Regularised Conditional Boosting Model with Neural Network Feature Augmentation

TO Yu Hin, Ivan

Abstract

Modern machine learning techniques such as gradient boosting and neural networks are not designed to handle data from discrete choice events directly. In this paper, we proposed a stage-wise gradient descent gradient boosting model for the conditional logit model. We explored augmenting the features between iterations with a neural network while regularising the complexity of the model. The methodology proposed is applied to the Hong Kong Horse Racing market. We show in experiments that the methodology can provide estimates that are stable to pre-processing. It outperformed tradition discrete choice models and previous related work.

Keywords : Discrete Choice, Gradient Boosting, Conditional Logit, Transfer Learning

1 Introduction

Modern machine learning techniques pervades every aspect of data analysis, with neural networks showing success in non-tabular data and general additive models succeeding in more traditional tabular data. However, given such advancements, most machine learning methods are designed for datasets with independent samples and hence have not been widely applied to discrete choice modelling. Various methods for discrete choice data had been investigated, with the Conditional Logit model (CL) suggested by McFadden (1974) being the most widely used method because of its simplicity and interpretability. However, it has not been updated since the advancement of machine learning.

Gradient Boosting algorithms representing general advice models has been widely popular in solving tabular data both in the form of regression and classification, showing outstanding performances in many real-world applications. However, the algorithm is initially developed for more traditional data without connections between observations. Applying it to the discrete choice setting yielded less successful result as it cannot consider the main characteristic of the data.

The goal of this paper is to extend the gradient boosting framework to the discrete choice modelling setting, where multiple observations are connected into an instance. Depending on the implantations of different use cases, researchers have demonstrated multiple techniques in successfully modelling discrete choice problems. These include staging conditional logistic regressions and combinations of neural networks and gradient boosting being incorporated in the conditional logit framework.

In this paper, we proposed a stage-wise gradient descent gradient boosting model for the conditional logit model. It is used to estimate the utility function of a conditional logit regression using a conditional logit loss function. It can be applied directly to discrete choice modelling problems. The aim of the experiments conducted is to access the performance of the proposed method on the Hong Kong Horse Racing market.

The main contributions of this paper are summarised as follows :

- We propose a new algorithm extending the traditional gradient boosting algorithm to accommodate discrete choice data directly. The model includes a neural network feature augmentation step to induce some notion of variability between iterations. It modifies the loss function to a heavily regularised conditional logit loss function, which regularises both the neural network and decision tree components in the system. The model initialises the utility function with the dominant feature in the dataset, which is a common theme in many discrete choice problems. The model includes a post-processing step to reduce the estimators with a LASSO penalty to more efficiently use the model in production.
- We conducted a case study on the Hong Kong horse racing market to validate the effectiveness and efficiency of our proposed model. Results show the model demonstrated better performance than traditional gradient boosting methods for discrete choice models. It also shows better handling of the dominant features compared to conditional logistic regression.

The remainder of the paper is organised as follows: Section 2 reviews the theory of discrete choice modelling, gradient boosting and neural nets.

Then the background of the Hong Kong horse racing market is covered. Section 3 proposed a regularised boosting procedure with neural network feature augmentation with a description of the rationale and algorithm implanted in Python. For illustration, section 4 applies the Python package to a Hong Kong horse racing case study to assess the performance of the proposed method. Finally, section 5 concludes the paper with some remarks and discussion on future work.

2. Literature Review

2.1 Related Work On Methodology

The logistic regression is a generalised linear model which works with binary outcomes, modelling the probability of an outcome. It assumes a linear relationship $F(\cdot)$ between the logged odds and the independent variables.

$$ln\left(\frac{p_h}{1-p_h}\right) = F\left(X_h\right)$$

The probability of an event occurring is defined as $p_h = \frac{1}{1+e^{F(X_h)}}$

In discrete choice events, an observation chooses an alternative among a choice set. Logistic regression and other classification-based techniques treat each sample independently, hence will not be able to capture the competing relationship among the alternatives. One of the most commonly used model to tackle this problem is the conditional logit model (CL) suggested by McFadden (1974). Unlike standard classification techniques, which treat each sample individually, CL considers the connections within the alternative of a choice set. Given an individual r choosing an alternative h in a choice set, the probability of individual r choosing h in the choice set is defined as

$$p_{rh} = \frac{e^{F(X_{rh})}}{\sum_{h\in r} e^{F(X_{rh})}}$$

with $F(X_{rh})$ being the utility function of the individual r choosing an alternative h.

For each alternative in the choice set, the predicted probability is conditioned on the predicted probabilities of other alternatives in the choice set. Normalising the probabilities of alternatives to sum to 1 across a choice set achieves such conditioning. Same as the traditional Logit model, the CL model also assumes the utility function to be linear. Such assumption limits

the amount of variation in the data the model can capture. With advancements in machine learning since its introduction, the use of CL models in conjunction with modern machine learning techniques allows the modelling of data in a more sophisticated manner.

Researches have attempted to bridge the gap between traditional discrete choice modelling since the development of more modern techniques, with most using a variation of the conditional logit model.

Lessmann et al. (2007) used a support vector regressor to model the target aiming to capture complex relationships among the data. This is a first-stage in a two-stage process which minimises the mean square error, weighting the samples by Normalised Discounted Cumulative Gain. Then a utility function is constructed using the predicted target and the dominating feature. A second-stage conditional logit model is fitted on the utility to consider the connections within the alternatives of a choice set. This method has the advantage of generalisation. With such a two-stage approach, the SVM model in the first stage can be replaced by models of any functional form of varying complexity to suit a specific use case. This method of separating the model into two stages requires separating the dataset into two sets. It resulted in the disadvantage of not modelling the discrete choice nature of the data directly. For half of the data, the information of all probabilities in a choice set should sum to one is lost. A more efficient way of minimising the conditional logit loss function throughout all stages could allow all data to be fully utilised and alleviate this problem.

Sifringer et al. (2018) proposed fitting all data into the conditional framework by only fitting one conditional logit model. After fitting the model, all insignificant variables in the utility functions are dropped, replaced by an extra term in the utility function modelled by a neural network. The term is constructed by predicting the target with the insignificant features using a dense neural network. Such separation of features is motivated by the conditional functional form allowing for high interpretability. At the same time, the additional term aims at capturing the non-linearities and subtle interactions in the data. However, this method may overfit the data because the neural

network component only works with features that do not have a significant linear relationship with the target, leaving it with features with higher-order relationships and random noise. The neural network may capture a lot of the noise in the data because the conditional logit model already identifies most of the explicit relationships.

Recently, Haolun She and Guosheng Yin (2018) showed that the powerful gradient boosting model could improve the conditional logistic regression using a combined framework.

Gradient Boosting Decision Trees (Friedman, 2001) is a highly popular ensemble learning algorithm that achieves state-of-the-art performances in popular data science competitions. The model is a stage-wise additive model which builds decision trees sequentially on the mistakes made by previous trees. The trees are built greedily without going back to amend trees of previous iterations. In each iteration, a decision tree is trained and added to a collection of base-learners. After completing the training process, the collection of base-learners are used as an ensemble of additive functions to predict the output. The author also suggested Gradient Boosting can also be viewed as an additive stage-wise gradient descent algorithm in a functional space, which consists of initialisation, projecting gradient to the base-learner, line search, iteration and combining estimators. Such a framework is a good starting point for formulating gradient boosting style models and will be utilised as the basis in section 3.

In the traditional GBDT process, samples are treated independently without mechanisms to consider the connections within the alternatives of a choice set of discrete choice events. Such a mechanism is contrary to conditional logit models which considers all alternatives collectively hence the probabilities of all options in a choice set sum up to one. Haolun She and Guosheng Yin (2018) modified the traditional loss functions that consider samples independently to a conditional logit loss function. Similar to the original model, the utility function of a logit function is modelled instead of the raw target. Instead of decision trees, the proposed model used smoothing spline models as base estimators. This framework has the advantage of

modelling residuals that consider the discrete choice nature of the data within the original gradient descent algorithm. Our proposed model in section 3 adopts this framework as the basis for enhancing it.

When applying gradient boosting on discrete choice data, some use-case specific problems arise. In most discrete choice modelling problems, such as sports betting, there is usually a dominating feature (Benter 1994), (Lessmann et al., 2007), (Silverman, 2013). Such a dominating feature is usually the market odds for a particular target, serving as a proxy for the market confidence. It is usually chosen by subjective experience-based judgement. They concluded the massive difference in predictive strength between the dominating feature and others might undoubtedly overrun the influence of other features. Current techniques for tackling this problem is by formulating models such that the dominating feature is denied from half of the modelling, only introducing them in a latter stage of the process. Gradient boosting cannot be applied directly in this framework without incurring consequence discussed above. Our proposed model in section 2 attempts to consider the dominating features directly in the initialisation step to limit its influence on other features.

Investigating the additive stage-wise gradient descent algorithm, researchers have suggested improvements in the initialisation and combining estimators step.

In traditional gradient boosting procedure, the model usually initialises at the mean of training targets or zero. Shor (1970) suggested that any gradient descent algorithm should converge to global minimum irrespective of the initialisation point with an appropriate learning rate. Mohan et al. (2011), however, suggested giving the algorithm a ‘head start’ by initialising with the predictions of a random forest model. Such initialisation is motivated by starting with a relatively smaller residual will help the algorithm reach the global minimum more efficiently with a larger learning rate. Such method suggests the possibility of using the initialisation step to provide some additional modelling beyond the algorithm.

Our proposed model in section 3 attempts to bridge Mohan and Lessmann's method by incorporating the dominating feature into the initialisation step of the model. Instead of initialising at zero, we suggested initialising with the utility function of a conditional logit model fitted only with the dominating feature. After the step, the dominating feature is denied from subsequent iterations of the boosting process. In most cases, the dominating feature is not as apparent as the sports betting use cases mentioned by researchers above. A line search that iterates through the feature set can be used to choose the dominating feature. The feature that is the best linear predictor of the target under the conditional logit framework could be chosen as the dominating feature.

In traditional gradient boosting, the estimators are usually combined by a simple average. Friedman and Popescu (2003) suggested combing the estimators with linear regression. They suggested viewing the boosting process as a Sequentially Sampled Learning Ensemble. First, a dictionary of base-learners are created, then they are combined in a second stage. The best model in terms of efficiency and out-of-sample accuracy could be obtained by modifying the original combination step with a linear regression fitting the base-learners with a regularisation penalty term.

When applying complicated methods such as gradient boosting to discrete choice problems, a common problem is usually the increase in computational time compared with traditional discrete choice models or data of different nature. Such a problem arise because in discrete choice modelling, multiple utility functions needed to be evaluated before predicting a single observation, with the computational time increasing with the number of alternatives in a choice set. Such constraint is a reason why traditional discrete choice models with simpler constructions are still popular. Computational time is especially crucial in discrete choice applications such as sports betting, and advertising suggestions are highly time-sensitive.

Friedman's method of 'post-processing' the estimators is especially suitable in this case. Our proposed model in section 3 implements a similar penalty on the estimators and pre-processing pipelines on the combination

step to reduce computation time. A post-processing step via a LASSO penalty selects a smaller subset of estimators and pre-processors, reducing the prediction computation time, leading not a more practical model to implement.

Using the information above, we have the elements to build a gradient boosting algorithm that utilises the conditional logit loss function with a modified initialisation and post-processing step.

More recently, Chen et al. (2018) showed that a framework combining gradient boosting and powerful neural network models could improve the predictive performance of a gradient boosting model. They proposed using a long short term memory network to learn the target, then using transfer learning, features are engineered to be passed to a gradient boosting decision tree. This transfer learning technique extracts the last hidden layer as features. The layer is chosen because the neural network and gradient boosting model shares the same target. This transfer learning technique can be viewed as a feature augmentation pre-processing step that takes in the raw features and outputs new features before the gradient boosting model. Such a process is performed once before the data reaches the model, hence the technique is not tailored to the iterative nature of gradient boosting.

This formulation suggests investigating modelling the discrete choice data directly with a neural network with a softmax output layer. This is because the softmax layer shares the same functional form as the conditional logistic regression. However, such a neural network cannot be used in this case. This is because softmax regressions evaluate the multiple class probabilities for each row of data, but conditional logistic regression evaluates individual binary class probabilities in a collection of rows. In many discrete choice applications, there are varying numbers of alternatives in a choice set across different samples. Different races in a horse racing event could have different numbers of horses competing in it. We cannot accommodate this property by dynamically varying the number of neurons in the output layer of a neural network.

Philip Tannor and Lior Rokach (2019) improved on Chen's work by proposing the augmentation of features with an artificial neural network between iterations of a boosting process. Using methods of transfer learning, an ANN has been used to pre-process the feature set before trining each base estimator. The last hidden layer of the ANN is extracted as features during the transfer learning process. The ANN is retrained every few iterations, which is controlled by a hyper-parameter, which results in different features across different iterations of boosting. Such retraining is different from the original framework in which features of a gradient boosting model stays consistent throughout all iterations of the process. However, the networks in different iterations share the same topology. In transfer learning, there is an assumption that the new and original task is similar (S.J. Pan, Q. Yang, 2009). In gradient boosting, each iteration attempts to reduce bias by focusing on regions missed by previous iterations through fitting the residual. Such moving target implies models fitted in different iterations does not share the same target variable and underlying problem. Because different iterations have different targets, there is no reason to suggest the same topology will be suitable for targets of different iterations.

Our proposed model in section 3 implements the ANN feature augmentation pre-processing steps mentioned above. However, in each iteration of the boosting process, we retrain the neural network allowing for different topologies across different iterations. The structure of the network, including the number of neurons and the number of layers, can vary across iterations. Our proposed model in section 2 implements a combination of gradient boosting and neural networks. With both methods being complex, there is a high chance of overfitting. We need some regularisation techniques to control overfitting. To regularise a neural net, we first need a metric to measure the complexity of a neural net.

Raghu et al. (2017) proposed a framework to measure the expressivity of a neural network architecture through its structural properties. Given a neural network constructed with only piecewise linear activation functions, the neural net's expressive power can be measured by the number of linear

regions it can form, hence determining how non-linear the function is. Given the same number of parameters, a shallower network creates less linear region than a deeper network. An activation pattern of a trained neural network is defined as an encoding of the linear regions of activation functions of all neurons. It measures the pattern of neurons switching on across the entire network. Given a fully connected neural network with n hidden layers of k ReLU neurons, and inputs in R^m, the number of Activation Patterns is upper bounded by $O\left(k^{mn}\right)$. This upper bound measure can be used as a proxy for the maximum expressivity of a neural net.

Our proposed model in section 3 uses this measure of expressivity as a measure of the complexity of a neural net. This measure can be used to regularise the contribution of each iteration in the gradient boosting process. The output of a base-learner will be regularised to a more considerable degree as the complexity of the companion neural network increase.

To summarise, many research efforts have addressed developing gradient boosting methods and discrete choice models. However, there is limited study that addresses the combination of gradient boosting, neural networks and discrete choice models. This motivates us to use the methods concluded above in section 2 to expand the gradient boosting framework of Friedman (2001) by

- Adopting the conditional logistic loss function by Haolun She and Guosheng Yin (2018) in the computation of negative gradients
- Initialising the model with a predictive model by Mohan et al. (2011)
- Pre-processing the features by a neural network augmentation proposed by Philip Tannor and Lior Rokach (2019)
- Regularising the pre-processing step with a complexity measure by Raghu et al. (2017)
- Post-processing the base estimators by Friedman and Popescu (2003)

2.2 Related Work On Horse Racing

Horse racing has long been of interest to academics because it serves as a proxy for the more expansive complex and noisy financial market. The parimutuel nature of the sport also serves as a representative for any other scenario where many people have a varying degree of information and skill. Horse Racing has been the most popular sport in Hong Kong, also being one of the largest investment markets in the country. Such lucrative nature of the market led to researchers focusing on that particular market. A horse racing wagering model built on machine learning approaches has the advantage of testability and consistency compared to the traditional domain experience based handicapping approach prevalent in the Hong Kong market.

At the Hong Kong racetrack, the game is parimutuel, meaning all betters place bets in a betting pool. A percentage of the pool is divided proportionally amongst the winners based on the amount they bet. Unlike most sports or casino betting, the public does not bet against the house, which sets the odds with a negative expectation. The market determines the odds after the house takes a fixed percentage from the betting pool (usually 18%). Such odds represents the market's 'confidence' on a particular horse. The horse with the lowest odds is viewed as the public's 'best guess' on the winner of the race.

Since the 1980s, academics have been researching methods and models to predict different aspects of horse racing, with most of the studies focusing on the Hong Kong or UK market. Bolton and Chapman (1986) were the first academics that used statistical techniques to predict horse racing results. They proposed the use of a conditional logistic regression model to predict the probability of a horse winning. They assumed a utility function determines the performance of a horse. Such function comprises of a utility function and an independently and identically distributed random error. The utility function is defined as a linear combination of deterministic performance indicators. This assumption proved to be valid by experiment as the utility function serves as a proxy for an experienced handicapper weighing different

aspects of horse racing in their mind. Based on the US market, they claim to obtain a profit of 3.6% over 200 simulated races.

Building on Chapman's work, Benter W. (1994) proposed the use of two of Bolton's model to build a two-stage conditional logistic regression to model the result of horse races. The first stage models the relative strength of a horse. Then the second stage combines the predicted strength with the logged public odds implied probability. This two-staged framework is widely considered to be the seminal work in the field, with most subsequent researchers adopting a similar method. The most important insight from his work is the separation of the public odds, the dominating feature. Such methods have the advantage of avoiding the dominating feature from overrunning other features in a single-stage regression, allowing their significance not to be masked. Benter made significant profits by operating a cartel in Hong Kong betting on the races using the proposed model. Another of Benter's significant contribution was in the field of feature engineering. He proposed using Auxiliary Regressions and its residuals to model horse racing facts. It has the advantage of singling out the marginal effects of particular racing events. Section 4 implements the methods above for experimental feature engineering.

Lessmann et al. (2007) extended Benter's framework by modifying the first stage of the process. Proposing the use of a support vector machine in replace of a conditional logistic regression during the first stage of the model. Motivated by the fact that SVMs will better model the non-linearities among the features, he used an SVM regressor to model the finishing position of a horse without using odds related variables. In the second stage, he followed Benter's method to combine information from the publicly available odds. The dataset used in the experiments mentioned contains racing data from 1995 to 2000, compared to Benter's 1988 to 1993. Based on simulations by the author, the profits of using a simple two-stage approach decreased significantly across the two time periods. Such convergence in handicapping ability suggests the market is getting more efficient over time, and the advances in Benter's model is absorbed into the market through the odds.

Lessmann and Sung (2009) improved on their work in 2007 by replacing the SVM regression in stage one with an SVM classifier. The SVC is motivated by the observation that finishing positions among minor places are noisy and do not provide much information as they do not earn any prize money. Their work established classification as a more straightforward and accurate approach in modelling the horse racing problem.

Silverman (2013) deviated from Benter's two-stage framework by directly considering the odds feature with a Cox Proportional Hazard model. He introduced a frailty parameter to model the odds. The frailty term has the advantage of fixing the coefficient for the odds feature, further ensuring the effect of odds does not significantly influence the coefficients of other features. This suggests further models could fix the influence of dominating features to allow more room for the remainder features. He demonstrated the frailty term of odds outperformed the two-stage conditional logistic regression utilised by most researchers.

Since Benter's work, the two-stage conditional logit framework has been the gold standard in horse racing prediction, with subsequent researchers modifying his work with modern machine learning techniques beyond 1994.

A review of currently available literature has failed to find any gradient boosting tree ensemble approaches to modelling the horse racing market. The case study in section 4 will attempt to investigate the elements necessary to build a horse racing handicapping model using the gradient boosting based method proposed in section 3.

3. Methodology

3.1 RACBoost Training Process

In this section, we extended the traditional gradient boosting algorithm to accommodate discrete choice data directly. We established a novel boosting approach to non-parametrically fit the utility function in the conditional logit model. The goal is to estimate an additive function of $F(X)$ mapping X directly to y that minimises the expected value of a loss function over the joint distribution of all (x, y) values. The description is based on the stage-wise additive gradient descent formulation (Friedman, 1999) with minor modifications. For a given dataset D with n samples and p features,

$$D = \left\{ \left(x_{rh}, y_{rh}\right) \right\}, \left(\left| D \right| = n, x_{rh} \in \mathbb{R}^p \right),$$

An ensemble model uses M additive functions h_m to estimate the utility function $F(X)$. The ensemble is used to predict the probability of each class by passing the estimated utility function through the conditional logit function as follows

$$\hat{y_{rh}} = \frac{e^{F(X_{rh})}}{\sum_{h \in r} e^{F(X_{rh})}}$$

A description of the proposed boosting procedure is detailed in the following sections. The training process of the proposed model is visualised in Figure 1 and Algorithm 1, and the inferential process is visualised in Figure 2.

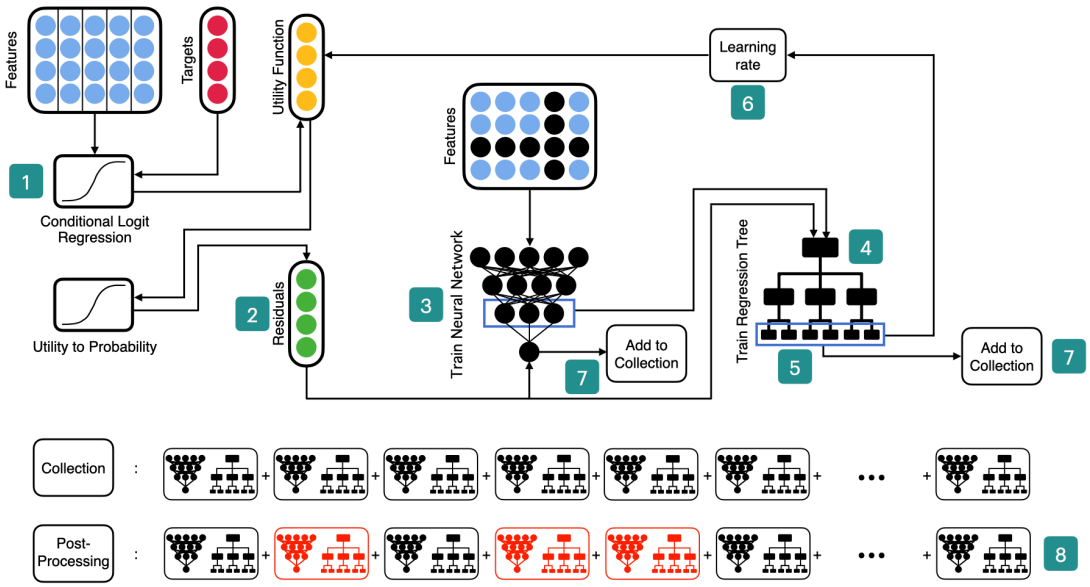


Figure 1 : This figure describes the training process of the model

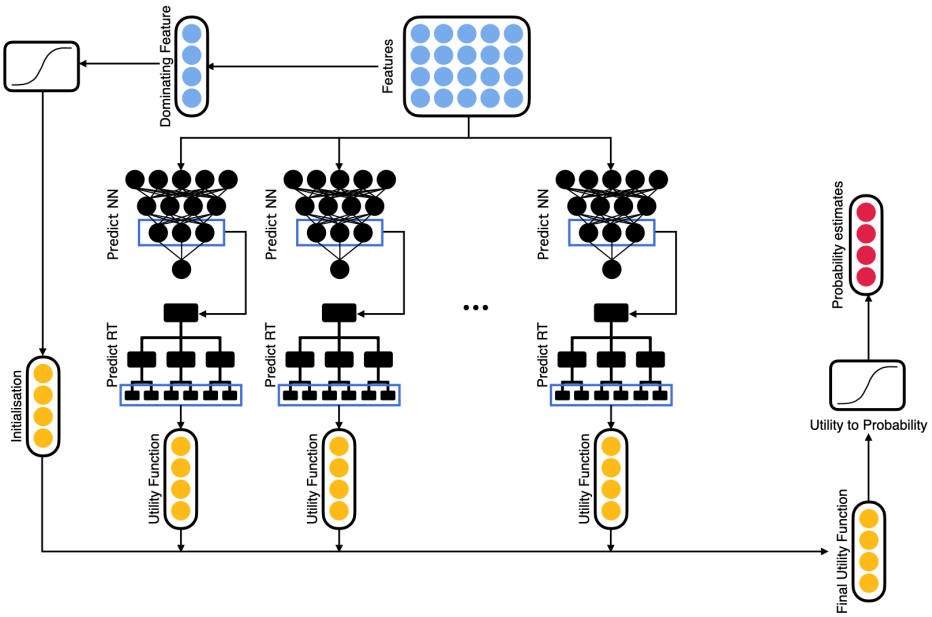


Figure 2 : This figure describes the inferential process of the model

Algorithm 1 : RACBoost Training Process

Input: Training Dataset : $\{X, Y\}$

Loss Function : $L(F) = \sum_{r=1}^{R} - l\left(F, y_r\right) + \sum_{m=1}^{M} \Omega\left(f_m, h_m\right)$

Output: A model $F^{[M]}$ containing trained neural networks and decision trees

1: **def** TRAIN()

2: Initialise $F^{[0]}(X)$ as $F^{[0]}(X^{(l)})$,

$$l = \underset{1 \le l \le p}{arg\,min} - \sum_{r=1}^{R} \left[\sum_{h \in r} F(X_{rh}^{(l)}) y_{rh} - ln \left\{ \sum_{h \in r} e^{F(X_{rh}^{(l)})} \right\} \right]$$

3: **for** m = 1 … M:

4: Compute negative gradient

$$-g_{rh}^{[m]} = - \left[\frac{\partial log L(F(X))}{\partial F(X)} \right]_{F(X)=F^{[m-1]}(X)} = y_{rh} - \frac{e^{F^{[m-1]}(X_{rh})}}{\sum_{h \in r} e^{F^{[m-1]}(X_{rh})}}$$

5: **if** ignore_feature != None:

6: X_sample <- X ∉ ignore_features

7: **if** row_subsample < 1:

8: X_sample <- X_sample ∈ sampled rows

9: **if** col_subsample < 1:

10: X_sample <- X_sample ∈ sampled columns

11: Train Feature Augmentation Neural Network f_m,
calculate complexity measure C_{f_m}

12: Fit Decision Tree h_m
calculate terminal values γ_{jm}

13: $F^{[m]}(X) = F^{[m-1]}(X) + \rho * \gamma_{h_m}^{j}$

14: Post-process $F^{[M]}$ with Lasso Regression

$$\alpha(\lambda) = \underset{\alpha}{argmin} \sum_{r=1}^{R} \left[\sum_{h \in r} l\left(\alpha_0 + \sum_{m=0}^{M} \alpha_m F^{[m]}, y_{rh} \right) \right] + \lambda_{post} \sum_{m=1}^{M} \left| \alpha_m \right|$$

15: return $F^{[M]}$

3.2 Regularised Loss Function

Under the boosting framework, we need a differentiable loss function for minimisation to learn the pre-processors and base-learners in the model

$$L(F) = \sum_{r=1}^{R} - l\left(F, y_r\right) + \sum_{m=1}^{M} \Omega\left(f_m, h_m\right)$$

The first term is a differentiable convex loss function. With the discrete choice nature of our proposed data, it is clear that the model needs to consider this fundamental characteristic of the data to guarantee good performance. The conditional logic log-likelihood is considered to measures how well our model fits the training data by measuring the difference between the true value of y (y_{rh}) and the predicted value of y ($\hat{y_{rh}}$). y_r is a vector of y_{rh}, representing all true values of y in the choice set. It considers the discrete choice nature of the data by penalising incorrect predictions in groups of observations.

$$l\left(F, y_r\right) = \left[\sum_{h \in r} F(X_{rh}) y_{rh} - log\left\{ \sum_{h \in r} e^{F(X_{rh})} \right\} \right]$$

With both gradient boosting and neural networks being complex algorithms, they tend to overfit. We need our loss function to be regularised as below.

The second term $\Omega\left(f_m, h_m\right)$ penalises the complexity of the neural nets and regression tree functions to avoid overfitting by selecting a simple model in the pre-processing and base-learner step.

$$\Omega\left(f_m, h_m\right) = \lambda_{L1} \sum_{i=1}^{n} \left\| w_{f_m}^{i} \right\|_1 + \lambda_{L2} \sum_{i=1}^{n} \left\| w_{f_m}^{i} \right\|_2 + \psi C_{f_m} + \frac{1}{2} \phi \sum_{j=1}^{J} \gamma_{h_m}^{j\ 2}$$

In the pre-processing step of the process, the first and second term penalises the weight matrixes of the neural networks in each iteration. With an

elastic-net penalty, where n is the number of parameters in the network, hyper-parameters ***reg_lambda_l1*** (λ_{L1}) or ***reg_lambda_l2*** (λ_{L2}) controls the regularisation. Increasing them will make the neural network more conservative, decreasing variance at the expense of bias. The penalty is constant across all iterations of the process.

In the pre-processing step, we will fit a neural network model for each iteration. The more complex the neural network is constructed, the easier it is to overfit the data. The ***nn_complexity*** (ψ) hyper-parameter is implemented to penalise the complexity by multiplying by a complexity measure C_{f_m}. The more complex the neural network is, the less the features hence the decision tree associated with the neural network should be updated to the model to avoid overfitting. A ***nn_complexity*** value larger than zero will decrease the output values of each leaf.

In the base-learner step of the process, the second term penalises the output value $\gamma_{h_m}^{j}$ of each leaf j in the decision tree with structure h_m. This method is also implemented in the XGBoost (Chen and Guestrin, 2016) model. The hyper-parameter ***reg_leaf*** (ϕ) aims to smooth the final learnt weights to avoid overfitting.

The additive gradient descent formulation of the model requires the calculation of the first and second partial derivatives with respect to the utility function as below

$$\frac{\partial l}{\partial F_{rh}} = y_{rh} - \frac{e^{F(X_{rh})}}{\sum_{h \in r} e^{F(X_{rh})}}$$

$$\frac{\partial^2 l}{\partial F_{rh}^2} = \frac{\left(\sum_{h \in r} e^{F(X_{rh})} \right) - \left(e^{F(X_{rh})} \right)^2}{\left(\sum_{h \in r} e^{F(X_{rh})} \right)^2}$$

3.3.1 Step 1 : Initialisation

Unlike traditional gradient boosting, we do not initialise $F(X)$ with zero, ideated we initialise the utility function with a transformation of the dominating feature. This has the effect of avoiding the dominating feature to overrun other features, allowing variation not captured by the dominating feature to be modelled to a more considerable degree. We first fit p conditional logit models using y_{rh} as target and the lth feature in X_{rh} as the predictor, for $l = 1,...,p$. Among these p conditional logit models, the dominating feature is defined as the feature l that gives us the largest conditional logit log-likelihood. It can be obtained from the solution of

$$l = arg \min_{1 \le l \le p} - \sum_{r=1}^{R} \left[\sum_{h \in r} F(X_{rh}^{(l)}) y_{rh} - ln \left\{ \sum_{h \in r} e^{F(X_{rh}^{(l)})} \right\} \right]$$

Then, we use l as the only feature in a dataset to fit a conditional logit model on y, obtaining the utility function $F^{[0]}(X^{(l)})$ from the conditional logit model. It is then used as the initialisation point $F^{[0]}(X)$ for the boosting model. At the same stage, we initialise the underlying number of iterations $m = 0$.

By initialising the model with the dominating feature, we ensured and controlled the influence of the dominating feature. In an attempt to avoid it from overrunning other features, a hyper-parameter ***ignore_features*** is available to deny subsequent steps of the process from utilising it, essentially excluding it from the iterations of the training process. This will prevent the model from using the feature over and over again, overfitting to a single feature. This will allow other features to shine without the dominating feature. Although other features can also be ignored, by default, the dominating feature is the only feature to be ignored.

3.3.2 Step 2 : Computing Negative Gradient

In each iteration, we compute the negative gradient of the loss function $-g_{rh}^{[m]}$ for each sample in the training dataset. Such a gradient by construction gives the best steepest-descent step direction in the data space. It will be the target for our base-learners.

$$-g_{rh}^{[m]} = -\left[\frac{\partial log L(F(X))}{\partial F(X)}\right]_{F(X)=F^{[m-1]}(X)} = y_{rh} - \frac{e^{F^{[m-1]}(X_{rh})}}{\sum_{h\in r} e^{F^{[m-1]}(X_{rh})}}$$

for $r = 1...R$ and $h = 1...H$.

3.3.3 Step 3 : Stage-Wise Neural Net Feature Augmentation

The following step is not included in the original Friedman (2001) framework. We attempt to augment the original dataset by passing the raw features into an artificial neural network f_m, which learns the updated target (negative gradient). This serves as a pre-processing step between the raw features and the base-learner models.

If the neural network achieves a descent performance, the information it modelled could be preserved and transferred to the base-learner models. The last hidden layer of the trained neural network is extracted as the augmented features. The last hidden layer is chosen because the boosting process and the neural network shares the same target and object function.

Next, we discuss the structure of the neural network. The objective function of the neural network is the Regularised Loss Function $L(\phi)$. The hidden layers all have ReLU activation functions and the output layer has a logistic activation function. A random 10% of the individuals in the training set will be set aside for validation during the training process, with early stopping occurring if the validation loss function does not improve for 10 epochs. Batch-normalisation is implemented to reduce the dependence between each

layer. This has the effect of allowing the models to be more stable to covariate shift in the dataset.

In section 2, we discussed the motivations of varying the topology of the network between iterations. The topology of the network is structured in a pyramid fashion and controlled by hyper-parameters ***nn_count*** and ***nn_shrink***. We initialise the first hidden layer of the first iteration with ***nn_count*** neurons, then add the next hidden layer with $int(nn-count * nn-shirnk)$ neurons until the neuron count of a layer is less than 30. With the final layer of the network having a maximum size of 30, the neural network augments the feature space into a lower dimensional space. Hence the pre-processing step performs dimension reduction which reduces the arbitrary feature set size to less than 30.

On the assumption that each subsequent boosting iteration attempt to model regions that was missed by pervious iterations, we could allow the model to start with a simpler network topology in the initial iterations then increase the complexity of the network as the model move on to less obvious variations in the data. A ***nn_size*** hyper-parameter controls the complexity of the network. The number of first layer neuron ***nn_count*** of each iteration is scaled by ***nn_size*** to determine the new ***nn_count*** of the next iteration. A value of 1 means the topology does not change over iterations, while a value larger than 1 increases the topology complexity with the number of iterations.

After the neural network is trained for an iteration, the complexity measure C_{f_m} of the network topology is computed . The complexity measure is used to penalise the output of the base-learners in step 5. The contribution of a complex network is scaled down to avoid overfitting. For a fully connected network with with n hidden layers of k ReLU neurons, and inputs in R^m, the complexity measure is defined as

$$C_{f_m} = k^{mn}$$

3.3.4 Step 4 : Fitting Base Learner To Negative Gradient

Since the data is finite and cannot accurately represent the underlying distribution, the gradients are approximated by greedily adding base-learners that improve our prediction to the most significant degree. In this implementation, the base-learners h_m are assumed to be regularised decision trees. The following optimisation problem selects the decision tree that best fits the augmented data $f_m(X)$ to the negative gradients.

$$h_m = \underset{h_m,\beta}{argmin} \sum_{r=1}^{R} l\left(F_r^{[m-1]}(X) + \beta h_m\left(f_m(X)\right), y_r\right)$$

A simple regularisation is implemented by a hyper-parameter ***max_depth*** which controls the maximum depth of the individual regression tree base estimators. The maximum depth also limits the number of nodes in the trees, controlling overfitting.

3.3.5 Step 5 : Line Search

After fixing the structure of the base-learner, we next calculate the terminal node γ_{jm} for each leaf j in the trained decision tree. The updated terminal node gives us the best greedy update to the model. This is required because the negative gradient only gives us the direction for updating, but not the magnitude. The following optimising problem calculates the multiplier :

$$\gamma_{h_m}^j = \underset{\gamma_{h_m}^j}{argmin} \sum_{X_i \in R_{ij}} l\left(F_r^{[m-1]}(X) + \gamma_{h_m}^j, y_r\right) + \Omega\left(f_m, h_m\right)$$

Because the loss function for optimising is involved, we use a second-order Taylor approximation of the objective function. The loss function could be rewritten as follows :

$$l\left(F^{[m-1]}(X) + \gamma_{h_m}^j, y_r\right) \approx l\left(F^{[m-1]}(X), y_r\right) + \gamma_{h_m}^j \frac{\partial}{\partial F} l\left(F, y_r\right) + \frac{1}{2}\gamma_{h_m}^{j2} \frac{\partial^2}{\partial F^2} l\left(F, y_r\right)$$

Setting $\frac{\partial}{\partial \gamma_{h_m}^{j}} \sum_{X_i \in R_{ij}} \left[l\left(F^{[m-1]}(X) + \gamma_{h_m}^{j} y_r\right) + \Omega\left(f_m, h_m\right) \right] = 0$

$$\sum_{X_i \in R_{ij}} \left[\frac{\partial}{\partial F} l\left(F, y_r\right) \right] + \gamma_{h_m}^{j} \sum_{X_i \in R_{ij}} \left[\frac{\partial^2}{\partial F^2} l\left(F, y_r\right) + \phi \right] = 0$$

the optimal value $\gamma_{h_m}^{j}$ of terminal node (leaf) j for updating the model is given by

$$\gamma_{h_m}^{j} = -\frac{\sum_{X_i \in R_{ij}} \left[\frac{\partial}{\partial F} l\left(F, y_r\right) \right]}{\sum_{X_i \in R_{ij}} \left[\frac{\partial^2}{\partial F^2} l\left(F, y_r\right) \right] + \phi} + \psi C_{f_m}$$

3.3.6 Step 6 : Updating Via Shrinkage

After building the pre-processor and base-estimator for this iteration, we update our estimates of the utility function with the following equation.

$$F^{[m]}(X) = F^{[m-1]}(X) + \rho * \gamma_{h_m}^{j}$$

A simple method of shrinkage for boosting is implemented by specifying a learning rate hyper-parameter ρ **(*learning_rate*).** It scales the contribution of each set of neural network and base-learner by a factor between 0 and 1. Friedman (2001) demonstrated smaller values of learning rates when tuned together with the number of estimators produces better out of sample performance. Such performance is achieved by reducing the influence of individual iterations and leave room for future estimators to improve the estimation.

3.3.7 Step 7 : Iteration

After updating the predicted utility function, we add the learnt models for this iteration to the collection of pre-processor and base-estimator to be used

in later steps. Then, we increment m by one and repeat steps 2 to 3 until a pre-specified number of iterations M controlled by hyper-parameter ***n_estimators***, or all the residual in the training set reaches zero.

3.3.8 Step 8 : Lasso Post-Processing

After all iterations are completed, we end up with a collection of pre-processors and base-estimators. In step 6, all of them are combined by summing then multiplied by the learning rate, with each having the same weighting. The following sums the estimators more efficiently by post-processing the model with a linear regression predicting the target with an L1 penalty term as below.

$$\alpha\left(\lambda\right) = \underset{\alpha}{argmin} \sum_{r=1}^{R} \left[\sum_{h \in r} l\left(\alpha_0 + \sum_{m=0}^{M} \alpha_m F^{[m]}, y_{rh} \right) \right] + \lambda_{post} \sum_{m=1}^{M} \left| \alpha_m \right|$$

The λ_{post} **(*post_lambda*)** hyper-parameter controls the degree of regularisation on the base-estimators.

Using such regression, we can allow for different weights on learners form different iterations, essentially having the decision trees scaled by different learning rates. This produces the best model of accuracy in terms of out-of-sample performance. We can also eliminate the pre-processor, base-learner combinations that do not significantly contribute to predicting the target. This reduces the number of iterations to go through in the prediction process, reducing prediction time and increasing efficiency.

This linear regression model concludes the training process and can be used for prediction.

3.4 Overfitting Control

In the regularised loss function, regularisation terms are implemented to control overfitting. In the steps above, multiple hyper-parameters are available

to achieve a similar goal. Additional techniques below were implemented to combat overfitting further.

Breiman (2001) demonstrated that bootstrap averaging improves out of sample performance of a noisy classifier by averaging. The first technique implemented is row-wise subsampling. At each iteration, a fraction of the training data is sampled randomly and passed to the feature augmentation neural network. Because of the discrete choice nature of the data, individuals who are choosing among a choice set is the unit for sampling instead of simply sampling rows. The **row_subsample** hyper-parameter controls the fraction of individuals in the training dataset passed into the feature augmentation training process. A sample of less than one will result in a reduction in variance and increase in bias.

The second technique implemented is column-wise subsampling. At each iteration, a fraction of the features is sampled and passed to the feature augmentation step. The ***col_subsample*** hyper-parameter controls the fraction of features to use in each iteration. It works in conjunction with the ***ignore_features*** hyper-parameter, where the features list is reduced before being sampled.

The full list of hyper-parameters is attached in Appendix 1.

3.5 Implementation

In order to compare the results of our proposed method with techniques proposed by other researchers, we implemented the algorithm as an open-source package RACBoost (Appendix 2) in Python utilising frameworks from Sklearn (Pedregosa et al., 2011) and TensorFlow.

4. Case Study - Horse Racing Prediction

We applied our algorithm to the Hong Kong horse racing market as an illustrative example. The following section investigates each horse's probability of winning a race against other horses in the same race and utilises such probabilities to build profitable wagering models. Using the results of the wagering model, we attempt to demonstrate the performance of our proposed method in sections below.

4.1 Dataset Preparation And Feature Engineering

The Hong Kong market is chosen because it has a small and tight race field, with 1500 horses racing every year, which provides a consistent dataset across time. The dataset used in this experiment is constructed by using a Python script to scrape freely available data from the website of the Hong Kong Jockey Club, which is the official organiser of all horse races in Hong Kong. Such methods of data collection increase the reproducibility of the results as the Python script automatically scrapes the website for new data points after a race-day without substantial human input. This method also serves as the foundation of an automated betting system, in which daily data is downloaded, processed and betted on by the system automatically.

Historical data from the 2012 to 2019 horse season were collected, totalling 73,245 observations from 5,929 races across 39 columns of racing facts. The goal of this experiment is to model the probability of a horse winning a race.

An essential step in the modelling process is the engineering of features. Traditionally in the Hong Kong market, horse racing is modelled based on experience. Given a four years old male horse running a 1200m class 1 grass race with twelve other horses, three of which he had beaten previously, fourteen days after his last race and thirty days since his last top-three finish. While an experienced handicapper may be able to deduce the chance of him winning and bet accordingly, it is difficult to break down the thought process in simple terms systematically to allow for reproducing the results consistently.

Silverman N. (2013) suggested that because of the easy reproducibility of statistical models, simple transformations of raw racing facts does not provide enough information to model the near efficient horse racing market. In order to establish the effectiveness of our proposed method, a better representation of racing facts through feature engineering is necessitated to model the problem to a meaningful degree.

For this experiment, 399 independent feature across 5929 races and 73,245 racing horses were engineered from complex non-linear derivations of base racing facts. Methods include auxiliary regressions predicting a specific element of horse racing, their residuals, nearest neighbour samples, composite running statistics, race adjusted speed figures and exponential smoothing. Features were constructed from reading literature on the domain of horse racing and feature engineering, conservations with local experts and numerous trial and error experiments. These features cover different aspects of a horse's information, each aiming to capture a unique factor that affects the winning potential of a horse, such as the current condition of the horse, past performances of the horse, jockey and stable, racetrack conditions and composite statistics. An example feature that we found significant in our experiments illustrates the complexity of the features used in this experiment:

"The feature Winning Energy Distribution on Profile measures the running style of the underlying horse and its suitability to the racecourse to today's race. For each race of a particular racetrack condition profile, including distance, going and racecourse, the energy profile of each winning horse is calculated as the ratio between the horse's pace in the early sections and the final sections. A profile energy distribution of the winning horses' energy profile is built using a radial basis function. For each of the underlying horse's past races, the energy profile is calculated. Then a horse energy distribution is constructed as above. When engineering this feature for a new race, modelling the horse's preference to today's racetrack conditions, the distance between the horse's energy distribution and the profile's energy distribution is calculated. The figure is finally weighted by the number of sample races for the horse and profile."

A brief description of all independent features used in this experiment is given in Appendix 3.

The feature engineering process exposes some limitations. Most of the features engineered include multiple hyper-parameters. For example, weights on auxiliary regressions and number of lags to include in a soothing exponential model involve tuning parameters to control the amount of information to include. Tuning hyper-parameters in the feature engineering stage may reduce the degree of freedom and significantly increase the computational cost beyond reason given the size of the data. Modern model interpretation technique Accumulated Local Effects plots, proposed by Apley D. W, Zhu J. (2019) were used to visualise the marginal relationship between the feature and the target. Hyper-parameter configurations for the features are selected by engineering features that show a linear relationship with the target on the ALE plots. This negated the need for grid search based selection methods.

Another limitation faced in the feature engineering process is the missing values created by the lagged variables and moving averages in time series based features. Features created from incomplete data will contaminate the dataset hence inducing noise in the model. To prevent the problem, the first thousand races were allocated for overhead and will not be used for training.

4.2 Experimental Setup

To evaluate the performance of our proposed method, we will compare the algorithm in section 2 with the following models. We first tested the following algorithms suggested by other researchers in section 2.2 as a base level of performance :

- CL_Ridge : One-stage conditional logistic regression model with ridge penalty (Bolton and Chapman, 1986)
- CL_CL_Ridge : Two-stage conditional logistic regression model with ridge penalty (Benter W., 1994)

- SVR_CL : Two-stage SVR x conditional logistic regression model (Lesson et al., 2007)
- SVC_CL : Two-stage SVC x conditional logistic regression model with ridge penalty (Lessmann and Sung, 2009)
- CL_Frailty : One-stage conditional logistic regression model with frailty term (Silverman, 2013)

For some of the above models, a ridge penalty is implemented to control overfitting. A regularisation parameter of zero will imply the original model without any regularisation.

Because of the gradient boosting nature of or algorithm, we need some gradient bossing models to serve as a benchmark. We compared our method with multiple popular gradient boosting libraries, namely XGBoost (Chen and Guestrin, 2016) and CatBoost (Prokhorenkova et al., 2018). With the proven success of Benter's two-stage method, we also implemented the above gradient boosting models in Benter's framework, resulting in the following models :

- XGBoost : One-stage XGBoost model (Chen and Guestrin, 2016)
- Catboost : One-stage CatBoost model (Prokhorenkova et al., 2018)
- XGBoost_CL : Two-stage XGBoost x conditional logistic regression model (Chen and Guestrin, 2016)
- Catboost_CL : Two-stage CatBoost x conditional logistic regression model (Prokhorenkova et al., 2018)

As a first step before fitting the model, the data needed to be pre-processed with a transformation. As we have a heavy neural network element in our model, standardising the data in some manner will help with convergence speed. Lesson et al. (2007) suggested standardising the horse racing data by choice set (race), which provides some modelling of the discrete choice nature of the data. Such standardisation is done to compensate for the fact that their first stage model does not model discrete

choice directly. Silverman (2013) suggested standardising the data by profile, because data of the same profile share a similar distribution for some key metrics. Along with the methods suggested above, we also pre-processed the data with standard machine learning techniques that shown positive results in our experiment as follows :

- Standardising by Race
- Standardising by Profile
- Log Transformation
- Box Cox Transformation
- Principal Component Analysis Transformation

The models used in the evaluation process include combinations of the nine candidate models and five pre-processing pipelines, totally 45 models.

In order to evaluate the model, the dataset is split into a training and testing dataset. Four datasets were sampled from the training dataset for cross-validation hyper-parameter tuning using stratified sampling. Simulated betting was conducted on the hold out testing dataset to archive a final model performance estimate.

In horse racing, our final goal is not to produce a model which gives the best prediction accuracy. Instead, we want a model to maximise profit, which is fundamentally a different problem. This is evidenced by the fact that predicting the horse with the lowest odds will give a high accuracy in general, but usually will incur a massive loss because of the low payoff. Therefore, we need a metric to evaluate model performance defined by profitability, measured by return on investment. This also implies simulated betting on real races is required for performance measure.

In order to estimate the profitability of the model, a simple Kelly Criterion (Kelly, 1956) is used to calculate the optimal proportion of money to wager on a particular horse given a predicted probability of winning. Given a horse with

an estimated probability of winning p, market odds, the Kelly Criterion suggested fraction of bankroll to wager f is defined as

$$f = \frac{p(b+1)-1}{b}$$

The Kelly Criterion determines the optimal amount to bet that maximises the expected log profit across all potential alternatives with a zero probability of ruin. The final performance measure is defined as the return on investment (ROI) as below :

$$ROI = \frac{1}{4}\sum_{i=1}^{4}\left[\sum_{r}^{R} b_{rh}^{w} \cdot o_{rh}^{w} - \sum_{h \in r} b_{rh}\right]$$

with b_{rh}^{w} representing the amount to bet on the winning horse, b_{rh} representing the amount to bet on horse h in race r, o_{rh} representing the winning odds of horse h in race r.

A limitation of evaluating the model through ROI is the quality of the data. When evaluating the ROI, we use the publicly available odds. This piece of information is exposed to the modelling process through feature engineering. In practice, the market odds keep changing until the market is closed. Hence we would not be able to use the closing odds in the modelling process.

When building automated betting systems, we need to buffer time for data scrapping and model prediction time. Hence the market odds are usually scrapped a few minutes before the market closure. Although we attempt to place bets as close to market closure as possible, such odds usually differs slightly with the actual odds after closure. This renders the ROI estimate slightly different from the one we would obtain if we actually bet with the model.

As discussed in section 3, several hyper-parameters in our proposed model needed to be optimised in order to achieve optimal out-of-sample performance. The same could be said for other model candidates stated above. For each model candidate, a model instance is built sequentially on

three partitions of the training data, then evaluated on the remaining one. The resulting four ROI values are averaged to provide an estimated out-of-sample ROI on the hyper-parameter set.

For Conditional Logit and SVM models, the hyper-parameter space is iterated through grid search. Our method proposed in section 3 and gradient boosting based models is based on a broader set of hyper-parameters that made grid search inefficient. Hence their hyper-parameters are optimised using 1000 rounds of Bayesian optimisation (Snoek et al. 2012). The full list of hyper-parameter search space is attached in Appendix 4.

Given the size of the dataset, with 399 features and 73,245 instances, there is a glaring curse of dimensionality problem. Fitting thousands of rounds of 4-fold cross-validation will be slow even on powerful hardware. In an attempt to reduce the computation time of the training process, we isolated steps that can benefit form parallel programming. Given the nature of stage-wise gradient boosting, the iterations in the model cannot be computed in parallel. However, different iterations in the Bayesian optimisation problem can be run in parallel. This is achieved by a mapping step fitting multiple instances of a model candidate. Then a reduce step updating the optimiser sequentially, fetching a new point before updating another one to maximise the efficiency of the suggestions.

4.3 Results Interpretation

This section summarises and discusses the findings of our simulation in this study, contrasting the performance of the proposed method with existing models.

In Table 1 below, we show comparisons between the results of our proposed model and the results of the candidate models mentioned above after the hyper-parameter section process.

Model Candidate	Source	Return on Investment
RACBoost	This Paper	18.13%

Model Candidate	Source	Return on Investment
CL_Ridge	Bolton and Chapman, 1986	-95.54%
CL_CL_Ridge	Benter W., 1994	4.07%
SVR_CL	Lesson et al., 2007	2.38%
SVC_CL	Lessmann and Sung, 2009	3.47%
CL_Frailty	Silverman, 2013	-72.31%
XGBoost	Chen and Guestrin, 2016	-12.5%
CatBoost	Prokhorenkova et al., 2018	-49.27%
XGBoost_CL	Chen and Guestrin, 2016	7.24%
CatBoost_CL	Prokhorenkova et al., 2018	10.2%

Table 1 : Candidate Model Return on Investment Results.

The ability of our proposed model to forecast horse racing results with positive results are demonstrated with out-of-sample simulated betting results. Despite the better performance of our proposed model compared with other candidate models, the complicated nature of the model means there is no easy way to break down the source of such performance.

First, we explore the structure of the models. Previous literature shown two-staged models outperform one-staged model when using Conditional Logit models. Our experiments confirm this hypothesis with all two-staged models earring a profit, and all traditional one-staged models incurring a loss. This is expected as the dominant feature overruns other features likely causing the subtle relationships to be missed by the model. A two-staged model solved this problem. The only profitable one-stage model is the model we proposed. This is likely due to our limitation of the dominating feature's influence on other features by a new initialisation method. This serves a similar purpose as the two-staged approach. It also outperformed the two-staged models, likely because all the data in our model is 'conditioned on', comparing to only a subset of it passed to the conditional logit model in traditional two-staged models.

Next, we look at the functional form of the models. Previous literature suggests any model without any element of conditional logistic regression will

suffer in terms of performance because the estimated winning probabilities do not sum to one across the choice set. This is confirmed by the single staged gradient boosting models incurring a loss. However, we observe the purely conditional logistic regression based models also do not perform well. This may indicate that the public has nearly fully discounted the developments made by the researchers above, and the market is approaching efficiency. This is evidenced by the emergence of betting cartels in Hong Kong algorithmically betting horse races. This may suggest a non-linear element needed to be included in order to have a decent performance.

Hence, we investigate the non-linear components in the models. We obverse all two-staged model reports a differing degree of profit. The main difference in profit is due to the complexity in the first stage of the process, with more complex models yielding better results. The most successful model among them is the CatBoost Conditional Logit model. However, our one-stage method significantly outperforms any two-stage methods. This is likely due to the combination of both components of the CatBoost CL model in one stage. The non-linearises of the gradient boosting and discreet choice nature of conditional logit are both considered in the model and trained on all parts of the data. An important point to note is that the return on investment is over a year of simulated betting, the edge over the public betting market is still small considering the number of races in consideration.

Next, we investigate the model's sensitivity to different pre-process pipelines. In table 2, we compare the performance between different model candidates across different pre-processing pipelines. They share the same functional form but different pre-processing pipelines. Hence their hyper-parameters are turned separated, each instance of the same model having very different hyper-parameters.

Model Candidiate	Normalise by Race	Normalise by Profile	Log	Box Cox	PCA
RACBoost	18.01%	18.13%	18.06%	17.95%	18.10%
CL_Ridge	-99.2%	-95.54%	-99.26%	-99.54%	-99.2%
CL_CL_Ridge	1.65%	0.77%	4.07%	3.22%	2.19%

Model Candidiate	Normalise by Race	Normalise by Profile	Log	Box Cox	PCA
SVR_CL	2.38%	-1.29%	2.07%	-0.87%	0.0%
SVC_CL	0.74%	3.47%	0.0%	2.09%	3.02%
CL_Frailty	-93.23%	-72.31%	-75.74%	-97.23%	-96.25%
XGBoost	-85.11%	-16.27%	-12.5%	-75.2%	-99.81%
CatBoost	-86.39%	-54.29%	-49.27%	-90.55%	-99.81%
XGBoost_CL	6.45%	4.76%	4.65%	1.38%	7.24%
CatBoost_CL	10.2%	6.15%	0.54%	7.98%	6.3%

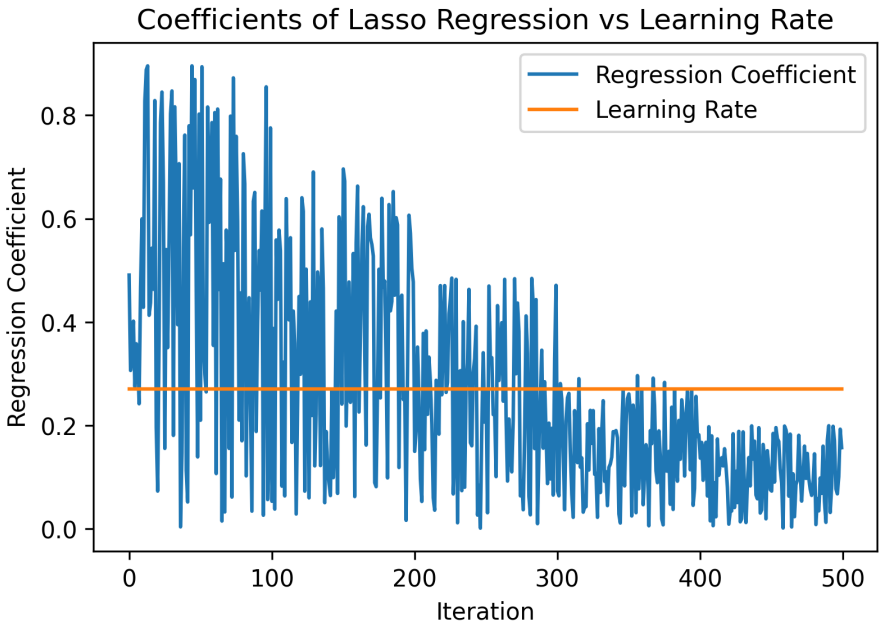
Table 2 : Candidate Model x Pre-Processing Pipeline
Return on Investment Results.

Form the table above, we observe gradient boosting models in general, have a more considerable variation in its performance. We notice that the performance of our proposed method has a smaller variability across different pre-processing pipelines compared with other methods. This reflects the model is relatively agnostic towards different transformations of the data. This is likely due to the neural network component of the model handling different styles of transformations. This characteristic will allow us to search through less pre-processing pipelines before reaching an optimal performance in production. However, it must be emphasised that the performance across different pre-process pipelines heavily depends on the dataset used in this experiment, which may not generalise to other use-cases.

Next, we analyse the effect of the post-processing step. In the step, we utilised a LASSO regression to allow for different learning rate across different iterations. It allows the model to combine the base-learners more effectively. To assess the effect of the process, we investigate the regression coefficients of the LASSO regression visualised in Figure 3.

Figure 3 : Coefficient visualisation of LASSO regression. The learning rate is obtained through cross-validation hyper-parameter selection.

The above figure visualises the regression coefficients of the LASSO post-processing model. It reflects the most efficient way to combine the



collection of pre-processors and base leaners in terms of out-of-sample performance. Traditionally, we combine the base-learners with a learning rate that is constant across different iterations. However, we can see that the most efficient combination does not have a constant multiplier to each base-learner. Such contrast indicates we are improving on the traditional method. We noticed that some of the regression coefficients are zero, caused by the LASSO penalty. This means they do not contribute to predicting the target, this is likely due to the part of the problem they attempted to solve is already covered by other iterations. When using the model, they can be dropped for efficiency. Without the post-process step, we would not be able to identify such opportunities to reduce computation time. A potential way of further regularising the model is to set a threshold for the coefficients, dropping all models with coefficients below it. Another observation is the decaying pattern of the regression coefficients, with the coefficients decreasing and the number of zero coefficients increasing. It reflects the usefulness of the base-leaners decrease as the number of iterations increase. Such effect is expected as the model learns more variability in the model, it starts to model the noise in the data hence its out-of-sample usefulness decreases.

5. Results And Discussion

5.1 Implementation Of A Handicapping System

This section discusses the viability of building a fully automated handicapping system using the techniques mentioned in the above sections to profit in the horse racing market. An automated system is superior to the traditional method because of its lack of human input required, only requiring periodic maintenance. This lack of human involvement also increases the consistency in results compared to traditional methods which fluctuate with the handicapper's form.

In order to build such a system, several components are required, which includes an automated data collection component, a predictive modelling component, and a testable wagering system component. All components should be efficient and consistent to ensure desirable results.

Because of the empirical nature of horse racing, an automated data scrapping system can be utilised to ensure the data collected and features generated are consistent throughout time. The increase in data availability online eases the development of such a system. The amount of time between data availability and deciding on a bet is minimal, usually minutes before a race. This leads to an automated system being more efficient when scrapping data from multiple sources. Data scraped from such systems should be used for the modelling component to ensure the feature distribution in training is the same as the distribution when placing bets.

The second component is a consistent predictive model that takes in a feature set and returns a vector of predicted probability of winning a race. Multiple predictive models mentioned above, including the one proposed in this paper, can be adopted for this component. An ensemble of all these methods will likely yield a better result. However, an essential point of caution is covariate shifting, in which the distribution of the raw data, hence the feature set is shifting over time. This is a common phenomenon in modelling real-life activities such as horse racing because of a different set of horses

race across different racing seasons. This change in data distribution affects the consistency of our prediction results. We partially addressed this issue in the model by batch normalising the output of each layer in the feature augmentation neural network, causing the model to be more stable to covariate shifts. However, a more thorough approach would be retraining the whole system from scratch periodically. Given the significant change in horses and a long layoff time between racing seasons, it is suggested to retrain the entire model after each racing season.

Most handicapping pipelines includes data scraping, machine learning model and wagering strategy, which are published online and easily reproducible. Therefore any advancement in modelling will be easily absorbed into the market, reflected in the market odds. This renders predictions made by the model not yielding a positive expected return. The only meaningful proprietary component in the system is the features engineered. This component has to rival the public in terms of sophistication, being the main differentiator among models. Hence the development of features is difficult and time-consuming. Therefore, the feature engineering process is the area in which betters should invest time for continuous improvement, including factors the public identified as necessary.

Given the nature of horse race bettings, the market odds fluctuate in no small magnitude until the market closes. This means we do not have the final market odds at the time we place a bet. This is a practical production issue of utter importance because of the massive significance of market odds in the predictive modelling and wagering component of the system. This is an area in which many researchers do not address. This leads to prediction speed as one of the most crucial metrics in selecting the appropriate predictive model besides profitability. The speed of prediction determines the buffer we need to allow between data scrapping and placing a bet before the market closes. Hence a faster prediction time could allow us to scrap the market odds and improve the estimates. The speed of prediction will be addressed in further sections.

Finally, we need a testable wagering system to select the right bets and optimal amount to bet that generate positive returns using the estimated probabilities. The system needs to be testable for model estimation, in which the model is optimised for profit using the wagering system. In predictive models, they usually yield a probability of a horse winning a particular race. A predetermined threshold is set to convert them into a class label. However, in horse racing, the payoff of each horse in a race is asymmetrical, leading to the need for a more conservative wagering strategy that takes into account the market odds of each horse. Among the wagering strategies having such characteristic, the Kelly Criterion mentioned above is the widely used wagering system as it has a zero chance of ruin. Given the stochastic nature of horse racing, it is guaranteed that losing streaks of various duration will occur. It is suggested to bet on a fractional Kelly Criterion. Given that the Kelly Criterion will eliminate many bets from the pool, there is a need to expand the number of betting opportunities. Exotic bets could be considered by further extending the winning probabilities to finishing order probabilities.

5.2 Limitations And Future Work

This section discusses the potential limitations of our methodology that may limit its applicability in production. Two limitations were identified along with methods that may alleviate them as follows :

The first limitation is the computational time. For both training and testing, the computational power required is significantly higher than any other conditional logit or gradient boosting model. This is caused by the training of both neural nets and decision trees in each iteration. This amounts to training n_estimators more neural nets compared to traditional gradient boosting algorithms. The importance of fast prediction time is established in the above sections. There are multiple methods to reduce training and predicting time.

The easiest method is the implementation of alternative base-learners. When implementing the model, we used simple decision trees from Sklearn (Pedregosa et al., 2011) in the base-learner step for ease of coding. Embedding more sophisticated trees from other packages for example

XGBoost (Chen and Guestrin, 2016) and CatBoost (Prokhorenkova et al., 2018) may improve the results and more importantly significantly reduce the prediction time.

The next method is the implementation of incremental learning. In many marketing applications of discrete choice modelling, such as advertisement recommenders and behaviour targeting, the task is usually time-dependent with real-time user-generated data. We often require a prediction immediately after receiving a new data point for the user. For example, a system may require advertising suggestions after clicking on a product page online. Given the slow training time of the model, it is not practical to retrain the model with the whole dataset. With incremental learning techniques applied to gradient boosting suggested by Zhang et al. (2019), models that can more efficiently incorporate new data points can be applied more practically. This is implemented by lazily updating the individual trees of a trained GBDT ensemble model to form a new model.

Second, despite the power in forecast accuracy, the model and neural-network-based models tend to be a black box, meaning there is a lack of interpretability compared to the traditional discrete choice model. Although the importance of individual features of our model can be estimated in the same way as other gradient boosting algorithms, they will be done on the augmented features. This means there is no way to track it back to the original features. This issue is of importance because many discrete choice problems are economical in nature. The interpretation of the model is just as crucial as the predictive power of the model, as many economic theories are based on the traditional conditional logit model functional form.

A method to remedy the situation is the use of model agnostic model interpretation techniques such as Accumulated Local Effects plots, proposed by Apley D. W, Zhu J. (2019). It graphs how a particular feature influences the predicted outcome on average for a set of correlated features. Using such plots, we can roughly interpret how will the prediction change with the input. However, we lose the interpretability of econometric implications associated with discrete choice theory.

6 Conclusion And Direction For Future Research

In this paper, we aimed to bridge the gap between discrete choice modelling and modern machine learning techniques. We presented different modifications to the classical gradient boosting decision tree algorithm by augmenting the initialisation, pre-processing and post-processing step.

We also conducted a case study on the Hong Kong Horse Racing market. It demonstrated the prediction performance of classical discrete choice models and gradient boosting decision trees could be improved by our proposed method. Such improvement is likely due to better handling of the dominant feature and the augmentation of the feature space allowing the decision trees to capture more complex non-linear structures as they have a completely different view of the data at each iteration.

References

Apley D. W, Zhu J. (2019) *Visualizing the Effects of Predictor Variables in Black Box Supervised Learning Models.*

Benter W. (1994). *Computer Based Horse Race Handicapping and Wagering Systems.* Academic Press: London; 1994. pp.183-198.

Bolton R. N. and Chapman R. G. (1986). *Searching for positive returns at the track: A multinomial logit model for handicapping horse races.* Management Science (1986) 32 1040–1060.

Breiman. L. (2001). Random forests. Machine Learning, 45(1):5–32, Oct. 2001.

Chen H., Lundberg S. and Lee S. (2018). *Hybrid Gradient Boosting Trees and Neural Networks for Forecasting Operating Data.* arVix preprint arVix:1801.07384, 2018

Chen T. and Guestrin C. (2016). *Xgboost: A scalable tree boosting system.* In Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining, pages 785–794. ACM, 2016.

Friedman J. (2001). *Greedy Function Approximation: A Gradient Boosting Machine.* The Annals of Statistics, 29(5):1189–1232.

Friedman J. and Popescu B. (2003). *Importance Sampled Learning Ensembles.*

Haolun Shi and Guosheng Yin (2018). *Boosting Conditional Logit Model.* Journal of Choice Modelling 26 (2018) 48-63.

Kelly J. L. (1956). *A new interpretation of information rate.* The Bell System Technical Journal (1956) 35 917 – 926.

Lessmann S., Ming-Chien Sung (2009). *Identifying winners of competitive events: A SVM-based classification model for horserace prediction.* European Journal of Operational Research 2009(7).

Lessmann S., Ming-Chien Sung and Johnson E.V. (2007). *Adapting Least-Square Support Vector Regression Models to Forecast the Outcome of Horseraces.* The Journal of Prediction Markets 2007;3;169-187.

McFadden, D. (1974). *Conditional logit analysis of qualitative choice behavior.* In: Zarembka, P. (Ed.), Frontiers in Econometrics. Academic Press, New York, pp. 105–142.

Mohan A., Chen Z. and K. Weinberger (2011). *Web-Search Ranking with Initialized Gradient Boosted Regression Trees.* JMLR Workshop and Conference Proceeding 14 2011 77-89

Pan S. J. and Yang Q. (2009). *A Survey on Transfer Learning.*

Pedregosa et al. (2011). *Scikit-learn: Machine Learning in Python.* JMLR 12, pp. 2825-2830, 2011.

Prokhorenkova L., Gusev G, Vorobev A, Dorogush A. V and Gulin A. (2018). *Catboost : unbiased boosting with categorial features.* 32nd Conference on Neural Information Processing Systems (NeurIPS 2018), Montréal, Canada.

Raghu M., Poole B., Kleinberg J., Ganguli S. and Dickstein J. S. (2017). *On the Expressive Power of Deep Neural Networks.* Proceedings of the 34th International Conference on Machine Learning, PMLR 70:2847-2854, 2017.

Shor N. (1970) *Convergence rate of the gradient descent method with dilatation of the space.* Cybernetics and Systems Analysis, 6(2):102–108, 1970.

Sifringer B., Lurkin L. and Alahi A. (2018). *Enhancing Discrete Choice Models with Neural Networks.* 18th Swiss Transport Research Conference, May 2018.

Silverman N. (2013). *Optimal Decisions with Multiple Agents of Varying Performance.*

Snoek J, Larochelle H. and Adams R. P. (2012). *Practical bayesian optimisation of machine learning algorithms.* In Advances in neural Information processing systems (pp. 2951-2959).

Tannor and Lior Rokach (2019). *AugBoost: Gradient Boosting Enhanced with Step-Wise Feature Augmentation.* Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence (IJCAI-19)

Zhang C., Zhang Y., Shi X., Almpanidis G., Fan G., Shen X. (2019). *On Incremental Learning for Gradient Boosting Decision Trees.* Springer Science and Business Media, LLC, part of Springer Nature 2019

Appendix I : List of Hyper-parameters

Hyper-Parameter	Description
reg_lambda_l1 λ_{L1}	L1 regularisation term of weights of neural net Range : [0 - ∞)
reg_lambda_l2 λ_{L2}	L2 regularisation term of weights of neural net Range : [0 - ∞)
nn_complexity ψ	Regularisation term on the complexity of the neural net Range : (-∞ - ∞)
reg_leaf ϕ	Regularisation term on the values in terminal nodes in regression tree. Range : [0 - ∞)
ignore_features	Features to exclude from the training process Default : dominating feature.
nn_count	Number of neurons in the first hidden layer of the neural network in the feature augmentation step Range : [1 - ∞)
nn_shrink	Scaler for number of neurons in subsequent hidden layers of the neural network in the feature augmentation step Range : (0 - 1)
nn_size	Scaler for nn_count between iterations Range : [0 - ∞)
max_depth	Maximum depth of the individual regression tree base estimators Range : [1 - ∞)
learning_rate ρ	Boosting learning rate that shrinks the contribution of each tree Range : (0 - ∞)
n_estimators M	The number of boosting stages to perform Range : [1 - ∞)
post_lambda λ_{post}	L1 regularisation term on the base-learners in the post-processing step Range : [0 - ∞)
row_subsample	The fraction of training instances used in the feature augmentation step Range : (0 - 1]
col_subsample	The fraction of columns used in the feature augmentation step Range : (0 - 1]

Appendix II : RACBoost

https://github.com/ivanyhto/RACBoost

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
@author: ivan
"""

import time
import random
import warnings
import numpy as np
import pandas as pd
from collections import OrderedDict
from scipy.special import logsumexp
from pylogit import create_choice_model

from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam
from tensorflow.keras import Input, regularizers
from tensorflow.keras.losses import MeanSquaredError
from tensorflow import convert_to_tensor, GradientTape

from sklearn.tree._tree import TREE_LEAF
from sklearn.tree import DecisionTreeRegressor
from sklearn.base import BaseEstimator, ClassifierMixin


class RACBoost(BaseEstimator, ClassifierMixin):
    """
    Regularised Augmented Conditional Boosting Model

    Parameters
    ----------
    alt_id_col :str.
        Denote the column in data X which contains the alternative
        identifiers for each row.

    obs_id_col : str.
        Denote the column in data X which contains the observation
        identifiers for each row.

    choice_col : str.
        Denote the column in data y which contains the ones and zeros that
        denote whether or not the given row corresponds to the chosen
        alternative for the given individual.

    loss : object, default : Regularised Conditional Logit Loss Function
        Loss Function to be optimized.

    reg_lambda_l1 : float, default=0.0
        L1 Regularisatio nterm of weight of neural net.

    reg_lambda_l2 : float, default=0.0
        L2 Regularisatio nterm of weight of neural net.

    nn_complexity : float, default=0.0
        Regularisation term on the complexity of the nerual net.
```

```
58
59      reg_leaf : int, default=0
60          Regularisation term on the terminal nodes in regression tree.
61
62      ignore_features : list / str, default='dominant'
63          Features to exclude from the training process
64
65      nn_count : int, default='n_features'
66          Number of neurons in the first hidden layer of the neural net in the
67          feature augmentation step
68
69      nn_shrink : float, default=0.3
70          Scaler for number of neurons in subsequent hidden layers of the neural
71          net in the feature augmentation step
72
73      nn_size : float, default=1
74          Scaler for nn_count between iterations
75
76      max_depth : int, default=3
77          Maximum depth of the individual regression tree base learners.
78          The maximum depth limits the number of nodes in the tree
79
80      learning_rate : float, default=0.1
81          Boosting learning rate that shrinks the contribution of
82          each regression tree base learners.
83          There is a trade-off between learning_rate and n_estimators.
84
85      n_estimators : int, default=500
86          The number of boosting stages to perform.
87
88      post_lambda : float, default=0.0
89          L1 regularisation term on the base learners in the post-processing step
90
91      row_subsample : float, default=1.0
92          The fraction of training instances used in feature augmentation step
93          If smaller than 1.0 this results in Stochastic Gradient Boosting.
94
95      col_subsample : float, default=1.0
96          The fraction of columns used in the feature augmentation step
97
98      random_state : int, default = None
99          Controls the random seed given to each regression tree base learners.
100         Controls the random seed given to each neural net feature augmentor.
101
102     verbose : Bool, defualt=True
103         Enable verbose output.
104         If True then prints progress and performance
105
106     Attributes
107     ----------
108     features_list_ : list
109         List of features from column names in dataset.
110
111     n_features_ : int
112         Nuumber of features in the dataset
113
114     dominant_feature_ : str
115         Feature Name of the dominant feature identified
116
117     loss_ : LossFunction
118         The concrete ititialised LossFunction object.
119
120     train_score_ : ndarray of training loss function score
121
122     feature_aug_ : ndarray of Keras Neural Nets, shape(n_estimators,1)
123         The collection of fitted feature augmentation Keras Neural Nets
124
```

```
    estimators_ : ndarray of DecisionTreeRegressor, shape (n_estimators,1)
        The collection of fitted base learners.

    decision_function :

    """

    def __init__(
            self,
            alt_id_col,
            obs_id_col,
            choice_col,
            loss=None,
            reg_lambda_l1=0.0,
            reg_lambda_l2=0.0,
            nn_complexity=0.0,
            reg_leaf=0.0,
            ignore_features='dominant',
            nn_count='n_features',
            nn_shrink=0.3,
            nn_size=1.0,
            max_depth=3,
            learning_rate=0.1,
            n_estimators=500,
            post_lambda=0.0,
            row_subsample=1.0,
            col_subsample=1.0,
            random_state=None,
            verbose=1):

        self.alt_id_col = alt_id_col
        self.obs_id_col = obs_id_col
        self.choice_col = choice_col
        self.loss = loss

        self.reg_lambda_l1 = reg_lambda_l1
        self.reg_lambda_l2 = reg_lambda_l2
        self.nn_complexity = nn_complexity
        self.reg_leaf = reg_leaf
        self.ignore_features = ignore_features
        self.nn_count = nn_count
        self.nn_shrink = nn_shrink
        self.nn_size = nn_size
        self.max_depth = max_depth
        self.learning_rate = learning_rate
        self.n_estimators = n_estimators
        self.post_lambda = post_lambda
        self.row_subsample = row_subsample
        self.col_subsample = col_subsample

        self.random_state = random_state
        self.verbose = verbose

        return None

    def fit(self, X, y):
        """
        Iteratively fit the stages of the Gradient Boosting Model
        For each iteration, it prints the progress score if verbose = 1

        Parameters
        ----------
        X : Pandas DataFrame with alt_id_col and obs_id_col
            of the shape (n_samples, n_features + 2)
            the dataframe should be in long format for the discrete choice model
            The input samples.
```

```
        y : Pandas DataFrame with alt_id_col, obs_id_col and choice_col
            of the shape (n_samples, 3)
            the dataframe should be in long format for the discrete choice model
            The targets.
        Returns
        -------
        self : object

        """
        # Initiate Verbose Reporter
        if self.verbose:
           self.verbose_reporter = VerboseReporter()

        # Base Attributes
        self.X_base_cols_ = [self.alt_id_col, self.obs_id_col]
        self.y_base_cols_ = [self.alt_id_col, self.obs_id_col, self.choice_col]
        self.individuals_ = list(X.loc[:, self.obs_id_col].unique())
        self.n_individuals_ = len(self.individuals_)
        self.features_list_ = list(X.columns.drop(self.X_base_cols_))
        self.n_features_ = len(self.features_list_)

        # Check input
        self._validate_data(X, y)

        # Check Parameters
        self._check_params()

        # Step 1 : Initialize model
        raw_predictions = self._initialisation(X, y)

        # Iteratively fits the boosting stages
        for i in range(self.n_estimators):

            # A copy is passed to negative_gradient() because raw_predictions
            # is updated at the end of the loop in Step 5
            raw_predictions_copy = raw_predictions.copy()

            """
            Step 2 : Computeing Negative Residual
            """
            residuals = self.loss_._negative_gradient(y, raw_predictions_copy)

            """
            Step 3 : Stage-wise Neural Net Feature Augmentation
            """
            features, nn, nn_complexity_measure, col_idx = \
                self._feature_augmentation(X, residuals)

            """
            Step 4 : Fitting Base Learner to Negative Gradient
            """
            tree = self._fit_base_learner(features, residuals)

            """
            Step 5 : Line Search Updating terminal regions
            """
            terminal_values = self._line_search(tree.tree_, features, \
                residuals, raw_predictions, nn_complexity_measure)

            """
            Step 6 : Update via Shrinkage
            """
            raw_predictions = self._update(i, terminal_values, raw_predictions)

```

```
            """
            Step 7 : Save Iteration
            """
            self._save_iteration(i, nn, tree, col_idx, y, raw_predictions)

        """
        Step 8 : LASSO Post-Processing
        """
        self._post_processing(y)

        return self

    def _validate_data(self, X, y):
        """
        Validate Input data : Datatype and existance of alt_id_col, obs_id_col
        Parameters
        ----------
        X : Pandas DataFrame with alt_id_col and obs_id_col
            of the shape (n_samples, n_features + 2)
            the dataframe should be in long format for the discrete choice model
            The input samples.

        y : Pandas DataFrame with alt_id_col and obs_id_col
            of the shape (n_samples, 3)
            the dataframe should be in long format for the discrete choice model
            The targets.
        """
        # Checks whether X amd y are Pandas DataFrames
        if not isinstance(X, pd.DataFrame):
            raise TypeError("X must be a pd.DataFrame but "
                            f"was {type(X)}")

        if not isinstance(y, pd.DataFrame):
            raise TypeError("y must be a pd.DataFrame but "
                            f"was {type(y)}")

        # Checks whether alt_id_col and obs_id_col is in Dataframes X and y
        problem_cols_X = [
            col for col in self.X_base_cols_ if col not in X.columns]
        problem_cols_y = [
            col for col in self.y_base_cols_ if col not in y.columns]

        if problem_cols_X != []:
            raise ValueError("The following columns in are not in "
                             f"X.columns : {problem_cols_X}")

        if problem_cols_y != []:
            raise ValueError("The following columns in are not in "
                             f"X.columns : {problem_cols_y}")

        return None

    def _check_params(self):
        """
        Check validity of parameters and raise ValueError if not valid.
        """

        if self.reg_lambda_l1 < 0.0:
            raise ValueError("reg_lambda_l1 must be non-negative but "
                             f"was {self.reg_lambda_l1}")

        if self.reg_lambda_l2 < 0.0:
            raise ValueError("reg_lambda_l2 must be non-negative but "
                             f"was {self.reg_lambda_l2}")

```

```
        if type(self.nn_complexity) != np.float64 and type(self.nn_complexity) != floa
            raise ValueError("nn_complexity must be a floating point nuumber but "
                             f"was {self.nn_complexity}")

        if self.reg_leaf < 0.0:
            raise ValueError("reg_leaf must be non-negative but "
                             f"was {self.reg_leaf}")

        if self.ignore_features != 'dominant':
            if not isinstance(self.ignore_features, list):
                raise TypeError("ignore_feature must be a list but "
                                f"was {type(self.ignore_features)}")

            # Check whether all ignored features are in the feature list
            problem_cols = [
                col for col in self.ignore_features if col not in self.features_list_]
            if problem_cols != []:
                raise ValueError("The following columns in are not in "
                                 f"feature list : {problem_cols}")

        if self.nn_count != 'n_features':
            if self.nn_count < 1:
                raise ValueError(
                    "nn_count must be greater than 0 or 'n_features' but "
                    f"was {self.nn_count}")

        if not (0.0 < self.nn_shrink < 1.0):
            raise ValueError("nn_shrink must be in (0,1) but "
                             f"was {self.nn_shrink}")

        if self.nn_size <= 0.0:
            raise ValueError("nn_size must be greater than 0 but "
                             f"was {self.nn_size}")

        if self.max_depth < 1:
            raise ValueError("max_depth must be greater than 0 but "
                             f"was {self.max_depth}")

        if self.learning_rate <= 0.0:
            raise ValueError("learning_rate must be greater than 0 but "
                             f"was {self.learning_rate}")

        if self.n_estimators < 1:
            raise ValueError("n_estimators must be at least 1 but "
                             f"was {self.n_estimators}")

        if self.post_lambda < 0.0:
            raise ValueError("post_lambda must be non-negative but "
                             f"was {self.post_lambda}")

        if not (0.0 < self.row_subsample <= 1.0):
            raise ValueError("row_subsample must be in (0,1] but "
                             f"was {self.row_subsample}")

        if not (0.0 < self.col_subsample <= 1.0):
            raise ValueError("col_subsample must be in (0,1] but "
                             f"was {self.col_subsample}")

        return None

    def _initialisation(self, X, y):
        """
        Step 1 : Initialize model state by identifying the dominant feature
        Parameters
        ----------
```

```
        X : Pandas DataFrame with alt_id_col and obs_id_col
            of the shape (n_samples, n_features + 2)
            the dataframe should be in long format for the discrete choice model
            The input samples.
        y : Pandas DataFrame with alt_id_col, obs_id_col and choice_col
            of the shape (n_samples, 3)
            the dataframe should be in long format for the discrete choice model
            The targets.
        Returns
        -------
        raw_predictions : Pandas DataFrame with alt_id_col, obs_id_col,
            utility (estimated utility function) and utility_obs
            (sum of all utilities in obs) of the shape (n_samples, 4)
            the dataframe should be in long format for the discrete choice model
        """
        # Initial Loss function and Predictior
        self.loss_ = self.loss(
            alt_id_col=self.alt_id_col,
            obs_id_col=self.obs_id_col,
            choice_col=self.choice_col)
        self.init_ = self.loss_.init_estimator()

        self.loss_.init_estimator_warnings('ignore')

        init_set = y.merge(X)

        max_likelihood = -np.inf
        for col in self.features_list_:
            spec = OrderedDict()
            spec[col] = 'all_same'
            zeros = np.zeros(len(spec))
            model = self.init_(data=init_set.loc[:, self.y_base_cols_ + [col]],
                               alt_id_col=self.alt_id_col,
                               obs_id_col=self.obs_id_col,
                               choice_col=self.choice_col,
                               specification=spec,
                               model_type="MNL")
            model.fit_mle(zeros, print_res=False)
            if model.log_likelihood > max_likelihood:
                max_likelihood = model.log_likelihood
                self.dominant_model = model
                self.dominant_feature_ = col

        # Initialise raw_predictions
        raw_predictions = X.loc[:, self.X_base_cols_]
        raw_predictions.loc[:,'utility'] = self.dominant_model.params.values \
            * X.loc[:, self.dominant_feature_]
        raw_predictions.loc[:, 'utility_obs'] = self.loss_._utility_obs(
            raw_predictions)

        self.loss_.init_estimator_warnings('default')

        # Initialise decision functino to store terminal regions
        self.decision_function = raw_predictions.loc[:,
                                             self.X_base_cols_ + ['utility']]
        self.decision_function.columns = self.X_base_cols_ + ['init']

        # Initialise length of train_score_ vector
        self.train_score_ = np.zeros((self.n_estimators,), dtype=np.float64)

        # Initialise length of feature_aug_ vector
        self.feature_aug_ = np.empty((self.n_estimators,), dtype=np.object)

        # Initialise length of col_idx_ vector
        self.col_idx_ = np.empty((self.n_estimators,), dtype=np.object)
```

```
450
451         # Initialise length of estimators_ vector
452         self.estimators_ = np.empty((self.n_estimators,), dtype=np.object)
453
454         # Verbose
455         if self.verbose:
456             self.verbose_reporter.start()
457
458         return raw_predictions
459
460     def _feature_augmentation(self, X, residuals):
461         """
462         Step 3 : Stage-wise Neural Net Feature Augmentation
463         Parameters
464         ----------
465         X : Pandas DataFrame with alt_id_col and obs_id_col
466             of the shape (n_samples, n_features + 2)
467             the dataframe should be in long format for the discrete choice model
468             The input samples.
469         residuals : Pandas DataFrame with alt_id_col, obs_id_col and residual
470             of the shape (n_samples, 3)
471             he dataframe should be in long format for the discrete choice model
472         Returns
473         -------
474         features : Pandas DataFrame without any id of the shape (n_samples, #nn_feature
475             the dataframe should be in long format for the discrete choice model
476         nn : Keras Neural Net model
477             A fitted Keras Neural Net Model
478         """
479
480         """
481         Subsampling
482         """
483         # Random state is not set in sampling to allow for differernt samples / feature
484         # Subsampling rows
485         if self.row_subsample < 1:
486             # Number of individuals / samples
487             num_samples = int(self.n_individuals_ * self.row_subsample)
488
489             # Sampling obs_id_col
490             row_idx = random.sample(self.individuals_, num_samples)
491         else:
492             row_idx = self.individuals_
493
494         # Subsampling columns
495         if self.col_subsample < 1:
496
497             # Number of columns
498             num_cols = int(self.n_features_ * self.col_subsample)
499
500             if self.ignore_features != []:
501                 if self.ignore_features == 'dominant':
502                     # Sampling features without the dominant term
503                     col_idx = random.sample(
504                         [col for col in self.features_list_ if col != self.dominant_fea
505                 else:
506                     # Sampling features ignoring a list of features
507                     col_idx = random.sample(
508                         [col for col in self.features_list_ if col not in self.ignore_f
509             else:
510                 # Sampling features
511                 col_idx = random.sample(self.features_list_, num_cols)
512         else:
513             col_idx = self.features_list_
514
515         subsampled_X = X.loc[X[self.obs_id_col].isin(row_idx), col_idx + [self.obs_id_c
516         subsampled_y = residuals.loc[residuals[self.obs_id_col].isin(row_idx), :]
```

```
517
518         """
519         Neural Net Model by a Custom Training Loop
520         The custom loop is used to incoperate the obs_ids in the loss function
521         Also control individuals are grouped together in a batch
522         """
523         # Formatting dataset
524         # Split obs_ids in batches according to obs
525         X_columns = [
526             col for col in subsampled_X.columns if col not in self.X_base_cols_]
527         train_dataset = [(subsampled_X.loc[subsampled_X[self.obs_id_col] == obs,
528                                            X_columns].values,
529                           subsampled_y.loc[subsampled_y[self.obs_id_col] == obs,
530                                            'residual']) for obs in self.individuals_]
531         input_dim = len(X_columns)
532
533         # Create model instance
534         model, nn_complexity_measure = self._generate_nn_model(input_dim)
535
536         # Initialise an optimizer
537         optimizer = Adam()
538
539         # Initialise an Loss Function
540         loss_fn = MeanSquaredError(reduction="auto", name="mean_squared_error")
541
542         epochs = 10
543         # Fitting the model
544         for epoch in range(epochs):
545
546             # Iterate over the batches of the dataset.
547             for step, (x_batch_train, y_batch_train) in enumerate(train_dataset):
548
549                 # train_step(x_batch_train, y_batch_train)
550
551                 # A GradientTape is used to record the operations during each
552                 # forward pass
553                 with GradientTape() as tape:
554
555                     # Run the forward pass
556                     y_pred = model(x_batch_train, training=True)
557
558                     # Compute the loss function for this batch.
559                     # Future Development : This allows for adding regularisation
560                     # parameter to change the loss function between epoches
561                     y_true = convert_to_tensor(y_batch_train.values.reshape(
562                         y_batch_train.size, 1), dtype='float32')
563                     loss_value = loss_fn(y_true, y_pred)
564
565                 # Use the gradient tape to automatically retrieve the gradients of the
566                 # trainable variables with respect to the loss.
567                 grads = tape.gradient(loss_value, model.trainable_weights)
568
569                 # Run one step of gradient descent by updating
570                 # the value of the variables to minimize the loss.
571                 optimizer.apply_gradients(zip(grads, model.trainable_weights))
572
573         # Getting feature set
574         extractor = Model(inputs=model.inputs, outputs=model.layers[-2].output)
575         features = extractor.predict(X.loc[:, col_idx].values)
576
577         # Updating neuron counts for next iteration
578         self.nn_count *= self.nn_size
579
580         return features, extractor, nn_complexity_measure, col_idx
581
582     def _generate_nn_model(self, input_dim):
583         """
```

```
        Generate model based on specifications
        Parameters
        ----------
        input_dim : int
        Dimension of input data

        Returns
        -------
        model : keras model object

        nn_complexity_measure : float
            Measure of complexity
        """
        # Define size of Input
        inputs = Input(shape=(input_dim,))

        # Layer 1 Neuron Count
        if self.nn_count == 'n_features':
            neuron_count = self.n_features_
            self.nn_count = neuron_count
        else:
            neuron_count = self.nn_count


        nn_complexity_measure = 0
        # Add Layers
        layer = inputs
        while neuron_count > 30:

            nn_complexity_measure += input_dim * np.log(int(neuron_count))
            layer = Dense(int(neuron_count),activation='relu',
                          kernel_regularizer=regularizers.l1_l2(
                              l1=self.reg_lambda_l1,l2=self.reg_lambda_l2))(layer)

            #Update Input dimension of next layer
            input_dim = int(neuron_count)

            # Shrink neuron count for next layer
            neuron_count *= self.nn_shrink
        # Add final Layer
        output = Dense(1, activation='sigmoid')(layer)
        model = Model(inputs=inputs, outputs=output)

        return model, nn_complexity_measure

    def _fit_base_learner(self, features, residuals):
        """
        Step 4 : Fitting Base Learner to Negative Gradient
        Parameters
        ----------
        X : Pandas DataFrame with features
            of the shape (n_samples, n_features)
            the dataframe should be in long format for the discrete choice model
            The input samples.
        residuals : Pandas DataFrame with alt_id_col, obs_id_col and residual
            of the shape (n_samples, 3)
            the dataframe should be in long format for the discrete choice model
        Returns
        -------
        tree : trained sklearn tree object
        """
        # Init Regression Tree
        tree = DecisionTreeRegressor(max_depth=self.max_depth,
                                     splitter='best',
                                     random_state=self.random_state)

        # Fit Model
        tree.fit(features, residuals.loc[:, 'residual'])
```

```
652
653             return tree
654
655         def _line_search(self, tree, features, residuals, raw_predictions, nn_complexity_m
656             """
657             Step 5 : Line Search Updating terminal regions
658             This function calls the loss.update_terminal_region for each leaf
659             """
660             #Compute leaf values for each sample in dataset, getting the leaf index
661             terminal_regions = tree.apply(np.array(features, dtype='float32'))
662
663             #neural net complexity penalty
664             complexity_penalty = self.nn_complexity * nn_complexity_measure
665
666             # update each leaf (= perform line search)
667             for leaf in np.where(tree.children_left == TREE_LEAF)[0]:
668                 self.loss_._update_terminal_region(tree, terminal_regions, leaf,
669                                                    raw_predictions, residuals, self.reg_le
670
671             terminal_values = tree.value[:, 0, 0].take(terminal_regions, axis=0)
672
673             return terminal_values
674
675         def _update(self, i, terminal_values, raw_predictions):
676             """
677             Step 6 : Update via Shrinkage
678             """
679             #Update raw_prediction
680             raw_predictions.loc[:,'utility'] += self.learning_rate * terminal_values
681             raw_predictions.loc[:,'utility_obs'] = self.loss_._utility_obs(raw_predictions
682
683             #Save a copy of terminal values
684             self.decision_function.loc[:,str(i)] = terminal_values
685
686             return raw_predictions
687
688         def _save_iteration(self, i, nn, tree, col_idx, y, raw_predictions):
689             """
690             Step 7 : Save components in the iteration
691             """
692             # Adding NN to ensemble
693             self.feature_aug_[i] = nn
694
695             # Saving columns used in nn
696             self.col_idx_[i] = col_idx
697
698             # Adding Tree to ensemble
699             self.estimators_[i] = tree
700
701             # Updating train_score_
702             self.train_score_[i] = self.loss_.loss(y, raw_predictions)
703
704             # Verbose
705             if self.verbose:
706                 self.verbose_reporter.update(i, self)
707
708             return None
709
710         def _post_processing(self, y):
711             """
712             Step 8 : LASSO Post-Processing - Be aware of singular matrix
713             """
714             X_post = self.decision_function
715             #Create specification dictionary
716             model_specification = OrderedDict()
717             for variable in X_post.columns[2:]:
718                 model_specification[variable] = 'all_same'
719             zeros = np.zeros(len(model_specification))
```

```
        X_post = X_post.merge(y.loc[:,['RARID','HNAME','RESWL']], on=['RARID','HNAME']
        X_post.reset_index(inplace=True, drop=True)

        self.loss_.init_estimator_warnings('ignore')
        model = self.init_(data=X_post,
                           alt_id_col=self.alt_id_col,
                           obs_id_col=self.obs_id_col,
                           choice_col=self.choice_col,
                           specification=model_specification,
                           model_type="MNL")
        model.fit_mle(zeros, print_res=False, ridge = self.post_lambda)

        self.post_processing_model  = model
        self.summary = self.post_processing_model.get_statsmodels_summary()
        self.loss_.init_estimator_warnings('default')

        # Verbose
        if self.verbose:
            self.verbose_reporter.complete()

        return None

    def predict(self, X):
        """
        Predict class probabilities for X.
        Parameters
        ----------
        X : Pandas DataFrame with alt_id_col and obs_id_col
            of the shape (n_samples, n_features + 2)
            the dataframe should be in long format for the discrete choice model
        Returns
        -------
        p : ndarray of shape (n_samples, n_classes)
            The class probabilities of the input samples. The order of the
            classes corresponds to that in the attribute :term:`classes_`.
        """
        raw_predictions = self._predict_utility_function(X)
        return self.loss_._raw_prediction_to_decision(raw_predictions)

    def predict_proba(self, X):
        """
        Predict class probabilities for X.
        Parameters
        ----------
        X : Pandas DataFrame with alt_id_col and obs_id_col
            of the shape (n_samples, n_features + 2)
            the dataframe should be in long format for the discrete choice model
        Returns
        -------
        p : Pandas DataFrame with alt_id_col, obs_id_col, proba
            of the shape (n_samples, n_features + 2)
            The class probabilities of the input samples. The order of the
            classes corresponds to that in the attribute :term:`classes_`.
        """
        raw_predictions = self._predict_utility_function(X)
        try :
            return self.loss_._raw_prediction_to_proba(raw_predictions)
        except AttributeError:
            raise AttributeError('loss=%r does not support predict_proba' %
                                 self.loss)

    def _predict_utility_function(self, X):
        """
        Computes the predicted utility function
        Utility Function = Sum of tree predictions + init
        Parameters
        ----------
```

```
        X : Pandas DataFrame with alt_id_col and obs_id_col
            of the shape (n_samples, n_features + 2)
        Returns
        -------
        raw_predictions : Pandas DataFrame with alt_id_col, obs_id_col, and utility
            (estimated utility function) of the shape (n_samples, 3)
            the dataframe should be in long format for the discrete choice model
        """
        #Validate data
        self._validate_testdata(X)
        raw_predictions = X.loc[:, self.X_base_cols_]
        decisions = X.loc[:, self.X_base_cols_]

        #Get initial prediction
        decisions.loc[:,'init'] = self.dominant_model.params.values * X.loc[:,self.dom

        #Get components
        nns = self.feature_aug_
        trees = self.estimators_
        col_idx = self.col_idx_

        #Predict Stages
        for i in range(self.n_estimators):
            #Pre-Processing
            nn = nns[i]
            col = col_idx[i]
            features = nn.predict(X.loc[:,col].values)

            #Base Learner
            tree = trees[i]
            decisions.loc[:,str(i)] = tree.predict(features)

        #Post-Processing
        self.loss_.init_estimator_warnings('ignore')
        raw_predictions.loc[:,'utility'] = self.post_processing_model.predict(decision
        self.loss_.init_estimator_warnings('default')

        return raw_predictions

    def _validate_testdata(self, X):
        """
        Validate Input data : Datatype and existance of alt_id_col and obs_id_col
        Parameters
        ----------
        X : Pandas DataFrame with alt_id_col and obs_id_col
            of the shape (n_samples, n_features + 2)
            the dataframe should be in long format for the discrete choice model
            The input samples.
        """
        # Checks whether X amd y are Pandas DataFrames
        if not isinstance(X, pd.DataFrame):
            raise TypeError("X must be a pd.DataFrame but "
                            f"was {type(X)}")

        # Checks whether alt_id_col and obs_id_col is in Dataframes X and y
        problem_cols_X = [
            col for col in self.X_base_cols_ if col not in X.columns]

        if problem_cols_X != []:
            raise ValueError("The following columns in are not in "
                             f"X.columns : {problem_cols_X}")

        return None

class Reg_ConditionalLogit_Loss():
    """
    Regularised Conditional Logit Loss Function
    """
```

```
856
857     def __init__(self, alt_id_col, obs_id_col, choice_col):
858         self.alt_id_col = alt_id_col
859         self.obs_id_col = obs_id_col
860         self.choice_col = choice_col
861
862     def init_estimator(self):
863         """
864         Returns
865         -------
866         Conditional Logistic Regression model class
867         """
868         return create_choice_model
869
870     def init_estimator_warnings(self, status='ignore'):
871         """
872         Changes the warnings setting for the init estimator
873         """
874         warnings.filterwarnings(status, category=FutureWarning)
875         warnings.filterwarnings(status, category=UserWarning)
876         warnings.filterwarnings(status, category=RuntimeWarning)
877
878         return None
879
880     def loss(self, y, raw_predictions):
881         """
882         Compute the Log Likelihood Function / Training Score
883         Parameters
884         ----------
885         y : Pandas DataFrame with alt_id_col and obs_id_col
886             of the shape (n_samples, 3)
887             the dataframe should be in long format for the discrete choice model
888             The targets.
889         raw_predictions : Pandas DataFrame with alt_id_col, obs_id_col, and utility
890             (estimated utility function) of the shape (n_samples, 3)
891             the dataframe should be in long format for the discrete choice model
892         """
893         data = y.merge(raw_predictions, on=[self.obs_id_col, self.alt_id_col])
894
895         # Initialise Log Likelihood
896         ll = 0
897         # Looping over all individuals
898         for _, group in data.groupby(self.obs_id_col):
899             ll += group.loc[:, 'utility'].dot(group.loc[:, 'RESWL'])
900             ll -= logsumexp(group.loc[:, 'utility'])
901
902         return ll
903
904     def _negative_gradient(self, y, raw_predictions):
905         """
906         Compute negative gradient
907
908         Parameters
909         ----------
910         y : Pandas DataFrame with alt_id_col and obs_id_col
911             of the shape (n_samples, 3)
912             the dataframe should be in long format for the discrete choice model
913             The targets.
914         raw_predictions : Pandas DataFrame with alt_id_col, obs_id_col, utility (estima
915             and utility_obs (sum of all utilities in obs) of the shape (n_samples, 4)
916             the dataframe should be in long format for the discrete choice model
917         Returns
918         -------
919         residuals : Pandas DataFrame with alt_id_col, obs_id_col and residual
920             of the shape (n_samples, 3)
921             the dataframe should be in long format for the discrete choice model
922         """
923         raw_predictions = self._overflow_guard(raw_predictions)
```

```
924
925         target = [col for col in y.columns if col not in [self.alt_id_col, self.obs_id_
926
927         residuals = y.merge(
928             raw_predictions, on=[
929                 self.alt_id_col, self.obs_id_col])
930         residuals.loc[:,'residual'] = residuals.loc[:,target] \
931             - np.exp(residuals.loc[:,'utility']) / residuals.loc[:,'utility_obs']
932
933         return residuals.loc[:, [self.alt_id_col, self.obs_id_col, 'residual']]
934
935     def _hessian(self, raw_predictions):
936         """
937         Compute second derivative of loss function
938
939         Parameters
940         ----------
941         raw_predictions : Pandas DataFrame with alt_id_col, obs_id_col, utility (estim
942             and utility_obs (sum of all utilities in obs) of the shape (n_samples, 4)
943             the dataframe should be in long format for the discrete choice model
944         Returns
945         -------
946         residuals : Pandas DataFrame with alt_id_col, obs_id_col and residual
947             of the shape (n_samples, 3)
948             the dataframe should be in long format for the discrete choice model
949         """
950         hessian = self._overflow_guard(raw_predictions)
951         hessian.loc[:,'hessian'] = (hessian.loc[:,'utility_obs'] \
952                                     - np.exp(hessian.loc[:,'utility']) ** 2) / hessian
953
954         return hessian.loc[:, 'hessian']
955
956     def _update_terminal_region(
957             self,
958             tree,
959             terminal_regions,
960             leaf,
961             raw_predictions,
962             residual,
963             reg_leaf,
964             complexity_penalty):
965         """
966         Update the terminal regions (=leaves) of the given tree
967         Mutates the tree
968         Parameters
969         ----------
970         tree : tree.Tree
971             The tree object
972         """
973         terminal_region = np.where(terminal_regions == leaf)[0]
974
975         utility_m_1 = raw_predictions.take(terminal_region, axis=0)
976         y_true = residual.take(terminal_region, axis=0)
977
978         gradient = self._negative_gradient(y_true, utility_m_1)
979         hessian = self._hessian(utility_m_1)
980
981         numerator =  np.sum(gradient.loc[:, 'residual'])  # -1*
982         denominator = np.sum(hessian) + reg_leaf
983
984         # Prevents overflow and divison by zero
985         if abs(denominator) < 1e-150:
986             tree.value[leaf, 0, 0] = 0.0
987         else:
988             tree.value[leaf, 0, 0] = complexity_penalty + \
989                 (numerator / denominator)
990
991         return None
```

```
992
993     def _utility_obs(self, raw_predictions):
994         """
995         Converts raw prediction of one individual to probabilities summing to 1.
996         Parameters
997         ----------
998         raw_predictions : Pandas DataFrame of one individual with alt_id_col, obs_id_c
999             (estimated utility function) of the shape (n_samples, 3)
1000            the dataframe should be in long format for the discrete choice model
1001        Returns
1002        ----------
1003        utility_sum : Pandas Series of sum of exp(utility) for each obs
1004            the dataframe should be in long format for the discrete choice model
1005        """
1006        raw_predictions_copy = raw_predictions.loc[:, [
1007            self.obs_id_col, 'utility']].copy()
1008        raw_predictions_copy = self._overflow_guard(raw_predictions_copy)
1009
1010        obs = raw_predictions_copy.groupby(self.obs_id_col)
1011        utility_sum = obs.transform(lambda x: np.sum(np.exp(x)))
1012
1013        return utility_sum.values
1014
1015    def _overflow_guard(self, raw_predictions):
1016        """
1017        Bound the utility function to avoid exp overflowing
1018        Parameters
1019        ----------
1020        raw_predictions : Pandas DataFrame of one individual with alt_id_col, obs_id_c
1021            (estimated utility function) of the shape (n_samples, 3)
1022            the dataframe should be in long format for the discrete choice model
1023        Returns
1024        ----------
1025        raw_predictions : Pandas DataFrame of one individual with alt_id_col, obs_id_c
1026            (estimated utility function) of the shape (n_samples, 3)
1027            the dataframe should be in long format for the discrete choice model
1028        """
1029        # Define the boundary values which are not to be exceeded during
1030        max_exp_val = 700
1031        min_exp_val = -700
1032        # The following guards against numeric under / over flow in the utility
1033        # function
1034        too_large = raw_predictions.loc[:, 'utility'] > max_exp_val
1035        too_small = raw_predictions.loc[:, 'utility'] < min_exp_val
1036        raw_predictions.loc[too_large, 'utility'] = max_exp_val
1037        raw_predictions.loc[too_small, 'utility'] = min_exp_val
1038
1039        return raw_predictions
1040
1041    def _raw_prediction_to_proba(self, raw_predictions):
1042        """
1043        Converts raw prediction of one individual to probabilities
1044        summing to 1.
1045        Parameters
1046        ----------
1047        raw_predictions : Pandas DataFrame of one individual with alt_id_col, obs_id_c
1048            and utility (estimated utility function) of the shape (n_samples, 3)
1049            the dataframe should be in long format for the discrete choice model
1050        Returns
1051        ----------
1052        probi_predictions : Pandas DataFrame of one individual with alt_id_col, obs_id
1053            and probability estimates (proba).
1054            the dataframe should be in long format for the discrete choice model
1055        """
1056        raw_predictions_copy = raw_predictions.loc[:, [
1057            self.alt_id_col, self.obs_id_col, 'utility']].copy()
1058        raw_predictions_copy = self._overflow_guard(raw_predictions_copy)
1059
```

```
        obs = raw_predictions_copy.groupby(self.obs_id_col)
        demom = obs.transform(lambda x: np.sum(np.exp(x)))
        num = obs.transform(np.exp)
        raw_predictions_copy['proba'] = num / demom

        return raw_predictions_copy.loc[:, [self.alt_id_col, self.obs_id_col, 'proba']

    def _raw_prediction_to_decision(self, raw_predictions):
        proba = self._raw_prediction_to_proba(raw_predictions)

        proba.loc[:,'decision'] = 0
        proba.loc[proba.loc[:,'decision'].idxmax(),'decision'] = 1

        return proba.loc[:, [self.alt_id_col, self.obs_id_col, 'decision']]


class VerboseReporter():
    def __init__(self):
        self.start_time = time.time()
        self.start_next = time.time()

    def start(self):
        self.start_next = time.time()
        print('Starting Training')

    def update(self, iteration, estimator):

        train_score = round(estimator.train_score_[iteration], 2)

        elapsed = time.time() - self.start_time
        remaining = (estimator.n_estimators - iteration) * \
            elapsed / (iteration + 1)
        delta = time.time() - self.start_next

        elapsed = self._format_time(elapsed)
        remaining = self._format_time(remaining)
        delta = self._format_time(delta)

        print(f"{iteration} : train_score : {train_score}     elapsed : {elapsed}"
              f"     delta : {delta}     remaining : {remaining}")
        self.start_next = time.time()

    def _format_time(self, x):
        # Time Formatting
        if x < 60:
            return '{0:.2f}s'.format(x)
        elif x < 3600:
            return '{0:}m'.format(int(x / 60.0)) + ' {0:}s'.format(int(x % 60))
        else:
            return '{0:}h'.format(int(x / 3600.0)) + ' {0:}m'.format(int(x % 60.0)) +

    def complete(self):
        elapsed = time.time() - self.start_time
        elapsed = self._format_time(elapsed)
        print(f"The training is completed in {elapsed}.")
```

Appendix III : 399 Independent Features Used

II.1 Market Related Variables

4 variables related to the Current Track Odds

4 variables related to the Previous Track Odds

II.2 Fundamental Variable Categories

3 variable related to the General Information of the Horse

17 variables related to the Recovery status of the Horse

8 variables related to the Class level of the Horse

8 variables related to the Bodyweight of the Horse

15 variables related to the Weight carried by the Horse

12 variables related to the Experience of the Horse

56 variables related to the Finishing History of the Horse

8 variables related to the Beaten Lengths of the Horse

49 variables related to the Speed Figures of the Horse

64 variables related to the Pace Figures of the Horse

10 variables related to the Price Money of the Horse

30 variables related to the past performance of the Jockey

14 variables related to the Jockey Horse Stable combination

29 variables related to the past performances of the Stable

14 variables related to the Distance of today's race

9 variables related to the Going of today's race

10 variables related to the Surface of today's race

4 variables related to the Location of today's race

5 variables related to the Profile of today's race

12 variables related to the Post Position bias of Horses

14 variables related to the composite statistics of the Horse

Appendix IV : Hyper-parameter Search Space

Model Candidate	No. Hyper-parameters	Hyper-parameters
RACBoost	14	reg_lambda_l1, reg_lambda_l2, nn_complexity, reg_leaf, ignore_features, nn_count, nn_shrink, nn_size, max_depth, learning_rate, n_estimators, post_lambda, row_subsample, col_subsample
CL_Ridge	1	ridge_penalty
CL_CL_Ridge	1	ridge_penalty
SVR_CL	3	C, gamma, ridge_penalty
SVC_CL	3	C, gamma, ridge_penalty
CL_Frailty	1	ridge_penalty
XGBoost	11	max_depth, gamma, n_estimators, learning_rate, subsample, colsample_bytree, colsample_bylevel, reg_lambda, reg_alpha, min_child_weight, scale_pos_weight
CatBoost	7	iterations, depth, learning_rate, random_strength, bagging_temperature, l2_leaf_reg, scale_pos_weight
XGBoost_CL	12	max_depth, gamma, n_estimators, learning_rate, subsample, colsample_bytree, colsample_bylevel, reg_lambda, reg_alpha, min_child_weight, scale_pos_weight, ridge_penalty
CatBoost_CL	8	iterations, depth, learning_rate, random_strength, bagging_temperature, l2_leaf_reg, scale_pos_weight, ridge_penalty