MASTERS THESIS

APPLICATION OF GAME THEORY IN ENSEMBLE LEARNING

December 5, 2017

Sethu Iyer

Abstract

Ensemble learning is a ML paradigm where a set of classifiers are combined to make the prediction instead of one. Ensembles can perform better than any single classifier, so studying the interaction between entities of ensembles can yield us better predictions than the one given by the two most influential ensemble algorithms which are *Bagging* and *Boosting*. Game theoretic concepts are applied whenever the actions of several agents are interdependent, so game theory is readily applied in ensemble learning.

This thesis reviews some of the game theoretic methods used in ensemble learning which include *weighted majority voting with local accuracy estimates*, *ensemble pruning through Simple Coalition games* and *Banzhaf random forests*. A new tree algorithm is also proposed which takes idea from the mentioned papers. The aim of this thesis is to do a comparative study of Banzhaf Random Forest vs proposed new tree algorithm.

By considering various datasets and applying these algorithms on them, we can identify the cases in which game theoretic approach to ensemble learning would be useful.

Contents

List of Figures

List of Tables

Chapter 1

Basics of Ensemble Learning

The objective of this chapter is to explain the basics of Ensemble Learning. Chapter begins with introducing ensemble learning followed by introduction of two most influential ensemble techniques - *Bagging* and *Boosting*. Throughout the whole chapter, only two class datasets are used.

1.1 Introduction

Ensemble learning is a ML paradigm where we consider the opinions of multiple classifiers and make a decision. Formally, it constructs a set of hypothesis and combines them to use instead of trying to learn one hypothesis from the training data. Each learner in the ensemble are called *base learners*. Ensemble learning is used because it is able to convert *weak learners* to *strong learners*.

Base learners are trained using a *base learning algorithm* which can be decision tree, neural network etc. Same base learners can be grouped to make *homogeneous* ensembles or different base learners can be grouped to make *heterogeneous* ensembles.

Base learners solve the same problem but the learners can be different from each other due to the difference in Population, Hypothesis, Modeling technique and initial seed.

The easiest ensemble methods are *majority voting* in case of classification and *averaging* in case of regression. In majority voting, each model makes a prediction and final output prediction is the one that receives more than half of the votes. If none of the classes have more than half of the votes, then the prediction is unstable [21].

1.2 Effectiveness of Ensemble Classifiers

In order to ensure that ensemble classifiers are more precise than any of its members, the classifiers should be precise and varied [4]. Diverse classifiers make dissimilar errors on new data points. If they are identical, then the errors made by them would be correlated, which defeats the purpose of ensemble as then it would be as good as a single classifier. Uncorrelated errors would mean that when the first classifier is wrong, second and third could be right and the majority vote would mostly correctly classify this new data.

Suppose we have a group of N classifiers, each having error rates $p < \frac{1}{2}$ and the errors are dissimilar. The error distribution of the ensemble follows a binomial distribution and the probability of majority vote being wrong is when the number of classifiers making wrong predictions is greater than $\frac{N}{2}$.

As the probability of the error of each classifier is less than $\frac{1}{2}$, we can intuitively see that probability of the error of the ensemble would be very low. To verify this, consider the plot of the error distribution of an ensemble of 21 classifiers, each having error rate of 0.3 (Shown in *figure 1.1*). The probability of majority vote taking wrong decision 0.026 which is very less than the error rate of individual classifiers.

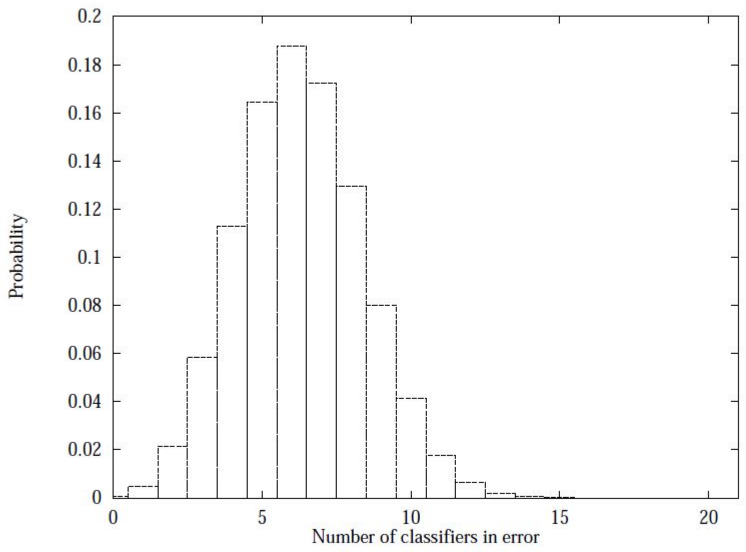


FIGURE 1.1: Error distribution of an ensemble having 21 classifiers, each having error rate of 0.3

From this analysis, we can infer that to successfully construct ensemble methods, construction of individual classifiers with low error rate is an essential step. Furthermore, the errors should be uncorrelated or the classifiers should be diverse.

In practice, the construction of very good ensembles is often possible because of three fundamental reasons which are statistical, computational and representational in nature.

- *Statistical Reason* : In the case where the training data available is restricted, single classifier selects different hypothesis in the hypothesis space which give the same accuracy on the same data. Therefore, the risk of selecting wrong classifier is high. Ensemble averages the votes and reduces the risk, hence.
- *Computational Reason*: Even in cases where there is ample amount of data, the computational process of finding the best hypothesis would still be very difficult. For example, optimal training of neural network is NP-hard [2]. Techniques like gradient descent used in optimizing the cost function performs local search and gets stuck in local optima. If we perform search with varied initial seed, it can provide better approximation of the true unknown function.
- *Representational Reason*: No matter the amount of training data, it is still finite. With finite training samples, the learning algorithms discover only finite set of hypotheses and there is every chance that it would not cover the true hypothesis f. By creating weighted sums of the hypothesis in the considered finite hypothesis space, it may be possible to expand the hypothesis space.

Hence, it is often possible to construct a good ensembles.

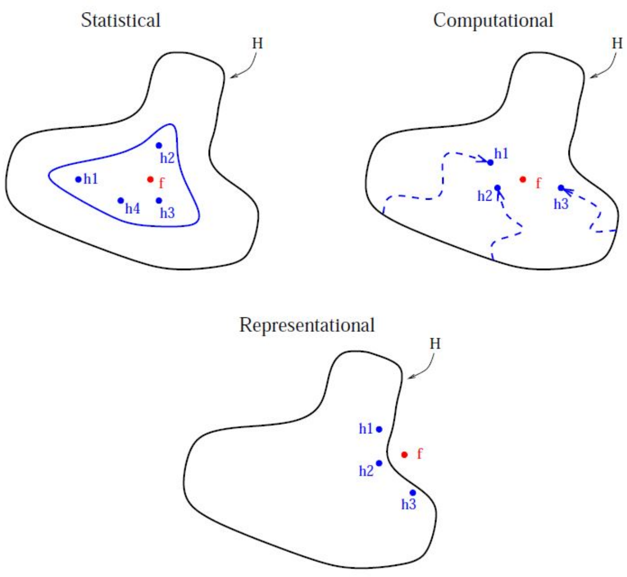


FIGURE 1.2: Reasons why ensemble may work better than a single classifier

1.3 Errors in Ensemble Classifiers

Understanding errors in the model is a crucial step in improving the ensemble accuracy. Mathematically, the error created by any model can be split down into three components, namely:

$$Err(x) = (E[\hat{f}(x)] - f(x))^2 + E[\hat{f}(x) - E[\hat{f}(x)]]^2 + \sigma_e^2 \tag{1.1}$$

which implies Error is summation of Bias squared, Variance and Irreducible Error.

Bias error is useful to measure the average of the difference between predicted values and actual value. High bias error is an indicative of an under-performing model.

Variance measures the dissimilarity of predictions made on same observation. A high variance model will over-fit the training population and performs badly on any observations from the test dataset.

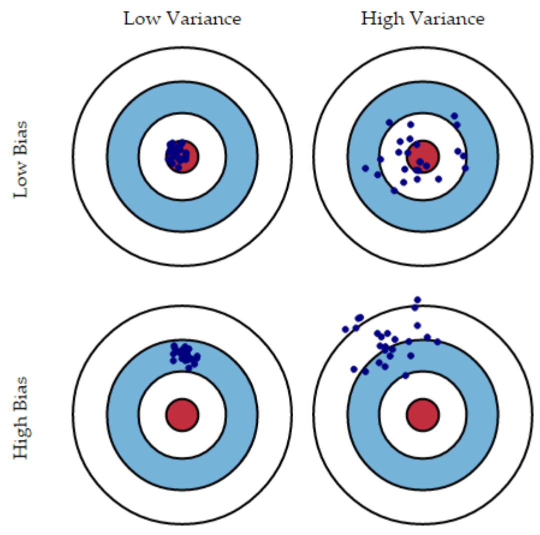


FIGURE 1.3: Visualizing the concepts of Bias and Variance.
Credits: Scott Fortman

High bias is indicative of low model complexity and high variance is indicative of high model complexity so one should find the optimum model complexity where both of them are minimized.

For the task of binary classification, Logistic Regression can be performed and bias and variance still hold significance.

1.4 Methods for Constructing Ensembles

There exists many methods for constructing ensembles in the literature. Here, we will review only those algorithms which will be relevant to our discussion ahead which include *Bagging* [3] and *Boosting* [16].

1.4.1 Bagging

To get a good ensemble, the base learners should be precise and diverse. In case of *Bagging*, the diversity is brought by subsampling the training data. *Bagging* trains the base learners each from a varied *boostrap* sample. A bootstrap sample is a subsample of training dataset with replacement, where the size of the sample is same as the size of the training dataset..

Mathematically, We have an original sample $x_1, x_2, \ldots, x_n$ with n items in it. We draw items with replacement from this set until we have another set of size n. As we are sub sampling, a natural question arises. What is the probability of an item *not* being chosen?

Probability of choosing any one item on the first draw is $\frac{1}{n}$. Therefore, the probability of **not** choosing that item is $1 - \frac{1}{n}$. Repeating this process for the total of n draws which are independent, the probability of never choosing this item on any of the draws is $(1 - \frac{1}{n})^n$. When n gets larger and larger, the value of this probability approaches $\frac{1}{e}$ which is 0.368. This is the probability of an item not being chosen. So, the probability of an item being chosen is $1 - 0.368 = 0.632$.

This can also be demonstrated quite easily via numerical simulation in Python.

```
N <- 1e7 # number of instances and sample size
bootstrap <- sample(c(1:N), N, replace = TRUE)
round((length(unique(bootstrap))) / N, 3)
```

After training the individual classifiers on bootstrapped samples, different voting techniques can be used to predict the final class.

Algorithm 1 shows the pseudo code of the bagging algorithm. Appendix A has the implementation of the algorithm in Python.

Algorithm 1: The Bagging algorithm

```
Input  : Data set $D = (x_1, y_1), (x_2, y_2), (x_3, y_3) \dots (x_n, y_n)$
         Base Learning Algorithm $L$
         Number of learning rounds $T$
1 for $t = 1 \dots, T$: do
2     $D_t = Bootstrap(D)$ ;                    // Generate a bootstrap sample from D
3     $h_t = L(D_t)$ ;                          // Train a base learner $h_t$ from the bootstrapped sample
4 end
Output : $H(x) = argmax_{y \in Y} \sum_{t=1}^{T} 1(y = h_t(x))$ ;    // 1(a) here is the indicator function
```

1.4.2 Boosting

Boosting is an another ensemble method like bagging. There are many boosting algorithms but in this section, we would discuss the most famous algorithm, **AdaBoost**.

The algorithm takes as input a training set $(x_1, y_1), \dots, (x_n, y_n)$ where each x_i belongs to some *domain* or *space* X and each label y_i is in some label set Y.

Adaboost calls a given *base learner* repeatedly in a series of rounds $t = 1, \dots, T$. The algorithm maintains a distribution of weights over the training dataset. Individual weights distribution depends on the training example i as well as the training round t and hence, weight is denoted by $D_i(t)$ for each training example i.

Initially, all weights are set equally, but on each round, the weights of incorrectly classified examples are increased so that weak learner is forced to focus on the hard examples of the training set.

Mathematically, the base learner's job is to find a *weak hypothesis* $h_t : X \to \{-1, +1\}$ appropriate for the distribution D_t. The performance of the hypothesis is measured by it's *error*

$$\epsilon_y = \sum_{i:h_t(x_i) \neq y_i} D_t(i) \tag{1.2}$$

This error is used to determine the weights of the training sample. Incorrectly classified example's weights are increased exponentially and correctly classified example's weights are decreased exponentially. Hence, the hard to classify examples have large weight associated with them, and usually they are the outliers of the training dataset. Hence AdaBoost algorithm can is deployed to identify outliers of a training dataset.

Algorithm 2 shows the pseudo code of the boosting algorithm. Appendix A has the implementation of the algorithm in Python.

Algorithm 2: The Boosting algorithm

```
Input : Data set $D = (x_1, y_1), (x_2, y_2), (x_3, y_3) \dots (x_n, y_n)$
        Base Learning Algorithm $L$
        Number of learning rounds $T$
1 $D_1(i) = \frac{1}{m}$
2 for $t = 1 \dots, T$: do
3     $h_t = L(D, D_t)$ ;                    // Train a base learner $h_t$ using distribution $D_t$
4     $\epsilon_t = \sum_{i:h_t(x_i) \neq y_i} D_t(i)$ ;                    // Measure the error
5     $\alpha_t = \frac{1}{2} ln(\frac{1-\epsilon_t}{\epsilon_t})$ ;                    // Determine the weight
6     $D_{t+1}(i) = \frac{D_t(i) exp(-\alpha_t y_i h_t(x_i))}{Z_t}$ ;                    // Update and Normalize
7 end
Output: $H(x) = sign(f(x)) = sign(\sum_{t=1}^{T} \alpha_t h_t(x))$
```

It is a well known fact that boosting is resilient to overfitting. Theoretical arguments of this phenomenon are still incomplete.

1.5 Selective Ensembles

According to the proof done in section 1.2, it seems as if we increase the number of classifiers, the error would decrease. But "many could be better than all" [20] theorem was proved which says that this is far from the truth. The paper tells us that a particular subset of classifiers always perform better. Such subset of ensembles are called *selective ensembles.*

There are data subset methods as well which are used in selective ensemble learning. They are

- Hill-Climbing (HC) Method
- Ensemble Backward Sequential Selection Method
- Ensemble Forward Sequential Selection Method
- Clustering Selection Ensemble

The first three algorithms use genetic algorithmic approach while the last uses clustering approach.

Recently, game theoretic approaches are also being used to study the interactions between the weak learners to select the best subset which maximizes the performance of the ensemble. In the upcoming chapters, we will discuss three of such methods.

1.6 Summary

Ensemble learning is a machine learning paradigm where multiple learners are trained to solve the same problem. It is appealing because it is able to boost weak learners to strong learners. In order to have a good ensemble, base learners should be strong and they should have *diversity*. The most influential ensemble methods are *Bagging* and *Boosting* which gives us general and effective methods to produce ensembles. Sometimes, it is better to consider a subset of ensemble rather than considering all. The subset is generally chosen using genetic and clustering algorithms. Game theoretic approach is also used in studying ensembles recently to select the best subset.

Chapter 2

Basics of Game Theory

The objective of this chapter is to explain the basics of Game Theory. Chapter begins with introducing the history of game theory, followed by *non cooperative games* & *co operative games*. We would then move on to voting theory, discussing power indexes and so on. In the end, we would discuss the potential application of concepts discussed in ensemble learning theory.

2.1 Introduction

When we hang out with the friends, we seldom think about the math behind the decisions we are making. But when we do think, we are using game theory.

Mathematician John Nash pioneered Game Theory in 1950s. Game theory is the science of decisions made by people when they interact. It is also applied to any situation where group of entities interact. Game theory analyzes the decision made by an entity of the group.

Game theory demands the individuals to be rational. An individual is rational if the individual always takes best possible action in accordance to his/her goal. For our discussion ahead, this assumption holds ground.

Because of it's range of application, Wide range of audience study Game Theory.

Game theory has two main branches. They are Cooperative Game Theory, Non-Cooperative/Competitive Game Theory . Non cooperative game theory covers competitive interactions. Cooperative game theory covers cooperative interactions. Ensemble learning theory uses concepts of cooperative game theory.

Let us study them one by one by considering examples.

2.2 Competitive Game Theory

Prisoners Dilemma is the most famous thought experiment in competitive game theory. It deals with a game consisting of two prisoners. The district attorney offers them the following deal. If both of them confess, They have five years of prison time. If one of them confess and another doesn't, the confessor is set free and the another has six years of prison time. If neither of them confesses, both get to serve one year.

This[9] shows why two rational individuals might not cooperate even if cooperation is in their best interest, thus resulting in a sub-optimal outcome..

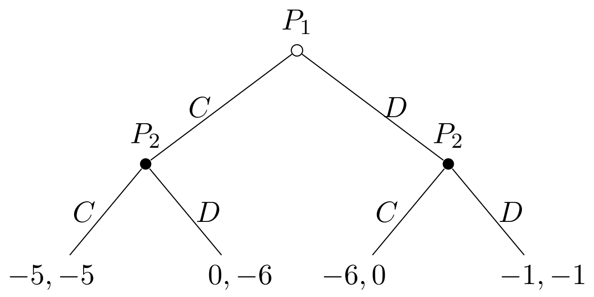
The below table shows the 2×2 matrix of the game. Here, C refers to the action of cooperation and D denotes the action of non cooperation.

		Player 2	
		C	D
Player 1	C	$-5, -5$	$0, -6$
	D	$-6, 0$	$-1, -1$

Table 2.1: 2x2 Matrix: Prisoner's Dilemma

As Player 1, no matter what Player 2 intends to do, Playing C yields a better outcome for Player 1. But it is not rational thing to do so because there is every chance of other player confessing the crime. Best outcome for Player 1 is the case in which he confesses and the player 2 does not. The optimal outcome of a game is one where no player deviates from the chosen strategy even after considering an opponent's choice. The action of (C, C) is called the *Nash Equlibrium*[5].

Figure 2.1: Extensive form of Prisoner's Dilemma



The set of actions of each player in the game is called *action profile.* In the prisoner's dilemma, we developed a reasoning and predicted how a game will be played. Predicting how a game will be played using formal rules is called **solution concept** in game theory. In this game, the actual optimum of (D, D) is actually, very unstable. Both people pick something that is not optimal globally.

2.3 Cooperative Game Theory

In cooperative games, Every player has agreed to work together towards a common goal. In cooperative game theory, the main question is 'How much should each player should contribute to the coalition and how much should they benefit from it'. It tries to determine the *fairness* of the game.

Like how competitive game theory has Nash equilibrium which is the stable solution concept of a competitive game, cooperative games have a concept called *Shapley Value*[8]. The Shaply value is a method of dividing up gains or costs among players according to the value of their individual contributions.

The Shapley value works by applying several axioms.

1. The contribution of each player is equal to the difference between total output before and after removing them from the game. This is their *marginal contribution*

2. Interchangeable players have equal value

3. Dummy players have zero value.

4. Splitting of payments occur in the case of multi-part games.

The first point can be understand by taking this example. Let's assume a group of people are making cookies. It is observed that when one person fell sick, the group produced 50 fewer cookies. So, the marginal contribution of that person to the coalition is 50 cookies.

Second and third points are self explanatory.

The last point stresses the fact that it is not always fair to use the same solution every time. The numbers should be reviewed regularly so that coalition makes adjustments.

If we find a way to divide up costs or payment to all of those players satisfying each of these axioms, that is the Shapley value. Shapley value captures the average marginal contribution of a player which is calculated in the following way:

Suppose in one hour, you can bake 10 cookies and your friend can bake up 20 cookies. When you two work together, suppose you make 40 cookies each hour which is 10 cookies extra when you two work alone. Suppose you sell those cookies for 1$ and you have earned $40.

In this problem, your friend's marginal contribution to you is $40 - 10 = 30$ cookies. Your marginal contribution is $40 - 20 = 20$ cookies. You can make 10 cookies an hour and your friend can make 20 cookies an hour. According to the Shapley value equation, you should average these two numbers to get your contribution to the coalition which is $\frac{10+20}{2} = 15$ and your friend's contribution is $\frac{20+30}{2} = 25$.

Cooperative and Non Cooperative game theory are used in reinforcement learning as well as deep learning. The field of reinforcement learning has lot of overlap with game theory and it is being actively used in research recently. Ensemble learning uses the concept of voting games to make better predictions, however.

2.4 Voting Games

In 1st Chapter, when we were introduced to ensembles, we discussed the concept of *majority voting* where if more than half of the votes favor one class of output, then that class is predicted. Of course, that is not the only one way to predict the final output and the way we consider the individual votes to deduce final outputs affect the ensemble accuracy a lot. Hence, in this section, we would discuss Voting Games.

The name is inspired from the phenomenon of elections because elections lend themselves to game theory. The question of 'How should people vote' is often tricky, hence there has been lot of studies conducted on voting games.

One of the straight forward method to choose winner of a voting game is *plurality voting.* If someone in an election receives *plurality* of votes, they have received more than any other candidate. This is not suitable when three or more candidates are contesting. If there are two candidates, then plurality of votes should be used because plurality voting automatically becomes majority voting.

Often in ensemble learning, we not only get the most likely class from each base learners, but we also get the probability distribution of the prediction from which we can rank the alternatives. In voting theory, this set of ranked preferences is called **preference ballot**.

Example Suppose we have 10 base learners trying to predict one of the three categories whose labels are H, O, A. The following table shows their votes.

	C_0	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9
First Choice	A	A	O	H	A	O	H	O	H	A
Second Choice	O	H	H	A	H	H	A	H	A	H
Third Choice	H	O	A	O	O	A	O	A	O	O

These individual votes are combined into one **preference schedule**, which shows the number of voters in top row and the corresponding ranked preference.

	1	3	3	3
First Choice	A	A	O	H
Second Choice	O	H	H	A
Third Choice	H	O	A	O

For the plurality method, we care only about the first choice options. If we total them up, we can easily see that A is the clear winner. But is it fair?

In the above preference schedule, let's just focus on H vs A. In the first two columns, A is the winner as it has higher preference than H. In the last two columns, H is the winner for the same reasons. It is clear that 6 out of 10 classifiers prefer the category H.

This doesn't seem fair and Condorcet noticed it and devised **Condorcet Criterion**[9] which says that If there is a choice that is preferred in every one-to-one comparison with other choices, then that choice should be the winner. We call this winner the Condorcet Winner.

In the above example, we can easily verify that H is preferred over O and hence, H is the Condorcet Winner.

Condorcet winner can also be unfair. A method called **Borda Count** [9] is majorly used worldwide in determining the winner. It describes a consensus-based voting system since it can sometimes choose a more broadly acceptable option over one majority support. It considers every voter's entire ranking to determine the outcome. Chapter 4 has the implementation of borda count method and it's performance is measured over various datasets in Chapter 5.

2.5 Weighted Voting Games

Weighted voting[14] is an electoral system in which not all voters have the same amount of influence over the outcome of the election. Different votes are given different weights. This is used in ensemble learning where some classifiers are given more preference than others.

Each individual or entity casting a vote is called a **player** in the election. They're often notated as $P_1, P_2, P_3, \ldots, P_N$ where N is the total number of voters. Each player is given a **weight** which usually represents how many votes they get. The **quota** is the minimum weight needed for the vote to pass or weight needed for the proposal to be approved.

In order to have a meaningful voting system, the quota must be more than $\frac{1}{2}$ the total number of votes and the quota can't be larger than the total number of votes.

A coalition is any group of players voting the same way. A coalition is **winning coalition** if the coalition has enough weight to meet the quota. A player is **critical** in a coalition if by leaving the coalition it would change it from a winning coalition to a losing coalition. Using this, we define the following three concepts:

- A player is a **dictator** if the single-player coalition containing them is a winning coalition.
- A player has **veto power** if they are critical in every winning coalition
- A player is a **dummy** if they are not critical in any winning coalition.

While categorizing the players according to their importance is one approach in determining the importance of any player, one can numerically represent the ability of a player to influence the outcome of the voting. The ability of a player to influence the outcome is defined as **power** of the player. In order to look at power in a weighted voting situation numerically, voting power index is used.

One of the earliest power index introduced is **Banzhaf power index**[14]. In a situation where voting rights are not necessarily equally divided, Banzhaf power index measures the probability of changing the outcome of vote. It does so by enlisting all the winning coalitions and counting the number of times a particular player is critical in the coalition.

To calculate the Banzhaf Power Index, these steps are to be followed.

1. List all winning coalitions.
2. In each coalition, identify the players who are critical
3. Count how many times each player is critical
4. Convert these counts to fractions or decimals by dividing by the total number of times any player is critical.

Example: Consider the voting system [**12:8,5,3,1**].

First, we have to list all the winning coalitions. First, we would look at one player winning coalitions. Does anyone have enough weight to pass the vote by themselves. Nobody has 12 or more weights, so there are no single player winning coalitions.

Now, let's focus on two player coalitions. The first two player coalition would be $\{P_1, P_2\}$ and their combined weight is 13, which is greater than quota. It's easy to check that P_1 doesn't form a two player winning coalitions with either $\{P_3\}$ or $\{P_4\}$ and there aren't any more two player coalition possible.

Focusing on three player coalitions, we find $\{P_1, P_2, P_3\}$, $\{P_1, P_2, P_4\}$ and $\{P_1, P_3, P_4\}$ form a coalition. Four player coalition is just all players which does exceed the quota. The following table summarizes the winning coalitions and the critical players.

2 Player	C.P.	3 Player	C.P.	4 Player	C.P.
$\{P_1, P_2\}$	P_1, P_2	$\{P_1, P_2, P_3\}$	P_1, P_2	$\{P_1, P_2, P_3, P_4\}$	P_1
		$\{P_1, P_2, P_4\}$	P_1, P_2		
		$\{P_1, P_2, P_3\}$	P_1, P_3, P_4		

Let us suppose for a player i, let B_i denote the number of times a player is critical. Then the counts are

- $B_1 = 5$
- $B_2 = 3$
- $B_3 = 1$
- $B_4 = 1$

Total number of players $= B_1 + B_2 + B_3 + B_4 = 10$.

Banzhaf power index for player i is therefore, $B_i/10$ which are 0.5,0.3,0.1,0.1 respectively for players 1,2,3,4.

It is observed that the number of coalitions one can form when there are N players is 2^N. As N increases, 2^N increases very rapidly. Problems with such growth are intractable and generally, they belong to a category called *NP-Complete*. It is very similar to 0-1 Knapsack problem, except the restriction here becomes that sum should be greater than the quota of game. Calculation of Banzhaf power index is shown to be NP-Hard[12]. However, using generating functions, Banzhaf power index can be calculated in $O(n2^{\frac{n}{2}})$[13] time complexity. There are also dynamic programming and Monte Carlo methods to calculate the power index[1].

Banzhaf power index can be used in case of random forests where instead of choosing the features based on information gain rate, we chose them according to their importance in the dependency among group of features. This type of random forests which uses Banzhaf power index for their prediction are called as **Banzhaf Random Forests**[18]. The theory of Banzhaf random forest is covered in Chapter 3.

2.6 Summary

Game theory is the science of making decisions. A *game* in Game theory refers to any interaction between multiple people in which each person's payoff is affected by the decision made by others. The interactions can be cooperative or non cooperative. In case of non cooperative game theory, we study the strategies of various people who are interacting and in case of cooperative game theory, we study the contribution of an individual in the coalition. Game theory tells how to be smart in non cooperative situations and how to be fair in cooperative situations.

One of main subtopic of game theory which is used in ensemble learning is voting theory. In ensemble learning, we can give equal weights to every classifier and that is dealt with non weighted voting games. If they are having unequal weights, we need a metric which measures the influence of a particular classifier. That is given by Banzhaf Power index which tells the probability of a classifier influencing the decision of the ensemble.

Calculating Banzhaf power index is NP-Hard because it enlists all the winning coalitions which forces us to check all the subsets of a set which increases exponentially to the size of the set. However, there do exists approximation algorithms, mainly monte carlo methods.

Chapter 3

Game Theoretic Ensemble Methods

In this chapter, We would cover the theory of three ensemble methods which uses game theoretic methods. We would start off with treating the ensemble as coalition games, specifically weighted voting games, then other methods are discussed. The final section introduces a slight improvement to the existing Banzhaf decision tree algorithm. Implementation of these techniques can be found in Appendix A.

3.1 Introduction

Ensemble learning is one of the most active research area because of it's effectiveness. Multitude of algorithms exist in literature designing the optimal ensemble system but most of them employ a heuristic based approach and rarely an analytic approach. A game theoretic approach is useful as a n-classifier ensemble can be considered as n-person game. Since all the classifiers in the ensemble work towards a common goal, *coalition games* becomes natural way to formulate the ensemble design problem.

The first section treats the ensemble design problem as weighted voting games (WVG) where the base classifier can be trained independently and the algorithm calculates voting weight of the classifiers. The calculation of voting weights involves prior estimation of the success rate of the classifiers, which is also called as *local accuracy estimates.*

The second section calculates the diversity of the classifiers based on Banzhaf power index and prune those classifiers to obtain a minimal winning coalition which has moderate diversity.

The third section uses random forests and cooperative game theory to evaluate power of each feature, particularly it uses banzhaf power index to calculate the power and the most important feature among the group of features.

3.2 Weighted Voting Games with Local Accuracy Estimates

For a little recap, a weighted voting game (WVG) is a n player game where there are k available choices to vote and a weight is assigned to each voter. We discussed the properties of WVG in Chapter 2, however we didn't discuss about the optimal decision rules for the WMG so that collective performance is maximized. It has been proven by Shapley and Grofman (1984) [17] that optimal rules for WMG is simply weighted majority rule. The result was later extended to multi class classification problems.

Also, if the classifiers are independent, one can arrive at a closed form solution for the voting weights which is given as

$$w_i = \log(\frac{p_i}{1-p_i}), p_i = P_i\{\theta = \omega_{correct}\}, i = 1 \dots n \tag{3.1}$$

where w_i is the weight of the i-th player, p_i is the estimated probability for correct classification measured in the validation set θ and $\omega_{correct}$ is the correct class label for the corresponding input [17].

Combining this with the weighted majority rule, the optimal voting scheme can be derived as weighted sum of individual player decisions multiplied by their weights. Although this may seen optimal, it doesn't take into account the dependencies among the classifiers of ensemble.

Instead of fixing the weight globally for each point, we can change the weight at each of the classifier points to reflect their competencies, that is:

$$w_i = \log(\frac{p_i}{1-p_i}), p_i = P_i\{\theta = \omega_{correct}|\mathbf{x}\}, i = 1 \dots n \tag{3.2}$$

This introduces a conditional competencies of the classifiers at each point. As weights change, this model is called as adaptive WMR. Harris V. Georgiou and Michael E. Mavroforakis [7] demonstrate the superiority of the adaptive WMR model over simple majority voting, simple averaging rule and Bayesian based combination rules.

Local accuracy can be calculated by calculating the percentage of training samples in that region that are correctly classified. It is kind of similar to K-nearest neighbors algorithm. Other methods used are dynamic width histogram [7] where we estimate $P_i\{\theta = \omega_{correct}|\mathbf{x}\}$ by estimating it's complement.

Like weighted voting games, if the total quota reaches a threshold, we consider the ensemble decision as one class, else the other class. Usually, the mid point of the range of the weights is considered as threshold. Chapter 4 has the implementation of local accuracy estimate using the K-nearest neighbor method.

3.3 Ensemble Pruning through Simple Coalition Games

In the first chapter, we touched upon the fact that increasing the number of members of the ensemble doesn't necessarily increase the ensemble accuracy. Very often, a subset of those ensembles offer higher performance than the whole ensemble. In this section, we shall look into the problem of ensemble pruning, which aims to extract sub ensembles which offers high performance.

In order to find a solution to this problem, we take the game theoretic approach by formulating the problem of ensemble pruning as non-monotone simple coalition game played among ensemble members. This game has two thresholds, and the sum of the weights of the members should lie in between those thresholds.

In order to calculate the weight, We use *diversity contribution* of each member and calculate the weight of the ensemble. Then we greedily use Banzhaf power index, adding ensembles in the set until the total diversity of the ensemble is in acceptable range.

We redefine Banzhaf power index using two concepts: *positive* and *negative swings*. This will also helps us to understand why we greedily add those members whose Banzhaf power index is high.

A coalition S is a *positive swing* for player i if $S \cup \{i\}$ wins and S loses. Conversely, a coalition S is a *negative swing* for player i if $S \cup \{i\}$ loses and S wins. Let $swing^{+}$ and $swing^{-}$ denote the set of positive and negative swing coalitions for player i. Then, the Banzhaf power index of the player i can be given by:

$$B_{Z_i}(G) = \frac{1}{2^{n-1}} \times (|swing_i^{+}| - |swing_i^{-}|) \tag{3.3}$$

Here, a classifier is *pivotal* if the classifier induces a proper amount of diversity and turns a losing coalition into a winning coalition. Hence, Banzhaf power index assigns high ranks to those classifiers which are pivotal in the decision making while maintaining moderate diversity.

We seek to obtain a set of classifiers with average diversity because several experimental studies have shown that *large diversity* leads to sharp drop in performance [6] and it is well known fact that ensemble of identical classifiers is ineffective.

There are many diversity measures, the popular one being *Disagreement measure*. Given two classifiers h_i and h_j, the disagreement measure is given by:

$$dis_{i,j} = \frac{N^{01} + N^{10}}{N^{11} + N^{00} + N^{01} + N^{10}} \tag{3.4}$$

where $N^{00}, N^{01}, N^{10}, N^{11}$ denote the number of correct/incorrect predictions made by h_i and $h)j$ on the training set.

Assuming every ensemble member is trained separately using the same training set τ, the problem of ensemble pruning is now selecting an ensemble ω from the initial ensemble Ω that yields the best predictive model. We do that by constructing a weighted voting game whose weights are number of classifiers which has lower diversity than the diversity of h_i. We measure the diversity by considering pairwise interactions between one classifier and the rest and we take average among them. that is:

$$Div_\rho(h_i) = \frac{1}{n-1} \sum_{h_j \in \Omega \setminus \{h_i\}} f(h_i, h_j) \tag{3.5}$$

After calculating diversity, the weight of classifier i (w_i) in this WVG can be calculated as

$$w_i = \sum_{h_j \in \Omega \setminus \{h_i\}} \mathbb{1}(Div_\Omega(h_i) \geq Div_\Omega(h_j)) \tag{3.6}$$

Then, we can consider the weighted voting game G given by $G = (N, [\vec{w}, q_1, q_2])$. The game is considered as win if in the coalition S, $q_1 \leq \sum_{i \in S} w_i \leq q_2$. This is because we want to introduce moderate diversity.

In order to properly define the game, the acceptable threshold should be greater than the maximum weight and their difference should be greater than the maximum weight to avoid dictatorship situations.

Computing this is certainly intractable, but still it can be computed in psuedo polynomial time[19] if we consider another formulation of the same problem. Let $B_{Z_i}(G)$ denote the banzhaf power index vector of a game G. Let us consider two weighted voting games $G_1 = (S, \vec{w}, q_1)$ and $G_2 = (S, \vec{w}, q_2 + 1)$ where q_1 and q_2 are the two thresholds of the weighted voting game. Then

$$B_{Z_i}(G) = B_{Z_I}(G_1) - B_{Z_2}(G_2) \tag{3.7}$$

Hence, we can compute them in Pseudo polynomial time, using this formulation. The power index of both the games can be calculated in parallel so the time complexity to calculate is really low.

Appendix A has the implementation of this algorithm. Below is the pseudo code of the algorithm.

Algorithm 3: Pseudo code of Ensemble pruning using Simple Coalition Games

```
Input      : $\tau$ : Training set
             $\Omega$ : Ensemble of classifiers
             $q_1, q_2$: Two thresholds
Initialize: $\omega = \phi$
;                                                    // Getting classifier's predictions
1  foreach $h_i \in \Omega$ do
2      foreach $(x_i, y_i) \in \tau$ do
3          $Preds^i_j = h_i(x_j)$
4      end
5  end
;                                      // Estimating classifier weights based on preds
6  foreach $h_i \in \Omega$ do
7      $Div_\rho(h_i) = \frac{1}{n-1} \sum_{h_j \in \Omega \setminus \{h_i\}} f(h_i, h_j)$ ;   // f is the disagreement measure
8      $w_i = \sum_{h_j \in \Omega \setminus \{h_i\}} \mathbb{1}(Div_\Omega(h_i) \geq Div_\Omega(h_j))$
9  end
10 $B_{Z_i}(G)$ = computeBanzhafIndices$(w, q_1, q_2)$
;                                               // Searching for minimal winning coalition
11 repeat
12     $h = argmax_{h_i} B_{Z_i}(G)$
13     $\omega = \omega \cup \{h\}$
14     $\Omega = \Omega \setminus \{h\}$
15 until $\omega$ wins;
Output    : $\omega$: Pruned ensemble
```

Refer Appendix A for implementation using disagreement measure as diversity measure.

3.4 Banzhaf Decision tree

Decision tree is a very popular machine learning algorithm. It helps in visualizing the prediction and doesn't act like a black box. In this section, we shall look into an alternative approach to build decision tree using cooperative game theory which uses Banzhaf power index to recursively select features and build tree. The goal of cooperative game theory is to distribute the total gain among each player in a fair way and Banzhaf power index provides us a way to approximate the fairness.

The root node is selected with information gain rate. In decision tree learning, Information gain rate is calculated using **Gini impurity** which is a measure of misclassification. The following steps are used to select the children nodes:

1. Evaluate the impact of a feature in a coalition using conditional mutual information.
2. If half of the features in coalition $\mathbb{S}$ is interdependent with a feature f_i outside coalition, then the coalition $\mathbb{S} \cup f_i$ is a winning coalition.
3. Calculate the Banzhaf Power Index of this Weighted Voting Game
4. Choose the feature with highest Banzhaf Power Index

Conditional mutual information to evaluate the interdependent between single feature f_j not in coalition $\mathbb{S}$ and a feature f_i which is in the coalition $\mathbb{S}$. Mutual Information $I(X, Y)$ measures the degree of dependence between two random variables X and Y. Conditional mutual information $I(X, Y|Z)$ is the expected value of mutual information of $I(X, Y)$ given the value of Z.

In our context, we define the conditional mutual information[18] as

$$I(f_j; f_i|\mathbb{S} \setminus f_i) = \sum_{x \in f_j} \sum_{y \in f_i} \sum_{z \in \mathbb{S} \setminus f_i} \log(\frac{p(x, y|z)}{p(x|z)p(y|z)}) \tag{3.8}$$

Given a feature coalition $\mathbb{S}$ and a feature $j \notin \mathbb{S}$, The game is considered win if the feature j is dependent on atleast half of the members of $\mathbb{S}$. This is used while calculating the Banzhaf power index which is given by

$$B_{Z_i}(G) = \frac{1}{2^{n-1}} \times |swing_i^+| \tag{3.9}$$

where Z_i is the feature considered, G is the cooperative game and n is the number of features present in the dataset.

By choosing the feature with highest Banzhaf power index, we are choosing that feature on which majority of the features are dependent on. However, as usual computing this is intractable and because of the definition of the game, it is not possible to calculate the index without going through all possible coalitions. While this is a limitation, We can combine the algorithm from previous section and possibly create an efficient ensemble of Banzhaf decision trees.

Appendix A has the implementation of Banzhaf Decision tree while the next chapter compares it's performance with standard decision tree.

3.5 Pruned Banzhaf Decision tree

In this section, we would try to improve Banzhaf Decision tree algorithm discussed in section 3.4. Calculation of Banzhaf power index is intractable and hence, we are forced to select a small subset out of available features and then recursively build the tree. However, some important features may be lost in this step while randomly selecting features and it would be great if we can choose the subset without worrying about potential loss of important features.

Feature selection is a machine learning problem which aims to extract the relevant subset of features which would improve the classification algorithm considered. It should be noted that feature selection is not an independent process and is certainly dependent on the learning algorithm considered.

There are many attempts done by researchers in the field of feature selection. We would be using a method which uses dynamic information criterion[11]. This is a greedy feature selection method.

Given a training dataset $D = (X_i, Y_i)_{i=1}^{n}$ with n samples and dimension of each sample being M, our task is to efficiently choose $K \ll M$ features such that performance of our Banzhaf decision tree increases. Let F denote the feature set, C denote the output column and S denote the currently selected feature subset. Below is the pseudo code of the algorithm.

Algorithm 4: Pseudo code of greedy feature selection using Conditional Mutual Information

```
Input     : $T$: Training set, $C$: output
Initialize: $S = \phi$
1 while $F \neq \phi$ do
2     $MI = \phi$ foreach $f$ in $F$ do
3         $MI = MI + I(C; f|S)$
4         remove $f$ from $F$ if $MI_i = 0$ and delete the column from training dataset.
5     end
6     Select the feature with maximal $I(c; f|S)$ and add that to $S$ and remove it from $F$
7     Remove the column associated with feature $f$ from the training dataset $T$.
8 end
Output    : $S$ : Selected feature subset
```

This algorithm can also be stopped when $|S| \geq K$ and then Banzhaf decision tree algorithm can be applied. In the next section, we discuss the algorithm of Banzhaf Random Forest which is an ensemble of decision trees.

3.6 Banzhaf Random Forest

Random forests are an ensemble of decision trees which aims to reduce the high variance susceptibility and improve their prediction by introducing appropriate diversity between them. It takes a similar approach to bagging, where different decision trees are trained on multiple samples of the training dataset. In decision trees, the selection of splitting point is greedy, so some rows are restricted from splitting in order to introduce more diversity. As Banzhaf trees split at midpoint, we use bagging to construct the random forest. Jianyuan Sun et.al [18] discusses the consistency of Banzhaf Random Forests.

The procedure of the algorithm is given below

1. Draw n samples with replacement for each tree in the ensemble (Bootstrap sample)
2. Prune the features using the algorithm discussed in 3.5
3. Build the Banzhaf tree
4. Use LAE (Local accuracy estimates) or Majority voting in case of two class classification and Borda count for multi class classifications.

3.7 Summary

In the case of two class classifications, we have seen local accuracy estimates method (3.2) which takes into account local performance in the validation set of each ensemble member for a test example. It assigns weights according to this local accuracy and more effective decision can be made.

As discussed in the section 1.5, having more members in the ensemble does not always mean better performance. In 3.3 we discussed about ensemble pruning method which induces a weighted voting game using the diversity of the ensemble members. A moderate diversity ensures the success of ensemble.

Decision tree uses information gain to greedily select features and the point to split, which makes it susceptible to high variance data. Hence, we discussed Banzhaf Decision trees which always splits at midpoint and uses conditional mutual information to induce a weighted voting game and chooses the splitting feature accordingly.

We also discussed a feature selection method which also uses conditional mutual information in 3.5 which can reduce the computational cost by choosing relevant features.

In the end, we discussed about Banzhaf random forest and how it could be improved using the techniques discussed earlier.

Chapter 4

Experiments

This chapter focuses on the methodology of the experiments. Datasets fetched from UCI Machine Learning Repository[10]. Discussion of these experimental results is covered in the next chapter.

The experiments are carried using Python 3. Scientific Python libraries such as scikit-learn[15], numpy, pandas are used to process the data while external python module entropy_estimators is used for computing information theoretic results.

4.1 Experiment 1: Ensemble with Borda Count

In this section, We take a multi class classification problem and an ensemble of K-Nearest Neighbors(KNN), Support Vector Classifier (SVC), Logistic Regression(LR) and Decision tree(DT) and generate the preference ballot and use Majority voting and Borda count. Chapter 5 discusses the result of the experiments.

Dataset chosen is Glass Identification dataset which predicts the type of glass given various chemical compositions of metals like sodium, potassium, magnesium etc. Seven types of glasses are identified.

We import the dataset using pandas library.

```python
import pandas as pd
columns=['id','RI','Na','Mg','Al','Si','K','Ca','Ba','Fe','Type']
dataset = pd.read_csv('glass.csv',names=columns)
out_distrib = dataset['Type'].unique()
dataset = dataset.values
X,y=dataset[:,0:-1], dataset[:,-1]
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.89, random_state=42)
```

Now, we construct four base learners as discussed above.

```
from sklearn import preprocessing,neighbors,svm
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn import tree

clf1 = neighbors.KNeighborsClassifier()
clf2 = svm.SVC(probability=True)
clf3 = LogisticRegression()
clf4 = tree.DecisionTreeClassifier()

clf1.fit(X_train, y_train)
clf2.fit(X_train, y_train)
clf3.fit(X_train,y_train)
clf4.fit(X_train,y_train)

accuracy_1 = clf1.score(X_test, y_test)
accuracy_2 = clf2.score(X_test, y_test)
accuracy_3 = clf3.score(X_test,y_test)
accuracy_4 = clf4.score(X_test,y_test)
print(accuracy_1,accuracy_2,accuracy_3,accuracy_4)
```

To implement the Borda count method, we need the preference ballot first. The following function generates it. It takes in predicted probability scores and returns the preference ballot string.

```
import operator
def generate_preference(pred_proba):
        pred_proba = pred_proba[0]
        num_class = pred_proba.shape[0]
        vote_dic = {}
        for i in range(num_class):
                vote_dic[out_distrib[i]] = pred_proba[i]
        sorted_x = sorted(vote_dic.items(), key=operator.itemgetter(1))
        sorted_x.reverse()
        preference = []
        for i in range(num_class):
                preference.append(int(sorted_x[i][0]))
        return list(map(str,preference))
```

And now we implement the borda count method which takes in input a list of list and returns the label.

```
def borda(preference_ballot):
'''
Accepts: list of list => preference_ballot
Returns: Winner
'''
        counts = {}
        candidates = list(set(preference_ballot[0]))
        max_point = len(candidates)
        for i in range(max_point):
                counts[candidates[i]] = 0
        for pref in preference_ballot:
                for i in range(len(pref)):
                        counts[pref[i]] += (max_point -i)
        return int(max(counts, key=counts.get))
```

In the end, we encapsulate everything in a prediction function which takes in the test example, generates preference ballot, uses borda count method and returns the winner.

```
def get_prediction_borda(test_example):
        ensemble = [clf1,clf2,clf3,clf4]
        preference_ballot = []
        for base_learner in ensemble:
                preference_ballot.append(generate_preference(base_learner.predict_proba(test_example)))
        return borda(preference_ballot)
```

We also implement the majority vote classifier, whose performance will be compared with the Borda count method.

```
def get_prediction_majority(test_example):
        ensemble = [clf1,clf2,clf3,clf4]
        predictions = []
        for base_learner in ensemble:
                predictions.append(int(base_learner.predict(test_example)[0]))
        occ = Counter(predictions)
        return max(occ,key=occ.get)
```

For the glass dataset, KNN, SVC, LR, DT gives $0.79, 0.61, 0.81, 0.81$ accuracy individually. When considered Borda count method, accuracy rises to 0.84 while majority voting gives 0.79 accuracy. Chapter 5 discusses the performance of Borda Ensembles on various datasets.

4.2 Experiment 2: Pruned Ensemble with LAE

In this experiment, We would be combining the algorithms discussed in sections 3.2 and 3.3 into one algorithm. The dataset used is Breast Cancer Wisconsin (Diagnostic) Data Set which aims to predict whether the cancer is benign or malignant.

Ten real-valued features are computed for each cell nucleus, which are

a) Radius (mean of distances from center to points on the perimeter)

b) Texture (standard deviation of gray-scale values)

c) Perimeter

d) Area

e) Smoothness (local variation in radius lengths)

f) Compactness ($\frac{perimeter^2}{area} - 1$)

g) Concavity

h) Number of concave points

i) Symmetry

h) Fractal dimension

We first load the data and preprocess it to facilitate further analysis.

```python
import pandas as pd
import numpy as np
columns = ['code_num','thickness','uofcsize','uofcshape','adhesion','secsize','bnuclei','chromatinb','\
                        nnucleoi','mitoses','output']
data = pd.read_csv('breast-cancer-wisconsin.data',names=columns)
data.drop(['code_num'],1,inplace=True)
data.replace('?',-99999, inplace=True)
data = data.astype(int)
X = np.array(data.drop(['output'], 1))
y = np.array(data['output'])
```

Secondly, we train four base learners, KNN (K Nearest Neighbours), SVM (Support Vector Machines), LR (Logistic Regression) and DT (Decision Tree) on very few training points. Training on very few examples ensures appropriate diversity of the ensemble.

```python
from sklearn import preprocessing,neighbors,svm
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn import tree

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.95,stratify=y)
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.80)

clf1 = neighbors.KNeighborsClassifier()
clf2 = svm.SVC()
clf3 = LogisticRegression()
clf4 = tree.DecisionTreeClassifier()

clf1.fit(X_train, y_train)
clf2.fit(X_train, y_train)
clf3.fit(X_train,y_train)
clf4.fit(X_train,y_train)

accuracy1 = clf1.score(X_test, y_test)
accuracy2 = clf2.score(X_test, y_test)
accuracy3 = clf3.score(X_test,y_test)
accuracy4 = clf4.score(X_test,y_test)

print(accuracy1,accuracy2,accuracy3,accuracy4)  # fetch the accuracy
```

Next we define two utility functions, one which returns the log odd given the probability, another one returns the neighborhood of a test point in the validation data space. The third function uses these two utility functions and assigns the weight of each classifier.

```
def get_weights(p):
p[p==1.0] = 0.99 #avoid inf error
odds = (p)/(1-p)
return np.log(odds)

from sklearn.neighbors import NearestNeighbors
neigh = NearestNeighbors(n_neighbors=3)
neigh.fit(X_val)

def get_local_weights(test_point,n_neigh):
        nearest_indices = neigh.kneighbors(test_point,n_neighbors=n_neigh,return_distance=False)[0]
        X_verify = X_val[nearest_indices]
        y_verify = y_val[nearest_indices]
        score_pred1 = clf1.score(X_verify,y_verify)
        score_pred2 = clf2.score(X_verify,y_verify)
        score_pred3 = clf3.score(X_verify,y_verify)
        acc_vector = np.array([score_pred1,score_pred2,score_pred3])
        weights=get_weights(acc_vector)
        return weights
```

Now, as this is a weighted voting game, If the classifier's weights reaches a particular quota, we would infer that the class is malignant else benign. The code for that is as follows

```
def get_weighted_prediction(sample_point):
        weights=get_local_weights(sample_point,4)
        prediction=np.array([clf1.predict([sample_point]),clf2.predict([sample_point]),\
                        clf3.predict([sample_point])])
        quota_weight = 0.0
        for _ in range(len(prediction)):
                if prediction[_] == 4:  # benign
                        quota_weight = quota_weight + weights[_]
                if quota_weight >= np.average(weights):
                        return 4
                else:
                        return 2
```

This gives us the implementation of local accuracy estimates part. Let's use the algorithm discussed in 3.3 to prune out the most effective subset.

First task here is to calculate the weights of the classifiers. We define disagreement measure, diversity measure and use that to calculate the weights.

```
def disagreement_measure(clf1,clf2,data):
        output_clf1 = clf1.predict(data)
        output_clf2 = clf2.predict(data)
        return 1- accuracy_score(output_clf1,output_clf2)

def diversity_measure(ensemble_list,data,i):
        ensemble_len = len(ensemble_list)
        diversity = 0
        for j in range(0,ensemble_len):
                if j == i:
                        continue
                diversity = diversity + disagreement_measure(ensemble_list[i],ensemble_list[j],data)
        return float(diversity)/float(ensemble_len-1)

diversity_values = []
for i in range(0,4):
        diversity_values.append(diversity_measure([clf1,clf2,clf3,clf4],X_val,i))
weights = [0,0,0,0]
for i in range(0,4):
        for j in range(0,4):
                if j == i:
                        continue
                if diversity_values[i] >= diversity_values[j]:
                        weights[i] = weights[i] + 1
```

Next, we find the Banzhaf power index of the game described in 3.3 using the following code. Refer Appendix A for the efficient implementation of Banzhaf Power Index.

```
double_banzhaf = banzhaf(weights,4) - banzhaf(weights,6)
print(double_banzhaf)
pruned_ensemble = []
pruned_weights = []

while sum(pruned_weights) <= 3:
        h = np.argmax(double_banzhaf)
        if sum(pruned_weights) + weights[h] >6:
                break
        pruned_ensemble.append(h)
        pruned_weights.append(weights[h])
        double_banzhaf[h] = -144
print(pruned_ensemble)
```

Lastly, we use this pruned_ensemble to get the weighted prediction. Again, if the quota_weight crosses a particular threshold, we take the decision of cancer being malignant else benign.

```
def get_weighted_prediction_ensemble(pruned_ensemble,sample_point):
        clf = {0: clf1, 1: clf2, 2: clf3, 3: clf4}
        weights=get_local_weights(sample_point,3)
        prediction=np.array([clf[i].predict([sample_point]) for i in pruned_ensemble] )
        quota_weight = 0.0
        for _ in range(len(prediction)):
                if prediction[_] == 4:
                quota_weight = quota_weight + weights[_]
        if quota_weight >= np.average(weights):
                return 4
        else:
                return 2
```

Next chapter discusses the result of this experiment over eight various datasets fetched from UCI Machine Learning repository. Note that this method is useful only for the case of two class classification. We can use one vs all method in the case of multi class classification but there are chances that more than one class may get selected using that method.

4.3 Experiment 3: Banzhaf Decision tree

In this section, we use banzhaf decision tree on Banknote dataset from UCI Machine Learning Repository. The banknote dataset predicts whether a given banknote is authentic given a number of measures taken from a photograph.

The dataset contains 1,372 with 5 numeric variables. It is a binary classification problem. The variables of the data are

1. Variance of the wavelet transformed image (continuous)
2. Skewness of the wavelet transformed image (continuous)
3. kurtosis of the wavelet transformed image (continuous)
4. entropy of the image (continuous)
5. class (integer)

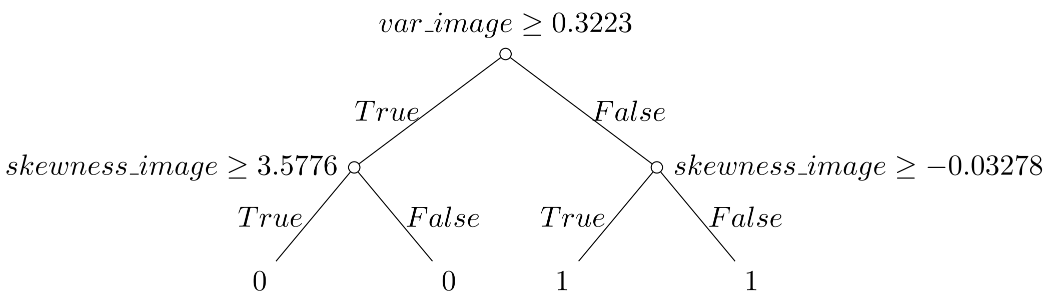
First step is data import and cleaning the dataset. The implementation of the Banzhaf decision tree is in the module *banzhaf_dt* and it's code can be found in Appendix A.

```
import pandas as pd
from sklearn.model_selection import train_test_split
data = pd.read_csv('data_banknote_authentication.txt',names=['var_image','skewness_image',
                        'kurtosis_image','entropy_image','output'])
X=data[data.columns[:-1]]
y = data[data.columns[-1]].to_frame()
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.40, random_state=42)
X_train.head()
```

The training dataset is stored in the variable *training_data* in the form of numpy array. We use this to build and visualize the Banzhaf decision tree using the following code.

```
from banzhaf_dt import DecisionTree
dt_banzhaf = DecisionTree(isBanzhaf=True)
dt_banzhaf.fit(X_train,y_train)
dt_banzhaf.print_the_tree()
```

Figure 4.1: Banzhaf Decision tree for Banknote Dataset



We see the data has already been segregated well by the information gain criterion so there is very little gain in extending the tree depth. The number of columns considered were also pretty low, so we can afford to do exponential computations.

4.4 Experiment 4: Pruned Banzhaf Decision Tree

In this section, we will again consider the Wisconsin breast cancer data and make a Banzhaf Decision tree. As the number of columns considered is high, we would need to select a small subset of features as exponential computations would consume a lot of time and hence, we will follow the algorithm described in 3.5

First, we import the necessary libraries and the dataset, do data preprocessing and then generate training and testing data using stratified sampling.

```
from banzhaf_dt import *
import pandas as pd
import numpy as np
import entropy_estimators
from sklearn.model_selection import train_test_split
import copy

columns = ['code_num','thickness','uofcsize','uofcshape','adhesion','secsize',\
                'bnuclei','chromatinb','nnucleoi','mitoses','output']
data = pd.read_csv('breast-cancer-wisconsin.data',names=columns)
data.drop(['code_num'],1,inplace=True)
data.replace('?',-99999, inplace=True)
data = data.astype(float)
data['output'] = data['output'].astype(int)
columns.pop(-1)
columns.pop(0)

X = data[columns]
y = data['output']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=42)
training_data = copy.copy(X_train)
training_data['output'] = y_train
training_data.head()
```

As we can see, there are many columns to be considered, we need to filter out the features such that the performance of the Banzhaf decision tree when it considers only the pruned features is more than the traditional Banzhaf tree algorithm. We use the algorithm discussed in 3.5which uses conditional mutual information for the feature selection.

The implementation of the algorithm discussed in 3.5 is as follows

```
import entropy_estimators as ee
def select_kbest_features(k,data):
        output = list(data.iloc[:,-1].values.reshape(-1,1))
        S = []
        while k > 0:
                curr_info = 0
                MI = []
                columns = list(data.columns.values)
                for f in columns[:-1]:
                        if len(S) == 0:
                                y = list(data[f].values.reshape(-1,1))
                                curr_info = ee.mi(output,y)
                        else:
                                if f in S:
                                        continue
                                else:
                                        y = list(data[f].values.reshape(-1,1))
                                        z = list(data[S].values.reshape(-1,len(S)))
                                        curr_info = ee.cmi(output,y,z)
                                if curr_info == 0:
                                        data.drop(f, axis=1, inplace=True)
                                else:
                                        MI.append({'value':curr_info,'label':f})
                maxMIfeature = max(MI, key=lambda x:x['value'])
                S.append(maxMIfeature['label'])
                k = k - 1
        return S
```

Now, we prune the features and build the Banzhaf decision tree, just like the last experiment.

```
from banzhaf_dt import select_kbest_features
from banzhaf_dt import DecisionTree
selected_features = select_kbest_features(5,training_data)
selected_features.append(training_data.columns[-1])
training_data = training_data[selected_features]
dt = DecisionTree(isBanzhaf=True)
X = training_data[training_data.columns[:-1]]
y = training_data[training_data.columns[-1]].to_frame()
dt.fit(X,y)
dt.print_the_tree()
```

The below diagram shows the structure of the Banzhaf decision tree when feature selection is considered.

```
Is bnuclei >= 3.0?
--> True:
        Is uofcshape >= 5.5?
        --> True:
                Is uofcshape >= 8.0?
                --> True:
                        Predict {2.0: 1, 4.0: 57}
                --> False:
                        Predict {2.0: 1, 4.0: 38}
        --> False:
                Is uofcshape >= 3.0?
                --> True:
                        Predict {2.0: 4, 4.0: 53}
                --> False:
                        Predict {2.0: 15, 4.0: 4}
--> False:
        Is uofcshape >= 5.11111111111?
        --> True:
                Is uofcshape >= 7.75?
                --> True:
                        Predict {4.0: 5}
                --> False:
                        Predict {2.0: 3, 4.0: 2}
        --> False:
                Is uofcshape >= 3.0?
                --> True:
                        Predict {2.0: 29, 4.0: 4}
                --> False:
                        Predict {2.0: 251, 4.0: 1}]
```

The following diagram shows the structure of Banzhaf decision tree when feature selection is not considered.

```
Is uofcshape >= 3.0?
--> True:
        Is thickness >= 5.5?
        --> True:
                Is thickness >= 8.0?
                --> True:
                        Predict {2.0: 1, 4.0: 57}
                --> False:
                        Predict {2.0: 1, 4.0: 38}
        --> False:
                Is thickness >= 3.0?
                --> True:
                        Predict {2.0: 4, 4.0: 53}
                --> False:
                        Predict {2.0: 15, 4.0: 4}
--> False:
        Is thickness >= 5.11111111111?
        --> True:
                Is adhesion >= 5.42857142857?
                --> True:
                        Predict {2.0: 1, 4.0: 3}
                --> False:
                        Predict {2.0: 2, 4.0: 4}
        --> False:
                Is thickness >= 3.0?
                --> True:
                        Predict {2.0: 29, 4.0: 4}
                --> False:
                        Predict {2.0: 251, 4.0: 1}
```

The performance analysis is done in Chapter 5. Appendix A has the implementation of the *banzhaf_dt* library.

4.5 Experiment 5: Banzhaf Random Forest

In this section, we will implement the Pruned Banzhaf random forest with majority voting. We will be using soybean dataset which can be obtained from UCI Machine learning repository. As usual, the first step is data import using pandas data frame.

```
import pandas as pd
from sklearn.model_selection import train_test_split
data = pd.read_csv('data_banknote_authentication.txt',names=['var_image',
                                        'skewness_image','kurtosis_image','entropy_image','output'])
X=data[data.columns[:-1]]
y = data[data.columns[-1]].to_frame()
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.40, random_state=42)
X_train.head()
```

Next step is to create the Banzhaf Random forests using *banzhaf_rf* library.

```
from banzhaf_rf import BanzhafRandomForest
brf = BanzhafRandomForest(num_of_trees=6,tree_depth=2)
brf.fit(X_train,y_train)
```

The structure of the code and other functions can be found in Appendix A. In the next chapter, we will be doing the performance analysis of Pruned Banzhaf Random Forest with Borda Count vs Normal Banzhaf Random Forest[18].

Chapter 5

Discussion of the Experimental Results

This chapter evaluates the algorithms described in Chapter 4 on several datasets from the UCI machine learning repository. This evaluation is being done to find out how certain algorithms of game theory can be applied to ensemble learning can help in making more sound decisions. We act on the assumption that only 66% of the data is being chosen for training and rest is for testing purposes. This is to prevent overfitting of the models.

5.1 Discussion 1: Borda Count vs Majority Voting

In this example, we chose an ensemble of KNN, SVC and DT. Four datasets were taken, namely Glass dataset, Wisconsin breast cancer dataset ,sonar dataset and ISOLET dataset. We compare the performance of majority vote classifier and Borda count classifier on these datasets .On each dataset, 10-Fold cross validation is applied. To tackle the class imbalance problem, the minority samples are up-sampled with replacement.

The following table gives us the performance of these algorithms on the datasets mentioned above in terms of cross validation accuracy.

	KNN	SVC	DT	BC-ENSEMBLE	MV-ENSEMBLE
GLASS	96.67 ± 5.37	90.22 ± 7.85	97.78 ± 4.68	97.78 ± 4.68	96.67 ± 5.37
BC-WINC	97.28 ± 4.61	96.23 ± 3.6	95.64 ± 5.66	97.81 ± 2.83	97.81 ± 2.83
SONAR	82.38 ± 7.79	58.57 ± 10.54	86.67 ± 4.38	83.81 ± 9.04	83.33 ± 10.1
ISOLET	68.8 ± 3.68	48.72 ± 6.6	60.13 ± 8.35	67.97 ± 8.15	64.76 ± 7.15

TABLE 5.1: Validation accuracy comparision between Borda Count and Majority Voting

	BASELINE	KNN	SVC	DT	BC-ENSEMBLE	MV-ENSEMBLE
GLASS	35.47	90.27	87.32	97.18	97.18	94.37
BC-WINC	65.54	97.0	95.5	92.91	97.14	97.0
SONAR	53.29	91.67	83.33	83.33	100.0	100.0
ISOLET	3.85	87.48	94.51	79.32	83.88	92.72

Table 5.2: Test accuracy comparision between Borda Count and Majority Voting

	KNN	SVC	DT	BC-ENSEMBLE	MV-ENSEMBLE
GLASS	0.81	0.55	0.67	0.77	0.68
BC-WINC	0.96	0.94	0.92	0.96	0.94
SONAR	0.67	0.5	0.8	0.85	0.81
ISOLET	0.87	0.74	0.78	0.83	0.72

Table 5.3: F measure comparision between Borda Count and Majority Voting

The following table gives us the performance of these algorithms on the datasets mentioned above in terms of test accuracy.

BASELINE is the classifier which always outputs the maximum occurring label in the testing data as it's prediction.

The following table gives us the performance of these algorithms on the datasets mentioned above in terms of F-1 score. This metric tells us how precise a classifier is in a true sense.

We have assumed the cost of classification and misclassification to be same here. If they are not same, we would consider F_β score. In the case of multi-class classification, it is to be noted that the average of F scores are taken against every class. This does not take class imbalance in account.

We can clearly see that Borda count performs well than Majority Voting except in the ISOLET dataset. F-Score of Borda count method is clearly more than that of majority voting. As borda count method requires the participants to deliver a honest opinion, we must re sample minority classes to match the majority classes else the classifier would be biased towards the majority class.

5.2 Discussion 2: Bagging vs Ensemble Pruning with LAE

In this section, we apply Bagging on the ensemble considered previously as well as SCG Pruning with LAE (Chapter 4) and compare their performances. Once again, 33% of the dataset is used for testing, rest for training. In case of more than two classes, one vs all method is used.

The following table gives us the performance of these algorithms on the datasets mentioned in terms of test accuracy.

	BASELINE	KNN	SVC	DT	Bagging	SCG Pruning with LAE
GLASS	35.47	90.27	87.32	97.18	98.18	99.97
BC-WINC	65.54	97.0	95.5	92.91	98.14	99.00
SONAR	53.29	91.67	83.33	83.33	99.33	100.0
ISOLET	3.85	87.48	94.51	79.32	87.99	95.72

Table 5.4: Test accuracy comparision between Bagging and SCG Pruning with LAE

The following table gives us the performance with respect to F- measures. Average of F measures is taken across every class.

	KNN	SVC	DT	Bagging	SCG Pruning with LAE
GLASS	0.81	0.55	0.67	0.87	0.98
BC-WINC	0.96	0.94	0.92	0.96	0.97
SONAR	0.67	0.5	0.8	0.89	0.91
ISOLET	0.87	0.74	0.78	0.92	0.93

Table 5.5: F measure comparision between Bagging and SCG Pruning with LAE

This indeed shows promising results. SCG Pruning with LAE shows better test accuracy as well as F scores. We can't generalize this performance metric because this is measured across just four datasets.

5.3 Discussion 3: BRF vs Pruned BRF with Borda Count

In this section, we check the performance of Banzhaf random forest when Majority voting is applied and Borda count is applied. The number of trees is set to $\lceil log_2 h \rceil$ as it is reported by the paper[18] to perform better than others.

Datasets	BRF	PRUNED BRF WITH BORDA
soybean	1.0000	0.9989
iris	0.9467	0.9767
wine	0.9717	0.9756
sonar	0.7120	0.8156
thyroid	0.9395	0.9673
ecoli	0.6665	0.6730
dermatology	0.9677	0.9710
musk2	0.8700	0.850
shuttle	0.9951	1.0000

Table 5.6: Classification accuracy comparision between BRF and Pruned BRF with Borda Count.

Although performance of the proposed algorithm is better, the run times are high because of resampling, selecting best features for every tree. Every Banzhaf tree of the Banzhaf random forest has depth set to 4. This is to reduce the computational time.

Datasets	BRF	PRUNED BRF WITH BORDA
soybean	1.654	4.835
iris	9.134	21.783
wine	8.778	19.392
sonar	2.297	5.8157
thyroid	3.168	9.963
dermatology	11.023	45.97
shuttle	80.660	156.003

Table 5.7: Running time comparision between BRF and Pruned BRF with Borda Count.

5.4 Conclusion

The proposed Pruned Banzhaf Random Forests with Borda count method gives us a better performance by taking into consideration the concepts of cooperative game theory. Experiments show that the pruned BRF performs slightly better than the BRF. Borda count method also give better F scores than Majority voting so Pruned BRF with Borda count is more robust than Majority Voting. In case of normal ensembles, SCG Pruning with LAE seems to give us better performance than bagging. This concludes the thesis that game theory can indeed be used in the case of ensembles to improve their performance even more.

Appendix A

Algorithmic Implementations

A.1 Bagging

```python
#Step 1: Fetch and clean the dataset
import pandas as pd

#fetch the dataset
df_wine = pd.read_csv('https://archive.ics.uci.edu/ml/'
                      'machine-learning-databases/wine/wine.data',
                      header=None)

#assign columns
df_wine.columns = ['Class label', 'Alcohol', 'Malic acid', 'Ash',
                   'Alcalinity of ash', 'Magnesium', 'Total phenols',
                   'Flavanoids', 'Nonflavanoid phenols', 'Proanthocyanins',
                   'Color intensity', 'Hue', 'OD280/OD315 of diluted wines',
                   'Proline']

# drop 1 class to make it two class classification problem
df_wine = df_wine[df_wine['Class label'] != 1]

y = df_wine['Class label'].values
X = df_wine[['Alcohol', 'OD280/OD315 of diluted wines']].values

#Step 2: Encode the output values and sample
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split


le = LabelEncoder()
y = le.fit_transform(y)

```

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
                             test_size=0.2,
                             random_state=1,
                             stratify=y)


#Step 3: Initialize the classifiers
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

tree = DecisionTreeClassifier(criterion='entropy',
                              max_depth=None,
                              random_state=1)

bag = BaggingClassifier(base_estimator=tree,
                        n_estimators=500,
                        max_samples=1.0,
                        max_features=1.0,
                        bootstrap=True,
                        bootstrap_features=False,
                        n_jobs=1,
                        random_state=1)


#Step 4: Check the accuracy
from sklearn.metrics import accuracy_score

tree = tree.fit(X_train, y_train)
y_train_pred = tree.predict(X_train)
y_test_pred = tree.predict(X_test)

tree_train = accuracy_score(y_train, y_train_pred)
tree_test = accuracy_score(y_test, y_test_pred)
print('Decision tree train/test accuracies %.3f/%.3f'
      % (tree_train, tree_test))

bag = bag.fit(X_train, y_train)
y_train_pred = bag.predict(X_train)
y_test_pred = bag.predict(X_test)

bag_train = accuracy_score(y_train, y_train_pred)
bag_test = accuracy_score(y_test, y_test_pred)
print('Bagging train/test accuracies %.3f/%.3f'
      % (bag_train, bag_test))


# Decision tree train/test accuracies 1.000/0.833
# Bagging train/test accuracies 1.000/0.917
```

A.2 Boosting

```python
# coding: utf-8

# In[1]:



#Step 1: Fetch and clean the dataset
import pandas as pd

#fetch the dataset
df_wine = pd.read_csv('https://archive.ics.uci.edu/ml/'
                      'machine-learning-databases/wine/wine.data',
                      header=None)

#assign columns
df_wine.columns = ['Class label', 'Alcohol', 'Malic acid', 'Ash',
                   'Alcalinity of ash', 'Magnesium', 'Total phenols',
                   'Flavanoids', 'Nonflavanoid phenols', 'Proanthocyanins',
                   'Color intensity', 'Hue', 'OD280/OD315 of diluted wines',
                   'Proline']

# drop 1 class to make it two class classification problem
df_wine = df_wine[df_wine['Class label'] != 1]

y = df_wine['Class label'].values
X = df_wine[['Alcohol', 'OD280/OD315 of diluted wines']].values


# In[2]:


#Step 2: Encode the output values and sample
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split


le = LabelEncoder()
y = le.fit_transform(y)

X_train, X_test, y_train, y_test = train_test_split(X, y,
                                 test_size=0.2,
                                 random_state=1,
                                 stratify=y)



# In[4]:
```

```


#Step 3: Initialize the Classifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
tree = DecisionTreeClassifier(criterion='entropy',
                              max_depth=1,
                              random_state=1)

ada = AdaBoostClassifier(base_estimator=tree,
                         n_estimators=500,
                         learning_rate=0.1,
                         random_state=1)


# In[6]:


#Step 4: Check the accuracy
from sklearn.metrics import accuracy_score

tree = tree.fit(X_train, y_train)
y_train_pred = tree.predict(X_train)
y_test_pred = tree.predict(X_test)

tree_train = accuracy_score(y_train, y_train_pred)
tree_test = accuracy_score(y_test, y_test_pred)
print('Decision tree train/test accuracies %.3f/%.3f'
      % (tree_train, tree_test))

ada = ada.fit(X_train, y_train)
y_train_pred = ada.predict(X_train)
y_test_pred = ada.predict(X_test)

ada_train = accuracy_score(y_train, y_train_pred)
ada_test = accuracy_score(y_test, y_test_pred)
print('AdaBoost train/test accuracies %.3f/%.3f'
      % (ada_train, ada_test))

# Decision tree train/test accuracies 0.916/0.875
# AdaBoost train/test accuracies 1.000/0.917
```

A.3 Banzhaf Power Index Calculation

```
def banzhaf(weight, quota):

    max_order = sum(weight)

    polynomial = [1] + max_order*[0]               # create a list to hold the polynomial coefficients

    current_order = 0                              # compute the polynomial coefficients
    aux_polynomial = polynomial[:]
    for i in range(len(weight)):
        current_order = current_order + weight[i]
        offset_polynomial = weight[i]*[0]+polynomial
        for j in range(current_order+1):
            aux_polynomial[j] = polynomial[j] + offset_polynomial[j]
        polynomial = aux_polynomial[:]

    banzhaf_power = len(weight)*[0]                                 # create a list to hold the Banzhaf Power f
    swings = quota*[0]                                              # create a list to compute the swings for e

    for i in range(len(weight)):                                    # compute the Banzhaf Power
        for j in range(quota):                                      # fill the swings list
            if (j<weight[i]):
                swings[j] = polynomial[j]
            else:
                swings[j] = polynomial[j] - swings[j-weight[i]]
        for k in range(weight[i]):                                  # fill the Banzhaf Power vector
            banzhaf_power[i] = banzhaf_power[i] + swings[quota-1-k]

    # Normalize Index
    total_power = float(sum(banzhaf_power))
    banzhaf_index = map(lambda x: x / total_power, banzhaf_power)

    return np.array(list(banzhaf_index))
```

A.4 Banzhaf Decision Tree

```python
'''

First we initialize the utility functions

'''
from itertools import chain, combinations
from statistics import mean
import numpy as np
import entropy_estimators as ee
import pandas as pd
from sklearn.metrics import accuracy_score

def argmax(lst):
    """ Returns the position of maximal element of the list """
    return lst.index(max(lst))

def unique_vals(rows, col):
    """Find the unique values for a column in a dataset."""
    return set([row[col] for row in rows])

def class_counts(rows):
    """Counts the number of each type of example in a dataset."""
    counts = {}  # a dictionary of label -> count.
    for row in rows:
        # in our dataset format, the label is always the last column
        label = row[-1]
        if label not in counts:
            counts[label] = 0
        counts[label] += 1
    return counts

def is_numeric(value):
    """Test if a value is numeric."""
    return isinstance(value, int) or isinstance(value, float)



def powerset(iterable):
    '''
    powerset([1,2,3]) --> () (1,) (2,) (3,) (1,2) (1,3) (2,3) (1,2,3)
    '''
    s = list(iterable)
    return list(chain.from_iterable(combinations(s, r) for r in range(len(s)+1)))[1:]

class Question:
    """A Question is used to partition a dataset.

    This class just records a 'column number' (e.g., 0 for Color) and a
    'column value' (e.g., Green). The 'match' method is used to compare
    the feature value in an example to the feature value stored in the
    question. See the demo below.
    """

    def __init__(self, columns, column, value):
        self.columns = columns
        self.column = column
        self.value = value

    def match(self, example):
        ''' Compare the feature value in an example to the
         feature value in this question. '''
        val = example[self.column]
        if is_numeric(val):
            return val >= self.value
        else:
            return val == self.value

    def __repr__(self):
        # This is just a helper method to print
        # the question in a readable format.
        condition = "=="
        if is_numeric(self.value):
            condition = ">="
        return "Is %s %s %s?" % (
            self.columns[self.column], condition, str(self.value))

def partition(rows, question):
    """Partitions a dataset.

    For each row in the dataset, check if it matches the question. If
    so, add it to 'true rows', otherwise, add it to 'false rows'.
    """
    true_rows, false_rows = [], []
    for row in rows:
        if question.match(row):
            true_rows.append(row)
        else:
            false_rows.append(row)
    return np.array(true_rows), np.array(false_rows)

def gini(rows):
    """Calculate the Gini Impurity for a list of rows.

    """
    counts = class_counts(rows)
    impurity = 1
    for lbl in counts:
        prob_of_lbl = counts[lbl] / float(len(rows))
        impurity -= prob_of_lbl**2
    return impurity

def info_gain(left, right, current_uncertainty):
    """Information Gain.

    The uncertainty of the starting node, minus the weighted impurity of
    two child nodes.
    """
    p = float(len(left)) / (len(left) + len(right))
    return current_uncertainty - p * gini(left) - (1 - p) * gini(right)

def find_best_split_inf_gain(rows, columns):
    """Find the best question to ask by iterating over every feature / value
    and calculating the information gain."""
    best_gain = 0  # keep track of the best information gain
    best_question = None  # keep train of the feature / value that produced it
    current_uncertainty = gini(rows)
    n_features = len(rows[0]) - 1  # number of columns

    for col in range(n_features):  # for each feature
        values = set([row[col] for row in rows])  # unique values in the column

        for val in values:  # for each value

            question = Question(columns, col, val)

            # try splitting the dataset
            true_rows, false_rows = partition(rows, question)

            # Skip this split if it doesn't divide the
            # dataset.
            if len(true_rows) == 0 or len(false_rows) == 0:
                continue

            # Calculate the information gain from this split
            gain = info_gain(true_rows, false_rows, current_uncertainty)

            # You actually can use '>' instead of '>=' here
            # but I wanted the tree to look a certain way for our
            # toy dataset.
            if gain >= best_gain:
                best_gain, best_question = gain, question

    return best_question

##### Banzhaf Part starts




def isWinning(player, coalition, threshold=0.05):
    ''' Checks if inclusion of the player in the coalition
        leads to winning situation.
        The player wins if the player is interdependent on
        atleast half of the members in the coalition.
        The interdependence is measured using conditional mutual information.
        Inputs: player: Pandas dataframe
         coalition: Pandas dataframe
         threshold: boundary value of interdependence
        Outputs: Boolean which returns True if inclusion leads to winning coalition.
    '''
    total_dependence = 0
    x = player.values.reshape(-1, 1).tolist()
    if coalition.shape[1] == 1:
        return ee.mi(x, coalition.values.reshape(-1, 1).tolist()) >= threshold

    for i in range(0, coalition.shape[1]):
        y = coalition.drop(coalition.columns[i], axis=1).values.tolist()
        z = coalition[coalition.columns[i]].values.reshape(-1, 1).tolist()
        if ee.cmi(x, y, z) >= threshold:
            total_dependence = total_dependence + 1
    return float(total_dependence)/float(len(coalition)) >= threshold

def banzhaf(df, column, threshold=0.05):
    '''
    Calculates the banzhaf power index of the feature whose data is given by the argument column
    Inputs: df: Pandas dataframe
            column_name: Pandas dataframe
    Outputs: Banzhaf Power Index
    '''
    assert column.columns[0] not in df.columns, "Player should not be part of coalition"
    assert column.shape[1] == 1, "Only one player is allowed"
    all_coalitions = powerset(df.columns)
    positive_swing = 0
    for coalition in all_coalitions:
        coalition = list(coalition)
        if isWinning(column, df[coalition], threshold):
            positive_swing = positive_swing + 1
    banzhaf_index = float(positive_swing) / (len(all_coalitions))
    return round(banzhaf_index, 4)
```

```python

def find_best_split_banzhaf(data):
    """Find the best question to ask by calculating the banzhaf power index
    and selecting the maximum value from it """
    banzhaf_rows = []
    for i in range(data.shape[1]):
        df = data.drop(data.columns[i], axis=1)
        column = data[data.columns[i]].to_frame()
        banzhaf_rows.append(banzhaf(df, column))
    idx_max = argmax(banzhaf_rows)
    mean_val = mean(unique_vals(data.values, idx_max))
    question = Question(data.columns, idx_max, mean_val)
    return question


class Decision_Node:
    """A Decision Node asks a question.

    This holds a reference to the question, and to the two child nodes.
    """

    def __init__(self,
                 question,
                 true_branch,
                 false_branch):
        self.question = question
        self.true_branch = true_branch
        self.false_branch = false_branch

class Leaf:
    """A Leaf node classifies data.

    This holds a dictionary of class (e.g., "Apple") -> number of times
    it appears in the rows from the training data that reach this leaf.
    """

    def __init__(self, rows):
        self.predictions = class_counts(rows)

def build_tree(data, isbanzhaf=False, depth=3):
    """ Builds the decision tree
    Inputs: data: the training dataset
            banzhaf: True if user wants Banzhaf Decision Tree
            depth: depth of decision tree desiried
    Outputs: Decision node
    """
    rows = data.values
    if depth <= 0:
        return Leaf(rows)
    question = find_best_split_inf_gain(rows, data.columns)
    true_rows, false_rows = partition(rows, question)
    if true_rows.size == 0:
        true_branch = Leaf(rows)
    if false_rows.size == 0:
        false_branch = Leaf(rows)
    if true_rows.size and false_rows.size:
        true = pd.DataFrame(true_rows, columns=data.columns)
        false = pd.DataFrame(false_rows, columns=data.columns)
        if isbanzhaf:
            true_branch = build_banzhaf_tree(true, depth=depth-1)
            false_branch = build_banzhaf_tree(false, depth=depth-1)
        else:
            true_branch = build_tree(true, depth=depth-1)
            false_branch = build_tree(false, depth=depth-1)
    return Decision_Node(question, true_branch, false_branch)


def build_banzhaf_tree(rows, depth=1):
    ''' Builds pure Banzhaf Decision Tree
    '''
    if depth <= 0:
        return Leaf(rows.values)
    question = find_best_split_banzhaf(rows)
    true_rows, false_rows = partition(rows.values, question)
    if true_rows.size == 0:
        true_branch = Leaf(rows)
    if false_rows.size == 0:
        false_branch = Leaf(rows)
    if true_rows.size and false_rows.size:
        true = pd.DataFrame(true_rows, columns=rows.columns)
        false = pd.DataFrame(false_rows, columns=rows.columns)
        true_branch = build_banzhaf_tree(true, depth=depth-1)
        false_branch = build_banzhaf_tree(false, depth=depth-1)
    return Decision_Node(question, true_branch, false_branch)

def print_tree(node, spacing=""):
    """World's most elegant tree printing function."""

    # Base case: we've reached a leaf
    if isinstance(node, Leaf):
        print (spacing + "Predict", node.predictions)
        return

    # Print the question at this node
    print (spacing + str(node.question))

    # Call this function recursively on the true branch
    print (spacing + '--> True:')
    print_tree(node.true_branch, spacing + "  ")

    # Call this function recursively on the false branch
    print (spacing + '--> False:')
    print_tree(node.false_branch, spacing + "  ")

def classify(row, node):
    """See the 'rules of recursion' above."""

    # Base case: we've reached a leaf
    if isinstance(node, Leaf):
        return max(node.predictions, key=node.predictions.get)

    # Decide whether to follow the true-branch or the false-branch.
    # Compare the feature / value stored in the node,
    # to the example we're considering.
    if node.question.match(row):
        return classify(row, node.true_branch)
    return classify(row, node.false_branch)

def select_kbest_features(k, data):
    """
    Implements the algorithm given in Section 3.5
    """
    k = k +1
    output = list(data.iloc[:, -1].values.reshape(-1, 1))
    S = []
    while k > 0:
        curr_info = 0
        MI = []
        columns = list(data.columns.values)
        for f in columns[:-1]:
            if len(S) == 0:
                y = list(data[f].values.reshape(-1, 1))
                curr_info = ee.mi(output,y)
            else:
                if f in S:
                    continue
                else:
                    y = list(data[f].values.reshape(-1,1))
                    z = list(data[S].values.reshape(-1,len(S)))
                    curr_info = ee.cmi(output,y,z)
            if curr_info == 0:
                data.drop(f, axis=1, inplace=True)
            else:
                MI.append({'value':curr_info,'label':f})
        maxMIfeature = max(MI, key=lambda x:x['value'])
        S.append(maxMIfeature['label'])
        k = k - 1
    return S


class DecisionTree:
    '''
    This class is the main decision tree
    '''
    def __init__(self, isBanzhaf):
        self.is_banzhaf = isBanzhaf
        self.data = None
        self.tree = None
    def fit(self, X, y, depth=2):
        """ Fits the decision tree
        """
        self.data = pd.concat([X, y], axis=1)
        self.tree = build_tree(self.data, self.is_banzhaf,depth)
    def print_the_tree(self):
        """
        Prints the tree
        """
        print_tree(self.tree)
    def predict(self, x_test):
        """
        Predicts the class
        """
        return classify(x_test, self.tree)
    def score(self, y_pred, X_test):
        """
        Returns the score
        """
        predictions = []
        for x_test in X_test:
            predictions.append(self.predict(x_test))
        return accuracy_score(y_pred, predictions)
```

A.5 Banzhaf Random Forest

```python
"""This module defines the functions required for the implementation of banzhaf random forests."""
import operator
from collections import Counter
from banzhaf_dt import DecisionTree, select_kbest_features
from sklearn.utils import resample
import pandas as pd
import numpy as np

def generate_preference(pred_proba, labels):
    '''
    Accepts:
        pred_proba: Numpy array containing probabilities
        labels: list containing output labels
    Returns: Preference in form of list of list
    '''
    num_class = pred_proba.shape[0]
    vote_dic = {}
    for i in range(num_class):
        vote_dic[labels[i]] = pred_proba[i]
    sorted_x = sorted(vote_dic.items(), key=operator.itemgetter(1))
    sorted_x.reverse()
    preference = []
    for i in range(num_class):
        preference.append(sorted_x[i][0])
    return list(preference)

def borda(preference_ballot):
    '''
    Accepts: list of list => preference_ballot
    Returns: Winner
    '''
    counts = {}
    candidates = list(set(preference_ballot[0]))
    max_point = len(candidates)
    for i in range(max_point):
        counts[candidates[i]] = 0
    for pref in preference_ballot:
        for i in range(len(pref)):
            counts[pref[i]] += (max_point -i)
    return max(counts, key=counts.get)

def get_prediction_borda(pred_proba_list, labels):
    '''
    Gets the Borda Winner
    Inputs: pred_proba: A numpy array
```

```
            labels: a list containing the output values
    '''
    preference_ballot = []
    for prob_list in pred_proba_list:
        preference_ballot.append(generate_preference(prob_list, labels))
    return borda(preference_ballot)

class BanzhafRandomForest:
    '''
    A Banzhaf Random Forest Library
    '''
    def __init__(self,num_of_trees,tree_depth):
        self.ensemble = [DecisionTree(isBanzhaf=True) for _ in range(num_of_trees)]
        self.depth = tree_depth
        self.considered_features = []
        self.labels = None
    def fit(self, X, y, depth=2):
        """
        Generates Bootstrap sampling, does feature selection and makes the ensemble
        of Banzhaf Decision Trees
        """
        self.labels = list(y[y.columns[0]].unique())
        dataframe = pd.concat([X,y],axis=1)
        for i in range(len(self.ensemble)):
            dataframe_bl = resample(dataframe,
                                    replace=True,     # sample with replacement
                                    n_samples=int((dataframe.shape[0])*0.66))
            if dataframe.shape[1] <=5:
                selected_features = list(X.columns)
                self.considered_features.append(selected_features)
            else:
                selected_features = select_kbest_features(min(dataframe.shape[1],5), dataframe_bl) #select five
                self.considered_features.append(selected_features)
            selected_features.append(y.columns[0])
            dataframe_bl = dataframe_bl[selected_features]
            X_bl = dataframe_bl[dataframe_bl.columns[:-1]]
            y_bl = dataframe_bl[dataframe_bl.columns[-1]].to_frame()
            self.ensemble[i].fit(X_bl, y_bl, depth=depth)
            selected_features.pop(-1)
            print(i+1, 'Trees Built!')

    def predict(self, x_test, isBorda=True):
        """
        Generates the prediction of Banzhaf Random Forests
        Inputs: x_test: The testing sample
                isBorda: True when Borda Count should be used for Prediction
        """
        if isBorda:
            pred_prob_list = []
            for i in range(len(self.ensemble)):
                x_test_bl = x_test[self.considered_features[i]]
                x_test_bl_values = x_test_bl.values.reshape(-1,)
                pred_prob_list.append(list(self.ensemble[i].predict_proba(x_test_bl_values).values()))
            pred_prob_list = np.array(pred_prob_list)
            return get_prediction_borda(pred_prob_list, self.labels)
        predictions = []
        for i in range(len(self.ensemble)):
            x_test_bl = x_test[self.considered_features[i]]
            predictions.append(self.ensemble[i].predict(x_test_bl))
        occ = Counter(predictions)
        return int(max(occ, key=occ.get))
```

References

[1] Yoram Bachrach et al. "Approximating power indices". In: *In AAMAS (2.* 2008, pp. 943–950.

[2] Avrim L. Blum and Ronald L. Rivest. "Training a 3-node neural network is NP-complete". In: *Machine Learning: From Theory to Applications: Cooperative Research at Siemens and MIT.* Ed. by Stephen José Hanson, Werner Remmele, and Ronald L. Rivest. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 9–28. ISBN: 978-3-540-47568-2. DOI: `10.1007/3-540-56483-7_20`.

[3] Leo Breiman. "Bagging predictors". In: *Machine Learning* 24.2 (1996), pp. 123–140. ISSN: 1573-0565. DOI: `10.1007/BF00058655`.

[4] Thomas G. Dietterich. "Ensemble Methods in Machine Learning". In: *Multiple Classifier Systems: First International Workshop, MCS 2000 Cagliari, Italy, June 21–23, 2000 Proceedings.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 1–15. ISBN: 978-3-540-45014-6. DOI: `10.1007/3-540-45014-9_1`.

[5] Prajit K. Dutta. *Strategies and games: theory and practice.* Massachusetts Institute of Technology, 1999.

[6] Nicolás García-pedrajas et al. "Nonlinear boosting projections for ensemble construction". In: *Journal of Machine Learning Research* ().

[7] Harris V. Georgiou and Michael E. Mavroforakis. "A game-theoretic framework for classifier ensembles using weighted majority voting with local accuracy estimates". In: *CoRR* abs/1302.0540 (2013).

[8] Sergiu Hart. "Shapley Value". In: *Game Theory.* Ed. by John Eatwell, Murray Milgate, and Peter Newman. London: Palgrave Macmillan UK, 1989, pp. 210–216. ISBN: 978-1-349-20181-5. DOI: `10.1007/978-1-349-20181-5_25`.

[9] Oskar Morgenstern John Von Neumann. *Theory of Games and Economic Behavior.* 3rd. Princeton University Press, 1966.

[10] M. Lichman. *UCI Machine Learning Repository.* 2013. URL: `http://archive.ics.uci.edu/ml`.

[11] Huawen Liu et al. "An Effective Feature Selection Method Using Dynamic Information Criterion". In: *Artificial Intelligence and Computational Intelligence: Third International Conference, AICI 2011, Taiyuan, China, September 24-25, 2011, Proceedings, Part I.* Ed. by Hepu Deng et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 450–455. ISBN: 978-3-642-23881-9. DOI: `10.1007/978-3-642-23881-9_59`.

[12] Yasuko Matsui and Tomomi Matsui. "NP-completeness for Calculating Power Indices of Weighted Majority Games". In: *Theoretical Computer Science* 263 (1998), pp. 98–01.

[13] Bartosz Meglicki. *Generating functions partitioning algorithm for computing power indices in weighted voting games.* 2010. eprint: `arXiv:1011.6543`.

[14] L. Nordmann and H. Pham. "Weighted voting systems". In: *IEEE Transactions on Reliability* 48.1 (1999), pp. 42–49. ISSN: 0018-9529. DOI: `10.1109/24.765926`.

[15] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[16] Robert E. Schapire. "The strength of weak learnability". In: *Machine Learning* 5.2 (1990), pp. 197–227. ISSN: 1573-0565. DOI: `10.1007/BF00116037`.

[17] Lloyd Shapley and Bernard Grofman. "Optimizing group judgmental accuracy in the presence of interdependencies". In: *Public Choice* 43.3 (1984), pp. 329–343. ISSN: 1573-7101. DOI: `10.1007/BF00118940`.

[18] Jianyuan Sun et al. *Banzhaf Random Forests.* 2015. eprint: `arXiv:1507.06105`.

[19] Hadjer Ykhlef and Djamel Bouchaffra. "An efficient ensemble pruning approach based on simple coalitional games". In: 34 (June 2016).

[20] Z. H. Zhou, J. Wu, and W. Tang. "Ensembling Neural Networks: Many Could Be Better Than All". In: *Artificial Intelligence* 137.1-2 (2002), pp. 239–263.

[21] Zhi-Hua Zhou. "Ensemble Learning". In: *Encyclopedia of Biometrics.* Ed. by Stan Z. Li and Anil Jain. Boston, MA: Springer US, 2009, pp. 270–273. ISBN: 978-0-387-73003-5. DOI: `10.1007/978-0-387-73003-5_293`.