CSE 6242 / CX 4242: Data and Visual Analytics | Georgia Tech | Spring 2025

HW 4: PageRank Algorithm, Random Forest, Scikit-learn

Download the HW4 Skeleton before you begin

Homework Overview

Data analytics and machine learning both revolve around using computational models to capture relationships between variables and outcomes. In this assignment, you will code and fit a range of well-known models from scratch and learn to use a popular Python library for machine learning.

In Q1, you will implement the famous PageRank algorithm from scratch. PageRank can be thought of as a model for a system in which a person is surfing the web by choosing uniformly at random a link to click on at each successive webpage they visit. Assuming this is how we surf the web, what is the probability that we are on a particular webpage at any given moment? The PageRank algorithm assigns values to each webpage according to this probability distribution.

In Q2, you will implement Random Forests, a very common and widely successful classification model, from scratch. Random Forest classifiers also describe probability distributions—the conditional probability of a sample belonging to a particular class given some or all its features.

Finally, in Q3, you will use the Python scikit-learn library to specify and fit a variety of supervised and unsupervised machine learning models.

The maximum possible score for this homework is **100 points.**

Important Notes

1. Submit your work by the due date on the course schedule.
    a. Every assignment has a generous 48-hour grace period, allowing students to address unexpected minor issues without facing penalties. You may use it without asking.
    b. Before the grace period expires, you may resubmit as many times as needed.
    c. TA assistance is not guaranteed during the grace period.
    d. Submissions during the grace period will display as "late" but **will not** incur a penalty.
    e. **We will not accept any submissions executed after the grace period ends**.
2. Always use the **most up-to-date assignment** (version number at bottom right of this document). The latest version will be listed in Ed Discussion.
3. You may discuss ideas with other students at the "whiteboard" level (e.g., how cross-validation works, use HashMap instead of an array) and review any relevant materials online. However, **each student must write up and submit the student's own answers**.
4. All incidents of suspected dishonesty, plagiarism, or violations of the Georgia Tech Honor Code will be subject to the institute's Academic Integrity procedures, directly handled by the Office of Student Integrity (OSI). Consequences can be severe, e.g., academic probation or dismissal, a 0 grade for assignments concerned, and prohibition from withdrawing from the class.

Submission Notes

1. All questions are graded on the Gradescope platform, accessible through Canvas.
2. We will not accept submissions anywhere else outside of Gradescope.
3. Submit all required files as specified in each question. Make sure they are named correctly.
4. You may upload your code periodically to Gradescope to obtain feedback on your code. **There are no hidden test cases.** The score you see on Gradescope is what you will receive.
5. You must **not** use Gradescope as the primary way to test your code. It provides only a few test cases and error messages may not be as informative as local debuggers. Iteratively develop and test your code locally, write more test cases, and follow good coding practices. Use Gradescope mainly as a "final" check.
6. **Gradescope cannot run code that contains syntax errors**. If you get the "The autograder failed to execute correctly" error, verify:
    a. The code is free of syntax errors (by running locally)
    b. All methods have been implemented
    c. The correct file was submitted with the correct name
    d. No extra packages or files were imported
7. When many students use Gradescope simultaneously, it may slow down or fail. It can become even slower as the deadline approaches. You are responsible for submitting your work on time.
8. Each submission and its score will be recorded and saved by Gradescope. **By default, your last submission is used for grading.** To use a different submission, **you MUST "activate" it** (click the "Submission History" button at the bottom toolbar, then "Activate").

Q1 [20 pts] Implementation of PageRank Algorithm

Technology	PageRank Algorithm Graph Python >=3.7.x. **You must use Python >=3.7.x for this question.**
Allowed Libraries	Do not modify the import statements; everything you need to complete this question has been imported for you. You MUST not use other libraries for this assignment.
Max runtime	5 minutes

Deliverables	[Gradescope]
	• **Q1.ipynb [12 pts]**: your modified implementation
	• **simplified_pagerank_iter{n}.txt**: 2 files (as given below) containing the top 10 node IDs (w.r.t. the PageRank values) and their PageRank values for n iterations via the provided run() helper function
	  ○ **simplified_pagerank_iter10.txt [2 pts]**
	  ○ **simplified_pagerank_iter25.txt [2 pts]**
	• **personalized_pagerank_iter{n}.txt:** 2 files (as given below) containing the top 10 node IDs (w.r.t the PageRank values) and their PageRank values for n iterations via the provided run() helper function
	  ○ **personalized_pagerank_iter10.txt [2 pts]**
	  ○ **personalized_pagerank_iter25.txt [2 pts]**

Important: Remove all "testing" code that renders output, or Gradescope will crash. For instance, any additional print, display, and show statements used for debugging must be removed.

In this question, you will implement the PageRank algorithm in Python for a large graph network dataset.

The PageRank algorithm was first proposed to rank web pages in search results. The basic assumption is that more "important" web pages are referenced more often by other pages and thus are ranked higher. To estimate the importance of a page, the algorithm works by considering the number and "importance" of links pointing to the page. PageRank outputs a probability distribution over all web pages, representing the likelihood that a person randomly surfing the web (randomly clicking on links) would arrive at those pages.

As mentioned in the lectures, the PageRank values are the entries in the dominant eigenvector of the modified adjacency matrix in which each column's values adds up to 1 (i.e., "column normalized"), and this eigenvector can be calculated by the power iteration method that you will implement in this question. This method iterates through the graph's edges multiple times to update the nodes' PageRank values ("pr_values" in Q1.ipynb) in each iteration. We recommend that you review the lecture video for PageRank and personalized PageRank before working on your implementation. At 9 minutes and 41 seconds of the video, the full PageRank algorithm is expressed in a matrix-vector form. Equivalently, the PageRank value of node v_j, at iteration $t+1$, can also be expressed as (notation different from video's):

$$PR_{t+1}(v_j) = (1-d) \times Pd(v_j) + d \times \sum_{v_i} \frac{PR_t(v_i)}{out\ degree(v_i)}$$

where

- v_j is node j

- v_i is any node i that has a directed edge pointing to node j
- $out\ degree(v_i)$ is the number of links going out of node v_i
- $PR_{t+1}(v_j)$ is the pagerank value of node j at iteration $t+1$
- $PR_t(v_i)$ is the pagerank value of node i at iteration t
- d is the damping factor; set it to the common value of 0.85 that the surfer would continue to follow links
- $Pd(v_j)$ is the probability of random jump that can be personalized based on use cases

Tasks

You will be using the "network.tsv" graph network dataset in the hw4-skeleton/Q1 folder, which contains about 1 million nodes and 3 million edges. Each row in that file represents a directed edge in the graph. The edge's source node id is stored in the first column of the file, and the target node id is stored in the second column.

Your code must **NOT** make any assumptions about the relative magnitude between the node ids of an edge. For example, suppose we find that the source node id is smaller than the target node id for most edges in a graph, we must **NOT** assume that this is always the case for all graphs (i.e., in other graphs, a source node id can be larger than a target node id).

You will complete the code in `Q1.ipynb` (guidelines also provided in the file).

1. Calculate and store each node's out-degree and the graph's maximum node id in `calculate_node_degree()`
    a. A node's out-degree is its number of outgoing edges. Store the out-degree in instance variable "node_degree".
    b. `max_node_id` refers to the highest node id in the graph. For example, suppose a graph contains the two edges (1,4) and (2,3), in the format of (source, target), the `max_node_id` here is 4. Store the maximum node id to instance variable `max_node_id`.
2. Implement `run_pagerank()`
    a. For simplified PageRank algorithm, where $Pd(v_j)$ = `1/(max_node_id + 1)` is provided as `node_weights` in the script and you will submit the output for 10 and 25 iteration runs for a damping factor of 0.85. To verify, we are providing the sample output of 5 iterations for a simplified PageRank (simplified_pagerank_iter5_sample.txt**)**.
    b. For personalized PageRank, the $Pd()$ vector will be assigned values based on your 9-digit GTID (e.g., 987654321) and you will submit the output for 10 and 25 iteration runs for a damping factor of 0.85.
3. Compare output
    a. Generate output text files by running the last cell of `Q1.ipynb`.
    b. **Note:** When comparing your output for simplified_pagerank for 5 iterations with the given sample output, the absolute difference must be less than 5%. For example, `absolute((SampleOutput - YourOutput) / SampleOutput)` must be less than `0.05`.

Q2 [50 pts] Random Forest Classifier

Technology	Python >=3.7.x
Allowed Libraries	Do not modify the import statements; everything you need to complete this question has been imported for you. You MUST not use other libraries for this assignment.
Max runtime	300 seconds

Deliverables	[Gradescope]
	• **Q2.ipynb [45 pts]**: your solution as a Jupyter notebook, developed by completing the provided skeleton code
	  ○ 10 points are awarded for 2 utility functions, 5 points for `entropy()` and 5 points for `information_gain()`
	  ○ 35 points are awarded for successfully implementing your random forest
	• **Random Forest Reflection [5 pts]**: multiple-choice question completed on Gradescope.

Q2.1 - Random Forest Setup [45 pts]

Note: You must use Python >=3.7.x for this question.

You will implement a random forest classifier in Python via a Jupyter notebook. The performance of the classifier will be evaluated via the out-of-bag (OOB) error estimate using the provided dataset `Wisconsin_breast_prognostic.csv`, a comma-separated (csv) file in the Q2 folder. Features (Attributes) were computed from a digitized image of a fine needle aspirate (FNA) of a breast mass. They describe characteristics of the cell nuclei present in the image. **You must not modify the dataset.** Each row describes one patient (a data point, or data record) and each row includes 31 columns. The first 30 columns are attributes. The 31st (the last column) is the label, and you must **NOT** treat it as an attribute. The value one and zero in the last column indicates whether the cancer is malignant or benign, respectively. You will perform binary classification on the dataset to determine if a particular cancer is benign or malignant.

Important:

1. Remove all "testing" code that renders output, or Gradescope will crash. For instance, any additional print, display, and show statements used for debugging must be removed.

2. You may only use the modules and libraries provided at the top of the notebook file included in the skeleton for Q2 and modules from the Python Standard Library. Python wrappers (or modules) must **NOT** be used for this assignment. Pandas must **NOT** be used — while we understand that they are useful libraries to learn, completing this question is not critically dependent on their functionality. In addition, to make grading more manageable and to enable our TAs to provide better, more consistent support to our students, we have decided to restrict the libraries accordingly.

Essential Reading

Decision Trees. To complete this question, you will develop a good understanding of how decision trees work. We recommend that you review the lecture on the decision tree. Specifically, review how to construct decision trees using *Entropy* and *Information Gain* to select the splitting attribute and split point for the selected attribute. These slides from CMU (also mentioned in the

lecture) provide an excellent example of how to construct a decision tree using *Entropy* and *Information Gain*. **Note:** there is a typo on page 10, containing the Entropy equation; ignore one negative sign (only one negative sign is needed).

Random Forests. To refresh your memory about random forests, see Chapter 15 in the Elements of Statistical Learning book and the lecture on random forests. Here is a blog post that introduces random forests in a fun way, in layman's terms.

Out-of-Bag Error Estimate. In random forests, it is not necessary to perform explicit cross-validation or use a separate test set for performance evaluation. Out-of-bag (OOB) error estimate has shown to be reasonably accurate and unbiased. Below, we summarize the key points about OOB in the original article by Breiman and Cutler.

Each tree in the forest is constructed using a different bootstrap sample from the original data. Each bootstrap sample is constructed by randomly sampling from the original dataset **with replacement** (usually, a bootstrap sample has the same size as the original dataset). Statistically, about one-third of the data records (or data points) are left out of the bootstrap sample and not used in the construction of the *kth* tree. For each data record that is not used in the construction of the *kth* tree, it can be classified by the *kth* tree. As a result, each record will have a "test set" classification by the subset of trees that treat the record as an out-of-bag sample. The majority vote for that record will be its predicted class. The proportion of times that a record's predicted class is different from the true class, averaged over all such records, is the OOB error estimate.

While splitting a tree node, make sure to randomly select a subset of attributes (e.g., square root of the number of attributes) and pick the best splitting attribute (and splitting point of that attribute) among these subsets of attributes. This randomization is the main difference between random forest and bagging decision trees.

Starter Code

We have prepared some Python starter code to help you load the data and evaluate your model. The starter file name is Q2.ipynb has three classes:

- `Utililty`: contains utility functions that help you build a decision tree
- `DecisionTree`: a decision tree class that you will use to build your random forest
- `RandomForest`: a random forest class

What you will implement

Below, we have summarized what you will implement to solve this question. Note that you must use **information gain** to perform the splitting in the decision tree. The starter code has detailed comments on how to implement each function.

1. `Utililty class`: implement the functions to compute entropy, information gain, perform splitting, and find the best variable (attribute) and split-point. You can add additional methods for convenience. Note: Do not round the output or any of your functions.
2. `DecisionTree class`: implement the `learn()` method to build your decision tree using the utility functions above.
3. `DecisionTree class`: implement the `classify()` method to predict the label of a test record using your decision tree.
4. `RandomForest class`: implement the methods `bootstrapping()`, `fitting()`, `voting()` and `user()`.
5. `get_random_seed()`, `get_forest_size()`: implement the functions to return a random seed and forest size (number of decision trees) for your implementation.

Important:

1. You **must** achieve a minimum accuracy of **90%** for the random forest. If the accuracy is turning out to be low, try playing around with hyper-parameters. If it is extremely low, try revisiting `best_split()` and `classify()` methods.
2. Your code must take **no more than 5 minutes** to execute (which is a very long time, given the low program complexity). Otherwise, it may time out on Gradescope. Code that takes longer than 5 minutes to run likely means you need to correct inefficiencies (or incorrect logic) in your program. We suggest that you check the hyperparameter choices (e.g., tree depth, number of trees) and code logic when figuring out how to reduce the runtime.
3. The `run()` function is provided to test your random forest implementation; do **NOT** modify this function.
4. **Note:** In your implementation, use basic Python Lists rather than the more complex Numpy data structures to reduce the chances of version-specific library conflicts with the grading scripts.

As you solve this question, consider the following design choices. Some may be more straightforward to determine, while some maybe not (hint: study lecture materials and essential reading above). For example:

- Which attributes to use when building a tree?
- How to determine the split point for an attribute?
- How many trees should the forest contain?
- You may implement your decision tree using the data structure of your choice (e.g., dictionary, list, class member variables). However, your implementation must still work within the `DecisionTree` class structure we have provided.
- Your decision tree will be initialized using `DecisionTree(max_depth=10)`, in the `RandomForest` class in the jupyter notebook.
- When do you stop splitting leaf nodes?
- The depth found in the learn function is the depth of the current node/tree. You may want a check within the learn function that looks at the current depth and returns if the depth is greater than or equal to the max depth specified. Otherwise, it is possible that you continually split on nodes and create a messy tree. The `max_depth` parameter should be used as a stopping condition for when your tree should stop growing. Your decision tree will be instantiated with a depth of 0 (input to the `learn()` function in the jupyter notebook). To comply with this, make sure you implement the decision tree such that the root node starts at a depth of 0 and is built with increasing depth.

Note that, as mentioned in the lecture, there are other approaches to implement random forests. For example, instead of information gain, other popular choices include the Gini index, random attribute selection (e.g., PERT - Perfect Random Tree Ensembles). We decided to ask everyone to use an **information gain** based approach in this question (instead of leaving it open-ended), because information gain is a useful machine learning concept to learn in general.

Q2.2 - Random Forest Reflection [5 pts]

On Gradescope, answer the following multiple-choice question. You can submit your answer **only once. Clicking the "Save Answer" button on Gradescope WILL immediately submit your answer, making it final and unchangeable. Select all that apply; your answer must be completely correct to earn the points. No partial marks will be awarded if all correct options are NOT selected**.

What are the main advantages of using a random forest versus a single decision tree?

Q3 [30 points] Using Scikit-Learn

Technology	Python >=3.7.x Scikit-Learn >=0.22
Allowed Libraries	Do not modify the import statements; everything you need to complete this question has been imported for you. You MUST not use other libraries for this assignment.
Max runtime	15 minutes
Deliverables	[Gradescope] **Q3.ipynb [30 pts]**: your solution as a Jupyter notebook, developed by completing the provided skeleton code

Scikit-learn is a popular Python library for machine learning. You will use it to train some classifiers to predict diabetes in the Pima Indian tribe. The dataset is provided in the Q3 folder as pima-indians-diabetes.csv.
For this problem, you will be utilizing a Jupyter notebook.

Important:

1. Remove all "testing" code that renders output, or Gradescope will crash. For instance, any additional print, display, and show statements used for debugging must be removed.
2. Use the default values while calling functions unless specific values are given.
3. Do not round off the results except the results obtained for Linear Regression Classifier.
4. Do not change the '#export' statements or add any other code/comments above them. They are needed for grading.

Q3.1 - Data Import [2 pts]

In this step, you will import the pima-indians-diabetes dataset and allocate the data to two separate arrays. After importing the data set, you will split the data into a training and test set using the scikit-learn function train_test_split. You will use scikit-learns built-in machine learning algorithms to predict the accuracy of training and test set separately. Refer to the hyperlinks provided below for each algorithm for more details, such as the concepts behind these classifiers and how to implement them.

Q3.2 - Linear Regression Classifier [4 pts]

Q3.2.1 - Classification
Train the Linear Regression classifier on the dataset. You will provide the accuracy for both the test and train sets. Make sure that you round your predictions to a binary value of 0 or 1. Do not use np.round function as it can produce results that surprise you and not meet your needs (see the official numpy documentation for details). Instead, we recommend you write a custom round function using if-else. See the Jupyter notebook for more information. Linear regression is most commonly used to solve regression problems. The exercise here demonstrates the possibility of using linear regression for classification (even though it may not be the optimal model choice).

Q3.3 - Random Forest Classifier [10 pts]

Q3.3.1 - Classification
Train the Random Forest classifier on the dataset. You will provide the accuracy for both the test and train sets. **Do not round your prediction.**

Q3.3.2- Feature Importance

You have performed a simple classification task using the random forest algorithm. You have also implemented the algorithm in Q2 above. The concept of entropy gain can also be used to evaluate the importance of a feature. You will determine the feature importance evaluated by the random forest classifier in this section. Sort the features in descending order of feature importance score, and print the sorted features' numbers.

Hint: There is a function available in sklearn to achieve this. Also, take a look at `argsort()` function in Python numpy. `argsort()` returns the indices of the elements in ascending order. You will use the random forest classifier that you trained initially in Q3.3.1, without any kind of hyperparameter-tuning, for reporting these features.

Q3.3.3 - Hyper-Parameter Tuning
Tune your random forest hyper-parameters to obtain the highest accuracy possible on the dataset. Finally, train the model on the dataset using the tuned hyper-parameters. Tune the hyperparameters specified below, using the GridSearchCV function in Scikit library:

```
'n_estimators': [4, 16, 256], 'max_depth': [2, 8, 16]
```

Q3.4 - Support Vector Machine [10 pts]

Q3.4.1 - Preprocessing
For SVM, we will standardize attributes (features) in the dataset using StandardScaler, before training the model.

Note: for StandardScaler,

- Transform both x_train and x_test to obtain the standardized versions of both.
- Review the StandardScaler documentation, which provides details about standardization and how to implement it.

Q3.4.2 - Classification
Train the Support Vector Machine classifier on the dataset (the link points to SVC, a particular implementation of SVM by Scikit). You will provide the accuracy on both the test and train sets.

Q3.4.3. - Hyper-Parameter Tuning
Tune your SVM model to obtain the highest accuracy possible on the dataset. For SVM, tune the model on the standardized train dataset and evaluate the tuned model with the test dataset. Tune the hyperparameters specified below in the same order, using the GridSearchCV function in Scikit library:

```
'kernel':('linear', 'rbf'), 'C':[0.01, 0.1, 1.0]
```

Note: If GridSearchCV takes a long time to run for SVM, make sure you standardize your data beforehand using StandardScaler.

Q3.4.4. - Cross-Validation Results
Let's practice obtaining the results of cross-validation for the SVM model. Report the rank test score and mean testing score for the best combination of hyper-parameter values that you obtained. The GridSearchCV class holds a `cv_results_` dictionary that helps you report these metrics easily.

Q3.5 - Principal Component Analysis [4 pts]

Performing Principal Component Analysis based dimensionality reduction is a common task in many data analysis tasks, and it involves projecting the data to a lower-dimensional space using Singular Value Decomposition. Refer to the examples given here; set parameters `n_component` to `8` and `svd_solver` to `full`. See the sample outputs below.

1. Percentage of variance explained by each of the selected components. Sample Output:

```
[6.51153033e-01 5.21914311e-02 2.11562330e-02 5.15967655e-03
6.23717966e-03 4.43578490e-04 9.77570944e-05 7.87968645e-06]
```

2. The singular values corresponding to each of the selected components. Sample Output:

```
[5673.123456  4532.123456   4321.68022725  1500.47665361
   1250.123456   750.123456    100.123456    30.123456]
```

Use the Jupyter notebook skeleton file called Q3.ipynb to write and execute your code.

As a reminder, the general flow of your machine learning code will look like:

1. Load dataset
2. Preprocess (you will do this in Q3.2)
3. Split the data into `x_train`, `y_train`, `x_test`, `y_test`
4. Train the classifier on `x_train` and `y_train`
5. Predict on `x_test`
6. Evaluate testing accuracy by comparing the predictions from step 5 with `y_test`.

Here is an example. Scikit has many other examples as well that you can learn from.