A divide-and-conquer approach to ordinal classification

RQ: To what extent is a divide-and-conquer algorithm applicable to solving the problem of ordinal classification with a binary classifier in terms of time complexity and performance?

A Computer Science Extended Essay

Word count: 3986

Contents

1 Introduction **2**

2 Multiclass classification algorithms **2**

2.1 One-versus-rest 2

2.2 A simple approach to ordinal classification 3

2.3 Divide-and-conquer algorithm 4

3 Time complexities **5**

3.1 Time complexities of the one-versus-rest and SAOC 6

3.2 Time complexity of the divide-and-conquer algorithm 7

4 Performance comparison **9**

4.1 Methodology 9

4.1.1 Datasets 11

4.1.2 Performance metric 12

4.1.3 Control Variables 13

4.2 Experimental results 14

4.3 Possible sources of error 15

4.3.1 Selection bias 15

4.3.2 Balanced and unbalanced datasets 15

5 Balance within the tree of classifiers **17**

6 Conclusion **19**

6.1 Implications and further research 19

7 References **21**

8 Appendix - code **23**

1 Introduction

Classification a machine learning problem concerned with choosing a class to which an input belongs. While today's state-of-the-art algorithms designed to solve hard classification problems mainly incorporate neural networks, traditional machine learning algorithms can sometimes be used just as - or even more - effectively on simpler problems: according to the "No Free Lunch Theorem" first presented in a 1996 paper by David Wolpert[1], there isn't a single model that performs better than any other for every existing machine learning problem. This makes simpler algorithms, such as **logistic regression**, a viable alternative to newer and more complex algorithms.

Logistic regression is a **binary classifier**, meaning that it can only produce two outputs: "true" or "false" (in fact, it outputs a **score** - the probability that the input belongs to the **positive class**, as opposed to the **negative class**), and so without any modifications it cannot be used to solve the problem of **multiclass classification**, where it has to deal with more than two possible classes. However, there are algorithms that can be used to perform multiclass classification with only binary classifiers.

2 Multiclass classification algorithms

2.1 One-versus-rest

The **one-versus-rest** (or one-versus-all) algorithm uses a binary classifier to solve multiclass classification problems in the following way: for a problem with k possible classes: $V_0, V_1, ..., V_{k-1}$, it trains k classifiers, i-th of which predicts the probability that an input belongs to class V_i, while all the other $k-1$ classes are agregated into the negative class. The input is then assigned a class, which respective classifier outputs the highest score.

2.2 A simple approach to ordinal classification

For some classification problems the one-versus-rest algorithm can be modified to produce even better results. For example, in 2001, Eibe Frank and Mark Hall wrote a paper titled "A simple approach to ordinal classification" (or **SAOC** for short)[2], where they presented an algorithm for classifying **ordinal variables** – variables "whose value exists on an arbitrary scale where only the relative ordering between different values is significant" [3]. In other words, in an ordinal classification problem classes can be ordered in such a way, that the closer two classes are, the more "similar" they are. An example of an ordinal classification problem is the problem of rating: if two items have similar ratings, they are more likely to be of similar quality than two items whose ratings vary significantly.

SAOC consists of training $k-1$ classifiers, where i-th classifier predicts the probability of an input belonging to one of the classes in the range $[V_{i+1}; V_{k-1}]$ (the classes in the range $[V_0; V_i]$ are all considered a part of the negative class). The probability $Pr(i)$ that a new **sample** (an input that the algorithm has to classify) belongs to class V_i is then found using the following algorithm, if S_i is the score given to the input by i-th classifier:

$$\begin{aligned} &Pr(0) = 1 - S_0 \\ &Pr(i) = S_{i-1} - S_i, 0 < i < k-1 \\ &Pr(k-1) = S_{k-2} \end{aligned} \tag{1}$$

The class with the highest probability $Pr(i)$ is then selected. Making use of a dataset's ordinal structure, this method usually outperforms the standard one-versus-rest algorithm.

This Extended Essay aims to investigate the extent to which a different algorithm, based on the **divide-and-conquer** approach can be considered an alternative to both one-versus-rest and SAOC algorithms.

2.3 Divide-and-conquer algorithm

In computer science, a divide-and-conquer algorithm is an algorithm that recursively breaks the problems down into smaller subproblems that are easier to solve, solves these subproblems and then "stitches" the results to get the solution for the bigger problem[4]. This approach can be applied to the problem of ordinal classification by building a **binary search tree**, every **node** of which stores a classifier. The algorithm of building the tree is as follows:

1. The binary classifier in the root of the tree operates on the entire class range, meaning that it is trained on all available samples. It considers all classes in the range $[V_0; V_{[\frac{k-1}{2}]}]$ a part of the negative class, and all classes in the range $[V_{[\frac{k+1}{2}]}; V_{k-1}]$ - a part of the positive class, breaking the class range in two smaller and roughly equal ranges.

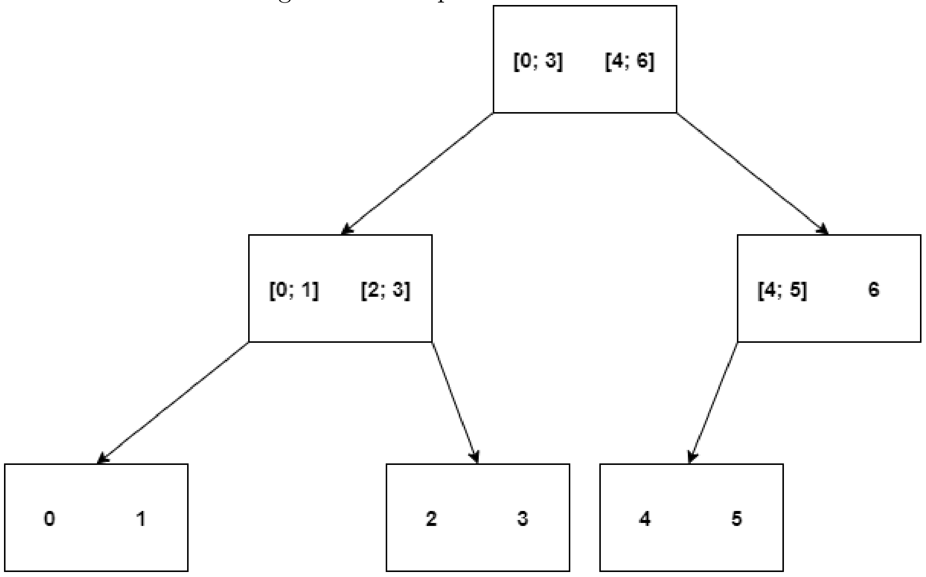
2. The tree is built recursively in such a way that each non-leaf node has one or two **children**. If a parent node operates on the range $[V_i; V_j]$, it considers classes in the range $[V_i; V_{[\frac{i+j}{2}]}]$ a part of the negative class, and those in the range $[V_{[\frac{i+j}{2}]+1}; V_j]$ - a part of the positive class. Its left child then operates on all the classes its parent aggregates into the negative class($[V_i; V_{[\frac{i+j}{2}]}]$), and its right child operates on all the initial classes aggregated into the positive class ($[V_{[\frac{i+j}{2}]+1}; V_j]$). Similarly to the parent node, the negative and positive classes are formed for each of the child nodes and so on. Thus, the number of classifiers in each consequtive level doubles, and the range a classifier operates on is cut in two.

3. Nodes, the classifiers in which only operate on the range of one class, are not created, since classification with only one possible class is impossible.

Based on the previous condition, it can be seen that nodes that operate on 3 classes only have a left child, and those operating on 2 classes are leaf nodes that have no children.

An example of such tree for $k = 7$ can be seen on the following diagram:

Figure 1: Example tree of classifiers



Here, the ranges in the left and right parts of each node represent the class ranges aggregated into the negative and positive classes for the the classifier in the node.

3 Time complexities

In machine learning, time is an important parameter: some algorithms, while giving good results, can be unusable simply because they take too long to run. This section explains the difference in time efficiency of making predictions for the three multiclass classification algorithms presented in the previous section.

Assume that each sample to be classified has n **features** (variables that define the sample), and at once the algorithm needs to process m samples. This way, the data to be processed can be represented by a matrix X with dimensions $m \times n$, where $X_{i,j}$ equals the value of j-th feature of the i-th sample to be classified.

3.1 Time complexities of the one-versus-rest and SAOC

In a typical one-versus-rest classifier, the results of predictions are obtained by multiplying the matrix X and the matrix θ with dimensions $n \times k$, where $\theta_{i,j}$ equals the **weight** by which i-th feature is multiplied in the binary classifier corresponding to the class j (the significance of the weights is not relevant to the Extended Essay). The outcome of the multiplication is a $m \times k$ matrix Y, where $Y_{i,j}$ represents the score given to the i-th sample by the j-th classifier (in fact, for logistic regression as well as other classification algorithms the process is more complicated, but it is also not relevant to the Extended Essay, and the functions omited here do not impact time complexity). The final prediction for i-th sample can then be obtained by finding $\text{argmax}_j Y_{i,j}$ - the index of a class with the highest score - for each sample. The time complexity of multiplying 2 matrices with dimensions $m \times n$ and $n \times k$ is $O(mnk)$, meaning that the time complexity of the entire algorithm is also $O(mnk)$, since finding the indices of maximal scores in each row has a time complexity of $O(mk)$.

Similarly, in the case of SAOC, θ has the dimensions $n \times k - 1$, since there are $k - 1$ classifiers, so the time complexity of acquiring Y is $O(mn(k - 1))$ which is the same as $O(mnk)$. The final probabilities for each class can be computed in one equation as $[\vec{1}|Y] - [Y|\vec{0}]$, where $[A|B]$ is the operation of matrix concatenation. It can be easily seen that this operation is identical to the one presented in Equation 1. Like in the case of the one-versus-rest classifier, the predictions can be found using $\text{argmax}_j Pr_{i,j}$, where Pr is the final matrix of probabilities. Since both vector-matrix concatenation and matrix subtraction can be performed in no more than $O(mk)$ operations, the final time complexity is, once again, $O(mnk)$.

3.2 Time complexity of the divide-and-conquer algorithm

For the divide and conquer algorithm, however, we don't need to use $O(k)$ classifiers for each sample. Predictions can be obtained by applying a classifier in the root node of the tree to X, and dividing X into 2 matrices X_l and X_r in such a way that all samples for which the negative class was predicted are a part of X_l, and those for which the positive class was predicted are a part of X_r. X_l is then passed to the left child node, and X_r - to the right child node. The process is repeated recursively, so that each sample follows a path from the root to one of the leaf nodes. Applying the classifier in one of the nodes of the binary search tree to the matrix X has a time complexity of $O(2mn) = O(mn)$, since in this case there is a classification with only two classes: the positive and the negative one, so $k = 2$.

Assume there are s nodes in a level of the binary tree, and i-th node has to classify m_i samples. Then the time complexity of one node's prediction is $O(m_i n)$. After classification in a node, the same sample can't be passed on to both its left and right child, meaning that on each level every sample from the initial matrix X is classified by exactly one node, and so $\sum_{i=0}^{s-1} m_i = m$. The time complexity of classifying all the samples on this level is, therefore, $O(m_0 n + m_1 n + ... + m_{s-1} n) = O(mn)$. Since the structure of the tree of classifiers is identical to a binary search tree built on an array of size k, the height of this tree of classifiers is also $O(log(k))$, meaning that the overall time complexity of the prediction algorithm is $O(mnlog(k))$ as opposed to $O(mnk)$ for the 2 other algorithms. This means that the divide-and-conquer algorithm scales better with the number of classes, and with a large k it can make predictions significantly more efficiently than both other algorithms.

The intuition behind the lower time complexity is that while one-versus-rest and SAOC act as **linear search**, where each sample is evaluated on every classifier, the divide-and-conquer approach can be seen as a generalization of **binary search**, where, instead of using a **comparator** - a function used to compare the input with an element of a collection, we use a classification algorithm to determine whether a sample should be propagated to the left or right part of the collection.

A big weakness of the divide-and-conquer approach, however, is its inability to be efficiently parallelized: the classifier in any node cannot be engaged in the prediction process before its parent node finishes making predictions, since only then the matrices X_l and X_r used by the classifiers in the child nodes are acquired. In the other algorithms, all binary classifiers are independent from each other, and so can make their predictions in parallel. One solution for overcoming this weakness of the divide-and-conquer algorithm is parallelizing classifiers within one level of the binary tree, since they are independent of each other and can start making predictions at once after their parent nodes have completed their respective computations. This way, on the first level of the tree, only one thread is engaged, since there is only one node, but then this number growth exponentially, with 2 nodes on the second level, 4 - on the third and so on. Considering k is large enough, after a few more levels the resources of the system will be used efficiently.

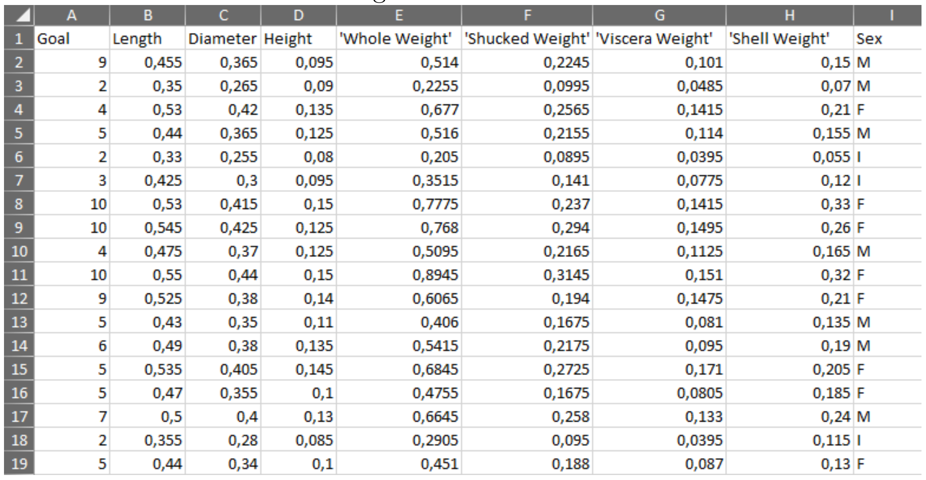
4 Performance comparison

Even though time efficiency is an important parameter that determines whether an algorithm should be applied to a given classification problem, a slower algorithm that can provide accurate classification is sometimes a better solution than a faster algorithm that often makes mistakes and misclassifies a large portion of samples. This section aims to compare the **performance** that the three algorithms yield when dealing with new data.

4.1 Methodology

To see how accurate the results produced by the algorithms are, it is necessary to evaluate them on various **datasets** with a **performance metric**. A dataset shares a similar structure with a database: it consists of a set of records, each representing one sample, with each field storing a feature of the sample. For example, below is a dataset storing features of abalones:

Figure 2: A dataset



	A	B	C	D	E	F	G	H	I
1	Goal	Length	Diameter	Height	'Whole Weight'	'Shucked Weight'	'Viscera Weight'	'Shell Weight'	Sex
2	9	0,455	0,365	0,095	0,514	0,2245	0,101	0,15	M
3	2	0,35	0,265	0,09	0,2255	0,0995	0,0485	0,07	M
4	4	0,53	0,42	0,135	0,677	0,2565	0,1415	0,21	F
5	5	0,44	0,365	0,125	0,516	0,2155	0,114	0,155	M
6	2	0,33	0,255	0,08	0,205	0,0895	0,0395	0,055	I
7	3	0,425	0,3	0,095	0,3515	0,141	0,0775	0,12	I
8	10	0,53	0,415	0,15	0,7775	0,237	0,1415	0,33	F
9	10	0,545	0,425	0,125	0,768	0,294	0,1495	0,26	F
10	4	0,475	0,37	0,125	0,5095	0,2165	0,1125	0,165	M
11	10	0,55	0,44	0,15	0,8945	0,3145	0,151	0,32	F
12	9	0,525	0,38	0,14	0,6065	0,194	0,1475	0,21	F
13	5	0,43	0,35	0,11	0,406	0,1675	0,081	0,135	M
14	6	0,49	0,38	0,135	0,5415	0,2175	0,095	0,19	M
15	5	0,535	0,405	0,145	0,6845	0,2725	0,171	0,205	F
16	5	0,47	0,355	0,1	0,4755	0,1675	0,0805	0,185	F
17	7	0,5	0,4	0,13	0,6645	0,258	0,133	0,24	M
18	2	0,355	0,28	0,085	0,2905	0,095	0,0395	0,115	I
19	5	0,44	0,34	0,1	0,451	0,188	0,087	0,13	F

The “Goal” column stores the **label** of each sample - its target class.

A performance metric is simply a function that shows how close or far the output of the model is from the true value (the label). The algorithms can be evaluated by selecting multiple ordinal datasets and splitting each of them into a **train set** and a **test set**. Each of the algorithms is trained on the train set, after which it is used to make predictions for samples in the test set without seeing the labels. This division of data into two sets is necessary, because during training, an algorithm learns the data it is given, so if the same data is used to evaluate it later, it can "remember" the correct answer. Therefore, if an algorithm's performance is good on the train data, this doesn't always imply that it has the ability to generalize and produce the same results on data it hasn't seen before - perhaps the good performance comes simply from remembering the answers[5]. This is called the problem of **overfitting**.

By comparing the outputs of the algorithms on the test set to the labels using the performance metric, it is possible to obtain one number that shows how well the algorithm performs on the dataset. In order to decrease the probability of randomness impacting the results of the experiment, the algorithms are evaluated using 10-fold **cross-validation**: the dataset is broken into 10 random parts and each classifier is trained 10 times, each time training on 9 of the parts, leaving one out and using it as a test set[10].The performances of the algorithm during each testing are then averaged to get a final score. This way, a situation where the selected test set is not representative of the entire dataset - and so can't be used for algorithm evaluation, since it doesn't reflect the algorithm's ability to generalize to normal data - becomes less likely. To increase the acuracy of evaluation even further, the process of cross-validation is repeated 10 times, and the scores are once again averaged out. Overall, each algorithm is trained and evaluated 100 times for each dataset.

4.1.1 Datasets

In the 2001 paper, Frank and Hall mention that due to the lack of benchmark datasets for ordinal classification they had to resort to using regression datasets for evaluating their classifier. Apparently, the problem still persists, judging by a recent paper on the application of ordinal classification in transportation[7]. Despite being published in 2019, the paper still vastly relies on regression datasets for evaluation.

In order to transform a regression dataset into an ordinal classification dataset, the authors of both articles use **equal-frequency binning**. This is a discretization technique that, when applied to a dataset with continous labels, breaks the range of labels into k smaller non-intersecting ranges, "bins", such that roughly the same amount of samples are in each bin. By assigning the index of the range in which a label falls as its new label, the regression problem is turned into a classification problem, since now the labels are discrete. If the indices of ranges for 2 samples are close to each other, it means that the initial labels were also relatively close, and so the two samples are "similar", indicating that this is a problem of ordinal classification.

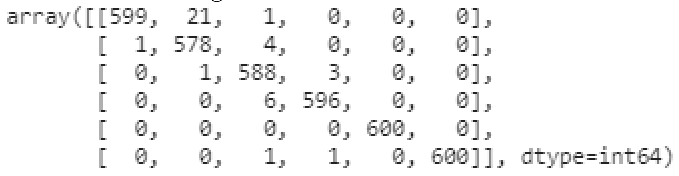
In this Extended Essay, the algorithms are be applied to the same datasets used in the 2001 paper, but some adjustments are made: Unlike the paper, this Extended Essay doesn't compare the performances of the algorithms on datasets split into a different number of bins. Namely, while in the 2001 experiment datasets were split first into 3 bins, then 5 and only then 10, the measurements presented in this Extended Essay are collected only using 10-bin partition. This adjustment was made because for $k = 3$ or $k = 5$ the tree used in the divide-and-conquer algorithm would have a very simple structure (while for $k = 3$ it would only have 2 nodes in total on 2 levels, for $k = 5$ the structure would not be very complex either, with 4 nodes and 3 levels). For $k = 10$, however, the numbers of nodes and levels increases to 10 and 4 respectively, and so the "tree-like" structure of the algorithm is more apparent. Under ideal circumstances, the number of bins would be increased even further, but this is not possible for many of the datasets used, since when k increases it becomes more likely that there are simply fewer different labels than bins in the initial dataset. In this circumstances, equal-frequency binning cannot be performed. The datasets that cannot be broken into 10 ranges by equal-frequency binning were omitted during the experiment.

4.1.2 Performance metric

In the 2001 paper, the algorithms are compared to each other based on their **accuracy** - the fraction of samples, for which the correct class was predicted. However, this is not a perfect performance metric for ordinal classification, since oftentimes we care not only about how many samples are classified correctly, but also about how close the predictions were to the true values. For example, if the task of an algorithm is to rate an input on a scale from 1 to 10, and the true class for a sample is 9, a classifier predicting a 10 is obviously more "correct" than a classifier predicting a 2. In the accuracy metric, however, this degree of "correctness" is ignored, and both classifiers receive the same penalty to their accuracy scores.

This is why it is preferable to use special performance metrics when dealing with ordinal classification. One of these metrics is a special type of a **mean squared error** (MSE)[8]. This error relies on a **confusion matrix** of the test set to evaluate a classifier. The confusion matrix is a $k \times k$ matrix C, such that $C_{i,j}$ equals the amount of samples in the test set that belong to class i but were predicted to belong to class j by the model. For example, below is the confusion matrix for a dataset with 6 classes:

Figure 3: Confusion matrix



```
array([[599,  21,   1,   0,   0,   0],
       [  1, 578,   4,   0,   0,   0],
       [  0,   1, 588,   3,   0,   0],
       [  0,   0,   6, 596,   0,   0],
       [  0,   0,   0,   0, 600,   0],
       [  0,   0,   1,   1,   0, 600]], dtype=int64)
```

The numbers on the main diagonal of the matrix represent the samples that were classified correctly. The MSE equals the mean of the squared distances between the true classes and the predicted classes for each sample in the test set, so its equation is $\frac{1}{m}\sum_{i=0}^{k-1}\sum_{j=0}^{k-1} C_{i,j}(i-j)^2$. It can be seen that the further the true and predicted classes are, the higher the error, which is suitable for ordinal classification. Thus, this error is the one used in the experiment.

4.1.3 Control Variables

In order to ensure that the MSEs of the algorithms are not affected by a change in some parameter unrelated to the algorithms, we keep some variables constant. Mainly, these are:

- **Base classifier**: one-versus-rest, SAOC and divide-and-conquer algorithms all rely on a base binary classifier used to calculate the score that a sample belongs to the positive class. Since different classification algorithms can be used as base, each yielding different performance, it is necessary to choose one base to be used. For evaluation, I decided to use logistic regression due to its simplicity.

- **Implementation of logistic regression and hyperparameters**: the implementation of logistic regression was taken from Python's *scikit-learn* library. The model has a set of **hyperparameters -** parameters that are not learnt by the model but are instead selected by the programmer (e.

g. **regularization parameter** which defines how much the algorithm is “punished” for having large weights). Similar algorithms trained with different hyperparameters often produce different results. To prevent this from impacting the experiment, all hyperparameters were set to default ones provided by the scikit-learn library[6].

- **Hardware**: every model was trained on a laptop with a 2.6GHz Intel Core i7 CPU. Multithreading and GPU acceleration was not used for any algorithm. Ideally, this should have no effect on the performance, but as faulty hardware can cause errors to occur, hardware is not changed throughout the experiment.
- **Number of classes**: as explained previously, in every dataset k was set to 10.

4.2 Experimental results

In the process of testing the algorithms on 8 datasets from the UCI Machine Learning Repository[9] according to the chosed method, the following results were obtained:

	MSE		
Dataset	**One-versus-rest**	**SAOC**	**Divide-and-conquer**
Abalone	4.04	3.83	4.02
Ailerons	7.17	3.18	4.03
Delta Ailerons	4.52	4.42	4.87
Elevators	10.31	7.72	9.18
2D Planes	2.23	1.63	2.02
Friedman Artificial	3.80	2.89	3.16
Kinematics of Robot Arm	7.64	6.36	8.09
Computer Activity	10.31	6.93	8.17

From the table, it can be inferred that in general, the divide-and-conquer algorithm performs better than the one-versus-rest algorithm, but worse than SAOC (lower error indicates better performance). Out of the 8 trials, it outperformed the former in 6 measurements, but it never gave more accurate predictions that the latter. This leads to believe that, in its current state, the divide-and-conquer approach is not as accurate as the classical approach to ordinal classification. However, for certain datasets (especially the Abalone dataset) the divide-and-conquer algorithm performed almost as well as SAOC.

4.3 Possible sources of error

The validity of the results can be impacted by multiple parameters, most of which arise because of the datasets used.

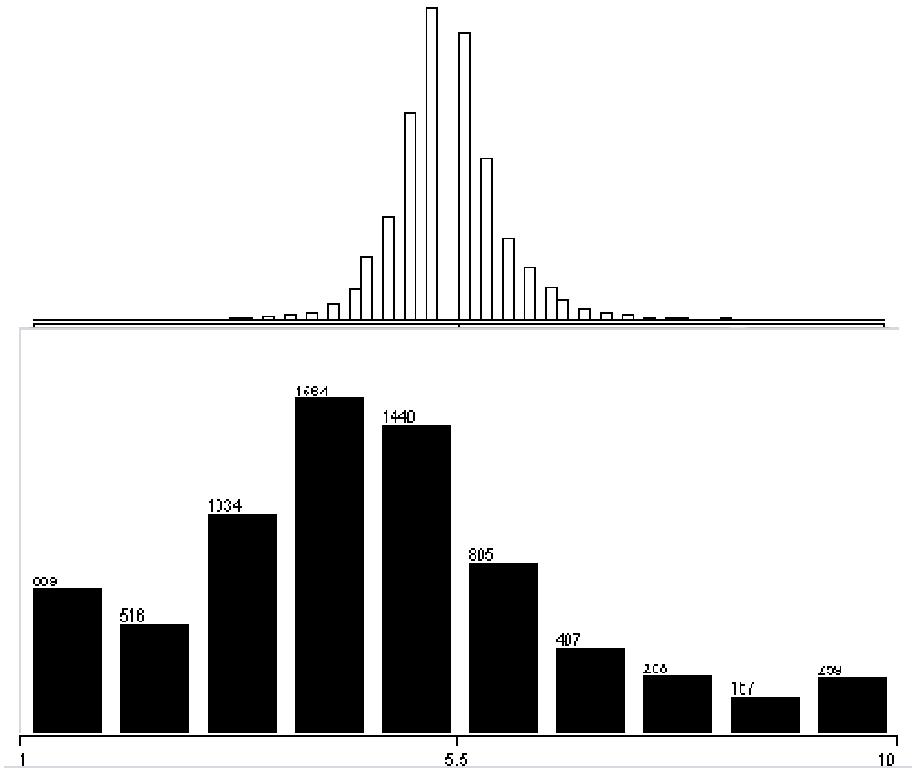
4.3.1 Selection bias

One of the possible sources of error lies in the method of selecting the datasets. After all, these datasets were synthetically discretized, and do not represent a real problem of ordinal classification, meaning that they may not follow the same rules as most real-world ordinal datasets, and so the results can be unrepresentative. Moreover, not all datasets could be split into 10 bins by equal-frequency binning. Some of the datasets in the UCI repository have fewer than 10 different labels, and so were omitted during the experiment. This could lead to a selection bias.

4.3.2 Balanced and unbalanced datasets

The datasets that were selected are not completely homogenous either. Some datasets are more **balanced** than others, meaning that for them, the samples are more equally distributed among the classes, and approximately $\frac{m}{k}$ samples in the dataset belong to each class. However, this is not true for some datasets. For example, below is the distribution diagram for labels of the Delta Ailerons dataset before and after equal-frequency binning is applied to it (this is one of the two datasets on which the divide-and-conquer algorithm performed worse than the one-versus-rest algorithm):

Figure 4: Label distributions for the Delta Ailerons dataset



In the first diagram, the height of a bar represents the number of samples in a fixed size label range. In the second diagram, the height of each bar shows the number of samples in a class after equal-frequency binning. As the initial distribution has most labels concentrated in the several peaks in the center, the equal-frequency binning algorithm fails to assign a similar amount of samples to each class, and the resulting dataset becomes very unbalanced, as seen from the second distribution. This imbalance can negatively impact the performance of classifiers[11].

5 Balance within the tree of classifiers

Imbalance in data impacts the algorithms differently. Indeed, in the case of one-versus-rest classification, when all classes but one are considered a part of a large negative class, the imbalance is $1 : k-1$ even if the dataset is perfectly balanced, as $k-1$ classes are aggregated into one. In SAOC, the classifiers operating at the boundaries of the class range have to deal with data unbalanced to the same degree, but those in the center deal with an almost balanced dataset(e. g. if $k = 10$, for a classifier that determines whether a sample is in the class range $[V_0; V_4]$ or $[V_5; V_9]$ there is an equal number of samples in the positive and negative class, assuming that the initial dataset is balanced). For the divide-and-conquer algorithm, however, all nodes operate on a roughly balanced dataset, where the number of initial classes agreggated into the negative and positive class varies by no more than 1. It can be shown that the nodes that operate on a range of 3 classes have to deal with the most unbalanced data with a $1 : 2$ imbalance. This leads to believe that a divide-and-conquer algorithm might perform better on balanced datasets, but when a dataset is unbalanced, the classifiers are not guaranteed to work with roughly balanced data, and the balancing property of the tree doesn't come into play. Unlike the other two algorithms, the divide-and-conquer approach caps the imbalance at $1 : 2$ and doesn't scale it with k. Based on this, two hypotheses can be formed:

- **Hypothesis 1**: If the initial dataset is balanced, classifiers in the divide-and-conquer algorithm deal with balanced data, while those in the other two algorithms do not, meaning that the divide-and-conquer algorithm benefits more if the dataset is balanced.

- **Hypothesis 2**: As the data balance in the divide-and-conquer algorithm doesn't depend on the number of classes, the algorithm's performance doesn't drop as much as for one-versus-rest and SAOC algorithms with an increase in k

These hypotheses, however, are not supported by the algorithm's poor performance on the Kinematics of Robot Arm dataset, which is almost perfectly balanced after equal-frequency binning, as well as by the algorithm's response to increasing k:

Figure 5: Label distribution for the Kinematics of Robot Arm dataset after equal-frequency binning

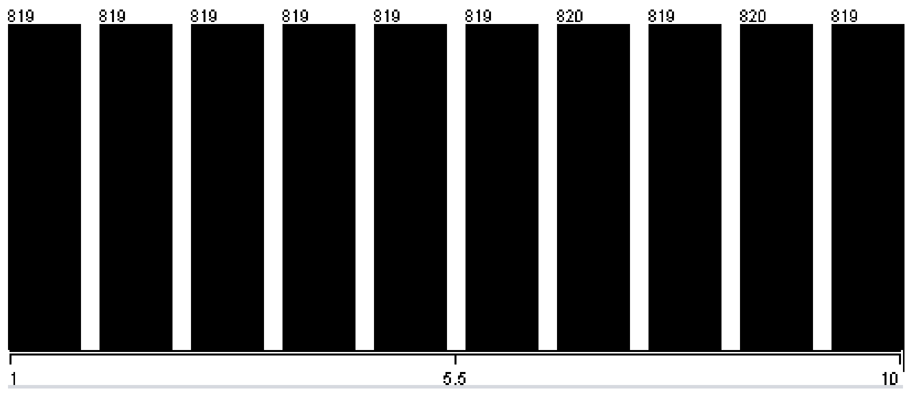
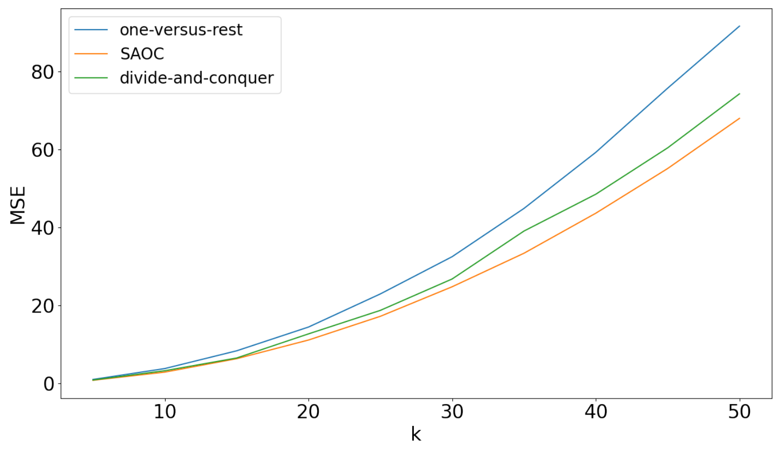


Figure 6: The relationship between k and the error for the algorithms



As k was increased from 5 to 50 in increments of 5, the algorithm reliably outperformed a one-versus-rest classifier, but could not match the results produced by SAOC on the balanced “Friedman Artificial” dataset.

6 Conclusion

While for a problem with a large number of classes the divide-and-conquer approach can yield results faster than the other algorithms, it - at least in its current implementation - significantly loses to SAOC in terms of performance. Perhaps, this tradeoff between speed and performance can be useful in situations when k is large, predictions have to be made quickly, and a slightly higher error can be tolerated.

6.1 Implications and further research

Outside evaluating the usability of a new classification algorithm, this Extended Essay also shows that both algorithms for ordinal classification outperform the traditional one-versus-all classifier, once again indicating that ordinal classification is a separate problem that shouldn't be mixed with other classification problems (like it often is). A possible development of the topic presented in this Extended Essay could include evaluating the divide-and-conquer algorithm based on a binary classifier different from logistic regression, which is not as susceptible to imbalance as some other techniques[12]. It is possible that in tandem with other classifiers the algorithm's balancing ability improves its performance to match that of the standard approaches to ordinal classification. Results of the experiment can also change if another degree of freedom is introduced through **hyperparameter tuning**: if the hyperparameters of a model are not fixed, and instead tuned to find the values which maximize the model's performance, the error of the algorithm is lower. However, thorough search for the best set of hyperparameters can be very time-consuming or require higher computational power[13].

Overall, it is clear that one abstract machine learning algorithm can have many concrete implementations differing in speed in performance, so an Extended Essay can only provide an overview of whether an algorithm is suitable for a tase. A decision of what algorithm should be used is upon the individual programmer, who ought to test as many hopotheses as possible to ensure the solution they find is optimal.

7 References

[1] Wolpert, David H. "The Lack of A Priori Distinctions Between Learning Algorithms." *Neural Computation* 8, no. 7 (1996): 1341–90. https://doi.org/10.1162/neco.1996.8.7.1341

[2] Eibe Frank and Mark Hall. "A Simple Approach to Ordinal Classification." *Machine Learning: ECML 2001 Lecture Notes in Computer Science* ,2001, 145–56. https://doi.org/10.1007/3-540-44795-4_13.

[3] Assagaf, Muhammad. "Simple Trick to Train an Ordinal Regression with Any Classifier." Medium. Towards Data Science, May 14, 2019/ Retreived October 14, 2020 https://towardsdatascience.com/simple-trick-to-train-an-ordinal-regression-with-any-classifier-6911183d2a3c.

[4] Skiena, S. (2008). Divide-and-Conquer. In *The algorithm design manual* (p. 135). New York: Springer. ISBN 978-5-9775-0560-4

[5] Tarang Shah. (2020, July 10). About train, validation and test sets in machine learning. *Towards Data Science*. https://towardsdatascience.com/train-validation-and-test-sets-72cb40cba9e7

[6] sklearn.linear_model.SGDClassifier — scikit-learn 0.24.1 documentation. (n.d.). Retrieved March 9, 2021, from https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html

[7] 'Pelin Yıldırım, Ulaş K. Birant, Derya Birant, "EBOC: Ensemble-Based Ordinal Classification in Transportation", *Journal of Advanced Transportation* vol. 2019, Article ID 7482138, 17 pages, 2019. https://doi.org/10.1155/2019/7482138

[8] Cardoso, J. S., Sousa, R. (2011). Measuring The Performance Of Ordinal Classification. *International Journal of Pattern Recognition and Artificial Intelligence* , 25(08), 1173-1195. doi:10.1142/s0218001411009093

[9] UCI Machine Learning Repository. Accessed October 17, 2020. `https://archive.ics.uci.edu/ml/index.php`.

[10] Geron, A. (2019). Better Evaluation Using Cross-Validation. In *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent* Systems (pp. 76-77). O'Reilly Media, Incorporated. ISBN 978-5-9500296-2-2

[11] Somasundaram, Akila & Reddy, U. Srinivasulu. (2016). Data Imbalance: Effects and Solutions for Classification of Large and Highly Imbalanced Data.

[12] Zheng, Wanwan, and Mingzhe Jin. "The Effects of Class Imbalance and Training Data Size on Classifier Learning: An Empirical Study." *SN Computer Science 1* , no. 2 (2020). `https://doi.org/10.1007/s42979-020-0074-0`.

[13] Koehrsen, W. (2018, July 4). Automated machine learning hyperparameter tuning in python. *Towards Data Science.* https://towardsdatascience.com/automated-machine-learning-hyperparameter-tuning-in-python-dfda59b72f8a

8 Appendix - code

```
import numpy as np

import pandas as pd


from sklearn.metrics import confusion_matrix

from sklearn.metrics import make_scorer

from sklearn.model_selection import cross_val_score as cvs

from sklearn.linear_model import SGDClassifier

from sklearn.preprocessing import OneHotEncoder

from sklearn.base import BaseEstimator, ClassifierMixin

#folder and file from which to fetch the data

folderName = 'abalone'

fileName = 'abalone'

#equal-frequency binning was performed in advance using the Weka

    ↪ software (https://www.cs.waikato.ac.nz/ml/weka/)

data = pd.read_csv("../datasets/{}/{}.csv".format(folderName,

    ↪ fileName))

#the number of classes

k = 10

#getting labels and features from the dataset

y = np.array(data['Goal'])

x = np.array(data.drop('Goal', axis=1))

#one-hot encoding was only used for the Abalone dataset, since it

    ↪  had a categorical feature

ohe = OneHotEncoder()

non_numerical = x[:, 0]

x = np.delete(x, 0, axis=1)
```

```
x = np.append(x, ohe.fit_transform(non_numerical.reshape(-1, 1)).
    ↪ toarray(), axis=1)


#the divide-and-conquer model
class DCModel(BaseEstimator, ClassifierMixin):


    def __init__(self, k):
        self.k = k
        #4 * k + 1 elements are always enough to store the nodes
            ↪ of the tree
        self.nodes = [None] * (4 * k + 1)


    def __fit_node_on_range(self, l, r, X, y):
        #selecting samples the classes for which lie in the range
            ↪ and assigning all classes to either the positive or
            ↪  the negative class
        ind = np.where(np.logical_and(y >= l, y <= r))
        m = (l + r) // 2
        x_in_range = X[ind]
        y_in_range = np.where(y[ind] > m, 1, 0)
        return SGDClassifier(loss='log').fit(x_in_range,
            ↪ y_in_range)


    def fit(self, X, y):
        #building the tree recursively, l and r represent
        #the class range on which the node number v operates
        def build(v, l, r):
            if l == r:
                return
            self.nodes[v] = self.__fit_node_on_range(l, r, X, y)
            m = (l + r) // 2
            build(2 * v, l, m)
            build(2 * v + 1, m + 1, r)
        build(1, 0, self.k - 1)
        return self

    def predict(self, X):
        #recursively getting predictions for X
        def run_dc(x, v, l, r):
            if l == r:
                return np.full(x.shape[0], l)
            m = (l + r) // 2
            if(x.shape[0] == 0):
                return np.array([])
            pred_binary = self.nodes[v].predict(x)
            indices_left = pred_binary == 0
            indices_right = pred_binary == 1
            preds_left = run_dc(x[indices_left], 2 * v, l, m)
            preds_right = run_dc(x[indices_right], 2 * v + 1, m +
                ↪ 1, r)
            pred = np.empty(x.shape[0])
            pred[indices_left] = preds_left
            pred[indices_right] = preds_right
            return pred
        return run_dc(X, 1, 0, self.k - 1)


#SAOC implementation
class SAOCModel(BaseEstimator, ClassifierMixin):


    def __init__(self, k):
        self.k = k
        self.models = [None] * (k - 1)


    def fit(self, X, y):
        for i in range(self.k - 1):
            y_relative = np.where(y > i, 1, 0)
            self.models[i] = SGDClassifier(loss='log').fit(X,
                ↪ y_relative)
        return self


    def predict(self, X):
        pred = np.array([model.predict_proba(X)[:, 1] for model in
            ↪  self.models]).T
        #getting probabilities for each class
        r = np.append(pred, np.zeros((X.shape[0], 1)), axis=1)
        l = np.insert(pred, 0, np.ones(X.shape[0]), axis=1)
        return np.argmax(l - r, axis=1)


#the error metric
def MSE(y, y_pred):
    conf_mat = confusion_matrix(y, y_pred)
    m = y.shape[0]
    diffs = [[None] * k for _ in range(k)]
    for i in range(k):
        for j in range(k):
            diffs[i][j] = (i - j) ** 2
    return 1 / m * np.sum(np.multiply(conf_mat, np.array(diffs)))
mse = make_scorer(MSE, greater_is_better=False)


#the total error for each algorithm
MSE_ovr = 0
MSE_dc = 0
MSE_saoc = 0
for i in range(10):
    print('Starting test number {}'.format(i))
    dc = DCModel(k)
    saoc = SAOCModel(k)
    #SGDClassifier uses one-versus-rest when dealing with more
        ↪ than 2 classes by default
    ovr = SGDClassifier(loss='log')
    #performing cross-validation
    MSE_ovr += cvs(ovr, x, y, scoring=mse, cv=10).mean()
    MSE_dc += cvs(dc, x, y, scoring=mse, cv=10).mean()
    MSE_saoc += cvs(saoc, x, y, scoring=mse, cv=10).mean()
#printing the mean error
print(-MSE_ovr / 10, -MSE_saoc / 10, -MSE_dc / 10)
```