KNN Sequential and Parallel Performance Analysis

Overview

We analyze the performance of different parallelization techniques on a K-Nearest Neighbors (KNN) classifier. We evaluate three versions of the classifier, each employing a different approach to parallelize the predictions or Euclidean distance calculations.

Approaches

1. Sequential Version

In this version, all predictions and Euclidean distance calculations are done sequentially:

- `predict()` method iterates through each test sample, calling `_predict()` to find the K-nearest neighbors.
- `_predict()` calculates the Euclidean distance between the input test sample and each training sample in a sequential loop.

2. Parallel `predict()`

In this version, the `predict()` method is parallelized using a process pool. This method distributes the prediction of each input sample in `X` across multiple processes:

- Each sample's prediction is independent of the others, making it straightforward to parallelize.
- `Pool().map()` is used to execute `_predict()` for each sample in parallel. Each process handles predicting a different test sample without needing to share data or coordinate with other processes.

3. Parallel + Vectorized Euclidean Distance Calculation

In this version, we combine parallelized `predict()` with a vectorized Euclidean distance calculation in `_predict()`:

- The `predict()` method is parallelized to distribute the test samples.
- Within `_predict()`, the Euclidean distance calculation is vectorized using NumPy's `np.linalg.norm()`, eliminating the need for an explicit loop:

  ```
  distances = np.linalg.norm(self.X_train - x, axis=1)
  ```

- We believe this vectorized approach is more efficient due to optimized operations in NumPy.

4. FAILED: Parallel Eucledian Distance Calculation in `predict_()`

Note that we could parallelize Eucledian distance calculation in `predict_()`, since the calculations between `x` and `x_train` for each `x_train` in `X_train` are independent from each other. However, this would mean creating a separate process for each `x` in `X` in `predict()`, which would introduce a huge overhead. In other words, it would only degrade the performance to parallelize eucledian distances calculations. Indeed, we tested this idea with the following code and got slower results then just calculating the Eucledian distances in sequential manner:

```
def _predict(self, x):
  pairs = [(x, x_train) for x_train in self.X_train]
  with Pool(self.n_jobs) as pool:
    distances = pool.starmap(self.euclidean_distance, pairs)
  ... # rest of the code is the same
```

Run Instructions

Setup the server

To run this notebook on HPC you will first need to follow these steps

1. Log into your HPC instance by using the command: `ssh -p 8022 <your-login>@access-aion.uni.lu` (replace your-login with your HPC login)
2. Once logged in, initialize a session using `salloc` and specifying a large enough timeframe (e.g. 4h) and multiple CPUs (e.g. 4) by typing this command `salloc --time=04:00:00 --cpus-per-task=4`
3. Create a directory inside HPC: `mkdir project_1`
4. Go to the directory: `cd project_1`
5. Load python: `module load lang/Python`
6. Install/Upgrade pip: `python -m pip install --upgrade pip`
7. Install the numpy module: `pip install numpy`

Copying files

1. On your local machine, in a different terminal/shell tab, within the directory where you have saved our project_1 folder, run the following commands: `scp -P 8022 project_1/* <your-login>@access-aion.uni.lu:~/project_1`
2. Enter your password as you would if you were going to log into HPC

Running the script

1. Once your files are copied to HPC, go back to the HPC server you have setup
2. From within the `project_1` directory run this command `nohup python your_script.py &`

3. The script will run in the background without interuption
4. When the script finishes running check the output using the command `cat nohup.out`
5. The output from running our script is detailed in the Results section.

Results

Metrics

Each version is evaluated based on the following metrics:

- Execution time, i.e. real time: Time taken from start to finish.
- CPU Time: Time the CPU spends actively computing (excluding I/O and waiting).

We report the average and standard deviation for each metric across multiple runs.

KNN Versions Results

Below is the output of our code after running it on HPC (Aion):

```
Measuring Sequential Function Performance
  Correct 742
  Correct 742
  ...(for all 30 runs)
  'sequential' Performance over 30 runs:
  Real Time - Avg: 70.7807 s, Std: 0.3678 s
  CPU Time  - Avg: 70.3930 s, Std: 0.3623 s


Measuring Parallel Functions Performances
Number of CPU cores and n_jobs: 128

Parallel Function Performance
  Correct 742
  Correct 742
  ...(for all 30 runs)
  'parallel' Performance over 30 runs:
  Real Time - Avg: 19.3559 s, Std: 0.1787 s
  CPU Time  - Avg: 2.1187 s, Std: 0.2065 s

Parallel Vectorized Function Performance
  Correct 742
  Correct 742
  ...(for all 30 runs)
  'parallelVectorized' Performance over 30 runs:
```

```
Real Time - Avg: 2.9927 s, Std: 0.1604 s
CPU Time  - Avg: 1.4172 s, Std: 0.1664 s
```

The results show that `parallel` version of the KNN algorithm significantly outperforms the `sequential` one, achieving approximately 33 times faster execution on average in CPU time over 30 runs. Additionally, our `parallelVectorized` version of the algorithm surpasses both `sequential` and `parallel` implementations, running approximately 1.5 times faster than the `parallel` version.