1

Project: Decision Trees

University of Oregon, DSCI/CIS 372

This project will take a lot of time and work. You will not be able to complete it in a single sitting. Start early.

Project Description

In this project you will:

1. Implement a decision tree classifier from scratch.
    - You will use a provided dataset to train and test your decision tree model.
    - You will assess the accuracy of your technique using k-fold validation.
    - You will create visualizations using your model.
2. Create a random forest classifier.
    - You will choose a dataset, prepare the data for use, and train and test your random forest model on the data.
    - You will use a library-provided reporting tool to produce summary statistics about the efficacy of your model.
    - You will use the library-provided random forest classifier on a data set of your choice and, using the library-provided reporting tool for insight, you will write a brief report of your findings.

Task 0 | Set Up

This section is only for setting up your environment and does not have a deliverable.

- Your submitted code will be run in a terminal. You may use whatever development environment which works for you (e.g., PyCharm), but make sure your code runs from a regular terminal before submission.
- Make sure you are running Python 3. You can verify this by executing the command in the terminal:

  `python -version`

  Another approach is to always run the command `python3`.

  WARNING: If you use the command `python3`, do so consistently. You must also install libraries used by `python3` with the command `pip3`.

  DO NOT CONTINUE UNTIL YOU ARE RUNNING PYTHON 3
- Install the required libraries for this project by running the following commands:

  ```
  pip install scikit-learn
  pip install scipy
  pip install numpy
  pip install matplotlib
  ```

  Or, if you are running python via Anaconda:

  ```
  conda install scikit-learn
  conda install scipy
  conda install numpy
  conda install matplotlib
  ```

Task 1 | Implement a Decision Tree Classifier

An example of an iris in bloom.

An iris is a kind of flower, of which there are several species. You are given 150 rows of data. Each row represents information about a particular species of iris flower someone went and looked at.

The data is stored in a file called `iris_data.csv`. The first row is the header.

Recorded in the first column is the species of the flower. It is encoded here as a number: 1, 2, or 3.

Recorded in the second column is the average area of the flower's sepals in cm^2.

Recorded in the third column is the average area of the flower's petals in cm^2.

Our goal is, given sepal area and petal area, to predict if a flower's species is 1, 2, or 3.

1.1 Implement a decision tree classifier from scratch.

This project's starter code will follow a slightly different software design pattern than previous projects, to expose you to different ways machine learning architects might develop code in practice.

There are two starter code files for this sub task.

One is called `main01.py` and the other is called `my_decision_tree_classifier.py`

1.1.1 main01.py

`main01.py` is the file you should run in your terminal via `$python main01.py`

This file sets up most of the infrastructure needed to efficiently run your decision tree classifier.

`main01.py` has one unwritten method `load_data` you need to complete. The unwritten method takes the data loaded from `iris_data.csv` and randomly partitions each row into two sets. One set should contain 80% of the data and be used for training. The remaining 20% of the data is to be used for testing.

1.1.2 my_decision_tree_classifier.py

`my_decision_tree_classifier.py` contains a class called `MyDecisionTreeClassifier`. This class has two methods you need to implement:

- `fit(X, y)` is the method you will write to train the decision tree. Precise details on the inputs and outputs are located in the docstring.
- `predict(X)` is the method you will write to provide predictions for new data. You can take as a given that `predict(X)` will always be called after `fit(X, y)` has been invoked. Precise details on the inputs and outputs are located in the docstring.

You may use (and are encouraged to use) as many "helper" methods as you need. You should especially rely on the constructor.

Note. You have to implement these two methods from scratch. No decision tree library is allowed.

1.1.3 Run main01.py

Once you have implemented the three methods (one in main01.py, two in my_decision_tree_classifier.py), you can run `$python main01.py`.

You should save the output of main01.py by piping it to an output file called `main01.log` (e.g., `$python main01.py > main01.log`).

1.2 Assess the accuracy of your model using k-fold validation.

(See: `https://en.wikipedia.org/wiki/Cross-validation_(statistics)#k-fold_cross-validation`)

`main02.py` is similar to `main01.py`, in that it sets up the infrastructure for training and testing. It also sets up k fold validation.

`main02.py` has two unwritten methods you need to complete.

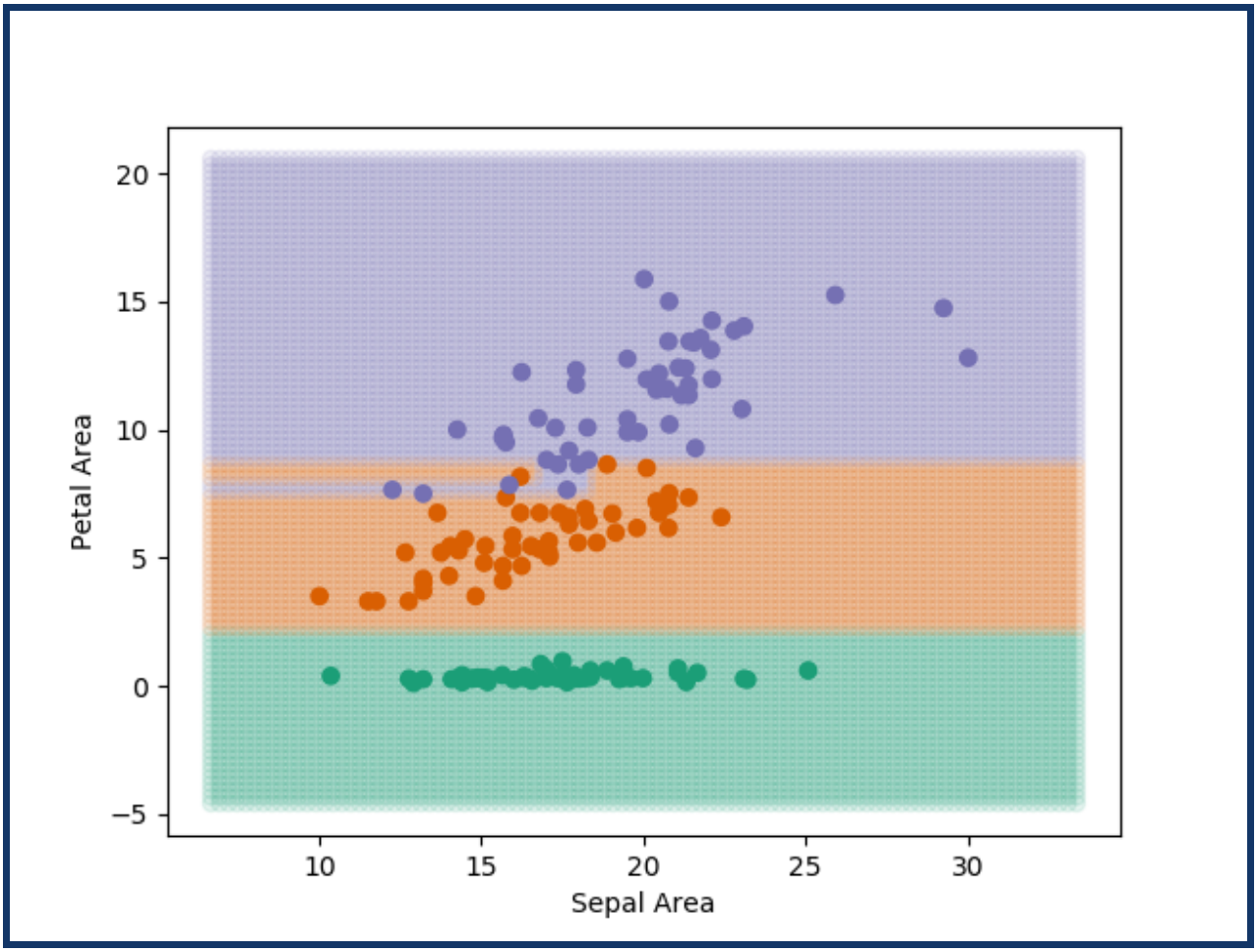
- `get_accuracy(y_true, y_pred)` returns a value in $[0, 1]$ reflecting the accuracy of a list of predictions when provided with their ground truth values. (0= all predictions were wrong, 1= all predictions were correct.) Precise details on the inputs and outputs are located in the docstring.
- `get_folds(k, X, y)` is the method you will write to partition data into k folds to be used for training and testing. Precise details on the inputs and outputs are located in the docstring.

Once you have implemented the three methods you can run `$python main02.py`.

You should save the output of main02.py by piping it to an output file called `main02.log` (e.g., `$python main02.py > main02.log`).

1.3 Create visualizations of your model.

This data has two factors we use to decide class. We can use these factors to plot in 2D which class our model predicts for a given pair of (sepal area, petal area).



Decision areas of a single decision tree. Training data is overlayed.

Your task is to replicate this image. In this image, a single decision tree model uses 100% of the data for training and then makes a prediction for $\sim$ 10,000 equally and rectangularly spaced points. Those points are then colored according to their assigned class and plotted.

Documentation for creating a scatter plot using matplotlib can be found here: `https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.scatter.html`

Create a single PDF called `decision_vis.pdf` with the following groups of images:

1. One image using all the data, and a single decision tree as the underlying model.
2. Five images, one for each fold of 5-fold cross validation using all the data, and a single decision tree as the underlying model.

Create new .py files to generate the imagery. It is best practice to keep visualization code separate from simulation/model code.

Write at least a paragraph in this PDF briefly analyzing the differences among these images. Can you come to an intuitive conclusion based on these images which model might be the "best"? Is it hard to tell? Explain your findings.

Task 2 | Implement a Random Forest Classifier

Task Overview

- You will use the a dataset of your choice to train and test a random forest model.
- The random forest model will use scikit-learn's built in ensemble bagger.
- You will use a library-provided reporting tool to describe summary statistics about the efficacy of your model. (scikit-learn's "classification report").
- You will use the library-provided random forest classifier on a data set of your choice and, using the library-provided reporting tool for insight, you will write a brief report of your findings.

2.1 Choose a dataset and do any preprocessing needed

Two repositories which house data sets are:

- Kaggle: `https://www.kaggle.com/datasets?tags=13302-Classification`
- UC Irvine's collection: `https://archive.ics.uci.edu/ml/datasets.php`

Find a classification dataset of your choosing from one of these two sites. You may also use another dataset of your choosing with the permission of the instructor.

Once you've decided on a dataset, you may need to do some preprocessing. Create an independent helper Python script to do this.

One pattern used for preprocessing is the "ETL" pattern (pronounce ee-tee-el). ETL stands for "Extract, Transform, Load."

- **Extract** Write code to open the data source (whether from a file, an online database, scraping a website, etc) and pull the raw data into the script.
- **Transform** Do the data wrangling in the code at this step. In a machine learning context, this can include the following:
  - Apply a mathematical function to some features. For example, for some positive value x, we might substitute it instead with $\ln(x)$.
  - Apply an imputation strategy. Imputation is how we "fill in" any missing values of a data set. Common imputation techniques include substituting the missing value(s) with the mode, median, or mean value of all the other entries of that feature.
  - Apply a "normalization." Normalization means different things to different people, but is usually some variant on one of the following:

    * Converting the values of a feature so that they all fall between two numbers (often 0 and 1).
    * Converting the values of a feature so that the mean of the feature is 0 and the standard deviation of the values of the feature is 1. (i.e., pretend that the distribution for an individual feature is a "standard normal distribution," even when it's not, since having values in that range is helpful for many ML techniques)

  Scikit-learn's Scalars, such as `StandardScalar` (found in its "preprocessing" section) are useful starting points.

- **Load** Take your wrangled data and load it into a usable format for the rest of your application.

  One way to do this is to simply write a new csv file called `cleaned_data.csv`, which is then called by your machine learning code in much the same way as the `iris_data.csv` was loaded.

  While you can certainly load the data directly into your machine learning apparatus, writing to a separate CSV file and then re-reading the cleaned data later confers some advantages:

  - It allows you to manually verify that the cleaned data is what you expect it to be (especially when opened with a third party tool like Excel or Google Sheets).
  - It can allow for less overhead. Once the data has been cleaned, you never have to run the preprocessing step again. This may be a huge time saver for large datasets.
  - It helps enforce "low cohesion, high coupling," which is a hallmark of most well written software.

2.2 Write a Random Forest class

`my_random_forest_classifier.py` contains a class called `MyRandomForestClassifier`. This class has two methods you need to implement:

- `fit(X, y)` is the method you will write to train the decision tree. Precise details on the inputs and outputs are located in the docstring.
- `predict(X)` is the method you will write to provide predictions for new data. You can take as a given that `predict(X)` will always be called after `fit(X, y)` has been invoked. Precise details on the inputs and outputs are located in the docstring.

You should use the BaggingClassifier class from scikit learn to implement your random forest. See: `https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.BaggingClassifier.html`

2.3 Run and assess your Random Forest model

Create a new file called `main03.py`. Write code in `main03.py` which

1. Loads your cleaned data.
2. Partitions the loaded data into two sets: training data (a randomly selected subset of your cleaned data) and testing data (the data not selected for training). You may choose the exact train/test ratio, but it should be fairly high. E.g., 80% training, 20% testing; 90% training, 10% testing.
3. Instantiates your MyRandomForestClassifier.
   - Trains your model on the training data.
   - Predicts on the testing data. Save the predictions to a variable, such as `my_random_forest_predictions`.

2.4 Report your findings

Write code in `main03.py` which runs scikit-learn's Classification Report (see: `https://scikit-learn.org/stable/modules/model_evaluation.html#classification-report`). One classification report should be done on your random forest's predictions.

Create a PDF called `report.pdf` which contains responses to the following:

1. The classification report for your random forest's predictions.
2. A brief (1 to 2 paragraph) explanation of your findings.

   This explanation should remark about how (un)successful the models were and potential reasons why.
3. Meta-question: We want to get feedback on your experience with this project. Please answer the following (they don't have to be long answers – a sentence can suffice).
   (a) How long did this project take? 2 minutes? 2 hours? 20 hours? 200 hours?
   (b) What's one thing about this project which should be changed for future editions of the class?
   (c) What's one thing about this project which should be kept for future editions of the class?
   (d) Can we use your (anonymized) submission as an example for students in the future?

Task 4 | What to Submit

You will submit two files:

`runnable_code.zip` and `reports_and_logs.zip`

4.1 Code

Put any executable code into a zip file called `runnable_code.zip`. The goal is that another human can unzip this folder and successfully run your code. You may put any special instructions or notes about how to run your code in a `readme.txt` file, also placed within `runnable_code.zip`.

This zip folder should contain at least the following:

- `main01.py`, `main02.py`, `main03.py`, and all of their dependencies and helper scripts.
- Code you used to generate visualizations.
- The dependencies `my_decision_tree_classifier.py` and `my_random_forest_classifier.py`, and any preprocessing script you wrote should also be included.
- The data you used to train. (Both raw and cleaned data, if possible.) If the data set is so large that you can't upload it to Canvas (even after ZIP-ing), then put a note in a `readme.txt` file indicating where a human could download the data.

4.2 Reports and Logs

Put any reports, logs, images, and output into a zip file called `reports_and_logs.zip`. This zip folder should contain at least the following:

- PDFs, including: `decision_vis.pdf` and `report.pdf`
- Logs, including: `main01.log` and `main02.log`

4.3 Upload to Canvas

Upload two separate files to Canvas: `runnable_code.zip` and `reports_and_logs.zip`.

Appendix A | Grading Criteria

This project will be primarily assessed along three axes:

- **Validity**

  Programs should conform to specifications stated in the project description.

  Some components of the project will have example output provided. Your program's output must match the provided output for full points. If your output does not match identically for deterministic code (or within a reasonable margin for non-deterministic/random-based code) then expect a heavy penalty.

- **Style, Documentation, and Readability**

  Programs should be easy to read and understand. Coding well is writing well. Writing well is thinking well.

  To help us write quality code, we will use Pylint. Pylint is used in industry (Google, Amazon, etc.) to enforce Python's coding standard. Pylint also provides messages about style / readability / documentation flaws. You should run Pylint before submitting your work. Pylint assigns a score out of 10 to the code it processes. It can be very, very difficult to get a 10. We will consider 6/10 to be "full credit" for most work.

- **Design and Fluency**

  Coding is a highly creative and highly controlled processes. In making decisions in how to tackle the objectives outlined in the project tasks, you should keep in mind the following principles:

  - Programs should be written with regard to efficiency of both time and space.
  - Programs should be "elegantly" written rather than "brute forced."
  - Programs should consist of small, coherent, independent methods.
    I follow the two inches rule-of-thumb: if a method is longer than two inches on the screen, it should probably be broken up into more methods.

  Design and fluency is a category which is difficult to automatically grade, yet incredibly important to get feedback on. Come and talk in office hours if you are routinely missing points in this category.

Rubric

Implement a decision tree from scratch (100 points possible)

my_decision_tree_classifier.py validity: _____ of 20

my_decision_tree_classifier.py readability: _____ of 6

my_decision_tree_classifier.py fluency: _____ of 4

main01.py validity: _____ of 10

main01.py readability: _____ of 6

main01.py fluency: _____ of 4

main02.py validity: _____ of 10

main02.py readability: _____ of 6

main02.py fluency: _____ of 4

Visualization code: _____ of 15

Visualization code: _____ of 6

Visualization code: _____ of 4

decision_vis.pdf: _____ of 5

Subtotal: _____ of 100

Implement a random forest classifier (50 points possible)

Preprocessing code validity: _____ of 20

Preprocessing code readability: _____ of 6

Preprocessing code fluency: _____ of 4

my_random_forest_classifier.py validity: _____ of 3

my_random_forest_classifier.py readability: _____ of 1

my_random_forest_classifier.py fluency: _____ of 1

main03.py validity: _____ of 7

main03.py readability: _____ of 2

main03.py fluency: _____ of 1

report.pdf: _____ of 5

Subtotal: _____ of 50

Total: _____ of 150

Additional comments from the grader (if any):

_____

_____

_____