ACTIVE CLASS SELECTION FOR REGRESSION

A Project Report

submitted by

PRUTHVI RAJ R G

in partial fulfilment of the requirements

for the award of the degree of

BACHELOR OF TECHNOLOGY

&

MASTER OF TECHNOLOGY

DEPARTMENT OF DATA SCIENCE & ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

JUNE 2022

THESIS CERTIFICATE

This is to certify that the thesis titled **ACTIVE CLASS SELECTION FOR REGRESSION**, submitted by **PRUTHVI RAJ R G**, to the Indian Institute of Technology, Madras, for the award of the degree of **BACHELOR OF TECHNOLOGY & MASTER OF TECHNOLOGY**, is a bona fide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Prof. Nandan Sudarsanam
Project Guide
Professor
Dept. of Management Studies
IIT Madras, 600 036

Place: Chennai

Date:

ACKNOWLEDGEMENTS

I want to thank my guide - Prof. Nandan Sudarsanam, for his constant guidance at every stage of the project. I want to thank my parents and family, who took care of me throughout my stay at home during the pandemic. I am also very thankful to my friends in the "Aalaporan Thamizhan" group for their motivation and support.

ABSTRACT

KEYWORDS: Machine Learning; Active Learning; Active Class Selection; Regression

In machine learning, Active Class Selection (ACS) algorithms aim to actively select a class and ask the oracle to provide an instance for that class to optimize a classifier's performance while minimizing the number of requests. In this project, we attempt to develop Active Class Selection algorithms for the case of regression where there exist no distinction of classes for samples. Our experimental evaluations show that algorithms directly inspired by existing ACS algorithms fail to perform better than the random sampling baseline. We propose a new algorithm, "Input Space Disparity," that we show outperforms the baselines on a logical subset of datasets in early iterations. It effectively prefers sampling difficult y-values, thereby improving the regression performance.

TABLE OF CONTENTS

LIST OF FIGURES

ABBREVIATIONS

ML	Machine Learning
AL	Active Learning
DL	Deep Learning
PAL	Probabilistic Active Learning
ACS	Active Class Selection
IID	Independent and Identically Distributed
WIIACS	Weighted Informative Inverse ACS
WIACS	Weighted Inverse ACS
ISD	Input Space Disparity
GMM	Gaussian Mixture Model
BCI	Brain Computer Interface

CHAPTER 1

INTRODUCTION

Classification techniques in medical research, economy, or neurobiology require human labeling effort during the training phase of the model. As this is usually very expensive and time-consuming, the field of active learning (AL) appeared. Here, the algorithm aims to actively select only the most informative samples from an extensive pool of unlabeled data and sequentially ask for their label. This way, better classification performance is reached with fewer training instances than passively feeding arbitrary instances to the classifier.

This report addresses a related field called active class selection (ACS), specifically in the context of regression. Instead of arbitrarily selecting an unlabeled instance and acquiring its label, ACS methods request an unseen sample by specifying its class. However, in the case of ACS, less information is available to determine what is valuable for training, e.g., unlabeled samples to approximate the data distribution are missing.

This section elaborates on machine learning(ML), active learning(AL), and active class selection. The rest of the article is structured as follows. In Sec. 2, we discuss the literature on active class selection, followed by implementing a few baselines. We then modify existing ACS algorithms to make them suitable for regression datasets using the 'binning' trick. In the last section, we propose a new algorithm, Input Space Disparity. We provide a pseudo code and performance on various datasets for each algorithm.

1.1 Machine Learning

Machine learning (ML) is a field of exploration dedicated to learning and building methods that 'learn,' that is, methods that leverage data to enhance performance on some tasks. ML is a part of Artificial Intelligence(AI). Machine learning algorithms build a model using training data. They make predictions without being explicitly programmed to do so. ML algorithms are used in various applications, such as email filtering, computer vision, medicine, speech recognition, and fields where it is challenging to develop conventional algorithms.

Model training in machine language is feeding an ML algorithm with data to help identify and learn good values for all attributes involved. There are several types of machine learning models, of which the most common ones are supervised and unsupervised learning. Supervised learning is possible when the training data contains input and output values. Each data set with the inputs and the expected output is called a supervisory signal. The training is done based on the deviation of the processed result from the documented result when the inputs are fed into the model. Unsupervised learning involves determining patterns in the data. Additional data is then used to fit patterns or clusters. It is an iterative process that improves accuracy based on the correlation to the expected patterns or clusters. There is no reference output dataset in this method.

There are seven steps involved in building ML models.

Describing The Problem : It the foremost step toward determining what an ML model should perform. This effort also enables identifying the right inputs and their outputs.

Data Collection : After describing the problem statement, it is necessary to investigate and collect the data needed to feed the ML model. Data Collection is a critical stage in creating an ML model because the quantity and quality of the data used will decide how effective the model will be. Data can be collected from existing databases or can be created from scratch.

Conditioning The Data : The data preparation stage is when data is cleaned and structured as required to prepare it for training the ML model. Data preparation is the stage where the appropriate features of data are selected. This stage can have an immediate effect on the execution time and results. This is also at the stage where data is segregated into two groups – one for training the ML model and the other for evaluating the model's performance. Pre-processing data by standardizing and eradicating duplicates is also carried out at this stage.

Selecting Appropriate Model : Picking and designing a model must be done according to the ML model's objective. Several models, like linear regression, k-means, and bayesian methods, are to pick from. The type of data usually determines the choice of models. For instance, image processing convolutional neural networks would be the ideal pick, and k-means would work best for segmentation.

The Model Training : ML algorithm is trained by feeding samples from the datasets. It is the phase where the understanding takes place. Proper training can immensely improve the prediction rate of the ML model. The weights of the model are usually initialized randomly.

Evaluating the model : The ML model must be evaluated against the "validation dataset." It is necessary to assess the accuracy of the model. Determinating the measures of success based on what the model is intended to achieve is crucial for explaining correlation.

Parameter Tuning : Picking the correct set of parameters used in the ML model is key to attaining good accuracy. The set of parameters that are picked based on their impact on the model architecture are called hyperparameters. The technique of identifying the hyperparameters by adjusting the model is called parameter tuning.

1.2 Active Learning

The fundamental concept behind active learning is that ML models can achieve greater accuracy with fewer training labels if they can carefully choose the data from which it learns. An AL agent may pose queries, often in the form of unlabeled data samples to be hand-labeled by an oracle. AL works well in most ML problems, where unlabeled data may be abundant or easily obtained, but labels are difficult, time-consuming, or costly to obtain.

There are several strategies in which AL agents may pose queries, and several different query strategies have attempted to decide which samples are most informative. In this section, we present two such strategies, the pool-based active learning setting) using an uncertainty sampling query strategy and the query-by-committee algorithm.

1.2.1 Uncertainty sampling

Extensive collections of unlabeled data can be gathered for many real-world learning problems at once. This motivates a pool-based sampling strategy, which assumes a small set of labeled data L and a large pool of unlabeled data U available. Queries are selectively pulled from the pool. Typically, instances are queried greedily, according to an informativeness score used to evaluate all instances in the pool.

The most straightforward and most commonly used query framework is uncertainty sampling. In this framework, an AL agent queries the samples that are least certain how to label. This approach is often straightforward for probabilistic learning models. For example, when using a probabilistic model for binary classification, uncertainty sampling queries the instance whose posterior probability of belonging to a class is nearest 0.5. For problems that have three or more classes, a more general uncertainty sampling variant might query the instance whose prediction is the least confident:

$$x^*_{LC} = \arg\max_x 1 - P_\theta(\hat{y}|x)$$

where $\hat{y} = \arg\max_x P_\theta(y|x)$ or the class label with the highest posterior probability under the model θ.

1.2.2 Query-By-Committee

Another, more theoretically-motivated query selecting framework is the query-by-committee (QBC) algorithm. The QBC approach involves maintaining a committee $C = \{\theta^1, \theta^2, .., \theta^{(C)}\}$ of models which are all trained on the current labeled set L but represent competing hypotheses. Each committee member is then allowed to vote on the labelings of query candidates. The most informative query is considered to be the instance about which they most disagree.

For measuring the level of disagreement, two main approaches have been proposed. The first is vote entropy:

$$x^*_{VE} = \arg\max_x \sum_i \frac{V_{y_i}}{C} \log \frac{V_{y_i}}{C}$$

where y_i again ranges over all possible labelings, and V_{y_i} is the number of "votes" that a label receives from among the committee members' predictions, and C is the committee size. This can be thought of as a QBC generalization of entropy-based uncertainty sampling. Another disagreement measure that has been proposed is average Kullback-Leibler (KL) divergence

$$x^*_{KL} = \frac{1}{C} \arg\max_x \sum_1^C D(P_{\theta^c} || P_C)$$

where :

$$D(P_{\theta^c})||P_C) = \arg\max_x \sum_i P_{\theta^c}(y_i|x) \log \frac{P_{\theta^c}(y_i|x)}{P_C(y_i|x)}$$

Here θ^c represents a particular model P in the committee, and C represents the committee as a whole, thus $P_C(y_i|x) = \sum_1^C P_{\theta^c}(y_i|x)$ is the "consensus" probability that y_i is the correct label.

To help understand the flow better, a typical cycle in an AL iteration is shown below:

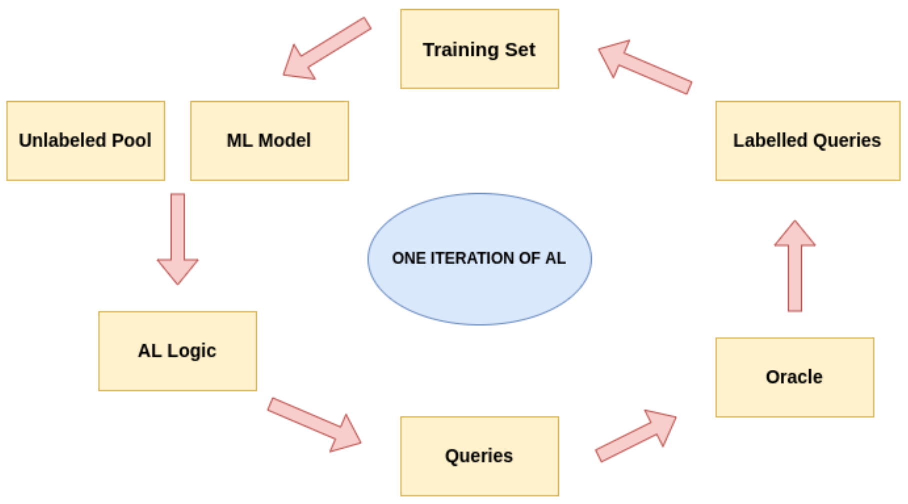


Figure 1.1: A typical cycle in an AL iteration is shown below

1.3 Active Class Selection

AL assumes that instances are inexpensively obtained, and the labeling process incurs a fee. Envision the opposite scenario, however, where an agent can query a known class label, and obtaining each instance incurs a cost. This reasonably new problem setting is called Active Class Selection(ACS). Lomasky et al. (2007) were the first to propose several ACS query algorithms for an “artificial nose” task, in which a machine learns to discriminate between different vapor types (the class labels), which must be chemically synthesized (to generate the samples). Some of their approaches show significant gains over the “passive” learning equivalent of uniform class sampling.

ACS algorithms have control over the data generation (or data synthesis process). They request to be fed a certain mixture of samples from different classes. The motivation behind such algorithms is that carefully controlling the class distribution of the samples fed to the ML model during training can help speed up the learning process compared to feeding arbitrary samples to the model.

1.3.1 Illustrative example

A vivid example of the application of ACS is the training of brain-computer interfaces for motoric prostheses as presented in the paper of PAL. To train such a prosthesis, an impaired patient has to imagine motoric movements, for example, of his fingers while his brain activity is recorded. The task is to build a model that reads the brain activity(X) and predicts the motoric movements(Y) he is trying to achieve. We do not have existing data, and neither do we have an oracle that helps us label these brain activities! We can only ask the patient to generate brain activity(X) for a motoric movement(Y) to get an (X, Y) pair that we can use to train our ML model. As usual, we must minimize the number of samples we collect from the patient to minimize the cost of acquisition and discomfort for the patient. Hence, ACS comes to our advantage in strategically asking the patient to generate the samples.

Figure 1.2 shows different learning stages of such an exemplary ACS process. In the beginning, the algorithm only knows the number of classes (fingers). If certain classes (fingers) are hard to distinguish in the data (here, class 1), learning should focus on these

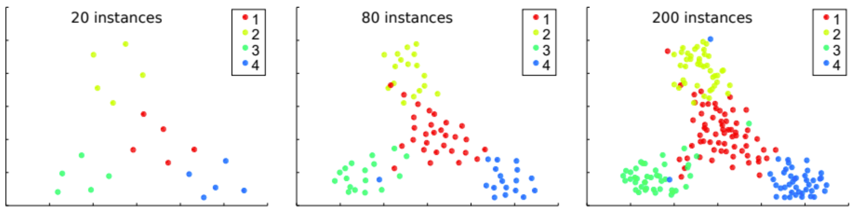


Figure 1.2: t-SNE plot of different learning stages in an ACS sampling process

classes. By requesting the patient to generate more training instances of these classes instead of spending time on already learned classes, a good classification performance is achieved earlier – an achievement that enables the patient to perform otherwise impossible tasks. As visualized by this example, ACS is useful whenever classes vary in difficulty.

CHAPTER 2

LITERATURE REVIEW

Active Class Selection (ACS) addresses the question: if one can collect n additional training instances, how should they be distributed with respect to class? In this section, we will briefly mention the theoretical developments in the field of ACS over the years.

2.1 Active Class Selection

This paper was the first on Active Class Selection (ACS), a new category of problems for multi-class supervised learning. The primary motivation for the paper was as follows: If one can control the classes from which training data is generated, using feedback during learning to steer the generation of new training data will yield better performance than learning from any a priori fixed class distribution. ACS is the process of iteratively selecting class proportions for data generation. In this paper, they present several methods for ACS. An empirical evaluation shows that for a fixed number of training instances, methods based on increasing class stability outperform methods that seek to maximize class accuracy or that use random sampling. Finally, they present the results of a deployed system for their motivating application: training an artificial nose to discriminate vapors.

They show that successful methods for ACS can be grounded by recent results in stability and generalization, which show that one can predict expected error based on empirical error with a stable learning algorithm that satisfies certain constraints. The goal of ACS is to minimize the number of new training examples needed in order to maximize learning performance. Given the ability to choose class proportions for data collection, they are interested in assessing for each class whether the empirical error is converging to the expected error, i.e., to determine when we can do no better for this class. Uniform sampling works best if the error rates of all classes are converging at the same rate. If this is not the case, then one heuristic is to sample in proportion to the inverse of the convergence rates for each class. Using this intuition, they propose two methods for ACS that base class proportions on heuristic assessments of class stability. They compare these two methods to uniform and random sampling (sampling in proportion to the distribution specified as best by the domain expert).

All of their methods begin with a small set of labeled training data T_1 of size $b[1]$, where $b[r]$ is the number of instances to add in round r. They perform f-fold cross-validation (CV) over T_1. From the CV, they obtain class predictions for T_1. Their methods differ in how they use these predictions to specify the class proportions ($Pr[c]$, $c \epsilon$ classes) for the next round of data generation. Specifically, on round r, they generate a new set of examples, T_{new}, a set of $b[r]$ examples generated using the class proportions $Pr[c]$. They add this data to the existing data to create a new set $T_r := T_{r1} + T_{new}$.

2.1.1 Uniform

Sample uniformly from all classes. This is especially useful when all the classes are of uniform difficulty.

$$P_r[c] := \frac{1}{|classes|} * b[r]$$

2.1.2 Inverse

Select class proportions $Pr[c]$ inversely proportional to their CV accuracy on round $r - 1$. Thus, they obtain more instances from classes where we have low accuracy. This method relies on the assumption that poor class accuracy is due to not having observed sufficient training data. Although this may be true initially, their results show that this method does not perform well in the long run.

$$P_r[c] := \frac{\frac{1}{acc[c]}}{\sum_1^{|classes|} \frac{1}{acc[i]}} * b[r]$$

2.1.3 Original Proportion

Sample in proportion to the class proportions in T_1. The idea is that domain knowledge led to these proportions, perhaps because they are the true underlying class distribution.

$$P_r[c] := n_c * b[r]$$

where n_c is the proportion of class c found in the collected data T.

2.1.4 Accuracy Improvement

Sample in proportion to each classes' change in accuracy from the last round. If the change for class c is 0, then $Pr[c] = 0$. The intuition is the accuracy of classes that have been learned as well as possible will not change with the addition of new data and thus we should focus on classes that can be improved. This method looks for stability in the empirical error of each class.

$$P_r[c] := max(0, \frac{currAcc[c] - lastAcc[c]}{\sum_1^{|classes|} currAcc[i] - lastAcc[i]})$$

2.1.5 Redistricting

The idea behind redistricting is that instances from T_{r1} whose classification changes when classified by a new classifier trained on $T_{r1} \bigcup T_{new}$ are near volatile boundaries. Thus, we strive to assess which classes are near volatile boundaries in order to sample from these these "unstable" classes. The pseudocode is shown below. They begin with a CV over T_1, the initial sample of the data. They obtain a prediction for each $x_i \epsilon T_1$. In the second round, they collect T_2 of size b[2]. They next perform a CV over all of the data collected thus far and create a classifier for each fold. Note that on subsequent iterations, they keep the data from T_{r1} in the same folds, and stratify only the newly generated data T_{new} into the existing folds. For each fold f, we compare the classification results of $C_{r,f}$ and $C_{r1,f}$ on each instance $x_i \epsilon T_{r1}$. If the labels are different, then the counter for the class specified by the true label y_i, $redistricted[yi]$ is incremented. They conclude by generating predictions of the new batch of data T_new and increment r.

After the second round they add instances using the formula in Step 12, where c is a class from the set of all classes in the dataset, $Pr[c]$ is the number of instances of c to add, n_c is the proportion of c in T_{r1} and $b[r]$ is the number of new training instances. They divide $redistricted[c]$ by n_c to keep small classes from being ignored and large classes from being overemphasized.

Algorithm 1 Redistricting Algorithm

```
Require: $b$, array of the number of instances to add in round $r$
 1: Generate a sample $T_1$ of size $b[1]$
 2: Divide $T_1$ into 10 stratified folds $T_{1,1}$, $T_{1,2}....T_{1,10}$
 3: for $f = 1$ to 10 do
 4:     Build Classifier $C_{1,f}$ from $\{T1 - T_{1,f}\}$
 5:     for all instances $x_i$ in $T_{1,f}$ do do $label_1[x_i] := C_{1,f}(x_i)$
 6:     end for
 7: end for
 8: $r := 2$
 9: while instance creation resources exist and stopping criteria not met do
10:     if $r = 2$ then $T_{new}$ := “random” sample of size $b[2]$
11:     else$T_{new}$ := sample of size $b[r]$ where the number of instances for class $c$ is
    computed as: $P_r[c] := \frac{\frac{redistricted[c]}{n_c}}{\sum_1^{|classes|} \frac{redistricted[i]}{n_c}} * b[r]$
12:     end if
13:     $T_r := T_{r1} + T_{new}$
14:     Initialize $redistricted[c]$, $\forall$ $c$ $\epsilon$ classes
15:     Divide $T_{new}$ into 10 stratified folds $T_{new,1}, T_{new,2}....T_{new,10}$
16:     for $f = 1$ to 10 do
17:         $T_{r,f} := T_{r1,f} \cup T_{new,f}$
18:         Build Classifier C$_{r,f} from\{$ T$_r - T_{r,f}\}$
19:         for all instances $x_i$ in $T_{r1,f}$ do
20:             $label_r[x_i] := C_{r,f}(x_i)$
21:             if $label_r[x_i] := label_{r1}[x_i]$ then
22:                 $redistricted[y_i] + +$ /* $y_i$ is the true label of $x_i$ */
23:             end if
24:         end for
25:         for all instances $x_i$ in $T_{new,f}$ do
26:             $label_r[x_i] := C_{r,f}(x_i)$
27:         end for
28:     end for
29:     $r + +$
30: end while
```

2.2 Probabilistic Active Learning for Active Class Selection

ACS algorithms usually seek to actively pick a class and ask the oracle to supply an instance for that class to optimize a classifier's performance while minimizing the number of requests. This paper proposes a new algorithm (PAL-ACS) that transforms the ACS problem into an active learning task by introducing pseudo instances. The performance gain model from probabilistic active learning is used to estimate the usefulness of an upcoming instance for each class. Our experimental evaluation (on synthetic and natural data) shows the advantages of their algorithm compared to state-of-the-art algorithms. It virtually prefers the sampling from challenging classes and hence improves the classification performance.

The core idea of their algorithm is to estimate the gain in classification performance for each class $y \epsilon Y = 1, ..., C$ when requesting one additional instance of that class y. Then, we request an instance x^* of the best class y^* and add this new training sample to the training set $L \leftarrow L \cup (x, y)$.

To estimate the expected gain in performance that a label request would probably induce, probabilistic active learning provides an effective tool. Its performance gain function can be calculated at any location in the feature space, regardless of the fact if there is a real unlabeled instance at this location. It only requires local statistics, which typically are labeling counts $\overrightarrow{k} = (k_1, ..., k_C)$. A common strategy to determine this vector are kernel frequencies as formulated below.

$$k_i = \sum_{(x_i, y_i) \epsilon L_i} \exp - \frac{||x - x'||^2}{2\sigma^2}$$

$$L_i = \{(x, y) \epsilon L : y = i\}$$

The performance gain function for label statistics $\overrightarrow{k}$ is defined by subtracting the current expected performance (0 labels added) from the future expected performance (m labels added).

$$perfGain(\overrightarrow{k}, M) = \max_{m \leq M} \frac{1}{m}(expPerf(\overrightarrow{k}, m) - expPerf(\overrightarrow{k}, 0))$$

More details regarding the computation of $expPerf$ function can be found in the original paper.

In contrast to active learning, they do not have access to a pool of unlabeled instances in ACS. Thus, they propose to generate pseudo instances x_p to transform the active class selection problem into an active learning task. They then use the pseudo instances to determine the most beneficial class y^* which is selected according to equation below.

$$y^* = \arg\max_{y} (\sum_{x_p} P(x_p|L) * P(x_p|L_y) * perfGain(\overrightarrow{k}))$$

Overall, PAL-ACS always identifies the difficult classes and samples accordingly. As a result, its performance is best (in cases some classes are more difficult than others) or equal with the best competitor (in cases all classes are equally difficult).

2.3 Weighted Informative Inverse Active Class Selection for Motor Imagery Brain Computer Interface

ACS can be a solution to non-uniform converging of error rates for different classes. ACS determines the class-wise ratio of samples in the training set based on observed error. One of the ACS methods is inverse proportion, i.e., the class proportion is the inverse of class accuracy. The entropy of unlabeled instances is an AL technique for finding the most informative samples. The intuition for their work is to feed the learner with the most informative instances while maintaining the class proportion given by the inverse ACS method. Following this idea, an improved weighted inverse ACS method is developed and combined with the AL query concept for BCI. This proposed weighted informative inverse active class selection method has been applied to BCI competition IV motor imagery (MI) binary class data set 2B. It showed better or similar performance on most of the subjects with less amount of training samples.

AL and ACS are iterative methods and address the problem of having limited training samples. So, ACS and AL can be combined as a complementary part to increase the BCI system's robustness using a lower number of training samples. The idea is to add the most informative or uncertain samples with the existing training set as per ACS prescribed class distribution iteratively. The authors introduce weighted accuracy based on the difference among accuracy of classes in the inverse ACS method for computing class distribution and integrate the AL method of finding informative instances with this proposed ACS method.

They first improve the $Inverse$ algorithm into *Weighted Inverse ACS (WIACS)* as follows:

$$w_c^r = \sum_1^C \exp \alpha * (A_{r-1}^c - A_{r-1}^i)$$

$$N_r^c = \frac{\frac{1}{A_{r-1}^c * w_c^r}}{\sum_{i=1}^C \frac{1}{A_{r-1}^i * w_c^r}} * n(r)$$

Here, w_{cr} is the weight for class c in r^{th} iteration and $alpha$ > 1, is weight constant determined by trial and error basis

They then propose the *Weighted Informative Inverse ACS (WIIACS)* method as follows: WIACS selects the class which will generate more instances in the next iteration. Usually, random instances are generated. They propose to get the most informative instances for chosen class. The idea is that informative training instances determined by the AL method of maximum entropy samples will be fed to the learner in the proportion given by WIACS. The hypothesis is that it will help the learner get adequate learning space, ensuring an equal convergence rate for all classes.

2.4 Optimal Probabilistic Classification in Active Class Selection

ACS aims to optimize the class proportions in newly acquired data; a classifier trained from that data should demonstrate maximum performance during its deployment. This paper provides an information-theoretic examination of the problem, resulting in an upper bound of the classifier's error. This upper bound shows that the more data is acquired, the better is the performance of the class proportions that occur during deployment; other class proportions can outperform these natural proportions at the beginning of data acquisition, but natural proportions indeed yield optimal probabilistic classifiers in the limit. The more data is acquired, the less beneficial ACS strategies are. Their bounds further reveal that the degree to which non-natural class proportions are eligible depends on the correlation between the features and the class label.

The core idea of this paper is motivated by the observation that sophisticated ACS strategies are often just as good as sampling randomly to the natural class proportions. Nevertheless, research on ACS tries hard to beat this straightforward baseline. Pursuing to explain this phenomenon, they investigate the theoretical background of ACS from the viewpoint of information theory. Their analysis yields an upper bound of the classifier's error, revealing that the natural class proportions provide optimal classifier performance in the limit of data acquisition.

For practical applications, they suggest preferring the natural class proportions over complicated active strategies—given that the natural proportions are precisely known and the acquisition budget allows for obtaining a sufficiently large sample. They show that research needs to focus on applications where one of these preconditions is not met. For example, in astroparticle physics, it is assumed that the ratio between the two prevalent classes of particles is roughly between 1:1000 and 1:10000, but the exact ratio is unknown. Also, the extreme level of such class imbalances requires a deeper examination in the context of ACS. While they have already identified the effect through which non-natural class proportions can be advantageous, how to optimize class proportions under such extreme imbalances remains open. Their experiments suggest that their theoretical conclusions about probabilistic classifiers also hold for non-probabilistic classifiers.

CHAPTER 3

THE PROBLEM STATEMENT

The formal problem statement is stated as follows:

We are allowed to query a maximum of N instances. We are allowed to query for a Y(class or y-value), the expert responds with a corresponding x(instance) for the Y. Our objective is to come up with a strategy of choosing Y s such that after we collect the dataset of N (x, Y) pairs, our "knowledge" about the dataset is maximum. We can define "knowledge" as our ability to predict on unseen data. We can assume $Y \epsilon$ [0,1] or that it can be normalized to [0,1]. Our target is to consistently beat the *Random Sampling* baseline over any sub-range of iterations by achieving a lower test RMSE score.

There can be many reasons for the problem at hand to be suitable for ACS. The absence of unlabelled data or oracle is an obvious case. However, there can be many other reasons to prefer ACS over other alternatives for the reasons listed below:

- The effort required for the oracle to process a query of ACS is less than traditional AL queries.
- The cost of conducting ACS queries is lesser than traditional AL queries.
- ACS is more effective/suitable for training an ML model on the given task (for example, if the AL method does not support the model we are planning to use).
- The problem domain is such that specific constraints prevent us from using traditional AL.

At the beginning of the project, we set out to answer the following questions:

- Find practical domains/constraints where ACS is applicable.
- Work out the mathematics to determine the label that needs to be queried in each iteration.
- Work out the model-specific mathematics of incorporating the label-sample pair while training the model.
- Conduct experiments to demonstrate that humans can indeed be effective ACS agents.

Once we discovered the existence of the field of Active Class Selection, we spent time conducting a literature survey, extending the ACS algorithms to regression tasks, and coming up with new ideas for the task of Active Class Selection for regression.

3.1 Motivating Examples

Assume that are interested in building a model to predict chess players' stable "rating." The rating demonstrates the skill of chess players. Let us hypothesize that chess players' rating(Y) can be predicted using input variables(X), including their demographics, practice schedules, frequency of tournament play, performance on benchmark skill tests, speed of calculations, etc. Predicting the rating is a regression task. It is suitable for ACS-Regression because of the following reasons. No data sets exist about the problem statement available to us. We will have to manually collect the data by meeting with chess players, interviewing them, and asking them to take the benchmark tests. However, each player will demand to be paid to attend the interview. Hence, we can only afford to interview a few players. If we wish to collect the data randomly before training our ML model, we do not know how much data we need to collect to attain the target performance. We note that the list of players and their ratings(Y) is already present on the FIDE website; therefore, we can choose to interview a player we wish to interview as our ACS-Regression algorithm dictates.

Another example can be estimating the percentage of particular types of seeds in a petri dish. Here, X = Image of the petri dish. Y = percentage of a particular type of seed in the petri dish. Again, we do not have an existing dataset that we can use. We are forced to generate the samples(images). Generating new images costs us labor, so we try to minimize the number of images generated. If we wish to collect the data randomly before starting to train our DL model, we do not know how much data we need to collect to attain the target performance. Also, in this case, we do not have a list of naturally occurring Y values available to us, unlike the chess rating example. We can, in theory, choose to generate an infinite amount of samples for a given "Y" value.

3.2 Observations

We made the following significant observations by doing a literature review of the field of Active Class Selection:

- Two major criteria for querying the next sample are diversity and informativeness.
- Poor performance on a class can occur because of low representation in the training set or because the class is harder to learn.
- We must not deviate too much from natural class proportions as it will violate the IID (Independent and Identically Distributed) constraint for training a model and will therefore lead to suboptimal performance.
- Although it is guaranteed that we can never asymptotically beat natural proportion based on querying strategy, we can still aim to achieve a better rate of convergence.

3.3 A Typical ACS-Regression Cycle

As we have seen before, there are existing algorithms for Active Class Selection in the classification setting. So, naturally, the first idea is to treat regression problems as classification problems by dividing the output variable range into equal-sized smaller intervals and considering them as separate classes; we call them y-bins. Whenever we are asked to provide an x from a given "class," we return a randomly selected sample from the corresponding y-bin.

Please refer to the image below for a pictorial representation of the setting.

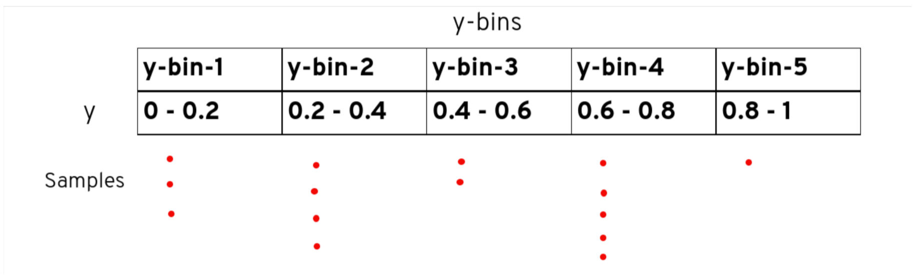


Figure 3.1: ACS-Regression setting if we break the y-range into 5 equal intervals. The red dots represent samples present inside these y-bins.

A typical ACS-Regression algorithm starts with initializing random samples as its training set. In each iteration, the algorithm spits out a "shopping list," consisting of counts of samples that it requires from different "classes." Once the requested samples are collected/generated, they are incorporated into the current training set. The algorithm then runs its custom method to create a new shopping list based on the model's performance and other evaluations. This cycle repeats until one of the stopping criteria is satisfied. Refer to the following image.

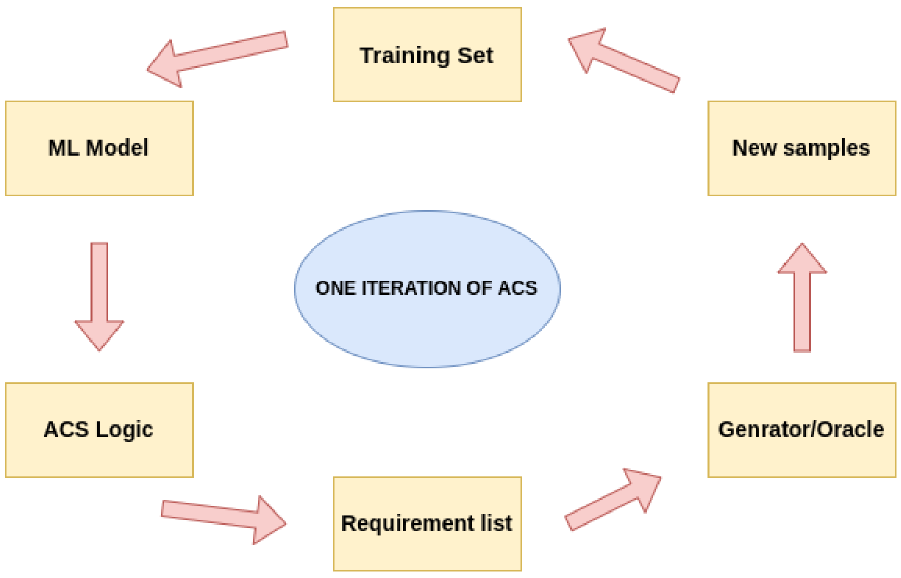


Figure 3.2: A typical cycle in an ACS-Regression iteration is shown below

CHAPTER 4

EVALUATION FRAMEWORK

Every algorithm we propose has to be evaluated to understand its performance. Typically for ML models, the evaluation is done by maintaining a separate test set. The performance on the test set is taken as an empirical performance of the model on unseen data. However, when it comes to the field of Active Class Selection, unlike common ML problems, we do not possess any data regarding the problem. Hence, we cannot create a separate test set at the beginning of the training process. Hence, we note that the a large test set is created only to demonstrate the algorithm's performance and will not be required during the actual usage of the algorithms.

Usually, when a new algorithm is designed, it is evaluated the performance of other state-of-the-art algorithms. However, when we design algorithms in areas with no existing algorithms, we test our new algorithms against "baselines," these are simple algorithms that offer a minimum natural performance that any newly designed algorithms are expected to beat. ACS-Regression is a new field of research with no existing algorithms; hence in the following sections, we will design a few standard baselines to help us understand the performance of our algorithms.

4.1 Datasets and Models

The datasets are taken from Penn Machine Learning Benchmarks from PMLB website [https://github.com/EpistasisLab/pmlb]. We have conducted experiments on all feasible datasets in the repository to make our deductions or analysis. However, for demonstration, we will stick to three datasets, namely – **'1191-BNG-pbc'**, **'1196-BNG-pharynx'**, **'1201-BNG-breastTumor'**. We have used **XGboost** as the core learning model in all algorithms. XGboost was chosen as the core model because it is suitable for both the task of classification and regression, and it just needs one sample to kick off the training process. The core model will be re-trained as new data is queried and added to the training set in each iteration.

4.2 Baselines

Notation:

b: Batch Size (Number of samples queried in a single iteration).

M: Total number of classes (The number of pieces in to which the Y range is split into).

T: Total number of iterations.

$ybin_i$: The i^{th} ybin.

n(i): Number of samples requested to be added from $ybin_i$ in that iteration.

These baselines dictate how to choose the ybin from which the new sample should be collected.

1. **Random Sampling** : This is the core baseline that we use throughout out project. It involves adding a random subset of size b (batch size) from the remaining dataset into the training set in each iteration.
2. **Uniform Querying**: This algorithm assumes that each class is of similar difficulty, and hence we query with equal probability a point from each ybin: i.e., the probability of querying a sample from any ybin = $\frac{1}{M}$.
3. **Cyclic Querying**: Here, we take a cyclic tour on all the ybins, i.e., $ybin_0$, $ybin_1$, $ybin_M$, and so on. We come back to $ybin_0$ after covering all the y-bins.
4. **Alternated Cyclic Querying**: In this algorithm, we cover the y-bins alternatingly, i.e., $ybin_0$, $ybin_M$, $ybin_1$, $ybin_{M-1}$, $ybin_2$, $ybin_{M-2}$, and so on. Like cyclic querying, we loop back to the other ends of the ybins if we cover all the ybins.
5. **Querying the Most Diverse ybin**: Here, we try to query from the y-bin farthest from all the ybins already existing in the training set. We query the median ybin to start the iteration.

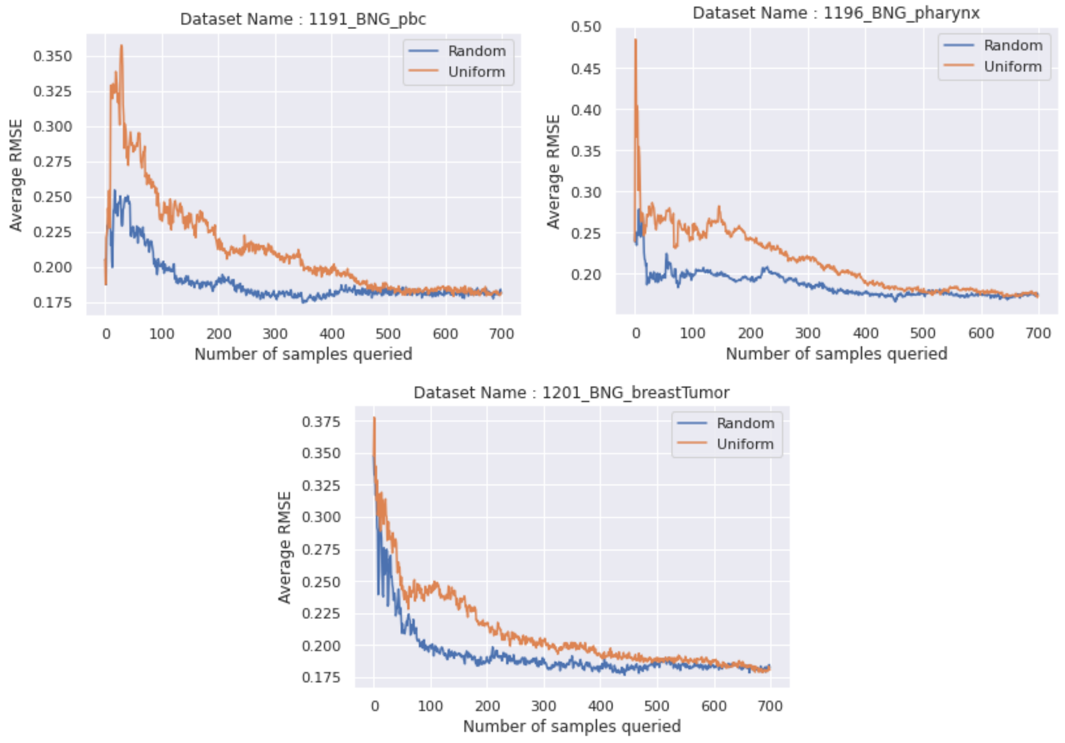


Figure 4.1: Performance of Uniform Querying algorithm

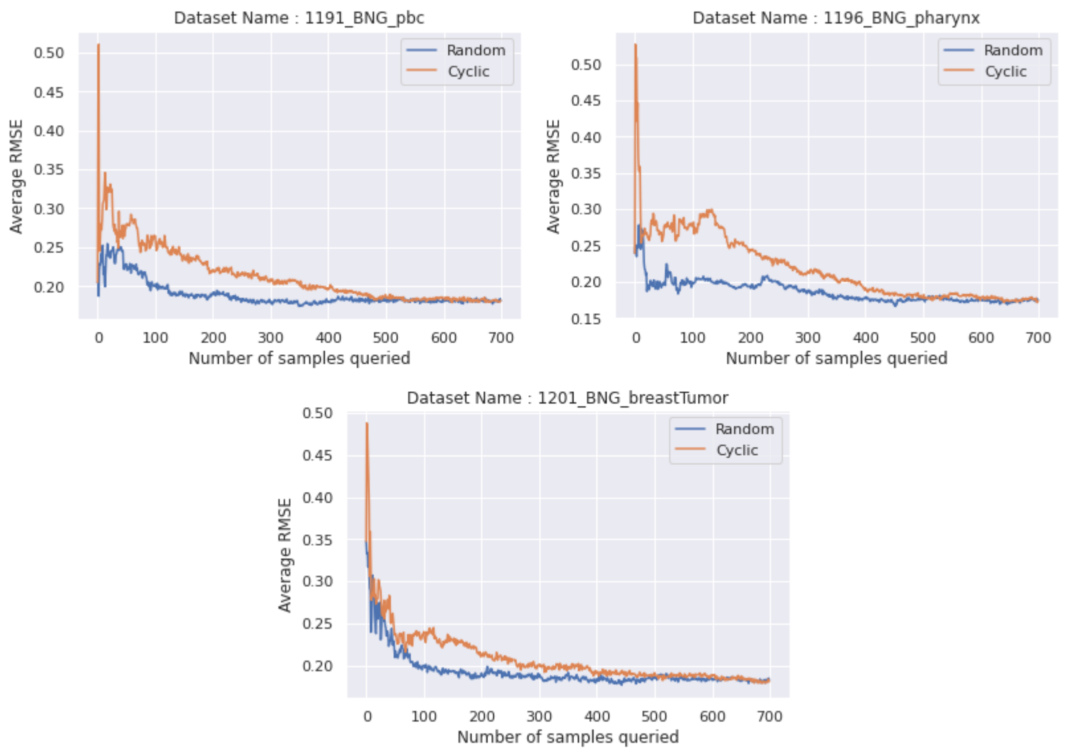


Figure 4.2: Performance of Cyclic Querying algorithm

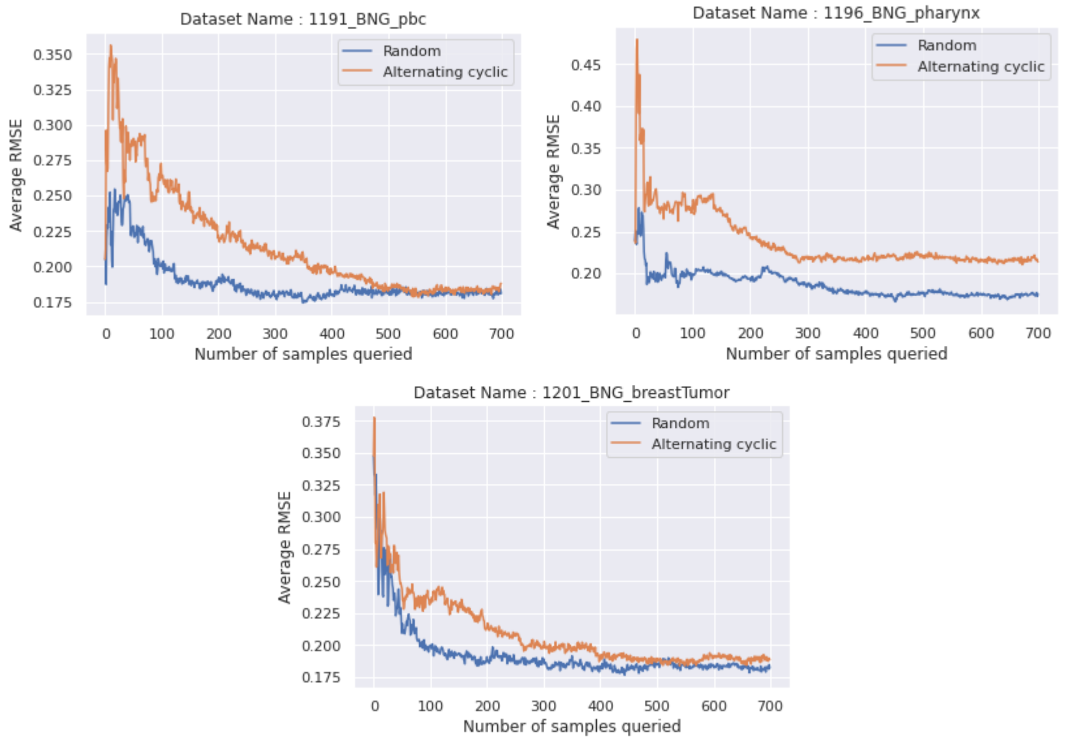


Figure 4.3: Performance of Alternated Cyclic Querying algorithm

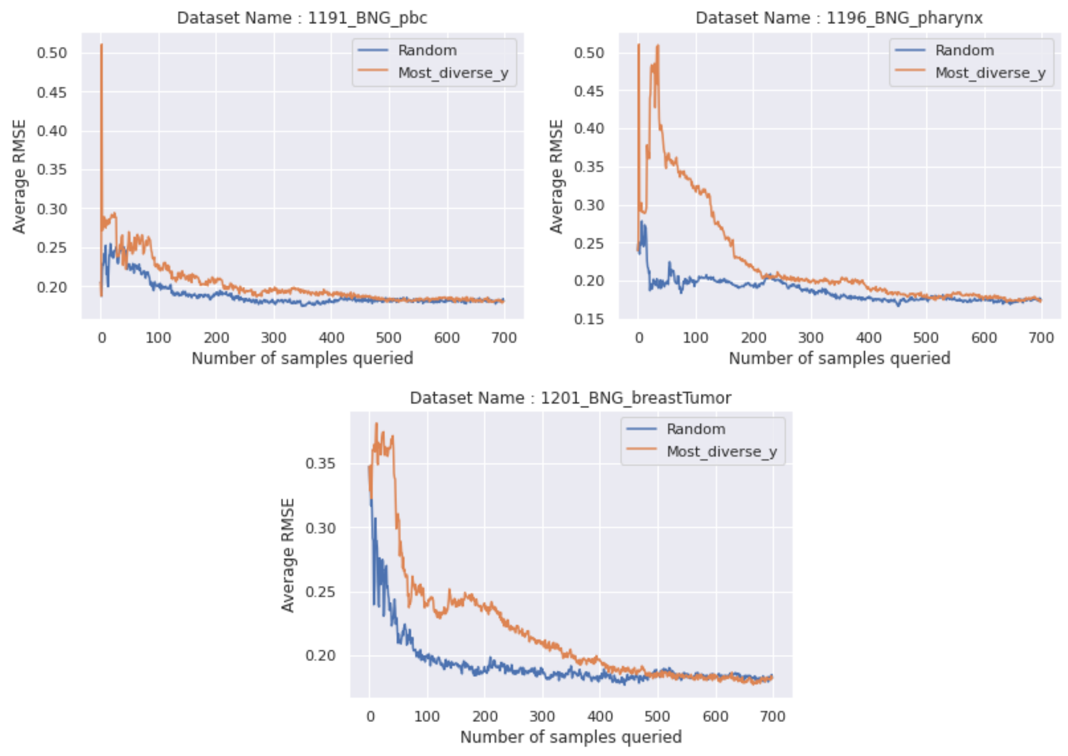


Figure 4.4: Performance of Querying the Most Diverse ybin algorithm

CHAPTER 5

LOMASKY INSPIRED ALGORITHMS

The idea of ACS is to distribute the number of instances per class such that a certain level of classification performance is reached with the lowest number of requested instances. The work presented in Lomasky et al. mentions different techniques to determine this class distribution for acquisition chunks. We will focus on two of these techniques, namely *Inverse* and *Redistricting*.

The approach *Inverse* distributes the samples according to the inverse of the class accuracy. An extension of this is called *Accuracy Improvement*. It distributes the values according to the accuracy difference between the two most recent chunks. The *Redistricting* method counts the number of labels that have been flipped (these instances are marked as redistricted) by adding the most recent chunk to the training set. Here, the upcoming instances are distributed with respect to the number of redistricted instances of the true classes.

In order to adapt these algorithms to the regression task, we need to make a few changes to how we treat the dataset and the algorithm. As regression deals with predicting the value of a continuous variable, it technically does not contain a countable number of output classes. Hence, we adopt the "y-binning " concept described in earlier sections. It helps us split up the y-range into a finite number of 'classes' – here, we treat a small range of y values as a class. The assumption of binning like this is that some natural similarities exist between the samples with close enough y values.

We observed that the baseline algorithms we constructed could not match the baseline of random sampling over all iterations. The methods were also much more unstable than random sampling - The spikes can be observed in the RMSE score in the initial iterations. However, most algorithms achieved a comparable asymptotic RMSE score towards the end of the iterations. We observed that the Inverse Accuracy was unable to perform well compared to the baseline. The rate of convergence of the Inverse Accuracy algorithm is low compared to the baseline. On the other hand, the Redistricting method, after being adapted for the regression task with a few modifications, was able to perform very close to the baseline of Random Sampling. In fact, on two datasets, Redistricting had a marginally better convergence rate during the initial iterations. However, RMSE was slightly worse than the baseline after a few iterations. To ensure that the performance of the algorithms was not a result of any 'lucky' hyperparameter setting, we thoroughly examined all the algorithms under various hyperparameter settings. We ensured that they were behaving in a stable manner. The redistricting method had comparable performance to the baseline under most hyperparameter settings. Similarly, we verified that the worse performance of other algorithms was not because of a particular hyperparameter setting.

5.1 Redistriction

In this section, we explain the details of how we modify Lomasky's redistricting algorithm for the regression task. Firstly, we use the concept of "binning" to separate the datasets into C classes. Although we have divided the dataset into different classes, this is only to return appropriate samples when the algorithm asks for them. Note that the model used at the algorithms' core will be Regression, i.e., it can take in a value of x and predict a continuous variable y. Lomasky's classification-redistricting algorithm works by checking the stability of model prediction on adding new samples. It checks if the "class label" of the previous training samples changed when the new samples are incorporated into the training set. We will use the "binning" classes we assigned at the beginning to see if the sample changed its class or not.

The performance of the Redistricting-Regression algorithm can be seen below.

5.2 Inverse score

Similar to the redistricting algorithm, we need to modify the Inverse method that Lomasky originally proposed. We need to use "binning" to create virtual classes for the regression datasets. As there is no concept of "accuracy" in regression tasks, we need to use a similar performance score for regression, the RMSE score. The core idea behind the inverse method is that the worse performance of the algorithm on a few classes is caused by the deficiency of samples in those classes.

Algorithm 2 Redistricting Algorithm for Regression

```
1: Generate a sample $T_1$ of size $b$
2: Divide $T_1$ into $k$ stratified folds $T_{1,1}$, $T_{1,2}....T_{1,k}$
3: for $f = 1$ to k do
4:     Build Regression Model $R_{1,f}$ from $\{T1 - T_{1,f}\}$
5:     for all instances $x_i$ in $T_{1,f}$ do do $label_1[x_i] := R_{1,f}(x_i)$
6:     end for
7: end for
8: $r := 2$
9: while instance creation resources exist and stopping criteria not met do
10:     if $r = 2$ then $T_{new}$ := "random" sample of size $b$
11:     else$T_{new}$ := sample of size $b$ where the number of instances for class $c$ is computed as: $P_r[c] := \frac{\frac{redistricted[c]}{n_c}}{\sum_1^{|classes|} \frac{redistricted[i]}{n_c}} * b$
12:     end if
13:     $T_r := T_{r1} + T_{new}$
14:     Initialize $redistricted[c]$, $\forall\ c\ \epsilon$ classes
15:     Divide $T_{new}$ into k stratified folds $T_{new,1}, T_{new,2}....T_{new,k}$
16:     for $f = 1$ to $k$ do
17:         $T_{r,f} := T_{r1,f} \cup T_{new,f}$
18:         Build Regression Model $R_{r,f}$ from $\{T_r - T_{r,f}\}$
19:         for all instances $x_i$ in $T_{r-1,f}$ do
20:             $label_r[x_i] := R_{r,f}(x_i)$
21:             if $ybin(label_r[x_i]) \neq ybin(label_{r-1}[x_i])$ then
22:                 $redistricted[ybin(y_i)] + +$ /* $y_i$ is the true label of $x_i$ */
23:             end if
24:         end for
25:         for all instances $x_i$ in $T_{new,f}$ do
26:             $label_r[x_i] := R_{r,f}(x_i)$
27:         end for
28:     end for
29:     $r + +$
30: end while
```

Algorithm 3 Inverse-RMSE Algorithm for Regression

```
1: Generate a sample $T_1$ of size b
2: while instance creation resources exist and stopping criteria not met do
3:     $T_{new}$ := sample of size b where the number of instances of class c is computed as $P_r[c] := \frac{\frac{1}{RMSE[c]}}{\sum_1^{|classes|} \frac{1}{RMSE[i]}} * b$
4:     $T_r := T_{r-1} + T_{new}$
5: end while
```

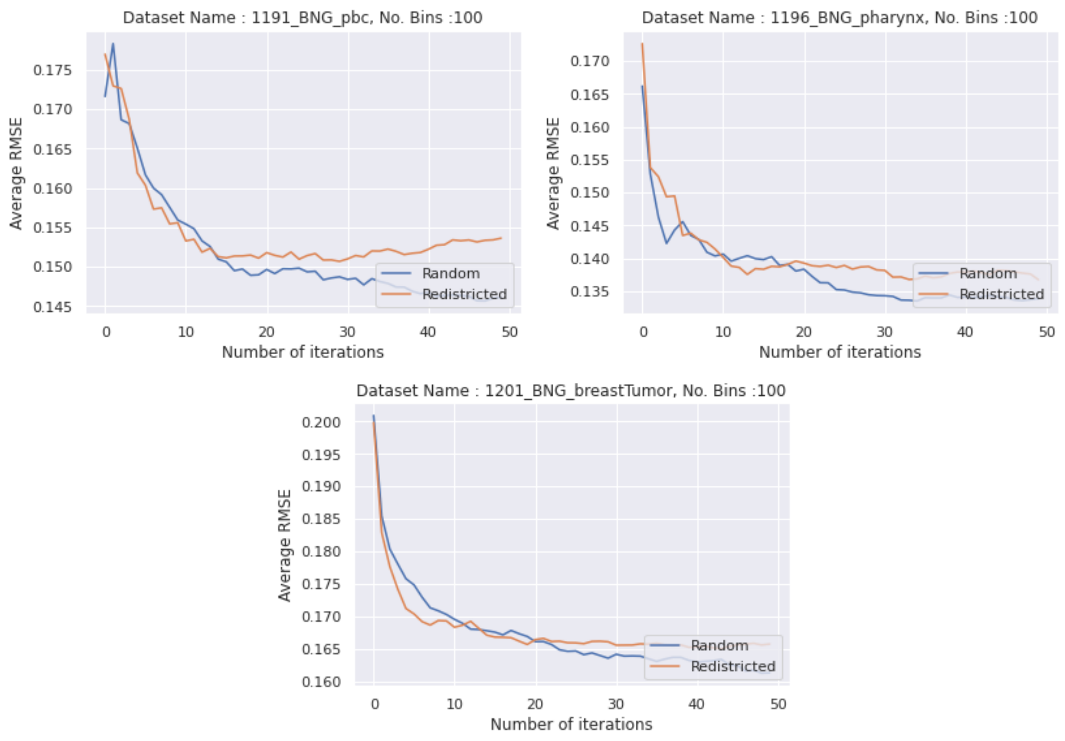


Figure 5.1: Performance of Redistricting-Regression algorithm

The assumption that having a low number of samples leads to a bad performance on that class is usually only justified at the start of the training. Another downside of this method is that it assumes that all classes are logically formed, and having a large number of samples from any class is sufficient for the model to have a good test RMSE on them. However, it does not take care of the case when a class is ill-formed or contains mostly outliers from which no information can be gathered, no matter how many samples we collect from those classes. It 'wastes' its resources by forcefully trying to improve the performance of under-performing classes. However, in many cases, bad performance is not only caused by having a low number of classes but usually classes being very hard to train due to the presence of outliers. It can also happen if the class distribution is highly complicated to decipher information from the training samples.

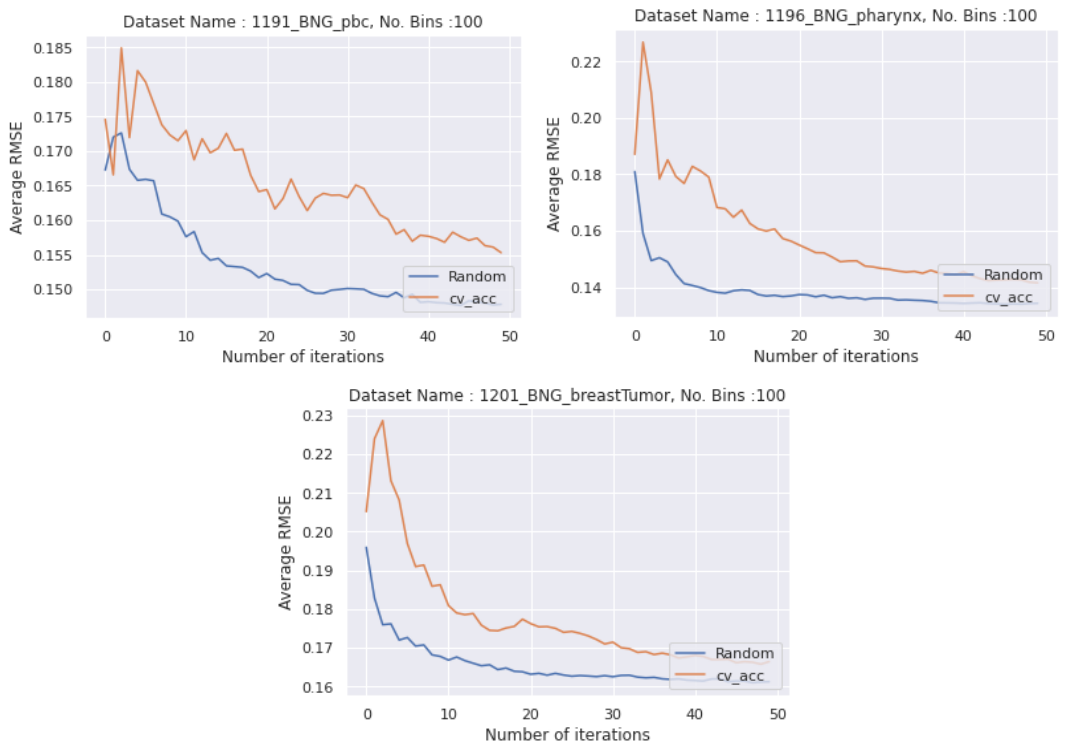


Figure 5.2: Performance of Inverse-RMSE algorithm

CHAPTER 6

MODIFIED REDISTRICTING ALGORITHMS

In this chapter, we will discuss the modifications attempted on the classical redistricting algorithm.

The objective was to improve the redistricting algorithm on these aspects:

1. How can we modify the redistricting algorithm to account for the magnitude of deviations after adding new samples and not just the redistricting counts?
2. As regression is inherently a continuous variable prediction task, can we redesign the redistricting algorithm without using the concept of binning?
3. Can we devise a better binning strategy for the y-range instead of dividing it uniformly?
4. Can we make the y-bins dynamic with iterations as the distribution of the remaining training data keeps changing?

6.1 Deviation dependent Redistricting Algorithm

As regression is a continuous method, maintaining a redistricting-score based on hard classifying the samples as 'changed class'/'not changed class' is inappropriate. Let us assume we classify a sample as 'redistricted' if the prediction after adding the new samples deviates from the original prediction by more than a threshold d. If we keep this threshold extremely low, we will observe that most of the sample will get counted as redistricted! On the other hand, if we keep the threshold very high, we might only consider those points to be redistricted that are outliers or are very hard to understand for the model.

To account for the amount of deviation that the sample has incurred after the new samples are added, we modify the redistricting score to be proportional to the average of deviations (i.e., $|prediction_{new} - prediction_{old}|$) of all samples in each ybin. Hence, we give more importance to those classes whose samples, on average, deviated more.

Algorithm 4 Deviation dependent Redistricting Algorithm

```
1: Generate a sample $T_1$ of size $b$
2: Divide $T_1$ into $k$ stratified folds $T_{1,1}$, $T_{1,2}....T_{1,k}$
3: for $f = 1$ to k do
4:     Build Regression Model $R_{1,f}$ from $\{T1 - T_{1,f}\}$
5:     for all instances $x_i$ in $T_{1,f}$ do do $label_1[x_i] := R_{1,f}(x_i)$
6:     end for
7: end for
8: $r := 2$
9: while instance creation resources exist and stopping criteria not met do
10:    if $r = 2$ then $T_{new}$ := "random" sample of size $b$
11:    else$T_{new}$ := sample of size $b$ where the number of instances for class $c$ is computed as: $P_r[c] := \frac{\frac{deviation[c]}{n_c}}{\sum_1^{|classes|} \frac{deviation[i]}{n_c}} * b$
12:    end if
13:    $T_r := T_{r1} + T_{new}$
14:    Initialize $deviation[c]$, $\forall\ c \ \epsilon$ classes
15:    Divide $T_{new}$ into k stratified folds $T_{new,1}, T_{new,2}....T_{new,k}$
16:    for $f = 1$ to $k$ do
17:        $T_{r,f} := T_{r1,f} \cup T_{new,f}$
18:        Build Regression Model $R_{r,f}$ from $\{T_r - T_{r,f}\}$
19:        for all instances $x_i$ in $T_{r-1,f}$ do
20:            $label_r[x_i] := R_{r,f}(x_i)$
21:            $deviation[ybin(y_i)] += abs(label_r[x_i] - label_{r-1}[x_i])$
22:        end for
23:        for all instances $x_i$ in $T_{new,f}$ do
24:            $label_r[x_i] := R_{r,f}(x_i)$
25:        end for
26:    end for
27:    $r++$
28: end while
```

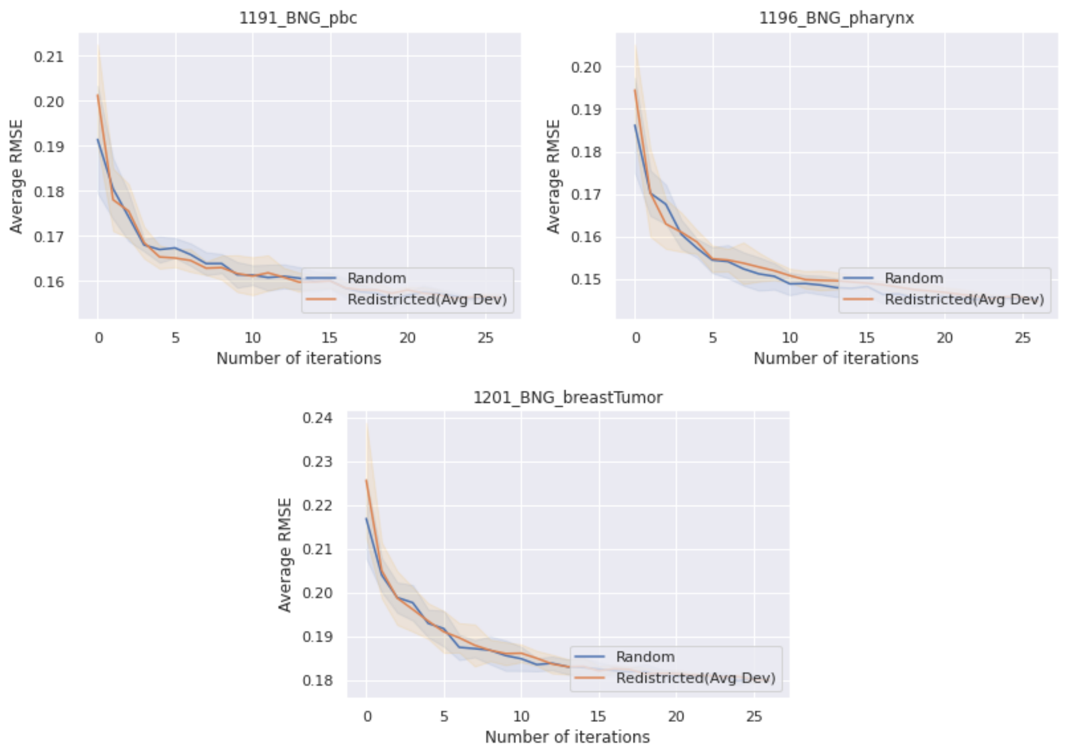


Figure 6.1: Performance of Deviation dependent Redistricting Algorithm

6.2 Continuous Redistricting Algorithm

One of the main objectives of this section is to attempt to build a redistricting algorithm that does not use binning. To achieve this, we extend the deviation-dependent Redistricting Algorithm proposed in the previous section. Ideally, it would be best if we could get a continuous PDF as a request from the redistricting algorithm. Assuming we have a magical method that allows us to sample from this PDF, we would be able to complete the loop of continuous redistricting. However, there are two challenges : (1) The previous method of the deviation-dependent redistricting algorithm gives us a discrete redistricting score and not a continuous PDF. (2) We need to design the function to return samples from the remaining training data following the continuous PDF.

The first issue can be addressed as follows. We do not compute the average deviation separately for each ybin. Instead, we generate (y_{true}, deviation) pairs for all the samples in the current training set. We train a separate regression model using these (y_{true}, deviation) pairs. Building this model allows us to have a model that can tell us the value of the expected deviation for any value of y, thereby giving us some sort of PDF on the y-range.

The second issue of designing a sampling function from the training set following a PDF can be built as follows. We sample a y-value from the PDF using statistical methods, then we return an instance in the training set closest to the y-value we obtained by sampling the PDF. We repeat this process to get the total number of samples required.

However, on implementing the algorithm we designed above, we noticed from the scatter plots of (y, deviation) points that the spread of these points varied heavily even around a single y value. Very high variation for a given input value meant that if we used any ML model to fit the (y, deviation) values, we would not get a good model. To prevent this, we removed all the points with less than 30% deviation. This got rid of many points. However, to remove the spread for a single y-value, we took the 75th percentile value of deviation within a (y - delta, y + delta) region to get a single estimate of deviation for the y value. Nevertheless, the algorithm's performance was still much worse than the random sampling baseline.

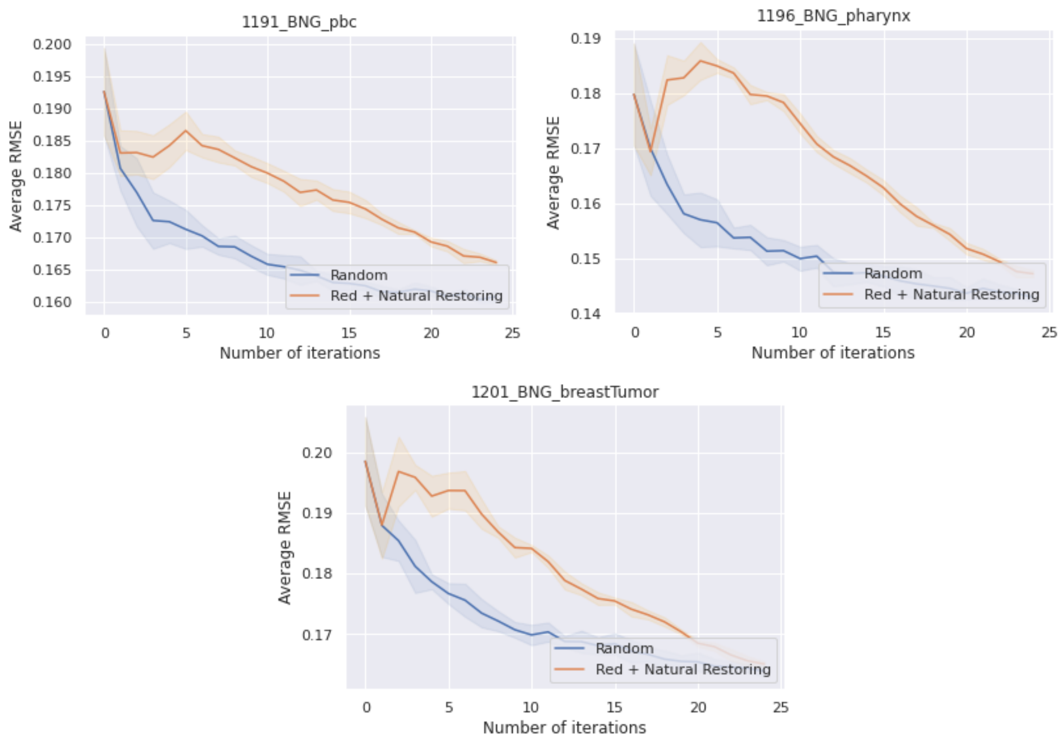


Figure 6.2: Performance of Continuous Redistricting Algorithm

Algorithm 5 Continuous Redistricting Algorithm

```
1: Generate a sample $T_1$ of size $b$
2: Divide $T_1$ into $k$ stratified folds $T_{1,1}$, $T_{1,2}$....$T_{1,k}$
3: for $f = 1$ to k do
4:     Build Regression Model $R_{1,f}$ from $\{T1 - T_{1,f}\}$
5:     for all instances $x_i$ in $T_{1,f}$ do do $label_1[x_i] := R_{1,f}(x_i)$
6:     end for
7: end for
8: $r := 2$
9: while instance creation resources exist and stopping criteria not met do
10:     if $r = 2$ then $T_{new}$ := “random” sample of size $b$
11:     else$T_{new}$ := sample of size $b$, each one sampled from PDF
12:     end if
13:     $T_r := T_{r1} + T_{new}$
14:     Divide $T_{new}$ into k stratified folds $T_{new,1}, T_{new,2} \dots T_{new,k}$
15:     $DeviationPairs = \{\}$
16:     for $f = 1$ to $k$ do
17:         $T_{r,f} := T_{r1,f} \cup T_{new,f}$
18:         Build Regression Model $R_{r,f}$ from $\{T_r - T_{r,f}\}$
19:         for all instances $x_i$ in $T_{r-1,f}$ do
20:             $label_r[x_i] := R_{r,f}(x_i)$
21:             Insert $\{y_i, abs(label_r[x_i] - label_{r-1}[x_i])\}$ into $DeviationPairs$
22:         end for
23:         for all instances $x_i$ in $T_{new,f}$ do
24:             $label_r[x_i] := R_{r,f}(x_i)$
25:         end for
26:     end for
27:     Train Regression model of $DeviationPairs$ and generate deviation PDF
28:     $r++$
29: end while
```

6.3 Gaussian Mixture Model based Binning

In section we introduced the concept of binning. Binning helps us modify the regression dataset into classification by dividing the continuous output value into bins. However, the binning strategy described in section divides the normalized y-range of [0,1] into smaller bins of size $\frac{1}{m}$ if m is set as the number of classes. However, there are multiple issues with this approach. It assumes that all segments of the y-range are of the same difficulty, sample density, and importance. It does not give us a good idea about the value of the hyperparameter m we must use. Even if the dataset had actual natural clustering in the output dimension, uniform binning would not think twice before breaking a coherent natural class into multiple separate classes. We use one-dimensional Gaussian Mixture Based (GMM) clustering to fix these issues. GMM can detect the presence of natural classes in the y dimension. It also gives us a good idea of the hyperparameter m if we use the elbow method to find the best value of m.

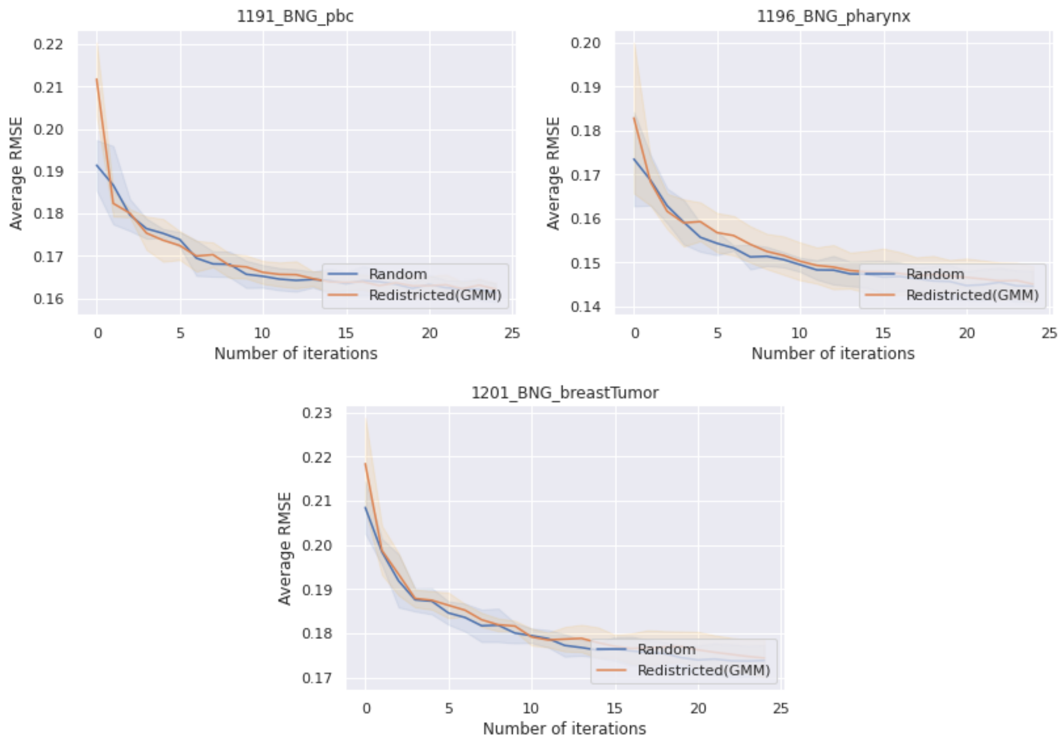


Figure 6.3: Performance of Gaussian Mixture Model based Binning

Algorithm 6 Gaussian Mixture Model based Binning

```
 1: Use GMM to separate training set y-values into bins
 2: Generate a sample $T_1$ of size $b$
 3: Divide $T_1$ into $k$ stratified folds $T_{1,1}$, $T_{1,2}$....$T_{1,k}$
 4: **for** $f = 1$ to k **do**
 5:     Build Regression Model $R_{1,f}$ from $\{T1 - T_{1,f}\}$
 6:     **for** all instances $x_i$ in $T_{1,f}$ **do** do $label_1[x_i] := R_{1,f}(x_i)$
 7:     **end for**
 8: **end for**
 9: $r := 2$
10: **while** instance creation resources exist and stopping criteria not met **do**
11:     **if** $r = 2$ **then** $T_{new}$ := “random” sample of size $b$
12:     **else**$T_{new}$ := sample of size $b$ where the number of instances for class $c$ is com-
    puted as: $P_r[c] := \frac{\frac{redistricted[c]}{n_c}}{\sum_1^{|classes|} \frac{redistricted[i]}{n_c}} * b$
13:     **end if**
14:     $T_r := T_{r1} + T_{new}$
15:     Initialize $redistricted[c]$, $\forall$ $c$ $\epsilon$ classes
16:     Divide $T_{new}$ into k stratified folds $T_{new,1}, T_{new,2} .... T_{new,k}$
17:     **for** $f = 1$ to $k$ **do**
18:         $T_{r,f} := T_{r1,f} \cup T_{new,f}$
19:         Build Regression Model $R_{r,f}$ from $\{T_r - T_{r,f}\}$
20:         **for** all instances $x_i$ in $T_{r-1,f}$ **do**
21:             $label_r[x_i] := R_{r,f}(x_i)$
22:             **if** $GMM_{bin}(label_r[x_i]) \neq GMM_{bin}(label_{r-1}[x_i])$ **then**
23:                 $redistricted[GMM_{bin}(y_i)] + +$ /* $y_i$ is the true label of $x_i$ */
24:             **end if**
25:         **end for**
26:         **for** all instances $x_i$ in $T_{new,f}$ **do**
27:             $label_r[x_i] := R_{r,f}(x_i)$
28:         **end for**
29:     **end for**
30:     $r + +$
31: **end while**
```

6.4 Dynamic Binning

In the previous sections, we described uniform and GMM-based binning strategies. However, as we move through iterations and sample points from the remaining training set, the distribution of the remaining training set changes as the distribution of the samples requested by the ACS algorithms might differ from the natural distribution of the remaining training set. We need a dynamic solution to make the binning adaptive to the changing distribution of the remaining training set. To achieve this, we use binning before every ACS iteration, thereby getting a fresh set of bins suitable for the current samples in the remaining train set.

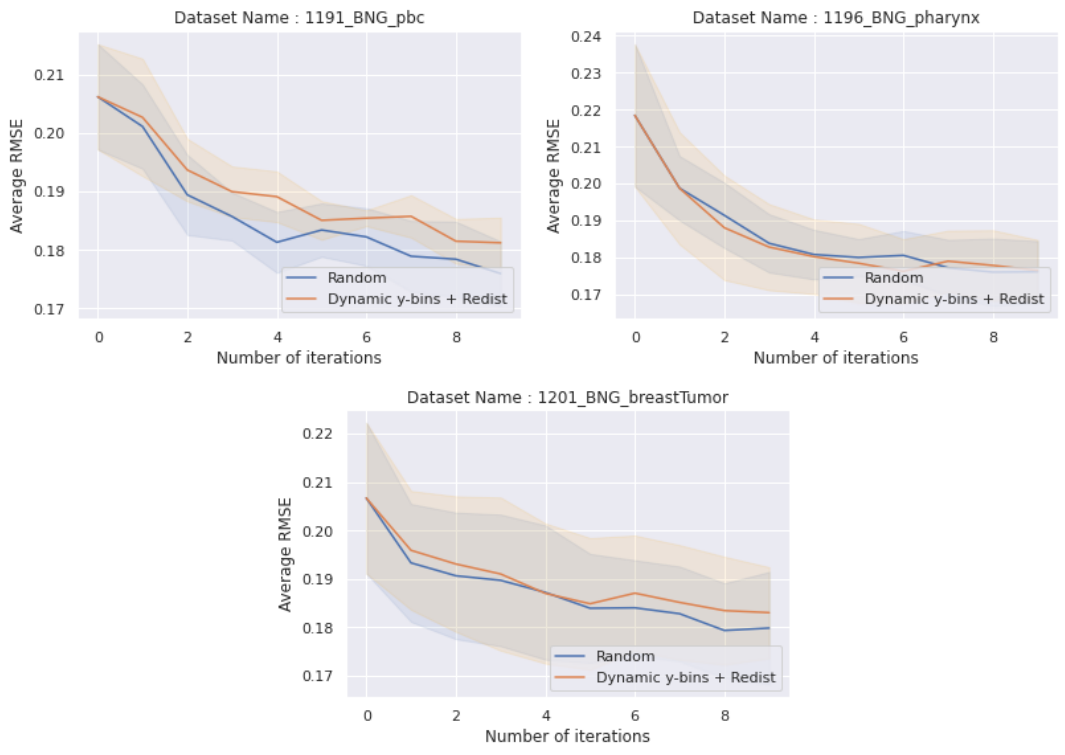


Figure 6.4: Performance of Dynamic Binning Redistricting Algorithm

Algorithm 7 Dynamic Binning

```
 1: Use GMM to separate training set y-values into bins
 2: Generate a sample $T_1$ of size $b$
 3: Divide $T_1$ into $k$ stratified folds $T_{1,1}$, $T_{1,2}$....$T_{1,k}$
 4: for $f = 1$ to k do
 5:     Build Regression Model $R_{1,f}$ from $\{T1 - T_{1,f}\}$
 6:     for all instances $x_i$ in $T_{1,f}$ do do $label_1[x_i] := R_{1,f}(x_i)$
 7:     end for
 8: end for
 9: $r := 2$
10: while instance creation resources exist and stopping criteria not met do
11:     if $r = 2$ then $T_{new}$ := “random” sample of size $b$
12:     else$T_{new}$ := sample of size $b$ where the number of instances for class $c$ is computed as: $P_r[c] := \frac{\frac{redistricted[c]}{n_c}}{\sum_1^{|classes|} \frac{redistricted[i]}{n_c}} * b$
13:     end if
14:     $T_r := T_{r1} + T_{new}$
15:     Use a new GMM to separate the remaining training set y-values into bins
16:     Initialize $redistricted[c]$, $\forall$ $c$ $\epsilon$ classes
17:     Divide $T_{new}$ into k stratified folds $T_{new,1}, T_{new,2}....T_{new,k}$
18:     for $f = 1$ to $k$ do
19:         $T_{r,f} := T_{r1,f} \cup T_{new,f}$
20:         Build Regression Model $R_{r,f}$ from $\{T_r - T_{r,f}\}$
21:         for all instances $x_i$ in $T_{r-1,f}$ do
22:             $label_r[x_i] := R_{r,f}(x_i)$
23:             if $GMM_{bin}(label_r[x_i]) \neq GMM_{bin}(label_{r-1}[x_i])$ then
24:                 $redistricted[GMM_{bin}(y_i)] + +$ /* $y_i$ is the true label of $x_i$ */
25:             end if
26:         end for
27:         for all instances $x_i$ in $T_{new,f}$ do
28:             $label_r[x_i] := R_{r,f}(x_i)$
29:         end for
30:     end for
31:     $r + +$
32: end while
```

CHAPTER 7

INPUT SPACE DISPARITY

In this section we propose a new way of querying in Active Class Selection for Regression settings.

The core idea of Input Space Disparity is to sample more points from those 'classes' which the 'understanding' of the models do not agree upon. To measure the 'understanding' of a model regarding a class, we estimate what the model thinks is the region in which the class exists in the input space.

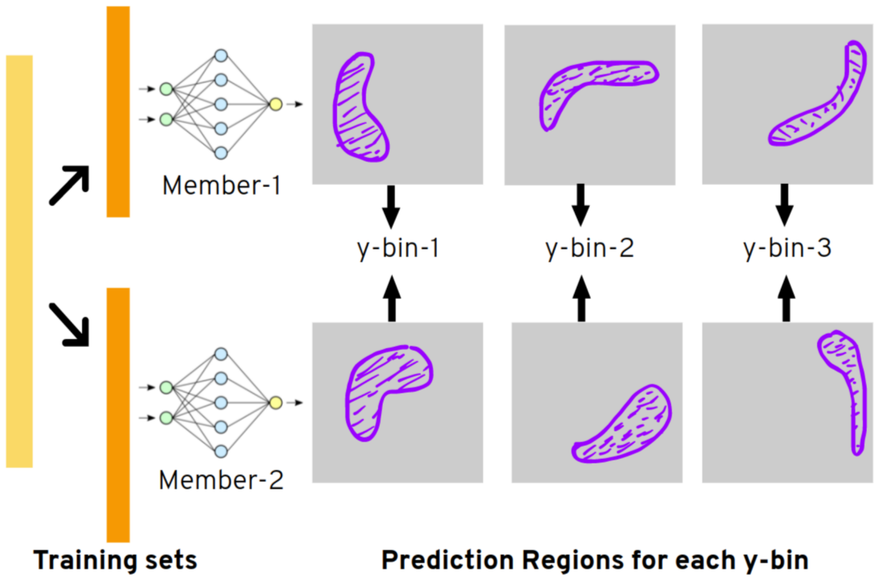
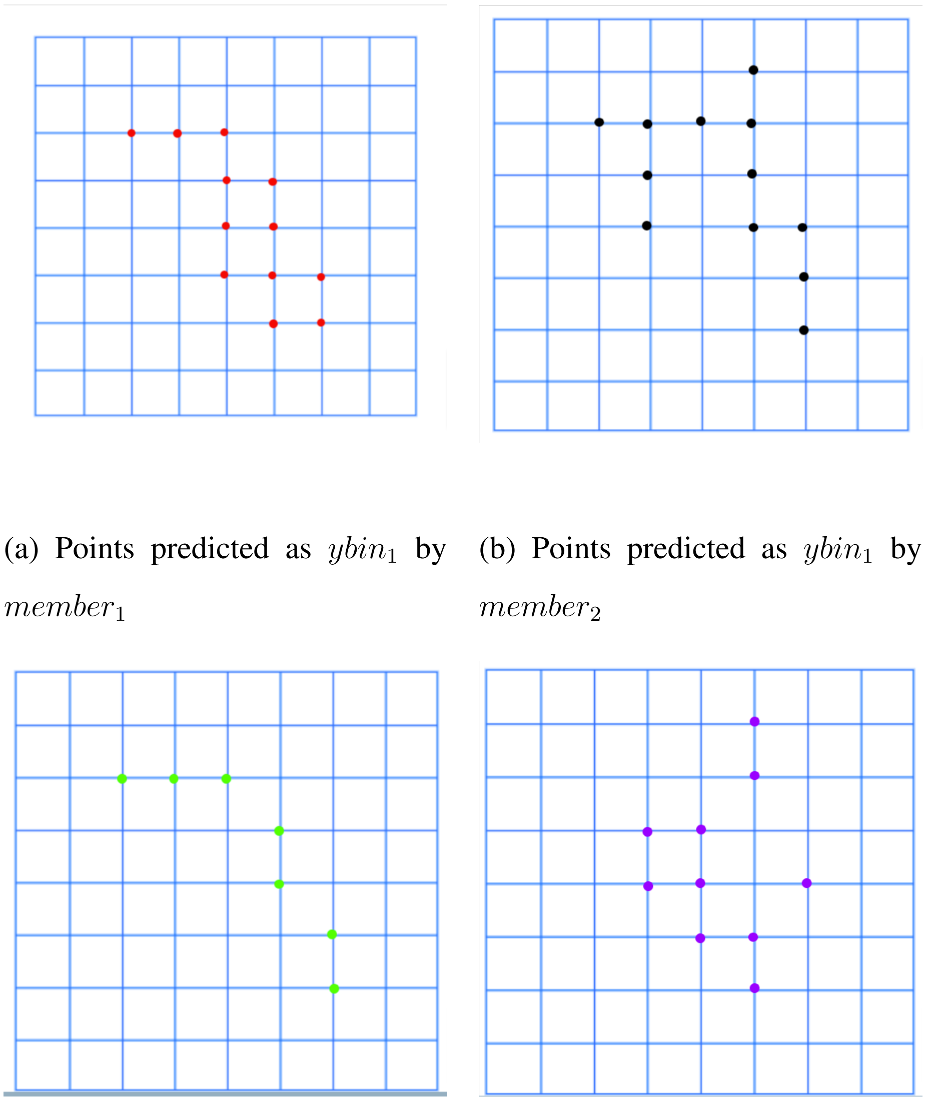


Figure 7.1: Pictorial representation of Input Space Disparity Algorithm

In each iteration, we train a committee of models using non-overlapping subsets of training data available up till that iteration, similar to QBC(Query by Committee). Now we consider the whole input space as a test set and find the predictions for each point in the test set for each committee member. Now we fix a ybin (say $ybin_i$) and consider the sets of points that were evaluated as $ybin_i$ by the members, Lets call them P_i^1 P_i^2 P_i^3 etc. Now, we calculate the 'disparity' across these sets as described by the formula below. We query more points from ybins that have larger disparity score. Disparity gives us a sense of dissimilarity among the sets. By repeating this process for each ybin, we end up with disparity scores for each ybin. The core idea is that we must sample more points from those ybins that are having conflicts between the committee members over what points belonged to those ybins. Refer to Figure 7.1 for a pictorial explanation. If m is the number of members in the committee, We define Disparity Score for $ybin_i$ as described below.



(a) Points predicted as $ybin_1$ by $member_1$

(b) Points predicted as $ybin_1$ by $member_2$

(c) Intersection Map for $ybin_1$

(d) Disparity Map for of $ybin_1$.

The above figure is an illustration for generating the disparity score for $ybin_1$. Here, we have shown in (a) the points in the input space that were predicted to belong to $ybin_1$ by committee $member_1$. In (b) we show the points in the input space that were predicted to belong to $ybin_1$ by committee $member_2$. We get a disparity map of these members in (d). Now we condense this disparity map into a disparity score using a suitable method as described by the formula. If both members predict similar points as $ybin_1$, the disparity map will contain negligible number of points and hence give a small disparity score. On the other hand, if the committee members do not agree on the input space location of points belonging to $ybin_1$, they will give completely different points as predictions for $ybin_1$ and therefore will have a large number of points in the disparity map which leads to a large disparity score.

$$Disparity(i) = 1 - \frac{| \cap_{j=1}^{m} P_i^j |}{| \cup_{j=1}^{m} P_i^j |}$$

Algorithm 8 Input Space Disparity

```
1: Generate a sample $T$ of size $b$
2: **while** instance creation resources exist and stopping criteria not met **do**
3:     Generate $N$ synthetic random samples $P_1$, $P_2$ ... $P_N$
4:     For $i\epsilon\{1, 2, ..k\}$ and $j\epsilon\{1, 2, ..M\}$ Initialize $S_i^j$ = {}
5:     Divide the current train dataset $T$ into $k$ equal parts : $T_1$, $T_2$ ...$T_k$
6:     **for** $f$ = 1 to k **do**
7:         Build Regression Model $R_f$ from $T_f$
8:         **for** all instances $P_1$, $P_2$ ..$P_i$. $P_N$ **do**
9:             $label_f^i$ := $R_f(P_i)$
10:            Add instance $P_i$ to $S_f^{label_f^i}$
11:        **end for**
12:    **end for**
13:    Define $disparity[i]$ = 1 - $\frac{|\cap_{f=1}^{k} S_f^i|}{|\cup_{f=1}^{k} S_f^i|}$ for $i\epsilon 1, 2..M$
14:    $T_{new}$ := sample of size b where the number of instances for class c is computed
       as $N[c] := \frac{disparity[c]}{\sum_{i=1} M disparity[i]}$
15:    $T := T + T_{new}$
16: **end while**
```

7.1 Weighted Input Space Disparity

In the description of the method, we wish to evaluate the committee members on all of the input space points. However this is not feasible as the number of dimensions in the input space increases. Hence, we need to estimate the disparity score of the ybins using an alternative method. We wish to bootstrap an estimate of the disparity scores. To do this, instead of considering the whole input space as the test set, we uniformly select a few points in the input space and treat it as a representative of the whole input space. As we increase the number of points, the disparity scores calculated using this method approaches the actual disparity scores.

We just described that we will randomly and uniformly select points from the input

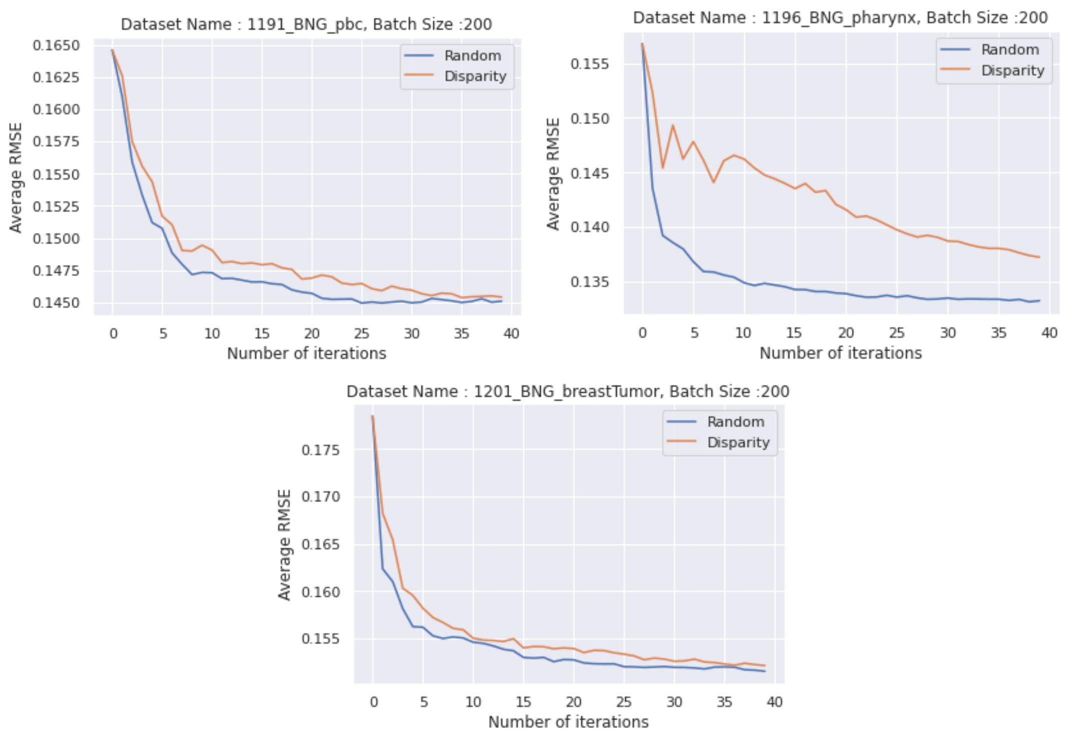


Figure 7.2: Performance of initial Input Space Disparity algorithm

space represent our input space. However, there is a problem with this assumption. The natural occurrences of samples for given task might not be uniformly spread across the input region. We should ideally not care about the model performance beyond the 'naturally occurring region' of the input space. By giving uniform importance to all points in the input space we give equal importance to the performance of the model on all regions of the input space. Hence, to give more importance to the areas that are close to the naturally occurring regions. In other words, instead of generating uniformly and randomly sampled points in the input space, we must try to sample points that actually represent the naturally occurring instances of the task at hand.

One way to achieve this is to give weights to the randomly sampled points. We give higher weights to those points that are close to the actual training set and assign less weight to those points that are far way from the training set points. This can be achieved by checking the average distance to the nearest k neighbours. In this way, we skew the randomly sampled points to act like instances from the original training set. The proximity scores can then be normalized to be finally used in calculating the disparity score.

Algorithm 9 Weighted Input Space Disparity

```
1:  Generate a sample $T$ of size $b$
2:  while instance creation resources exist and stopping criteria not met do
3:      Generate $N$ synthetic random samples $P_1$, $P_2$ ... $P_N$
4:      Compute ProximityScore for each random $P_i$ as $\frac{\sum_{i=1}^{K} Distance(P_i, KNN_i)}{K}$
5:      Standardize the ProximityScores by dividing ProximityScores by
    $\max(ProximityScores)$ to get Weights $w_i, i\epsilon\{1, N\}$
6:      For $i\epsilon\{1,2,..k\}$ and $j\epsilon\{1,2,..M\}$ Initialize $S_i^j = \{\}$
7:      Divide the current train dataset $T$ into $k$ equal parts : $T_1$, $T_2$ ...$T_k$
8:      for $f$ = 1 to k do
9:          Build Regression Model $R_f$ from $T_f$
10:         for all instances $P_1$, $P_2$ ..$P_i$. $P_N$ do
11:             $label_f^i := R_f(P_i)$
12:             Add instance $P_i$ to $S_f^{label_f^i}$
13:         end for
14:     end for
15:     Define $IntersectionSet_i = \{\cap_{f=1}^{k} S_f^i\}$
16:     Define $UninonSet_i = \{\cup_{f=1}^{k} S_f^i\}$
17:     Define $disparity[i]$ = 1 - $\frac{\sum_{j\epsilon IntersectionSet_i} w_j}{\sum_{j\epsilon UninonSet_i} w_j}$ for $i\epsilon 1, 2..M$
18:     $T_{new}$ := sample of size b where the number of instances for class c is computed
    as $N[c] := \frac{disparity[c]}{\sum_{i=1} Mdisparity[i]}$
19:     $T := T + T_{new}$
20: end while
```

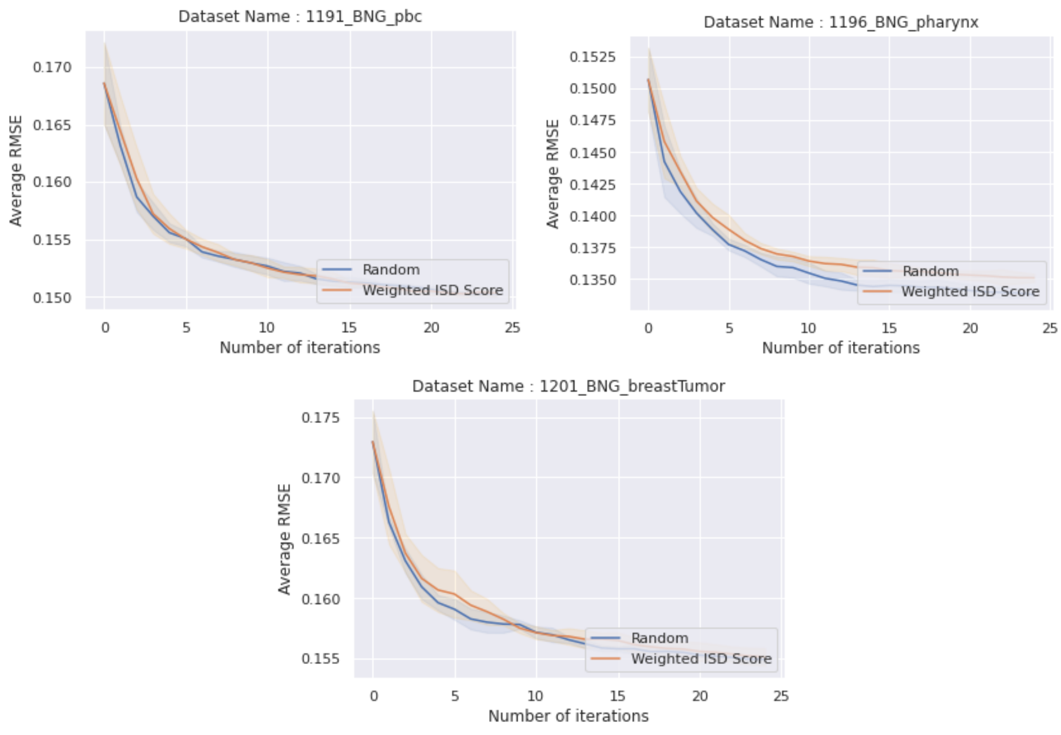


Figure 7.3: Performance of Weighted Input Space Disparity algorithm

7.2 The Final Algorithm

In this section, we describe a few final modifications to the input space disparity algorithm.

Firstly, we observed that the disparity scores for most classes were extremely close, leading to a very similar request amount for most classes. We found that the intersection counts while computing the disparity scores were very low. A low intersection count meant that the models were often vastly disagreeing on the 'understanding' of the classes. There could be two reasons for this, i.e., the models were not well trained, or the samples used for evaluating the models were from a different distribution compared to the training set distribution. We quickly figured out the latter was the problem as the train RMSE was relatively low.

To fix the issue of low intersection counts, we tried to improve the similarity of the random samples generated while evaluating the disparity score. We attempted to use CTGAN (CTGAN user guide), a framework capable of generating pseudo samples similar to a given set of training samples. We also increased the total number of samples generated to estimate the disparity score better. However, this still did not fix the issue.

We observed that random samples generated were highly spread out on the input space due to the curse of dimensionality. This meant that we needed to reduce the number of input features to the model. Hence, we had to restrict ourselves to the datasets with a low number of features (1 to ~ 8). However, to utilize our existing datasets, we created multiple smaller datasets by only considering a subset of features at a time.

We also found that the computation of the disparity score can be improved by changing the score a bit.

$$Disparity(i) = \frac{|\cup_{j=1}^{m} P_i^j|}{|\cap_{j=1}^{m} P_i^j|}$$

On a separate observation, we divided the datasets based on the distribution of y-variable into three categories : a) Normal Distribution, b) Skewed Distribution, and c) Clustered Distribution. The performance shown below is only applicable for the datasets belonging to (a) category.

We also override the algorithm to provide a better initialization by passing original class proportion based samples. In the later iterations we also add a percentage of samples according to original class proportion in order to prevent the algorithm from heavily deviating from the original class proportions.In the final algorithm we did not use the support of CTGAN samples. Finally, we carefully make sure that we never consume any extra samples as compared to the baseline to keep the performance evaluation fair.

Algorithm 10 Input Space Disparity -Final Algorithm

```
1: Generate a sample T of size $b$
2: while instance creation resources exist and stopping criteria not met do
3:     Generate $N$ synthetic random samples $P_1$, $P_2$ ... $P_N$
4:     For $i \epsilon \{1, 2, ..k\}$ and $j \epsilon \{1, 2, ..M\}$ Initialize $S_i^j = \{\}$
5:     Divide the current train dataset $T$ into $k$ equal parts : $T_1$, $T_2$ ...$T_k$
6:     for $f$ = 1 to k do
7:         Build Regression Model $R_f$ from $T_f$
8:         for all instances $P_1$, $P_2$ ..$P_i$. $P_N$ do
9:             $label_f^i := R_f(P_i)$
10:            Add instance $P_i$ to $S_f^{label_f^i}$
11:        end for
12:    end for
13:    Define $disparity[i] = \frac{|\cup_{f=1}^k S_f^i|}{|\cap_{f=1}^k S_f^i|}$ for $i \epsilon 1, 2..M$
14:    $T_{new}$ := sample of size b where the number of instances for class c is computed
       as $N[c] := \frac{disparity[c]}{\sum_{i=1}^M disparity[i]}$
15:    $T := T + T_{new}$
16: end while
```

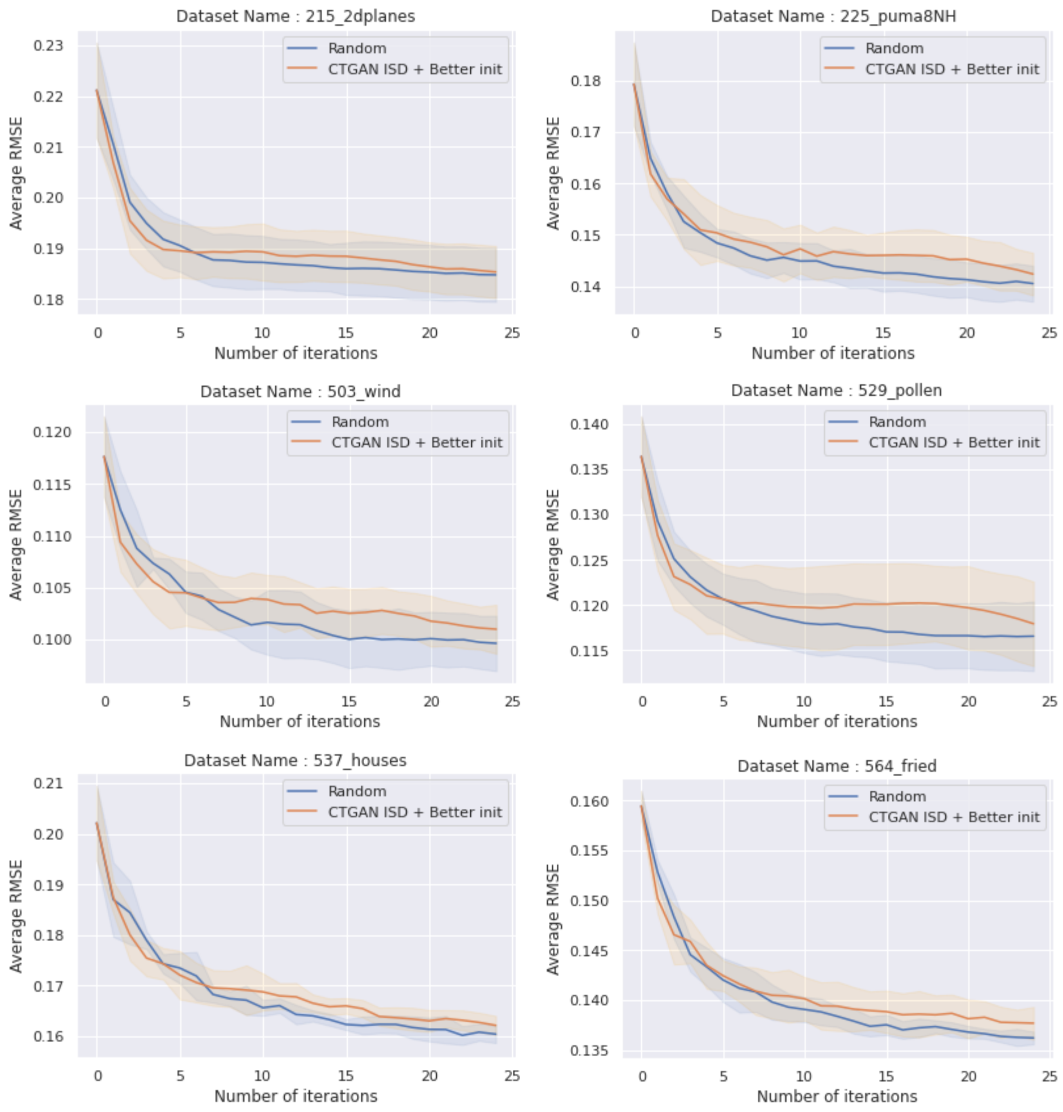


Figure 7.4: Performance of the final Input Space Disparity algorithm on the 'Normal' Datasets (Part-1)

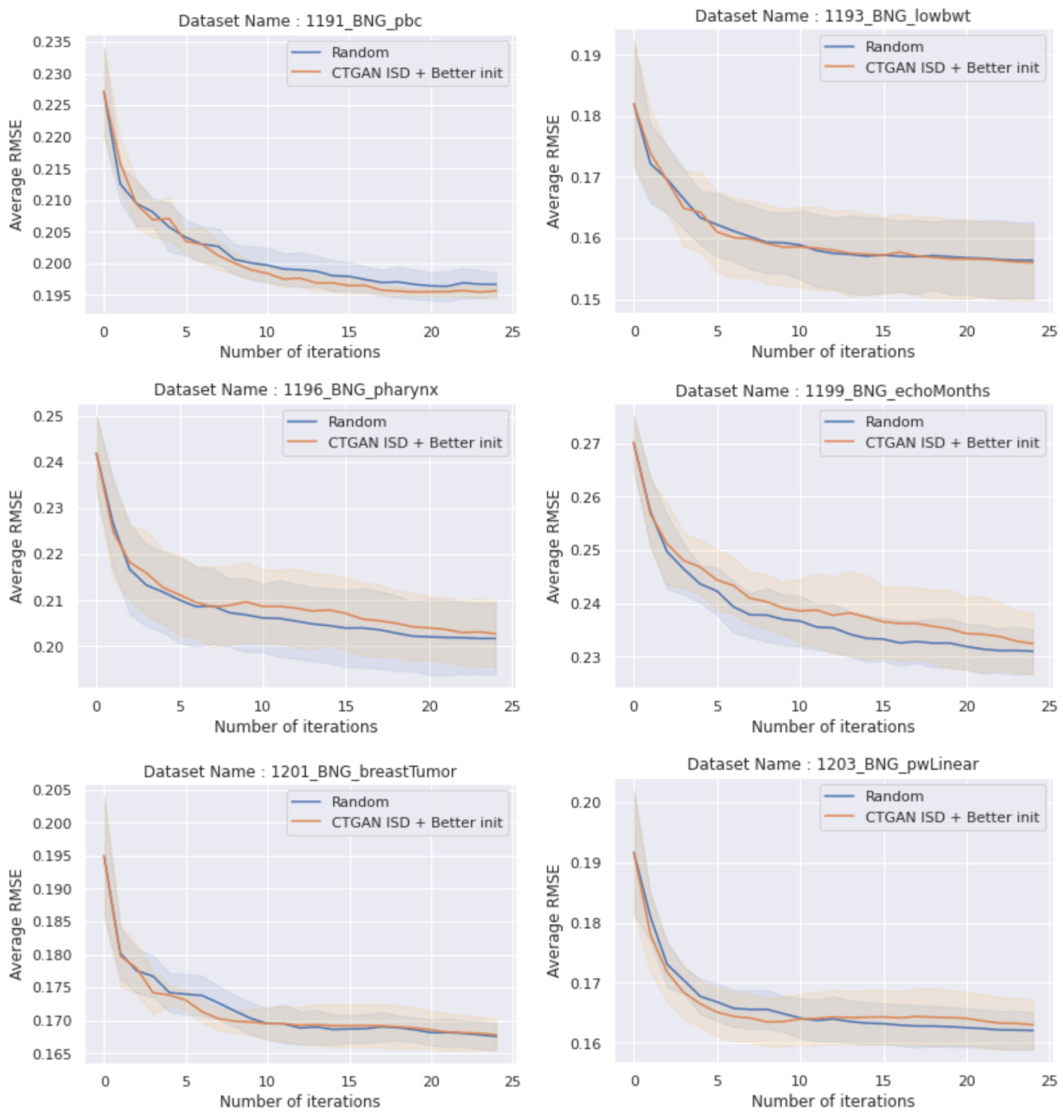


Figure 7.5: Performance of the final Input Space Disparity algorithm on the 'Normal' Datasets (Part-2)

CHAPTER 8

CONCLUSION AND FUTURE WORK

In this project, we attempted to build an active class selection model for regression. We achieved better performance than the baseline on a few iterations on datasets with a standard spread in the y distribution and a limited number of features. However, there is still a long way to go to design an algorithm that supports all types of datasets irrespective of their y distribution or the number of features. We note a few improvements aspects to conclude this report :

1. In the ISD algorithm, we have used binning to classify the random samples into classes before calculating the disparity scores. More 'softer' methods should be employed to estimate the disparity score better. Similarly, most of the algorithms we have proposed above for the Regression task relied on cutting up the y-range to convert it into a Classification problem. More time must be invested in looking for algorithms that act directly on the samples instead of binning.

2. We need theatrical work to identify complementary scores to the Disparity Score in the information domain. We can combine them to get better performance, as suggested by the work on Active Regression Bullard *et al.* (2018). Also, the idea of Input Space Disparity is yet shown substantial performance on regression; it might be better suited for the task of classification, where there is a clear distinction between the classes.

3. As identified in recent papers, we must aim to combine ACS learning techniques with other learning techniques such as AL, Reinforcement Learning, continuous learning, etc.

4. Recent papers on ACS have an observation that the random strategies "proportional" and "uniform" perform highly competitively. Moreover, they come for free, whereas the informed (i.e., non-random) strategies imply an unavoidable computational overhead that needs to be justified with the data acquisition cost. In our report, we observed that random sampling baseline was tough to beat. The reason is that random sampling ensures that the samples remain independent and identically distributed. It is also perfect for maintaining the same distribution in the train set as the actual distribution.

5. In recent papers, the focus has been collecting samples from particular classes as requested by the ACS algorithms. We must explore a different way of collecting information to feed into the model. For example, consider asking users, "Can you give me an instance farthest from the ones you gave me earlier that belongs to the same class?". Different kinds of queries would require different effort and time to be generated; algorithms need to be designed by keeping these factors in mind.

Also, by doing a literature review, we arrive at a general idea that we can try to adapt AL methods by designing algorithms in the reverse direction using pseudo samples whenever we do not have the actual data.

6. In this report, we have presented the performance of the new algorithms using a plot of performance against a baseline. Analyzing the performance by looking at the plot is sometimes challenging when the performances are close by or if they perform differently on a different portion of the iterations. Hence, a mathematical way of understanding the performance needs to be developed.

7. We note that the presence of outliers in the datasets can meddle with the binning strategies or the ACS logic by skewing the classes towards the extreme. Hence, efforts must be employed to avoid collecting outliers. Gradient-based methods must be designed to take advantage of the random samples generated without actual data. One advantage of gradient-based methods is that they are naturally resistant to outliers, as explained in the paper Bullard *et al.* (2018).

8. We must design a "relaxed - ACS" field, as suggested in the paper Bunse and Morik (2019) , where we are allowed to have better control over the sample generation process - either by controlling some of the input features or their distribution.

9. We must explore the possibility of having an initial validation set, contrary to a validation that is used at the end of the training phase in ML. This initial validation set could be used to choose hyperparameters for the ACS algorithm. It can also be used to estimate the classes' relative difficulties and do further analysis that might lead to better initial performance. However, the cost of obtaining such an initial validation set must be incorporated into the algorithm.

10. Finally, suppose we cannot find a universally better performing algorithm (compared to the random baseline) on all datasets. In that case, we can consider a particular problem statement and try to beat the baseline on that particular problem statement. For example, BCI-based ACS Regression tasks that require user-level finetuning for every user. Applying ACS to areas of calibration, where data is collected separately for a different user, would ensure that the advantage we obtained by designing the ACS algorithm does not go to waste after a single round of data collection.

References : Lomasky *et al.* (2007), Bunse and Morik (2019), Kottke *et al.* (2016), Bunse *et al.* (2019), Wu and Parsons (2011*a*), Wu and Parsons (2011*b*), Wu *et al.* (2013), Wu *et al.* (2018), Settles (2010), Hossain *et al.* (2017), Bullard *et al.* (2018), Romano *et al.* (2020), Torgo and Gama (1996)

REFERENCES

1. **Bullard, K.**, **A. L. Thomaz**, and **S. Chernova**, Towards intelligent arbitration of diverse active learning queries. *In 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2018.

2. **Bunse, M.** and **K. Morik**, What can we expect from active class selection? *In* **R. Jäschke** and **M. Weidlich** (eds.), *Lernen, Wissen, Daten, Analysen (LWDA) conference proceedings*. Berlin, Heidelberg, 2019. ISBN 978-3-540-74958-5.

3. **Bunse, M.**, **K. Morik**, **D. Weichert**, and **A. Kister**, Optimal probabilistic classification in active class selection. *In International Conference on Data Mining (ICDM)*. IEEE, 2019.

4. **Hossain, I.**, **A. Khosravi**, and **S. Nahavandi**, Weighted informative inverse active class selection for motor imagery brain computer interface. *In 2017 IEEE 30th Canadian Conference on Electrical and Computer Engineering (CCECE)*. 2017.

5. **Kottke, D.**, **G. Krempl**, **M. Stecklina**, **C. Styp von Rekowski**, **T. Sabsch**, **T. Pham Minh**, **M. Deliano**, **M. Spiliopoulou**, and **B. Sick**, Probabilistic active learning for active class selection. *In* **K. Mathewson**, **K. Subramanian**, and **R. Loftin** (eds.), *Proc. of the NIPS Workshop on the Future of Interactive Learning Machines*. 2016.

6. **Lomasky, R.**, **C. E. Brodley**, **M. Aernecke**, **D. Walt**, and **M. Friedl**, Active class selection. *In* **J. N. Kok**, **J. Koronacki**, **R. L. d. Mantaras**, **S. Matwin**, **D. Mladenič**, and **A. Skowron** (eds.), *Machine Learning: ECML 2007*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. ISBN 978-3-540-74958-5.

7. **Romano, J. D.**, **T. T. Le**, **W. La Cava**, **J. T. Gregg**, **D. J. Goldberg**, **N. L. Ray**, **P. Chakraborty**, **D. Himmelstein**, **W. Fu**, and **J. H. Moore** (2020). Pmlb v1.0: An open source dataset collection for benchmarking machine learning methods. URL `https://arxiv.org/abs/2012.00058`.

8. **Settles, B.** (2010). Active learning literature survey.

9. **Torgo, L.** and **J. Gama**, Regression by classification. *In* **D. L. Borges** and **C. A. A. Kaestner** (eds.), *Advances in Artificial Intelligence*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996. ISBN 978-3-540-70742-4.

10. **Wu, D.**, **B. Lance**, and **T. Parsons** (2013). Collaborative filtering for brain-computer interaction using transfer learning and active class selection. *PloS one*, **8**, e56624.

11. **Wu, D.**, **C.-T. Lin**, and **J. Huang** (2018). Active learning for regression using greedy sampling. *Information Sciences*, **474**.

12. **Wu, D.** and **T. D. Parsons**, Active class selection for arousal classification. *In* **S. D'Mello**, **A. Graesser**, **B. Schuller**, and **J.-C. Martin** (eds.), *Affective Computing and Intelligent Interaction*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011*a*. ISBN 978-3-642-24571-8.

13. **Wu, D.** and **T. D. Parsons**, Inductive transfer learning for handling individual differences in affective computing. *In* **S. D'Mello**, **A. Graesser**, **B. Schuller**, and **J.-C. Martin** (eds.), *Affective Computing and Intelligent Interaction*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011*b*. ISBN 978-3-642-24571-8.