Computing Assignment 1:
Report

Task 1:

- **Task description:**

This task was about building a binary decision tree to classify a dataset with 3 classes, while showing as much details in the intermediate steps as possible. To complete this task, I read about the use and construction algorithm of decision trees (part 1), I modified and completed the given code to use information gain as a split measure (part 2), and investigated ways to display the intermediate split tests and resulting trees programmatically (part 3).

- **Part 1: About decision trees**

Decision trees are a supervised learning algorithm used in both regression and classification tasks [1]. It is considered to be one of the simplest such algorithms because of its high interpretability and relatively simple implementation. At its core, a decision tree is a series of tests on data attributes, inductively splitting the dataset into homogeneous groups until such groups are 'pure' (i.e., all their elements belong to a single class), or we reach a predefined depth or impurity ratio [1]. This series of tests lends itself intuitively to a tree representation, hence the name "decision trees".

An important decision to make when creating a decision tree is the criteria based on which we will choose the decision tests. In general, we follow a greedy approach: we would like each test to be as well performing as possible on its corresponding dataset, regardless of the performance of subsequent tests. In general, the performance of a test is measured through the purity of its resulting splits. There are many metrics to evaluate this purity. Some of the metrics used in classification problems for instance are:

- **Error rate:**

The error rate is the ratio of data points not belonging to the majority class as revealed by a particular test. We consider these data points to be misclassified. The ratio of misclassified data is calculated as:

$$1 - \frac{number\ of\ nodes\ in\ majority\ class\ of\ a\ branch}{number\ of\ nodes\ in\ all\ classes\ of\ a\ branch} = 1 - \frac{\max N(b,c)}{Nb}$$

We take the weighted sum of the error rate in each branch to get the error rate of the test, as shown in the lecture notes:

$$\text{error rate} \qquad \sum_{b\in\{\text{yes},\text{no}\}} \frac{1}{N}\left(N_{\text{b}} - \max_{c\in\{\bullet,\bullet\}} N_{\text{b,c}}\right)$$

The error rate alone is not enough to generate high performing compact trees [2]. More advanced metrics originating from information theory are used as default measures in state-of-the-art decision trees.

- **Gini index:**

Gini index measures the likelihood of mislabeling a random data point if it were labeled according to the labels' distribution in the dataset [3]. This is quantified by multiplying the probability of taking a data point of a class 'i' times the probability of 'misclassifying it' [4]:

$$Gi = p_i * (1-p_i)$$

P_i is equal to the number of data points from class (i) over the total number of data points in a branch/split (from all target classes). To get the gini index of a branch, we sum the gini index for every class in that split, then we do a weighted sum across branches to get the score of a test/node. The resulting formula was given in the lecture notes as:

$$\text{Gini index} \quad \sum_{b\in\{\text{yes,no}\}} \frac{N_b}{N}\left(1 - \sum_{c\in\{\bullet,\bullet\}} \left(\frac{N_{b,c}}{N_b}\right)^2\right)$$

- **Information gain:**

Information gain is the most commonly used split measure in classification trees. Intuitively, it measures the amount of information gained through a particular test, by subtracting the information score of its children from that of its parent.

$$\overbrace{IG(T,a)}^{\text{Information Gain}} = \overbrace{\mathrm{H}(T)}^{\text{Entropy (parent)}} - \overbrace{\mathrm{H}(T|a)}^{\text{Weighted Sum of Entropy (Children)}}$$

The equation answers the following question: *how much more do we know about the dataset given a particular value of a given attribute?* The concept of information is measured by Shannon's entropy, which first appeared in information and communication theory. In short, entropy measures the randomness of a given dataset by calculating the number of bits necessary to transmit the information contained in said dataset [6]. The equation for entropy is thus:

$$\log\left(\frac{\textit{nodes in a class}}{\textit{nodes in a branch}}\right) = \log\left(\frac{N(b,c)}{N(b)}\right)$$

The entropy across multiple classes is the weighted sum of the entropies of each class.

$$e(C) = \sum_{i}^{C} -\frac{N(b,i)}{N(b)} * \log\left(\frac{N(b,i)}{N(b)}\right)$$

The entropy across splits/branches is also generated from a weighted sum:

$$H(T|test) = \sum_{i}^{C} \frac{N(b)}{N} * e(C)$$

The information gain of the parent is:

$$H(T) = \sum_{i}^{C} -\frac{N(i)}{N} * \log\left(\frac{N(i)}{N}\right)$$

Combining both formulas, we get the expression given in the lecture notes:

$$\text{information gain} \quad \sum_{c\in\{\bullet,\bullet\}} -\frac{N_c}{N}\log_2\left(\frac{N_c}{N}\right) - \sum_{b\in\{\text{yes,no}\}} \frac{N_b}{N} \sum_{c\in\{\bullet,\bullet\}} -\frac{N_{b,c}}{N_b}\log_2\left(\frac{N_{b,c}}{N_b}\right)$$

- **Part 2: Information gain in our task**

In what follows, I will explain how a single score of information gain is computed for one split candidate (using the formulas given above), then comment on how these formulas are implemented and point to their location in the code.

Let's try to apply the information gain formula to the second split candidate: attribute v1 with threshold value 1 (t_1: $v_1 >= 1$). We have 3 classes, so we expect the formulas to be the weighted sums of the entropy of each class. The parent class distribution in this case is that of the original dataset: 3 blues, 5 red, and 4 yellow. Using the formula for the entropy of the parent node, we get:

$$\mathrm{H(T)} = -\frac{3}{12} * \log\left(\frac{3}{12}\right) - \frac{5}{12} * \log\left(\frac{5}{12}\right) - \frac{4}{12} * \log\left(\frac{4}{12}\right) = 1.5546$$

We compute then the entropy of the split we get using this test.

Entropy for branch 'no' with distribution 0/1/0: $H(T|t_1, \text{no}) = -\frac{1}{12} * \log\left(\frac{1}{12}\right) = 0.2987$

Entropy for branch 'yes' with distribution 3/4/4: $H(T|t_1, \text{yes}) = -\frac{3}{12} * \log\left(\frac{3}{12}\right) - \frac{4}{12} * \log\left(\frac{4}{12}\right) - \frac{4}{12} * \log\left(\frac{4}{12}\right) = 1.5566$

Total entropy for the children of t_1: $H(T|t_1) = \frac{1}{12} * H(T|t1, \text{no}) + \frac{11}{12} * H(T|t1, \text{yes}) = 1.4518$

Information gain for t_1: $H(t_1) = H(T) - H(T|t_1) = 1.5546 - 1.4518 = 0.1028$

This way we get the score as reported on the table (the difference in the value is due to rounding off). We repeat these calculations for every split candidate, i.e. every pair of (attribute, value) possible from the training set. The test chosen at every iteration is the one with the highest value. After every iteration, we recompute all scores to choose a new candidate.

This procedure is implemented in file 'resources.py', line 141 (function called 'entropy'). I also changed the implementation of the function 'information_gain' (*resources.py*, line 163) to return positive gain values, and changed the condition for choosing the best split in 'get_split' function to get the highest of positive values (*resources.py*, line 229). These changes allowed generating split measure tables similar to the ones provided in the assignment handout.

- **Part 3: visualization of split measure tables and corresponding trees**

The task asked for detailed computations of split measure scores for all split candidates at a given step, following the model given in the handout. The simplest way to generate such data was to extract this information from the tree construction code/functions and print it in a table following the suggested pattern. I modified the 'get_split' function to store the split candidates it considers at every step (*resources.py*, line 220), then modified 'split' function to log every run of the algorithm (including the run number, the candidates considered, and the candidate/test chosen) by adding it to a list of steps to be printed later on (*resources.py*, line 259). The printing/displaying happens in function 'build_tree' which calls the function 'disp_tree' for parsing the run data, formatting the figures and plots, and saving everything on a pdf file (*resources.py*, line 374).

Initially, I planned to log all the information of a run using plotting utilities from matplotlib. I faced many issues with formatting, so I settled for printing the tables and adding the other information manually. I also explored the possibility of the plotting the tree programmatically using network [7] and tree-plot [8], but decided at last that it would be a lot more efficient to draw the trees manually since I have no experience with figure generation.

The generated tables and trees can be found in Appendix A. The source codes I used for plotting (and any documentation/articles I consulted in the process) are cited in sources.

Task 2:

- **Task description:**

In this task, we were asked to use the decision tree built in Task 1 to classify iris flowers from the dataset irisSV. To perform this classification, we need to train the classifier on a portion of the data (4/5), then test it on the remaining unseen portion (1/5). The hyper parameters given for the decision tree were maximum depth of 5 (the classifier contains at most 5 split tests) and minimum leaf size of 3 (minimum number of data points contained in a leaf). After training and testing the classifier, we were asked to report the following evaluation metrics: confusion matrix, accuracy, recall and precision. The report below provides the requested values, points to the code modifications I made to generate them, and explains the significance of the evaluation metrics when relevant.

- **Solution:**

The codes used to compute the requested metrics (confusion matrix, accuracy, recall and precision) can be found in *resources.py* from line 82 to line 103. The generated values are shown below:

- **Confusion matrix:**

	Predicted ‘Setosa’ (positive)	Predicted ‘Versicolor’ (negative)
Actual ‘Setosa’ (positive)	8 (TP)	0 (FN)
Actual ‘Versicolor’ (negative)	0 (FP)	9 (TN)

- **Recall [9]:**

Recall answers the question: *among the positive data points in our set, how many did we capture (or recall) as positive?* The answer lies in the formula $\frac{TP}{TP+FN}$, where TP + FN is the total ‘true’ data points we have in our set.

In this run of the algorithm, we get a recall value of: 1.0

- **Precision [9]:**

Precision answers the question: *from all the data points we recognized as positive, how many are actually positive?* i.e., *how precise is our identification of positive points?* The formula used to answer this question is $\frac{TP}{FP+TP}$.

The precision value in this example is: 1.0

- **Accuracy:**

Accuracy quantifies the correctness of the predictions of a model. It is the fraction of correctly predicted values over the total predicted values: $\frac{TP+TN}{N}$

The value we get for this task is: 1.0

Task 3:

- **Task description:**

This task was an exploration of two types of Support Vector Machines (SVMs): hard-margin and soft-margin. We were asked to implement both algorithms, train and test them on separate data sets, and report the equations of the separating hyperplanes and corresponding plots, along with the accuracy of each model. The following report provides a brief overview of SVMs (part 1), with a particular focus on the two variants used in the task, along with a report on the values/plots requested (part 2).

- **Part 1: Theory**

SVMs belong to the family of linear models, which try to define a linear hyperplane to separate different data classes. In practice, these models try to define the equation of the hyperplane $W.x + b = 0$, by estimating the parameters W and b [15].

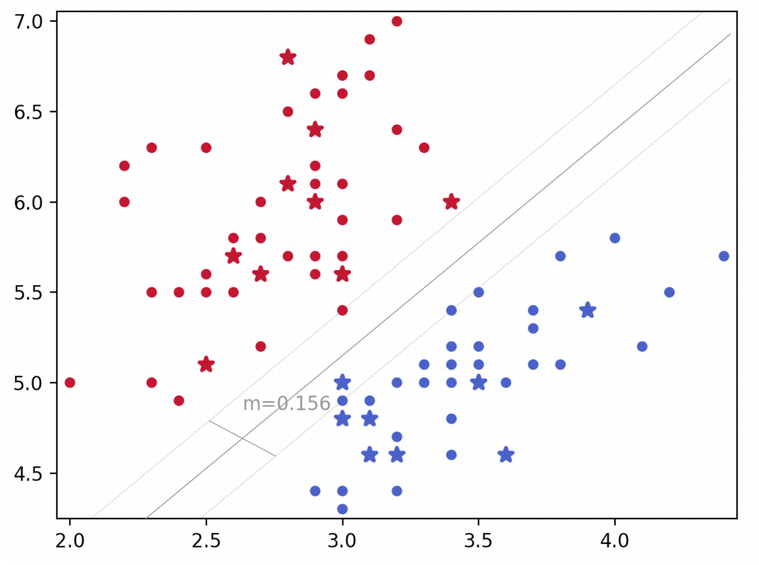
Different linear models propose different algorithms for estimating the ‘best’ W and b. SVMs seek to maximize the margin between the separating hyperplane and the data points closest to it, under the constraint of correct classification. The idea is to stay as far as possible from the ‘danger’ zone that is the area where data points from the two classes are closest to one another, making classification of unseen data points particularly difficult. Finding the value of W which meets these conditions comes down to a quadratic optimization problem, solved using Lagragian classifiers. In the process of finding W, we can identify support vectors (the points closest to the hyperplane from either side) and use their values in the equation of their respective hyperplanes to solve for b [15 & 16].

In a perfectly linearly-separable dataset, a hyperplane which separates the two classes and maintains the widest margin leads to the best classification results. In practice however, datasets rarely fit this scenario, and we usually can tolerate a few misclassified points and outliers, as long as the classification and its generalization are reasonable. This is where the ‘soft-margin’ variant comes to play, by adding ‘slack variables’ (can be thought of as measuring the distance between points within the margin and the separating hyperplane) onto the optimization algorithm of SVMs, weighted by a value C, which allows us to control the flexibility of our margins. As C tends to infinity, we fall back to the hard-margin variant of the algorithm [17 & 18].

- **Part 2: SVMs applied to the task at hands**

- **Hard-margin SVMs:**

When applying hard-margin SVM to the irisSV dataset, we get a hyperplane in the right direction (slope), but with a skewed position (bias). After modifying the code as suggest by the comment (*resources.py*, line 476, function prepare_svm_model()), we get a correct hyperplane plot and an accuracy of 1.

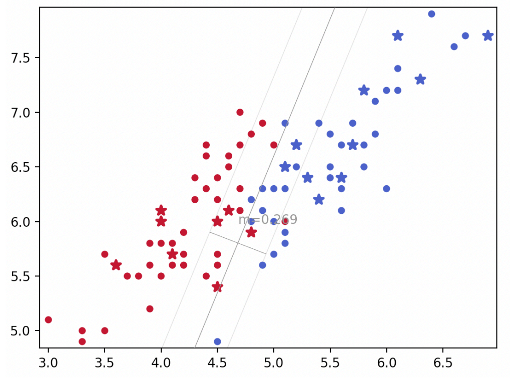


The equation corresponding to this hyperplane can be retrieved from the coeffs variable inside visu_plot_svm (*resources.py*, line 502). The equation of the hyperplane above is:

$$5.0 * x1 + -5.0 * x2 - 11.0 = 0$$

- **Soft-margin SVM:**

The soft-margin variant, when applied to the irisVV dataset, returns the plot shown below.



To get this model, we had to use the code for the multipliers provided in the compute_multipliers function, but initially commented out (*resources.py*, line 441). The accuracy score reported for this model is 0.94. The equation of the given hyperplane is:

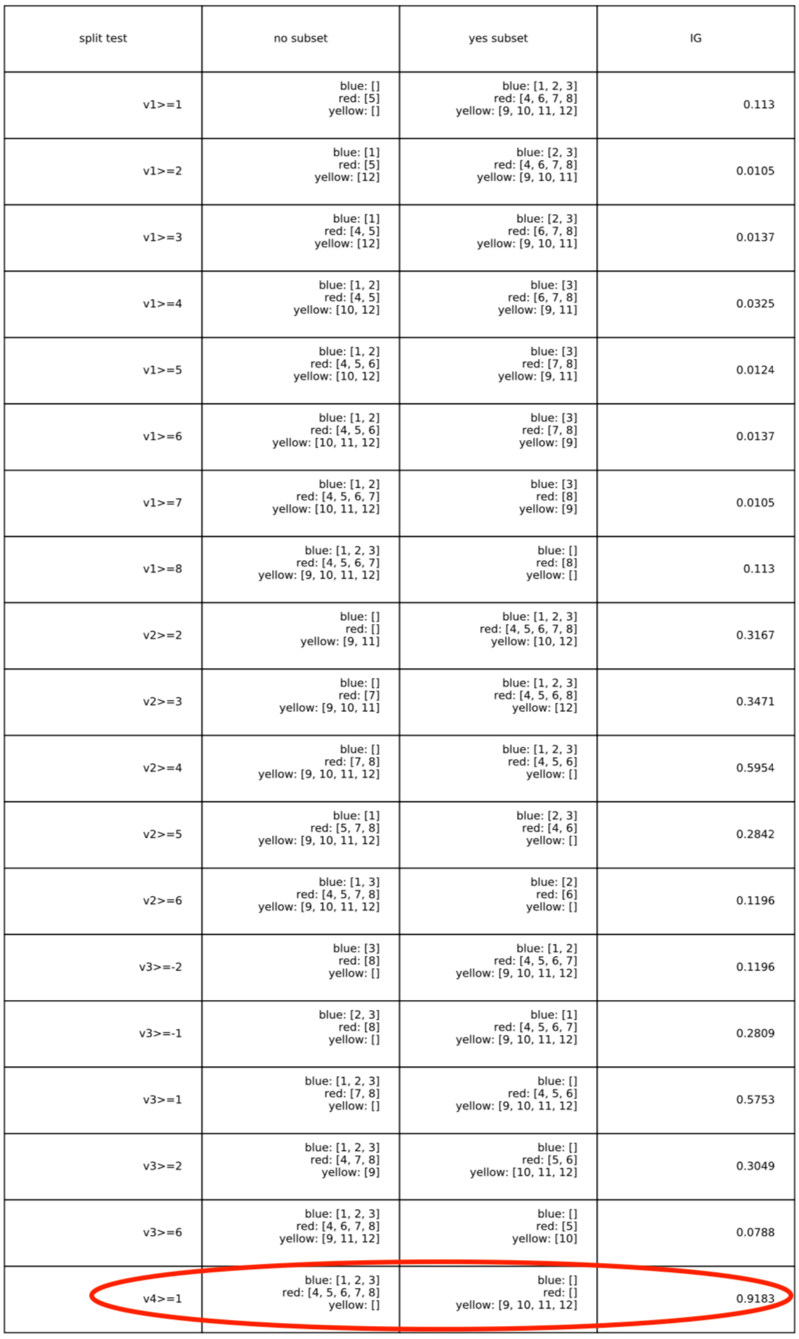
$$1.38 * x1 - 3.45 * x2 + 8.16 = 0$$

Appendix A: tables and tree diagrams for decision tree

Note: the chosen test at each step, corresponding to the candidate with the highest information gain, is circled in red.

Step 1:

Table 1 split measures for candidates in step 1



split test	no subset	yes subset	IG
v1>=1	blue: [] red: [5] yellow: []	blue: [1, 2, 3] red: [4, 6, 7, 8] yellow: [9, 10, 11, 12]	0.113
v1>=2	blue: [1] red: [5] yellow: [12]	blue: [2, 3] red: [4, 6, 7, 8] yellow: [9, 10, 11]	0.0105
v1>=3	blue: [1] red: [4, 5] yellow: [12]	blue: [2, 3] red: [6, 7, 8] yellow: [9, 10, 11]	0.0137
v1>=4	blue: [1, 2] red: [4, 5] yellow: [10, 12]	blue: [3] red: [6, 7, 8] yellow: [9, 11]	0.0325
v1>=5	blue: [1, 2] red: [4, 5, 6] yellow: [10, 12]	blue: [3] red: [7, 8] yellow: [9, 11]	0.0124
v1>=6	blue: [1, 2] red: [4, 5, 6] yellow: [10, 11, 12]	blue: [3] red: [7, 8] yellow: [9]	0.0137
v1>=7	blue: [1, 2] red: [4, 5, 6, 7] yellow: [10, 11, 12]	blue: [3] red: [8] yellow: [9]	0.0105
v1>=8	blue: [1, 2, 3] red: [4, 5, 6, 7] yellow: [9, 10, 11, 12]	blue: [] red: [8] yellow: []	0.113
v2>=2	blue: [] red: [] yellow: [9, 11]	blue: [1, 2, 3] red: [4, 5, 6, 7, 8] yellow: [10, 12]	0.3167
v2>=3	blue: [] red: [7] yellow: [9, 10, 11]	blue: [1, 2, 3] red: [4, 5, 6, 8] yellow: [12]	0.3471
v2>=4	blue: [] red: [7, 8] yellow: [9, 10, 11, 12]	blue: [1, 2, 3] red: [4, 5, 6] yellow: []	0.5954
v2>=5	blue: [1] red: [5, 7, 8] yellow: [9, 10, 11, 12]	blue: [2, 3] red: [4, 6] yellow: []	0.2842
v2>=6	blue: [1, 3] red: [4, 5, 7, 8] yellow: [9, 10, 11, 12]	blue: [2] red: [6] yellow: []	0.1196
v3>=-2	blue: [3] red: [8] yellow: []	blue: [1, 2] red: [4, 5, 6, 7] yellow: [9, 10, 11, 12]	0.1196
v3>=-1	blue: [2, 3] red: [8] yellow: []	blue: [1] red: [4, 5, 6, 7] yellow: [9, 10, 11, 12]	0.2809
v3>=1	blue: [1, 2, 3] red: [7, 8] yellow: []	blue: [] red: [4, 5, 6] yellow: [9, 10, 11, 12]	0.5753
v3>=2	blue: [1, 2, 3] red: [4, 7, 8] yellow: [9]	blue: [] red: [5, 6] yellow: [10, 11, 12]	0.3049
v3>=6	blue: [1, 2, 3] red: [4, 6, 7, 8] yellow: [9, 11, 12]	blue: [] red: [5] yellow: [10]	0.0788
v4>=1	blue: [1, 2, 3] red: [4, 5, 6, 7, 8] yellow: []	blue: [] red: [] yellow: [9, 10, 11, 12]	0.9183

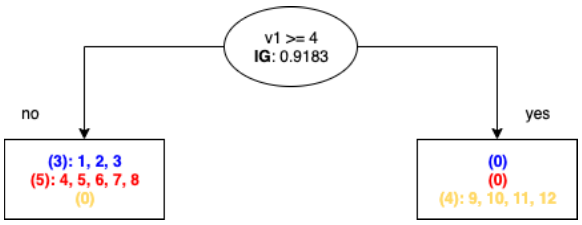
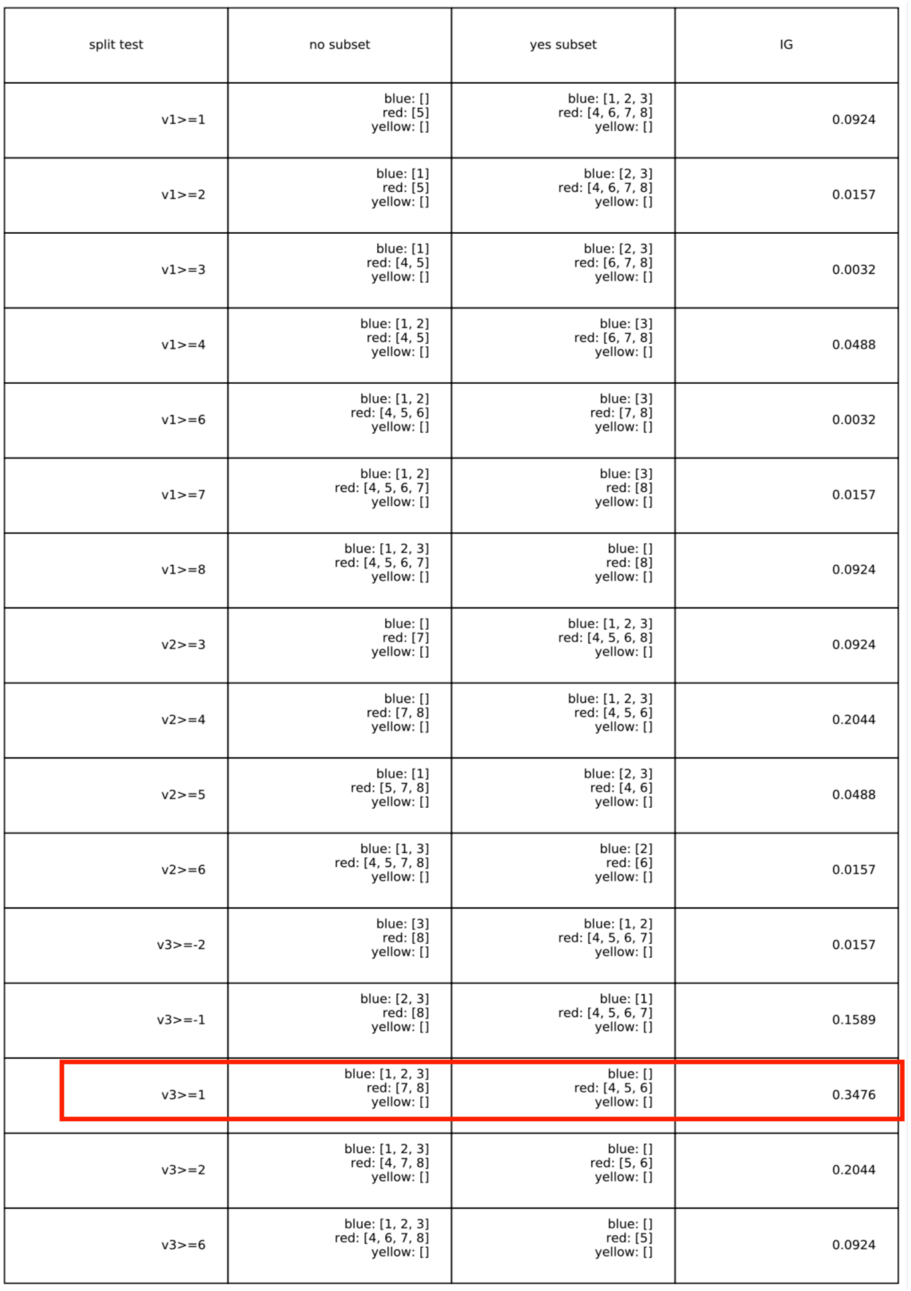


Figure 1 decision tree at step 1

Step 2:

Table 2 split measures for step 2



split test	no subset	yes subset	IG
v1>=1	blue: [] red: [5] yellow: []	blue: [1, 2, 3] red: [4, 6, 7, 8] yellow: []	0.0924
v1>=2	blue: [1] red: [5] yellow: []	blue: [2, 3] red: [4, 6, 7, 8] yellow: []	0.0157
v1>=3	blue: [1] red: [4, 5] yellow: []	blue: [2, 3] red: [6, 7, 8] yellow: []	0.0032
v1>=4	blue: [1, 2] red: [4, 5] yellow: []	blue: [3] red: [6, 7, 8] yellow: []	0.0488
v1>=6	blue: [1, 2] red: [4, 5, 6] yellow: []	blue: [3] red: [7, 8] yellow: []	0.0032
v1>=7	blue: [1, 2] red: [4, 5, 6, 7] yellow: []	blue: [3] red: [8] yellow: []	0.0157
v1>=8	blue: [1, 2, 3] red: [4, 5, 6, 7] yellow: []	blue: [] red: [8] yellow: []	0.0924
v2>=3	blue: [] red: [7] yellow: []	blue: [1, 2, 3] red: [4, 5, 6, 8] yellow: []	0.0924
v2>=4	blue: [] red: [7, 8] yellow: []	blue: [1, 2, 3] red: [4, 5, 6] yellow: []	0.2044
v2>=5	blue: [1] red: [5, 7, 8] yellow: []	blue: [2, 3] red: [4, 6] yellow: []	0.0488
v2>=6	blue: [1, 3] red: [4, 5, 7, 8] yellow: []	blue: [2] red: [6] yellow: []	0.0157
v3>=-2	blue: [3] red: [8] yellow: []	blue: [1, 2] red: [4, 5, 6, 7] yellow: []	0.0157
v3>=-1	blue: [2, 3] red: [8] yellow: []	blue: [1] red: [4, 5, 6, 7] yellow: []	0.1589
v3>=1	blue: [1, 2, 3] red: [7, 8] yellow: []	blue: [] red: [4, 5, 6] yellow: []	0.3476
v3>=2	blue: [1, 2, 3] red: [4, 7, 8] yellow: []	blue: [] red: [5, 6] yellow: []	0.2044
v3>=6	blue: [1, 2, 3] red: [4, 6, 7, 8] yellow: []	blue: [] red: [5] yellow: []	0.0924

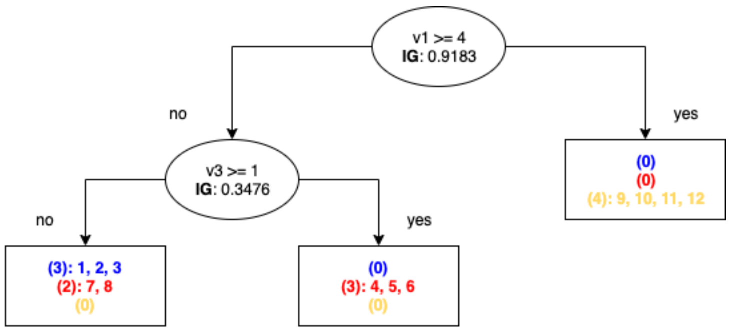
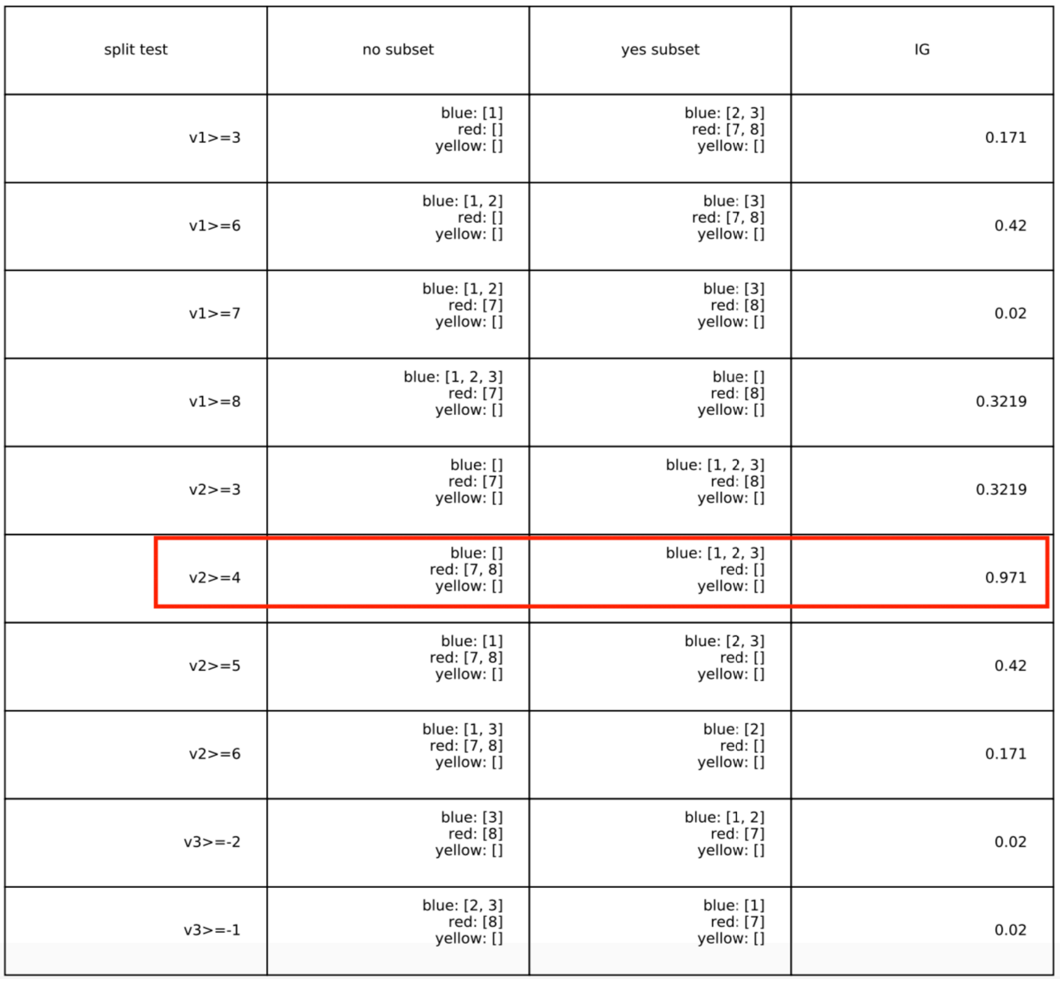


Figure 2 decision tree at step 2

Step 3:

Table 3 split measures for step 3



split test	no subset	yes subset	IG
v1>=3	blue: [1] red: [] yellow: []	blue: [2, 3] red: [7, 8] yellow: []	0.171
v1>=6	blue: [1, 2] red: [] yellow: []	blue: [3] red: [7, 8] yellow: []	0.42
v1>=7	blue: [1, 2] red: [7] yellow: []	blue: [3] red: [8] yellow: []	0.02
v1>=8	blue: [1, 2, 3] red: [7] yellow: []	blue: [] red: [8] yellow: []	0.3219
v2>=3	blue: [] red: [7] yellow: []	blue: [1, 2, 3] red: [8] yellow: []	0.3219
v2>=4	blue: [] red: [7, 8] yellow: []	blue: [1, 2, 3] red: [] yellow: []	0.971
v2>=5	blue: [1] red: [7, 8] yellow: []	blue: [2, 3] red: [] yellow: []	0.42
v2>=6	blue: [1, 3] red: [7, 8] yellow: []	blue: [2] red: [] yellow: []	0.171
v3>=-2	blue: [3] red: [8] yellow: []	blue: [1, 2] red: [7] yellow: []	0.02
v3>=-1	blue: [2, 3] red: [8] yellow: []	blue: [1] red: [7] yellow: []	0.02

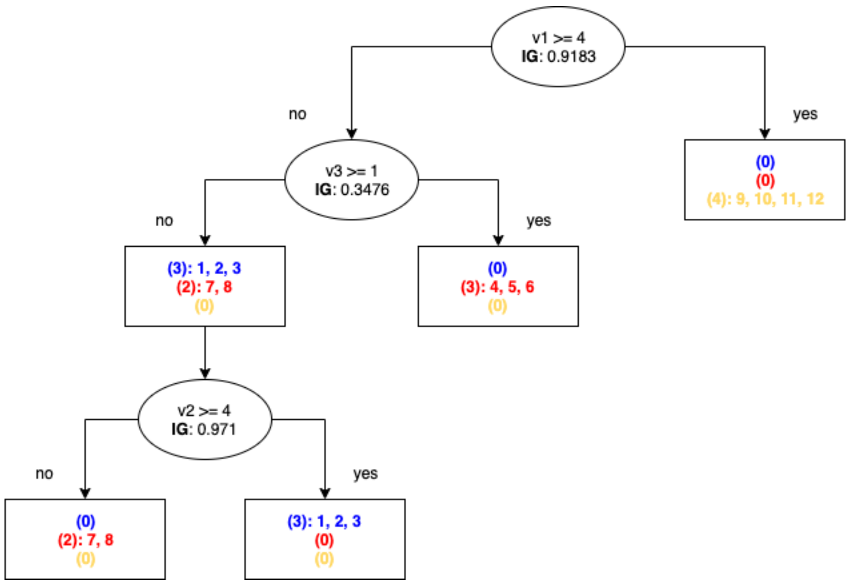


Figure 3 final decision tree

Collaborators

None.

Sources

Note: I did not cite the textbook or lecture slides 'academically' in this report. My impression was that the citations in these assignments are informal. Please correct me if I am wrong, and I will be more mindful in upcoming submissions.

In addition to the textbook and lecture notes, I used the following sources in this assignment.

Task 1: Decision trees

[1] PEIXEIRO, M. (2019, JANUARY 16). EVERYTHING YOU NEED TO KNOW ABOUT DECISION TREES. RETRIEVED JANUARY 3, 2020, FROM https://towardsdatascience.com/everything-you-need-to-know-about-decision-trees-8fcd68ecaa71.

[2] Why we use information gain over accuracy as splitting criterion in decision tree? (2016, October 10). Retrieved January 3, 2020, from https://datascience.stackexchange.com/questions/14433/why-we-use-information-gain-over-accuracy-as-splitting-criterion-in-decision-tre.

[3] Agarwal, R. (2019, September 29). The Simple Math behind 3 Decision Tree Splitting criterions. Retrieved January 3, 2020, from https://towardsdatascience.com/the-simple-math-behind-3-decision-tree-splitting-criterions-85d4de2a75fe.

[4] Decision tree learning. (n.d.). Retrieved January 3, 2020, from https://en.wikipedia.org/wiki/Decision_tree_learning#Gini_impurity.

[5] T, S. (2019, January 11). Entropy: How Decision Trees Make Decisions. Retrieved January 3, 2020, from https://towardsdatascience.com/entropy-how-decision-trees-make-decisions-2946b9c18c8.

[6] Eichenlaub, M. (2016, October 23). What's an intuitive way to understand entropy? Retrieved January 3, 2020, from https://www.quora.com/Whats-an-intuitive-way-to-understand-entropy.

[7] Software for complex networks. (n.d.). Retrieved January 3, 2020, from https://networkx.github.io/.

[8] Tree-plots in Python. (n.d.). Retrieved January 3, 2020, from https://plot.ly/python/tree-plots/.

Sources used for plotting the tables (source codes and documentation):

[9] How can I print the tables in a .pdf file using python. (2018, December 13). Retrieved January 3, 2020, from https://stackoverflow.com/questions/53755995/how-can-i-print-the-tables-in-a-pdf-file-using-python.

[10] How do I get multiple subplots in matplotlib? (2018, March 1). Retrieved January 3, 2020, from https://stackoverflow.com/questions/31726643/how-do-i-get-multiple-subplots-in-matplotlib.

[11] matplotlib.pyplot.subplots_adjust. (n.d.). Retrieved January 3, 2020, from https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.subplots_adjust.html#matplotlib.pyplot.subplots_adjust.

[12]_ Matplotlib - adding subplots to a subplot? (2016, January 21). Retrieved January 3, 2020, from https://stackoverflow.com/questions/34933905/matplotlib-adding-subplots-to-a-subplot.

[13] Matplotlib Row heights table property. (2015, January 15). Retrieved January 3, 2020, from https://stackoverflow.com/questions/27972524/matplotlib-row-heights-table-property.

Task 2: Model evaluation
[14] Saxena, S. (2018, May 11). Precision vs Recall. Retrieved January 3, 2020, from https://towardsdatascience.com/precision-vs-recall-386cf9f89488.

Task 3: SVMs
[15] Ray, S. (2017, September 13). Understanding Support Vector Machine algorithm from examples (along with code). Retrieved January 3, 2020, from https://www.analyticsvidhya.com/blog/2017/09/understaing-support-vector-machine-example-code/.

[16] Pupale, R. (2018, June 16). Support Vector Machines(SVM) — An Overview. Retrieved January 3, 2020, from https://towardsdatascience.com/https-medium-com-pupalerushikesh-svm-f4b42800e989.

[17] Misra, R. (2019, April 30). Support Vector Machines — Soft Margin Formulation and Kernel Trick. Retrieved January 3, 2020, from https://towardsdatascience.com/support-vector-machines-soft-margin-formulation-and-kernel-trick-4c9729dc8efe.

[18] Chauhan, V. K. (2016, January 1). Can anyone explain to me hard and soft margin Support Vector Machine (SVM)? Retrieved January 3, 2020, from https://www.researchgate.net/post/Can_anyone_explain_to_me_hard_and_soft_margin_Support_Vector_Machine_SVM.