Big Data Analytics Programming
Assignment 3: *Nearest Neighbors with Product Quantization*

Maaike Van Roy *maaike.vanroy@cs.kuleuven.be*
Laurens Devos *laurens.devos@cs.kuleuven.be*

Collaboration policy
Projects are independent: no working together! You must come up with how to solve the problem independently. Do not discuss specifics of how you structure your solution, etc. You cannot share solution ideas, pseudocode, code, reports, etc. You cannot use code that is available online. You cannot look up answers to the problems online. If you are unsure about the policy, ask the professor in charge or the TAs.

1 Introduction

The nearest neighbor algorithm (NN) is frequently applied to massive datasets. The naive algorithm has $O(n^2)$ complexity, and thus does not scale to large data. Many approaches exist to speed up nearest neighbors, like for example, index structures. In this assignment, you will implement NN with **product quantization** and measure how it performs compared to a naive, brute-force algorithm.

Concretely, you will:

- Implement a naive, brute-force nearest neighbor algorithm using only Python and `numpy`.
- Implement an approximate nearest neighbor algorithm using product quantization in C++.
- Compare the different approaches in terms of accuracy, run time, and the effect of the hyper-parameters of product quantization on its performance.

Template code and an efficient `scikit-learn` solution is provided as a baseline to compare to (`SklearnNN` in `nn.py`). These are the provided files:

```
assignment3
 ├─ main.py (program entry point)
 ├─ nn.py (base classes)
 ├─ bindings.cpp (makes C++ functions available from Python)
 ├─ setup.py (Python project configuration)
 ├─ util.py (utility functions and settings)
 ├─ pydata.hpp (Python pointer wrapper)
 ├─ (*) functions.py
 ├─ (*) prod_quan_nn.hpp
 └─ (*) prod_quan_nn.cpp
```

Make modifications to and implement the functions in the files indicated with a `(*)`.

You can compile the project using Python `setuptools`:

```
python setup.py build
```

This should create a `build` folder containing the C++ Python extension binary. Make sure you have a C++ compiler installed on your system.

To run the program, simply execute the `main.py` script.

```
python main.py
```

This script exposes a command line interface (CLI). The `--help` argument prints usuage instructions:

```
> python main.py --help
usage: main.py [-h]
               [--task {time_and_accuracy,distance_absolute_error,plot,retrieval,hyperparam}]
               [-k K] [-n N] [--seed SEED]
               [{covtype,emnist,emnist_orig,higgs,spambase}]

Execute the code and experiments for BDAP assignment 3.

positional arguments:
  {covtype,emnist,emnist_orig,higgs,spambase}

optional arguments:
  -h, --help           show this help message and exit
  --task {time_and_accuracy,distance_absolute_error,plot,retrieval,hyperparam}
  -k K                 number of neighbors
  -n N                 size of test set sample
  --seed SEED          seed for random sample selection
```

Five datasets have been provided, and can be found in `/cw/bdap/assignment3`:

- `spambase` (default): a small dataset useful for quick debugging
- `emnist`: extended-`mnist`, a dataset of hand-written digits and characters
- `emnist_orig`: same as previous, but different data representation
- `higgs`: dataset with signal processes some of which produce Higgs bosons (class 1)
- `covtype`: forest cover type dataset classifying lodgepole pine versus others

The CLI prodives a `--task plot` option that only works with the `emnist` and `emnist_orig` datasets. It plots a hand-written digit or character with its k nearest neighbors. Try it out!

We recommend you use a virtual environment for Python. There are multiple options, like `venv` or conda. For `venv`, you can create a new virtual environment as follows:

```
python -m venv <name_of_venv>

# Activate the virtual environment (Unix):
source <name_of_venv>/bin/activate

# Install required packages in environment:
pip install numpy matplotlib scikit-learn
```

2 Naive Nearest Neighbors with `numpy` in Python (5 pts)

For this part of the assignment, implement the nearest neighbor algorithm in Python using only `numpy`.

In `functions.py`, implement the function `numpy_nn_get_neighbors`. Given a training set `xtrain`, a test set `xtest`, and a parameter k, compute for each example in `xtest` the k nearest neighbors in `xtrain`. The method returns two $n \times k$ matrices (n is number of rows in `xtest`), one containing the indices of the nearest neighbors in `xtrain`, and another containing the respective Euclidean distances.

You are not allowed to use any libraries other than `numpy`. Any use of another library will result in a zero score for this exercise.

Implementing the `numpy_nn_get_neighbors` will complete the `NumpyNN` class in `nn.py`, which you will use for the experimentation.

3 Nearest Neighbors with Product Quantization in C++ (10 pts)

For this part of the assignment, implement an approximate nearest neighbor algorithm in C++ using *product quantization* as seen in the lecture. Code for the clustering of the partitions is provided. You have to write the evaluation function `compute_nearest_neighbors` in `prod_quan_nn.cpp`.

The provided C++ code consists of a header file (`prod_quan_nn.hpp`) and implementation file (`prod_quan_nn.cpp`). Look at the comments in the files and implement the required parts. Doing this will complete the `ProdQuanNN` Python class in `nn.py`, which you will use for the experimentation.

4 Experiments (10 pts)

After completing the first two tasks, you now have three different implementations (all in `nn.py`):

1. `SkleanNN`: a provided NN implementation based on scikit-learn.
2. `NumpyNN`: your own `numpy`-based NN implementation.
3. `ProdQuanNN`: your own *product quantization* NN implementation.

Run the following experiments, each corresponding to a Python function in `functions.py`, to see how well these three methods perform:

1. Compare the evaluation time and accuracy of each implementation. To do this, implement the `time_and_accuracy_task` function.
2. Compute the mean absolute error on the distance to the nearest neighbors made by `ProdQuanNN`. To do this, implement the `distance_absolute_error_task` function.
3. Compute how often the true nearest neighbor is in the set of k closest neighbors as reported by `ProdQuanNN`. To do this, implement the `retrieval_task` function.
4. Determine the best values for the hyper-parameters of `ProdQuanNN`: (a) the number of partitions `npartitions` and (b) the number of clusters within each partition `nclusters`. To do this, implement the `hyperparam_task` function.

For each experiment above, a corresponding `--task` option value exists in the CLI exposed by `main.py`. Make sure you do not change the output formats of the functions, as we will be using your output to check your solutions.

5 Report (5 pts)

Write a **one-page** report answering the following questions. Keep your answers brief and to the point.

1. Why is the approximate NN method using product quantization faster than the naive NN method?
2. Show in a plot the effect of the two hyper-parameters of product quantization (`npartitions` and `nclusters`) on the accuracy of `ProdQuanNN`.
3. What is the best set of hyper-parameters? Interpret the results in the plot: do you have an intuitive explanation of why this is the case? Is there a trade-off you have to make when choosing the parameters?
4. Include a table showing the results for questions 1-3 of the experiments for each dataset using the best hyper-parameters for `ProdQuanNN`. Make sure to include the accuracies of `SkleanNN` and `NumpyNN`. Use $k = 10$.

6 Submission

The project is due **Monday, June 13th at 9am**. Because this due date is close to the examination period, no late submissions are allowed.

Test your implementation before submission:

Only the following files are graded:

```
assignment3
 ├─ functions.py
 ├─ prod_quan_nn.hpp
 └─ prod_quan_nn.cpp
```

Any changes to made to the other files **are not considered** during grading; we will use our own test files that expect the same interface as in the code template.

Specifically, your code should compile and run successfully on the departmental computers using the unmodified versions of the files `main.py`, `nn.py`, `bindings.py`, `pydata.hpp`, `setup.py`, `util.py`. Make sure to follow these guidelines:

- Do not change the method headers of any C++ function in `prod_quan_nn.hpp`.
- You may add fields and methods to the ProdQuanNN C++ class to structure your implementation. Put your method implementations in `prod_quan_nn.cpp`. If you choose to use additional class fields, you may initialize them in the constructor, and/or in `initialize_method()`.
- Do not change the input arguments or the output format of the Python functions in `functions.py`.

You are allowed to use the C++ standard library. You may not use any other C++ libraries.

7 Grading Criteria

Your submission will be graded on correctness, efficiency, and style. Note that even though `ProdQuanNN` is an approximate algorithm, the results are deterministic given the partitions and its clusters.