CSE 6242/CX 4242: Data and Visual Analytics | Georgia Tech | Fall 2017

Homework 4 : Scalable PageRank via Virtual Memory (MMap), Random Forest, Weka

Due: Friday, December 1, 2017, 11:55 PM EST

Prepared by Kiran Sudhir, Varun Bezzam, Yuyu Zhang, Akanksha Bindal,
Vishal Bhatnagar, Vivek Iyer, Polo Chau

Submission Instructions and Important Notes:

It is important that you read the following instructions carefully and also those about the deliverables at the end of each question or you may lose points.

- ❏ Always check to make sure you are using the most up-to-date assignment PDF (e.g., re-download it from the course homepage if unsure).
- ❏ Submit a single zipped file, called "HW4-{YOUR_LAST_NAME}-{YOUR_FIRST_NAME}.zip", containing all the deliverables including source code/scripts, data files, and readme. Example: 'HW4-Doe-John.zip' if your name is John Doe. **Only .zip is allowed** (no other format will be accepted)
- ❏ You may collaborate with other students on this assignment, but you must write your own code and give the explanations in your own words, and also mention the collaborators' names on T-Square's submission page. All GT students must observe the honor code. **Suspected plagiarism and academic misconduct will be reported to and directly handled** by the Office of Student Integrity (OSI). Here are some examples similar to Prof. Jacob Eisenstein's NLP course page (grading policy):
  - ❏ **OK:** discuss concepts and strategies (e.g., how cross-validation works, use hashmap instead of array)
  - ❏ **Not OK:** several students work on one master copy together (e.g., by dividing it up), sharing solutions, or using solution from previous years or from the web.
- ❏ If you use any "*slip days*", you must write down the number of days used in the T-square submission page. For example, "Slip days used: 1". Each slip day equals 24 hours. E.g., if a submission is late for 30 hours, that counts as 2 slip days.
- ❏ At the end of this assignment, we have specified a folder structure about how to organize your files in a single zipped file. 5 points will be deducted for not following this strictly.
- ❏ We will use auto-grading scripts to grade some of your deliverables (there are hundreds of students), so it is extremely important that you strictly follow our requirements. Marks may be deducted if our grading scripts cannot execute on your deliverables.
- ❏ Wherever you are asked to write down an explanation for the task you perform, stay within the word limit or you may lose points.
- ❏ In your final zip file, please do not include any intermediate files you may have generated to work on the task, unless your script is absolutely dependent on it to get the final result (which it ideally should not be).
- ❏ After all slip days are used, 5% deduction for every 24 hours of delay. (e.g., 5 pts for 100-point homework)
- ❏ We will not consider late submission of any missing parts of a homework assignment or project deliverable. To make sure you have submitted everything, download your submitted files to double check.

Download the HW4 Skeleton before you begin.

Q1 [30 pts] Scalable single-PC PageRank on 70M edge graph

In this question, you will learn how to use your computer's **virtual memory** to implement the PageRank algorithm that will scale to graph datasets with as many as billions of edges using a single computer (e.g., your laptop). As discussed in class, a standard way to work with larger datasets has been to use computer clusters (e.g., Spark, Hadoop) which may involve steep learning curves, may be costly (e.g., pay for hardware and personnel), and importantly may be "overkill" for smaller datasets (e.g., a few tens or hundreds of GBs). The virtual memory based approach offers an attractive, simple solution to allow practitioners and researchers to more easily work with such data (visit the NSF-funded MMap project's homepage to learn more about the research).

The main idea is to place the dataset in your computer's (unlimited) virtual memory, as it is often too big to fit in the RAM. When running algorithms on the dataset (e.g., PageRank), the operating system will automatically decide when to load the necessary data (subset of whole dataset) into RAM.

This technical approach to put data into your machine's virtual memory space is called "memory mapping", which allows the dataset to be treated as if it is an in-memory dataset. In your (PageRank) program, you do not need to know whether the data that you need is stored on the hard disk, or kept in RAM. Note that memory-mapping a file does NOT cause the whole file to be read into memory. Instead, data is loaded and kept in memory only when needed (determined by strategies like *least recently used* paging and *anticipatory* paging).

You will use the Python modules `mmap` and `struct` to map a large graph dataset into your computer's virtual memory. The `mmap()` function does the "memory mapping", establishing a mapping between a program's (virtual) memory address space and a file stored on your hard drive -- we call this file a "memory-mapped" file. Since memory-mapped files are viewed as a sequence of bytes (i.e., a binary file), your program needs to know how to convert bytes to and from numbers (e.g., integers). `struct` supports such conversions via "packing" and "unpacking", using format specifiers that represent the desired endianness and data type to convert to/from.

Q1.1 Set up Pypy

Install PyPy, which is a Just-In-Time compilation runtime for python, which supports fast packing and unpacking. (As mentioned in class, C++ and Java are generally faster than Python. However, several projects aim to boost Python speed. PyPy is one of them.)

Ubuntu	`sudo apt-get install pypy`

MacOS	Install Homebrew Run `brew install pypy`
Windows	Download the package and then install it.

Run the following code in the Q1 directory to learn more about the helper utility that we have provided to you for this question.

```
$ pypy q1_utils.py --help
```

Q1.2 Warm Up (10 pts)

Get started with memory mapping concepts using the code-based tutorial in `warmup.py`.
You should study the code and modify parts of it as instructed in the file. You can run the tutorial code as-is (without any modifications) to test how it works (run "`python warmup.py`" on the terminal to do this). The warmup code is setup to pack the integers from 0 to 63 into a binary file, and unpack it back into a memory map object. You will need to modify this code to do the same thing for all odd integers in the range of 1 to 42. The lines that need to be updated are clearly marked. **Note:** You must not modify any other parts of the code. When you are done, you can run the following command to test whether it works as expected:

```
$ python q1_utils.py test_warmup out_warmup.bin
```

It prints `True` if the binary file created after running `warmup.py` contains the expected output.

Q1.3 Implementing and running PageRank (20 pts)

You will implement the PageRank algorithm, using the power iteration method, and run it on the LiveJournal dataset (an online community with millions of users to maintain journals and blogs). You may want to revisit the MMap lecture slides (slide 9, 10) to refresh your memory about the PageRank algorithm and the data structures and files that you may need to memory-map. (For more details, read the MMap paper.) You will perform three steps (subtasks) as described below.

Step 1: Download the LiveJournal graph dataset (an edge list file)

The LiveJournal graph contains almost 70 million edges. It is available on the SNAP website. We are hosting the graph on our course homepage, to avoid high traffic bombarding their site.

Step 2: Convert the graph's edge list to binary files (you only need to do this once)

Since memory mapping works with binary files, you will convert the graph's edge list into its binary format by running the following command at the terminal/command prompt:

```
$ python q1_utils.py convert <path-to-edgelist.txt>
```

Example: Consider the following `toy-graph.txt`, which contains 7 edges:

```
0 1
1 0
1 2
2 1
3 4
4 5
5 2
```

To convert the graph to its binary format, you will type:

```
$ python q1_utils.py convert toy-graph/toy-graph.txt
```

This generates 3 files:

```
toy-graph/
```

`toy-graph.bin:`	binary file containing edges (source, target) in little-endian "int" C type
`toy-graph.idx:`	binary file containing (node, degree) in little-endian "long long" C type
`toy-graph.json:`	metadata about the conversion process (required to run pagerank)

In `toy-graph.bin` we have,

```
0000 0000 0100 0000   # 0 1   (in little-endian "int" C type)
0100 0000 0000 0000   # 1 0
0100 0000 0200 0000   # 1 2
0200 0000 0100 0000   # 2 1
0300 0000 0400 0000   # 3 4
0400 0000 0500 0000   # 4 5
0500 0000 0200 0000   # 5 2
ffff ffff ffff ffff
...
ffff ffff ffff ffff
ffff ffff ffff ffff
```

In `toy-graph.idx` we have,

```
0000 0000 0000 0000 0100 0000 0000 0000   # 0 1  (in little-endian “long long” C type )
0100 0000 0000 0000 0200 0000 0000 0000   # 1 2
...
ffff ffff ffff ffff ffff ffff ffff ffff
```

Note: there are extra values of -1 (`ffff ffff or ffff ffff ffff ffff`) added at the end of the binary file as padding to ensure that the code will not break in case you try to read a value greater than the file size. You can ignore these values as they will not affect your code.

Step 3: Implement and run the PageRank algorithm on LiveJournal graph’s binary files

Follow the instructions in `pagerank.py` to implement the PageRank algorithm.
You will only need to write/modify a few lines of code.

Run the following command to execute your PageRank implementation:

```
$ pypy q1_utils.py pagerank <path to JSON file for LiveJournal>
```

This will output the 10 nodes with the highest PageRank scores.

For example: `$ pypy q1_utils.py pagerank toy-graph/toy-graph.json`

```
node_id score
1       0.4106875
2       0.2542078125
0       0.1995421875
5       0.0643125
4       0.04625
3       0.025
```

(Note that only 6 nodes are printed here since the toy graph only has 6 nodes.)

Step 4: Experiment with different number of iterations.

Find the output for the top 10 nodes for the LiveJournal graph for n=10, 25, 50 iterations (try the `--iterations n` argument in the command above; the default number of iterations is 10). A file in the format **pagerank_nodes_n.txt** for “n” number of iterations. For example:

```
$ pypy q1_utils.py pagerank toy-graph/toy-graph.json --iterations 25
```

You may notice that while the top nodes' ordering starts to stabilize as you run more iterations, the nodes' PageRank scores may still change. The speed at which the PageRank scores converge depends on the PageRank vector's initial values. The closer the initial values are to the actual pagerank scores, the faster the convergence.

Deliverables

1. **warmup.py [6pt]**: your modified implementation.
2. **out_warmup.bin [3pt]**: the binary file, automatically generated by your modified warmup.py.
3. **out_warmup_bytes.txt [1pt]**: the text file with the number of bytes, automatically generated by your modified warmup.py.
4. **pagerank.py [14pt]**: your modified implementation.
5. pagerank_nodes_n.txt **[6pt]**: the 3 files (as given below) containing the top 10 node IDs and their pageranks for n iterations, automatically generated by q1_utils.py.
    - **pagerank_nodes_10.txt [2pt]** for n=10
    - **pagerank_nodes_25.txt [2pt]** for n=25
    - **pagerank_nodes_50.txt [2pt]** for n=50

Q2 [50 pts] Random Forest Classifier

Note: You must use Python 2.7 for this question.

You will implement a random forest classifier in Python. The performance of the classifier will be evaluated via the out-of-bag (OOB) error estimate, using the provided dataset.
Note: You **must not** use existing machine learning or random forest libraries like scikit-learn.

You will use the UCI Credit Approval Dataset where each record is a credit card application. All attribute names and values have been changed to meaningless symbols to maintain confidentiality. The dataset has been cleaned to remove missing attributes. The data is stored in a comma-separated file (csv) in your Q2 folder as **hw4-data.csv**. Each line describes an instance using 16 columns: the first 15 columns represent the attributes of the application, and the last column is the ground truth label for credit card approval (0 means "denied", 1 means "approved"). **Note:** The last column **should not** be treated as an attribute.

You will perform binary classification on the dataset to determine if a credit card application is safe to approve or not.

Essential Reading

Decision Trees

To complete this question, you need to develop a good understanding of how decision trees work. You can refer to the lecture slides from class. Specifically, you need to know 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 provide an excellent example of how to construct a decision tree using *Entropy* and *Information Gain*.

Random Forests

To refresh your memory about random forests, see Chapter 15 in the "Elements of Statistical Learning" book, lecture slides. 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 (also discussed in class). Out-of-bag (OOB) error estimate has shown to be reasonably accurate and unbiased. Below, we summarize the key points about OOB described 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 cases are left out of the bootstrap sample and not used in the construction of the *kth* tree. For each record left out in the construction of the *kth* tree, it can be assigned a class 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 predicted class is not equal to the true class of a record averaged over all records is the OOB error estimate.

Starter Code

We have prepared starter code written in Python for you to use. This would help you load the data and evaluate your model. The following files are provided for you:

- `util.py`: utility functions that will help you build a decision tree
- `decision_tree.py`: a decision tree class that you will use to build your random forest
- `random_forest.py`: a random forest class and a main method to test your random forest

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. `util.py`: implement the functions to compute entropy, information gain, and perform splitting.
2. `decision_tree.py`: implement the `learn()` method to build your decision tree using the utility functions above.
3. `decision_tree.py`: implement the `classify()` method to predict the label of a test record using your decision tree.
4. `random_forest.py`: implement the functions `_bootstrapping()`, `fitting()`, `voting()`

As you solve this question, you will need to think about multiple parameters in your design, some may be more straightforward to determine, some may be not (hint: study lecture slides and essential reading above). For example,

- Which attributes to use when building a tree?
- How to determine the split point for an attribute?
- When do you stop splitting leaf nodes?
- How many trees should the forest contain?

Note that, as mentioned in class and on lecture slides, there are other approaches to implement random forests. For example, instead of information gain, other popular choices include 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), to help standardize students solutions to help accelerate our grading efforts.

Deliverables

1. **hw4-data.csv**: The dataset used to develop your program. Do not modify this file.
2. **[10 pts] util.py**: The source code of your utility functions.
3. **[30 pts] decision_tree.py**: The source code of your decision tree implementation.
4. **[10 pts] random_forest.py**: The source code of your random forest implementation with appropriate comments.

Q3 [30 points] Using Weka

You will use Weka, a popular machine learning software, to train classifiers for the same dataset used in Q2, and to compare the performance of your random forest implementation with Weka's.

Download and install Weka. Note that Weka requires Java Runtime Environment (JRE) to run. We suggest that you install the latest JRE, to avoid Java or runtime-related issues.

How to use Weka:

- Load data into *Weka Explorer* : Weka supports file formats such as arff, csv, xls.
- Preprocessing: you can view your data, select attributes, and apply filters.

- Classify: under *Classify -> Classifier* you can select the different classifiers that Weka offers. You can adjust the input parameters of many models by clicking on the text to the right of the *Choose* button in the *Classifier* section.

A. Experiments (10 pts)

Run the following experiments. After each experiment, report your **parameters, running time, confusion matrix, and prediction accuracy**. An example is provided below, under the "**Deliverables**" section. For the Test options, choose **10-fold cross validation.** You need to preprocess the data before using classifiers (see the note below).

1. **[2 pts] Random Forest**. Under *Classify -> Classifier -> trees*, select *RandomForest*.
2. **[2 pts] Logistic Regression**. Under *Classify -> Classifier*, select *Logistic*.
3. **[2 pts] Multi-layer Perceptron**. Under *Classify -> Classifier*, select *MultilayerPerceptron*.
4. **[2 pts] SVM**. Under *Classify -> Classifier*, select *SGD*. Right-click the classifier, and click on "*Show properties...*" to set parameters. Choose *Hinge loss* as the loss function, so that you can use linear SVM with SGD optimization.
5. **[2 pts] Your choice** -- choose any classifier you like from the numerous classifiers Weka provides. You can use package manager to install the ones you need.

Note: You may not be able to create the confusion matrix initially (it may be grayed out). The reason is that all your features/columns are **Numeric**. Convert the last column (your label) to **Nominal** using filters to resolve this issue.

B. Parameter Tuning and Discussions (20 pts)

1. For each classifier you used above, try to modify **at least one default parameter** and see how the performance changes. **Report your observations for each classifier, the parameter(s) that you modified and any difference(s) observed (e.g., running time, confusion matrix, and prediction accuracy).** Briefly explain why your change of parameter can improve or worsen the prediction accuracy. Use no more than 50 words for each classifier. An example is provided below, under the "**Deliverables**" section.
    a. **[2 pts] Random Forest** parameter tuning and report.
    b. **[2 pts] Logistic Regression** parameter tuning and report.
    c. **[2 pts] Multi-layer Perceptron** parameter tuning and report.
    d. **[2 pts] SVM** parameter tuning and report.
    e. **[2 pts] Your choice classifier** parameter tuning and report.
2. **[5 pts]** Compare the Random Forest result from A1 to your implementation in Q2 and discuss possible reasons for the difference in performance, using no more than 50 words.
3. **[5 pts]** Among the 5 approaches you used in B.1, select the one that you think is the best performing, and justify your selection. Your justification may consider the approach's running time, accuracy, confusion matrix, etc., or a combination of them. Use no more than 100 words.

Deliverables

report.txt - A text file containing the Weka result and your discussion for all questions above. For example:

```
Section A
1.
        J48 -C 0.25 -M 2
        Time taken to build model: 3.73 seconds
        Overall accuracy: 86.0675 %
        Confusion Matrix:
            a     b   <-- classified as
         33273  2079 |     a = no
          4401  6757 |     b = yes
2.
…

Section B
1a.
        I modified ___.
        Runtime increased by  ___.
        It was because ___.
        …

1b.
        …

2.
        The ___ of my implementation was ___, compared to weka’s ___. It was because ___.

3.
        In terms of ___, the best performing approach was ___, because ___.
```

Submission Guidelines

Submit the deliverables as a single **zip** file named ***hw4-LastName-FirstName.zip*** (should start with lowercase hw4). Write down the name(s) of any students you have collaborated with on this assignment, using the text box on the T-Square submission page.

The zip file's directory structure must exactly be (when unzipped):

```
hw4-LastName-FirstName/
     Q1/
          warmup.py
          out_warmup.bin
          out_warmup_bytes.txt
          pagerank.py
          pagerank_nodes_10.txt
          pagerank_nodes_25.txt
          pagerank_nodes_50.txt

     Q2/
          hw4-data.csv
          util.py
          decision_tree.py
          random_forest.py

     Q3/
          report.txt
```

You must follow the naming convention specified above.