Software Engineering Group Project

COMP2002/G52GRP: Final Report

Project Information:

Project Title: Results Visualisation

Project Sponsor : UoN

Academic Supervisor: Andrew Parkes

Team Information:

Team Number: 44

Team Members:

Kieran Patel [20371226] psykp1 (Team Leader)
Ethan Temple-Betts [20354385] psyet5 (Team Admin)
Nathan Burgess [20363169] psynb7 (Git Admin)
Dominic Cripps [20334929] psydc8
Alfred Greenwood [20281530] psyag7
Daniel Ferring [20393362] psydf3
Dixon Okoro [20363242] psydo4

Documentation (links):

- Trello Board (Project Management)
- Trello Board (Kanban)
- Code Repository
- Meeting Minutes
- Full Requirements Document
- Document Repository

Contents

Main Report

1.1 Project Introduction and Background

Artificial intelligence is one of the most impactful and fastest growing areas of Computer Science. It is also one of the least understood. Addressing this issue is the main aim of this project: to take AI models and represent them in an informative and understandable way. Our project focusses on Decision Tree models but could easily be expanded to encompass other machine learning models by a future software engineering team.

The primary audience of this project is students. As students ourselves, we witnessed the lack of intuitive tools for visualising AI models during our first year AI module. Our project addresses this issue by providing a tool that generates interactive, easy to understand visualisations of decision tree results.

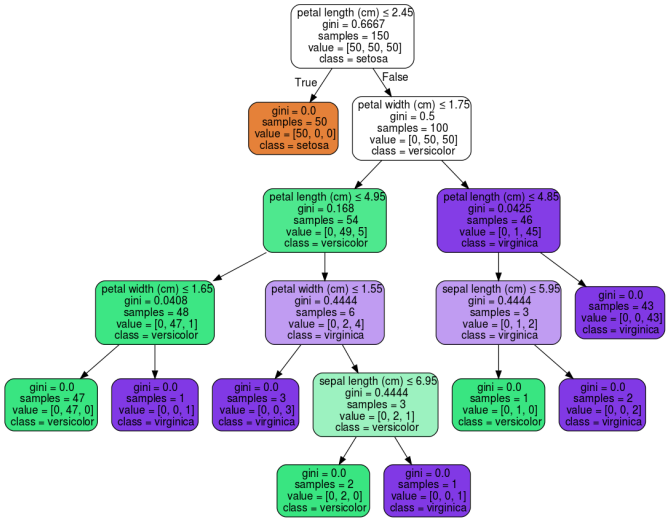


Figure 1.1: A standard Decision Tree structure produced by the sklearn library

Understanding this project requires an understanding of decision trees themselves. A decision tree (Figure 1.1) is a machine learning model that attempts to make a classification given a set of data by traversing a tree, taking different nodes depending on certain criteria: for example petal length > 5. This will eventually reach a leaf node which gives a classification. One way of interpreting the results of a decision tree's classification is a decision boundary. This is an analytical tool used to compare the correlation between two features. It partitions the feature space by creating a line that separates the data points according to their likely classification.

As part of our research for this project we looked at the following existing systems for results visualisation:

- The standard sklearn visualisations
- dtreeviz, a bespoke Python library for decision tree visualisations
- plotly and matplotlib, Python libraries for more general data visualisations

Of these, the most useful was dtreeviz which takes in trained decision trees and outputs a graphical representation of the tree as an image. The user is able to select options for the visualisation through lines of Python code.

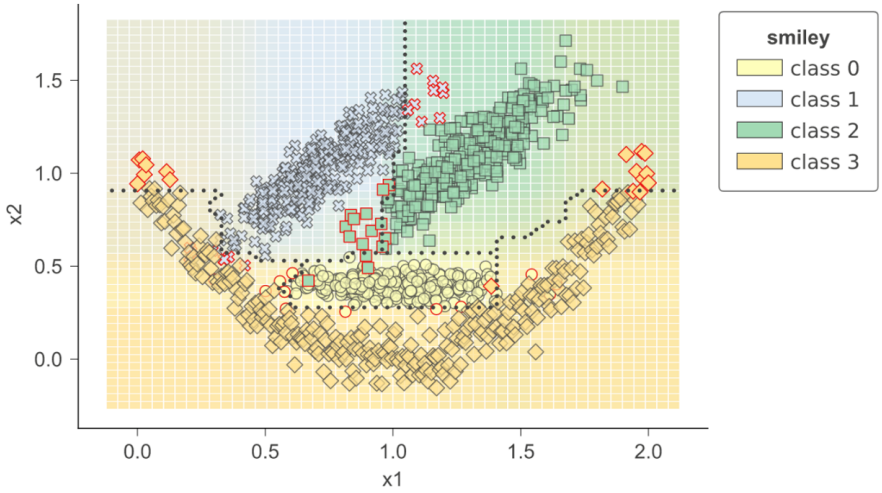


Figure 1.2: A Decision Tree Feature Space as visualised by dtreeviz

Looking at Figure 1.2, there are key strengths to this approach. The use of different colours and shapes for instances of each class makes it obvious how the data has been classified. We also appreciate the graphical representation of the feature space; we feel that it is the best technique for representing trees in one or two dimensions.

There are, however, areas in which dtreeviz is lacking. Firstly, it is unclear how the displayed decision boundaries were created, something that could cause confusion for learners. The system also lacks any interactivity: in Figure 1.2, the instances are clumped together in a way that makes it difficult to interpret the data. This issue could easily be solved by allowing the user to zoom into and pan around the feature space. Furthermore, dtreeviz requires the user to use and have knowledge of other Python libraries such as matplotlib. This increases the prerequisite knowledge required to use the system which would be off-putting to beginners.

1.2 Initial Requirements Analysis

ID	User Story or Quote From the Brief	Requirement	Priority
	Functional		
1	"build a system that will be able to take the decision trees produced by some standard "off-the-shelf" machine learning system"	The user must be able import their off the shelf model and data into the system	HIGH
2	As a student, I want to be able to fully navigate around all 1D, 2D and 3D graphs – including zooming, traversing, and rotating - so that I can better understand the feature space.	The user must be able to navigate through the feature space	HIGH
3	As a student, I want to be able to see the decision boundary so that I can identify data instances that are close to being assigned a different classification.	The user must be able to clearly visualise decision boundaries and classifications	HIGH
4	As a student, I want to be able to see the various weak models used to create the ensemble model so that I can understand the resulting classification.	The user must be able to compare models from the same ensemble-based classifier	HIGH
5	As a student, I want to be able to see the areas of weak consensuses within my ensemble model so that I can assess the validity of the final model.	The user should be alerted to areas of weak consensus	MEDIUM
6	As a student, I want to view the decision boundaries drawn by a SVM model in the feature space, so that I can evaluate whether SVM was a good method for my classification task.	The user should be able to view the decision boundary made by their SVM model	MEDIUM
7	As a student, I want to be able to hover over data instances and be shown a summary of its details, so that I can better understand my dataset.	The user should be able to hover over data instances and be shown technical information pertaining to that instance	LOW
8	As a student, I want to have the ability to test my machine learning model to check if it has a monotonic feature so that I can explain to customers and colleagues why the model classifies data in a certain way.	The user should be able to check for monotonic features	LOW
9	"show the data points along with their actual or predicted class labels"	The user should be able to distinguish between the actual and predicted class of data instances where supervised learning has been used	LOW
	Non-Functional		
10	As a student, I need the system to accurately portray the data so that I can have confidence in my results.	The system must accurately portray the provided machine learning model	HIGH
11	As a student I want the system to be quick and responsive so that I can work more efficiently.	The system must interpret models efficiently in real time	HIGH
12	As a student I want the system to be easy to understand so that I don't have to spend a lot of time trying to understand the technology.	The system should be simple to use and understand	MEDIUM

Figure 1.3: A selection of the initial project requirements grouped by functionality

Upon considering how best to gather requirements, it became evident that surveys, interviews and questionnaires were not a necessary part of this process. This is because we fall into the category of an end user as we are all students. We therefore concluded that these would not further our knowledge of what is required of the system.

We found performing a textual analysis on the brief to be a more appropriate task and drew upon our own experiences in the first year AI module to produce a backlog of user stories. Our initial analysis for the pitch was then used in conjunction with these user stories to inform the requirements for the project. The requirements were then categorised based on their perceived priority and whether they were functional or non-functional as can be seen in Figure 1.3.

We used this to inform our choice of technologies for the project, so that we could begin rapid prototyping. At first we considered the use of JavaFX, as it allows for the creation of highly customizable interfaces. Once we began developing a prototype, however, it became clear that Java was a poor choice for the project due to the amount of overhead required when working with the machine learning models. This would be detrimental to the performance and consequently the functionality of the final application.

With performance and simplicity in mind, we turned to Python. Due to its popularity for both machine learning and visualisation, there are a range of libraries available that make development more rapid and streamlined. For example, the scikit-learn library supports all of the desired decision tree model types. This simplified the development process significantly as we only had to make the system work with one library, saving time without sacrificing functionality.

Because of the project's focus on visualisation, the overall visual clarity of our design is of the utmost importance. For this reason we began to discuss the benefits of making a web app instead of a desktop application. We quickly agreed that the functionality and appearance of the final application would be vastly improved by this transition. Although the initial brief was aimed at the production of a desktop application, we brought the idea to our supervisor, Andrew Parkes, and he agreed to it.

After researching technologies for web development and considering options like Flask and Django, we decided that the Dash Python library was the best fit for the project. Dash is specialised for the production of interactive visualisation website dashboards. It integrates Plotly, a Python library that facilitates the creation of highly customisable visual aids such as graphs and charts. Dash will also allow us to rapidly develop new features without having to spend large amounts of team resources on creating specialised front and back ends. This is because Dash integrates Flask and automatically generates the JavaScript required to make fully functioning dynamic web dashboards.

It was therefore decided that we would begin development using Dash and Plotly for the visualisation and the pickle and scikit-learn libraries for the machine learning models themselves.

1.3 Initial Software Implementation

To demonstrate the feasibility of our vision for the project, we chose to develop a prototype that covered the basic requirements of the desired system. The result of this was a web application that can import a CSV data file and display the data feature space in one, two or three dimensions and provides a visualisation of the decision boundaries of a two-dimensional tree.

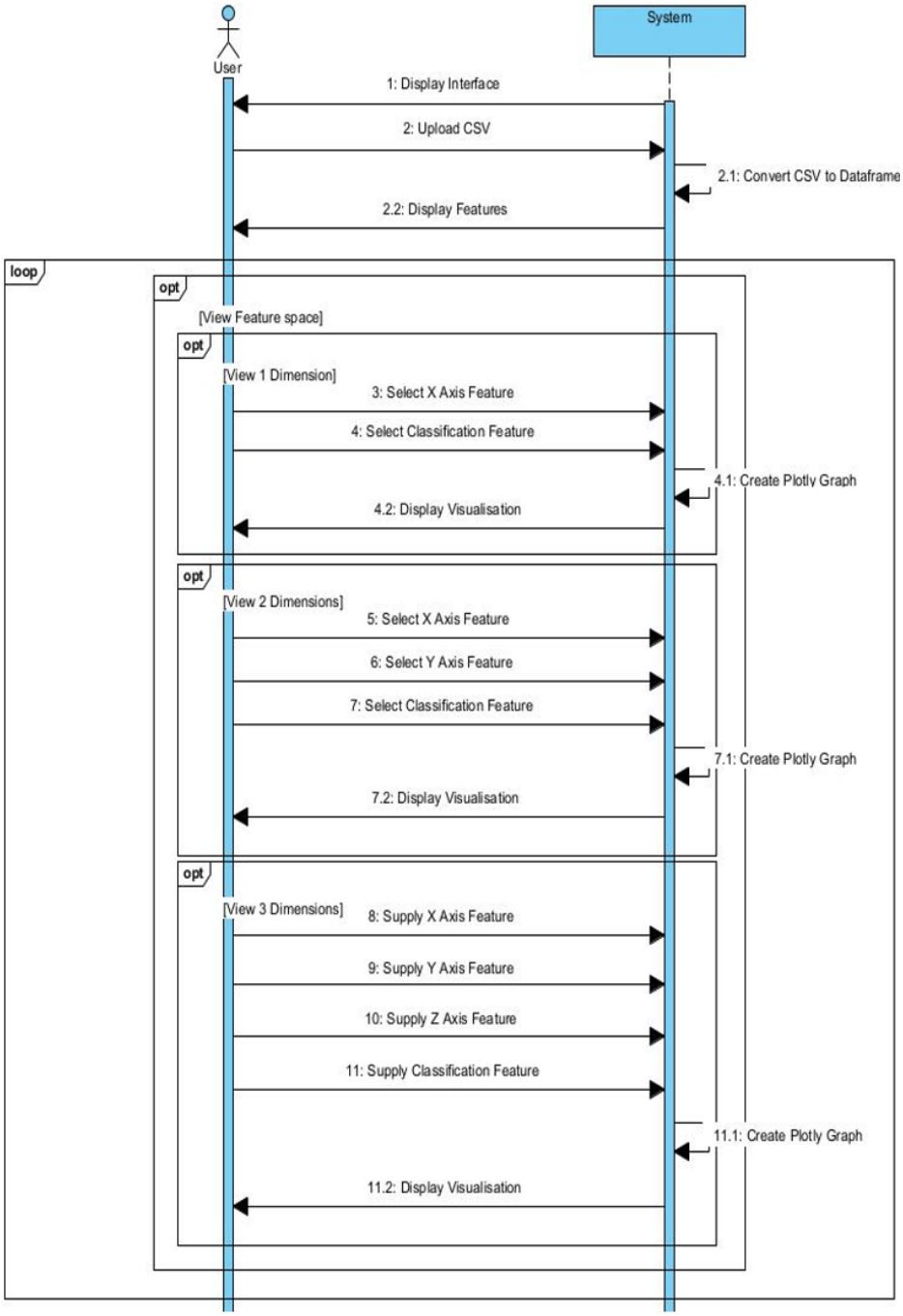


Figure 1.4: A Sequence Diagram detailing the program flow for importing a CSV file

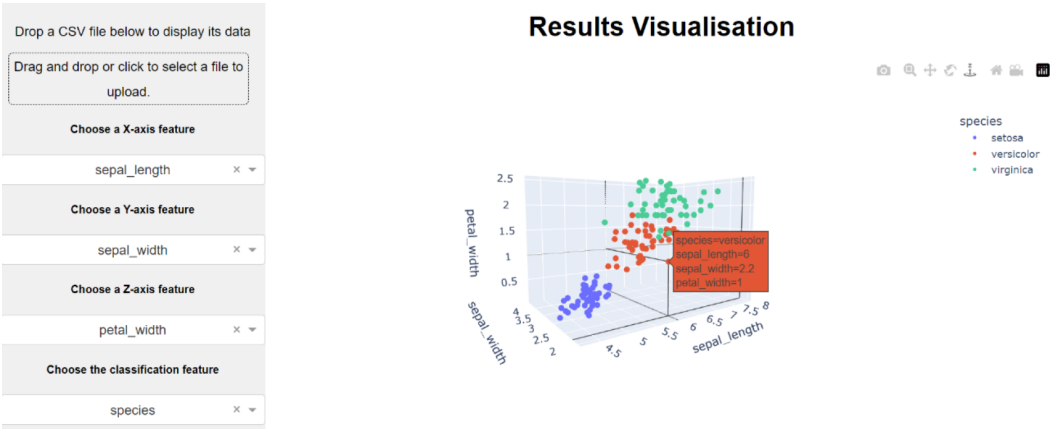


Figure 1.5: The feature space of the iris data set, as displayed by the prototype

As demonstrated in Figure 1.4 and Figure 1.5, the prototype provides a simple and informative interface for importing CSV files. The panel at the left side of the screen allows the user to drag and drop any CSV file into the system. The drop-down menus below are then configured with the column headers from the file, allowing the user to select any feature for each axis. The classification feature can then be used to colour the data instances based on the chosen feature. The user is also able to hover over data instances to display information about them. The prototype also supports moving through the feature space by panning and zooming.

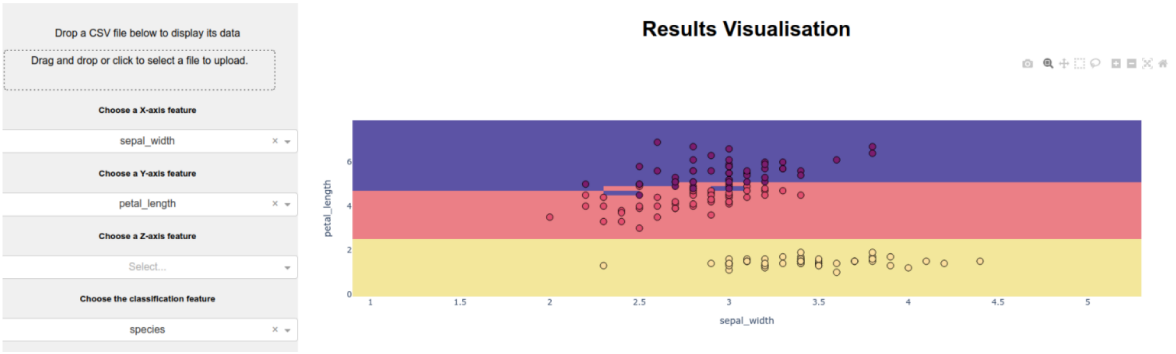


Figure 1.6: A visualisation of the decision boundaries of a two dimensional decision tree

Figure 1.6 shows an example of the prototype displaying the decision boundaries of a tree trained on two dimensions. The background colours represent the classifications made by the model while the colours of the instances indicate their actual classification. This enables the user to gain an insight into how accurate their model is. The options on the left allow the user to select the features for which they would like to see the decision boundaries.

This prototype was the culmination of our effort throughout the first term. While it lacked many of the required features, it proved the viability of the project. We had shown that the technologies we elected to use were suitable and that as a team we were capable of delivering a functioning web app that fulfilled many of the primary requirements. Although much of the code behind it was rewritten for the current implementation, many of the design decisions present in the initial prototype were carried over into the current implementation. In this regard, the prototype served as a foundation upon which the current software was built and so, despite the limited functionality, it can be viewed as a successful stage of the project.

1.4 Evolution of Requirements

The basis of our initial requirements came from our project supervisor. The project brief and our discussions with him enabled us to produce a comprehensive list of potential requirements. These were primarily focused on making AI methods "explainable." That is, the use of visualisation techniques by which we could show how the data was processed by the models. The aim being to aid the user, both in interpreting the results themselves and in comprehending of the process by which those results were reached.

The initial requirements were quite broad, describing a desktop application aimed at learners, businessmen and AI researchers that provided features including:

- The ability to visualise the results of various decision tree models, including those produced by ensemble learning, regression and SVM methods
- Interactive visualisations to allow for easier interpretation by providing actions such as panning and zooming
- Visualisations of decision boundaries that could support trees of 3 or more dimensions
- A technique for identifying and visualising monotonic relationships present in the models
- The option to input a new data instance and be shown its classification
- The ability to import pre-trained models from a variety of machine learning libraries
- (As a stretch goal) Support for Artificial Neural Networks

As development progressed, our understanding of the theory and technologies of the project grew and we became more aware of how much work was possible in each sprint. As a result of this we realised that we were not going to be able to deliver all of the requirements; some for practical reasons, others due to time constraints. This provoked an evolution of our requirements in which we shifted the primary focus of the project onto just one of the identified target audiences: learners. The reasoning behind this was that, by dropping some of the more niche requirements, we would have enough time to create a comprehensive and cohesive program aimed at a specific user base. The alternative would have been an incongruous jumble of features that would have left each target group unsatisfied.

The changes in requirements can therefore be placed into two categories: those made for practical reasons and those made as a result of the shift in focus.

Practical Changes
The first practical change was the decision to make a web-app instead of a desktop application, the reasoning of which is discussed in Section 1.2. Having settled on a web-app, our initial vision was to have it hosted as a website as this would make it incredibly easy for users to access it. However, this led to a number of issues (See Section 1.5.3), the foremost being the trade-off between performance and expense. There are excessive hardware limitations for cheaper plans, to the point where the app could barely run. Paying for a more expensive plan would only offset the problem: any concurrent use, and the performance would again crumble. As such, we settled on producing both a website, meant for demonstration only, and a locally hosted containerised version of the application for serious use.

The other main practical change pertained to the requirement for visualising the boundaries of trees with more than two features. This is a challenging task and we spent a considerable amount of time trying to come up with a solution. It eventually got to the point where the workload from other modules was picking up and we had to make a change if we wanted to deliver something usable within

the project time frame. We therefore decided to change the requirement to visually representing the results of higher dimension trees such that the user could infer the same information that they could have from a decision boundary. To this end, we made use of the parallel coordinates and pairwise plot visualisations (Section 1.5.1) which we prototyped and showed to our supervisor. With his approval, we proceeded to properly implement the visualisations in our system.

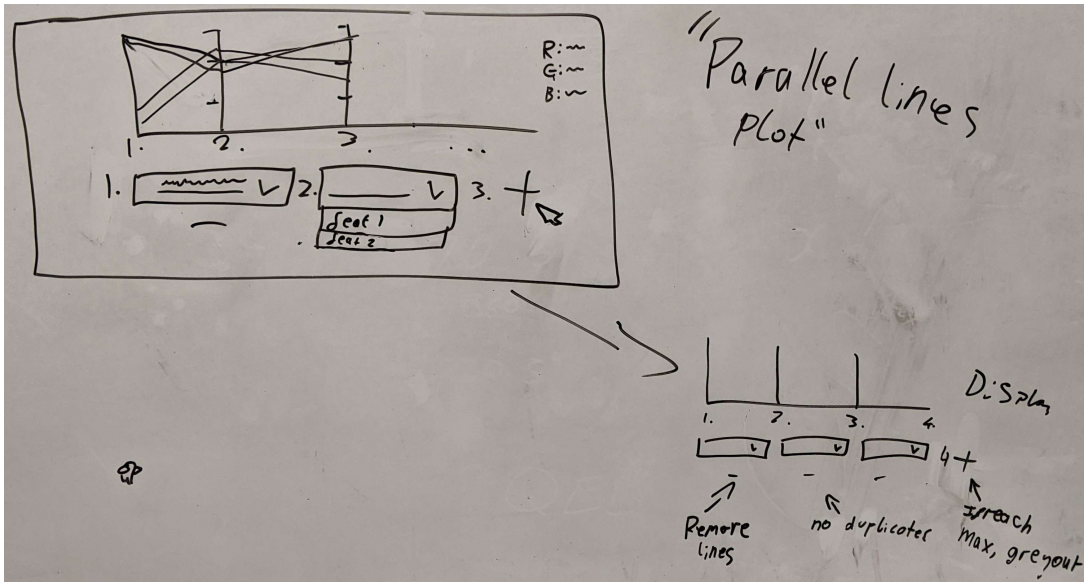


Figure 1.7: The initial design for the Parallel Coordinates component

Focus-Based Changes :
As a result of the shift in focus, and the need to adhere to the project timescale, we cut requirements that we deemed beyond the scope of initial learners. These choices were based on what was covered in our introductory AI module; while we included concepts beyond the scope of the module, we tried to ensure that they were at the same level of complexity. This led to the removal of: support for regression models, the stretch goal of supporting Artificial Neural Networks and the explicit identification of monotonic relationships. Doing this reduced the project scope to the point that it was viable to have it completed by the module deadline and it was therefore a necessary step in the development process.

Another change caused by this shift was to alter the requirement of importing pre-trained models to providing an intuitive, GUI-based mechanism for training models. The reasoning behind this was that it would remove a barrier for learners attempting to understand decision trees as they would no longer have to learn how to use a selection of libraries and could instead focus solely on interpreting results. This was also beneficial from a practical standpoint: the user would be able to retrain models with different parameters and observe the results without having to exit the program.

The shift also entailed the addition of certain requirements. The main addition was the introduction of Dash Tooltips to the system. The Tooltips provide information about each component when they are hovered over, communicating what they do and how to interpret them. The idea being that this would make the program more accessible and would save users having to search through the entire user manual any time they need information. Alongside tooltips, we added components related to the training of the model such as a confusion matrix to help the user interpret the results of their training parameters.

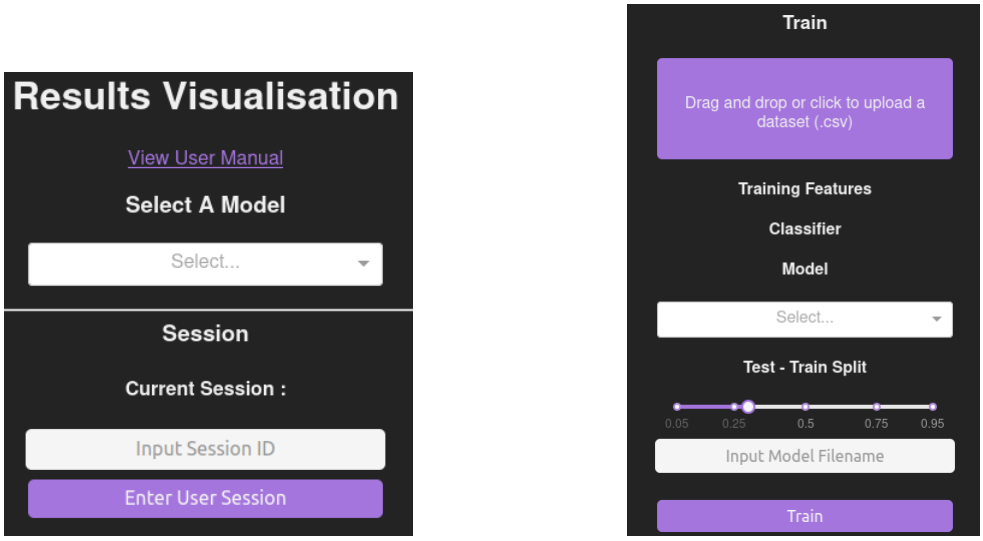
Our final requirements therefore described a web app designed for learners that provided the following general functionality:

- The ability to visualise the results of standard decision trees, as well as those produced by the gradient boosted, random forest and SVM methods.
- Interactive visualisations to allow for easier interpretation by providing actions such as panning and zooming
- Visualisations for the results of trees of 3 or more dimensions
- The ability to input a new data instance and be shown its classification
- The ability to train models and vary their parameters
- Useful descriptors to help learners interpret the visualisations

1.5 Current Software Implementation

1.5.1 System Features

The final web-app contains various features all of which are designed to help learners train decision trees and interpret their results.



(a) Model Selection

(b) Model Training

Figure 1.8: The Interfaces for model selection and model training

The system provides functionality for users to select previously trained models. The interface for this can be seen in Figure 1.8a. The user must also enter a Session ID before training models. This allows us to keep each user's models separate without needing to store their IP addresses. It also means that the website could be used as a collaborative tool, with learners working on the same session ID and comparing results.

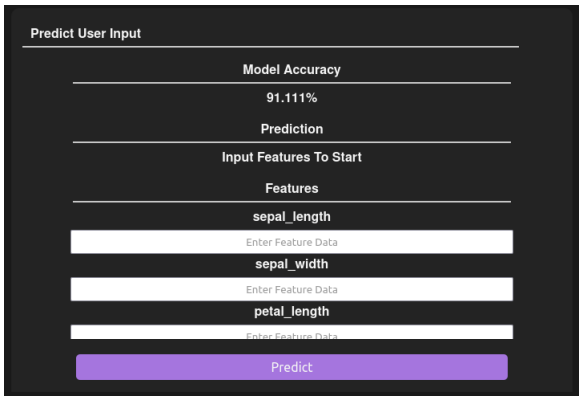
The generic part of the interface for training a model is shown in Figure 1.8b. Users are able to upload CSV data sets and can choose the features and classifier to be used in training. The user can then choose the type of model to train as well as the split between testing and training data. Once this has been entered, relevant adjustable parameters for that model type will appear below the Train button, to be set as the user sees fit. All user input is sanitised with error pop-ups being called for invalid inputs.

Once the user hits the train button, they will be presented with a series of visualisations to help them interpret the results and decision making of the trained model. For brevity, this section will only cover visualisations common to every model type and those used for standard decision tree models.

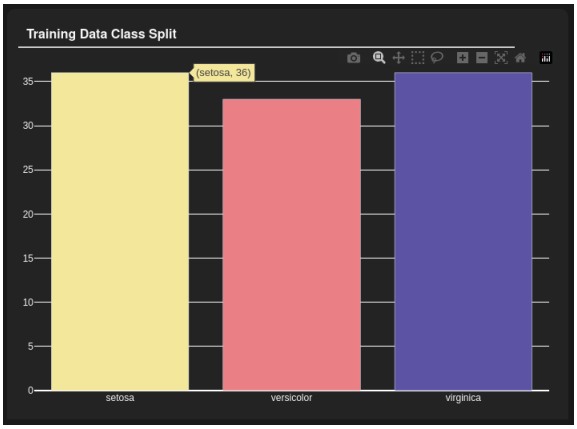
The ModelInfo component can be seen in Figure 1.9a. It provides the user with a summary of the trained model, showing general information like the classifications present in the model as well as the parameters used to train it.

Figure 1.9b shows the Predict User Input component. This displays the calculated accuracy of the model and allows the user to enter their own data instances in order to see how the model would classify them. This feature is particularly useful in the analysis of the decision making process as the user can vary the values for each feature and see how this impacts the instance's classification.

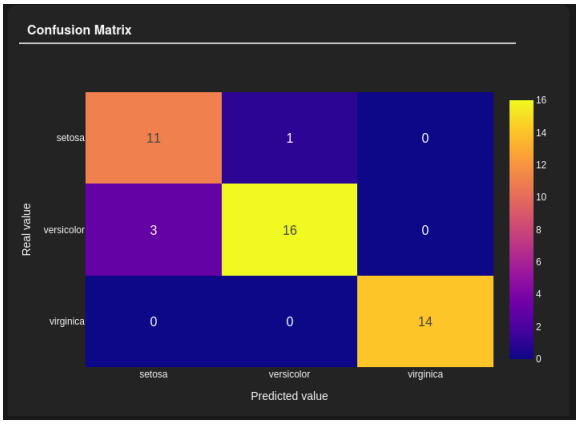
(a) The ModelInfo Component



(b) The Predict User Input Component



(c) The Training Data Class Split Component



(d) The Confusion Matrix Component

Figure 1.9: The first four components presented to the user

The confusion matrix, shown in Figure 1.9d, provides a visual representation of the model's performance. It displays the correct and incorrect predictions made by the model for each class, helpful for users who want to understand the model's accuracy and identify areas that need improvement.

The Training Data Class Split Component can be seen in Figure 1.9c, it provides the user with a view of the split of classifications in the training data. This is useful for analysing biases and inaccuracies in the trained models.

The Parallel Coordinates Component is shown in Figure 1.10. It shows the relationships between instance's feature values and their classifications. It is capable of supporting high dimensional models and allows the user to add, remove, and rearrange the order of features as they see fit.

The Model Decision Tree component can be seen in Figure 1.11. It displays the decision-making process of the model in a hierarchical structure, showing the gini value at each node the user hovers over.

The Decision Boundary component's functionality has been extended from that seen in the prototype (Section 1.3). For models trained on more than 2 features the user can now choose pairs of these features to be plotted so that they can visualise the relationships between them. The interface for this can be seen in Figure 1.12.

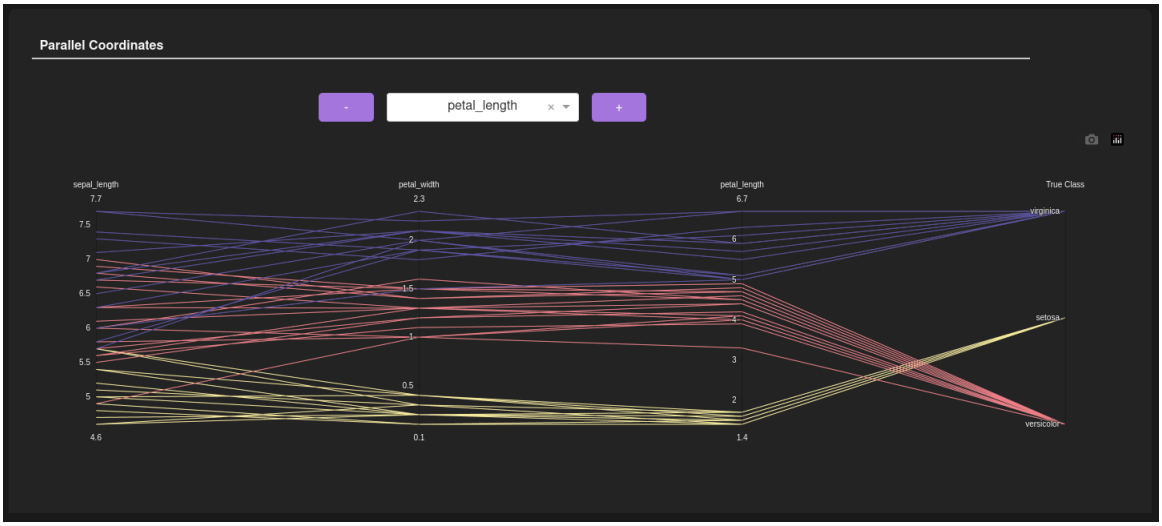


Figure 1.10: The Parallel Coordinates Component

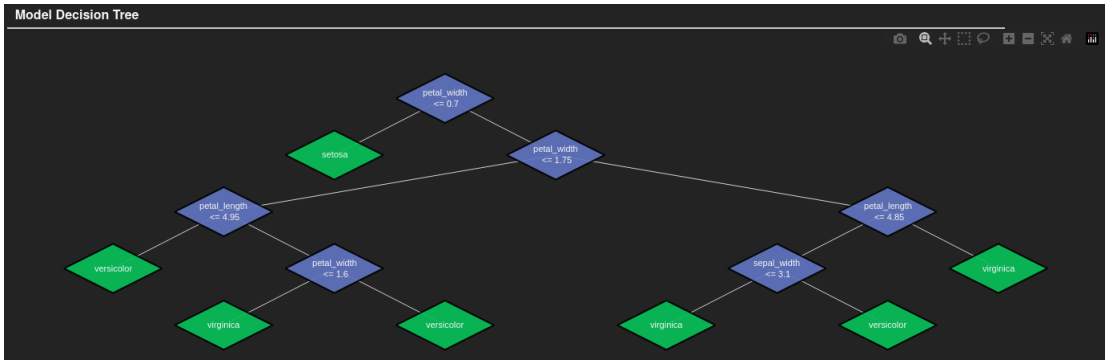


Figure 1.11: The Model Decision Tree Component

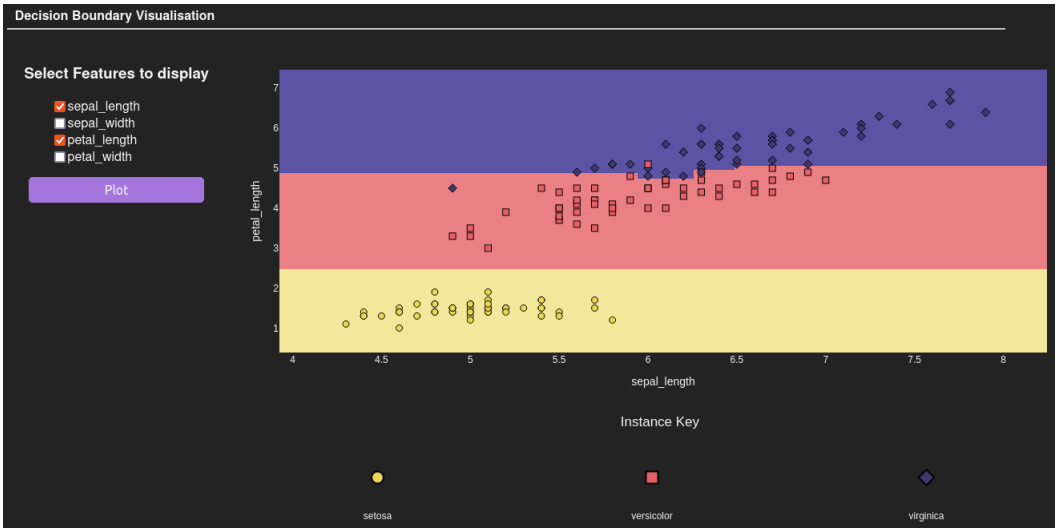


Figure 1.12: The Updated Decision Boundary Component

1.5.2 System Architecture

The program is designed to be modular and can be better described through decomposition into four main sections: 'Model Settings', 'Classifier Components', 'Dash Callbacks', 'Utility Classes'. The solution makes use of various software engineering concepts and design patterns, including the Single Responsibility Principle, Singleton Pattern, and Factory Pattern. The design promotes maintainability, scalability, and the use of design patterns makes it more flexible. Each class focuses greatly on specific tasks which increases code readability and promotes better testing.

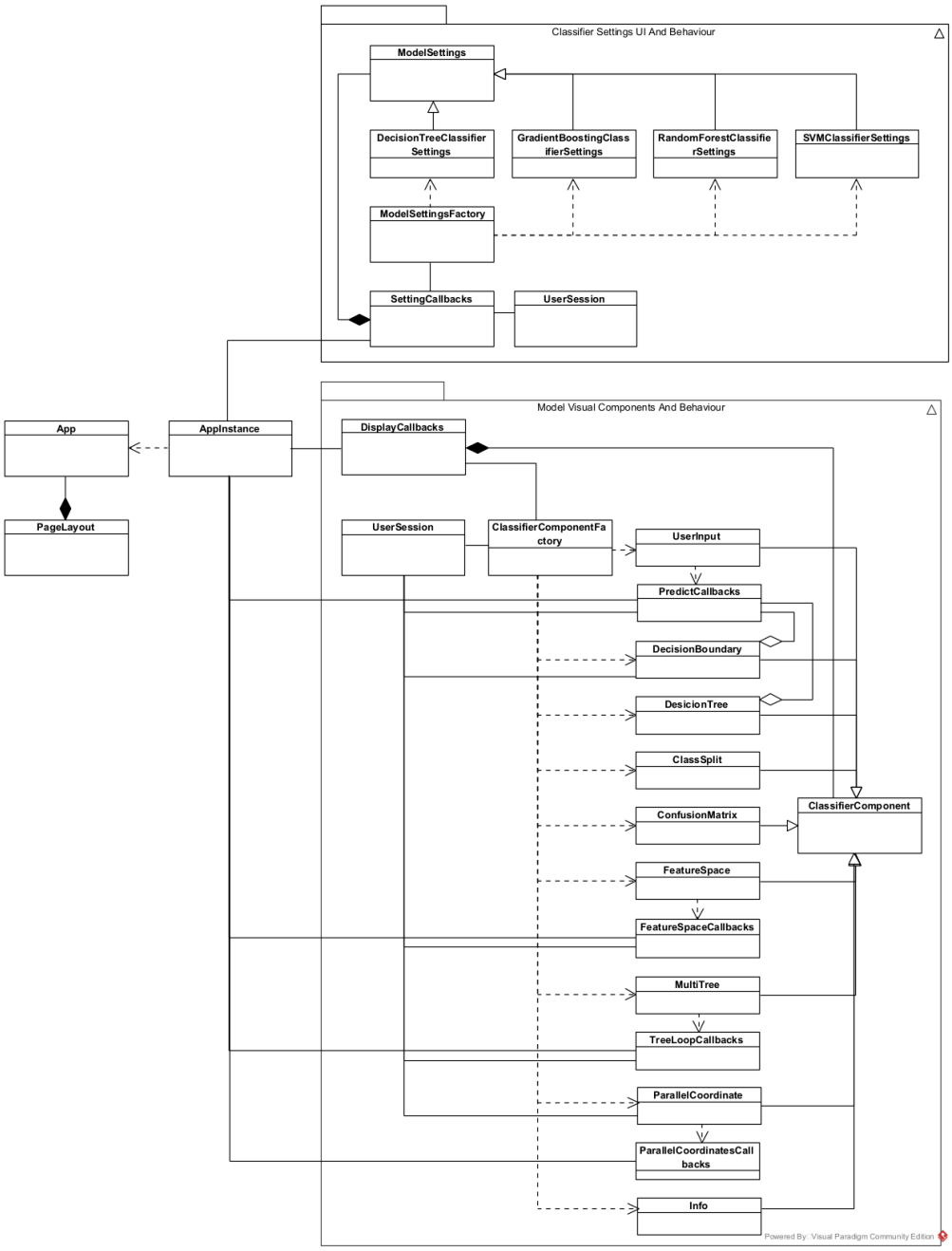


Figure 1.13: The class diagram for the overall system design

Model Settings :
The management of the training options presented to the user is the responsibility of the 'Model Settings' section. All classifier setting classes include three core properties: a reference to the classifier type, an HTML layout containing input elements, and an array of supported parameters. This repetitive structure lends itself to an inheritance relationship, as such the Factory Design Pattern is used to instantiate the appropriate model settings. This design choice was intentional and allows for the efficient implementation of new models.

Classifier Components :
Classifier Components provide visualizations for AI classification models. Each model type is provided a combination of components, each of which contain unique information. Whilst some components are specific to certain model types (for example the 'MultiTreeComponent' is unique to the Random Forest Classifier) many are generic and are used for all model types (for example the 'InfoComponent' which shows information such as the parameters used to train the model).

All 'Classifier Components' are made up of a label (title) that is displayed on top of the component, and an array containing the HTML structure of the component – this includes any diagrams, graphs, or figures. Utility classes support the creation of the components HTML, this helps maintain Single Responsibility. The Factory Design Pattern is used to instantiate the correct components for each model, simplifying the addition of new classifiers. By separating the page into rows and specifying which components should be displayed on those rows the 'ClassifierComponentFactory' dynamically sizes all components to fit onto the webpage.

Dash Callbacks :
Any interaction the user has with the website is done through a 'Dash Callback'; callbacks are linked to the HTML elements of the solution. Anytime an element is interacted with, an associated callback – if defined – will be triggered, and that element will be used as input. The return value of these functions are HTML components that will be displayed to the user.

The system uses callbacks to display components, interact with existing components, and train the selected model. The Dash 'App' is instantiated after the page layout is defined and as such 'Dash Callbacks' need to be defined after this. The class 'SystemCallbacks' handles the instantiation of all callback files; this not only separates these sections into different files, keeping the project organized, but also allows for new callbacks to be defined with ease.

'Pattern Matching Callbacks' take a variable number of inputs / outputs, this has been used to take only specified classifier arguments when training a model, and only output components to existing HTML elements. This has greatly simplified the implementation of callbacks.

Utility Classes :
Various 'Utility Classes' are used to support the creation of graphs, and HTML structures for example the decision tree. These 'Utility Classes' help simplify the 'Classifier Components' and keep 'Single Responsibility.' The functions defined in them are general enough to be of use elsewhere in the program, therefore avoiding repetitive code and keeping the components clean and maintainable.

A singleton class 'UserSession' allows the user to store multiple models and their relevant information in one persistent class, avoiding circular dependencies. This class is referenced by components when model information is needed, and by the 'SettingCallbacks' class to be updated after a new model has been trained.

1.5.3 Hosting

Current Solution :
Our current delivery system is comprised of two options: a local system and a remote system. The local system consists of a docker build file that allows you to build a container to locally host the application on your machine. This can then be accessed through a web browser using *localhost*. This approach ensures that the app will run on any machine and the use of containers avoids the user needing to setup a virtual environment or worry about dependencies. The remote system stores the app image on a Google Cloud Artifact repository which is in turn deployed on GKE (Google Kubernetes Engine). This allows for the app to be used as a remotely hosted website.

The remote system is primarily for demo use and can be used by users to test the system before installing the local version. It could also be suitable by users with very low end machines who only need to train models using small CSV files, but this is not recommended. This is due to the severe performance limitations of the current hosting solution: any CSV file over 500KB takes an inordinate amount of time to process. The local system is recommended for most users, the performance of which will be dependent on the specifications of the device it is run on. Both versions are easy to access and use, with minimal setup required, meeting the requirement for the program to be easily accessible.

Evolution of Hosting :
Originally, we wanted to pull directly from the repository onto an Azure app-service for remote hosting. We were successful in implementing this, but faced multiple issues while doing so. Firstly, to pull from the GitLab repository onto Azure would require a runner, something we didn't have at the time. As a temporary workaround, we mirrored the GitLab repository onto GitHub. This was done for two reasons: GitHub provides a runner and Azure natively supports CI/CD for Flask applications stored on GitHub.

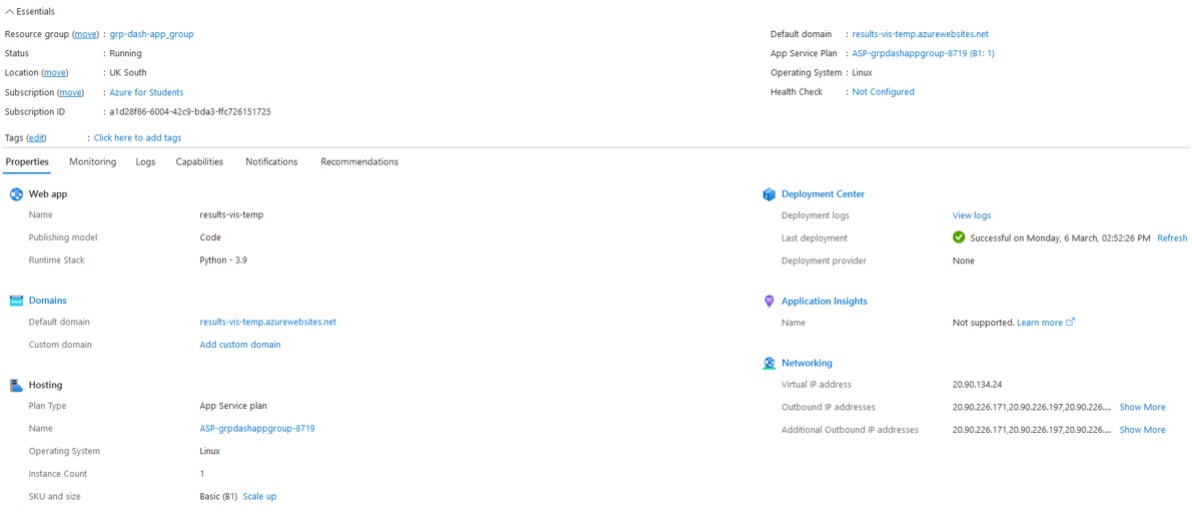


Figure 1.14: The Azure overview of the initial web application

The execution of the workaround was far from seamless. The use of Dash meant that the Flask server object was embedded within the Dash object which prevented Gunicorn from finding it. The application therefore had to be refactored to be compatible with Gunicorn. This also entailed the creation of a custom startup script which took a significant amount of time to get working.

Having implemented the initial hosting solution, the plan was to create a runner and adapt the system to work directly from GitLab. However, before this could be done, there were performance issues that needed to be dealt with: the application ran slowly and was incapable of handling concurrent use. This was due to the limitations of the remote hardware that the program was being run on.

To improve the responsiveness of the application we decided to move to a Google Cloud approach using their Google Kubernetes Engine (GKE). We still wanted to implement CI/CD for this, so we got the runner working by hosting it on a Google Compute Engine Linux VM. This allowed us to run our workflows from GitLab rather than GitHub. Like Azure, GKE required a containerised version of the application. Luckily, it was possible to reuse the aforementioned Gunicorn startup script and the refactored app structure greatly simplified the creation of the Docker build file.

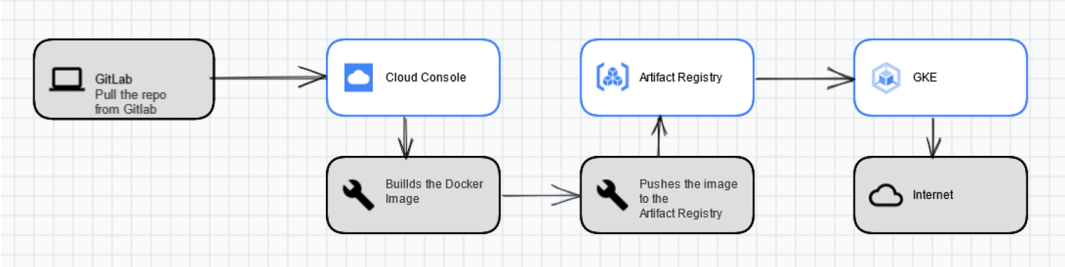


Figure 1.15: The process for uploading to GKE

We were then able to manually push a container to the Google Artifact Repository to allow other Google Cloud services access to our app image. We then created a GKE service with three pods onto which the app image could be pulled. This would allow us to do dynamic load balancing, a process in which more pods are added to the service as demand increases. This greatly improved the application's responsiveness as the resources allocated to the application were able to grow dynamically with demand.

Alternatively, we could have upgraded the hardware that the Azure design was on. However, it was decided that this would be too expensive for the marginal increase in performance it would provide.

Due to time constraints re-implementing CI/CD to Google Cloud was not something we were able to achieve. However, the GitLab runner we created rebuilt the docker container on each push to main. This allowed us to ensure that no changes to the project codebase broke the build system.

Most of the issues experienced during this process were down to two main factors. The first was a lack of experience with cloud hosting services, which resulted in the lengthy process listed above. If a team member had previously worked on hosting services, it is possible that these issues could have been predicted and avoided. The other issue came from Dash being a relatively new framework. Documentation outside of the official sources was scarce, and these sources did not cover all of the issues that we encountered. This lack of support material increased the difficulty of troubleshooting, significantly extending the amount of time it took.

1.6 Version Control

Our chosen tool for version control was Gitlab. To maintain coherent and consistent usage of git we created a guide outlining how to use issues, how to branch and merge and the required style for commit messages.

Throughout the project we made use of multiple branches, two of which were always in use. The first such branch was "main" which acted as our master branch and was only used for finalised code. As such, it was protected; only the git admin could merge into it. The other constant branch was "dev" which held the code that was currently being worked on. Branches for new features were made from "dev" and were merged back into it upon completion. The stability of dev was checked at routine intervals and, if it was satisfactory, the dev branch was merged into main.

Whenever a team member wished to edit the codebase the following process was followed. First the team member had to create an issue, regardless of the reason for change. Each issue had to have a relevant title and an in-depth description to outline what was being done and why. Additionally, issues needed to be associated with a milestone for the project. In cases where there was no relevant milestone for the issue being created, a new one was made. A branch for the issue was then made and, at the point of the issue's resolution, merged back into its parent branch and deleted.

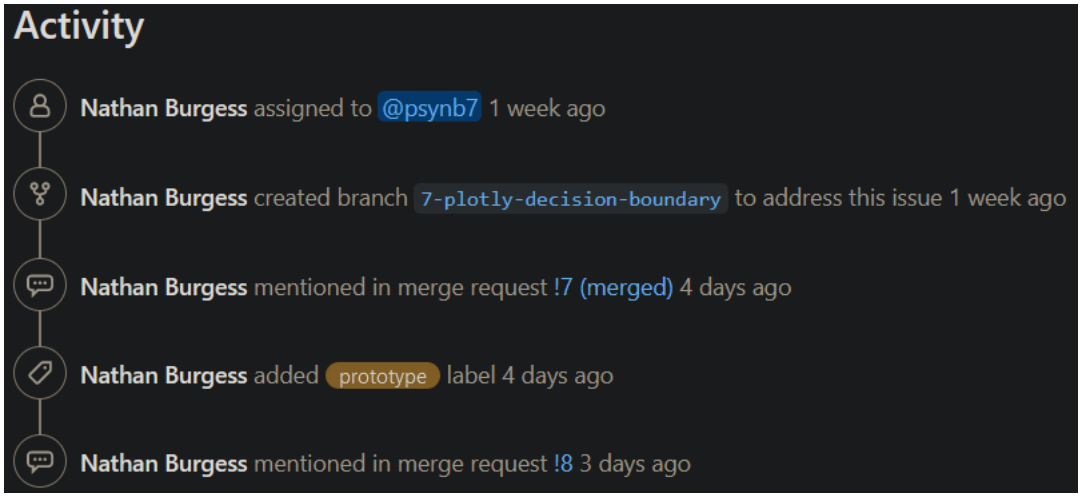


Figure 1.16: An activity log for one of our git issues

The team elected to utilise the following conventions for git commit messages:

- The subject of the commit message cannot exceed 50 characters
- The type of the commit must be indicated at the start of the message eg. "fix:" indicates a bug fix
- The type must be followed by an imperative sentence describing the purpose of the commit
- If any additional detail is required it must be included in the body of the commit message
- Lines in the body of the commit message are limited to 72 characters
- When merging, the same syntax must be followed but with "merge request:" as the type

We took inspiration from the *paired programming* development technique for our approach to merging. The team member that created the merge request was required assign it to someone else from the team to review it. The role of the reviewer was to make sure that it passed any predefined tests and met the criteria outlined in the original issue. If it did, they merged the branch into dev and closed the initial issue. If it failed to meet these requirements, the issue was reassigned to the person who made the merge request with a comment explaining what needed to be done before the branch could be merged. This approach ensured that the code produced was of a high quality and that the features being added met the project requirements.

1.7 Testing

For testing purposes, we used the Python's inbuilt PyUnit Framework. This enabled us to perform multiple tests per function and to evaluate these using different parameters. Another advantage of this approach was that it could be used for both unit testing and integration testing. The module came pre-installed with Python, which was incredibly convenient as any device we used to work on the project was able to run the tests with no extra steps.

So that everyone on the team understood how to use the library, we created a document outlining the setup and usage of PyUnit. It contained an example script with tests written for it, along with clearly outlined steps for writing such tests. We also defined testing conventions to ensure that everyone's tests remained congruous.

The conventions are as follows:

- Any class containing tests should be named in the format 'Test[NameOfClassToBeTested]'. Where the first letter of each word should be capitalised.
- Any function within a test class should be named in the format 'test_[What_Is_Being_Tested]'. Where the first letter of every word following 'test_' is capitalised and each word is separated by an underscore.
- Every class containing tests should appear at the bottom of the Python file it belongs to.
- There should only be one test class for every actual class in order to adhere to the single responsibility principle.
- Every test function should be preceded by a comment stating what the function is testing.

```
class TestImportUtil(unittest.TestCase):
    #Testing that the csv to dataframe function returns the same dataframe as if it were imported directly.
    def test_CSV_To_Dataframe(self):
        with open('CSVs/iris.csv', newline='') as irisCSV:
            csvStr = ""
            irisReader = csv.reader(irisCSV, delimiter=',')
            for row in irisReader:
                csvStr += (', '.join(row))

            df = iu.csvToDataFrame(csvStr)
            pandasDF = pd.read_csv('CSVs/iris.csv')
            self.assertEqual(df.columns[0], pandasDF.columns[0], "The first column labels do not match.")


if __name__ == '__main__':
    unittest.main()
```

Figure 1.17: An example test for the CSV input system

1.8 Project Management

The Agile methodology formed the basis of our approach, a choice that was vindicated by our ability accommodate the frequent evolution of requirements throughout the project. Our sprints centred on a weekly meeting where we would evaluate the progress made from the last sprint before setting out what needed to be accomplished in the next. Once the tasks for the sprint had been identified, work would be split between members. Effort was made during the assignment of tasks to ensure that the split of work was even and that the tasks assigned to a team member matched their individual skill set. For particularly large tasks or when a team member needed support, sub-teams were created and assigned to the task to ensure its timely completion.

Another important aspect of our strategy was the choice to minimise formal documentation, something our supervisor was very keen on. Expansive documentation leads to a lot of overhead for the creation of new features, violating a key principle of the agile methodology. Instead, we focussed on creating iterations of the prototype with the aim of maintaining a constant flow of new features. We held regular meetings at two sprint intervals with our academic supervisor to gain feedback and ensure that the project was moving in the direction he wanted.

To track tasks, we made use of a Kanban board hosted on Trello. The aim was to facilitate self-organisation while maintaining a centralised record of the tasks assigned and the progress made. A card was created for each task and a member was assigned to it. At first we assigned work-points[1] to each task. When the time came to divide up the tasks for each sprint, we tried to ensure that each person had an equal amount of work-points assigned to them. However, it quickly became evident that it was very difficult to accurately assign these points. We eventually settled on assigning the tasks based on a variety of factors, including how much work each member had from other modules and how well each team member's skills aligned with a given task. Whenever a task was conducive to collaboration, multiple members were assigned to the corresponding Kanban card as a sub-team.

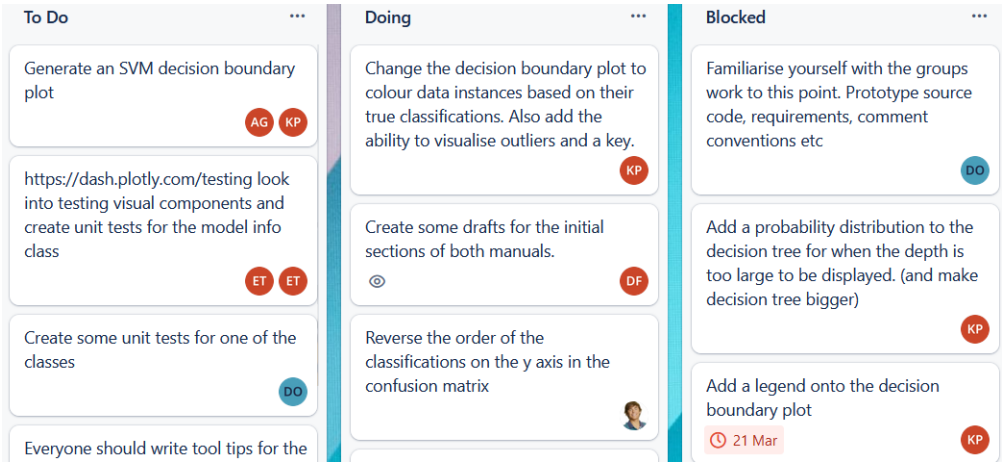


Figure 1.18: A section of the Kanban board in use

Most team communication outside of meetings was handled through a Discord server. This allowed announcements to be made to the entire team but also provided an avenue for more informal discussion and collaboration. OneDrive was used extensively for collaboration where documentation was involved. This proved especially useful for the reports as team members were able to work in parallel on their assigned sections.

[1] A measure of the perceived amount of time needed for a task

1.9 Reflection on Project Management

As a team, we endeavoured to employ several project management techniques in order to ensure a smooth development process (Section 1.8). Our approach continually evolved over the course of the project, always with the aim of increasing the efficiency of our process and the quality of our output.

The first aspect of our management that we identified as requiring improvement were our meetings. Initially, they were quite ineffective, often overrunning their allotted time due to discussions going off-topic. To improve the efficacy of the meetings, we therefore adopted a more rigid structure. Meetings had to have a predefined purpose and would start with a demonstration of each member's progress since the last meeting followed by a discussion of what needed to be done during the next sprint. Tasks would then be assigned and it was only after this point that a more informal discussion could take place. This change was an immediate success, facilitating a more focused and productive use of time with meetings lasting an hour at most. Consequently, our meetings became a very valuable use of time and were a driving force for progress on the project.

Development sprints were another area which required improvement. Initially, sprints were loosely followed and poorly defined, with members sometimes unaware of the tasks assigned and their deadlines. This resulted in internal deadlines occasionally being missed, undermining the use of sprints. In an attempt to improve this, we began to make greater use of the kanban board which allowed us to organise our task backlog more effectively and provided an easy point of reference for due dates. Although we became better organised the adherence to deadlines remained a weakness, partly due to a failure to regularly update the board. This is something we would seek to improve in any future projects by making the maintenance of the kanban board much more of a priority.

As a group we also tended to underestimate timescales. After returning from the winter break, we assumed that we could make quick and significant progress on the project. This turned out to be unrealistic as there were many competing demands from other modules. After realising this, we were forced to prioritise key features and had to omit some functionality from the final piece of software (Section 1.4). This experience taught us the importance of remaining realistic and performing comprehensive planning at the start of a project in order to avoid underdelivering on promises.

The effectiveness of our retrospectives was mixed. They resulted in consistent incremental improvements to our management approach but were often rushed in the closing minutes of a meeting and were not properly documented. We could have improved this process by creating a document outlining the required structure for a retrospective and allocating a reasonable amount time for one within each meeting. This, combined with the aforementioned difficulty with going off-track, has demonstrated to us the benefits structure can provide when working as a team.

Communication improved substantially throughout the process and was fundamental to our success. We ensured there was the option to attend meetings on teams when members were unable to attend in person and maintained comprehensive meeting minutes. This proved to be a powerful mitigation technique that allowed several members to keep up to date despite some spouts of coronavirus infection. The use of the discord server also increased with time. It proved to be a very effective tool for asking quick questions and getting updates on progress without having to call a full team meeting.

Overall, the project management was handled very well, with a sustained effort being made to improve on our shortcomings. We learned the importance of forward planning and greatly improved our communication. By intelligently assigning tasks to the individual best positioned to tackle them, we managed to avoid situations where team members were not capable of completing their tasks to a high enough quality or in line with the deadlines. These management skills were honed over the course of the project and will be put to good use in any future projects we are a part of.

1.10 Division of Work

Dominic Cripps:

- Contributions to EOIs and Pitches during the bidding process
- Contributions during requirements gathering
- Contributed to reports and manuals
- Dynamic Layout of Components
- Underlying modular system design
- Training Settings Component
- Model Information Component
- User Input Component
- Class Split Component
- Confusion Matrix Component
- Sub-Tree Component for Random Forest Models
- Ability to train and retrain models
- Sanitation Checks and Error Messages
- Multiple Prototypes
- CSS Styling
- Website Layout
- Video Editor
- Contributions to Demo Video script

Ethan Temple-Betts:

- Contributions to EOIs and Pitches during the bidding process
- Contributions during requirements gathering
- Contributed to reports and manuals
- Website layout and style for the user manual
- Organised and Edited the user manual
- Initial Dash prototype
- Parallel Coordinates Component
- Feature Space Component
- Model Decision Tree Component
- CSV input and Sanitation
- Testing Framework
- CSS styling
- Research for the Dash Python library
- Responsible for meeting minutes and booking meeting spaces
- Demo Video Recording

Nathan Burgess:

- Contributions to EOIs and Pitches during the bidding process
- Contributions during requirements gathering
- Contributed to reports and manuals
- Trello Board management
- Created Docker build files
- Created the Runner for the project, with automated testing
- Refactored the app to work with cloud deployment
- Various cloud deployments (Section 1.5.3)
- Decision Boundary Component for the prototype
- Research for alternatives to Java
- Git management
- Contributions to Demo Video script

Daniel Ferring:

- Contributions to EOIs and Pitches during the bidding process
- Contributions during requirements gathering
- Contributed to reports and manuals
- Edited and organised Reports and Software Manual
- Helped organise the User Manual
- Final Decision Boundary Component
- Pairwise Plots
- Standardised Colours and Symbols for classifications
- Instance Key
- Helped develop the SVM Decision Boundary Component
- Research for plotly and tooltips
- OneDrive Management
- Contributions to Demo Video script

Alfred Greenwood:

- Contributions to EOIs and Pitches during the bidding process
- Contributions during requirements gathering
- Contributed to reports and manuals
- Initial framework for the final software
- SVM Decision Boundary Component
- Design ideas for multiple components
- CSS Styling for initial prototype
- Contributions to Demo Video script

Kieran Patel:

- Contributions to EOIs and Pitches during the bidding process
- Contributions during requirements gathering
- Contributed to reports and manuals
- Styling and extra functionality for the Model Decision Tree Component
- SVM Design Ideas
- Paper-based prototypes
- Handled communication with the Project Supervisor

Dixon Okoro:

- Recorded a section of a pitch video during the bidding process

1.11 Reflection on Results

1.11.1 Successes

The primary success of the project is the app itself. While the scope of its functionality is somewhat less than originally planned, this turned out to be a somewhat of a positive: the app is much more streamlined, containing only features that are relevant for learners. The design of the app is simple and intuitive with users being able to clear up any confusion they might have through the tooltips or the user manual. Furthermore, the app provides a variety of visualisations all aimed at helping the user to understand their model's results and the decision making behind them. As students that have gone through the process of having to learn AI, and we can confidently say that the app would have helped us during the learning process and would still be useful for any projects that involve training decision tree models.

Additionally, the underlying structure of the code is a key strength of the project. The final structure was designed by Dominic Cripps and places an emphasis on modularity and extensibility. The efficacy of this approach has been demonstrated since the moment the design was first implemented. The output of new features increased greatly, with team members being able to work on new features separately without impacting previous or ongoing work. The codebase is also well commented, with comments following a consistent convention that ensures the purpose, inputs and outputs, of each function are clear. In addition to being useful for our development cycle, these two aspects also mean that the project can be easily picked up and extended by a future development team, fulfilling a key tenet of software engineering.

Another strength has also been our team cohesion, something that our other successes can definitely be attributed to. Effort was made in every team meeting to ensure that every team member's voice was heard. As a result of this, we always had a wide array of ideas from which to choose our approach to solving any problems. Where appropriate, multiple team members were assigned to single tasks. These assignments were particularly effective, work was delegated effectively and relatively large tasks were completed in a short amount of time due to the utilisation of each member's strengths. Had we had less of a team synergy, it is undoubtable that far less would have been delivered.

Our choice of technologies and research should also be considered a success. The discovery of the dash and plotly libraries greatly streamlined the development process, giving us the tools necessary to execute our vision. The plotly library was particularly useful: the ability to combine low level graph objects into higher level figures enabled the creation of many of our bespoke visualisations. Without these lower-level constituent parts being provided, it would have taken far longer to develop any aspect of the application and it is unlikely that all of our final requirements would have been met.

1.11.2 Shortcomings

The most glaring shortcoming is the absence of features that we promised at the start of the project. As is often the case projects such as this, we were too ambitious at the start, promising functionality that we would not be able to provide. There are two main reasons for this overpromising. The first was that we failed to account for the workload of other modules when creating our initial roadmap; once other modules started releasing the majority of their coursework, the development of new features stagnated. The other is the imbalance in the contributions of team members (Section). The roadmap and requirements were created on the assumption that seven team members would be contributing equally to the project. As this turned out not to be the case, it was impossible to fully deliver what we had initially envisioned.

Another failing of the project was a lack of testing. We had initially planned to follow test driven

development but this was quickly pushed to the wayside in favour of developing prototypes in a shorter amount of time. Testing therefore became something of an afterthought and was seen as a low priority task. The ramifications of this can be clearly seen in the app. While it runs smoothly for the most part, there are some bugs and glitches. This is something that might have been avoided had we had a more rigorous approach to testing. The most egregious of these bugs have been fixed and we have attempted to mitigate the effects of those that remain through the use of a troubleshooting guide in the user manual. Nonetheless, it remains a huge oversight on our part.

1.11.3 Future of the Project

At this stage the team has produced a functioning web app capable of training and showing the results of various decision tree models. The features supported are primarily intended for learners but could conceivably be useful for use in projects that involve training such models. The future direction that is easiest to envision is for the project to be picked up and extended by another software development team. Given the fact that the codebase is well commented and the detail provided by the software manual, it should be relatively straightforward to understand the existing system. Adding new features would also be incredibly simple, thanks to the modular structure of the system (Section 1.5.2).

The simplest extension would be to add support for further AI models. This could include other decision tree methodologies such as regression models but could also encompass adding support for other learning methods like artificial neural networks. Such additions would increase the usefulness of the app to learners without altering its core functionality. If such a team had members specialising in data visualisation, new visualisations could also be added for models that are already supported. Again this would not be difficult to implement and should help the aim of making AI explainable; the more options the user has in visualising their results, the better.

The next development team would also have the option of expanding the scope of the project's target audience. The most straightforward audience to reach would be AI researchers. All that would be required for this would be the addition of support for more advanced concepts such as recognising monotonic relationships. The application would then become a powerful tool for streamlining the research process. This is because the graphical interface for training the models is much simpler than having to write code and, as we found during debugging, can make the process take a matter of seconds. Furthermore, it would also simplify the process of interpreting the results as the system would automatically generate most of the visualisations that they would need, saving time.

Alternatively, the project could easily have a future without additions. The features currently available are comprehensive enough for learning the basics that it could easily be integrated into an introductory AI course. The only prerequisite to this would be to remove some of the bugs that are still present in the system as running into these could be off-putting to users.

1.11.4 Final Thoughts

As a team, we feel that it is fair to label the project as a successful one. We have delivered a functioning web-app that meets all of our finalised requirements and has scope to be extended in the future. Our skills in project management and collaborative working have increased greatly over the course of development. We faced issues during this process due to a lack of experience but mostly found ways to mitigate them, building our resilience as a team and as individuals. For the issues that we failed to resolve, we have learned from them and will avoid repeating the same mistakes in the future. Completing this project was a learning experience for us, and we can confidently say that it has made us better software engineers.

Software Manual

2.1 Introduction

The primary goal of this project is to provide easy to understand visualisations of decision trees, their results, and the steps taken to reach them. The intended audience is first year computer science students studying an introductory artificial intelligence module and all design decisions have been made with this group in mind.

The project has been developed as a web app and is accessible in two ways: as an externally hosted website or as a containerised version hosted locally. The system makes extensive use of the Dash and plotly libraries, the documentation for which can be found here and here respectively.
This manual aims to aid any future developers in quickly getting to grips with the system in order to be able to extend it. As such, it contains sections on project terminology, the high level architecture of the system, external components used, coding conventions and the testing frameworks.
However, any extensions to this project must fit in with the goals and underlying design philosophy of the existing project. A summary of which can be found below.

1. Any new additions fit into and adhere to the existing modular design
2. Any new additions must contribute to the goal of making AI more explainable and accessible
3. Any features that are not intended for learners must be added in a way that does not complicate their interactions with the program

1. Modularity has been a key tenet of our design philosophy, the efficacy of which has been demonstrated by the success of our project thus far. The reasoning for continuing to follow this principle is twofold. Firstly, it will make the extension process far easier. By following the pre-laid structure, you will be able to make additions to the codebase without risk of breaking the existing components. Secondly by choosing not to follow the structure you will invariably make any future extension much harder, breaking a core principle of effective software development.

2. The aim of this project has always been to make AI models more explainable by providing better visualisations of their results and their decision making process. Any additions that do not contribute to this or the ease of use of the program will therefore be superfluous, cluttering the application without benefit.

3. Learners have been the primary audience of the app from the very beginning. While there is scope to include features that would appeal to AI researchers and businessmen, it is imperative that such additions are not to the detriment of a learner's experience. Any features not suitable for learners should be added in a way that does not clutter the layout of the application or cause extra steps

in processes related to learning. Any feature that cannot adhere to this principle should either be scrapped or put into a separate application designed with its relevant target audience in mind.

Now that the prerequisites for additions have been defined we can provide some potential ideas for extension:

- Support for other AI methodologies such as Artificial Neural Networks
- Support for Regression decision tree models
- Additional visualisations for models that are already supported
- Support for recognising monotonic relationships

2.2 Project Terminology

It is assumed that any developer working on the project should have a fundamental understanding of what a decision tree is. In the unlikely event that this is not the case, a quick introduction to the very basics of decision trees can be found in the User Manual. For more advanced topics, a brief definition with links to further information is provided.

Decision Boundary A hyperplane that separates instances of a data set into separate classifications. Visualisations of decision boundaries are very useful in representing how the model uses specific features to classify instances as well as the relationships between features. A more in-depth explanation can be found here.

SVM Decision Trees A more advanced decision tree technique that can be used for non-linear classification tasks through the use of the *kernel trick*. More information can be found here.

Ensemble Learning A machine learning approach in which the predictions of multiple trees are combined with the aim of mitigating biases, creating a more accurate model. Example of ensemble learning include Random Forest and Gradient Boosting, details of which can be found here.

2.3 High Level Architecture

2.3.1 General Overview

The program has a modular design and makes use of various software engineering concepts and design patterns, including the Single Responsibility Principle, Singleton Pattern, and Factory Pattern. This architecture promotes maintainability, scalability, and flexibility. Each class focuses greatly on specific tasks which increases code readability and promotes better testing. The implementation and justification of the chosen methods will be further discussed.

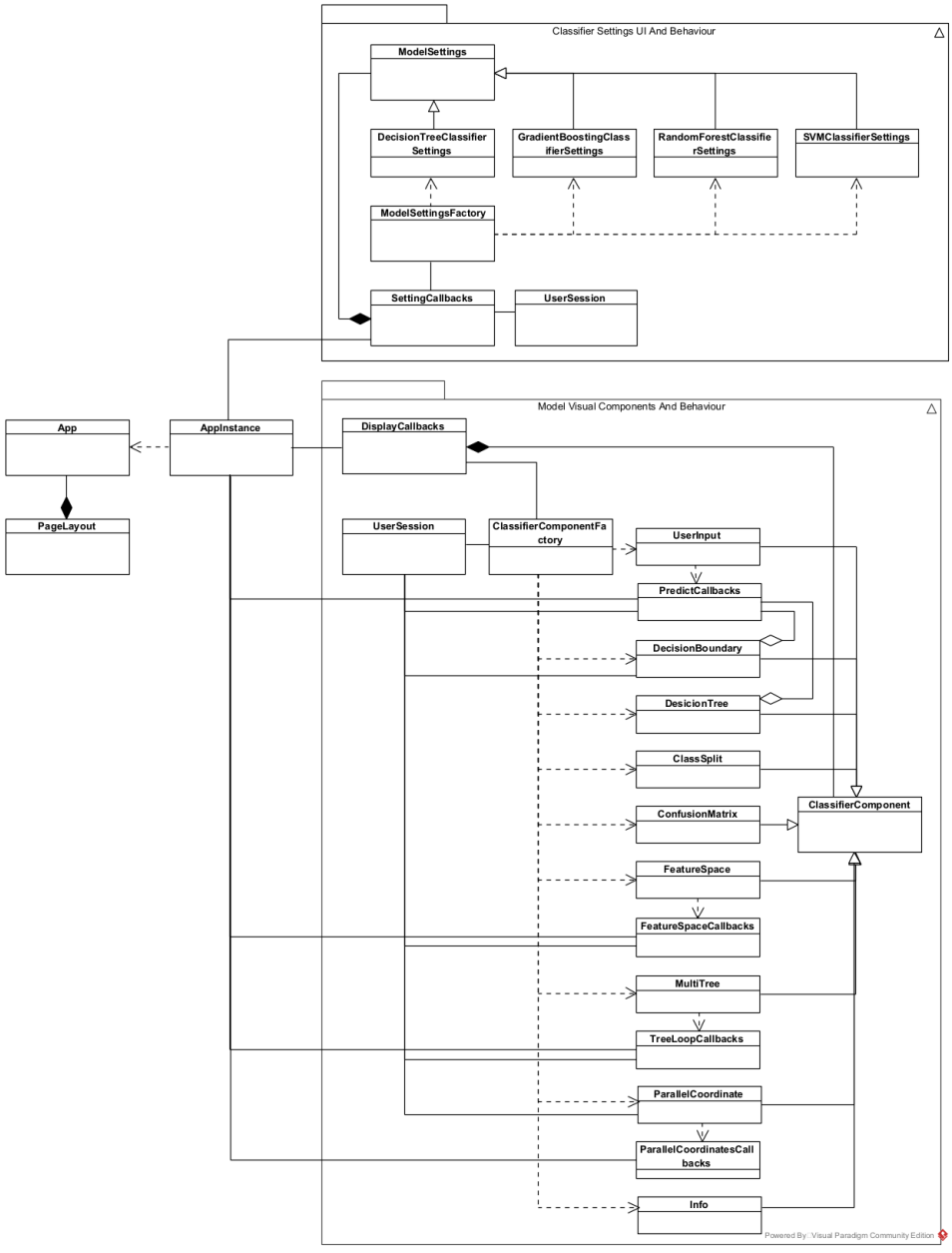


Figure 2.1: The class diagram for the overall system design

2.3.2 Components

The aim of the project was to enhance explainable AI, and as such all supported models have a collection of components that will be shown upon training. This section will explain the implementation of the component system.

'ClassifierComponent' is the parent class of all displayable components. It only contains two attributes: 'componentChildren' and 'componentTitle'. 'componentChildren' is an array that will contain the content to be displayed, this could be a html div, text, or graph etc. 'componentTitle' is just a string that contains the title of the component, this will be shown above the 'componentChildren'.

To create a new classifier component, you should create a child class of 'ClassifierComponent' and set both 'componentChildren' and 'componentTitle' appropriately. To adhere to the Single Responsibility Principal utility classes have been used to generate diagrams and figures to be displayed.

The factory design pattern is used in 'ClassifierComponentFactory' to generate the correct components for each training model. A dictionary 'components' (index by model name) is used to return an array of rows. Each row is an array containing the classes of components to be displayed on that row. An example row could look like: '[ClassifierInfoComponent, ClassifierUserInputComponent, ClassifierClassSplitComponent]'.

To display a newly created component simply add a row or extend a row using the name of the new class. The factory class will iterate through each row, work out the correct margins to correctly space each component, will assign both 'componentChildren' and 'componentTitle' to a new div which is then appended to another row div. An array containing the row divisions is then returned.

The 'ClassifierComponentFactory' method 'Factory' is used when a trained model is selected, its return value is used as the output to a Dash Call-back and is displayed on the website in the div 'model-components'. The code for this can be found in 'DisplayCallbacks'.

2.3.3 Settings

To provide the same freedom a user would have when training a model themselves we have support for setting arguments when training a model. The parameters for each model are different and as such the display must change to reflect this. This section will explain the implementation of the model settings. 'ModelSettings' is the parent class of all displayed model parameters. It contains three attributes: 'classifierLayout', 'parameters', and 'classifier'. 'classifierLayout' is an array that will contain the content to be displayed, a typical structure would look like:

- Classifier Name e.g., 'Decision Tree Params'
- A checkbox labelled with the corresponding parameter
- Dropdown menu / Numeric Input / Slider – some way to take user input
- Bullet points 2 and 3 repeat until all training parameters are accounted for

The IDs of the checkboxes must take the form 'dict (name = "classifier-settings-custom", idx = "ACTUAL PARAMETER NAME") '. The IDs of the user input elements must take the form 'dict (name = "classifier-settings", idx = "ACTUAL PARAMETER NAME") '. This is so the training function can use pattern matching callbacks to take in the parameters as input.

The 'classifier' is just a class reference to the classifier type e.g., 'DecisionTreeClassifier'. The 'parameters' is an array of strings containing the names of all supported parameters. To provide support

for a new classifier model you should create a child class of 'ModelSettings' and set 'classifierLayout', 'parameters', and 'classifier' appropriately.

The factory design pattern is used in 'ModelSettingsFactory' to generate the correct settings to display for each training model. A dictionary 'settings' (index by classifier name) is used to return which 'ModelSettings' subclass to use. The classifier name that is used as the dictionary key must also be added to the array 'SUPORTED_CLASSIFIERS' found in 'PageLayout' for it to be selected.

The 'ModelSettingsFactory' method 'Factory' is used when a value from 'SUPPORTED_CLASSIFIERS' is selected to be the model to train, its return value is used as the output to a Dash Call-back and is displayed on the website in the div 'classifier-settings'. The code for this can be found in 'SettingCallbacks' under the function 'updateClassifierSettings'.

When the train button is pressed the function 'train' will run. It uses pattern matching callbacks to take in the values of each currently displayed parameter and parameter checkbox. It will use the values of the checkboxes to determine which values to pass into the training function. The model is then trained using the 'classifier' attribute of the selected settings and the appropriate arguments. The model is then fitted using the training data.

2.3.4 Model Information

The information about all trained models must be stored to correctly populate their respective components and switch between previously trained models. This section will explain the implementation of the user session.

A singleton class 'UserSession' is used to store dynamic data relating to the trained models. It is accessed using: 'UserSession().instance'.

It currently stores four dictionaries:

- 'modelInformation'
- 'selectedModel'
- 'selectedTree'
- 'selectedBoundary'

'modelInformation' is itself a dictionary and stores:

- The trained model
- The training data used in the form [xTrain, yTrain]
- The testing data used in the form [xTest, yTest]
- The arguments used to train the model
- The test train split – in the range of 0.0 to 1.0
- The classifier types
- The user inputted name of the model
- The selected settings used when training the model

- The colour key for consistency between plots
- The shape key for consistency between plots

'selectedModel' stores references to the models that are currently selected by users.

'selectedTree' stores copies of any Decision Tree plot that is currently in use.

'selectedBoundary' stores copies of any Decision Boundary plot that is currently in use.

Before training a model, the user must specify a session ID. Any model trained whilst that ID is selected will be stored under that ID, only models stored under the currently selected ID are accessible to the user. This is done by indexing all the previously mentioned dictionaries using this ID. This allows for the illusion of independent user sessions without raising ethical concerns from storing data permanently or using the users IP address. This does however mean that two users can both use the same session ID simultaneously, therefore it is in the best interest of the user to select a unique string that is likely not selected by anyone else.

This singleton class can be used to access any information about any model anywhere in the codebase.

2.4 External Components and Build Guide

As a Python program this application requires dependencies downloaded through pip. To get your system ready for development, creating a virtual environment is recommended. The methods for doing this will depend on your operating system, but this guide can be used for windows and linux systems. Make sure that you install Python version 3.9 as this is the version our project uses.

After following the guide, you will have created a virtual environment called "venv" within your project folder with python3.9 installed. You should make sure your new venv folder is added to your .gitignore so that unnecessary files are not added to the repository.

To download the required dependencies, run the following commands:

- *Pip install –upgrade pip*
- *Pip install –r requirements.txt*

If you make changes during development that add a new dependency, you will need to update requirements.txt. This can be done by running:

- *pip freeze > requirements.txt*

Our main delivery system uses Docker. As such, you will need Docker installed to use the container. Installation instructions for all supported operating systems can be found here.

If you wish to test the website on localhost there are two main options. You can run the program from app.py in your virtual environment or run the docker container outside of your virtual environment. The way that we chose to run the program from app.py was through the VSCode IDE. In order to activate the virtual environment in VSCode, do the following:

- Open team44_project in VSCode
- Press Ctrl + shift + p
- Select "Interpreter"
- Select "Enter interpreter path ..."
- Click "Find"
- Go to venv/bin
- Select python3.9 − > open

If this was successful, your VSCode terminal will indicate that you are working within venv.

To run the docker container, do the following:

- Cd into team44_project
- Run *docker build -t aiVis*[1]
- Run *docker images* to view your currently built images
- Run *docker run -p 8080:80 aiVis*

[1]If you get a permission denied you can either run the command with sudo or add docker to the appropriate user group. Instructions can be found here.

- Navigate to: “localhost:8080” In your internet browser URL bar

If you wish to change the port at which the container is accessible on, you can edit the final line of the Docker file:

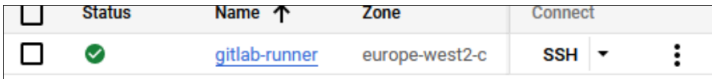
- CMD gunicorn –bind=0.0.0.0:80 –chdir src app:app

The Docker file can be found at the front of the repository along with the *.Dockerignore* and *requirements.txt*.

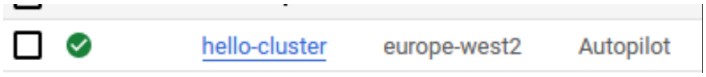
2.4.1 Cloud Components

The current system is made up of 3 cloud components

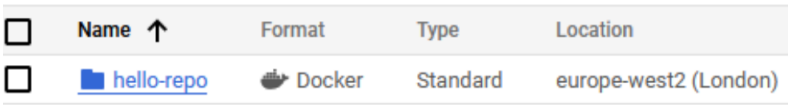
A Google Compute Engine instance – *gitlab-runner*:



A Google Kubernetes Engine instance – *hello-cluster*:



A Google Artifact Registry – *hello-repo*:



Follow this tutorial to upload new versions to GKE, replacing names where appropriate. Figure 2.2 provides an overview of this process

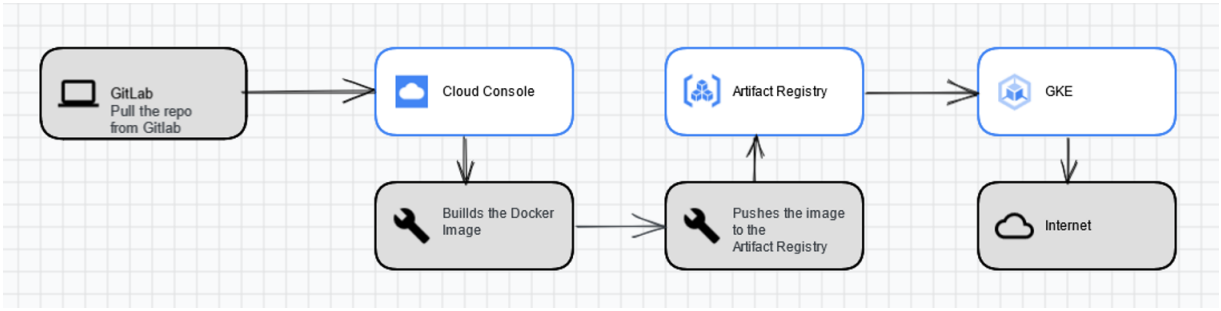
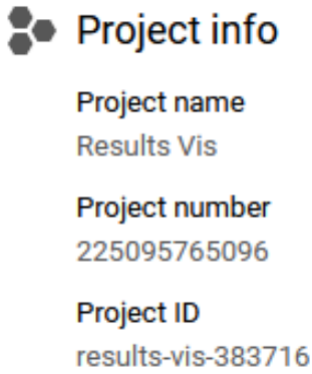


Figure 2.2: The process for uploading to GKE

NOTE:

- Your Google Cloud account will need to be added to have access to the project
- When running commands, change the variable names to those shown in the project info (see underneath) and the cloud component names (listed above)



2.5 Coding Conventions

Naming Conventions :

- Use descriptive names for variables, functions, and classes to improve readability.
- Follow the PascalCase naming convention for class, function and variable names.
- Avoid using single-letter variable names unless they are used as indices or have a widely accepted meaning (e.g., x, y for coordinates or i for loop iterations).

Code Structure :

- Organize the code into separate files based on their purpose (e.g., call-backs, layout, utilities)
- Use functions and classes to modularize the code and improve reusability
- Follow the DRY (Don't Repeat Yourself) principle to minimize code duplication

Error Handling :

- Use proper error handling to manage exceptions and provide helpful error messages
- Check for potential edge cases and handle them appropriately to ensure robustness

Imports :

- Keep imports at the top of the file
- Use import statements for packages and modules
- Use from ... import statements for specific functions or classes

Comments :
For a Class, list the following:

- Author (the person who originally created it)
- Date Created
- Previous Maintainer (the person who last modified it)
- Date last modified
- A sentence describing its purpose

For a Function, list the following:

- Author (the person who originally created it)
- Date Created
- Previous Maintainer (the person who last modified it)
- Date last modified
- A sentence describing its purpose

- The function parameters and their purpose, should you expect only specific values to be entered for a parameter, state that here

Other rules:

- Clearly state "TO-DO" for functions that are yet to be implemented
- Add a comment for each logical section of code within the function, briefly explaining what it does

```
"""
AUTHOR: Daniel Ferring
DATE CREATED: 19/02/2023
PREVIOUS MAINTAINER: Daniel Ferring
DATE LAST MODIFIED: 24/02/2023

Used to create visualisations of a given model's decision
boundaries. Can create either 1D or 2D representations.
"""
class DecisionBoundaryUtil():

    """
    AUTHOR: Daniel Ferring
    DATE CREATED: 24/02/2023
    PREVIOUS MAINTAINER: Daniel Ferring
    DATE LAST MODIFIED: 21/03/2023

    Generates the values required to create a heatmap using values from
    the model's training data.

    INPUTS:
    trainingData: the data used to train the model
    """
    def getHeatmapValues(self, trainingData, testingData):

        #Stores minimum values for each feature
        mins = []
        #Stores maximum values for each feature
        maxs = []

        #Extracts feature values from the training data
        features = pd.concat([trainingData[0], testingData[0]])
```

Figure 2.3: An excerpt of commented code from DecisionBoundaryUtil.py

2.6 Testing Frameworks

All the unit tests can be found in tests/ at the root of the project directory. Tests are further decomposed into two directories: test_callbacks and test_utils.
Three Python libraries are required in order to run the test cases:

- *pytest*
- *coverage*
- *pandas*

2.6.1 Running Unit Tests

Running the following commands in the terminal should ensure you meet these requirements.

- *python -m pip install pytest*
- *python -m pip install pytest-cov*
- *python -m pip install coverage*
- *python -m pip install pandas*

Below is an example of how to run multiple unit test files and generate a html report to assess line coverage.

The following command was run in the terminal, from the src directory:

- *python -m pytest –cov-report=html –cov=callbacks/callbacks/ tests/test_callbacks*

–cov-report=html specifies that a html report should be generated, displaying the result of the test cases and their line coverage. The *index.html* file is be generated and stored in the *htmlcov* folder, at the root of the project directory. *–cov=* is where you should specify the path to the code to be tested. *tests/...* is used to specify which tests are to be run.

In the example above, all of the tests for callbacks were executed. The command was run from the *src* directory. The code to be tested was in *src/callbacks/callbacks* so the relative file path *callbacks/callbacks* was used. The tests were in *tests/test_callbacks* so the full file path had to be used.

The following command will run all the tests on all the source code:

- *python -m pytest –cov-report=html –cov=../src tests/*

2.6.2 HTML Test Report

The report in Figure 2.4 is created when these commands are run successfully. You can then view specific .py files to assess their test coverage.

Coverage report: 43%

coverage.py v7.2.5, created at 2023-05-06 11:48 +0100

Module	statements	missing	excluded	coverage
AppInstance.py	6	0	0	100%
DashInstance.py	0	0	0	100%
UserSession.py	9	3	0	67%
app.py	17	1	0	94%
callbacks\callbacks\BoundaryCallbacks.py	38	26	0	32%
callbacks\callbacks\DisplayCallbacks.py	14	3	0	79%
callbacks\callbacks\FeatureSpaceCallbacks.py	33	21	0	36%
callbacks\callbacks\ParallelCoordinatesCallbacks.py	20	9	0	55%
callbacks\callbacks\PredictCallbacks.py	77	62	0	19%
callbacks\callbacks\SettingCallbacks.py	91	63	0	31%
callbacks\callbacks\TreeLoopCallbacks.py	18	8	0	56%
classifier_components\ClassifierComponent.py	6	2	0	67%
classifier_components\ClassifierComponentFactory.py	28	13	0	54%
classifier_components\components\ClassifierClassSplitComponent.py	27	20	0	26%

Figure 2.4: The generated html report

The Run and Missing tags at the top of the file shown in Figure 2.5 refer to the line coverage. For this file 15 lines have been covered by at least one testcase and 1 line has been missed.

Coverage for **tests\test_callbacks\test_SettingCallbacks.py**: 94%

16 statements | 15 run | 1 missing | 0 excluded

« prev ^ index » next coverage.py v7.2.5, created at 2023-05-06 11:48 +0100

```
1  import os, sys, unittest
2
3  # This allows python to find modules stored in parent directories
4  # specifically in this case the src directory as it is 2 parent folders
5  # up from this directory
6  fpath = os.path.join(os.path.dirname(__file__), '..\\..')
7  sys.path.append(fpath)
8
9  # The path had to be changed so app.py cpould be imported
10 # This is important as it will initialise AppInstance.app,
11 # which is needed for callbacks to work
12 from app import app
13
14 # All callbacks from SettingCallbacks.py can then be imported
15 import callbacks.callbacks.SettingCallbacks as SettingCallbacks
16
17 class test_readDataframe(unittest.TestCase):
18
19     # setUp and can initialise resources to be used in test cases
20     # I think it has to be named setUp so it doesnt get run as a test
21     def setUp(self):
22         self.uploadMessage = "Drag and drop or click to upload a dataset (.csv) "
23
24     def test_readDataframe_000(self):
25         # Specific callbacks in SettingCallbacks can be accesed
```

Figure 2.5: A view of test coverage for a .py file

2.6.3 Test_utils

All the util classes have a separate test file that contains their specific test cases. What follows is a guide for creating a unit test for a util class.

Firstly, the path must be amended so that the util modules can be located. After this, you can import individual util classes that are in src/utils/:

```
fpath = os.path.join(os.path.dirname(__file__), '..\\..')
sys.path.append(fpath)
from utils.Util import ImportUtil
```

The next step is to create a testcase class:

```
class test_ImportUtil(unittest.TestCase):
```

Within this class you can then setup any necessary variables that are required for the test cases by creating a setup function. You must follow the naming convention, as otherwise this function will be run as a test:

```
def setUp(self):
    self.uploadMessage = “Upload a .csv file”
```

Then you can define your test cases:

```
def test_readContent_0(self):
    file = "test.txt"
    content = "text/plain;base64,SGVsbG8gV29ybGQ="
    expected_output = "Hello World"
    self.assertEqual(ImportUtil.readContent(file, content),
                     expected_output)
```

All unit tests must include the following at the bottom of the implementation file:

```
if __name__ == '__main__':
    unittest.main()
```

This is necessary to ensure that the tests are run correctly and that the report is generated.

2.6.4 Test_callbacks

Dash callbacks are organized in separate files, according to the feature they relate to. Below is a guide on how to perform unit tests on the callbacks contained in one of these files.

Firstly, the path must be amended so that Python can locate the local app module. This initializes the AppInstance class with the 'dash_app' which is necessary as the individual callbacks need a reference to the 'dash_app' instance in order to function correctly.

```
fpath = os.path.join(os.path.dirname(__file__), '..\\..')
sys.path.append(fpath)
from app import app
import callbacks.callbacks.SettingCallbacks as SettingCallbacks
```

Next, you can import the callback file you need to test. In the example above the SettingCallbacks.py file is imported, which contains three individual callback functions, *readDataframe()*, *updateClassifierSettings()* and *train()*. Below is an example of how to test an individual callback.

```
def test_readDataframe_000(self):
    result = SettingCallbacks.readDataframe(["hello"], [])
    self.assertEqual(result,
                     (False,"No Contents!", self.uploadMessage,
                      [], [], [], []))
```

As you can see you can now reference specific callback functions such as *readDataFrame(filename, contents)*. From this point you can use the existing unit test framework as normal such as the *assertEqual* function. Callbacks often return multiple values. When this occurs, Python stores all the outputs in a tuple. You can then test each value individually or all at once as in the example above.

User Manual

As a team we elected to host our user manual as a website. This has two benefits. The first is that it allows for greater control of the styling and layout of the manual, resulting in better readability. The second is that it improves the accessibility of the manual: it can be accessed from anywhere, on any device. Furthermore, as our project is accessible as a website, it would be more intuitive to the user to have the manual hosted as well.

The manual can be found here: https://team44usermanual.netlify.app/

We have also provided a version of the website converted into a pdf format in this report. Please note that the formatting is slightly different due to this conversion.

Results Visualisation User Guide

The primary goal of this software is to provide easy to understand visualisations of decision trees, their results, and the steps taken to reach them. The intended audience is first year computer science students studying an introductory artificial intelligence module and, as such, the software is designed with this group in mind. It is our hope, however, that the software and the visualisations themselves will be clear and useful enough that it may find scope outside of this specific use case.

Another issue that we have tried to address with this software is that of “Explainable AI”. AI is increasingly being used in decision-making processes which impact people’s lives. While it can be a useful tool to quickly make predictions, there can be difficulty in explaining how it reached those predictions. This can be problematic; an artificial intelligence model could reject a mortgage application but it would be impossible to explain why to the applicant if there is no way of understanding how the decision was reached. To this end we have implemented visualisations which highlight the decision making process of the tree, to better understand the factors that went into each classification. Additionally, we have created visualisations that highlight inaccuracies within the model to illustrate the potential limitations and shortcomings of a given decision tree model.

The aim of this manual is to supplement the software and enhance your understanding of decision trees. Contained within it is an overview of the concepts behind decision trees, a guide on using the web-app itself, a description of the limitations of the system, and a section on troubleshooting. By reading this manual, you will provide yourself with all of the prerequisite knowledge required to use and understand our software.

A Guide To Decision Trees

A decision tree is a machine learning model used for classification and regression tasks . A decision tree model is produced by applying a decision tree algorithm to a labelled data set. Before explaining what the algorithm does, we need to establish some terminology.

A labelled data set is simply a collection of data with labelled rows and columns that we can use to train machine learning models. One of the most widely used data sets is the iris data set. Features are the columns within this data set. When training a model, you choose the features that you want to use for the decision making (input features) and the feature you want to use to classify the instances (target feature). In the case of the iris data set, your input features could be the sepal length and petal width while your target feature would be the species of flower. An instance is one row within the data set, this would be a single flower in the iris data set. Below is an image of the iris data set imported into python as a pandas DataFrame.

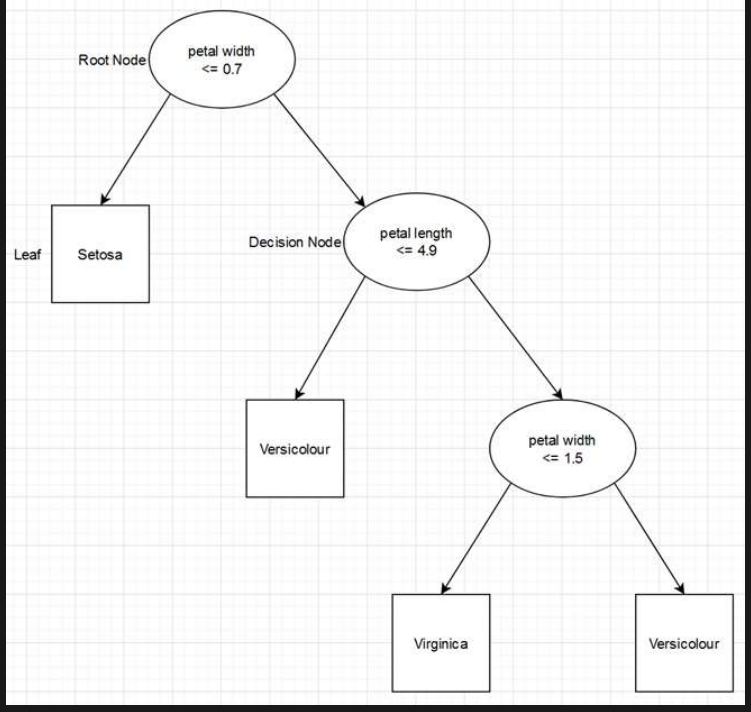
	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	1	5.1	3.5	1.4	0.2	Iris-setosa
1	2	4.9	3.0	1.4	0.2	Iris-setosa
2	3	4.7	3.2	1.3	0.2	Iris-setosa
3	4	4.6	3.1	1.5	0.2	Iris-setosa
4	5	5.0	3.6	1.4	0.2	Iris-setosa
...	...	...	...	...	...	...
145	146	6.7	3.0	5.2	2.3	Iris-virginica
146	147	6.3	2.5	5.0	1.9	Iris-virginica
147	148	6.5	3.0	5.2	2.0	Iris-virginica
148	149	6.2	3.4	5.4	2.3	Iris-virginica
149	150	5.9	3.0	5.1	1.8	Iris-virginica

150 rows × 6 columns

The aim of the decision tree algorithm is to create a model capable of predicting the classification of instances. These classifications are the possible values of the previously defined target feature. For instance the iris data set has three possible classifications when species is selected as the target feature: Setosa, Versicolour and Virginica. The algorithm does this by splitting the data set, with the goal of creating subsets in which only instances of a single class are present. These splits are made by selecting an input feature and defining a threshold value. For instance, a tree could split the iris data set by saying that any plant with a petal width <= 0.7 is a Setosa.

It should be noted that this example is somewhat of an outlier. It is quite often the case that classifications are not linearly separable and require multiple splits, potentially using a different feature for each, before they can be separated. The features and thresholds used in these data splits are what form the decision rules of the model. These rules can then be applied to unclassified instances in order to attempt to predict what their classification should be, often to varying degrees of success.

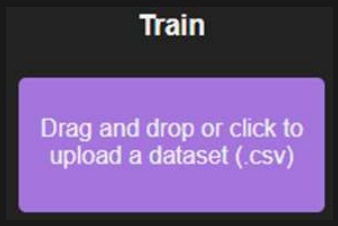
The basic structure of a decision tree usually follows that of a binary tree with the root and internal nodes representing the decisions (splits) made at each stage and the leaf nodes representing the separated data and its classification.



The tree will classify an unclassified instance by starting at the root node and moving down each node according to each feature threshold until it reaches a leaf, at which point the instance has been classified. In the example above, an instance with a petal width > 0.7 and a petal length <= 4.9 would be classified as a Versicolour plant.

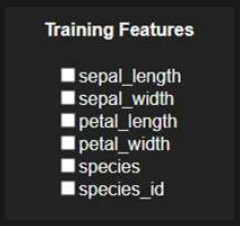
Training a Model

Upload Datasets



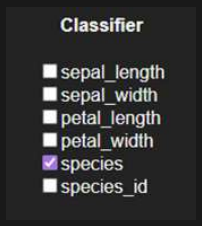
Drag and drop or click this box to upload your CSV file. Some automatic formatting will be applied to the data which will remove null values as these cannot be used to train the model. A good source of CSV files you can use can be found here.

Selecting Training Features



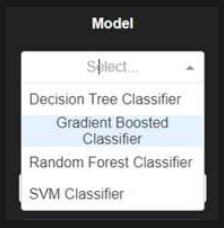
Once you have uploaded a CSV file, you will be presented with all the features identified in the file. Use the checkboxes to select whether you want this feature to be used to train the model. Make sure not to select the feature used to classify your data.

Selecting a Classifier



After you have chosen the training features, you need to tell the model what feature should be used to classify the data. Regression models are not currently supported so whichever feature you select will be interpreted as discrete rather than continuous data.

Selecting a Model Type

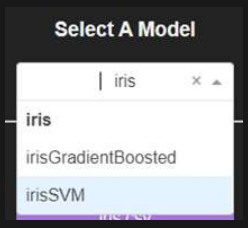


These are the supported model types. The first three options are all decision tree-based models. Gradient boosted and random forest classifiers refer to two different ensemble learning techniques. SVM is a support vector machine, which is a linear based model. You can learn more about different models here.

Train - Test Split

This allows you to specify what ratio of the data should be used for testing the model and what ratio should be used for training. In the example provided, 75% of the total data will be used for testing and 25% for training. You should then name your model and press the train button to generate the data visualizations.

Switching Models



This dropdown provides the ability to switch between the models you have already trained. If you use the same name for all the models you create, the data will be overwritten so make sure to use unique names if you want to retain your data.

Decision Tree parameters

Criterion

This allows you to specify how the impurity of a split will be measured. The default value is to measure using "gini". Your selection will only be applied if you select the checkbox and choose a measure of impurity. More information about the difference between these measures of impurity can be found here.

Splitter

This allows you to specify how the model should search for features to split. The default is best, which means at each node in the tree, the feature which results in the most optimal split of data is chosen. Random means that the best feature in a random subset of features will be chosen. The size of this subset can be specified using the max_feature parameter. The measure of how "good" the split is will be determined by what criterion you specify, by default the measure is gini.

Max Depth

Specify the maximum depth that the tree is allowed to reach. This is a useful tool for preventing overfitting. By default, there is no maximum tree depth.

Min Sample Split

Changing this parameter will change the minimum number of samples a node can contain if it is to be split, the default is 2.

Min Sample Leaf

This is the minimum number of samples that a node can contain and still be considered a leaf. The default is 1.

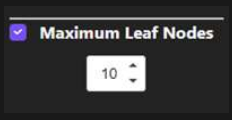
Max feature To Split

This allows you to tell the model to only consider a random set of features which will be as large as the size you specify. When the model attempts to find the best feature to use in order to split the data at a given node, it will select one from this random set. By default there is no maximum.

Random State

By setting a random state, the train and test sets, which are subsets of the total dataset, will remain unchanged every time you train the model. By default, it is not set, but by providing any number you prefer, and reusing this number over different model types, you can ensure that each model is trained and tested with identical train and test datasets, given that the train-test split is also the same.

Maximum Leaf Nodes



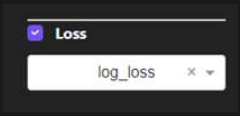
You can use this to limit the maximum number of leaf nodes in your model. By default there is no maximum.

Min Impurity Decrease

This parameter specifies the minimum value of impurity at which the model should still attempt to split the data. The lower this number is, the purer the data will be in each leaf node, however having a number too low can cause overfitting.

Gradient Boosted Parameters

Loss

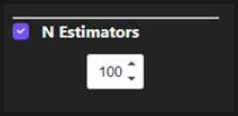


You can choose which loss function to use for your model. Loss is a measure of "how good" each weak learner is a making predictions about your data. You can find out more about the different loss functions here.

Learning Rate

Reducing the learning rate will improve the generalization of the model. There is a point at which you will receive diminishing gains and the time to train the model increases the lower the learning rate is. The default value is 0.1.

N Estimators

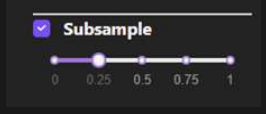


This parameter determines the number of “boosting stages to perform”. In other words, it tells the algorithm how many “weak learners” to create for the ensemble based model. The default is 100.

Subsample

Validation Fraction

N Iter No Change



For each iteration of the algorithm, a subsample of the training data is selected to train the model. You can change the size of this subsample by using the slider. The default value is set to 1.0 meaning use all the data.

The validation fraction is the proportion of the training data to be set aside to assess the validation loss of the model. The training will stop if in the last N iterations (specified by n iter no change) the model has not improved by at least the amount specified by the tolerance parameter. By default, this parameter is set to 0.1, but it is not used unless you supply an integer value to n iter no change.

This allows you to specify the number of past iterations of the model to be used to assess the validation loss. If the loss has not improved by the amount specified by tolerance, over the number of iterations you specify here, then training will finish. The default is not set.

Tolerance

You can specify the tolerance (which is a measure of loss) at which training should finish, if in the past N (N iter No Change) iterations the loss has not decreased by more than the amount specified.

Random Forest Parameters

Bootstrap

By default, the bootstrap parameter is set to True. You can turn off bootstrapping, Which will have the effect of training each tree in the ensemble using the same sample of data at each stage.

Max Samples

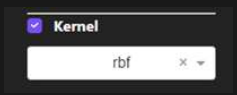
This determines what fraction of the data is passed to any individual tree within the model during training. It can only be used when bootstrap is set to true. By default, it is not set. values in the range 0.01 - 1 are accepted.

SVM Parameters

C

The C parameter adds a penalty for each misclassified data point. When c is small this penalty is kept low resulting in a decision boundary with a large margin, however the caveat is tere will be a greater number of misclassifications. Higher c values result in a tighter fitting boundary.

Kernel



Specifies the kernel type to be used in the algorithm. For more information on the differences between kernels here is a useful link.

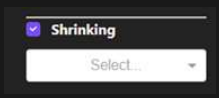
Degree

If you use the "poly" kernel, this parameter allows you to specify the degree of the polynomial function. The default is 3.

Gamma

Coefficient 0

Shrinking

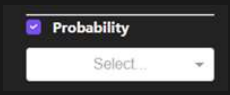


The default is scale. When gamma is set to "scale" a value of 1 / (number of features * variance) will be used. For auto this value is set to 1 / number of features. A higher gamma value means only the closest points to the decision boundary will carry weigth leading to a smoother boundary.

This is an independent term in the kernel function. It is only significant in 'poly' and 'sigmoid' kernels. Most of the time it won't be necessary to change this paramater but in short it is used to balance how much the model is influenced by high or low degree polynomials.

Here you can specify whether to use the shrinking heuristic. When used on large datasets, it can potentially optimise training time, however this is not always the case. Altenatively, you can decrease training time by increasing the tolerance parameter.

Probability



Whether to enable probability estimates. for more information click here.

Cache Size

Specify the size of the kernel cache (in MB). The size of the kernel cache has a strong impact on training times for larger problems. If you have enough RAM available, it is recommended to set this to a higher value than the default of 200(MB), such as 500(MB) or 1000(MB).

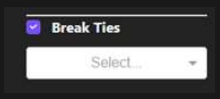
Max Iter

This controls how many steps the model will take in the gradient descent before giving up. The algorithm will stop when either updates to accuracy are within the margin you sepcify for tolerance or you've run for max iter many steps.

Decision Function Shape

Whether to return a one-vs-rest or one-vs-one decision function. More details can be found here.

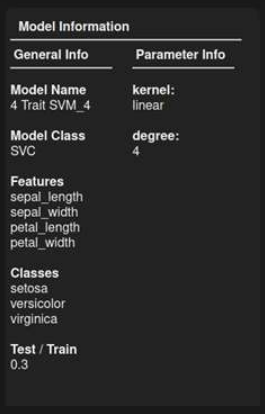
Break Ties

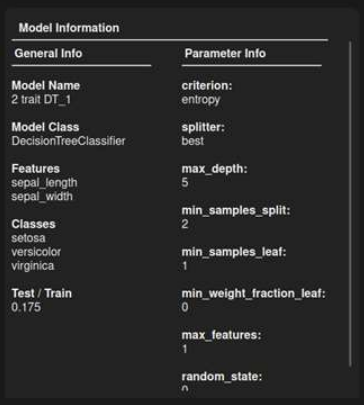


If true, decision function shape='ovr', and the number of classes > 2, predict will break ties according to the confidence values of decision_function; otherwise the first class among the tied classes is returned.

Visualisation Features

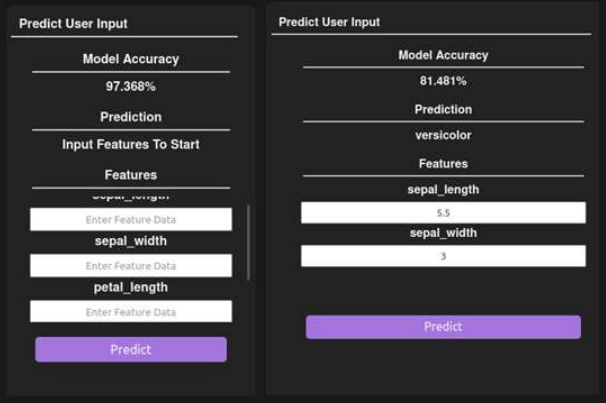
Model Information





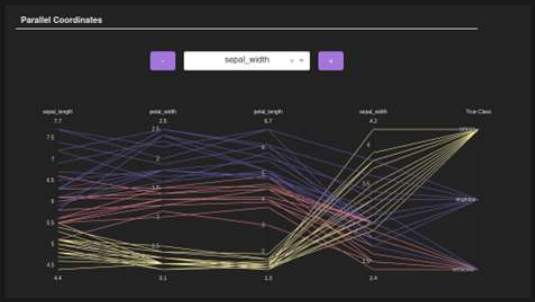
The model information component is used as a display for all input information. This includes the name of the model, the features the model is trained on, what parameters were used, the test train split and all of the classes identified in the csv file. The examples above show different machine learning models after training, one an SVC and the other a decision tree.

Predict User Input



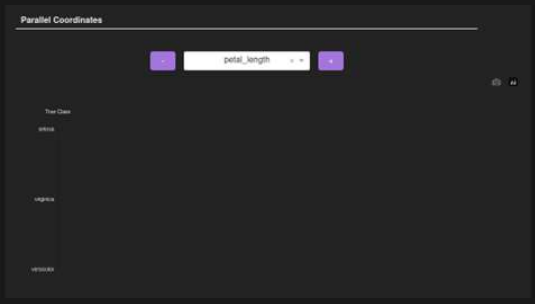
After training the model this section will show the accuracy of the model along with a field for each of the selected training features. As shown in the example Numerical values can be entered into the training feature boxes. Once you fill out all the training features and click predict, it will enter the values into the machine learning model and show the predicted class. The predictions will only be as accurate as the model you have trained.

Parallel Coordinates



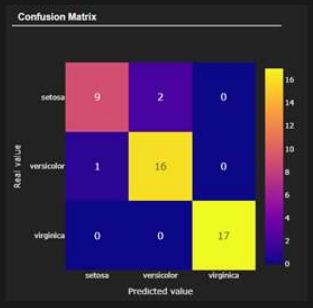
The parallel coordinates component acts as a method to visualise any number of input features. This component solves the problem of visualising more than three dimensions on a graph.

Each line represents a single data instance from the testing dataset, with its colour representing the class that the machine learning model has predicted. Each axis correspons to one of the training fetaures. The line will go to a value on the axis representing its own value for that feature. Compared the colour of the lines allows visualising how the model groups data for each individual classification.



By default, after a model is trained the parallel plot is empty as no axis' have yet been added to it. Above is a dropdown box containing all features the model was trained on. The buttons either side will add the currently selected feature to the plot, or if it is already on the plot and the remove button is clicked, that axis will be removed from the graph. If add is pressed when the feature is already on the graph or remove when a feature is not on the graph, nothing will happen.

Confusion Matrix



The confusion matrix is located just below the top section of the site, only shown after the model has been trained. It is a table that shows the performance of your decision tree model by comparing the actual values

(labelled on the y-axis) of your dataset to the predicted values generated from the model (labelled on the x-axis).

The number of classifications is shown in each cell and through colour, where low values are displayed in blue to purple, moderate amounts in pink, and higher values in orange to yellow.

Depending on the number of classes in your dataset, the confusion matrix will display an appropriate matrix size such as 2 by 2 or 3 by 3. The diagonal of the matrix represents the true positives and true negatives (correctly classified instances), while the off-diagonal elements represent the false positive and false negative predictions.

The confusion matrix is a powerful tool that can help you understand the performance of your model in more detail. It allows you to identify areas of the dataset where the model is performing well and areas where it may be struggling. A model may have a high accuracy score yet also miss-classify specific subsets of data very frequently.

Understanding the performance of your decision tree model through the confusion matrix can help you improve your model's accuracy and optimize its performance for your specific use case.

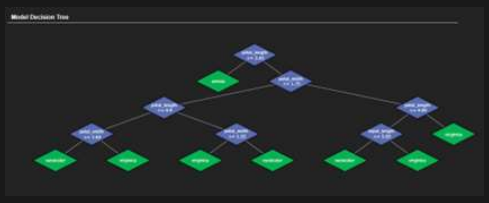
Decision Boundary visualization

This visualization includes an x and y-axis labelled with the feature names you have selected. Each instance is colour-coded to help you easily distinguish between different classes. Additionally, each data instance has a unique shape corresponding to the class it belongs to. beneath the visualization, there is a key that displays the class names and their respective colours.

You can use the following functions to interact with the Decision Boundary Visualization:

- Zoom in and out: Use the zoom in and out buttons to zoom in and out of the feature space.
- Pan/move: Click and drag the feature space to move around and explore different parts of the graph.
- Download as a PNG: You can download the visualization as a PNG file for further use.
- Box and lasso selection: You can select specific data points on the graph using the box and lasso selection tools. This is useful if you want to isolate and analyze a specific subset of the data.
- Auto scale: Clicking on the "autoscale" button will adjust the axis range automatically to fit the data.
- Reset axes: Clicking on the "reset" button will reset the x and y-axes back to their original range.

Model Decision Tree

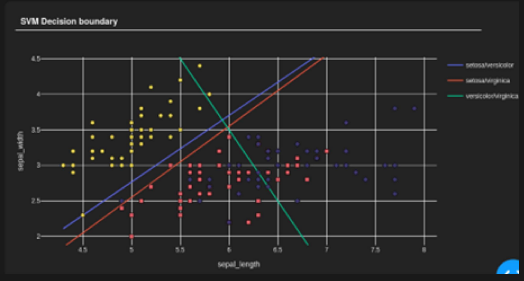


The Decision Tree visualisation allows you to view the structure of your decision tree model. The decision nodes (non-leaf nodes) are highlighted in blue, while the leaf nodes are highlighted in green. Additionally, you can hover over the nodes to view their Gini values. The size of the nodes changes to ensure that the entire tree can be viewed, regardless of its size.

You can also use the built-in Plotly functionalities, such as downloading as a png to closely analyse large trees. The functionalities are explained in the decision boundary visualisation section of the user manual to interact with the Model Decision Tree.

Using the Model Decision Tree visualisation along with the feature space can allow you to gain a better understanding of how your decision tree model is making decisions based on the features in your dataset. This information can be valuable for identifying potential areas for improvement or optimization in your model.

SVM Decision Boundary



The SVM decision boundary component is unique to SVM models. It shows the hyperplanes generated by the model where a single hyperplane separates two unique classes. The model will predict data dependent on which side of the hyperplane it is plotted. The legend shows which two classes each hyperplane separates.

The SVM decision boundary component is currently limited. It is only able to visualise models with two training features that are trained with a linear SVM kernel.

Troubleshooting Guide

This guide will highlight the issues a user may come across whilst using the website. Any issue not covered here will likely be resolved by refreshing the website.

Uploading

"Wrong File Type!"

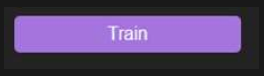
A dataset must be of the filetype ".csv" to be accepted as valid input.

Training Features Not displayed.

Check the uploaded CSV and ensure it contains column names and data.

Training

Training Button Is Not Responsive



Refresh the website and reupload your dataset.

"You Must Select a Model Class"

A model class must be selected, the dropdown menu beneath the heading "Model" can be used to do this.

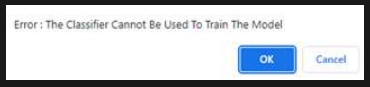
The trained model does not use the specified parameters.

Make sure that the check box next to the parameter you are trying to use has been highlighted. The model will not use any parameter that has not been selected, even if it has taken user input.

"You Cannot Select More Than One Classifier"

Classifier models work by mapping input data to a specific category, as such check to ensure that only one heading under "Classifier" has been selected.

"The Classifier Cannot Be Used to Train the Model"



Classifier models map input data to a specific category, as such only categorical data can be selected as the target Classifier. Check to make sure the selected Classifier contains only categorical data.

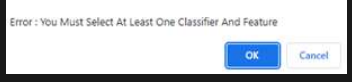
"You Need to Provide a Filename"

Check that the input text box above the Train button contains text. For a model to be trained it needs to have a user inputted name.

"A Selected Parameter Has No Value"

"You Must Select At Least One Classifier and Feature"

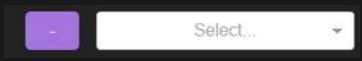
Check all selected parameters. If a parameter was selected by accident uncheck the box next to it. If the parameter was intentionally selected, then ensure it has an associated value.



For a model to be trained it needs a Classifier and a minimum of one Feature selected. The first name of each column from your CSV are used as identifiers. These can be selected under the headings "Training Features" and "Classifier". If you do not see anything under these headings, check that your CSV has been successfully uploaded, and check the contents of the CSV to ensure each column has a heading.

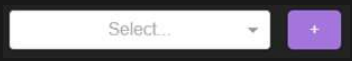
Parallel Coordinates Component

The remove (-) button is non-responsive



Check to ensure that a feature is selected (using the dropdown menu) and that it currently exists on the parallel coordinates diagram.

The add (+) button is non-responsive



Check to ensure that a feature is selected (using the dropdown menu) and that it does not currently exist on the parallel coordinates diagram.

Predict User Input Component

The predict button is non-responsive.

Check to ensure that all provided features have an input value before pressing predict. The model will not predict an outcome unless all features that it was trained with have a provided value.

Current Limitations

Currently the website is limited in a few ways. For instance when training an SVM model we currently only support displaying the decision boundary when a linear kernel is used on two training feature inputs. Because of the amount of computational complexity that is required to train a machine learning model, you may also notice a slowdown when the dataset you use to train you model is particularly large.

You my also notice that we only accept csv files as data input. In some cases these csv files may not be suitable for training. Trailing commas can cause errors when training and generatating visualisations, so you must make sure your csv file is correctly formated.

Currently only classification based models are supported, however support for regression based models, semi-supervised learning and neurel networks may be added in the future.

Local Install Guide

To run the program, you will need docker installed. Installation instructions for all supported operating systems can be found here.

- Cd into team44_project.
- Run the following command, “aiVis” provides the tag for the image and can be changed: docker build -t aiVis.
- If you get a permission denied you can either run the command with sudo or add docker to the appropriate user group. Instructions can be found here.

You can view your currently built images by running: docker images

Run the following command to start the container

- docker run -p 8080:80 aiVis
- Navigate to: “localhost:8080” In your internet browser URL bar