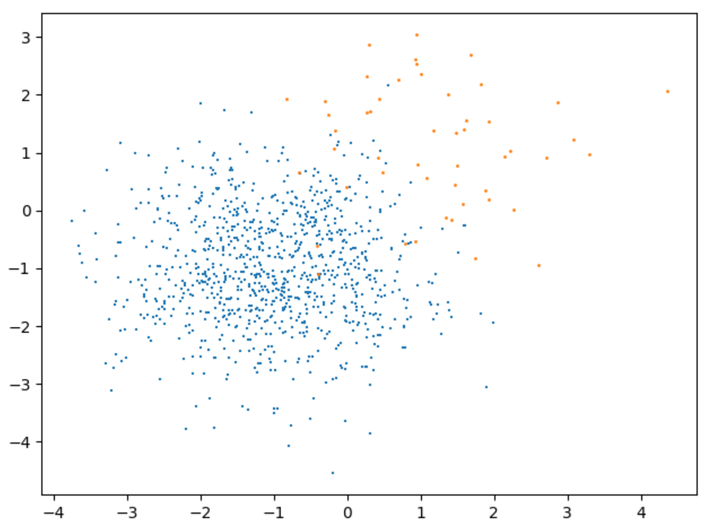
Figure 8: Loss during R-CSMLP Levenberg Marquardt weight optimization, $\mu = 0.1, \beta = 10$, repeat $\mu = 0.001, \beta = 3$

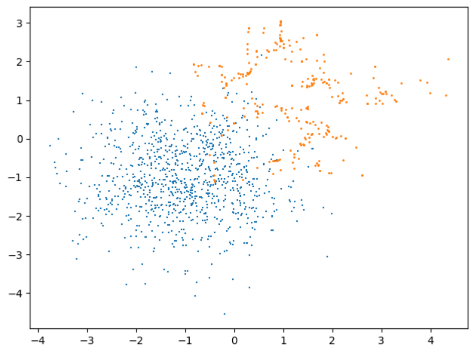
4.2.2 Synthetic Minority Oversampling Technique

Algorithm Overview
The algorithm is simple and sufficiently explained in 4.1 and the subsection of [5], Algorithm *SMOTE*, provides clear pseudo-code for the algorithm.

Extension
Although [10] states briefly why SMOTE may not work for MLP's our hypothesis about why, more specifically, is that in example sets with non-convex boundaries between classes, SMOTE's linear interpolation generates incorrect examples that will not generalize well. Although SMOTE attempts to generate points roughly in the neighborhood of each point in the minority set, in imbalanced scenarios, points may be very spread out, and newly generated points will often lie in the space that logically suits the majority class. The authors of [10] refer to this as "overlapping".

Noting this, we can see that instead of simply generating points in local regions, it might be more beneficial to focus on maintaining the grouping of points in the minority set. The proposed extension called K-SMOTE implements this by first performing a K-means clustering of the minority class, and then performing SMOTE within those clusters. The primary result of this method is that it removes the possibility that points will be interpolated between groups of the minority class, which will fit the original distribution of the points better. Figure 9 shows an example of a synthetic dataset where K-SMOTE generates more appropriate synthetic samples. It should be noted though that for uniform and roughly convex class boundaries, K-SMOTE acts very similarly to SMOTE. It was noticed after implementation that [3] cites a similar algorithm, cited as Jo and Japkowicz [2004] in which clusters samples and then performs oversampling. Although the authors do use k-means, they use random oversampling on each cluster as opposed to SMOTE.


(a) Original, imbalanced dataset, ratio 19:1


(b) Dataset after SMOTE

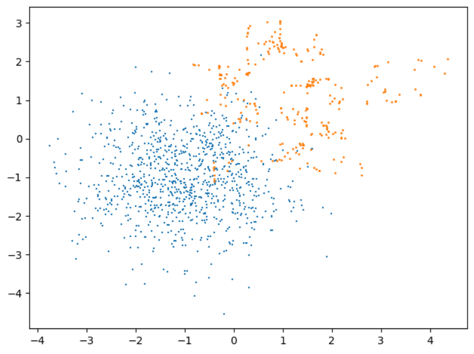

(c) Dataset after K-SMOTE

Figure 9: Synthetic datasets with SMOTE and K-SMOTE oversampling

4.3 Experiments and Results

Datasets and Tests Performed

The tests were performed over two different datasets. These are the UCI SolarFlareF and Vehicle1 datasets. These datasets were pre-processed such that nominal attributes were mapped to integer values, and all attributes were normalized from $[0,1]$. The MLP optimization was run with parameters $\mu = 0.1$ and $\beta = 10$. The number of hidden units for each dataset was determined experimentally and were $h = 7$ for both.

For the UCI datasets, k-fold stratified cross validation was used. The metrics used to evaluate the classifier performance were accuracy, weighted accuracy, and G-mean. The results are shown in Table 10.

For testing the effect of SMOTE and the K-SMOTE, the same stratified cross validation was performed for a base MLP with a SMOTE dataset, and a base MLP with a K-SMOTE dataset, where the validation sets were generated from the original, non-oversampled set.

Results

Metric	MLP	CSMLP	CSMLP Ext.	SMOTE	K-SMOTE
Accuracy	0.74	0.75	0.701	0.726	0.709
Weighted Accuracy	0.48	0.69	0.69	0.643	0.717
G-mean	0.14	0.55	0.69	0.631	0.717

Metric	MLP	CSMLP	CSMLP Ext.	SMOTE	K-SMOTE
Accuracy	0.798	0.741	0.710	0.779	0.788
Weighted Accuracy	0.595	0.666	0.689	0.791	0.778
G-mean	0.454	0.651	0.688	0.791	0.778

Figure 10: Performance metrics of algorithms on the SolarFlareF (top) and Vehicle1 (bottom) datasets

Paired t-tests were performed between algorithms for their weighted accuracies and G-mean values. Difference are considered "significant" for p-value < 0.05. On the vehicle dataset, the difference between CSMLP and MLP was not significant, but had a p-value of 0.09, and was significant for G-mean with p-value 0.009. The CSMLP extension showed no significant performance increase in terms of the best cross validation performed. Comparing SMOTE and the base classifier, SMOTE was shown to give a significant improvement on the vehicle dataset with weighted accuracy p-value 0.003 and G-mean 0.006 but was not significant on the flare dataset. K-SMOTE compared to the base MLP classifier showed significant improvement on both datasets, with weighted accuracy and G-mean p-values of 0.003 and 0.007 for vehicle and 0.002 and 0.003 for flare. Comparing SMOTE and K-SMOTE, K-SMOTE showed significant improvement over SMOTE on the flare dataset with weighted accuracy and G-mean p-values of 0.004 and 0.006. On the vehicle dataset only G-mean was shown to be of a significant difference with a 0.05 p-value. Lastly, comparing both SMOTE and K-SMOTE to CSMLP on both datasets did not yield statistically significant improvements.

4.4 Conclusion

The results of the experiments demonstrated that CSMLP significantly improves performance versus a standard MLP on a dataset with a more reasonable class imbalance, but does not on a dataset with a larger imbalance. Tests on more datasets would be necessary to confirm this conclusion.

It was also shown that R-CSMLP, while producing similar results in the long term, does significantly improve the probability of low-loss convergence of the CSMLP algorithm. Although R-CSMLP was originally formulated to reduce the number of training repetitions, it also has application in boosting, where the training of multiple sub-optimal classifiers can force the re-training of the much longer boosting algorithm. In section 6, the primary learning algorithm used in the RUSboost implementation was the base MLP from section 4. It was found that many training iterations were required in order to produce good results from the RUSboost classifier, since many of the MLP's during boosting were not converging to satisfactory minima. However, in theory, using R-CSMLP would produce more satisfactory results in fewer trainings of the boosting algorithm, considering that the probability of optimal convergence from R-CSMLP is so much higher than the base MLP.

Interestingly, contrary to [4], it was found that the SMOTE based oversampling methods did significantly improve performance on imbalanced datasets as compared to the base MLP. Specifically on the flare dataset, it's likely that the reason for this significance is not actually due to the improvement from SMOTE but rather on the poor performance of the base classifier. While the poor performance is expected in an imbalanced scenario, it is well known that MLPs are very sensitive to tuning of hyperparameters, and it is possible that a much better result could have been extracted if more time was spent finely tuning the hyperparameters.

Lastly, K-SMOTE did show significant improvement as compared to SMOTE on the flare dataset. This is likely due to the topology of the boundary between the classes in the datasets, indicating that SMOTE generates more incorrect border samples in vehicle than it does in flare.

5 Yuqi Hu

Algorithms for Imbalance Learning

Yuqi Hu
Department of Computer and Data Sciences
Case Western Reserve University
Cleveland, Ohio
yxh713@case.edu

Abstract—**This paper explains the machine learning algorithms that can be used to classify examples can achieve better results when the data is augmented.**

I. Introduction

To deal with the significant portions of machine learning problems with imbalance datasets, generating new examples using auto encoder is one of the important ways to create a balanced dataset that is easier to classify. Auto encoders can be used on different things. Auto encoders are normally used to compress the data, or do principal components analysis. They can also be trained to de-noise noisy images. The de-noising auto encoders will be implemented along with this paper to create synthetic examples that increase the weights of positive examples in the datasets. This paper first summarizes two ways to perform imbalance learning. One of the way is using auto encoder to create synthetic examples, which will be implemented and thoroughly experimented to see if it has signifiant advantages to increase the ability of normal classification algorithms to classify imbalanced problems compared to using these normal classification algorithms on the original datasets.

II. Paper Summaries

A. Synthetic Oversampling for Advanced Radioactive Threat Detection

The paper deals with a specific real-world application of using auto encoder to generate synthetic data to help to deal with significant skewed datasets. This paper deals with the problem of detecting radioactive threats using machine learning algorithms[1], which has a high value for the few examples in the data that are identified as positive threats. The dataset they used has 39,000 negative examples and only 39 positive examples[1]. Many parts of the paper are about discussing how to analysis radioactive threats, which are not related to the topic of this paper, so they will not be summarized here. To discuss their innovative data generation technique—Denoising Autoencoder-Based Minority Oversampling, the authors compared this algorithm with a well established algorithm—SMOTE: Synthetic Minority Over-Sampling Technique[1]. They find that Denoising Autoencoder-Based Minority Oversampling is significantly better than SMOTE when comparing AUC[1].

B. The PerfSim Algorithm for Concept Drift Detection in Imbalanced Data

The

III. Algorithms implemented

```
Algorithm 1 daego($\mathcal{X}$, $e$, $h$, $\sigma$, $\mathcal{X}_{init}$)
Input:
  i)   $\mathcal{X}$, the instances of the class training set.
  ii)  $e$, the number of training epochs.
  iii) $h$, the number of hidden units in the autoencoder.
  iv)  $\sigma$, the training noise.
  v)   $\mathcal{X}_{init}$, the sample initiation set.
Output:
  i)   $\mathcal{Y}$, the synthetic samples.
Method:
  1: Normalize the training dataset $\mathcal{X}$ into [$\mathcal{X}_{norm}$, $norm\_params$].
  2: Apply the normalization parameters $norm\_params$ to the sample initiation set $\mathcal{X}_{init}$.
  3: Create a denoising autoencoder network $dea$ from the parameter set $e$, $h$, $\sigma$.
  4: Train the network on the normalized training data $x\_norm$.
  5: Map $\mathcal{X}_{init}$ to the induced manifold via $dea$.
  6: Denormalize the mapped synthetic instances $\mathcal{Y}$ with the normalization parameters $norm\_params$.
  7: Return $\mathcal{Y}$
End Algorithm
```

Fig. 1 Implemented Algorithm[1]

IV. Experimentation

References

1. C. Bellinger, N. Japkowicz and C. Drummond, "Synthetic Oversampling for Advanced Radioactive Threat Detection," 2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA), Miami, FL, 2015, pp. 948-953, doi: 10.1109/ICMLA.2015.58.
2. D. K. Antwi, H. L. Viktor and N. Japkowicz, "The PerfSim Algorithm for Concept Drift Detection in Imbalanced Data," 2012 IEEE 12th International Conference on Data Mining Workshops, Brussels, 2012, pp. 619-628, doi: 10.1109/ICDMW.2012.122.

6 Cathy Tao - RUSBoost

6.1 Papers Read

RUSBoost: A Hybrid Approach to Alleviating Class Imbalance [13]:
This paper proposes the use of Random Under Sampling (RUS) in conjunction with AdaBoost in a hybrid sampling/boosting algorithm called RUSBoost for learning from imbalanced data. This algorithm is intended to be a simpler and faster alternative to SMOTEBoost, another hybrid sampling and boosting algorithm using SMOTE, or Synthetic Minority Oversampling Technique. The authors of the paper decided to approach the issue of class imbalance by using random undersampling in order to achieve a specific class ratio/distribution so that the class would no longer be imbalanced. In this case, several ratios were used: 0.35, 0.5, and 0.75 minority to majority class labels called the RUS ratio. This is combined with Adaboost, a boosting algorithm used to reduce bias in the data and create a strong classifier from several weak classifiers. The algorithm itself is described more in detail below. The paper found that RUSBoost performed similarly to SMOTEBoost using four primary metrics including Area Under the ROC Curve, Kolmogorov-Smirnov test value, Area under the Precision-Recall Curve, F-measure, and several other commonly used analysis metrics.

Improved PSO_AdaBoost Ensemble Algorithm for Imbalanced Data [?]:
Although the AdaBoost algorithm can be applied directly to the imbalanced data, the ensemble algorithm focuses more on misclassified samples, compared to samples of minority class. Based on the error calculation formula of the weak learner of AdaBoost, the error is only related to the weight and the number of misclassified samples. There is no additional processing for the misclassified samples of minority class, so the AdaBoost ensemble algorithm is not well suited for processing imbalanced data This paper proposes an ensemble algorithm, PSOPD-AdaBoost-A, that can reinitialize parameters to avoid falling into local optimum, and optimize the coefficients of AdaBoost weak classifiers. This algorithm uses Area Under Curve (AUC) as an indicator to reflect the comprehensive performance of the model and uses the AUC index in the error calculation formula of the weak classifier. This results in the improvement of the error calculation performance of the AdaBoost algorithm because it considers the effects of misclassification probability and Area under the curve, both useful metrics to emphasize areas of improvement for subsequent iterations and the overall unthresholded performance of the algorithm. PSO, or Particle Swarm Optimization, is an iteration based model where each solution has an adaptive value that represents the state of its own solution [?]. At each iteration, the solution adjusts its moving direction and velocity based on the global and the self found optimal solution.

6.2 Implemented Algorithms

6.2.1 RUSBoost

Both RUSBoost and SMOTEBoost were implemented and compared to each other in terms of amount of time spent classifying the dataset and its' metrics. Both RUSBoost and SMOTEBoost are compatible with three weak learners: Naive Bayes (My own code), Gaussian Naive Bayes (SciKit Learn) and Multi-Layer Perceptron (Tucker's project).

Algorithm Overview

RUSBoost is a hybrid sampling/boosting algorithm that combines data undersampling and boosting.

Data sampling techniques are used to help reduce the issue of class imbalance by modifying the class distribution of the training data set. In RUS, or random undersampling, examples in the majority class are randomly removed from the training data set until the class distribution reaches a target ratio.

Boosting is an ensemble meta-algorithm for primarily reducing bias and essentially creates an ensemble of weak learners in order to create a single strong learner. Boosting iteratively trains weak classifiers with respect to a distribution and adds them using weights in a overall final classifier. This weight is determined by the learners' accuracy and adjusted with every iteration. The misclassified data is then assigned a higher weight and the correctly classified data is assigned a lower weight. As a result, the proceeding weak learners will focus more on examples that were preciously misclassified.

Pseudo Code of Algorithm

```
RUSBoost
Given:    Set $S$ of examples $(x_1,y_1),...,(x_m,y_m)$ with minority class $y^T \in Y, |Y| = 2$
Weak learner, WeakLearn
Number of Iterations, T
Desired percentage of total instances to be represented by the minority class, N
    1. Initialize $D_1(i) = 1/m$ for all i.
    2. Do for t = 1, 2, ..., T.
        a. Create temporary training data set $S'_t$ with distribution $D'_t$ using random
           undersampling
        b. Call (WeakLearn), providing it with examples $S'_t$ and their weights $D'_t$
                        c. Get back a hypothesis $h_t$: X $\times Y \rightarrow [0,1]$.
        d. Calculate the pseudo-loss (for $S$ and $D_t$):
           $\varepsilon_t = \sum_{(i,y):y_t \neq y} D_t(i)(1 - h_t(x_i,y_i) + h_t(x_i,y))$.
        e. Calculate the weight update parameter:
           $\alpha_t = \frac{\varepsilon_t}{1-\varepsilon_t}$.
        f. Update $D_t$:
           $D_{t+1}(i) = D_t(i)\alpha_t^{\frac{1}{2}(1+h_t(x_i,y_i)+h_t(x_i,y:y\neq y_i))}$.
        g. Normalize $D_{t+1}$: Let $Z_t = \sum_i D_{t+1}(i)$.
           $D_{t+1}(i) = \frac{D_{t+1}(i)}{Z_t}$.
    3. Output the final hypothesis:
        $H(x) = \underset{y\ \varepsilon\ Y}{\text{argmax}} \sum_{t=1}^{T} h_t(x,y) log \frac{1}{\alpha_t}$
```

Extension

As an extension to the RUSBoost algorithm, I implemented a weighting scheme in the undersampling portion of the algorithm. The original RUSBoost algorithm used random undersampling following a uniform distribution. Rather than placing an equal weight on all values, there was an higher weight placed on

6.3 SMOTEBoost

Algorithm Overview
SMOTEBoost operates in the same manner as RUSBoost with the exception that it uses an oversampling technique. It iterates over every example in the minority class and randomly selects k nearest neighbors to that point. It will then randomly picks a point between the original point and the random neighbor to be a new synthetic sample of the minority class. It repeats this until the percent of oversampling desired is achieved.

6.4 Experiments and Results

6.4.1 Experiment Goals

The goal of this experiment is to determine the effects of combining sampling, a data pre-processing step, with boosting, a meta-algorithm designed to ensemble several models of a weak learner into a single strong learner. The output from the algorithm can be measured using several metrics in order to compare the performance of the original classifier, RUSBoost, the extension created for RUSBoost and SMOTEBoost.

6.4.2 Experimental Methodology

Three classifiers were used for the base learner. A simple Naive Bayes algorithm that I programmed. The Gaussian Naive Bayes classifier in the scikit-learn library, and a Multi-layer Perceptron algorithm written by Tucker.

Each algorithm was run by itself to get a baseline for metrics. Each algorithm was then used as the weak learner in the RUSBoost algorithm, RUSBoost extension, and SMOTEBoost algorithm using the following parameters as seen in section 6.4.3.

Several metrics were used such as: Accuracy, F1 score/Weighted Accuracy, Precision, Recall, F-measure, Area under the Receiver Operating Characteristic curve, and Area under the Precision-Recall Curve.

6.4.3 Parameters

Number of Iterations	10
Minority/RUS Ratio	0.5
Beta	1

6.4.4 Comparison of Algorithms

Weak Learner: Naive Bayes

Vehicle Dataset

	Naive Bayes	RUSBoost	RUSBoost Ext.	SMOTEBoost
Accuracy	0.650	0.706	0.696	0.746
F1 Score	0.762	0.773	0.763	0.811
Precision	0.401	0.439	0.430	0.480
Recall	0.975	0.900	0.890	0.935
F-measure	0.568	0.590	0.580	0.634
A-ROC	0.767	0.773	0.763	0.811
A-PRC	0.677	0.681	0.673	0.715

Flare Dataset

	Naive Bayes	RUSBoost	RUSBoost Ext.	SMOTEBoost
Accuracy	0.814	0.313	0.313	0.814
F1 Score	0.500	0.500	0.500	0.500
Precision	1.000	0.350	0.350	1.000
Recall	0.000	0.800	0.800	0.000
F-measure	0.000	0.252	0.252	0.000
A-ROC	0.500	0.500	0.500	0.500
A-PRC	0.593	0.593	0.593	0.593

Weak Learner: Gaussian Naive Bayes

Vehicle Dataset

	Naive Bayes	RUSBoost	RUSBoost Ext.	SMOTEBoost
Accuracy	0.656	0.726	0.727	0.696
F1 Score	0.732	0.789	0.790	0.777
Precision	0.396	0.480	0.461	0.433
Recall	0.875	0.909	0.910	0.930
F-measure	0.545	0.620	0.610	0.590
A-ROC	0.732	0.789	0.790	0.777
A-PRC	0.650	0.705	0.696	0.7689

Flare Dataset

	Naive Bayes	RUSBoost	RUSBoost Ext.	SMOTEBoost
Accuracy	0.803	0.560	0.581	0.609
F1 Score	0.626	0.520	0.541	0.629
Precision	0.464	0.307	0.283	0.251
Recall	0.344	0.458	0.478	0.661
F-measure	0.391	0.280	0.284	0.360
A-ROC	0.626	0.520	0.541	0.629
A-PRC	0.465	0.433	0.429	0.488

Weak Learner: Multi-Layer Perceptron

Vehicle Dataset

	MLP	RUSBoost	RUSBoost Ext.	SMOTEBoost
Accuracy	0.747	0.731	0.747	0.766
F1 Score	0.588	0.540	0.551	0.504
Precision	0.596	0.655	0.734	0.933
Recall	0.287	0.180	0.179	0.010
F-measure	0.278	0.133	0.122	0.019
A-ROC	0.588	0.540	0.551	0.504
A-PRC	0.526	0.514	0.554	0.588

Flare Dataset

	Naive Bayes	RUSBoost	RUSBoost Ext.	SMOTEBoost
Accuracy	0.814	0.502	0.587	0.785
F1 Score	0.500	0.526	0.522	0.490
Precision	1.000	0.521	0.420	0.444
Recall	0.000	0.565	0.418	0.019
F-measure	0.000	0.199	0.190	0.227
A-ROC	0.500	0.526	0.522	0.490
A-PRC	0.593	0.583	0.473	0.323

When using Naive Bayes as a weak learner, RUSBoost was able to increase the accuracy on the vehicle dataset by around 5% and both AROC and APRC by around 1%. This improvement was much more significant when using the Gaussian Naive Bayes algorithm as the weak learner, with over a 5% increase over most metrics. On the other hand, the statistics for MLP lowered when using RUSBoost which may be attributed to the fact that it is a neural network and not designed to be used in an ensemble nor would benefit from being included in one. Metrics for the MLP RUSBoost algorithm tended to vary the most so the data written above is one that was more in the middle of all the outputted data.

In terms of the Flare dataset, the algorithm performed relatively poorly. Both the Naive Bayes and and MLP algorithms had extremely low recall score and a very high precision indicating that the algorithm was most likely classifying all or most of the data as the minority class label rather than correctly identifying all of the examples. In this dataset, RUSBoost significantly decreased almost all of the metrics including accuracy, precision, AROC and APRC.

The extension appeared to have performed relatively similar to the original RUSBoost algorithm and discrepancies most likely are caused by the randomly generated folds and the random undersampling. In this case, more tests may need to be performed to test for significance in difference using multiple sets of data and using different parameters. A comparison of RUSBoost and SMOTEBoost show that performance is very similar. Overall, SMOTEBoost performed slightly better which can be expected because there is no loss of data unlike RUSBoost. The only exception appears to be when MLP is used as the weak learner. As already established, MLP does not perform well as a weak learner in an ensemble so it is difficult to determine whether it was caused by the specific models used or the algorithm itself.

6.5 Conclusion

Overall, when comparing the metrics between the base learner, RUSBoost, the RUSBoost extension and SMOTEBoost, the hybrid algorithms generally performed better. Because these algorithms integrate sampling to achieve a more balanced dataset, the classifier has a better distribution of class labels to work with. In addition, combining sampling with boosting allows for some of the bias inherent in sampling and inbalanced data to be removed. Changing the weightings of the random sampling did not affect the performance at all. In some cases, it decreased some of the metrics and in some, there was a slight improvement. However, there is not enough to be able to definitively determine if changing the weighting had a significant effect or not. Comparing RUSBoost and SMOTEBoost showed that the two algorithms yielded similar results. However, because RUSBoost processes much fewer data than SMOTEBoost due to undersampling rather than oversampling, it is a competitive alternative, especially when deciding which algorithm to use on a much larger dataset. This difference in performance on datasets can most likely be attributed to the way the data was processed and the type of data contained in the dataset. While the vehicle dataset contained all continuous examples, the flare dataset had a combination of nominal and discrete features. The Naive Bayes algorithm used a bin system which may have caused an incorrect distribution of values and errors in multiplying probabilities during the classification portion.

7 Mei Wong - Hellinger Distance Decision Trees

7.1 Papers Read

Hellinger Distance Decision Tree (HDDT) The first paper [6] proposes a new type of decision tree called Hellinger Distance Decision Trees (HDDT), which calculates and utilizes Hellinger distance to split nodes. The Hellinger distance of an attribute is calculated by taking the square root difference of the true positive and false positive rate. The authors analyzed the advantages of Hellinger distance over more popular splitting criterions, such as information gain. They demonstrated that even without sampling methods, HDDT with bagging performed significantly better on imbalanced data due to its strong skew insensitivity.

Class Confidence Proportion Decision Tree (CCPDT) The second paper [11] is related to HDDT, and implements another decision tree algorithm called Class Confidence Proportion Decision Tree (CCPDT), which is also insensitive to imbalanced data. Traditional decision trees, such as information gain finds the attribute that is more predictive of the class label, and as a result it is biased to the majority class. On the other hand, this new decision tree uses a new concept called Class Confidence (CC) to find the most interesting attribute from all the classes. While traditional confidence calculates the precision, CC calculates true positives and true negatives over actual positives and actual negatives, respectively. In order words, CC finds how many actual positive and negatives are predicted correctly. Then, the proportion of one CC for one class over the CC of all the classes is used to calculate Class Confidence Proportion (CCP).

The difference between HDDT and CCPDT is that Hellinger distance takes the square root distances of the true positive and false positive rates while CCP takes the proportion of true positive and true negatives rates. The experiments did not confirm that CCPDT is significantly better than HDDT. However, the combination of both CCP and Hellinger distance is statistically better than using either algorithm individually. The combination works by choosing the attribute with the highest CCP by default, and then choosing the attribute with the highest Hellinger distance to break ties.

robROSE The third paper [2] presents a robust version of ROSE called robROSE. The original ROSE (Random Over-Sampling Examples) algorithm works by selecting (x_i, y_i), an example from a set of training examples, and generating new examples in its neighborhood from $K_{H_j}(x_i)$, where K_{H_j} is a probability distribution centered at x_i and H_j is a covariance matrix. ROSE works well with imbalanced data. However, a weakness of ROSE is that it can create artificial examples using outliers. The new algorithm robROSE only creates artificial examples using clean examples. Let X_1 and X_0 represent the examples belonging to the minority and majority class, respectively. The algorithm uses calculations of the robust covariance $\sum_1$ and robust center μ_1 of X_1 in order to find the robust Mahalanobis distance for $X1$. The Mahalanobis distance is used to identify the index of a non-outlying minority example. Artificial examples are generated until balance is reached. The authors found that robROSE had slightly better performance than ROSE by comparing area under precision recall curve results.

7.2 Implemented Algorithms

The first algorithm implemented is Hellinger Distance Decision Tree because HDDT is more complex and has more room for analysis than robROSE. Since the paper on HDDT focused on unpruned decision trees, the extension implements a HDDT algorithm with pruning.

7.3 Hellinger Distance Decision Tree

7.3.1 Algorithm Overview

Hellinger distance is a measure of distributional divergence. Let P and Q represent two continuous distributions with respect to the paramters λ. Then the Hellinger distance is:

$$d_H(P,Q) = \sqrt{\int (\sqrt{P} - \sqrt{Q})^2 d\lambda} \tag{12}$$

In the above equation, P and Q are the frequency of an attribute across the two classes, and from here we can define the "affinity" between the two probabilities. If P equals Q, then the distance is 0, which is the maximum affinity. If either P or Q is 0, then the distance is $\sqrt{2}$, which is zero affinity. Therefore, Hellinger distance can be used to calculate how good an attribute is at separating the class distribution.

To apply Hellinger distance as a decision tree criterion, we count the frequencies over all subsets of our training set. Since our data sets have two classes (0 and 1), let $X_{y=0}$ represent a subset of examples of class 0 and let $X_{y=1}$ represent a subset of examples of class 1. Let $X_{k=v,y=0}$ represent a subset of examples with value v for feature k. Let $X_{k=w,y=0}$ represent a subset of examples that do not have value j for feature k. The Hellinger distance for feature f with value j is:

$$d_h(f,j) = \sqrt{(\sqrt{\frac{|X_{f=v,y=0}|}{|X_{y=0}|}} - \sqrt{\frac{|X_{f=v,y=1}|}{|X_{y=1}|}})^2 + (\sqrt{\frac{|X_{f=w,y=0}|}{|X_{y=0}|}} - \sqrt{\frac{|X_{f=w,y=1}|}{|X_{y=1}|}})^2} \tag{13}$$

In the above equation, Hellinger distance will create a binary split for nominal attributes because Hellinger distance performs better on attributes with lots of branches. If an attribute is continuous, then we would sort the attribute by its value, calculate the binary Hellinger distance at each split, and return the value with the highest Hellinger distance.

7.3.2 Extension

In Cieslak's paper on HDDTs, they created unpruned decision trees with Laplace smoothing. For the research extension, HDDT is combined with pruning. First, half of the examples are held aside as the validation set, and the HDDT is built normally on the rest of the examples, which act as the training set. Then the greedy post pruning algorithm is ran. Each node is traversed using a breadth first search algorithm and a FIFO queue. Every time a new node is explored, a new decision tree is reconstructed without that node, and the decision tree is valuated on the validation set. The node with the best performance is returned.

For regular data sets, where the both classes have equal or close to equal distribution, pruning works well at reducing the complexity of the decision tree. This allows the decision tree to be able to generalize well, leading to better accuracy. Before implementing pruning with HDDT, the hypothesis is that pruning will worsen the performance of the original algorithm. This is because pruning deletes a node and all its child nodes. For imbalanced data sets, this procedure can delete a large portion of the minority class subtree. Additionally, it is possible that pruning will have no affect on the decision tree because each node that get deleted creates a new decision tree that performs worse than the original tree. As a result, the algorithm will just return the original decision tree.

7.4 Implementation

The algorithms were implemented in Python using the numpy, math, and random libraries. The numpy library was used for handling data. The math library was used to perform mathematical calculations. The random library was used to shuffle examples for stratified cross validation. The SolarFlareF and Vehicle1 data sets from the University of California Irvine was used for testing.

7.5 Experiments and Results

Experiments The HDDT algorithm and HDDT with pruning algorithm is compared with the a decision tree algorithm using information gain, which is probably the most popular splitting criterion for decision trees. All of the algorithms are trained separately by the flare and vehicle dataset. The results are shown in the table below.

	Flare		
Metric	HDDT	HDDT&Prune	InfoGain
Average	99.1	99.1	81.3
Precision	100	100	100
Recall	99.2	99.2	80.3

Table 1: Weighted Accuracy of each algorithm trained by Flare Dataset

	Vehicle		
Metric	HDDT	HDDT&Prune	InfoGain
Average	76.4	76.4	71.4
Precision	100	100	100
Recall	75.4	75.4	70.4

Table 2: Weighted Accuracy of each algorithm trained by Vehicle Dataset

Results HDDT and HDDT with pruning performed better than information gain. This is as expected because HDDT is proven to be insensitive to imbalanced data while information gain is skewed towards the majority class. Furthermore, HDDT and HDDT demonstrated no difference in performance, which is also as expected. This matches with the hypothesis stated in section 7.3.2. Interestingly, the algorithms performed better on the flare data set than the vehicle data set even though the flare data set is more imbalanced than the vehicle data set. This could be because the algorithms performed better on nominal attributes. The flare data set contains only nominal attributes while the vehicle data set only contains continuous attributes. Overall, HDDT performs better than information gain on imbalanced data sets.

8 Bringing it all Together

After reviewing a host of different approaches to handling imbalanced data, two distinct methodologies emerged. The first type of algorithms were dedicated to prepossessing the data and then teaching an algorithm independently with the new data. These include ADASYN, DSUS, RUSBoost, and SMOTE. The second type of algorithms were machine learning algorithms in and of themselves that had special modifications to handle imbalanced data. These include the Cost Sensitive Multilayer Perceptron and Hellinger Distance Decision Trees. In order to properly compare the results of different algorithms, it is important that the same types of algorithms are used. Therefore, we will first analyze all preprocessing algorithms, and then all machine learning algorithms.

8.1 Preprocessing Algorithms

All preprocessing algorithms encountered in our research fall under one of the following categories: majority data undersampling, minority data oversampling, or a combination of the two. ADASYN and SMOTE are examples of minority data oversampling methods, whereas RUS and DSUS are algorithms that focus on undersampling majority data.

An important factor to consider while comparing these algorithms is which machine learning algorithm was used to test the datasets. For each one, the best performing algorithm was chosen for comparsion. The selected testing machine learning algorithms are as follows:

ADASYN - Decision Tree

DSUS - Multi-Layer Perceptron

RUSBoost - Multi-Layer Perceptron

SMOTE - Multi-Layer Perceptron

8.1.1 Similarities

As ADASYN and SMOTE are both minority data oversampling methods, they are naturally very similar. In fact, ADASYN was developed as the next step for SMOTE by focusing on creating more synthetic examples based on K-nearest neighbors for difficult examples, rather than for random examples as is done in SMOTE.

By the same principle, RUS and DSUS are similar majority data undersampling methods. DSUS seeks to improve upon RUS by favoring data based on sensitivity and K-nearest neighbors of other data points, whereas RUS picks data points randomly.

Lastly, the Bidirectional ADASYN algorithm and RUSBoost share the dual oversampling and undersampling property, although done in significantly different ways.

8.1.2 Differences

When contrasting preprocessing algorithms, the greatest apparent difference is the mathematical basis behind each algorithm. The simplest imbalanced data algorithm is RUS, which randomly picks points from the majority data to be excluded. SMOTE similarly picks random points in the minority data and creates synthetic examples between them and their K-nearest neighbors. ADASYN is the next level of complexity as it focuses on difficult to learn examples, as DSUS does with sensitivity. Lastly, Boosted algorithms such as RUSBoost tend to be more complex as they add a weight to all examples of whichever algorithm is being boosted.

8.1.3 Weighted Accuracy Comparison

Running the original and modified versions of ADASYN, DSUS, and SMOTE on the shared Vehicle1 and SolarFlareF datasets produces the following results:

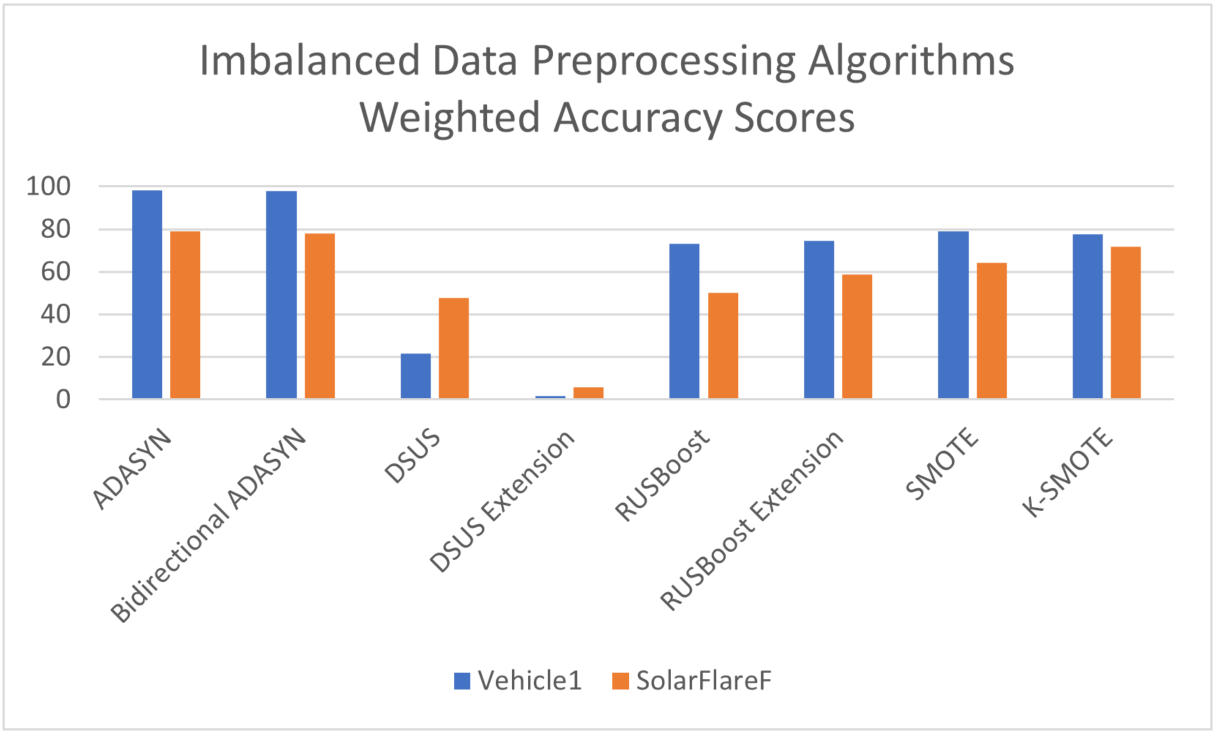


Figure 11: Weighted accuracy scores of original and modified versions of ADASYN, DSUS, and SMOTE algorithms.

Based on these results, it is evident that ADASYN and SMOTE produce the most balanced datasets for training. As ADASYN is meant to be an improvement on SMOTE, it is naturally expected that its results perform better, as the data demonstrates. It should be noted, however, that ADASYN was tested on a Decision Tree rather than a Multi-Layer Perceptron algorithm like the remaining values. The RUSBoost algorithm had mixed performance, performing similar to SMOTE on the Vehicle1 dataset but consistently worse on the SolarFlareF dataset. A likely explanation is that the Vehicle1 dataset uses continuous data, which is easier to boost without signficiantly altering the values. The DSUS algorithm performed relatively poorly, which may also be attributed to the datasets for different reasons. As DSUS is a majority data undersampling method, running it on highly diverse data where all examples are crucial to the learning algorithm will lead to naturally poor results. This is also supported by the fact that Bidirectional ADASYN performed slightly worse than the purely oversampling ADASYN approach.

8.2 Machine Learning Algorithms

As only two machine learning algorithms were chosen for implementation, they cannot be used to generalize about how all machine learning algorithms handle imbalanced datasets. However, by comparing and contrasting them, they may still serve as case studies that a basic understanding of how Decision Trees and Neural Networks can handle imbalanced data can be obtained.

8.2.1 Similarities

Both the Cost Sensitive Multilayer Perceptron and Hellinger Distance Decision Tree focus on weighing the data differently based on if it belongs to a majority or minority class. The Cost Sensitive Multilayer Perceptron can directly modify the weights and costs of each Perceptron in the network to accomodate for this. While the Hellinger Distance Decision Tree algorithm lacks the ability to add weights to tree nodes, it can still mimic this behavior by modifying its node splitting algorithm to take the square root difference of the true and false positive rate. As a result, both algorithms are primed to learn based on weighted accuracy rather than general accuracy.

8.2.2 Differences

As the algorithms are based on completely different machine learning techniques, they are naturally substantially different from one another. However, further details of their differences can be explored by taking a closer look at how they operate. Most notably, the Hellinger Distance Decision Tree updates its node selection effectively as a data pre-processing method, whereas the The Cost Sensitive Multilayer Perceptron uses cost updates as a prediction post-processing method. The significance of the two methods is discussed in more detail in the Introduction section.

8.2.3 Weighted Accuracy Comparison

Running the original and modified versions of the Cost Sensitive Multilayer Perceptron and Hellinger Distance Decision Tree on the shared Vehicle1 and SolarFlareF datasets produces the following results:

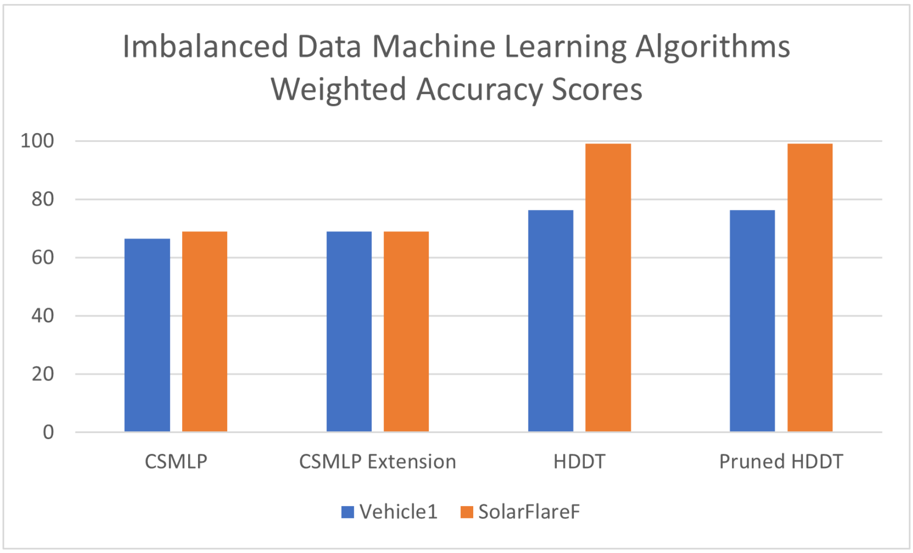


Figure 12: Weighted accuracy scores of original and modified versions of Cost Sensitive Multilayer Perceptron and Hellinger Distance Decision Tree.

From the results, it is clear that both algorithms had satisfactory performance (above 50%) on the weighted accuracy of the Vehicle1 dataset, with the Hellinger Distance Decision Tree performing slightly better. However, on the SolarFlareF dataset, while the Cost Sensitive Multilayer Perceptron remained consistent, whereas the Hellinger Distance Decision Tree performed exceedingly well, with a weighted accuracy score of over 99%. This discrepancy can be explained primarily by the format of the two datasets. Namely, the Vehicle1 dataset consisted of continous data values, whereas for the SolarFlareF dataset all values were treated as effectively nominal. Given that the Hellinger Distance Decision Tree is designed primarily for splits along clear-cut data, it is no surprise that it performed almost perfectly on the nominal data both with and without pruning.

9 Conclusion

Imbalanced data is one of the biggest challenges to traditional machine learning algorithms. Over time, several approaches have been tried to rectify this. Some algorithms focus on pre-processing the data through a combination of majority class undersampling and minority class undersampling in order to simplify the learning process. Others use prediction post-processing to modify the threshold and cost-sensitivity in order to favor minority examples over majority examples. Both types of algorithms are active research areas that data scientists continue to make progress on. As a result, most of the algorithms we have sampled can be traced to be improvements and combinations of one another, serving as building blocks for future algorithms. We have successfully added our own modifications to a handful of these algorithms, in the aim of contributing to the ever-improving study of imbalanced machine learning.

References

[1] N. Ampazis and S. J. Perantonis. Levenberg-marquardt algorithm with adaptive momentum for the efficient training of feedforward networks. In *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*, volume 1, pages 126–131 vol.1, 2000.

[2] Bart Baesens, Sebastian Hoppner, Irene Ortner, and Tim Verdonck. robrose: A robust approach for dealing with imbalanced data in fraud detection. 2020.

[3] Paula Branco, Luís Torgo, and Rita P. Ribeiro. A survey of predictive modeling on imbalanced domains. *ACM Comput. Surv.*, 49(2), August 2016.

[4] C. L. Castro and A. P. Braga. Novel cost-sensitive approach to improve the multilayer perceptron performance on imbalanced data. *IEEE Transactions on Neural Networks and Learning Systems*, 24(6):888–899, 2013.

[5] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. Smote: Synthetic minority oversampling technique. *Journal of Artificial Intelligence Research*, 16:321–357, Jun 2002.

[6] David A. Cieslak, T. Ryan Hoens, Nitesh V. Chawla, and W. Philip Kegelmeyer. Hellinger distance decision trees are robust and skew-insensitive. *Data Min. Knowl. Discov.*, 24(1):136–158, 2012.

[7] Haibo He, Yang Bai, E. A. Garcia, and Shutao Li. Adasyn: Adaptive synthetic sampling approach for imbalanced learning. pages 1322–1328, 2008.

[8] Y. E. Kurniawati, A. E. Permanasari, and S. Fauziati. Adaptive synthetic-nominal (adasyn-n) and adaptive synthetic-knn (adasyn-knn) for multiclass imbalance learning on laboratory test data. pages 1–6, 2018.

[9] Kuan Li, Xiangfei Kong, Zhi Lu, Liu Wenyin, and Jianping Yin. Boosting weighted elm for imbalanced learning. *Neurocomputing*, 128:15 – 21, 2014.

[10] M. Lin, K. Tang, and X. Yao. Dynamic sampling approach to training neural networks for multiclass imbalance classification. *IEEE Transactions on Neural Networks and Learning Systems*, 24(4):647–660, 2013.

[11] Wei Liu, Sanjay Chawla, David A. Cieslak, and Nitesh V. Chawla. pages 766–777, April 2010.

[12] W. W. Y. Ng, J. Hu, D. S. Yeung, S. Yin, and F. Roli. Diversified sensitivity-based undersampling for imbalance classification problems. *IEEE Transactions on Cybernetics*, 45(11):2402–2412, 2015.

[13] C. Seiffert, T. M. Khoshgoftaar, J. V. Hulse, and A. Napolitano. Rusboost: A hybrid approach to alleviating class imbalance. pages 185–197, 2010.

[14] Stavrianos Skalidis. Adaptive synthetic sampling approach for imbalanced learning.

[15] S. Wang, W. Liu, J. Wu, L. Cao, Q. Meng, and P. J. Kennedy. Training deep neural networks on imbalanced data sets. In *2016 International Joint Conference on Neural Networks (IJCNN)*, pages 4368–4374, 2016.