CME 211: Lecture 24

Topic: C++ Object Oriented Programming, Part Duex

Example 1: name algorithm

`../lecture-09/code/names.py`:

```python
class NameClassifier:
    def __init__(self, femalefile, malefile):
        self.LoadNameData(femalefile, malefile)

    def LoadNameData(self, femalefile, malefile):
        # Creates a dictionary with the name data from the two input files
        self.namedata = {}
        f = open(femalefile,'r')
        for line in f:
            self.namedata[line.split()[0]] = 1.0
        f.close()

        f = open(malefile,'r')
        for line in f:
            name = line.split()[0]
            if name in self.namedata:
                # Just assume a 50/50 distribution for names on both lists
                self.namedata[name] = 0.5
            else:
                self.namedata[name] = 0.0
        f.close()

    def ClassifyName(self, name):
        if name in self.namedata:
            return self.namedata[name]
        else:
            # Don't have this name in our data
            return 0.5
```

`../lecture-09/code/main.py`:

```python
import names

# Create an instance of the name classifier
classifier = names.NameClassifier('dist.female.first', 'dist.male.first')

# Setup test data
testdata = ['PETER', 'LOIS', 'STEWIE', 'BRIAN', 'MEG', 'CHRIS']

# Invoke the ClassifyName() method
for name in testdata:
    print('{}: {}'.format(name, classifier.ClassifyName(name)))
```

Output:

```
$ python3 main.py
PETER: 0.5
```

```
LOIS: 1
STEWIE: 0.5
BRIAN: 0.5
MEG: 1
CHRIS: 0.5
$
```

Top level

name/main.cpp:

```cpp
#include <iostream>
#include <string>
#include <vector>

#include "names.hpp"

int main(int argc, char* argv[]) {
  std::string female_file = "dist.female.first";
  std::string male_file = "dist.male.first";
  if (argc == 3) {
    female_file = argv[1];
    male_file = argv[2];
  }

  auto classifier = NameClassifier(female_file,male_file);
  std::cout << "There are " << classifier.getNumberNames();
  std::cout << " names in our reference data." << std::endl;

  std::vector<std::string> testdata;
  testdata.push_back("PETER");
  testdata.push_back("LOIS");
  testdata.push_back("STEWIE");
  testdata.push_back("BRIAN");
  testdata.push_back("MEG");
  testdata.push_back("CHRIS");

  for (auto& name : testdata) {
    std::cout << name << ": " << classifier.classifyName(name) << std::endl;
  }

  return 0;
}
```

Interface

name/names.hpp:

```cpp
#ifndef NAMES_HPP
#define NAMES_HPP

#include <string>
#include <tuple>
#include <unordered_map>
```

```
typedef std::unordered_map<std::string,std::tuple<double,double,unsigned int>> namemap;

class NameClassifier
{
  namemap female;
  namemap male;

  void readData(std::string filename, namemap &names);

 public:
  NameClassifier(std::string female, std::string male);
  double classifyName(std::string name);
  namemap::size_type getNumberNames(void);
  namemap::size_type getNumberFemaleNames(void);
  namemap::size_type getNumberMaleNames(void);
};

#endif /* NAMES_HPP */
```

Implementaion

name/names.cpp:

```
#include <fstream>
#include <iostream>
#include <string>
#include <tuple>

#include "names.hpp"

NameClassifier::NameClassifier(std::string file_female, std::string file_male) {
  // Read each of the files
  readData(file_female, female);
  readData(file_male, male);
}

void NameClassifier::readData(std::string file, namemap& names) {
  std::ifstream f(file);
  if (not f.is_open()) {
    std::cerr << "ERROR: Could not open file " << file << std::endl;
    exit(1);
  }

  std::string name;
  double perc1, perc2;
  unsigned int rank;
  while (f >> name >> perc1 >> perc2 >> rank) {
    names[name] = std::make_tuple(perc1, perc2, rank);
  }

  f.close();
}
```

```
double NameClassifier::classifyName(std::string name) {
  auto f = female.find(name);
  auto m = male.find(name);

  // name was not found
  if (f == female.end() and m == male.end()) return 0.5;

  // definitely male or female
  if (f == female.end()) return 0.;
  if (m == male.end()) return 1.;

  // somewhere in between
  return std::get<0>(f->second)/(std::get<0>(f->second) + std::get<0>(m->second));
}

namemap::size_type NameClassifier::getNumberNames(void) {
  return getNumberFemaleNames() + getNumberMaleNames();
}

namemap::size_type NameClassifier::getNumberFemaleNames(void) {
  return female.size();
}

namemap::size_type NameClassifier::getNumberMaleNames(void) {
  return male.size();
}
```

Putting it together

Output:

```
$ make -C name clean
make: Entering directory '/home/nwh/Dropbox/courses/2015-Q4-cme211/lecture-prep/lecture-24/name'
rm -f main
make: Leaving directory '/home/nwh/Dropbox/courses/2015-Q4-cme211/lecture-prep/lecture-24/name'
$ make -C name main
make: Entering directory '/home/nwh/Dropbox/courses/2015-Q4-cme211/lecture-prep/lecture-24/name'
g++ -std=c++11 -Wall -Wextra -Wconversion main.cpp names.cpp -o main
make: Leaving directory '/home/nwh/Dropbox/courses/2015-Q4-cme211/lecture-prep/lecture-24/name'
$ ./name/main name/dist.female.first name/dist.male.first
There are 5494 names in our reference data.
PETER: 0.0026178
LOIS: 1
STEWIE: 0.5
BRIAN: 0.00135685
MEG: 1
CHRIS: 0.108597
```

Example 2: user similarity

- Homework 2
- Uses MovieLens dataset

- Computes user similarities based on Pearson Correlation Coefficient (PCC)

Python performance

On my workstation.

```
$ python3 similarity.py u.data sim_cpy.txt
Input MovieLens file: u.data
Output file for similarity data: sim_cpy.txt
Minimum number of common users: 5
Read 100000 lines with total of 1682 movies and 943 users
Computed similarities in 39.075 seconds
```

C++ implementation

- Uses same algorithm and data structures
- Only difference is high level versus low level language
- Let's review the implementation…

C++ performance

```
$ g++ -std=c++11 -O3 -Wall -Wextra -Wconversion similarity.cpp -o similarity
./similarity u.data sim_cpp.txt
Input MovieLens file: u.data
Output file for similarity data: sim_cpp.txt
Minimum number of common users: 5
Read 100000 lines with total of 1682 movies and 943 users
Computed similarities in 4.83 seconds
```

Easier options?

There are alternatives to pure Python or pure C++:

- PyPy: Implementation of Python that uses Just-in-Time compilation to improve performance (see: http://pypy.org/).
- Numba: Uses annotations in Python code to speed up your application using compilation (see: http://numba.pydata.org/). This seems to work best on operations with NumPy arrays.
- Cython: Extends Python with C like constructs to create compiled extensions (see: http://cython.org/).

We will look at PyPy.

Warnings:

- Not all of these options support code that uses NumPy
- Also, they tend to be more experimental so you will need to retest if your code works properly

PyPy performance

```
$ pypy similarity.py u.data sim_pypy.txt
Input MovieLens file: u.data
Output file for similarity data: sim_pypy.txt
Minimum number of common users: 5
Read 100000 lines with total of 1682 movies and 943 users
Computed similarities in 19.097 seconds
```

Performance summary

- CPython: 39.1 seconds
- PyPy: 19.1 seconds
- C++: 4.83 seconds (with `-O3`)

(On Nick's workstation)