Benchmarking Numpy and Breeze

Thanh Trung Nguyen

tnguy404@ucsc.edu

Summary

- Numpy (Python) vs Breeze (Scala)
- Application: Principal Component Analysis (PCA) algorithm implemented from scratch
- Input set:
  - Randomly Generated matrices (and saved as csv files) of different sizes generated based on Normal distribution with mean = 1 and variance = 1
  - Input size ranging in 2 dimensions: # of features and # of samples:

    # of features: 4, 8, 16, 32, 64, 128

    # of samples: 1000, 2000, 4000, 8000, 16000, 32000

Summary (continue)

- Time: Numpy was in generally faster than Breeze across datasets, especially with small dataset/early steps of large datasets

- Performance metrics:
  - Numpy has better performance (Lower execution time, higher IPC, lower branch prediction miss rate, lower cache miss rate) in the initial phase
  - Breeze catches up quickly, has higher performance after initial phase

Numpy

- https://github.com/numpy/numpy
- Numerical library for Python
- Written in C
- Supports large, multi-dimensional arrays and matrices
- The base of a lot of other libraries: SciPy, Scikit-learn, Tensorflow, Pytorch, openCV, etc
- Targeted users: Data Scientists, Mathematicians

Breeze

- https://github.com/scalanlp/breeze
- Numerical library for Scala
- Written in Scala
- Target: becoming a “Numpy” library for Scala
- Written in Scala so convenient to use with Apache Spark (distributed computing engine) to build ETL to process data in realtime/large scale
- Targeted users: Data Engineers

Application

- Principal Component Analysis: Removing highly dependent features to avoid overfitting in Machine Learning
- I.e.: running ML algorithm on a housing dataset: House price feature and Annual property tax feature is highly dependent

=> PCA can automatically remove one of them

- I implemented this algorithm from scratch

Application Structure

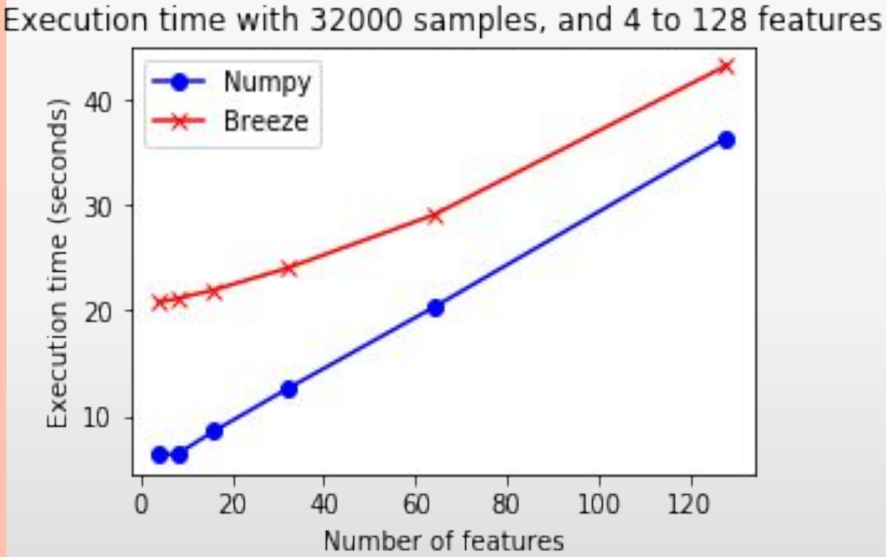
- Read csv dataset and create the dataset matrix
- Arithmetic computations:
  - Compute the d-dimensional mean vector
  - Compute the covariance matrix
  - Compute eigenvectors (e_1, e_2,..., e_d) and corresponding eigenvalues (λ_1, λ_2,..., λ_d)
  - Sort the eigenvectors by decreasing eigenvalues, choose k eigenvectors with the largest eigenvalues
  - Use this d*k eigenvector matrix to transform the samples onto the new subspace
- Write the result to a csv file

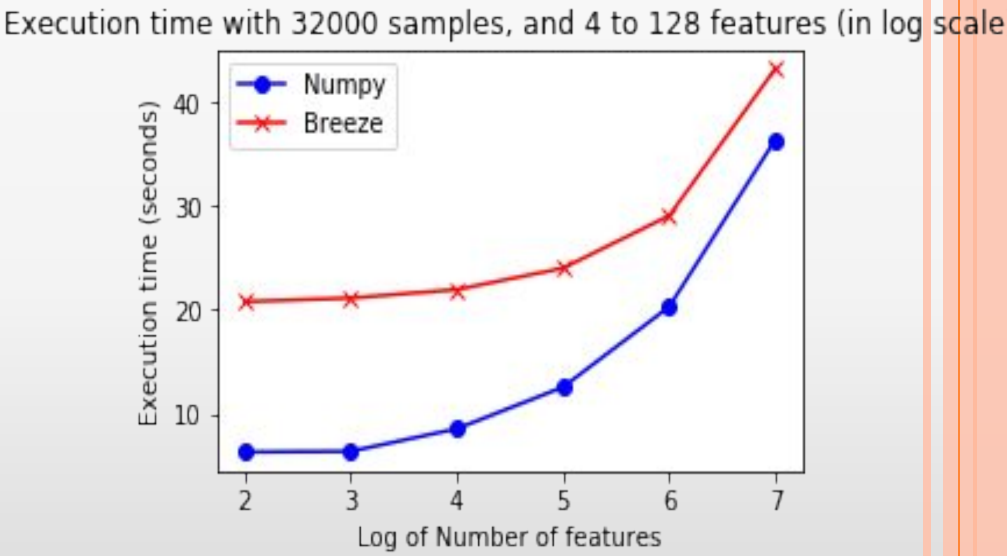
Input sets

- csv files of different sizes
- Data randomly created from a normal distribution with mean = 1 and variance = 1
- Number of features (dimensions): 4, 8, 16, 32, 64, 128
- Number of samples: 1000, 2000, 4000, 8000, 16000, 32000

Execution Time

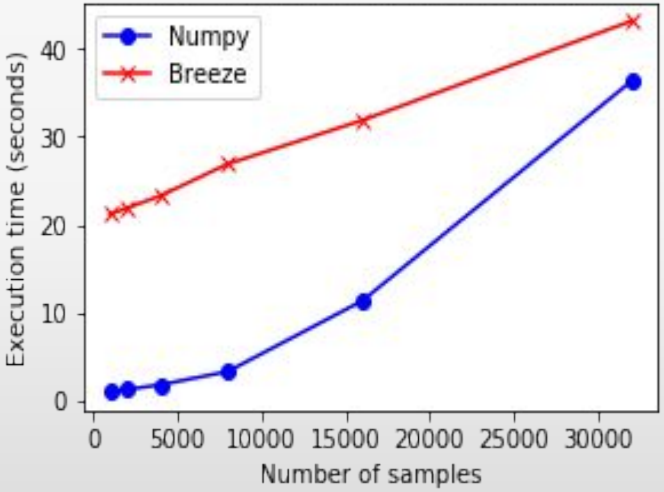
# samples = 32000, different feature sizes:



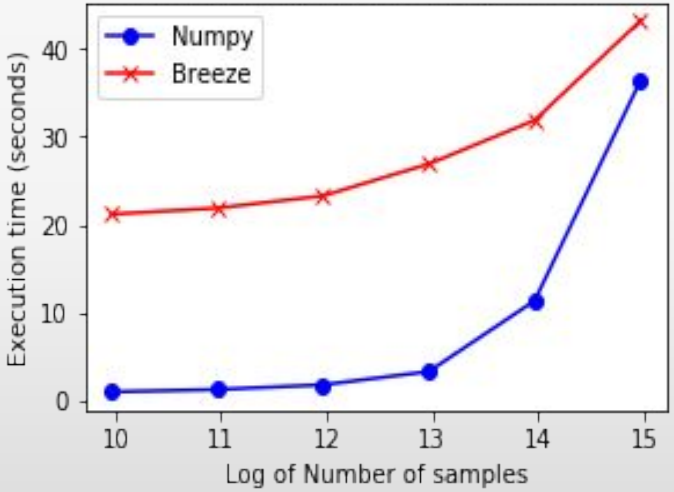


Execution Time

# features = 128, different sample sizes:



Execution time with 128 features, and 1000 to 32000 samples



Execution time with 128 features, and 1000 to 32000 samples (in log sca

Execution Time Plot Analysis

- Breeze was a lot slower than Numpy with small datasets (for 4 features, 1000 samples: Breeze took ~20 seconds, Numpy took ~1 second)
- Breeze catched up quickly with bigger datasets
- The execution time is as my expectation:
  - Since the dataset size is doubled every time, I expect the runtime increases with exponential rate
  - Therefore the plot with respect to log(# of features/samples) of run time should be exponential and plot wrt # of features/samples should be linear (which is true)

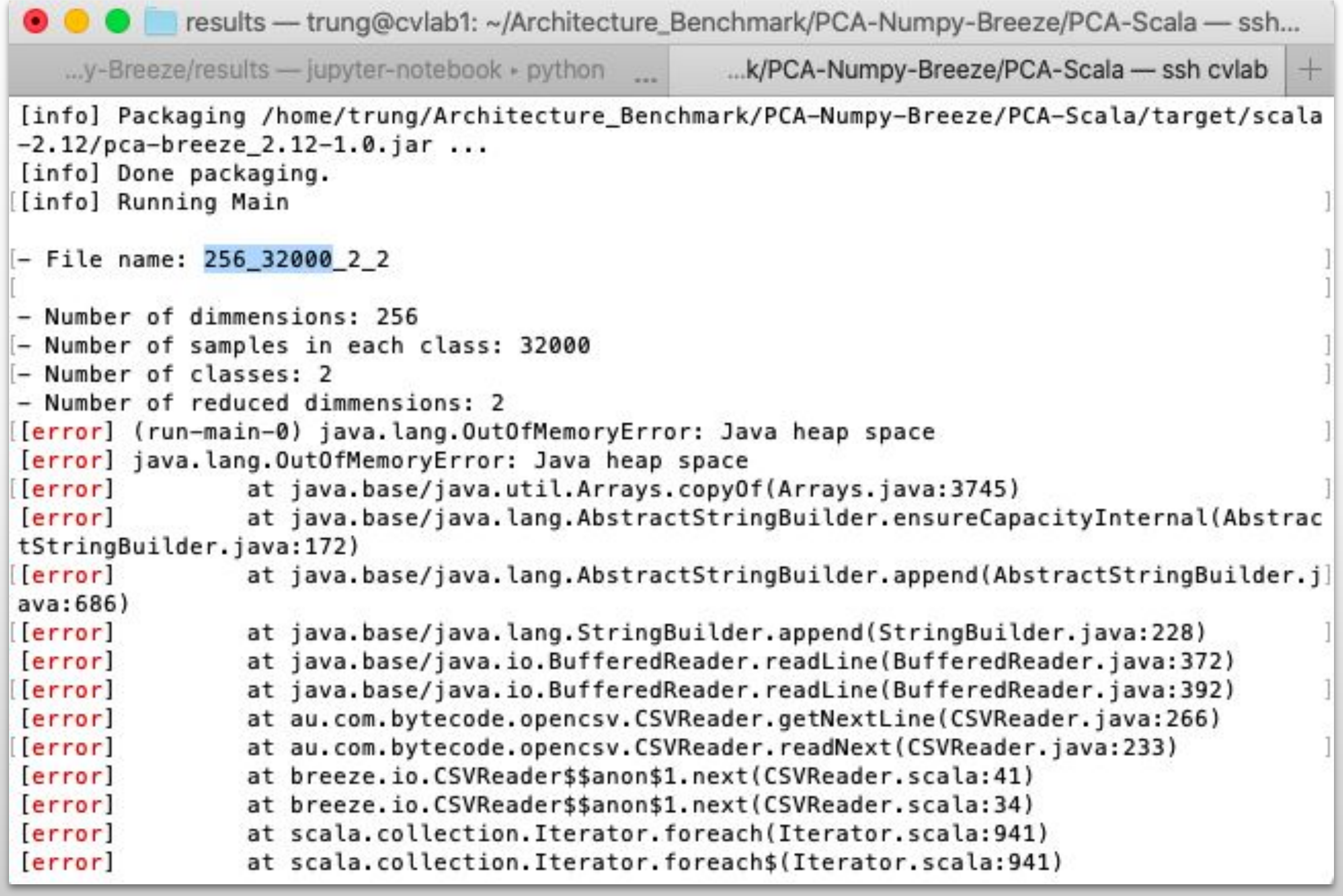
Dataset Limit

It would be interesting to see if Breeze can exceed Python's runtime performance with a bigger dataset, however:

- Python could process dataset of upto 512 dimensions and 32000 samples
- Scala could not process dataset of 256 dimensions and 32000 samples and up (Java heap space problem)

=> I wasn't able to run Scala on a bigger dataset

Dataset Limit


results — trung@cvlab1: ~/Architecture_Benchmark/PCA-Numpy-Breeze/PCA-Scala — ssh...
...y-Breeze/results — jupyter-notebook ▸ python
...k/PCA-Numpy-Breeze/PCA-Scala — ssh cvlab
+

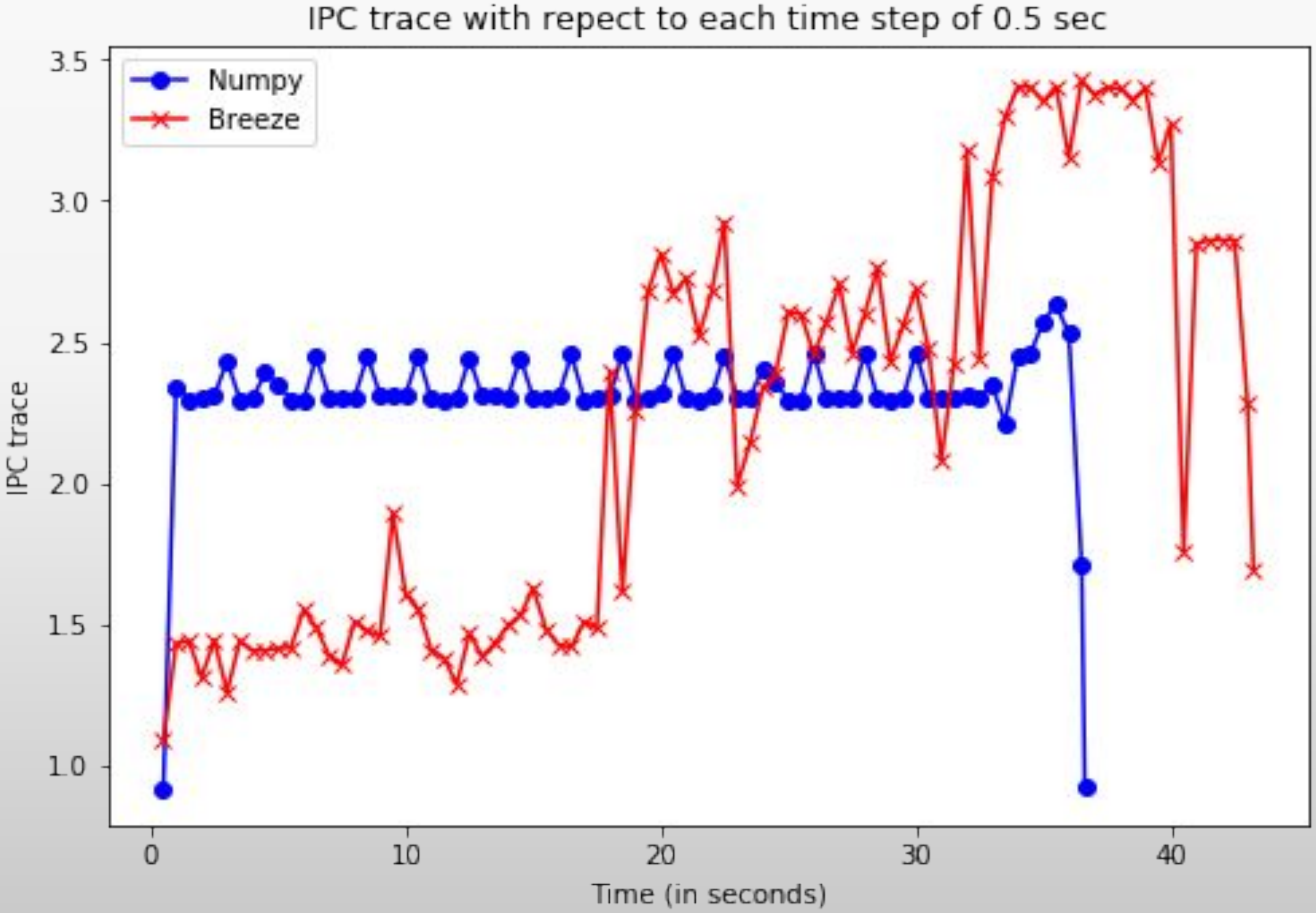
```
[info] Packaging /home/trung/Architecture_Benchmark/PCA-Numpy-Breeze/PCA-Scala/target/scala
-2.12/pca-breeze_2.12-1.0.jar ...
[info] Done packaging.
[info] Running Main

- File name: 256_32000_2_2

- Number of dimmensions: 256
- Number of samples in each class: 32000
- Number of classes: 2
- Number of reduced dimmensions: 2
[error] (run-main-0) java.lang.OutOfMemoryError: Java heap space
[error] java.lang.OutOfMemoryError: Java heap space
[error]         at java.base/java.util.Arrays.copyOf(Arrays.java:3745)
[error]         at java.base/java.lang.AbstractStringBuilder.ensureCapacityInternal(Abstrac
tStringBuilder.java:172)
[error]         at java.base/java.lang.AbstractStringBuilder.append(AbstractStringBuilder.j
ava:686)
[error]         at java.base/java.lang.StringBuilder.append(StringBuilder.java:228)
[error]         at java.base/java.io.BufferedReader.readLine(BufferedReader.java:372)
[error]         at java.base/java.io.BufferedReader.readLine(BufferedReader.java:392)
[error]         at au.com.bytecode.opencsv.CSVReader.getNextLine(CSVReader.java:266)
[error]         at au.com.bytecode.opencsv.CSVReader.readNext(CSVReader.java:233)
[error]         at breeze.io.CSVReader$$anon$1.next(CSVReader.scala:41)
[error]         at breeze.io.CSVReader$$anon$1.next(CSVReader.scala:34)
[error]         at scala.collection.Iterator.foreach(Iterator.scala:941)
[error]         at scala.collection.Iterator.foreach$(Iterator.scala:941)
```

IPC

Note: The IPC and all the MPKI plots correspond to the biggest dataset with 32000 samples and 128 features



IPC Plot Analysis

- Breeze's IPC was lower than Numpy in the first 20 seconds, higher in the later 20 seconds
- This corresponds correctly to the Execution Time plot above: even with a small dataset, Breeze took ~20 seconds to process

=> Up on investigation, I reason this as Scala runs on JVM which initializes needed classes and the classes must be loaded when they are used

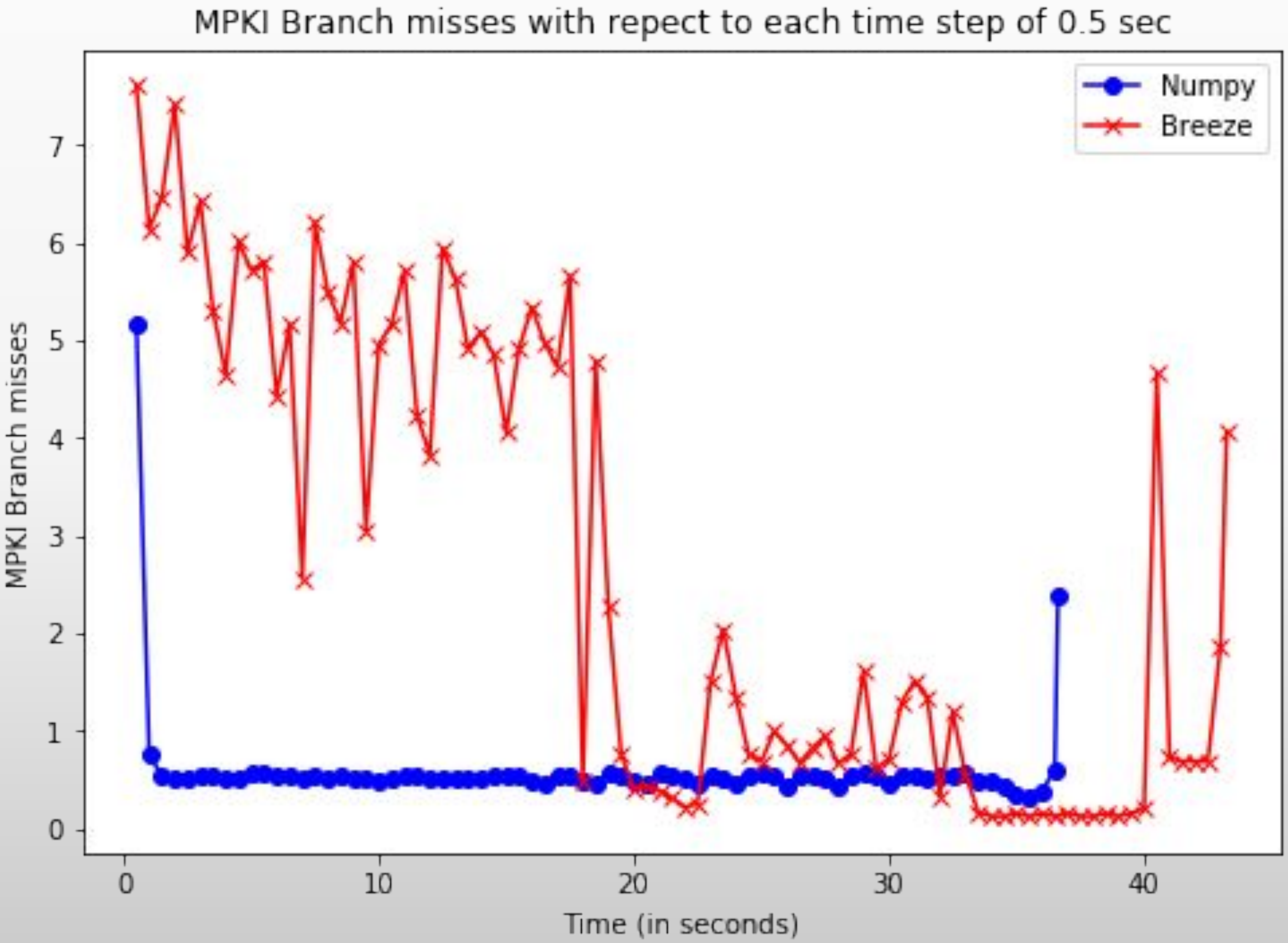
=> Take much more time than C/C++ initially

IPC Plot Analysis (Continue)

- Breeze’s IPC fluctuated quite a lot
- Numpy’s IPC was stable, except in the beginning and at the end

=> This may be due to the program’s structure: In the beginning it has to load the csv dataset (fluctuate phase), then process using mostly arithmetic computations (stable phase), and then save the result to a csv file (fluctuate phase)

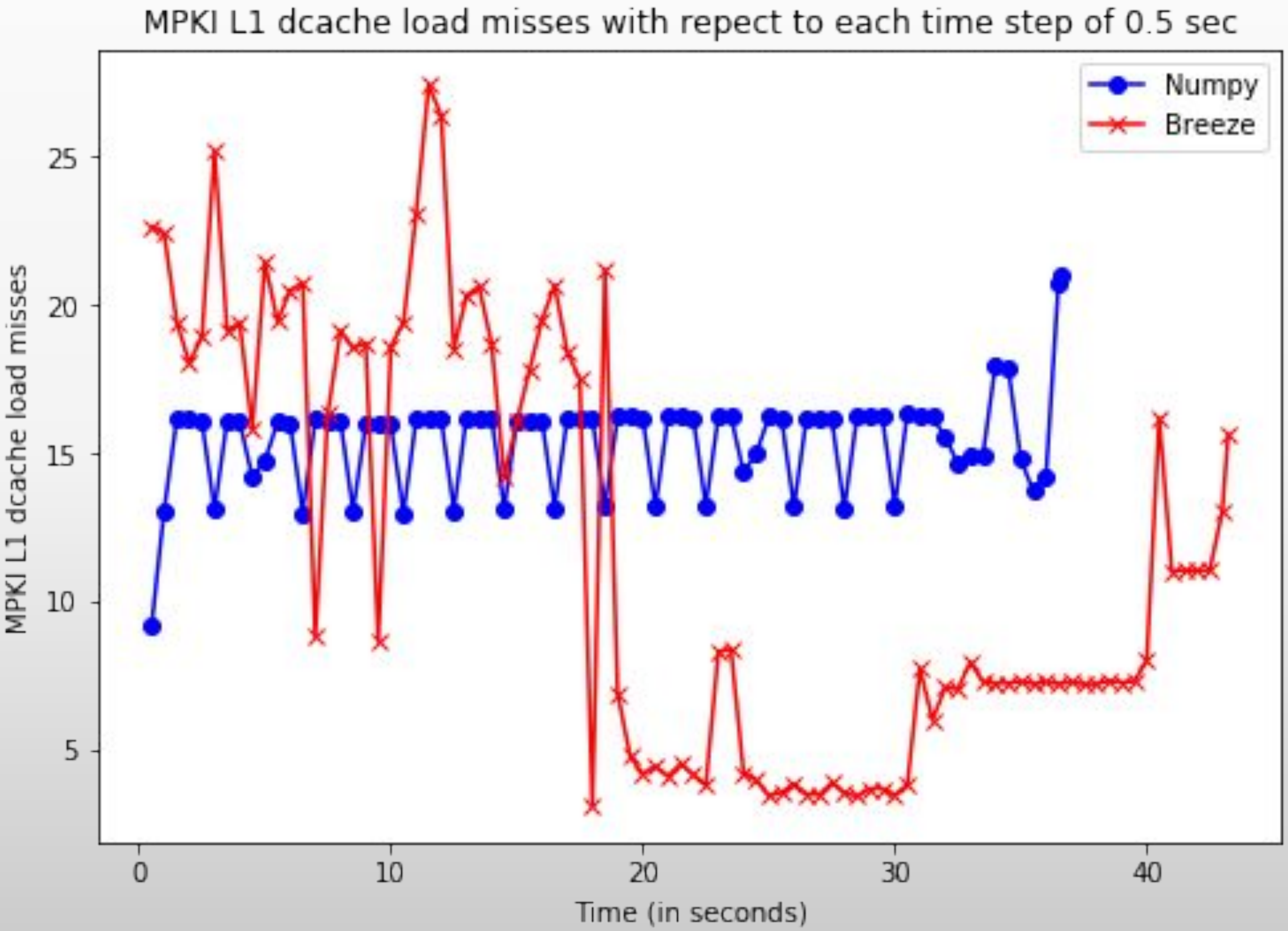
Branch Prediction (MPKI)



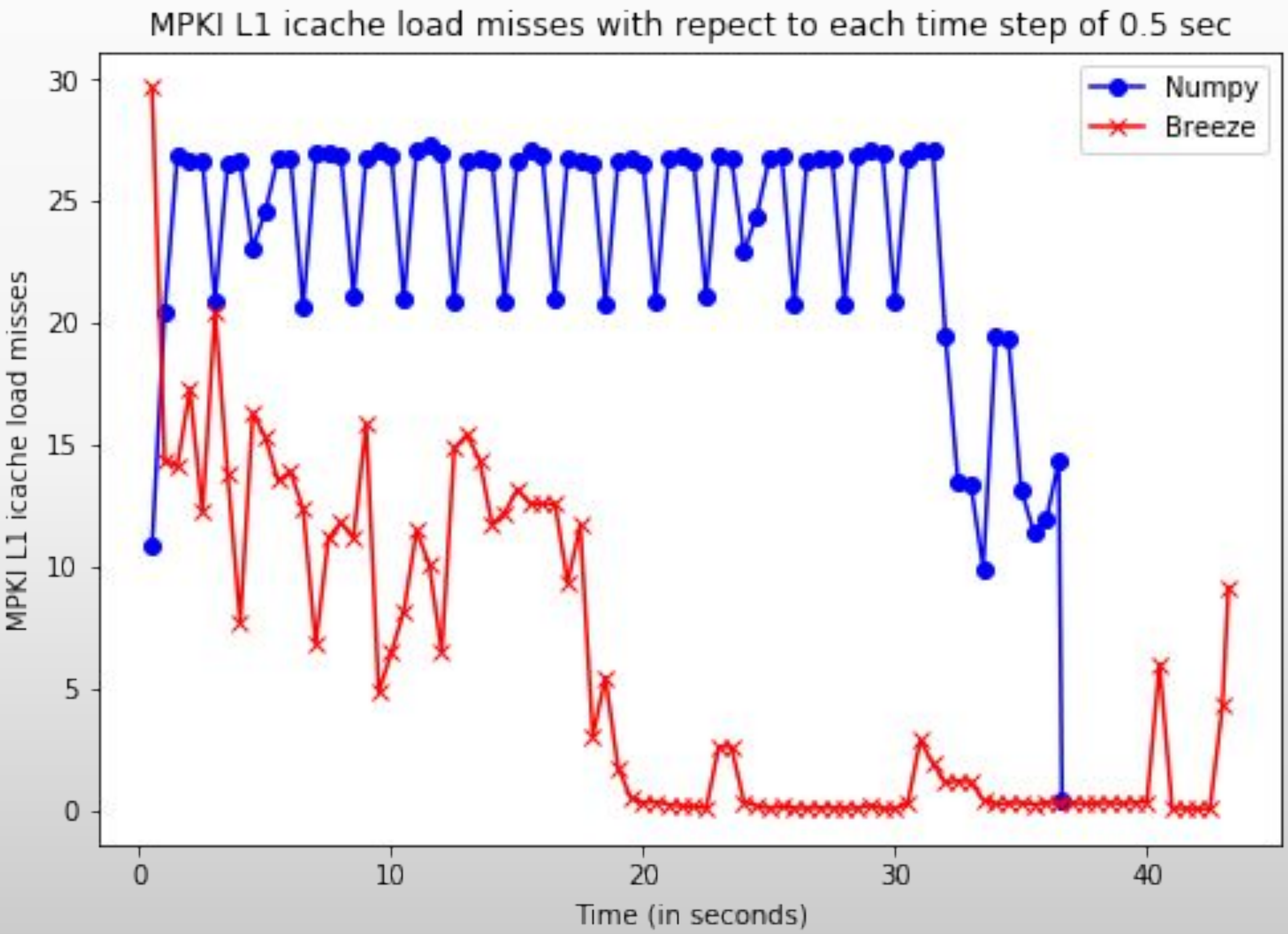
Branch Prediction (MPKI) Analysis

- This looks reasonable: Prediction miss rates are only ~ 0-8 instructions for every 1024 instructions
- Numpy has an overall better branch prediction rate
- Both has higher miss rates at begin phase (for Numpy: first 1 second, for Scala: first 20 seconds) and end phase, and lower at the computation phase

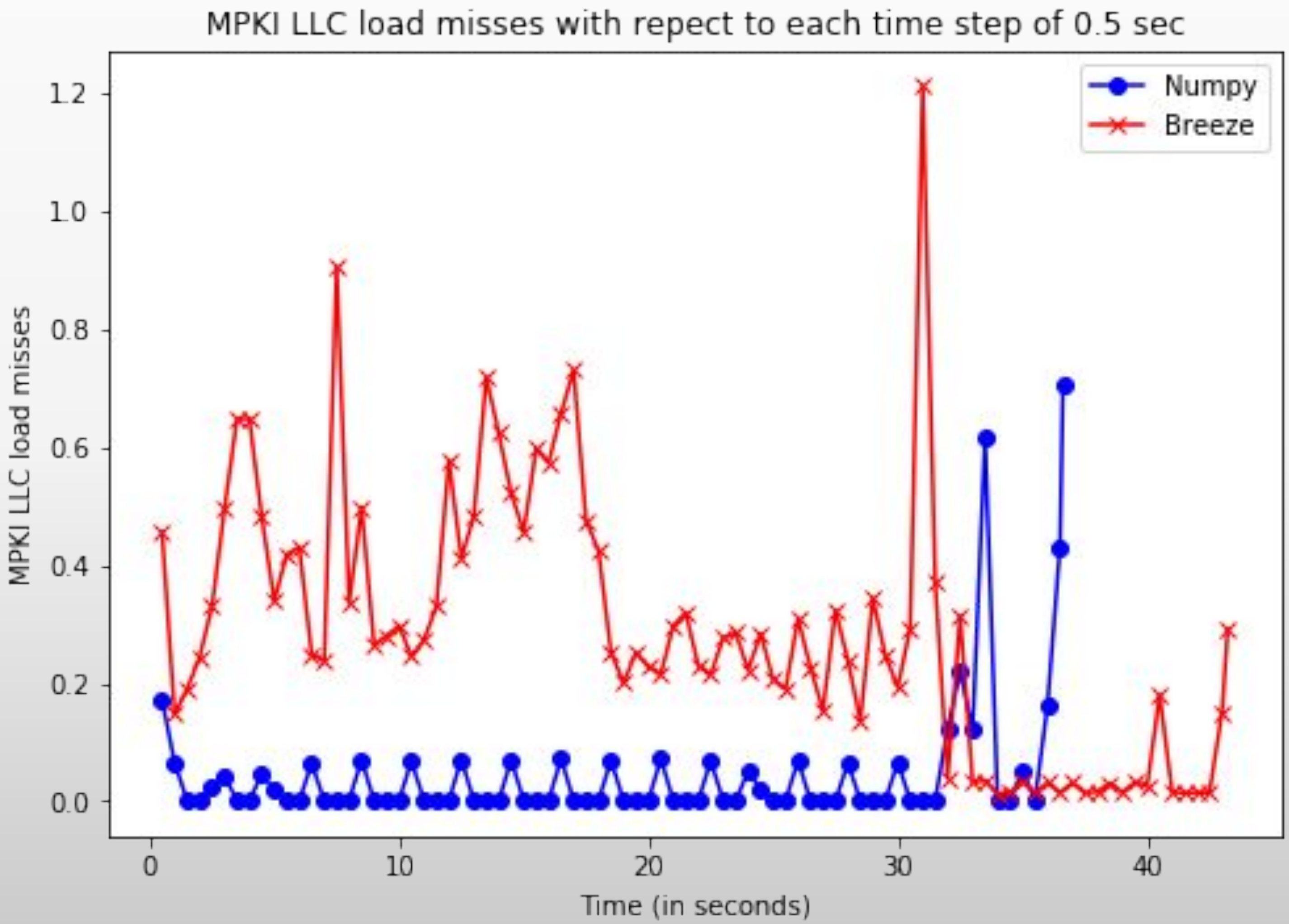
L1 d-cache misses (MPKI)



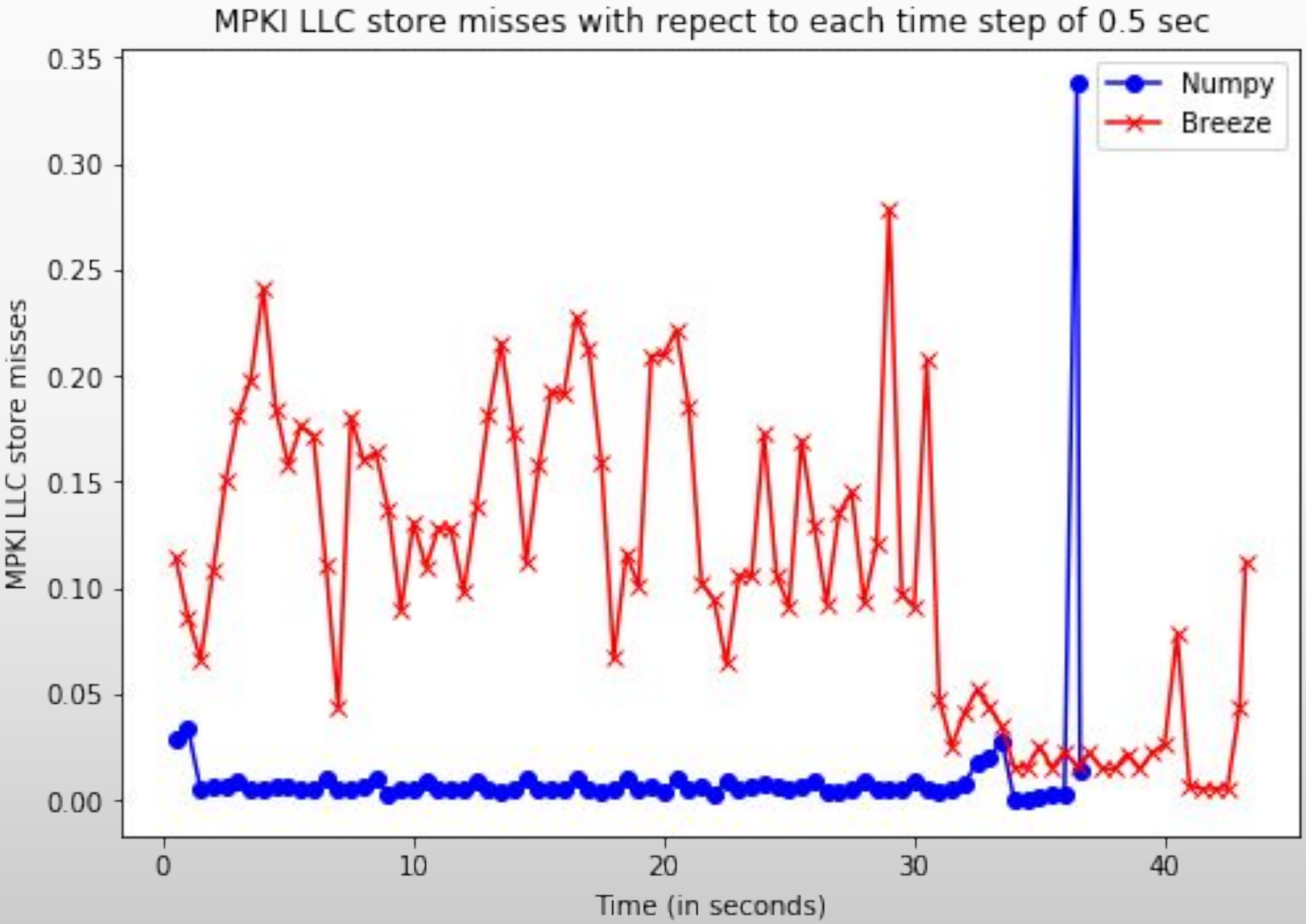
L1 i-cache misses (MPKI)



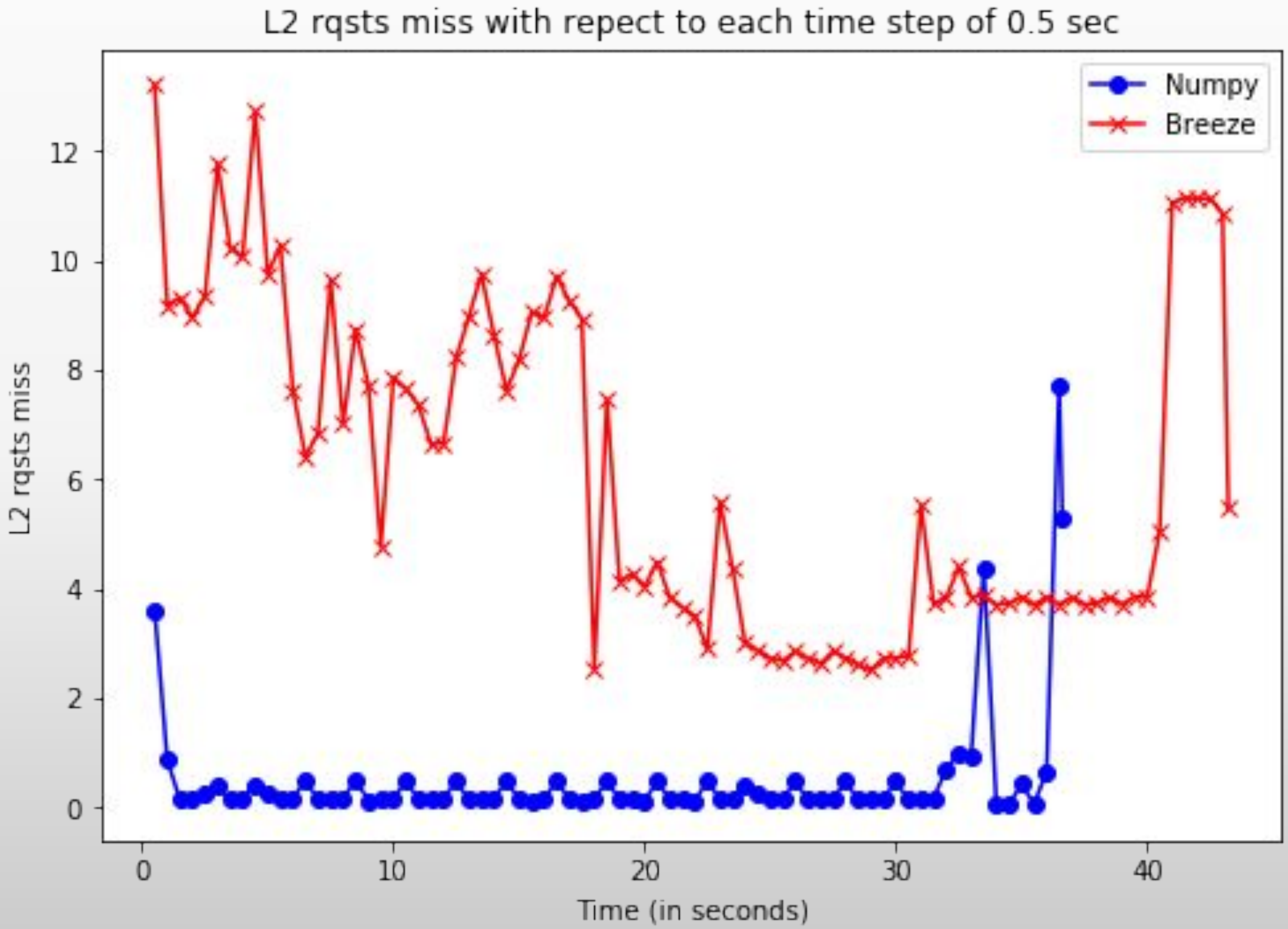
LLC load misses (MPKI)



LLC store misses (MPKI)



L2 rqsts misses (MPKI)



Cache misses (MPKI) Analysis

- Breeze has higher L1 d-cache miss rates in the initial phase (first 20 seconds), but lower L1 Cache Miss Rates than Numpy after that.

=> This is in accordance with the previous Execution Time and IPC plots

- Numpy has lower L2 cache miss rates across runtime

Main observations, Conclusions

- Numpy’s performance was quite consistent
- Breeze’s performance was worse that Numpy in the initial phase, but got better after that
- Why Numpy was faster than Breeze in the early steps?
  - Numpy is not written in Python but C, so could achieve a really good speed
  - Breeze is Scala-based so runs on JVM. Therefore it initializes needed classes and the classes must be loaded when they are used => The start-up time of a Scala program is often longer than C/C++

How to Reproduce results

Generate input csv files:

- Run file data_generator.py
- It creates csv files with sizes
- Number of features: ranging from 4 to 512
- Number of samples: ranging from 1000 to 128000
- To modify number of maximum features: edit line 7 of data_generator.py
- To modify number of maximum data lines: edit line 10 of data_generator.py

How to Reproduce results (continue)

- Run the Python Numpy application:

```
$ python PCA-Numpy-Breeze/PCA-Python.py
```

- Run the Scala Breeze application:

```
$ cd PCA-Numpy-Breeze/PCA-Scala/
$ sbt run
```

CPU Information

CV Lab's CPU

```
processor	: 5
vendor_id	: GenuineIntel
cpu family	: 6
model		: 85
model name	: Intel(R) Xeon(R) Bronze 3106 CPU @ 1.70GHz
stepping	: 4
microcode	: 0x2000064
cpu MHz		: 800.169
cache size	: 11264 KB
physical id	: 0
siblings	: 8
core id		: 5
cpu cores	: 8
apicid		: 10
initial apicid	: 10
fpu		: yes
fpu_exception	: yes
cpuid level	: 22
wp		: yes
```