School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

Ripple Down Rules for Explainable AI

by

Nathan Driscoll

Thesis submitted as a requirement for the degree of
Bachelor of Engineering in Software Engineering

Submitted: Nov 2021

Student ID: z5204935

Supervisor: Dr Mike Bain

Assessor: Prof. Paul Compton

Abstract

Owing to the difficulty of implementing AI systems, many real-world applications of AI are based on machine learning. Currently the most popular machine learning methods result in models that are too complex to be understood by humans. This is a barrier to adoption in many real-world settings which require the predictions of a machine learned model to be explainable on demand, for legal or other reasons.

This project will investigate state-of-the-art methods for explaining the predictions of AI systems based on complex machine learned models, specifically exploring the use of Ripple Down Rules for explaining the predictions of AI.

Acknowledgements

I would like to thank my supervisor, Mike Bain, for his supervision. I have learnt much about the topics included in this thesis due to Mike. Mike's ideas, assistance and feedback have been helpful and encouraging and this thesis did feel like a team effort. This thesis idea and topic is also due to Mike, so without him, this thesis would not exist.

I would also like to thank my assessor, Paul Compton, for his time, feedback and advice. Additionally, I would like to thank Paul for his fantastic textbook [1] which has taught me everything about RDR.

An additional thanks to M.T. Ribeiro, S. Singh and C. Guestrin for their paper [2] which explained Explainable AI and LIME.

Abbreviations

AI Artificial Intelligence

ML Machine learning

RDR Ripple Down Rules

SCRDR Single Classification Ripple Down Rules

MCRDR Multiple Classification Ripple Down Rules

GRDR General Ripple Down Rules

LIME Local Interpretable Model-agnostic Explanations

Contents

Table of Contents

List of Figures

Chapter 1

Introduction

There are an increasing number of real-world applications that use artificial intelligence (AI), many of which are based on machine learning (ML). An example of an industry with a rising usage of ML is the financial sector [4]. Decisions to approve or deny lending finance to businesses or individuals at financial institutions are increasingly being assisted or decided by complex ML algorithms. The ML algorithms used by financial institutions are “black boxes” which are generally too complicated to be understood by humans [5] [6]. However, in many real-world settings, the predictions of a machine learned model must be explainable on demand for legal or other reasons. This means that conclusion reached by the algorithm and the process used to reach that conclusion must be explained. However, as the machine learned model is too complex to be understood by humans, a human usually cannot explain its decisions. This has been a barrier to the adoption of ML in real world settings.

Explainable AI attempts to break this barrier that has held AI back by explaining complex machine learnt algorithms that cannot be understood or explained by humans. Explainable AI attempts to explain why and how conclusions are made by a black box model. It does this be creating an explainable model will explains the predictions of the “black box” classifier it is built to explain. The explainable model explains the black box decision by showing all the information used by the model to reach its conclusion – letting a human understand the reasons behind its conclusion.

This report will specifically outline LIME, which is an explainable AI framework [2]. LIME creates an interpretable model which can explain the predictions of any black box model, allowing humans to understand the reasons behind the classifications, no matter how complicated they are. This project will attempt to create an alternative explainable AI framework to LIME by using RDR.

Ripple down rules (RDR) is a knowledge acquisition system which involves incrementally building up the knowledge in the system. The system is built up by adding rules. This report will discuss different forms of RDR: mainly Single Classification RDR, with brief discussion on Multiple Classification and General Classification RDR [1].

In this thesis, we will explore the use of RDR in place of LIME as an explainability model. The ultimate goal of the project is to create an explainability model that is built using with rules using RDR.

There are several motivations behind using RDR for explainable AI. Firstly, RDR allows the capture of an expert humans understanding – due to it being a knowledge acquisition system. The expert human knows how to classify the cases but creating an explainability model from scratch is quite difficult. Using RDR allows the expert to deal with the cases in a case-by-case manner, which leads to the an explainability model being built up incrementally. This also allows the rules to be added in context, making the rules easier to understand for the expert user since they made them and therefore know the thinking behind them.

The advantage that RDR brings to explainable AI is that it contains an expert human in the loop. Due to the human in the loop creating the rules, the rules are naturally understandable to humans and therefore explainable. As the rules form the explainability model, this makes the explainability model easy to access and understand itself. This is different to other explainable AI frameworks such as LIME which rely on algorithmic techniques to explain "black box" models.

Lastly, this method is a "no code" way of building explainable AI. This method allows both programmers and non-programmers to create the rules and therefore an explainability model. This means that using RDR for explainable is a more accessible method of explainable AI, available to people whether they code or do not code. Experts in fields other than programming which are required for the system generally do not have programming knowledge. This system allows these non-programmer experts to use an explainable AI system. Also, an easy-to-use frontend UI can aid non-programmers even further to use a no-code system like the one proposed in this thesis.

In this thesis report, Chapter 1 provides a brief introduction to the motivations behind this project. Chapter 2 explains the background information required for this project, covering the motivation for, implementation, and applications of Explainable AI – including LIME, RDR and its multiple forms and also machine learning algorithms including decision trees, random forests and XGBoost. Chapter 3 gives a project online including the project hypothesis and method of implementation for the project including evaluation. Chapter 4 explains the implementation of the project plan for the thesis, including what tasks needed to be completed and how they were implemented. Chapter 5 is experimental evaluation which includes a demonstration of the use of the created system as well as various evaluation techniques including the use of a new dataset. Following this is the conclusion which includes further work.

Chapter 2

Background

In this chapter we will discuss the main background topics behind the thesis which are Explainable AI, RDR and machine learning algorithms.

2.1 Explainable AI

There is currently a trade-off between accuracy and interpretability in machine learning (ML) models. Accuracy in this context refers to how well a ML model predicts a class. Interpretability refers to the ability of humans to understand how a ML model works. However, both accuracy and interpretability are desirable attributes for a ML model. Simple ML models are interpretable but not accurate. The opposite is also true, that is, accurate ML models are generally too complicated to be interpreted by humans. This means for higher accuracy; we must use models which are not interpretable. The benefits of higher accuracy are obvious: the more accurate a ML model, the better. Maximizing the accuracy of ML models means that predictions are correct. But why is interpretability important?

An important reason for why interpretability of a ML model is for legal requirements. There may be cases where it must be known why or how a ML model reached its conclusion. This is a significant issue, since in May 2018 the General Data Protection

Regulation (GDPR) came into effect [7]. The GDPR is an EU privacy and security law that regulates the processing and storage of data from around the world, as long as it deals with data to do with the EU. It is the "toughest privacy and security law in the world" [7]. The GDPR stipulates that people have a "right to explanation". That means that AI and ML systems and their decisions must be explainable on demand. For example, a reason may be required from a person who has had a bank loan application was denied by a ML model or algorithm. Due to this right, ML models must be explainable but these models are too complex to be interpretated by humans.

Another reason is trust. Understanding the reasons behind predictions is important for trust. Trust is important if users are to take a prediction seriously or take action based on a prediction. For example, in medicine, trust is very important. Predictions cannot be acted on without an explanation as there may be catastrophic consequences. Also, understanding a model builds trust in that model. The issue of trust is very important – if users do not trust a model – they will not use it [2].

Explainable AI attempts to solve the trade-off between accuracy and interpretability in ML models. It does this by creating an interpretable model to explain a black box ML model, which would typically be expected to have high accuracy. This interpretable model allows us to see how/why a black box ML model reached its conclusion. LIME is an explanation technique which can explain the predictions of any classifier.

2.1.1 LIME

LIME (Local Interpretable Model-agnostic Explanations) is an explainable AI framework [2]. LIME creates an interpretable model that explains a black box ML model. The interpretable model it creates explains the conclusions of the black box model which allows humans to understand the decisions behind the model.

Figure 1 shown below is an example of LIME explaining a prediction for an image classification task:

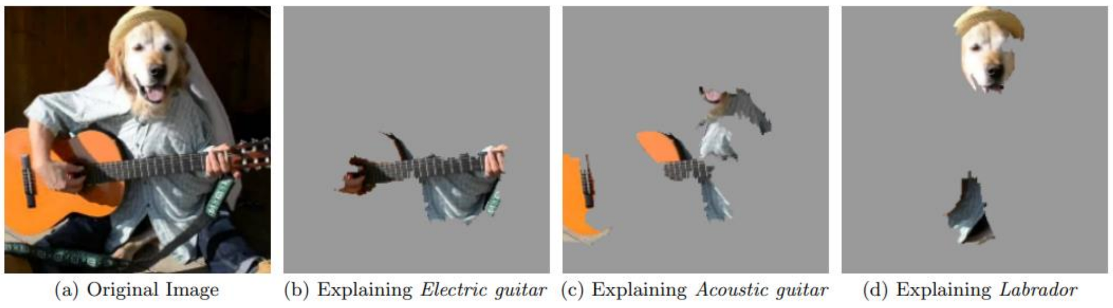

(a) Original Image
(b) Explaining *Electric guitar*
(c) Explaining *Acoustic guitar*
(d) Explaining *Labrador*

***Figure 1 -** Explaining an image classification prediction made by Google's Inception neural network [2]*

On the left of Figure 1 is the original image which has been classified by Google's Inception neural network. The top 3 predicted classes by the Inception neural network (what Inception predicts the image is) are: Electric guitar, Acoustic guitar, and Labrador. Images (b), (c) and (d) respectively show LIME's explanation for each of these classes. Each of these images shows the patches of the original image which caused the classifier to make its prediction. For example, in Image (c), we can see that mainly the orange parts of the original image caused the classifier to predict acoustic guitar. Similarly in Image (d), we can see that the classifier predicted Labrador mainly due to the parts of the original image which are a Labrador's head. Image (b) is particularly interesting. If not given the explanation, we would think: why did the classifier predict electric guitar for the original image? But having been given the explanation, we can see that the classifier predicted electric guitar because of the fretboard. These explanations give a clear insight into the reasons behind the classifications made by the neural network.

2.2 Ripple Down Rules

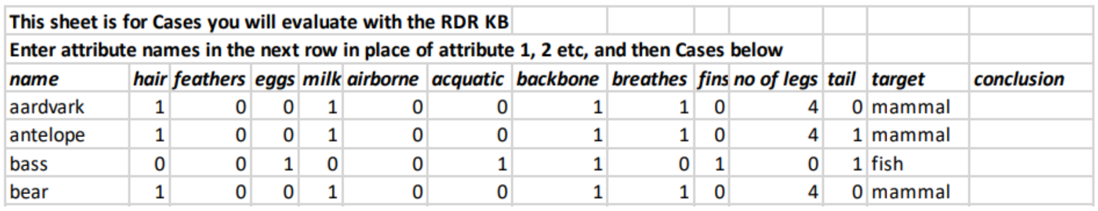
Ripple Down Rules (RDR) is a knowledge acquisition system which involves incrementally building up the knowledge in the system. The knowledge of the system is built up by adding rules. Every time the system does not deal with a case correctly – a new rule is added to correctly deal with that case. This allows the system to be built up incrementally while it is in use. There are multiple types of RDR. The ones that will be covered in this report are: single classification and multiple classification RDR.

There are many industry applications of RDR. Two of these are:

- Pacific Knowledge Systems (PKS) – uses MCRDR for interpreting laboratory data and auditing data entry to ensure requests are appropriate [1].
- IVIS – provides technology for multi-channel retailing. Uses rules to customize user experience

There have also been many research applications of RDR not yet been put to use in industry such as: RDR wrappers, text processing, natural language processing, conversational agents, anomaly and event detection, image and video processing [1]. However, creating Explainable AI with RDR is not mentioned – this is the gap in the literature that is the topic of this thesis.

The different types of RDR will demonstrated in this report by using the following Excel RDR example which comes from the *Ripple Down Rules: The alternative to machine learning (Compton & Kang, 2021)* textbook [1]. The example involves classifying a dataset of animals. A subset of the data which contains the cases to be classified is shown in Figure 2 below:



This sheet is for Cases you will evaluate with the RDR KB													
Enter attribute names in the next row in place of attribute 1, 2 etc, and then Cases below													
name	*hair*	*feathers*	*eggs*	*milk*	*airborne*	*acquatic*	*backbone*	*breathes*	*fins*	*no of legs*	*tail*	*target*	*conclusion*
aardvark	1	0	0	1	0	0	1	1	0	4	0	mammal	
antelope	1	0	0	1	0	0	1	1	0	4	1	mammal	
bass	0	0	1	0	0	1	1	0	1	0	1	fish	
bear	1	0	0	1	0	0	1	1	0	4	0	mammal	

Figure 2 *- a subset of the animal dataset [1]*

In Figure 2, we can see the cases to be classified. Each case is a separate row and each row represents a different animal. Each animal is represented as a set of attributes. All the attributes have boolean value (where 1 represents true and 0 represents false) except for “no of legs” which is a numerical count of the number of legs for the animal. For example, the aardvark, the first case shown in Figure 2, is shown to have hair, produce milk, have a backbone, breathe and has 4 legs. In Figure 2 we are shown 4 cases, but the complete dataset has 100 cases to classify.

2.2.1 Single Classification RDR

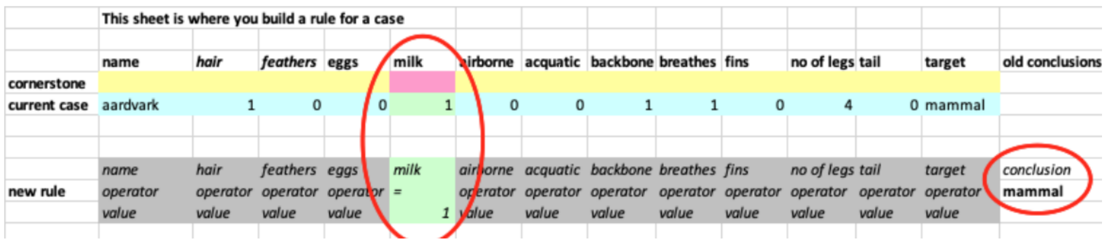
Single Classification RDR (SCRDR) is a form of RDR which allows only a single conclusion. For our animal example, this conclusion will be the species of the animal. We reach this conclusion by creating rules that involve the relevant animal attributes.

In Figure 2, the second last column, labelled “target”, shows the target species we want to conclude for each case. Our conclusion – which is the species that our rule concludes, goes in the conclusion column. If these 2 columns match for a given row, then the animal in that row has been correctly classified by the rules that are in the system.

<u>Creating a rule</u>

The first case – the aardvark, is a mammal. We need to create a rule for this case to classify it as there are no rules in the system yet. Figure 3 below shows the page where a rule is added.

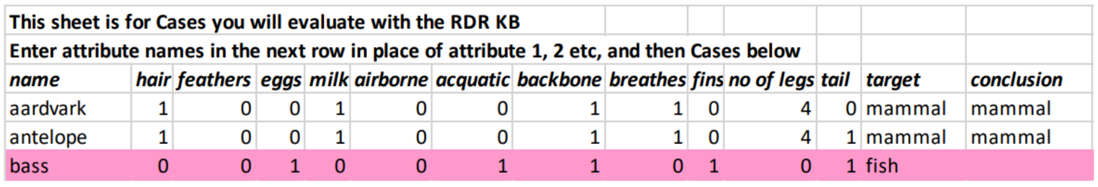
Mammals have multiple characteristics that define them, but the one we will use is milk – that is the production of milk. The rule we create is a simple if statement: if milk is 1, then our conclusion is mammal.



	This sheet is where you build a rule for a case													
	name	*hair*	*feathers*	eggs	milk	airborne	acquatic	backbone	breathes	fins	no of legs	tail	target	old conclusions
cornerstone														
current case	aardvark	1	0	0	1	0	0	1	1	0	4	0	mammal	
	name	*hair*	*feathers*	*eggs*	*milk*	*airborne*	*acquatic*	*backbone*	*breathes*	*fins*	*no of legs*	*tail*	*target*	*conclusion*
new rule	*operator*	*operator*	*operator*	*operator*	=	*operator*	*operator*	*operator*	*operator*	*operator*	*operator*	*operator*	*operator*	mammal
	value	*value*	*value*	*value*	1	*value*	*value*	*value*	*value*	*value*	*value*	*value*	*value*	

Figure 3 – Adding a rule to classify the aardvark as a mammal [1]

After this, we continue adding rules. For each animal that is not classified or misclassified, we make a new rule to correctly classify it.

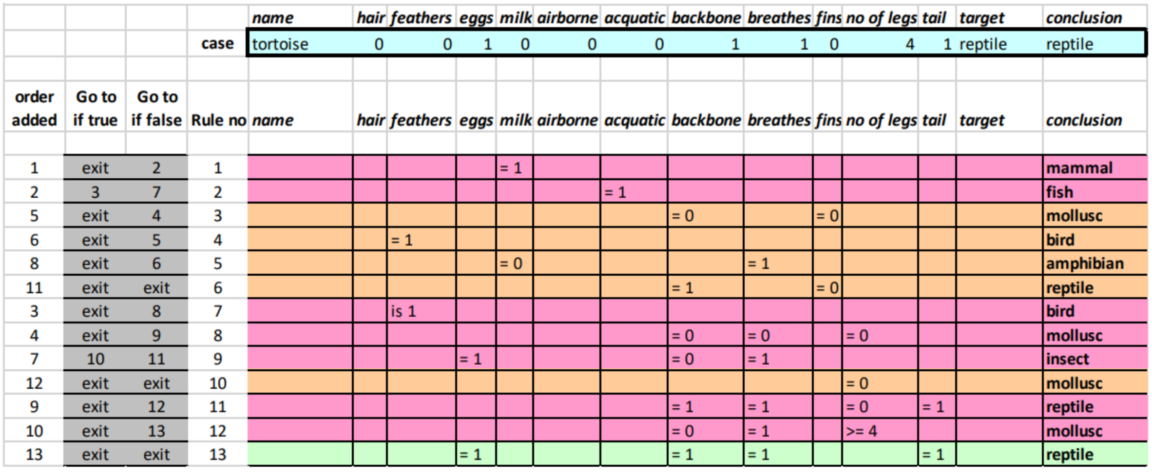


This sheet is for Cases you will evaluate with the RDR KB													
Enter attribute names in the next row in place of attribute 1, 2 etc, and then Cases below													
name	*hair*	*feathers*	*eggs*	*milk*	*airborne*	*acquatic*	*backbone*	*breathes*	*fins*	*no of legs*	*tail*	*target*	*conclusion*
aardvark	1	0	0	1	0	0	1	1	0	4	0	mammal	mammal
antelope	1	0	0	1	0	0	1	1	0	4	1	mammal	mammal
bass	0	0	1	0	0	1	1	0	1	0	1	fish	

***Figure 4 -** the result from selecting run till error after the rule in Figure 2 has been added [1]*

As we can see in Figure 4, the rule we created for the aardvark has been used to correctly classify it as a mammal. Beneath the row containing the aardvark is a row containing the antelope. The Antelope also has milk=1 as it is also a mammal. We can see that it too has been correctly classified as a mammal using the same rule (being milk = 1) which was created to classify the aardvark. However, the bass is not classified by this rule as it has milk = 0. Therefore, it currently has no classification as the rule used to classify the aardvark is the only rule in our system. To fix this we must make a new rule to classify the bass as a fish using characteristics that we think may define a fish, for example aquatic, or fins.

We continue through the cases until we have correctly classified all cases. Below in Figure 5, we have the complete knowledge base of rules added after we have gone through all cases.



				name	*hair*	*feathers*	*eggs*	*milk*	*airborne*	*acquatic*	*backbone*	*breathes*	*fins*	*no of legs*	*tail*	*target*	*conclusion*
			case	tortoise	0	0	1	0	0	0	1	1	0	4	1	reptile	reptile
order added	**Go to if true**	**Go to if false**	**Rule no**	*name*	*hair*	*feathers*	*eggs*	*milk*	*airborne*	*acquatic*	*backbone*	*breathes*	*fins*	*no of legs*	*tail*	*target*	*conclusion*
1	exit	2	1					= 1									**mammal**
2	3	7	2							= 1							**fish**
5	exit	4	3								= 0		= 0				**mollusc**
6	exit	5	4			= 1											**bird**
8	exit	6	5					= 0				= 1					**amphibian**
11	exit	exit	6								= 1		= 0				**reptile**
3	exit	8	7			is 1											**bird**
4	exit	9	8								= 0	= 0		= 0			**mollusc**
7	10	11	9				= 1				= 0	= 1					**insect**
12	exit	exit	10											= 0			**mollusc**
9	exit	12	11								= 1	= 1		= 0	= 1		**reptile**
10	exit	13	12								= 0	= 1		>= 4			**mollusc**
13	exit	exit	13				= 1				= 1	= 1			= 1		**reptile**

***Figure 5 -** the complete knowledge base [1]*

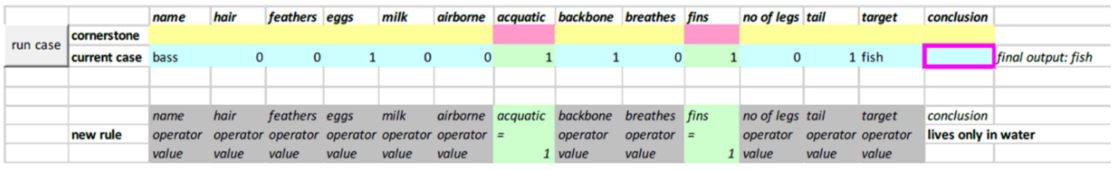
In Figure 5, we can see the case shown at the top of the image is a tortoise, which is a reptile.

The bottom part of the image is a rule trace. This shows the system using the rules we have created to attempt to classify the tortoise. On the left is the order we will evaluate the rules. For example, after the first rule is evaluated for this case, if the rule is true, we exit, if it is false, then we continue to rule 2 and so on.

It goes through the first rule – the rule we added earlier to classify the aardvark in Figure 3 – if milk = 1, our conclusion is mammal. That is not true so it is shown in red and we go to the next – also red. The rules coloured in orange mean they were not evaluated. This is because they are refinement rules of the second rule – the fish rule – all of which have aquatic = 1 – which is not true for the tortoise. We continue down the rules until we find a rule that is true and that classifies the tortoise as a reptile. This rule that is true is shown in green and the rule trace exits.

2.2.2 Multiple Classification RDR

Multiple Classification RDR (MCRDR) is very similar to single classification, except that now multiple conclusions are allowed. For example, in our animals' example – we now have multiple conclusions. Our conclusions will be the species of the animal as well as the habitat of the animal.



		name	*hair*	*feathers*	*eggs*	*milk*	*airborne*	*acquatic*	*backbone*	*breathes*	*fins*	*no of legs*	*tail*	*target*	*conclusion*	
run case	cornerstone															
	current case	bass	0	0	1	0	0	1	1	0	1	0	1	fish		*final output: fish*
		name	*hair*	*feathers*	*eggs*	*milk*	*airborne*	*acquatic*	*backbone*	*breathes*	*fins*	*no of legs*	*tail*	*target*	*conclusion*	
	new rule	*operator*	*operator*	*operator*	*operator*	*operator*	*operator*	=	*operator*	*operator*	=	*operator*	*operator*	*operator*	lives only in water	
		value	*value*	*value*	*value*	*value*	*value*	1	*value*	*value*	1	*value*	*value*	*value*		

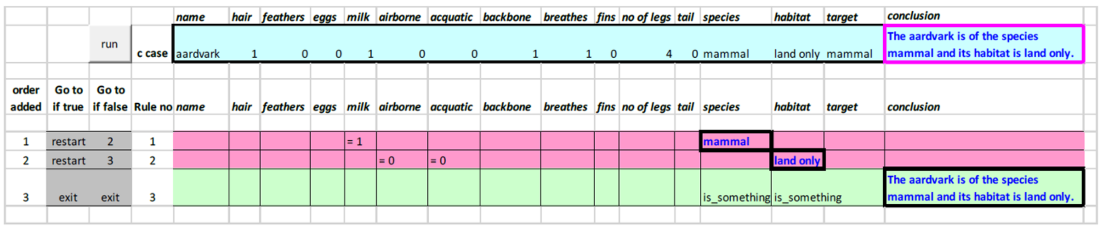
Figure 6 - *adding a rule to give a second conclusion for habitat for animals that live only in water [1]*

In Figure 6 shown above, we can see the case of the bass. At this stage we already have rule that classifies it as a fish. But we also want to have its habitat in the conclusion. So, we create a rule to conclude that if aquatic and fins are true, then we give the conclusion "lives only in water". Now for the bass, 2 rules will be true – the rule that classifies it as a fish and this rule here. These conclusions will both appear in final output.

2.2.3 General RDR

General RDR (GRDR) can do more than just classification – though it can still be used for classification. An important feature of General Classification is repeat inference. This means that the knowledge base is traversed potentially multiple times with facts being asserted and retracted each time the knowledge base is traversed.

General RDR will be demonstrated in this report using the animal example shown previously in 2.2.1 and 2.2.2. Below in Figure 7 is an example of GRDR in action:



				name	*hair*	*feathers*	*eggs*	*milk*	*airborne*	*acquatic*	*backbone*	*breathes*	*fins*	*no of legs*	*tail*	*species*	*habitat*	*target*	*conclusion*
		run	c case	aardvark	1	0	0	1	0	0	1	1	0	4	0	mammal	land only	mammal	The aardvark is of the species mammal and its habitat is land only.
order added	**Go to if true**	**Go to if false**	**Rule no**	*name*	*hair*	*feathers*	*eggs*	*milk*	*airborne*	*acquatic*	*backbone*	*breathes*	*fins*	*no of legs*	*tail*	*species*	*habitat*	*target*	*conclusion*
1	restart	2	1					= 1								mammal			
2	restart	3	2						= 0	= 0							land only		
3	exit	exit	3													is_something	is_something		The aardvark is of the species mammal and its habitat is land only.

Figure 7 - the final conclusion after the knowledge base is traversed three times [1]

In this case, the aardvark is being classified. For now, ignore the red and green colours in the rule trace. The first time going through the rule trace, we see the familiar rule: if milk = 1, species = mammal. We created this exact rule in 2.2.1. As with SCRDR, this rule is true for the aardvark since it has milk = 1 so we have the conclusion species = mammal. Now the difference with GRDR is shown on the left side in grey. It shows that if True, restart. Now, we restart with knowledge that Rule 1 is true. In our second traversal of the knowledge base, since Rule 1 is true, we skip it and go straight to rule 2. Now rule 2 is also true for the aardvark – as it has both airborne and aquatic = 0 – so we have another conclusion that habitat = land only. Now on our third traversal of the knowledge base, both rule 1 and rule 2 are true so we go straight to rule 3. Rule 3 is very simple and is: if species and habitat = is_something, then conclusion. This is true for both because of the 2 rules above which we both know are true. This gives us a final conclusion which uses the cells in the spreadsheet to construct a sentence about the animal and its species and habitat.

2.3 Machine learning algorithms

A definition of machine learning by Professor Tom Mitchell in his book "Machine Learning" is: "The field of machine learning is concerned with the question of how to construct computer programs that automatically improve from experience." [8] This project will focus on classification problems which involve training a classifier, which is a function mapping from an input data to one of a set of discrete outputs (classes). The more data that the classifier is trained on, the higher accuracy that the classifier can predict the labels of future data. The process of machine learning is that a model is created and then trained on a portion of a dataset called the "training set". After the model is trained, it is tested on the portion of the same dataset that not used for training the model, called the "test set". The percentage of cases in the test set that the model predicts correctly determines the model's accuracy. A popular evaluation of machine learning models is called k-fold cross validation. In k-fold cross validation, the splitting of the dataset and training as just described above is done k times and the total accuracy is the average of the k accuracy values returned. For this project, 3 different machine learning algorithms are used: decision trees, random forests and XGBoost.

2.3.1 Decision trees

A decision tree is a tree of decision nodes where cases are split at each decision node by testing a feature. The result of the test on a feature(s) for a particular will determine which branch of the decision tree is taken next. Once each particular case reaches a leaf node of the tree, a classification is assigned that that case by the decision tree. Decision trees are very popular machine learning tools because, they are very simple to understand and use.

2.3.2 Random forests

Random forests are an ensemble learning method which are created by randomised bagging of decision trees. Ensemble learning methods combine multiple machine learning models together by combining their predictions in some way, normally by using averaging or weighted voting. Ensemble methods are popular because using multiple methods can (and often do) give a higher accuracy than single models. Bagging (or Bootstrap aggregation) uses the simplest way of combining predictions – by calculating the average prediction of all models (every model receives equal weight). When decision trees are bagged, the result is a "forest" of trees – or just a forest. If randomness is introduced into the forest learning algorithm to provide some variety into the ensemble, then the forest is now called a random forest.

2.3.3 XGBoost

eXtreme Gradient Boosting (XGBoost) is an optimised, distributed gradient boosting library [9]. It is a machine learning algorithm which is an implementation of gradient boosted decision trees. boosting is similar to bagging mentioned above for random forests, but instead of taking an average for all models (meaning all models have equal weight), each model is weighted according to its performance. It uses an iterative learning procedure where new models are influenced by the performance of the models that come before it. The weighted These trees are designed for maximum speed and performance.

Chapter 3

Project Outline

Outlined below is an outline of the project including a high-level algorithm followed by the implementation and evaluation methods used.

3.1 Hypothesis

Project hypothesis: RDR can be used in place of LIME to explain predictions of a Black Box ML model.

The aim of this project was to create a system where an explainability model can be build using RDR to explain a "black box" classifier. The form of RDR to be used for the project is SCRDR. SCRDR is sufficient for the project as only a single conclusion is necessary. The ruleset created by the RDR system will form the explainability model. An expert human-in-the-loop will create the rules using the RDR system based of the predictions of the "black box" classifier. To achieve this, an RDR system must be combined with a "black box" classifier to form a single, cohesive system where the expert human user can create rules based off the predictions of the "black box" classifier.

3.1.2 High Level Algorithm

For each case, the case is be run through the "black box" classifier which give its classification. Next, the case is shown to the expert human-in-the-loop. The classification of the case by the "black box" model is seen by the expert and the case is run through the

RDR explainability model. If the RDR system's prediction for the case matches the "black box" prediction, no action is taken as the RDR has made a correct classification. However, if the RDR system's classification does not match, the expert user creates a new rule to correctly classify it by selecting relevant attributes. Importantly, selection of these attributes is under the control of the expert user. That is, they will only define a rule for a classification in terms of combinations of attributes that make sense to them. It follows that a knowledge-base constructed in this way will be an explainable model of the predictions of the black-box model on the test set of cases. Figure 8 shows a diagram of the algorithm:

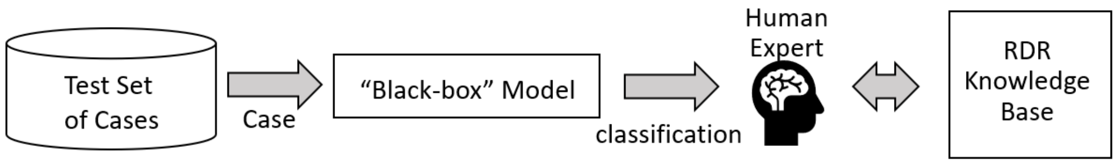


***Figure 8** – high-level algorithm*

3.2 Method

The process to complete the project was as follows:

1. **Proof of concept** – build explainability model using SCRDR which explains decisions of black box classifier
2. **Add proper "black box" classifiers** – add an actual "black box" classifier so the predictions of the "black box" can be explained by the RDR rules created by the user.
3. **Hints** – the user will be given hints in order to help them make rules. These hints will include information of what features are important, and therefore should have rules made in relation to them.

The implementations of these tasks are described in chapter 4 of this report.

3.2.1 Proof of Concept

To establish the proof of concept, the RDR system was used to attempt to explain the decisions of the black box classifier which will be using known data. Known data was used to check that the explainability model was working correctly and as intended. That data was the animal's dataset allows any person to act as the expert in the system as there is no specialized knowledge needed to classify the species of animals. During this step, a pseudo "black box" classifier was used to ensure it could be understood what it was doing to make sure everything was working correctly.

3.2.2 Using a proper black box classifier and a new dataset

After the proof of concept was established, the RDR explainability model and black box classifier system was extended to implement proper "black box" algorithms as well as a new dataset. A limitation of this project was that we can only use data from domains where a normal person can act as an expert or given enough information to be an expert. However, after testing and using various datasets, the titanic dataset was selected.

3.2.3 Hints

The various "black box" models give hints to the user on what features of a dataset are important by looking at what attributes each classifier uses to reach its conclusions. Even "black-box" models can provide the user with some information on how the model was learned. For example, some ML algorithms return measures on the relative "importance" of the attributes used in training the model and these give a partial, although incomplete picture of what the classifier is doing.

3.3 Evaluation

The evaluation phase of the project consists of various evaluations of the RDR system. These include:

- Using new datasets
- Comparing the number of rules vs number of decision nodes in a decision tree
- Learning curve evaluation

Chapter 4

Implementation

4.1 Proof of concept

The proof of concept involved creating the system while using a decision tree as a pseudo "black box" classifier to ensure that the system was working correctly. Involved in the proof of concept was to explore the tasks required to implement the system and to test and choose the most appropriate tool(s) for each task. Throughout the proof of concept and implementation phase, the animals dataset was used as it is a simple dataset that allowed easy creation of rules as a normal person could act as the expert human in the loop.

The goal for the proof of concept was to lay the foundations of the system in place: data manipulation, RDR rule creation and logic, and simple ML implementation. After completion of the proof of concept, additional features and functionality were easily added on.

To create the system as it was outlined in chapter 3, the first task was to decide on what components of the system were necessary. The following components were identified as needed for the implementation:

- programming language
- data manipulation

- rules data structure
- machine learning

The following subchapters cover the design designs for each of these components and provide justification for their selection. The source code has been uploaded to GitHub and the URL can be found in the bibliography [3].

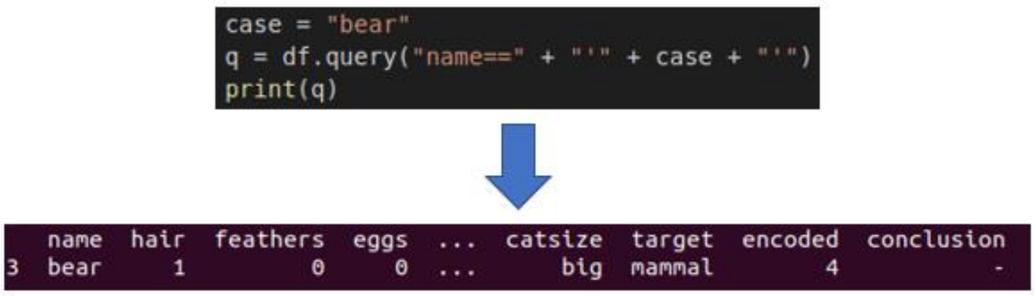
4.1.1 Programming language – Python

Python is a general-purpose programming language – meaning it can be used for a wide variety of development tasks. Python can be used for these tasks because of the large number of libraries and frameworks that can be imported into a Python program. These libraries can be used to assist with tasks like data analysis and machine learning. In this project, a rules data structure, machine learning and handling of data are all tasks that need to be completed. As Python offers many libraries to assist or fully implement these tasks, python was a suitable choice for programming language. Python is also a simple language which is easy to understand and code in. Furthermore, I personally have used Python in the past extensively, so I have knowledge of the language and some of its libraries. All of these reasons led to Python being chosen as the programming language for this project.

An alternative considered was the R programming language. R is designed for statistical and data analysis making it a good choice for handling data. Though a weakness is that R is a complicated language that is hard to use and understand. This means that programming with R would be a difficult task, so implementing the rules data structure and rules application using R may have been challenging. R is also a language I have not used before, so time would have needed to be spent on learning it, whereas this was not the case with Python

4.1.2 Data manipulation – Pandas

Following the selection of python, the next task was to explore methods to store and manipulate the data as needed for this project. The Python library Pandas was able to handle all the data manipulation related tasks that were necessary for the project quite easily. Data can be simply read in from a comma separated vales (CSV) file into a dataframe with a single command e.g., df = read_csv("animals.csv"). A dataframe is a table containing rows and columns of data which can be modified in many ways such as adding/deleting new rows/columns and modifying cells. Therefore, the cases can be read into a dataframe from a csv file and subsequently can be manipulated as needed. Importantly, Panda's Dataframes can also be queried, using queries similar to database queries. This querying functionality is needed in order to apply the rules created by the user to the cases in the dataframe. The simplicity of querying in Pandas is shown in figure 9 below.



```
case = "bear"
q = df.query("name==" + "'" + case + "'")
print(q)
```

```
   name  hair  feathers  eggs  ...  catsize  target  encoded  conclusion
3  bear     1         0     0  ...      big  mammal        4           -
```

Figure 9 *- a Pandas query*

In figure 9, the query function is called on the dataframe of cases (df) and argument to query is "name=='bear'". This means that the query attempts to find all rows where the name column is equal to the string "bear". In this example, a single row is returned which contains the row corresponding to the bear.

When exploring data manipulation and storage methods, other tools were considered. A method that was tested was python's built-in lists and dictionaries. When testing these tools, they were found to usable for the project, but compared to the Panda's dataframe, they required a great deal of additional work for the same functionality. This additional code required for the lists and dictionaries made them hard to use and not easy to understand when using it for the data manipulation tasks required for this project.

4.1.3 Rules data structure – Python class instances stored in a list

Formulating the rules data structure was one of the most important steps in the development of the system. This is because the rules data structure lies at the centre of the system, meaning that all other systems build on top of it. However, at the beginning of the project, design choices were not obvious for the rules data structure.

The first method that was investigated were tree data structure implementations in Python. Tree implementations were considered as there was a possibility that the rules could be stored in a tree and a tree traversal algorithm such as depth first search (DFS) could be used to traverse the rules. However, there was no in-built implementation of trees in Python. Therefore, the next step was then to create a tree implementation using python's classes and this became the final solution. Using this method, using a python class to store each rule became the final solution. While a tree implementation led to a python class being used as the rules data structure, the rules were not stored in a tree.

The resulting class, called "Node", stores all the information required for a single rule, much like a struct in C. The information required for each rule is:

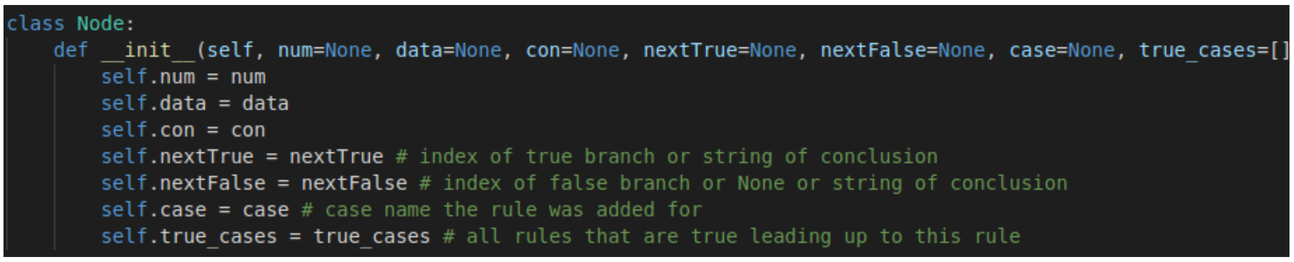
- **Number –** each rule has a number, allowing rules to refer to other rules. It also allows the system and user to identify a rule. Rule numbers start at 1
- **Data –** contains the condition of the rule. For example, if the rule is: IF milk == 1 THEN mammal, then the data portion is milk == 1
- **Conclusion –** contains the conclusion of the rule. For example, if the rule is: IF milk == 1 THEN mammal, then the conclusion portion is mammal
- **Next True –** is the index (an integer) of the next rule which is to be evaluated if the current rule is true OR a string containing the conclusion if the current rule is a leaf rule (there are no rules to evaluate after the current rule if it is true) and the current rule is false. This cannot be none as a rule must have a conclusion. As the rules are stored in a Python list, the Next True rule can be accessed by indexing the rules_list using the Next True (or Next False) value.
- **Next False –** is the index (an integer) of the next rule which is to be evaluated if the current rule is false OR a string containing the conclusion if the current rule is a leaf rule (there are no rules to evaluate after the current rule if it is true) and the current rule is false OR none if there are no rules or

conclusion following the current rule if it is false. This is allowed to be none as a none will follow at the end of the last tule if they are all false.

- **Case –** is the case name of the case which required a new rule to be added (as the current rules did not deal with this case correctly according to the black box classifier)
- **True cases –** is a list of rules that are true leading up to the current rule. This is used to create a list of features that should be used to make a refinement rule on the current rule by listing the features that are different in both the current case and all the true cases

Instead of creating a tree implementation as planned, each rule instance was initially stored in a linked list (hence the class name node), but the final data structure chosen to store all the rules was python's version of arrays which is a python list. A list allows rules to be indexed which makes it easier to access specific rules as opposed to iterating through a tree every time a rule needs to be accessed. This means that tasks such as printing can be performed by simply iterating through the rules list.

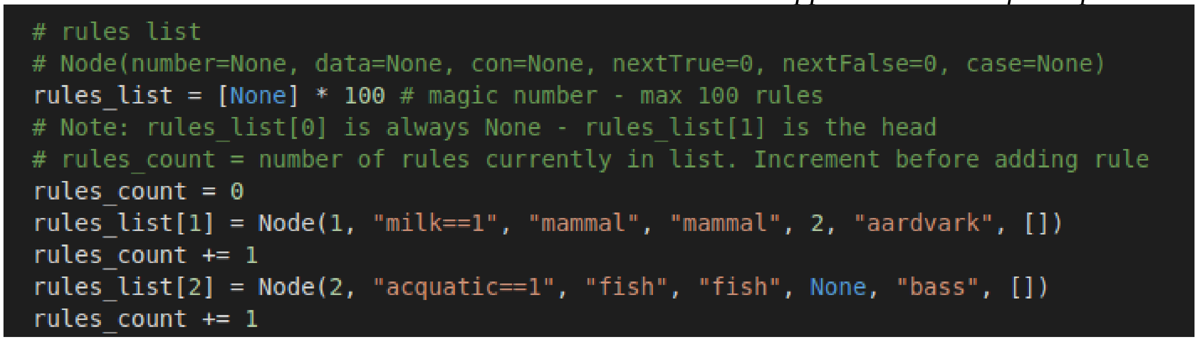
In Figure 10 shown below, the implementation of the python class "Node" – the class which stores a single rule is displayed. The class "Node" has a single constructor, the "__init__" method which is used to construct an instance of "Node" when a new rule is made.



```
class Node:
    def __init__(self, num=None, data=None, con=None, nextTrue=None, nextFalse=None, case=None, true_cases=[]
        self.num = num
        self.data = data
        self.con = con
        self.nextTrue = nextTrue # index of true branch or string of conclusion
        self.nextFalse = nextFalse # index of false branch or None or string of conclusion
        self.case = case # case name the rule was added for
        self.true_cases = true_cases # all rules that are true leading up to this rule
```

***Figure 10** - python class "Node"*

The image below in Figure 11 shows the list used to store the rules called "rules_list" – note the simplicity of this storage solution. A count of the number of rules called "rules_count" is stored to so the system knows the number of rules in the "rules_list". This helps the creation of new rules by storing the index a new rule will be made at. In the image, there are two hardcoded examples shown of the construction of a rule.



```
# rules list
# Node(number=None, data=None, con=None, nextTrue=0, nextFalse=0, case=None)
rules_list = [None] * 100 # magic number - max 100 rules
# Note: rules_list[0] is always None - rules_list[1] is the head
# rules_count = number of rules currently in list. Increment before adding rule
rules_count = 0
rules_list[1] = Node(1, "milk==1", "mammal", "mammal", 2, "aardvark", [])
rules_count += 1
rules_list[2] = Node(2, "acquatic==1", "fish", "fish", None, "bass", [])
rules_count += 1
```

***Figure 11** - the "rules_list"*

To summarise the rules data structure, each time a new rule is created, the rule and all its information is stored in an instance of a python class called "Node". Each instance of a rule is stored in a python list called "rules_list" with a counter called "rules_count" keeping track of the number of rules.

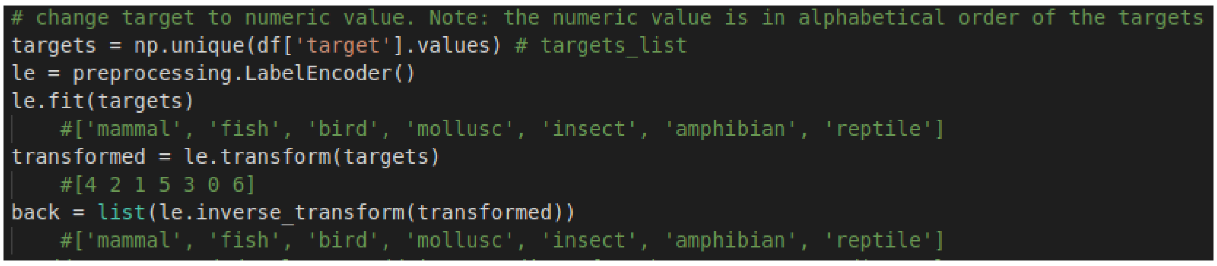
4.1.4 Machine learning ("black box" classifiers) – Scikit learn (sklearn)

To implement the "black box" classifiers which are required inside the system, the Python library Scikit-learn (sklearn) was used [10]. Sklearn is a popular and easy to use machine learning Python library that offers implementations for many different machine learning models. Its implementations provide methods such as "fit" to train the classifier and "predict" to give a classification for a given case. These methods make integrating the machine learning in the system straightforward. Importantly it offers implementations of all machine learning algorithms required for this project: decision trees, random forests and XGBoost. Sklearn additionally offers tools to help with machine learning tasks such as cross validation and label encoding which will be discussed further below. Sklearn also takes in a Pandas dataframe as its data inputs which means it works seamlessly with the Pandas used in this project for data storage and manipulation.

As discussed earlier in the thesis plan, a pseudo "black box" classifier was used during the proof-of-concept stage of the project in the form of a decision tree. A pseudo "black box" was used to ensure all components were working correctly before moving on to actual "black box" classifiers and new datasets.

Creating the decision tree model using sklearn was very simple. The decision tree was instantiated with default parameters.

The main issue encountered with the sklearn and its decision tree implementation (as well as its implementation of random forests and XGBoost) was that it does not accept numeric data. This was a problem because in the animal dataset, the species value (the target value) is a string. The first attempted solution was to encode the species string as an int, but this did not work as the resulting encoded int ended up being too large. The final solution was a tool from sklearn called label encoder [11]. Label encoder very easily allowed the species string to be encoded as a number by using a method called "transform". It also provided a reversing function of the encoding to get the species string back from the number called "inverse_transform". In Figure 12 below you can see how it label encoder works.



```
# change target to numeric value. Note: the numeric value is in alphabetical order of the targets
targets = np.unique(df['target'].values) # targets_list
le = preprocessing.LabelEncoder()
le.fit(targets)
    #['mammal', 'fish', 'bird', 'mollusc', 'insect', 'amphibian', 'reptile']
transformed = le.transform(targets)
    #[4 2 1 5 3 0 6]
back = list(le.inverse_transform(transformed))
    #['mammal', 'fish', 'bird', 'mollusc', 'insect', 'amphibian', 'reptile']
```

Figure 12 *- using label encoder. The commented lines show what the values would be if they were printed*

Figure 12 firstly shows that the targets are taken as a list of species (in string form) and "le" is created as the label encoder. We fit "le" to the targets and then use the function "transform" to convert the list of strings into a list of integers. Lastly, the "inverse_transform" function converts these integers back to the list of strings we had originally.

For the proper "black box" implementations to be used later in the project, these same data conversions can be used.

In Figure 13 is an image of a decision tree that was fitted on the animals dataset. This image was made using a tool provided by sklearn called "export_graphviz" [12]. Every time the program is run, a new image of the training decision tree is saved.

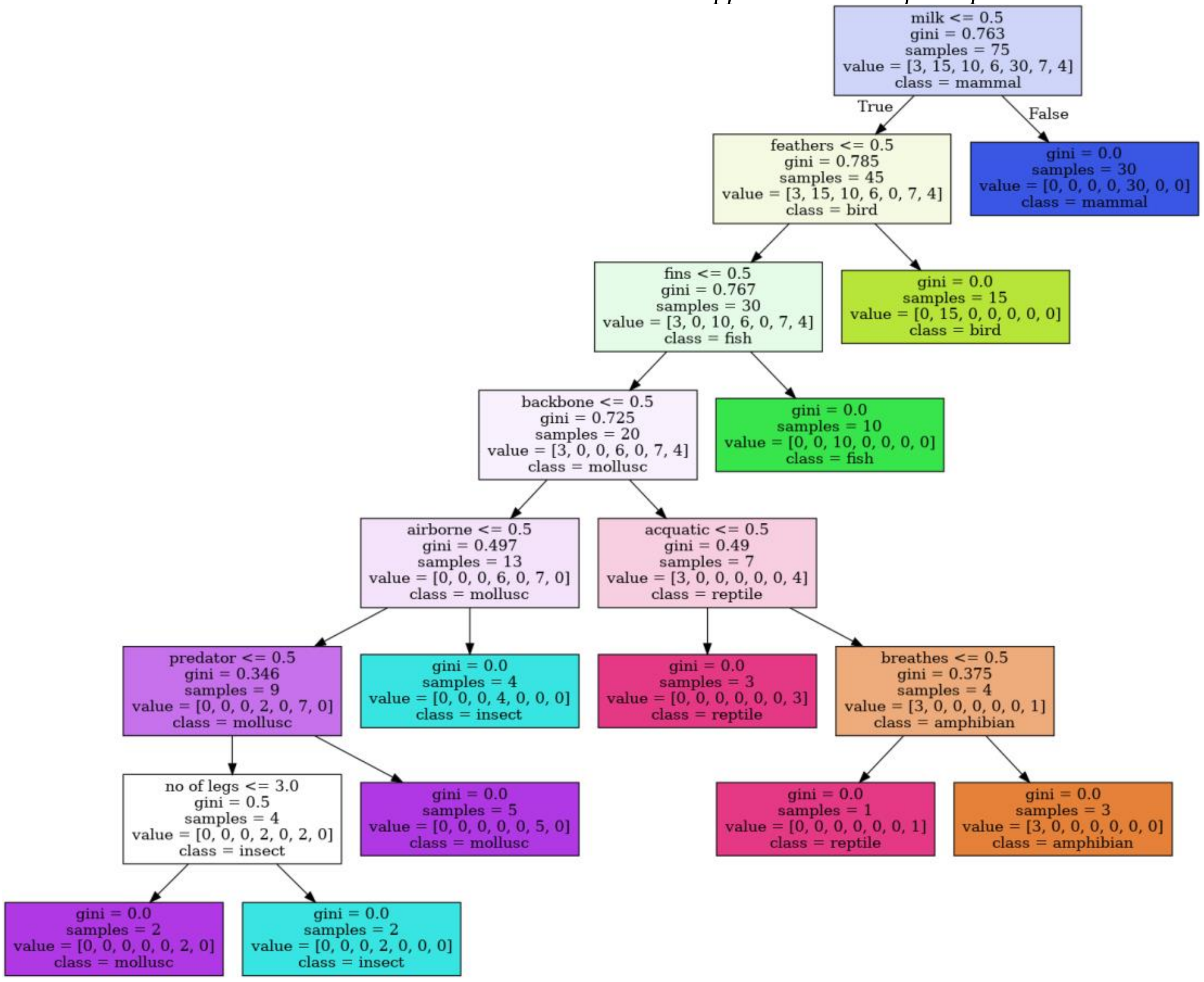


***Figure 13** - decision tree fitted on animals dataset*

4.1.5 Implementation

After selecting on the most appropriate tools for the tasks to be completed, next was to use those tools in the system’s implementation. It was decided that this program would use a command line interface where the user is presented with options they may take.

The following system architecture diagram includes the chosen technologies in the previously established high-level algorithm (note that the dataset in a csv file is imported into the dataframe):

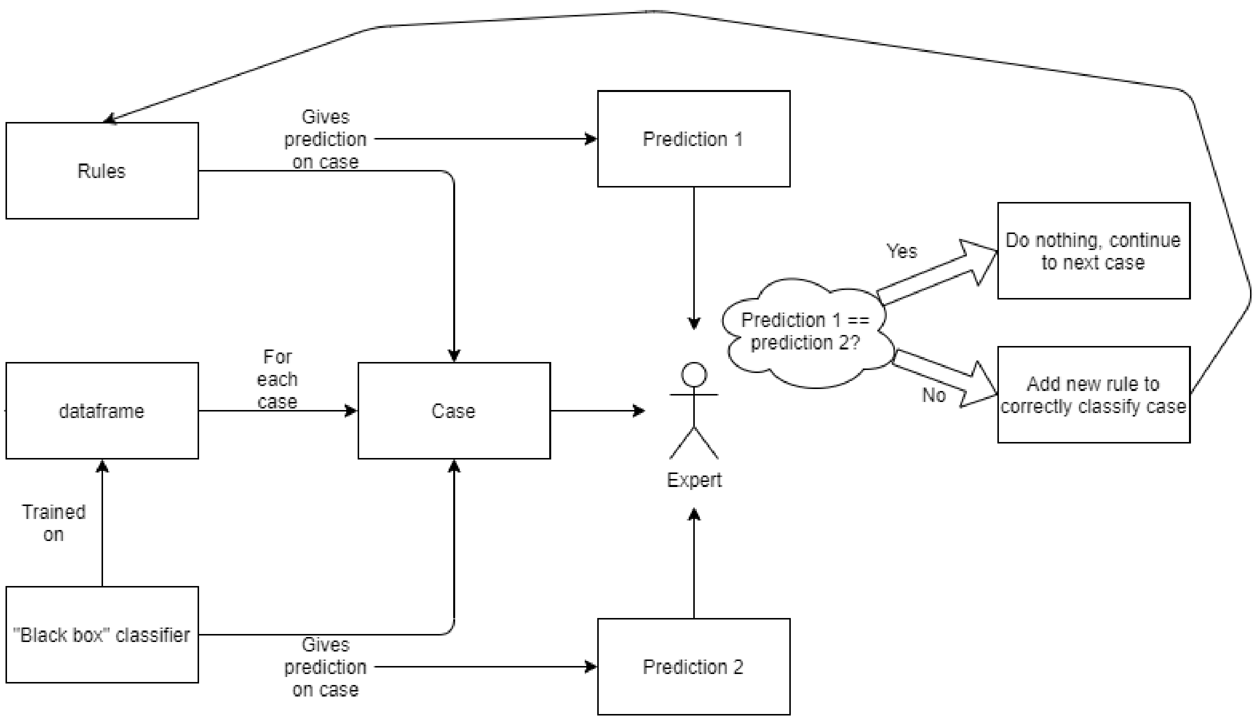


***Figure 14** - system architecture diagram*

Before being used in the system, the animals dataset had to undergo slight modifications, e.g., removing all columns with string values because as mentioned earlier, sklearn does not work with non-numeric data. Firstly, Pandas is used to read the csv file into a dataframe. After this the dataframe undergoes slight manipulation: adding an empty 'conclusion' column where the RDR system's classification will eventually be stored, as well as an 'encoded' column which stored the label encoder's encoding of the target strings. The next task was to add the ability for the user to add rules. Cornerstone rules are to be added when a case has no conclusion and refinement rules are required when a wrong conclusion is given for a case. These two different rules require similar but not identical implementation. The machine learning was then added within this system as the "black box" that the user attempts to make rules on to explain. To make use of the system as simple as possible, all these features were put into a single function which follows the high-level algorithm shown earlier.

Some issues faced: the rules data structure caused some problems as it sits at the core of the program. Any changes made to the rules data structure needed to be made in many places. As it continually evolved during development (from a tree to a linked list to a list) and increased in size as more data needed to be stored (adding fields for case and true cases), it caused edits throughout the program.

Following this, the implementation of the proof of concept was complete. The rules form an explainable model which explain the pseudo "black box" classifier. Figure 15 below shows the menu screen presented when the Python program was run after the proof of concept was completed.

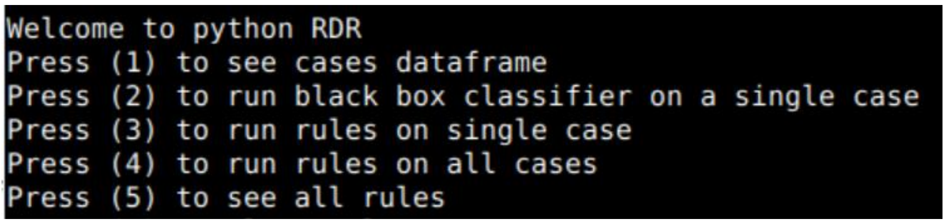

```
Welcome to python RDR
Press (1) to see cases dataframe
Press (2) to run black box classifier on a single case
Press (3) to run rules on single case
Press (4) to run rules on all cases
Press (5) to see all rules
```

Figure 15 *- option screen presented after the completion of the proof of concept*

The options and their actions are explained below:

1. Show the dataframe which contains all the data of the dataset. The RDR's classification is also shown in the conclusion column if the case has been classified
2. Run the "black box" classifier on a case to be inputted by the user. This returns a string – the prediction of the "black box" classifier
3. Run the RDR ruleset on a case to be inputted by the user
4. Run the RDR ruleset on all cases. If a case is not classified or classified incorrectly, the user is prompted to add a new rule to correctly classify the case. This option is the main option which follows the high-level algorithm and allows the development of the RDR ruleset. When a new rule is added, the user is taken back to the menu screen where they can press 4 again to continue the rule adding process
5. Show all the rules in the RDR ruleset that have been added by the user

A full walkthrough of the system is demonstrated in chapter 5.

4.2 Implementing proper "black box" classifiers

Now that the proof of concept was complete, the next task was the implementation of proper "black box" classifiers. Two classifiers were used for this part of the project:

Random Forests and XGBoost. Now when entering the program, the user is prompted to choose one of the three classifiers to be used as the “black box”. This selection can be changed at any time as all classifiers are trained when the program starts regardless of the user’s selection.

4.2.1 Random Forests

The sklearn implementation of random forests was used for this project [13]. The random forest was initialised with default parameters (100 trees).

4.2.2 XGBoost

The second “black box” algorithm used was XGBoost. The implementation of XGBoost provided by sklearn was used for the project [14]. XGBoost takes in a dmatrix which is a matrix specifically designed to be used with XGBoost. This is different to the decision tree and random forest which both take in a dataframe. This required the data to be transformed from the dataframe into a dmatrix. The XGBoost model was initialised with default parameters including 10 rounds.

4.3 Hints

An important idea of the project is that the expert user may need help deciding what features to make rules on. To help out the expert user in making rules, the system will give hints by showing, using various means, what features are or may be important for either the entire dataset or a particular case according to each particular machine learning classifier.

4.3.1 Decision tree

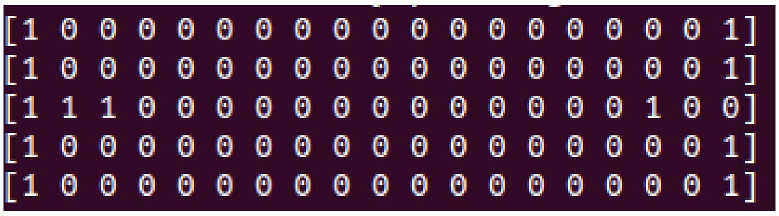
The user could potentially look at the attributes used in the decision nodes of the tree (see Figure 13) and use this information to assist them in creating a new rule. This option can only be used if the user understands the created decision tree, which is not always the case.

4.3.2 Decision path

A tool that can be used to give hints on what features are important in a dataset was decision path. Decision path is a method provided by sklearn that is applied to a decision tree [10]. It shows which features were used by the decision tree as a decision node when making a classification. A user can use the features showed by this tool to make a rule on.

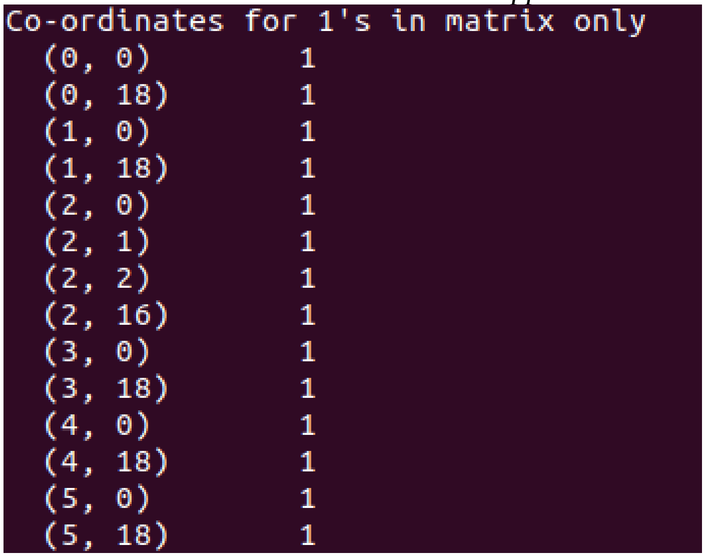
In the system, decision path gives the user is given a trace through the decision tree of what nodes were used by the decision tree to reach its conclusion for a particular case. This shows the features used by the decision tree to reach its conclusion. These features may or may not be useful to create a rule, the expert user can interpret the decision path given as appropriate and choose the important feature(s).

The decision path method from sklearn for a decision tree returns a matrix. For index (I, J) in the matrix, I indicates the case index and j indicates the node number. If the value at (I, J) in the matrix == 1, then decision node J of the decision tree was used by the decision tree when the case I was run through the decision tree to give a conclusion. If the value at (I, J) == 0 then decision node J was not used when case I was classified by the tree. Below in Figure 16 is the first 5 rows of the matrix for a decision tree trained on the animals dataset while Figure 17 shows the co-ordinates of all 1's in the first 5 rows of Figure 16.



```
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
[1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
```

***Figure 16** – the matrix returned by decision path*

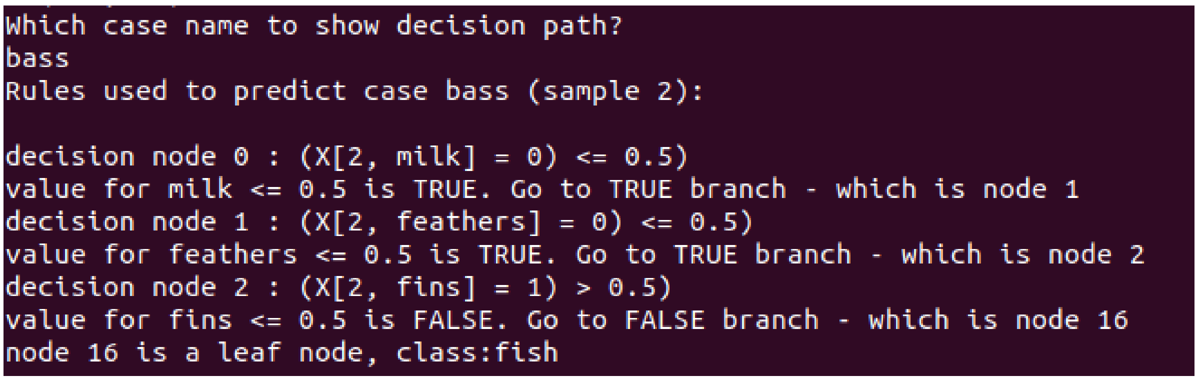


```
Co-ordinates for 1's in matrix only
  (0, 0)        1
  (0, 18)       1
  (1, 0)        1
  (1, 18)       1
  (2, 0)        1
  (2, 1)        1
  (2, 2)        1
  (2, 16)       1
  (3, 0)        1
  (3, 18)       1
  (4, 0)        1
  (4, 18)       1
  (5, 0)        1
  (5, 18)       1
```

Figure 17 *- printing the co-ordinates of only the 1's in the matrix in figure 16*

Taking the first row of Figure 16 (row 0 – which is the case "aardvark"), this case uses nodes 0 and 18 in the decision tree (as shown in Figure 17). Going back to the decision tree in Figure 13, node 0 is the first node in the decision tree (the light blue node) which splits the data on the rule milk <= 0.5. Node 17 node is the dark blue node which returns classification mammal. However, there the user does not know all this information. That is the motivation behind the decision path method in the system – to convey the information provided in this matrix in a way that can be understood.

Therefore, to help the user, the program takes the matrix given by decision path and prints out the actual decision path that the decision tree took, conveying the information provided in the matrix in a way that can be understood. Figure 18 shows the decision path for the case "bass" of the animals dataset.



```
Which case name to show decision path?
bass
Rules used to predict case bass (sample 2):

decision node 0 : (X[2, milk] = 0) <= 0.5)
value for milk <= 0.5 is TRUE. Go to TRUE branch - which is node 1
decision node 1 : (X[2, feathers] = 0) <= 0.5)
value for feathers <= 0.5 is TRUE. Go to TRUE branch - which is node 2
decision node 2 : (X[2, fins] = 1) > 0.5)
value for fins <= 0.5 is FALSE. Go to FALSE branch - which is node 16
node 16 is a leaf node, class:fish
```

Figure 18 *- decision path as implemented in the system*

Firstly, the user inputs a case which is “bass” for this example and the resulting decision path is shown. The first step is to check the value at row 2 (the row containing information about the bass) and column “milk” (this is X[2, milk]). The “milk” column is checked because the first decision node in the decision tree is “milk <= 0.5”. This value is 0 for the bass, which is less than 0.5. Therefore, on the next line, it is clear that the value for the first decision node “milk <= 0.5” is TRUE for the bass, and the TRUE branch is taken as a result. We then repeat this process for 2 more decision nodes until a leaf node is reached which returns a conclusion, classifying the bass as a fish. The output shown here for the decision tree classification of the bass can be followed in the decision tree in Figure 13.

This implementation of the decision path, while more useful that a full matrix, is still very hard to understand. The reason it is hard to understand is because the decision tree is actually quite hard to understand.

This leads into an argument about the explainability of decision trees vs RDR rules. The claim that decision trees are interpretable depend on both the implementation of the decision tree as well as the user perspective. The general perception of decision trees is that they are interpretable. However, this is not always the case. The tree learner is a simple algorithm – it may give a small accurate tree for small datasets and this tree will probably be interpretable.

However, the CART implementation of decision trees (as used in the sklearn implementation of decision trees) leads to a complicated and at times, bizarre decision tree. An example is that when rules made on boolean features (only two options of 0 and 1) in the decision tree, the decision node uses a rule such as “ATTRIBUTE <= 0.5”. Having rules being <= 0.5 for boolean may lead to the decision tree being hard to understand for a human, even an expert human who understands the data from which the decision tree is created from. This is because a human would never build such a rule for boolean features the way CART does.

For example, take the first decision node in the decision tree for the animals dataset is a rule on the feature milk (Figure 13). Milk is a boolean feature – either milk == 1 and the classification is mammal or milk == 0 and therefore the classification is not a mammal and the decision tree moves to the next decision node. However instead of a rule such as "IF milk == 1, THEN mammal", the decision node can be thought of as "IF milk <= 0.5 THEN NOT a mammal". Using a threshold in between the 2 possible values creates a confusing rule. This is not intuitive to the user because they may think that if that rule regarding milk is true then the classification is mammal, else, go to the false branch, but it is the opposite for the decision tree – making it hard to understand.

From the experience described in the example, it can be seen that decision trees are not as good as we thought – possibly not interpretable. Rules have an advantage in this instance. Rules are made by a human-in-the-loop and therefore are naturally understandable to humans, unlike decision nodes made by an algorithm. Rules also allow flexible operators (<, >, =, ...) which allow rules to be more flexible and easier to understand than decision tree implementation. Therefore, if someone was to look at the rules created by the CART decision tree versus rules made by a human, the rules created by a human would be easier to understand since they would make more sense.

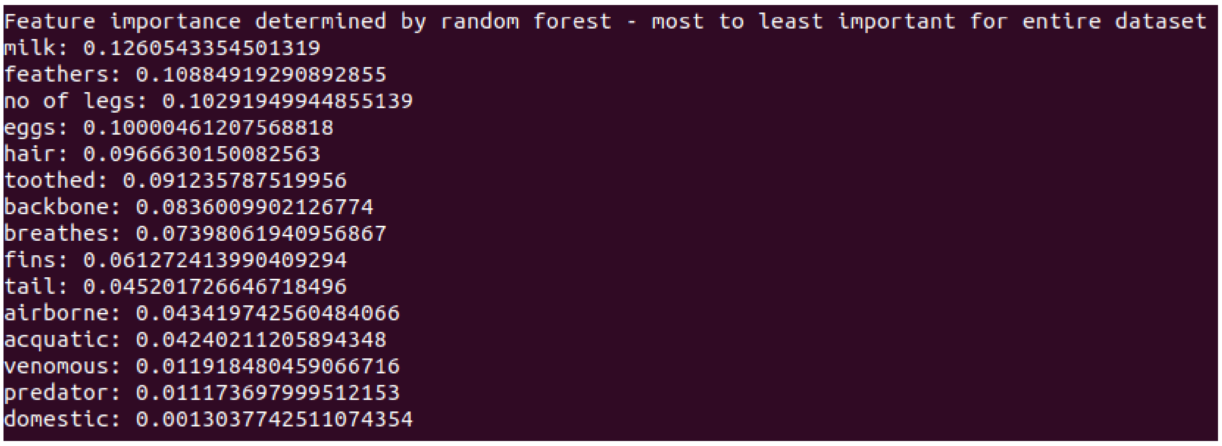
This gives an advantage of using RDR for explainable AI over algorithmic methods such as decision trees and even LIME since there is a human-in-the-loop during the development of the RDR ruleset – a human will be creating the ruleset – making the rules understandable for humans and therefore explainable.

4.3.3 Feature importances - Random forests

The random forest instance has a property called feature_importances_. The property returns an array of feature importances which has a length of the number of features. The feature importances contained in this array show how important each feature was to the random forest – with higher values meaning higher importance. The sklearn documentation gives the following description:
"The impurity-based feature importances. The higher, the more important the feature. The importance of a feature is computed as the (normalized) total reduction of the

criterion brought by that feature. It is also known as the Gini importance. Warning: impurity-based feature importances can be misleading for high cardinality features (many unique values).” [15]

In other words – the feature importance value is the mean decrease in impurity (MDI) to the random forest brought by that feature. Therefore, the array contains an array of numbers showing which features were important for the entire forest. These numbers sum to 1 – due to normalisation. When run multiple times on the same dataset, feature importance changes are minimal. Figure 19 shows the feature importances for the animals dataset.



```
Feature importance determined by random forest - most to least important for entire dataset
milk: 0.1260543354501319
feathers: 0.10884919290892855
no of legs: 0.10291949944855139
eggs: 0.10000461207568818
hair: 0.0966630150082563
toothed: 0.091235787519956
backbone: 0.0836009902126774
breathes: 0.07398061940956867
fins: 0.061272413990409294
tail: 0.045201726646718496
airborne: 0.043419742560484066
acquatic: 0.04240211205894348
venomous: 0.011918480459066716
predator: 0.011173697999512153
domestic: 0.0013037742511074354
```

***Figure 19** - feature importances for the animals dataset*

Figure 19’s feature importance values are represented on the following graph in Figure 20.

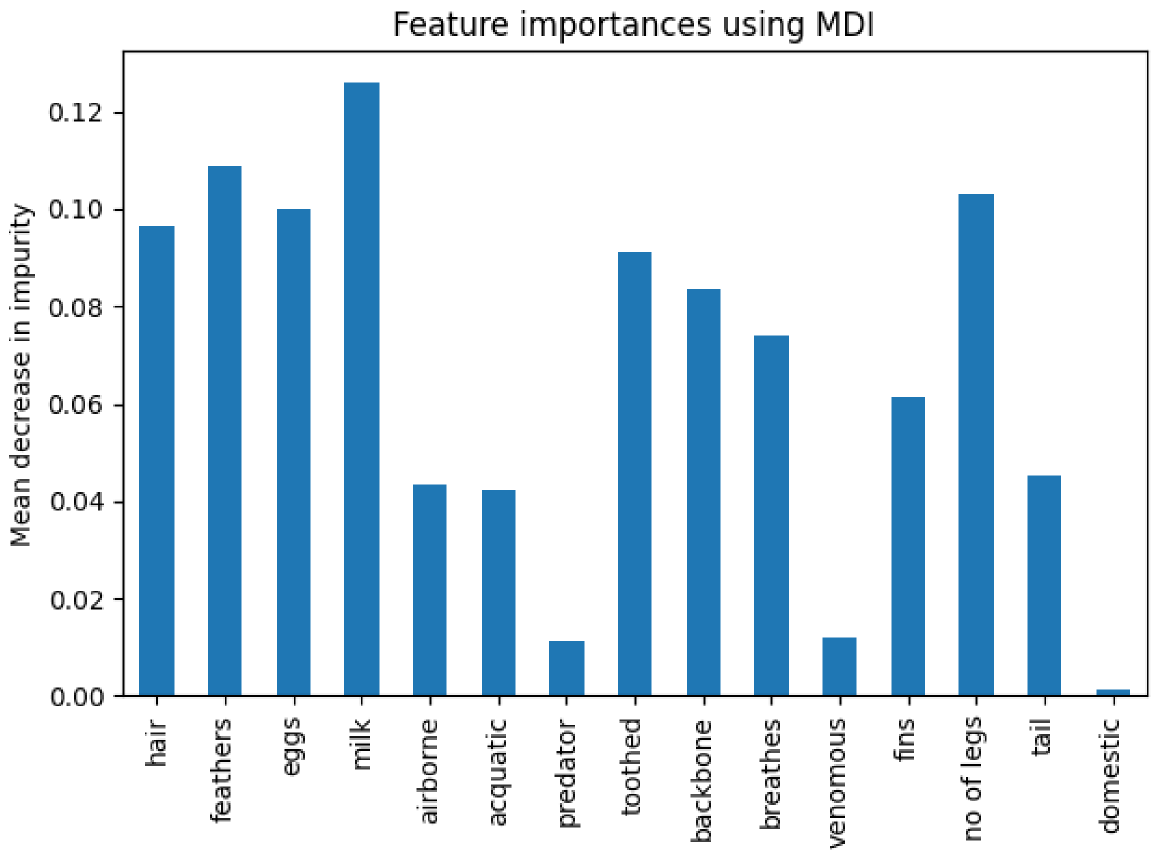


***Figure 20** - graphed feature importance values*

From looking at both Figure 19 and Figure 20, a user can see that features such as milk and feathers are important to the random forest and they would be ideal features to make a rule on.

Another method called permutation importance was tested to find the important features according to a random forest to potentially give another view on feature importance. This method used a function from sklearn called permutation_importance to calculate feature importance. Unlike the feature_importances method which uses MDI to find the important features, permutation importance uses mean accuracy decrease (MAD).

The sklearn documentation describes how it is calculated:

"The permutation importance of a feature is calculated as follows. First, a baseline metric, defined by scoring, is evaluated on a (potentially different) dataset defined by the X. Next, a feature column from the validation set is permuted and the metric is evaluated again. The permutation importance is defined to be the difference between the baseline metric and metric from permutating the feature column.” [15]

The sklearn documentation also outlines some benefits of this approach:
“Permutation feature importance overcomes limitations of the impurity-based feature importance: they do not have a bias toward high-cardinality features and can be computed on a left-out test set. Features are shuffled n times and the model refitted to estimate the importance of it. “ [15]

The resulting values can be plotted to find the most important features. From testing the permutation importance method, the results were wild. The following figures show the results from permutation importance after running the program 2 separate times on the animals dataset.

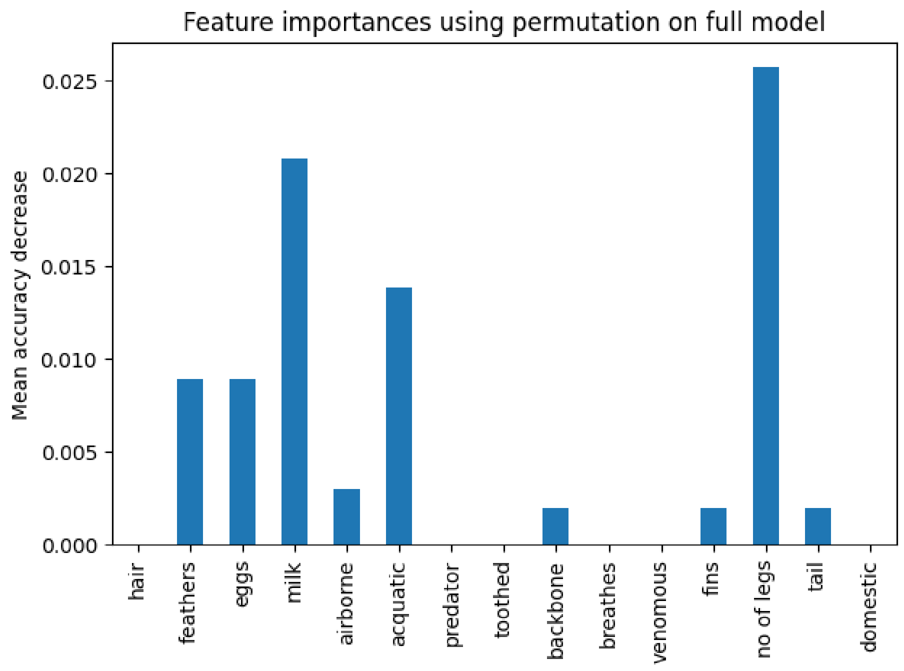


Figure 21 *- feature importances using permutation run one time*

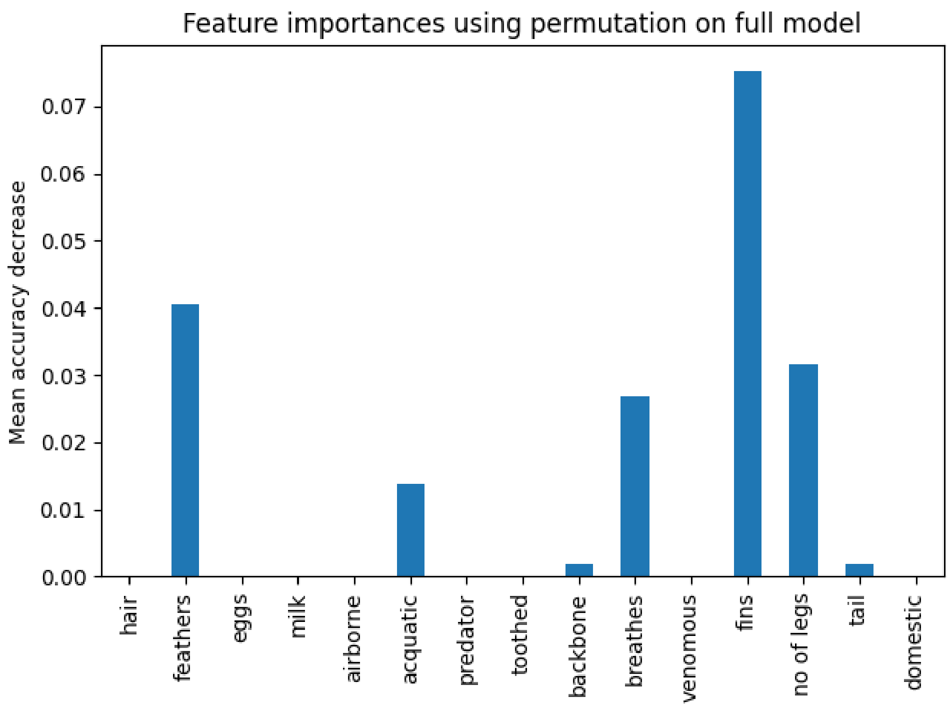


***Figure 22** - feature importances using permutation run another time*

As can be seen from Figure 21 and Figure 22, there are wild disparities in the results when permutation importances is run several times on the same dataset. For example, in Figure 21, milk is shown as the second most important, but in Figure 22, milk has a MAD score of 0. However, the figures also show that some features have a high reading most of the time, such as feathers and no of legs.

Comparing MDI to MAD methods of finding important features, the MDI method is less likely to fully omit a feature (only the feature "domestic" is close to 0) giving values for all features. It can also be seen that the same features are generally found to be most important in both methods (milk, no of legs). Due to having more consistent readings, the feature importances (using MDI) is preferred to permutation importance (using MAD). Nevertheless, both methods can be used to give the user some information on the relative importance of features giving them hints on what features to make rules upon.

4.3.4 Feature importances – XGBoost

XGBoost also has a feature importance attribute called: importances. XGBoost's method plot_importance shows this graph of the importances values. [16] The features are measured on an "F score" which refers to the important of that feature according to a criterion – the higher the score for a feature, the higher importance. A split refers to the feature being used in a decision node in a tree. There are 3 criteria which are:

- **Weight** – the number of times the feature appears in the tree in a split
- **Gain** – the average gain of splits which use the feature. In other words, this is the relative contribution of the corresponding feature to the model calculated by taking each feature's contribution for each tree in the model. A higher value for a certain feature when compared to another feature implies it is more important for prediction
- **Cover** – the average coverage where coverage is the number of samples affected by splitting on this feature

Below are the plots for the 3 criteria for feature importance on the animals dataset.

Feature importance - Weight

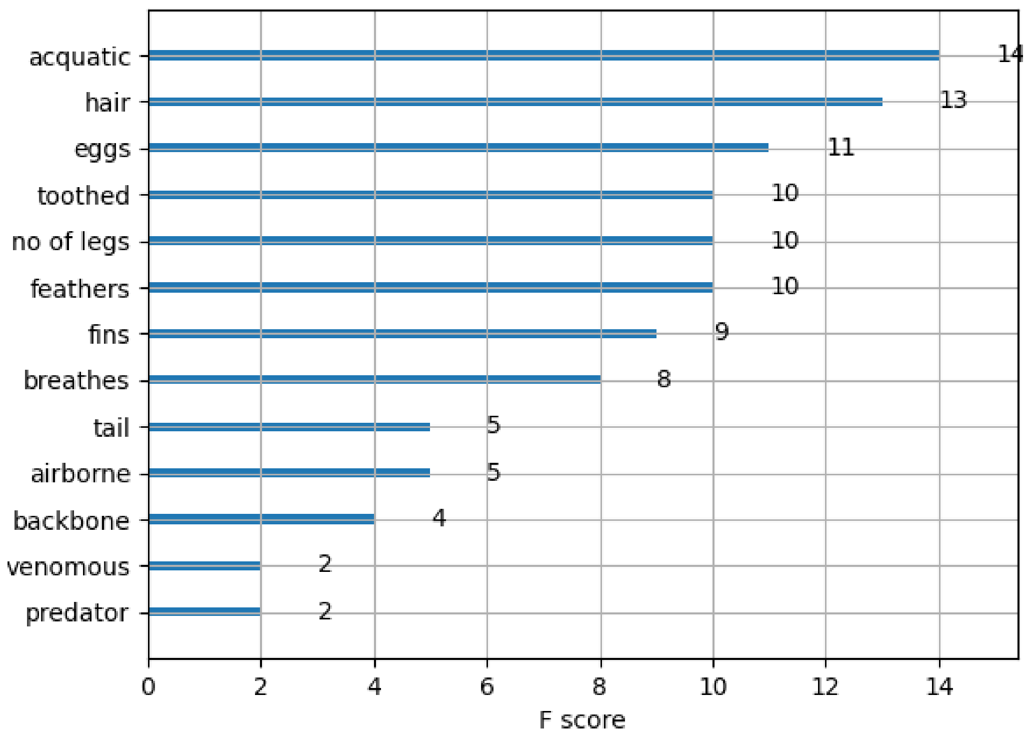


***Figure 23** - feature importance using criteria weight*

Feature importance - Gain

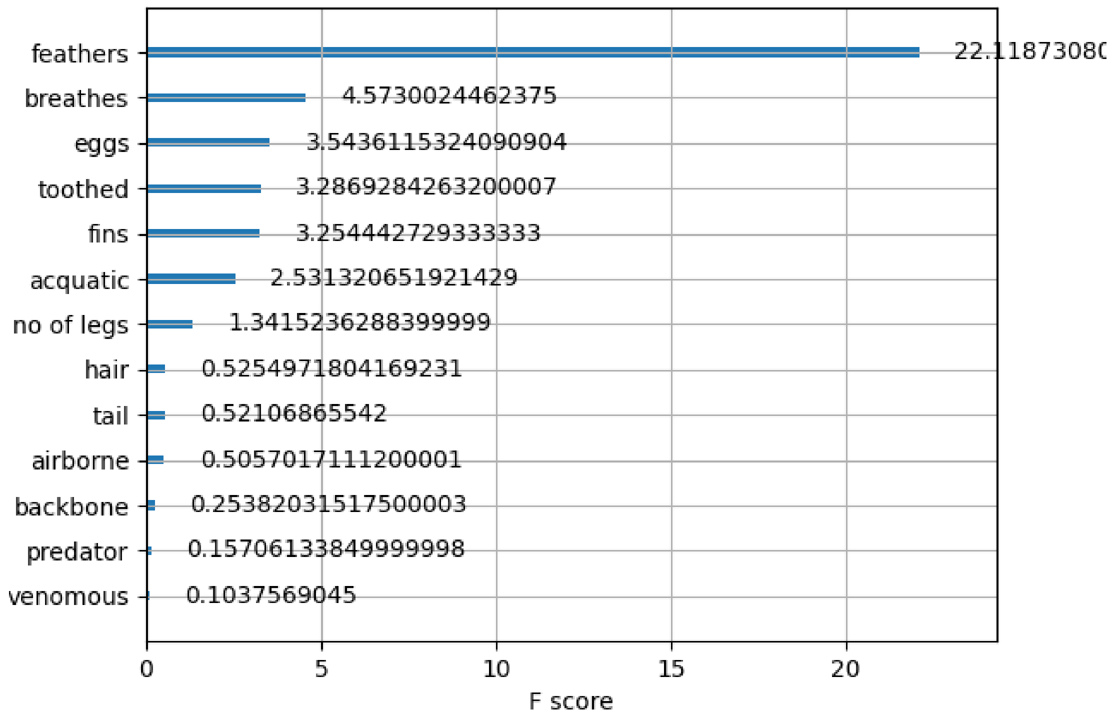


Figure 24 - *feature importance using criteria gain*

Feature importance - Cover

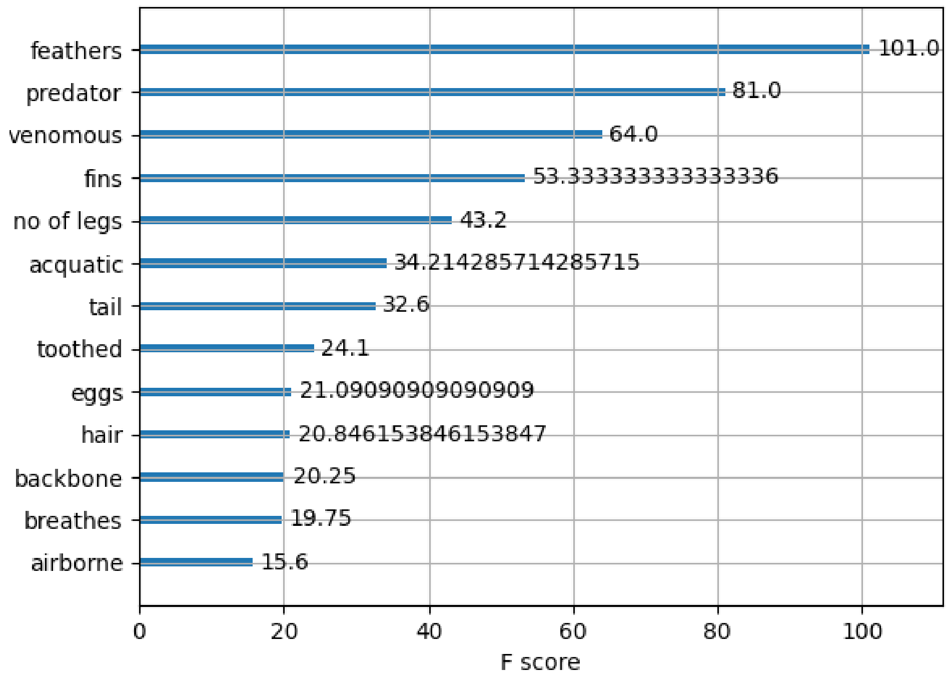


Figure 25 - *feature importance using criteria cover*

Figures 23, 24, 25 show the most important features of the animals dataset according to the 3 criteria. By looking at these graphs, users can see what XGBoost believes are important features and may choose to use these features in a rule.

A large amount of time was spent trying to work out why the features “milk” and “domestic” were omitted from the graphs for XGBoost feature importance as there were only 13 features included on the graphs (instead of the full 15 features used from the animals dataset). Upon investigation, it was found that features with 0 importance are not included in function get_score. The function get_score must be used by plot_importances therefore leaving out feature that have 0 score. This raises an interesting observation. The feature_importances of random forests found that the feature “milk” was one of the most important features, however, all 3 of the criteria for feature_importances in XGBoost gave milk a score of 0. It is believed that XGBoost used other features to classify mammals, features which it thought were more important than milk. This left the feature milk obsolete leaving it with a score of 0. However, this is only a theory and it is not known for sure why there is such a disparity between random forest’s feature_importances and XGBoost’s feature_importance for milk.

4.3 Other features

There were also other features that developed in order to help the user experience or give additional tools to the user:

- Clear conclusions
- Clear rules
- Run rules on all cases without stopping – this method runs through the cases without making a rule and prints the accuracy pf the ruleset of the data
- Run 10-fold cross validation on all “black box” algorithms
- save/load ruleset – the ability to save/load the rules was added. Before this, when the program was exited, all rules were lost. With this feature, rules can be saved to a file and also loaded from a file. Note: only data containing integers and strings can use this feature.

- Various evaluation methods

These features will be demonstrated in chapter 5 during the program walkthrough or in other evaluation sections in chapter 5.

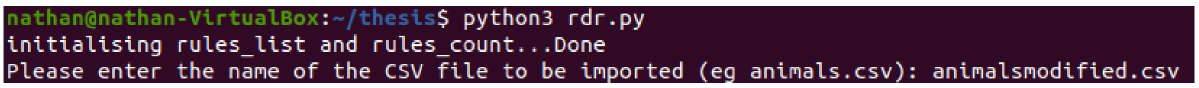
Chapter 5

Experimental evaluation

This chapter will give a demonstration of how the system is used by showing the results from a sample use of the system on the animals dataset including the addition of some rules. Afterwards is evaluation which includes using a new dataset as well as other evaluation methods. Evaluating the explainability level of an explainability model is not a straightforward task as there is no easy measure of explainability. The explainability of a model may be different for different people depending on their knowledge or background in the topic. Therefore, the following evaluation will focus on evaluating the created system's effectiveness as a whole.

5.1 Walkthrough

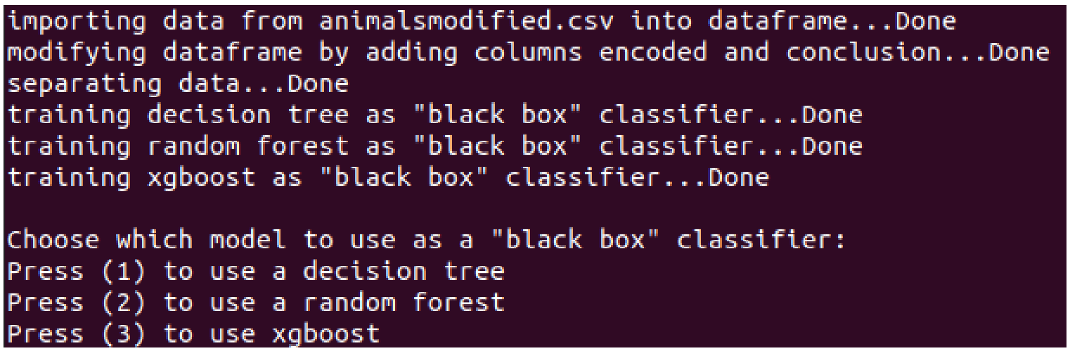
In a terminal, if we run the python program using python3, we are greeted with the following screen:



```
nathan@nathan-VirtualBox:~/thesis$ python3 rdr.py
initialising rules_list and rules_count...Done
Please enter the name of the CSV file to be imported (eg animals.csv): animalsmodified.csv
```

Figure 26 – *user inputs CSV file name which is to be imported*

We are prompted to input the name of the CSV (comma separated values) file which is to be imported into the system. There are multiple conditions that must be met for this file. The file must be a CSV file and have a .csv format in its name. The first column must be called “name’ and contain the name of each case. The last column must be called “target” and contain the labels for each case. The labels must be of type string due to label encoding. The data (all columns in between the first and last column) must contain numeric data only. There must be no missing values. Examples on how to edit CSV files into a format suitable for this program are provided in this project’s GitHub repository. [3]

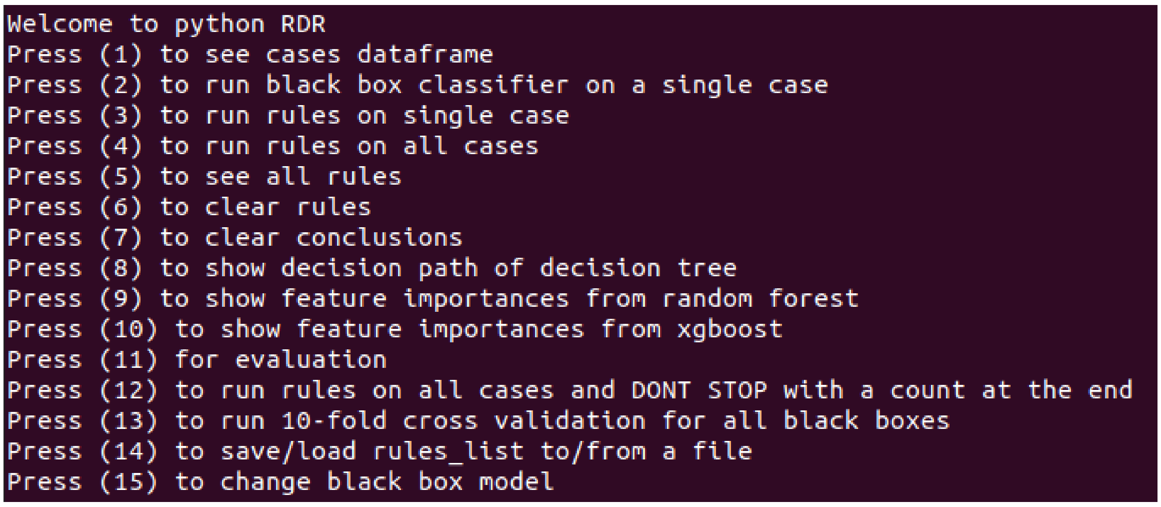


```
importing data from animalsmodified.csv into dataframe...Done
modifying dataframe by adding columns encoded and conclusion...Done
separating data...Done
training decision tree as "black box" classifier...Done
training random forest as "black box" classifier...Done
training xgboost as "black box" classifier...Done

Choose which model to use as a "black box" classifier:
Press (1) to use a decision tree
Press (2) to use a random forest
Press (3) to use xgboost
```

Figure 27 *- user inputs a number corresponding to the black box classifier they want to use as a black box*

After entering the CSV file to be used, we now must select a classifier to be used as the black box classifier. The selection can be changed later as all classifiers are trained when the program starts (as shown in Figure 27). For this example, option 2 - random forest is chosen.

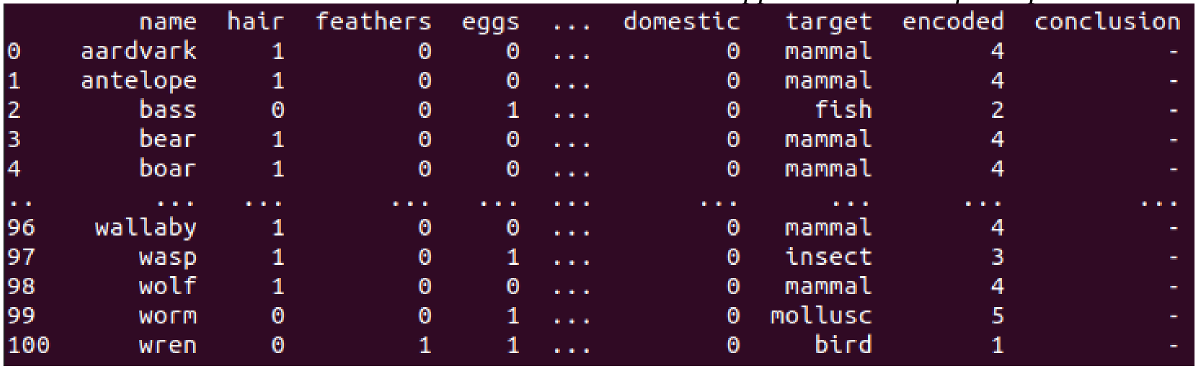
We are now shown a menu screen containing all options available to a user.



```
Welcome to python RDR
Press (1) to see cases dataframe
Press (2) to run black box classifier on a single case
Press (3) to run rules on single case
Press (4) to run rules on all cases
Press (5) to see all rules
Press (6) to clear rules
Press (7) to clear conclusions
Press (8) to show decision path of decision tree
Press (9) to show feature importances from random forest
Press (10) to show feature importances from xgboost
Press (11) for evaluation
Press (12) to run rules on all cases and DONT STOP with a count at the end
Press (13) to run 10-fold cross validation for all black boxes
Press (14) to save/load rules_list to/from a file
Press (15) to change black box model
```

Figure 28 *- menu screen*

Entering 1 (and pressing enter), the user is shown the dataframe of cases.

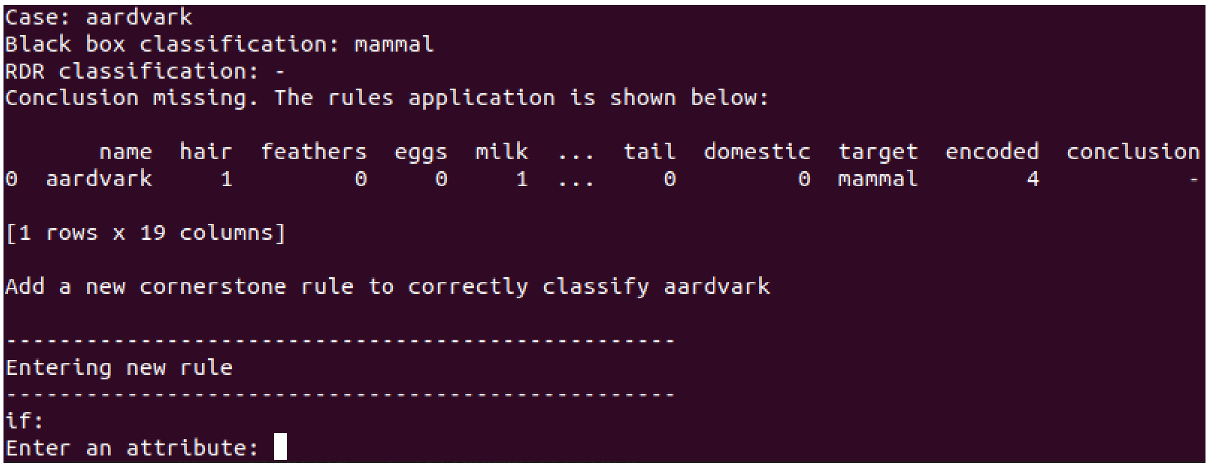


```
         name  hair  feathers  eggs  ...  domestic   target  encoded  conclusion
0    aardvark     1         0     0  ...         0   mammal        4           -
1    antelope     1         0     0  ...         0   mammal        4           -
2        bass     0         0     1  ...         0     fish        2           -
3        bear     1         0     0  ...         0   mammal        4           -
4        boar     1         0     0  ...         0   mammal        4           -
..        ...   ...       ...   ...  ...       ...      ...      ...         ...
96    wallaby     1         0     0  ...         0   mammal        4           -
97       wasp     1         0     1  ...         0   insect        3           -
98       wolf     1         0     0  ...         0   mammal        4           -
99       worm     0         0     1  ...         0  mollusc        5           -
100      wren     0         1     1  ...         0     bird        1           -
```

***Figure 29** - dataframe of cases*

Figure 29 shows a truncated version of the dataframe containing the rows of animals and their data as well as the 2 added columns, an encoded version of the target and empty conclusion column. Note: for this demonstration, there is a large amount of truncation in the images shown in order allow the text in these images to be readable. When using the program, if the terminal is enlarged before entering options on the menu screen, the user would be shown more information.

Now it is time to add some rules. The option which provides the implementation of the high-level algorithm is option 4 – run rules on all cases. Press 4.



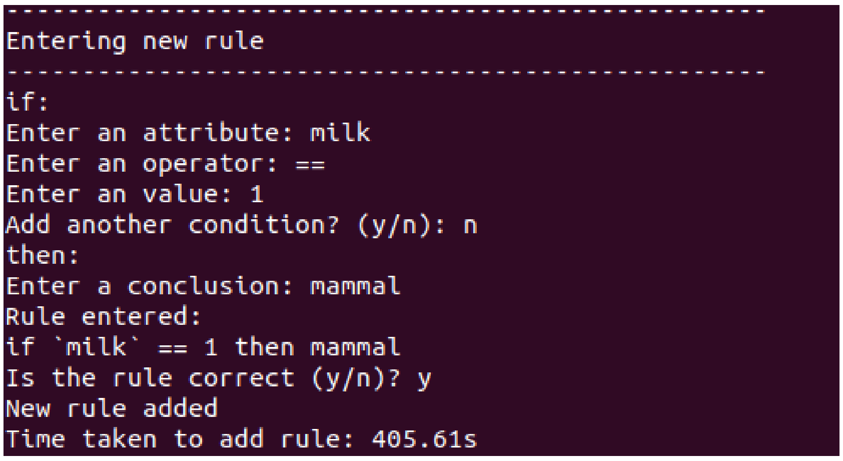
```
Case: aardvark
Black box classification: mammal
RDR classification: -
Conclusion missing. The rules application is shown below:

       name  hair  feathers  eggs  milk  ...  tail  domestic  target  encoded  conclusion
0  aardvark     1         0     0     1  ...     0         0  mammal        4           -

[1 rows x 19 columns]

Add a new cornerstone rule to correctly classify aardvark

--------------------------------------------------
Entering new rule
--------------------------------------------------
if:
Enter an attribute: █
```

***Figure 30** - the first case*

As described by the option description, the rules are run on all the cases in the dataframe. The first case in the dataframe is the aardvark. The black box prediction is mammal, but the RDR classification is “-“ (or none). This is because when we begin, there are zero rules in the system. Next, the user is told that the conclusion is missing and is shown the application of the rules on the case which led to this classification.

However, as stated, there are zero rules currently, so there is no rules application. The user is then shown the aardvark case and instructed to add a new cornerstone rule. A cornerstone rules means that no rules are true for this case so therefore it has no classification.

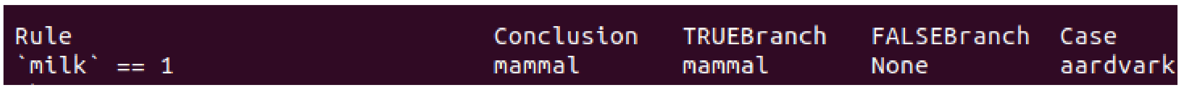
To enter a new rule, the user must first enter an attribute. We know from before that milk is a defining attribute for mammals, so write "milk".



```
--------------------------------------------------
Entering new rule
--------------------------------------------------
if:
Enter an attribute: milk
Enter an operator: ==
Enter an value: 1
Add another condition? (y/n): n
then:
Enter a conclusion: mammal
Rule entered:
if `milk` == 1 then mammal
Is the rule correct (y/n)? y
New rule added
Time taken to add rule: 405.61s
```

***Figure 31** - adding a new rule*

Next, an operator is required. The operators available are: ==, >, <, >=, <=. We will choose ==. Next the value we will choose 1 to make the rule: if milk == 1. It is also possible to add another condition, for example: if milk == 1 AND feathers == 0. However, this single condition will suffice in this case so we enter "n" to decline this option. After this we enter the conclusion that the rule will give as a classification. For this rule, we need to classify as a mammal as the black box predicts so choose mammal. After confirming our rule, it is added and we are taken to the menu. At the menu we can press option 5 to see all rules.

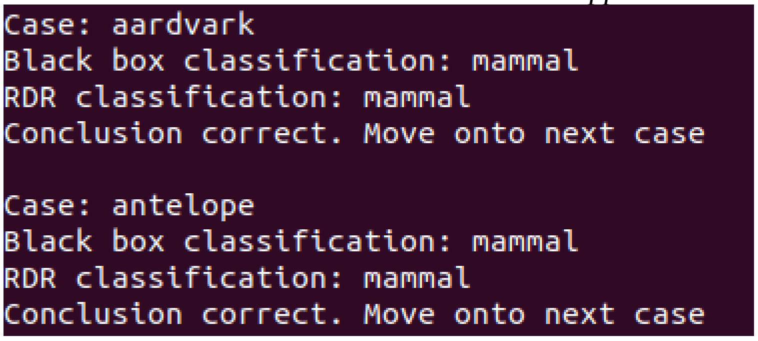


```
Rule                                 Conclusion   TRUEBranch   FALSEBranch   Case
`milk` == 1                          mammal       mammal       None          aardvark
```

***Figure 32 -** the single rule that has been added*

The single rule we have made appears. True_cases is also an attribute of a rule that would be displayed here but it has been omitted to save space.

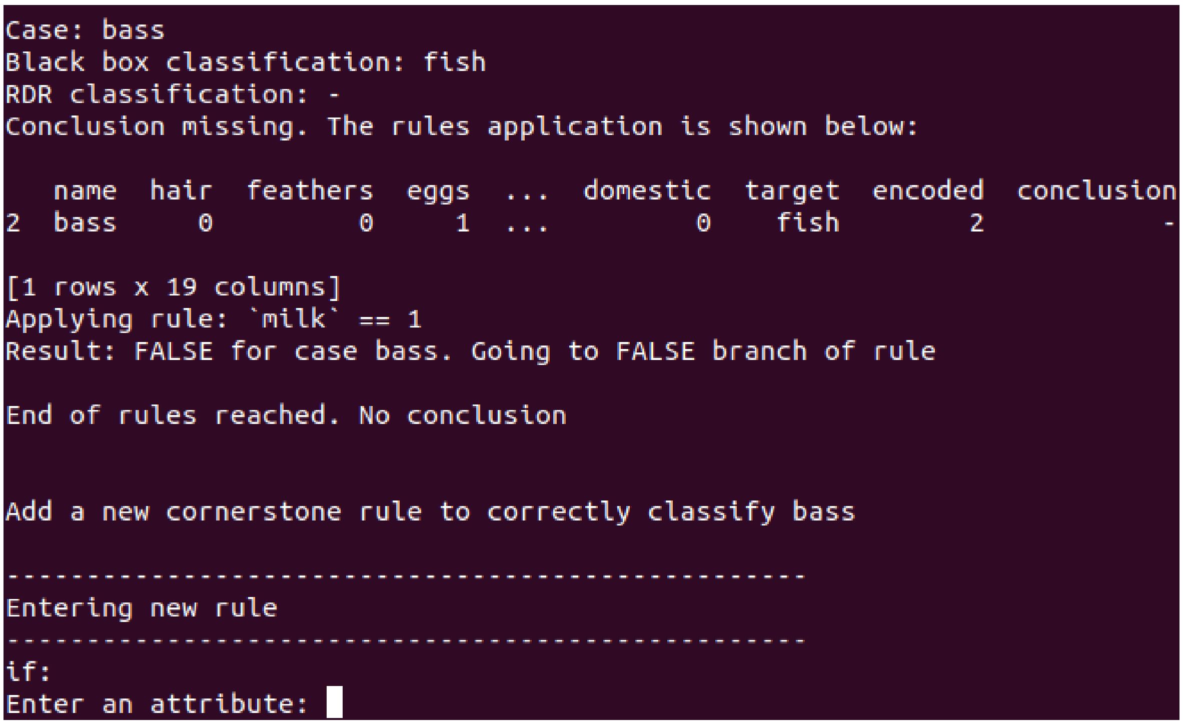
If we press 4 again, the rules are run on all the cases just as before, but this time, there is a rule in the ruleset – the one we just created.



```
Case: aardvark
Black box classification: mammal
RDR classification: mammal
Conclusion correct. Move onto next case

Case: antelope
Black box classification: mammal
RDR classification: mammal
Conclusion correct. Move onto next case
```

Figure 33 *- the aardvark and antelope and now correctly classified*

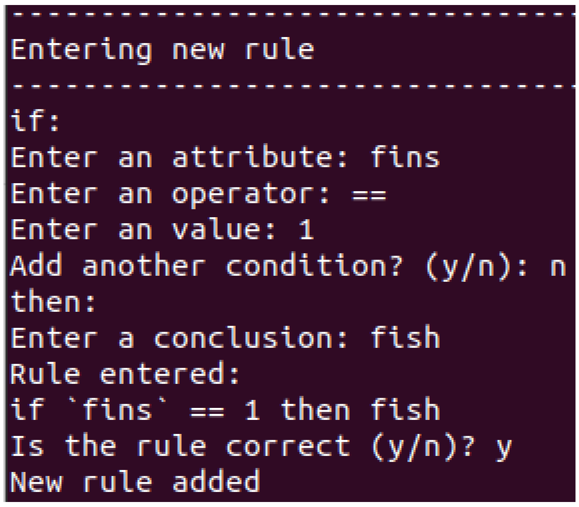
The aardvark is now correctly classified as a mammal by the rule that was just created. When a case is classified correctly, we move to the next case – just as shown in the system architecture diagram. The antelope is the next case and it is also correctly classified as a mammal by our rule. The next case is the bass.



```
Case: bass
Black box classification: fish
RDR classification: -
Conclusion missing. The rules application is shown below:

   name  hair  feathers  eggs  ...  domestic  target  encoded  conclusion
2  bass     0         0     1  ...         0    fish        2           -

[1 rows x 19 columns]
Applying rule: `milk` == 1
Result: FALSE for case bass. Going to FALSE branch of rule

End of rules reached. No conclusion


Add a new cornerstone rule to correctly classify bass

--------------------------------------------------
Entering new rule
--------------------------------------------------
if:
Enter an attribute:
```

Figure 34 *- the fish is not classified by our 1 rule*

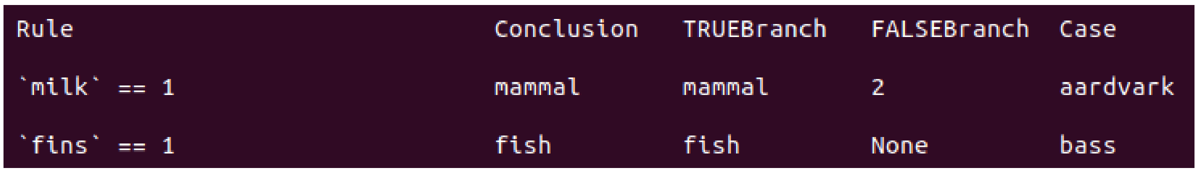
The bass however has no conclusion given by the RDR – just like we saw with the aardvark when we first saw it. As with the aardvark, the rules application is shown, this time with a rule. When the rule milk == 1 is applied to the bass, this is false – therefore the false branch os taken. As shown in the rules print out, the false branch for this rule is none – so no classification is given. We must now add a cornerstone rule to classify the fish. We add the rule as follows (note: fins == 1 for the bass):



```
--------------------------------
Entering new rule
--------------------------------
if:
Enter an attribute: fins
Enter an operator: ==
Enter an value: 1
Add another condition? (y/n): n
then:
Enter a conclusion: fish
Rule entered:
if `fins` == 1 then fish
Is the rule correct (y/n)? y
New rule added
```

Figure 35 *- entering a rule to classify the bass*

We add a rule on fins for the bass to classify it as a fish – the classification given by the “black box” classifier. We now have 2 rules in the ruleset, which can be seen by pressing 5.

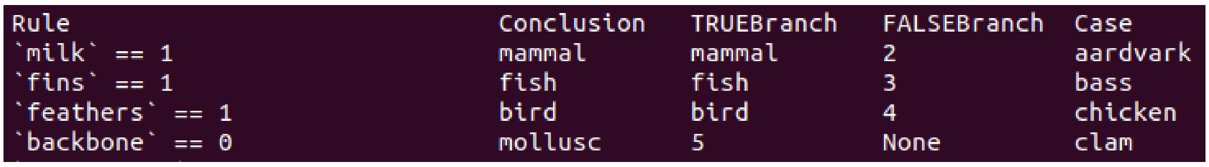


```
Rule                              Conclusion    TRUEBranch    FALSEBranch    Case

`milk` == 1                       mammal        mammal        2              aardvark

`fins` == 1                       fish          fish          None           bass
```

Figure 36 *- there are now 2 rules in the system*

As you can see, the FALSEBranch for rule 1 has been updated to point to the second rule – the rule we just added. Now when milk is not equal to 1 for a case, the next cornerstone rule is applied, which if successful, will classify the case as a fish. If both rules are false for a case, then we will have no conclusion and we need to add a new cornerstone rule.

Choosing the fourth option 2 more times will make the user make 2 more cornerstone rules to classify cases which have no RDR classification. After adding 4 rules, looking at all the rules looks like this:

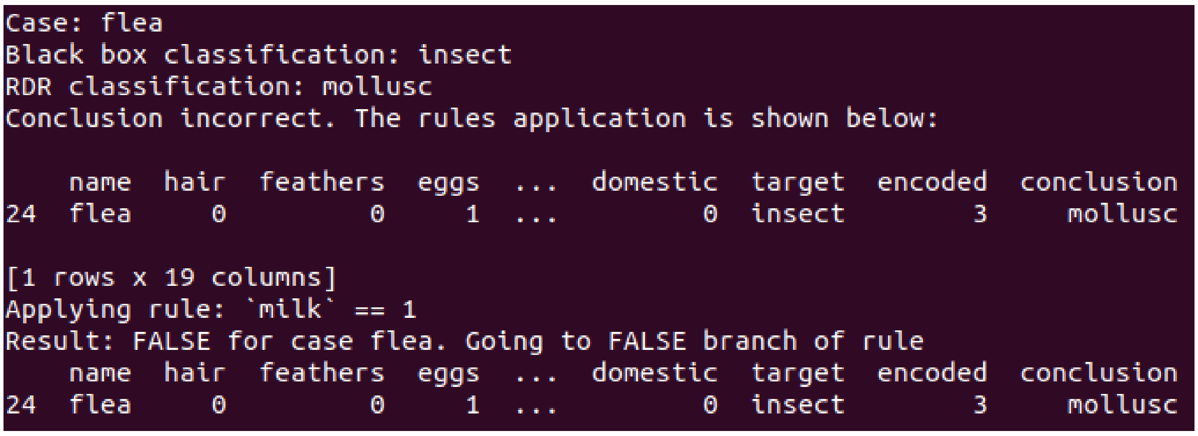


```
Rule                              Conclusion    TRUEBranch    FALSEBranch    Case
`milk` == 1                       mammal        mammal        2              aardvark
`fins` == 1                       fish          fish          3              bass
`feathers` == 1                   bird          bird          4              chicken
`backbone` == 0                   mollusc       5             None           clam
```

Figure 37 - *the ruleset after adding 3 more rules*

Since all 4 rules are cornerstone rules, if the rule is false for a case, we simply apply the next rule in the ruleset.

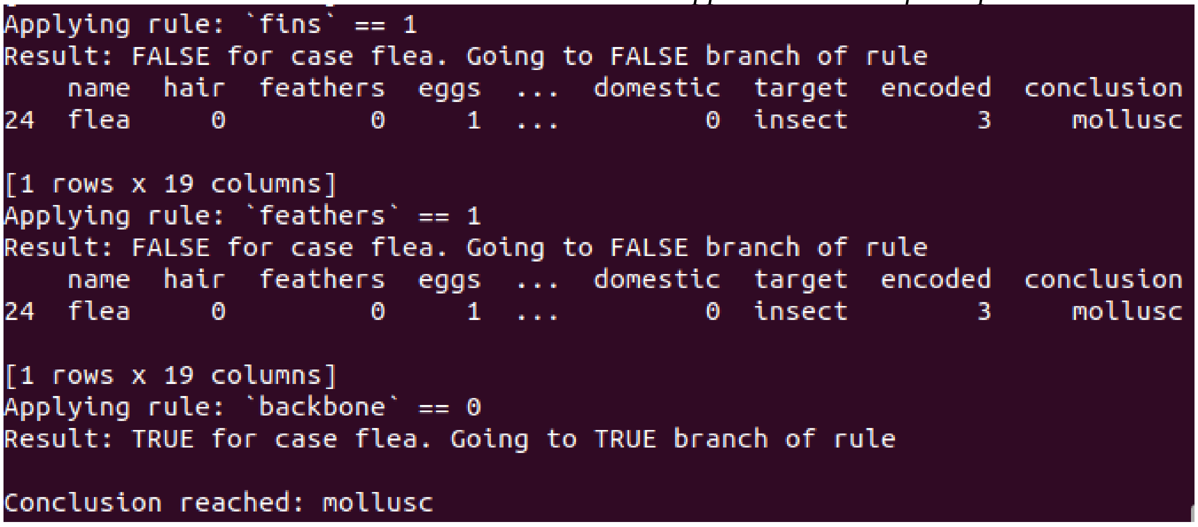
Now when running option 4 once more, we are presented with the case "flea".



```
Case: flea
Black box classification: insect
RDR classification: mollusc
Conclusion incorrect. The rules application is shown below:

    name  hair  feathers  eggs  ...  domestic  target  encoded  conclusion
24  flea     0         0     1  ...         0  insect        3     mollusc

[1 rows x 19 columns]
Applying rule: `milk` == 1
Result: FALSE for case flea. Going to FALSE branch of rule
    name  hair  feathers  eggs  ...  domestic  target  encoded  conclusion
24  flea     0         0     1  ...         0  insect        3     mollusc
```

Figure 38 *- incorrect RDR conclusion*

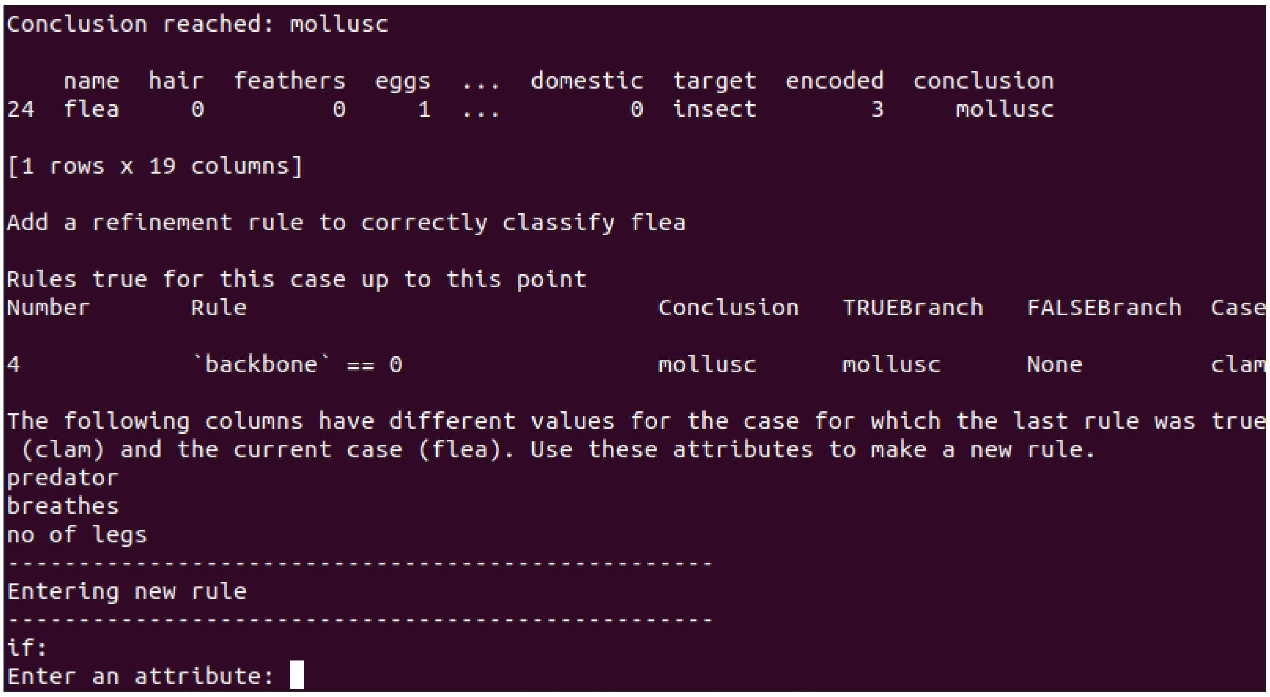
The black box prediction is insect whereas the RDR classification is mollusc. Therefore, the RDR rules have got this case's classification wrong, meaning a refinement rule must be added to correct this mistake. The system now shows us the application of all the rules in the system. The first rule to be applied was the first rule we added, a rule on milk. Unfortunately, in figure 38, the value for milk for the flea is omitted due to an effort to make the output readable in this paper. The value for the rule milk == 1 for the flea is, as shown by the output, false. The false branch, as shown in the ruleset (in figure 37) is rule 2. Rule 2 is on fins. Again, this is excluded from Figure 38, but it is 0 for the flea. Going to the false branch we apply rule 3, a rule on feathers. As can be seen, this value is 0 for the flea so again we go down the false branch. The final rule, backbone = 0 is true for the flea, so it is classified as a mollusc.



```
Applying rule: `fins` == 1
Result: FALSE for case flea. Going to FALSE branch of rule
    name  hair  feathers  eggs  ...  domestic  target  encoded  conclusion
24  flea     0         0     1  ...         0  insect        3     mollusc

[1 rows x 19 columns]
Applying rule: `feathers` == 1
Result: FALSE for case flea. Going to FALSE branch of rule
    name  hair  feathers  eggs  ...  domestic  target  encoded  conclusion
24  flea     0         0     1  ...         0  insect        3     mollusc

[1 rows x 19 columns]
Applying rule: `backbone` == 0
Result: TRUE for case flea. Going to TRUE branch of rule

Conclusion reached: mollusc
```

***Figure 39** - rules application for the flea*

But this conclusion of mollusc is different to the black box classification of insect - so a refinement rule must be added.

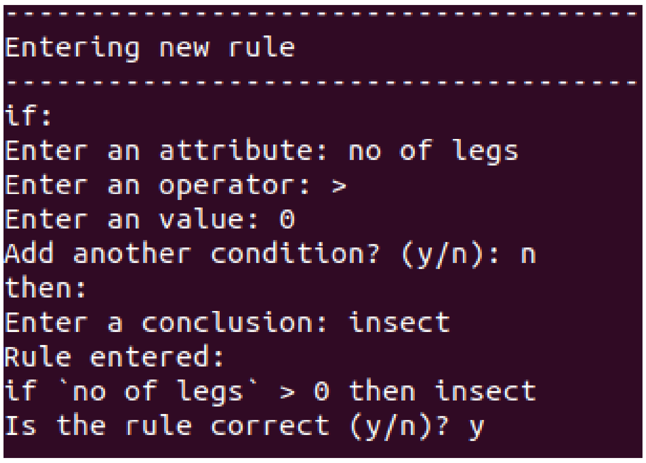


```
Conclusion reached: mollusc

    name  hair  feathers  eggs  ...  domestic  target  encoded  conclusion
24  flea     0         0     1  ...         0  insect        3     mollusc

[1 rows x 19 columns]

Add a refinement rule to correctly classify flea

Rules true for this case up to this point
Number       Rule                           Conclusion    TRUEBranch    FALSEBranch   Case

4             `backbone` == 0               mollusc       mollusc       None          clam

The following columns have different values for the case for which the last rule was true
 (clam) and the current case (flea). Use these attributes to make a new rule.
predator
breathes
no of legs
--------------------------------------------------
Entering new rule
--------------------------------------------------
if:
Enter an attribute: 
```

***Figure 40** - the system gives assistance to the user making a rule*

The system now shows us all the rules that are true for this case. In this case it is a single rule, rule 4, which makes a rule on the attribute “backbone”. Therefore, since this rule is true, there is no point in making a rule in terms of backbone. Now the system attempts to help us find an attribute to make a rule on by listing all the attributes that are different between this case (the flea) and the case the true rule was made on (the clam). The new rule must involve features that are different to ensure that the rule already true applies for the case it was made on (since it classifies this case correctly), and the refinement rule only works on this rule.

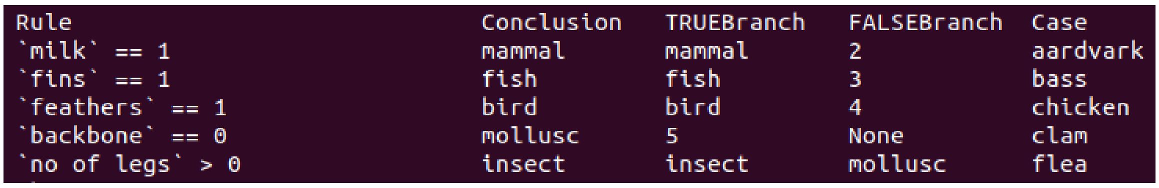
Now we need to enter the refinement rule. We choose the attribute "no of legs" from the provided list and make the rule as follows:



```
------------------------------------
Entering new rule
------------------------------------
if:
Enter an attribute: no of legs
Enter an operator: >
Enter an value: 0
Add another condition? (y/n): n
then:
Enter a conclusion: insect
Rule entered:
if `no of legs` > 0 then insect
Is the rule correct (y/n)? y
```

***Figure 41** - creating the refinement rule*

Now we have a refinement rule that correctly classifies the flea as an insect.

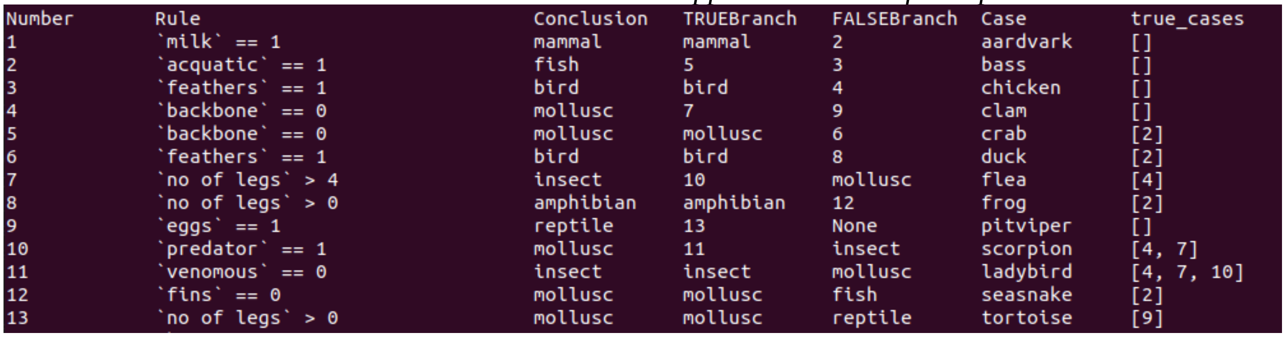


```
Rule                              Conclusion   TRUEBranch   FALSEBranch   Case
`milk` == 1                       mammal       mammal       2             aardvark
`fins` == 1                       fish         fish         3             bass
`feathers` == 1                   bird         bird         4             chicken
`backbone` == 0                   mollusc      5            None          clam
`no of legs` > 0                  insect       insect       mollusc       flea
```

***Figure 42** - the ruleset after adding 5 rules*

In Figure 42, we can see that rule 4's TRUEBranch and FALSEBranch values have been modified by this action. Previously, if rule 4 was true, then a classification of mollusc was returned. But the refinement rule added to this rule changes this. Now when backbone == 0 is true for a case, we go to rule 5 and evaluate it. If rule 5 is true, like it was for the flea, then we are given a classification of insect. However, if it is false, the classification which was originally made for rule 4 – mollusc – is returned. The false branch of rule 4 retains a value of none. This all means that rule 5 will always but only we applied if rule 4 is true – as it is a refinement rule.

If we continue adding rules until every single animal in the animals dataset is classified correctly, this is a ruleset that we may finish on:

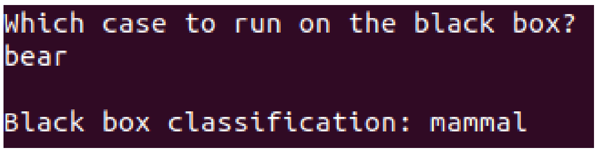


```
Number        Rule                            Conclusion   TRUEBranch   FALSEBranch  Case         true_cases
1             `milk` == 1                     mammal       mammal       2            aardvark     []
2             `acquatic` == 1                 fish         5            3            bass         []
3             `feathers` == 1                 bird         bird         4            chicken      []
4             `backbone` == 0                 mollusc      7            9            clam         []
5             `backbone` == 0                 mollusc      mollusc      6            crab         [2]
6             `feathers` == 1                 bird         bird         8            duck         [2]
7             `no of legs` > 4                insect       10           mollusc      flea         [4]
8             `no of legs` > 0                amphibian    amphibian    12           frog         [2]
9             `eggs` == 1                     reptile      13           None         pitviper     []
10            `predator` == 1                 mollusc      11           insect       scorpion     [4, 7]
11            `venomous` == 0                 insect       insect       mollusc      ladybird     [4, 7, 10]
12            `fins` == 0                     mollusc      mollusc      fish         seasnake     [2]
13            `no of legs` > 0                mollusc      mollusc      reptile      tortoise     [9]
```

Figure 43 - *a ruleset that correctly classifies all cases*

With this print out of the ruleset, we have created an explainable model that explains the predictions of the "black box" classifier.

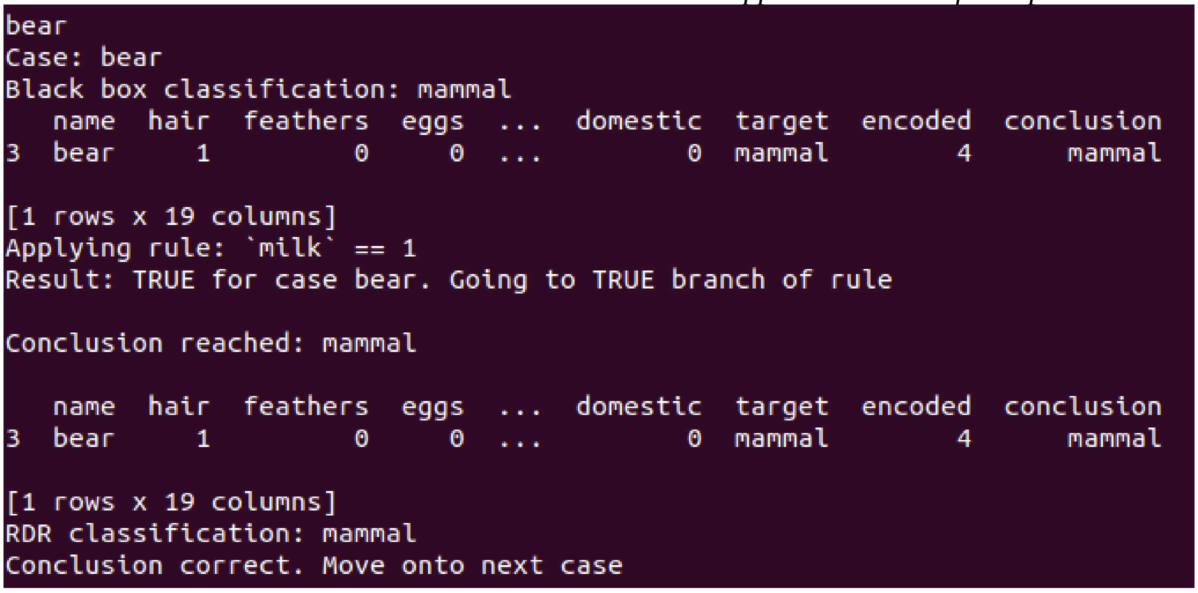
Now let's go over some of the other options in the menu. Option 2 runs the black box classifier on a case inputted by the user.



```
Which case to run on the black box?
bear

Black box classification: mammal
```

Figure 44 - *running the black box on case bear*

As is intended, this classification gives no information regarding how the black box reached its conclusion. To see an explainable version of this, we can use our RDR model to explain this prediction. Choosing option 3 will run the rules on a single case which is inputted by the user, explaining the application of the rules as makes a classification.



```
bear
Case: bear
Black box classification: mammal
   name  hair  feathers  eggs  ...  domestic  target  encoded  conclusion
3  bear     1         0     0  ...         0  mammal        4      mammal

[1 rows x 19 columns]
Applying rule: `milk` == 1
Result: TRUE for case bear. Going to TRUE branch of rule

Conclusion reached: mammal

   name  hair  feathers  eggs  ...  domestic  target  encoded  conclusion
3  bear     1         0     0  ...         0  mammal        4      mammal

[1 rows x 19 columns]
RDR classification: mammal
Conclusion correct. Move onto next case
```

Figure 45 - *the rules application for case bear*

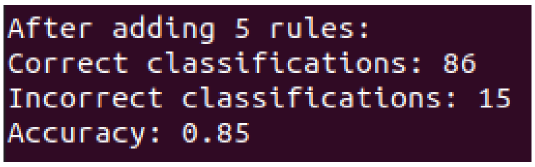
As we can see in Figure 45 above, the bear was classified as a mammal because of rule 1 which is IF milk == 1, THEN mammal. We the user can clearly see that the RDR ruleset reaches the classification of mammal because for the bear, milk == 1. In this case, the interpretable model has explained to the user how the conclusion was reached.

Now for the rest of the options in the menu:

6. Clears all the rules in the ruleset. If there are rules in the ruleset, they are removed and the user is placed back in the beginning as if the program was just run
7. Clears all conclusions in the “conclusion” column in the dataframe which are placed there when the RDR gives a classification of a case
8. Shows the decision path of the decision tree of a particular case as described in chapter 4. Note: since all classifiers are trained as the program started, it does not matter which black box was selected at the program start, options 8, 9 ,10 can be run at any time.
9. Shows the feature importances given by the random forest as shown in chapter 4. The graphs shown in chapter 4 are shown to the user and also saved as PNG files.
10. Shows the feature importances graphs given by XGBoost as shown in chapter

4 and saves them as PNG files.

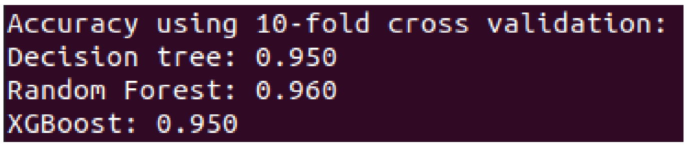
11. Shows an evaluation of the system. This will be discussed later this chapter in 5.4.
12. Run all the rules on all the cases without stopping to create a rule if a case is not classified or classified incorrectly. It is very similar to option 4, except no rules are added. After all the cases have been run through, an accuracy of the ruleset is calculated and shown to the user. For example, after adding the 5 rules shown in this walkthrough, this is the resulting accuracy of the rules:



```
After adding 5 rules:
Correct classifications: 86
Incorrect classifications: 15
Accuracy: 0.85
```

Figure 46 *- accuracy of rules on animals dataset after adding 5 rules*

13. Run 10-fold cross validation using all three black boxes and print the accuracy of each one for the current dataset. This may be useful when deciding if a dataset should be used for this system. Only datasets which give a high accuracy for the black boxes should be used for this system. The animals dataset accuracies are shown below. It has a high accuracy so is suitable for this system.



```
Accuracy using 10-fold cross validation:
Decision tree: 0.950
Random Forest: 0.960
XGBoost: 0.950
```

Figure 47 *- accuracy results for 10-fold cross validation on the animals dataset*

14. Save the current ruleset to a file or load a ruleset from a file. Note: only works for data in integer and/or string form.
15. Change the black box classifier that is used in options 2 and 4.

5.2 Using a new dataset – Titanic dataset

Using a new dataset was important for this project for multiple reasons. The first reason was that using a new, larger dataset allowed for a stress test of the system – in order to test how the system copes with a dataset containing many cases. This forms an evaluation on the system. Another reason was to prove that the system can indeed deal with datasets other than the animals dataset. Also, a new dataset provides new challenges in terms of creating new rules to add to new cases which have not been seen yet. The dataset that was used as the new dataset was a titanic dataset. The titanic dataset is from a Kaggle challenge [17] which involves using features about the people aboard the titanic to predict whether they survived the ships sinking or not.

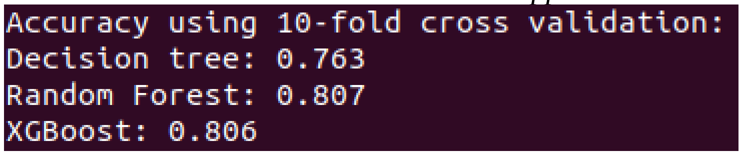
As with the animals dataset, the titanic dataset needed modifications before it could be used in the system. All features containing string values were removed (or converted into integers in the case of Sex). The target column was converted from binary integer values to strings "lived" or "died". Columns were shuffled around and renamed as needed. There were also many missing values in the data. Using a tool provided by Pandas called "fillna", these missing values were filled using a method called "ffill" – where a missing value is filled with the value in the corresponding column from the row above. [18] The program used to modify the titanic dataset is in this project's GitHub repository and is called "edittitanic.py" [3].

After the appropriate data modifications were made, the dataset could be imported into the system. The dataset contains 892 cases, making it almost 9 times larger than the animals dataset. After removing some features, 6 features remained. They were:

- Pclass – the passenger class of the passenger. The values ranged from 1 (the highest class) to 3 (the lowest class)
- Sex – 0 for male, 1 for female
- Age
- SibSp – number of siblings/spouses aboard
- Parch – number of parents/children aboard
- Fare – cost of ticket

The target values were "lived" or "died".

Firstly, 10-fold cross validation was used to check that a high accuracy could be obtained for this dataset (using option 13 from the menu).



```
Accuracy using 10-fold cross validation:
Decision tree: 0.763
Random Forest: 0.807
XGBoost: 0.806
```

***Figure 48** - 10-fold CV for the titanic dataset*

The accuracy was quite high for the three ML algorithms, meaning making meaningful rules based off the "black box" predictions was reasonably possible on the dataset.

Next, the feature importances were used. These are the feature importances given by the random forest using MDI (using option 9 from the menu).

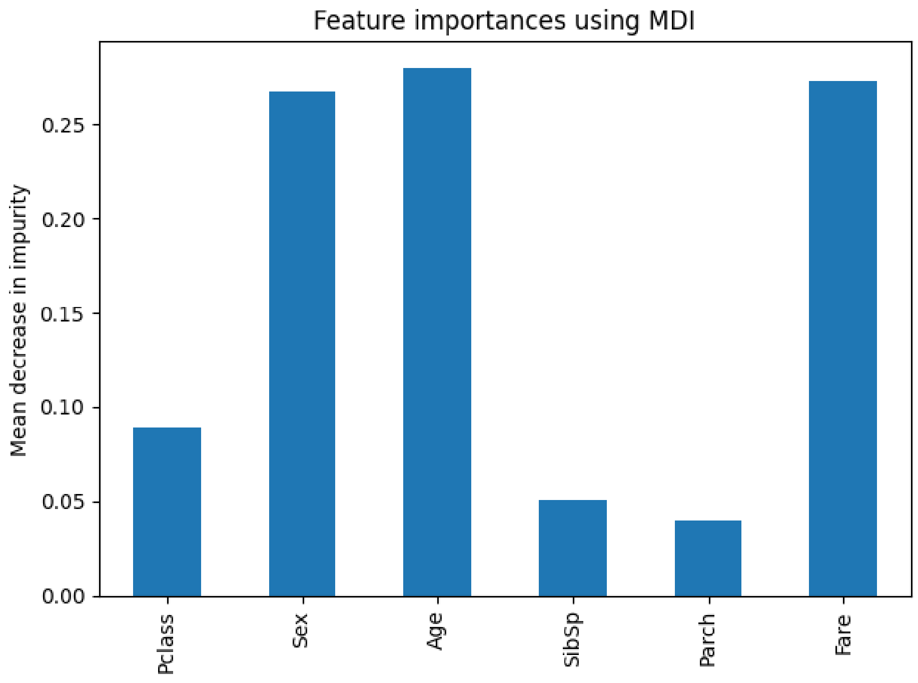
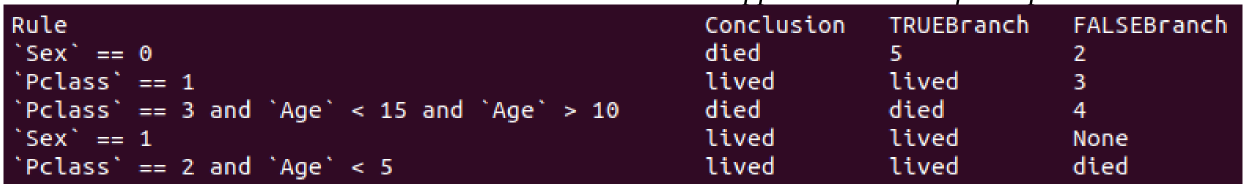


***Figure 49** - feature importances (from random forest) on titanic dataset*

The random forest believes that Sex, Age and Fare are important features, with Pclass being a somewhat important feature.

Running the same for XGBoost and without showing the resulting graphs, for attributes weight and gain, Sex clearly the most important while for cover, Age was most important.

Using these hints accordingly, the following 5 rules were created for the titanic dataset.

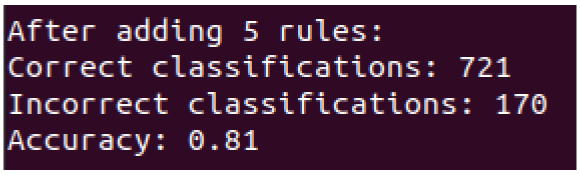


```
Rule                                              Conclusion   TRUEBranch   FALSEBranch
`Sex` == 0                                        died         5            2
`Pclass` == 1                                     lived        lived        3
`Pclass` == 3 and `Age` < 15 and `Age` > 10       died         died         4
`Sex` == 1                                        lived        lived        None
`Pclass` == 2 and `Age` < 5                       lived        lived        died
```

***Figure 50** - 5 rules created for the titanic dataset*

Note the use of "and" statements in these rules. These statements were necessary for a more complicated dataset such as the titanic dataset as opposed to the animals dataset where they were not required.

Running these 5 rules on the entire dataset and printing the accuracy (using option 12 from the menu) gives:



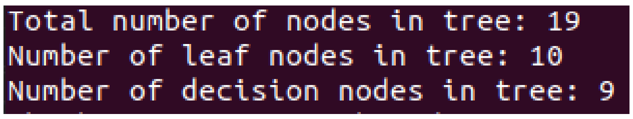
```
After adding 5 rules:
Correct classifications: 721
Incorrect classifications: 170
Accuracy: 0.81
```

***Figure 51** - accuracy of RDR rules on dataset*

As shown by the accuracy value in Figure 51, after only adding 5 rules, the RDR system actually gives a greater accuracy than the 3 black box algorithms.

5.3 The number of rules added

Another evaluation step undertaken was comparing the number of rules made by an RDR system to the number of decision nodes in the corresponding decision tree. For the animals dataset, the number of nodes in the decision tree was 9 which was slightly lower than the 13 rules in created in the full ruleset which correctly classifies the entire dataset (from Figure 43).



```
Total number of nodes in tree: 19
Number of leaf nodes in tree: 10
Number of decision nodes in tree: 9
```

***Figure 52** - number of decision tree nodes*

Considering that the decision tree is optimised to be as small as possible, and the ruleset that was created was not optimised, the slightly higher number of rules in the RDR system is insignificant. A real expert in animals who was trying to make the smallest ruleset possible may be able to match or even go lower than the decision tree's number of decision nodes. This comparison shows that only slightly more logic than a decision tree is required to create an explainable RDR system.

5.4 The effect of adding rules – learning curve evaluation

Another evaluation conducted on the system was a learning curve evaluation which shows the effect of adding rules. The set up for the learning curve evaluation is shown in the diagram below.

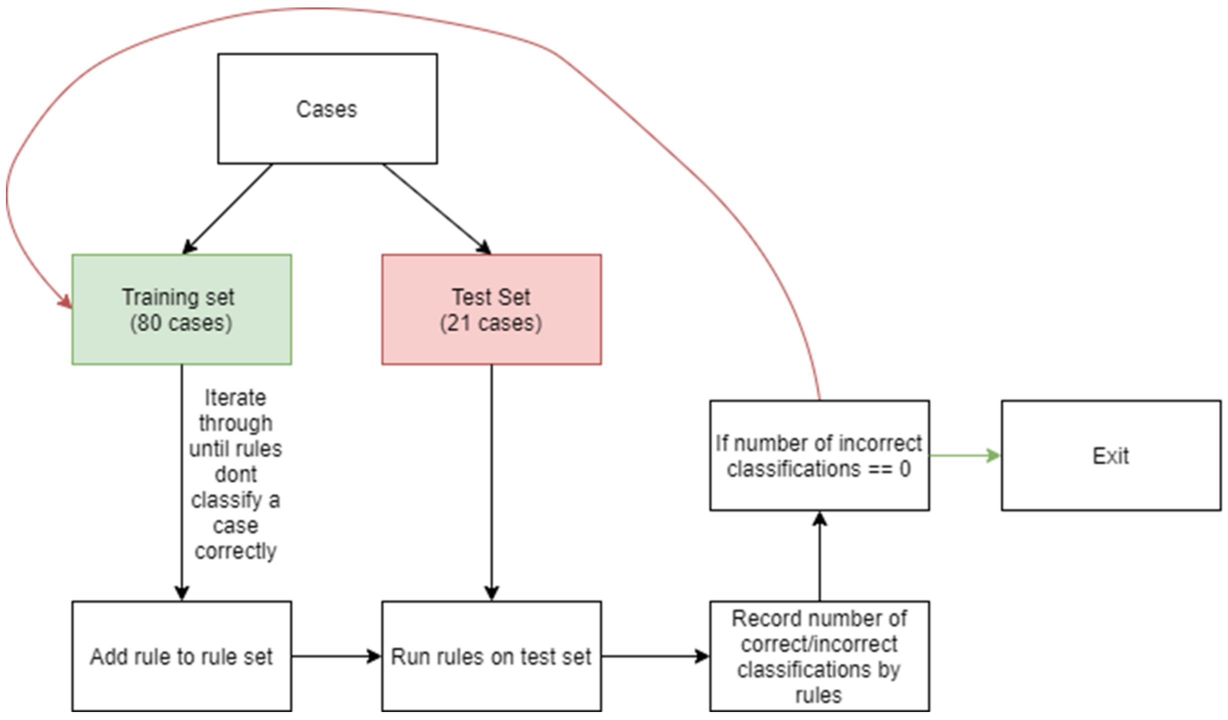

Cases
Training set (80 cases)
Test Set (21 cases)
Iterate through until rules dont classify a case correctly
If number of incorrect classifications == 0
Exit
Add rule to rule set
Run rules on test set
Record number of correct/incorrect classifications by rules

Figure 53 *- set up of learn curve evaluation*

The rules are run on a training set until a case is incorrectly classified – which is when the user will add a rule to correctly classify it. The ruleset is now run on a test set recording the accuracy each time a new rule is added. If the accuracy is 100%, exit, if now, more rules must be added until 100% accuracy is reached on the test set. Option 11 in the menu screen provides this workflow. The result of running this evaluation on the animals dataset is shown below.

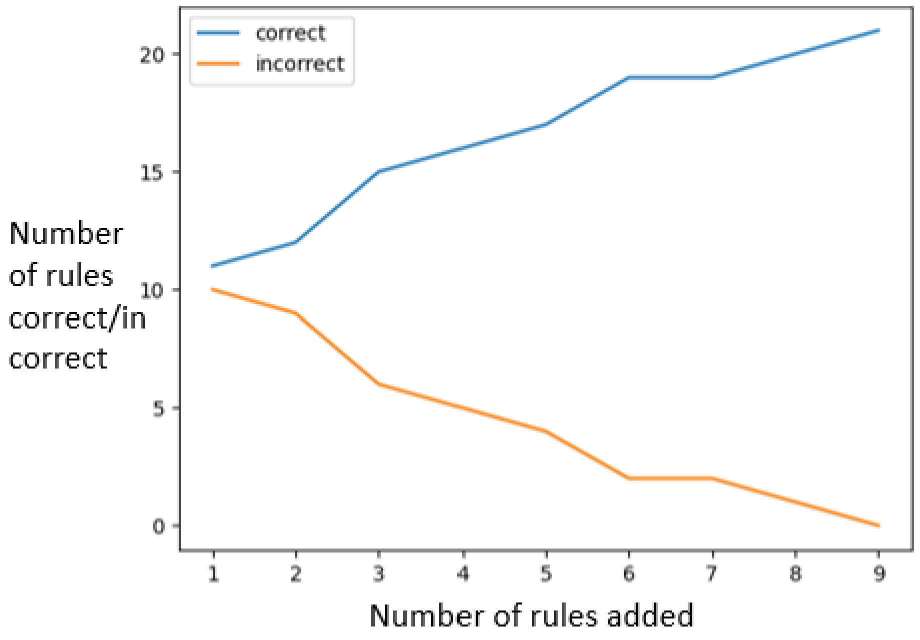


Figure 54 *- learning curve evaluation for the animals dataset*

As can be seen, when there is zero or a low number of rules in the ruleset, adding rules causes a large leap in accuracy for the RDR. However, as the number of rules added increases, the increase in correct classifications is small as these rules generally deal with edge cases. This evaluation shows that the effect of adding only a few rules is a decently high accuracy, and as the number of rules increases, the increase in accuracy increases but at a slower rate than when there were only a few rules.

Chapter 6

Conclusion

This report has demonstrated that RDR can be used for explainable AI. The ruleset that is created by an expert human in the loop user is explainable and can be used as an interpretable model to explain the predictions of a black box classifier. This report has outlined the motivations behind explainable AI itself, as well as motivations for using RDR for explainable AI. It has described background knowledge required for the project, and the methods used to prove that RDR can be used for explainable AI.

The system created to prove the hypothesis uniquely combines machine learning “black boxes” alongside a human based RDR system to form a singular system used to create an interpretable model to explain a black box classifier. Using a human in the loop distinguishes this method of explainable AI from other algorithmic based methods as humans can understand human made models, whilst humans can not always understand models created by algorithms and AI.

The system also provides helpful features to the user such as hints including feature importances as well as methods to obtain accuracy and evaluation feedback about both the RDR rules and machine learned models.

The results from using the system on multiple datasets proved that RDR can indeed be used for Explainable AI.

6.1 Further work

Unfortunately, there were some tasks for this project which were planned to be completed, but were either not completed or were incomplete. There is some possible further work that can be completed to make this RDR explainability system more accessible. There is also further work that could be done in evaluating this explainable AI method. These tasks are:

- **Frontend UI** – a frontend UI was planned to be implemented for this project. A UI would have made the system easier to use due to the ability to add UI elements that help users such as dropdown boxes for selections. It would especially be beneficial to "no code" users who have little to no programming experience as a UI can be used by people with no programming experience. Features such as dropdown boxes would also decrease errors in the system, as users are limited to correct options only, as opposed to a command line where anything can be typed – meaning typos by a user may cause the system to crash. A UI could also allow easier and more informative displays (including images and colours) to be shown to the user instead of a terminal full of text. The reason that a UI was not completed was that time was better spent on implementation of the system itself instead of a UI which adds no additional features
- **Integration of LIME** – an integration of LIME would have allowed an evaluation of the explainability of the RDR system by comparing it to a current RDR system LIME. It would have also allowed a comparison between a real-world explainable AI application and this system to see how effective the system was and also how effective RDR can be for explainable AI. However, there was no time to include such an integration or comparison to occur.
- **Using the RDR system to lift the performance of ML classifiers** – as seen in the titanic dataset example, the RDR ruleset actually had higher accuracy

than all three “black box” models. A possible use for this system would be to wrap and an RDR system similar to the one described in this project, where rules can be made to correctly classify cases which are misclassified by the ML classifier.

- **Using more datasets** – further evaluation of the system would include using more datasets on the system. The development of the system has only covered 2 datasets, so there is no doubt that there is more to learn about the system with the use of even more datasets.

Bibliography

[1] P. C. a. B. Kang, RIPPLE-DOWN RULES: The Alternative to Machine Learning, 2021.

[2] S. S. a. C. G. M.T. Ribeiro, “"Why should I trust you?": Explaining the Predictions of Any Classifier,” 2016.

[3] N. Driscoll, “rdr_python,” [Online]. Available: https://github.com/nathand99/rdr_python. [Accessed November 2021].

[4] A. Bachinskiy, “Towards data science: The Growing Impact of AI in Financial Services: Six Examples,” 21 February 2019. [Online]. Available: https://towardsdatascience.com/the-growing-impact-of-ai-in-financial-services-six-examples-da386c0301b2.

[5] G. Zochodne, “What's in the black box? Rise of artificial intelligence in banking presents challenge for OSFI,” 6 April 2021. [Online]. Available: https://financialpost.com/news/fp-street/whats-in-the-black-box-rise-of-artificial-intelligence-in-banking-presents-challenge-for-osfi. [Accessed May 2021].

[6] toolkitai, “Pitfall of Black Box AI at Banks: Explaining Your Models to Regulators,” [Online]. Available: https://www.tookitaki.ai/news-views/pitfall-of-black-box-ai/. [Accessed May 2021].

[7] General Data Protection Regulation, “What is GDPR, the EU’s new data protection law?,” 2018. [Online]. Available: https://gdpr.eu/what-is-gdpr/. [Accessed May 2021].

[8] T. Mitchell, Machine Learning, 1997.

[9] xgboost, “Introduction to Boosted Trees,” XGBoost, [Online]. Available: https://xgboost.readthedocs.io/en/stable/tutorials/model.html. [Accessed November 2021].

[10] scikit learn, “sklearn.tree.DecisionTreeClassifier,” [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html. [Accessed November 2021].

[11] s. learn, “sklearn.preprocessing.LabelEncoder,” [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html. [Accessed November 2021].

[12] s. learn, “sklearn.tree.export_graphviz,” [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.tree.export_graphviz.html. [Accessed November 2021].

[13] s. learn, “sklearn.ensemble.RandomForestClassifier,” [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html. [Accessed November 2021].

[14] XGBoost, “Python API Reference,” [Online]. Available: https://xgboost.readthedocs.io/en/latest/python/python_api.html. [Accessed November 2021].

[15] scikit learn, “Feature importances with a forest of trees,” [Online]. Available: https://scikit-

learn.org/stable/auto_examples/ensemble/plot_forest_importances.html. [Accessed November 2021].

[16] J. Brownlee, “Feature Importance and Feature Selection With XGBoost in Python,” [Online]. Available: https://machinelearningmastery.com/feature-importance-and-feature-selection-with-xgboost-in-python/. [Accessed November 2021].

[17] Kaggle, “Titanic - Machine Learning from Disaster,” [Online]. Available: https://www.kaggle.com/c/titanic. [Accessed November 2021].

[18] Pandas, “pandas.DataFrame.fillna,” [Online]. Available: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.fillna.html. [Accessed November 2021].