The Korean Journal of Applied Statistics (2020) DOI: https://doi.org/10.5351/KJAS.2020.36.6.665
33(6), 665–682

Introduction to numba library in Python for efficient statistical computing

Younsang Cho[a] · Donghyeon Yu[a] · Won Son[b,1] · Seoncheol Park[c]

[a]Department of Statistics, Inha University; [b]Department of Information Statistics, Dankook University; [c]Pacific Climate Impacts Consortium, University of Victoria

(Received September 22, 2020; Revised October 15, 2020; Accepted October 16, 2020)

Abstract

This paper introduces `numba` library in Python, which improves computational efficiency of the provided implemented code written by naive Python language by applying just-in-time (JIT) compilation. To apply just-in-time compilation, the `numba` only needs to use a decorator on a target Python function. We provide implementation examples with `numba` for the permutation test and the parameter estimation for Gaussian mixture distribution. We also numerically show the efficiency of `numba` by comparing the total computation times of the implementation using naive python and the implementation using `numba` for each application.

Keywords: statistical computing, python, numba, just-in-time compilation

1. 서론

최근 데이터의 생산 및 저장 능력의 급격한 상승으로 대용량 자료 분석에 대한 수요가 증가하고 있으며 머신러닝 및 딥러닝 등 데이터 분석 기법의 발전으로 보다 복잡한 계산이 요구되고 있는 상황이다. 따라서 이를 해결하기 위하여 분산 계산(distributed computing) 및 병렬 계산(parallel computing)이 많이 활용되고 있다. 하지만 분산 및 병렬 계산을 위해서는 먼저 계산하고자 하는 문제가 분산 및 병렬 계산에 맞는지 확인하는 절차가 필요하며 맞지 않는 경우에는 이를 해결하기 위하여 계산 과정 자체를 수정해야할 필요성이 있다. 또한 분산처리 및 병렬 계산을 활용하기 위해서는 각 기능을 지원하는 라이브러리들을 이용하거나 연구자가 직접 저수준 언어들을 기반으로 message passing interface (MPI)를 이용하여 구현해야 한다. 최근 활용도가 높은 GPU를 기반으로 하는 병렬 연산의 경우에는 openCL (Stone 등, 2010) 또는 CUDA C 등을 이용하여 구현해야 한다.

이러한 저수준 언어 기반의 연산은 사용자가 원하는 기능을 효율적으로 설계하고 실행하도록 할 수 있는 장점이 있으나 기본적으로 저수준 언어의 학습과 코드 작성 이후 컴파일 과정 등 추가적인 절차가 요구되어 연구자에게 부담이 될 수 있다. 따라서 본 논문에서는 최근 데이터 분석에 활용도가 높아지고 있는 파이썬(python)에서 통계학 연구자들이 파이썬 구문으로 작성한 통계 계산 과정을 비교적 간단한 절차를 통

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT and Future Planning (NRF-2018R1C1B6001108).
[1] Corresponding author: Department of Information Statistics, Dankook University, 152, Jukjeon-ro, Suji-gu, Yongin-si, Gyeonggi-do 16890, Republic of Korea. E-mail: son.won@dankook.ac.kr

하여 저수준 언어로 구현한 것과 동일한 성능을 지니도록 할 수 있는 numba 라이브러리 (Lam 등, 2015)를 소개하고자 한다. numba는 이미 대용량 데이터에 대한 연산이 필요한 여러 분야에 적용되고 있으며 파이썬 기반의 프로그램의 속도 향상을 위해 사용되고 있다. 예를 들어 Manifold approximation에 활용된 연구 (McInnes 등, 2018)와 박테리아의 유전적 진화 연구의 도구 개발 (Lees 등, 2019)의 구현에서 속도 향상을 위해 적용되었다.

파이썬의 numba 라이브러리 소개에 앞서 통계학에서 많이 활용되는 R에 대하여 언급하면, R에서도 저수준 언어를 활용할 수 있는 여러 함수 및 라이브러리가 존재하며 대표적으로 Rcpp 패키지 (Eddelbuettel 등, 2011)가 많이 활용되고 있다. 하지만 Rcpp는 저수준 언어인 C++로 구현된 코드에 대하여 just-in-time (JIT) 컴파일 기능을 제공하며 numba는 파이썬 구문으로 구현된 코드에 대하여 JIT 컴파일을 제공하는 측면에서 차이가 있다. 여기서 JIT 컴파일이란 미리 컴파일한 라이브러리를 사용하는 것이 아니라 프로그램에서 코드가 처음 호출될 때 한번 컴파일 과정을 거치고 이후에는 컴파일 된 라이브러리를 사용하는 것을 의미한다. 추가로 numba와 유사한 기능을 제공하는 Cython 라이브러리 (Behnel 등, 2010)가 존재하지만 Cython 라이브러리는 기본적으로 C-extensions for python (https://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/)에 해당하는 pyrex(.pyx) 파일로 작성하여 Cython 컴파일러 기반으로 컴파일 하게 된다. 이 경우, Python 코드 및 C 코드들은 모두 C 코드로 변환 후 컴파일 된다. 물론, ipykernel 기반의 jupyter notebook을 사용할 경우 cython magic 명령어를 제공하여 셀 단위로 컴파일 과정을 자동화해주는 기능이 제공되나 사용자 측면에서는 numba의 데코레이터 기반의 JIT 컴파일이 사용 편의성이 높기 때문에 본 논문에서는 numba의 소개를 고려하였다. numba와 Cython의 효율성을 비교한 논문은 찾지 못하였으나 Pairwise distance 계산에 대한 효율성 비교에 대한 온라인 문서(https://jakevdp.github.io/blog/2013/06/15/numba-vs-cython-take-2)가 존재하며 해당 문서의 결과에서 numba가 Cython보다 계산 시간이 빠름이 보고 되었다. 보다 자세한 정보는 해당 문서를 참고하기 바란다.

본 논문에서 파이썬을 기반으로 한 통계 계산에 대하여 소개하는 추가적인 이유는 2019년 데이터 사이언티스트를 대상으로 한 조사(https://www.burtchworks.com/2019/08/21/2019-sas-r-or-python-survey-update-which-tool-do-data-scientists-analytics-pros-prefer/)에서도 나타난 것처럼 데이터 처리 및 분석에 있어서 파이썬의 활용이 증가하고 있는 추세이며 통계 분석에 필요한 다양한 라이브러리들도 개발되어 제공되고 있기 때문이다. 또한 효율적인 통계 계산 측면에서 R에 비해 파이썬이 갖는 강점 중 하나는 바로 연산정밀도에 대한 부분이다. R은 현재 배정밀도(double-precision)에 대한 형식만을 지원한다. 배정밀도의 경우 정확도는 더 높지만 그만큼 데이터를 저장하는 메모리의 크기가 크며 단정밀도(single-precision)에 비해 상대적으로 연산에 많은 시간이 걸린다. 반면에 파이썬은 단정밀도와 배정밀도에 대한 형식을 모두 지원하므로 필요에 따라 정확도를 조금 낮추더라도 연산 속도를 향상시킬 수 있다.

파이썬이 R과 비교 시 데이터에 대한 다양한 형식을 제공하지만, 두 프로그래밍 언어 모두 공통적으로 대화형 인터프리터 방식을 사용하고 있다. 프로그래밍 언어는 기본적으로 컴파일러 방식 또는 대화형 인터프리터 방식을 사용한다. 컴파일러 방식은 전체 프로그램을 한 번에 기계어로 번역하여 호출 가능하도록 만들어 주어 컴파일에 시간이 소비되지만 프로그램의 실행 속도는 빠르다. 반면에 대화형 인터프리터 방식은 입력된 코드를 한 줄 단위로 읽고 번역하여 실행하기 때문에 컴파일 과정이 요구되지 않지만 전체 실행 속도는 컴파일러 방식에 비해 느리다. 따라서 기본적으로 반복문을 많이 사용하여 구현된 연산은 R과 파이썬 모두에서 효율적이지 않다. 일반적으로 미리 컴파일된 라이브러리 기반의 함수를 활용하여 연산을 많이 진행하지만 연구자가 진행하고자 하는 모의 실험 또는 개발하는 방법론 등에서는 반복문의 사용을 피할 수 없는 경우가 있다. numba는 이러한 경우 유용한 라이브러리로 간단한 절차를 통하여 파이썬에서 사용된 반복문을 JIT 컴파일 하여 컴파일 방식으로 구현된 반복문의 성능으로 사용할 수 있게 해준다.

단, numba는 수치형 데이터의 연산에만 적용할 수 있으므로 다양한 데이터 형식에 적용하기 위해서는 적절한 전처리를 수행하여 수치형 자료로 변환해야 한다.

만약 수치형 자료로 변환이 어려운 경우에는 numba를 사용할 수 없으므로 연산 속도 향상을 위해 저수준 언어를 사용해야 한다. R과 C, C++, Fortran 등의 연동과 마찬가지로 Python에서도 저수준 언어와 연동이 가능하다. C 언어의 예를 들면, 파이썬의 기본 라이브러리인 ctypes를 이용하여 C 언어의 함수들을 호출 할 수 있다. 간략히 소개하면, 먼저 C 언어 코드를 gcc와 같은 컴파일러를 통하여 동적라이브러리(Dynamic-link library; DLL)를 생성하고 ctypes 라이브러리의 CDLL 함수를 통하여 라이브러리를 읽고 인스턴스(instance)로 정의한다. 인스턴스는 C 언어 코드에서 정의된 함수 이름을 멤버 변수로 포함하고 있으므로 이를 활용하여 함수를 호출하여 사용할 수 있다. 파이썬에서 C 언어와의 연동에 대한 자세한 방법은 파이썬의 ctypes 문서(https://docs.python.org/3/library/ctypes.html)를 통해 확인할 수 있다.

R과 파이썬에서의 저수준 언어와의 연동을 다시 한 번 정리하면 다음과 같다. 먼저 저수준 언어로 이루어진 함수를 동적라이브러리로 컴파일한다. 그리고 연동에 필요한 함수들(R:.C, .Call, Python:ctypes.CDLL)을 이용해 동적라이브러리를 불러온 뒤 저수준 언어에서 정의된 함수의 이름을 호출하여 사용할 수 있다. 하지만 저수준 언어의 함수를 수정해야 할 경우 매번 컴파일을 다시 진행해야 되는 번거로움이 있다. 본 논문에서 소개하는 numba 라이브러리의 경우 파이썬 환경에서 파이썬 언어로 구현된 연산에 대하여 바로 JIT 컴파일을 제공하므로 보다 간단하게 컴파일러 방식을 사용할 수 있다. numba는 Anaconda (https://anaconda.org) 또는 Miniconda(https://docs.conda.io/en/latest/miniconda.html) 환경에서 "pip install numba" 또는 "conda install numba" 명령어로 간단하게 설치할 수 있다. 설치 후 파이썬 환경에서 import numba로 라이브러리를 불러 온 뒤 numba는 데코레이터(decorator)와 데이터 타입(data type)을 사용할 수 있다.

본 논문은 다음과 같이 구성되었다. 먼저 2절에서는 numba의 기본적인 소개와 데코레이터에 대하여 설명하고, 3절에서는 반복문이 사용되는 통계 계산 문제들 중 순열 검정(permutation test)과 정규 혼합 분포(Gaussian mixture distribution)의 모수 추정 문제에 대한 EM 알고리즘 예제를 통하여 순수한 파이썬 함수를 이용한 계산과 numba를 활용한 계산의 총 계산 시간에 대하여 비교 결과를 제시하였다. 마지막으로 4절에서는 전체 결과에 대한 요약과 numba의 활용 가능성 및 장단점에 대해 논의하며 논문을 마무리한다.

2. Numba 라이브러리

numba는 파이썬의 라이브러리 중 하나로 Low Level Virtual Machine (LLVM) 컴파일러를 사용하여 파이썬의 코드를 기계어로 번역하는 즉, 컴파일을 자동적으로 해주는 라이브러리이다. 이해를 돕기 위해 numba 라이브러리의 컴파일 과정을 Figure 2.1에 나타내었다. Figure 2.1에 제시된 것과 같이 파이썬의 함수는 기본적으로 함수 정의 시 바이트 코드(bytecode) 형태로 저장이 된다. 따라서 numba는 제어 흐름 그래프(control flow graph)를 이용하여 파이썬 함수의 바이트 코드를 numba의 중간 언어(intermediate representation; IR) 형태로 만든다. 이후 입력 인수뿐만 아니라 함수 내에서 사용되는 모든 변수들의 데이터 타입을 추론하여 중간 언어 상태에서 최적화한다. 변수에 대한 추론이 끝나면 저수준 언어로 번역할 수 있는지의 여부에 따라 nopython LLVM 또는 object mode LLVM의 중간 언어를 생성한다. 마지막으로 두 가지의 LLVM의 중간 언어를 통합하여 기계어로 컴파일하게 된다. 여기서 nopython이란 파이썬 객체(object)가 아닌 저수준 언어 기반으로 컴파일 되는 것을 의미하며 object mode란 파이썬의 객체 구조가 반영되는 컴파일 과정을 의미한다.

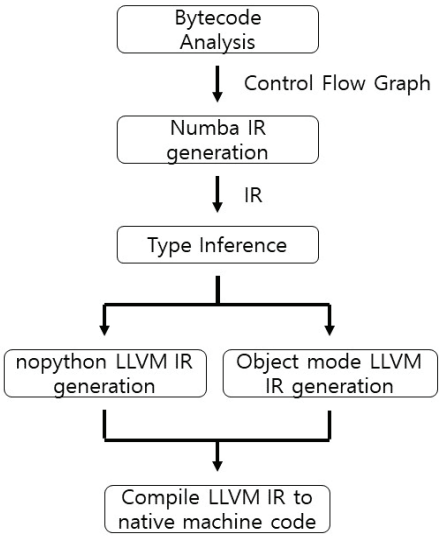

Figure 2.1. Compilation process in numba.

numba 라이브러리는 데코레이터를 파이썬 함수에 적용하여 손쉽게 사용할 수 있다. 데코레이터에 대해 간략히 설명하자면 함수를 인자로 받는 함수로써 흔히 래퍼(wrapper) 또는 래핑 함수(wrapping function)라고도 한다. 이 데코레이터를 파이썬 함수의 함수 선언부 위에 `@decorator_function`을 추가하는 방식으로 사용할 수 있다. 여기서 편의 상 데코레이터의 이름을 `decorator_function`으로 가정하고 서술하였다. 위와 같이 numba의 데코레이터로 감싼 함수는 JIT 컴파일 방식으로 컴파일된다. 컴퓨터 프로그램의 실행은 크게 Interpretation 방식과 Ahead-of-time (AOT) 방식으로 분류된다. Interpretation 방식은 사용자가 프로그램을 실행시키는 동시에 코드를 해석하여 기계어로 번역하는 방식으로 R과 파이썬도 이 방식을 사용한다. AOT 컴파일은 미리 바이트코드와 같은 기계어로 번역해둔 후 실행하는 방식으로 Java의 바이트코드가 이러한 예이다. JIT 컴파일은 두 방법을 절충한 방법으로 사용자가 코드를 처음 실행시킬 때 기계어로 컴파일하며, 이후에는 이전 실행에서 컴파일된 실행파일을 다시 사용하기 때문에 컴파일 작업을 필요로 하지 않아 함수의 실행을 빠르게 할 수 있다. numba에서 사용하는 JIT 컴파일러와 JIT 컴파일에 대한 자세한 설명은 Lam 등 (2015)에서 확인할 수 있다.

파이썬의 numba 라이브러리와 유사하게 R에서도 `Rcpp` 패키지 함수를 통해 JIT 컴파일을 사용할 수 있다. `Rcpp`와 numba의 차이점은 `Rcpp`의 경우 JIT 컴파일 되는 함수 자체가 R코드가 아닌 C++ 코드로 이루어진 함수라는 점이다. `Rcpp` 패키지의 내장함수인 cppFunction은 인수(argument)로 C++ 코드를 입력한 후 실행함과 동시에 컴파일을 진행한다. JIT 컴파일 방식을 사용하므로 처음 이후에는 컴파일 없이 R의 내장함수와 같이 함수 이름을 호출하여 사용할 수 있다.

2.1. 벡터 연산

numba는 벡터 연산을 위해 `vectorize`와 `guvectorize` 데코레이터를 제공하고 있다. `vectorize` 데코레이터는 입력 인수와 출력값이 동일한 길이를 가지는 벡터들의 원소 연산을 다루기 위한 데코레이터이다.

물론 numpy 라이브러리에도 벡터들 사이의 연산에 대한 함수들이 구현되어있다. 단순한 벡터의 내적 또는 행렬 곱과 같은 선형 대수적인 연산은 numpy 라이브러리가 효율적이다. 하지만 벡터의 연산이나 행렬의 연산에 조건문을 사용하여야 하는 특수한 경우 함수를 직접 구현해야 할 수 있다. 이러한 상황에서 numba를 활용할 경우 손쉽게 계산 속도를 향상시킬 수 있다. 아래 예제 2.1과 예제 2.2는 각각 vectorize 데코레이터에 입력 인수와 출력값의 데이터 타입을 명시적으로 컴파일러에게 알려주는 eager compilation과 컴파일러가 자동적으로 입력 인수와 출력값의 데이터 타입을 추론하여 컴파일하는 lazy compilation을 보여준다. eager compilation의 경우, vectorize 데코레이터에 "([출력 데이터타입(입력 인수의 데이터타입)])" 형식으로 추가하여 표현한다.

예제 2.1: 두 벡터의 원소별 최댓값 계산 (eager compilation)

```
@vectorize([float64(float64, float64)])
def eager_ele_max(x, y):
    if x >= y:
        return x
    else:
        return y
```

예제 2.2: 두 벡터의 원소별 최댓값 계산 (lazy compilation)

```
@vectorize
def lazy_ele_max(x, y):
    if x >= y:
        return x
    else:
        return y
```

위의 예제들은 두 벡터의 원소별로 최댓값을 반환하는 함수로 예제 2.1은 두 벡터의 데이터 타입이 float64(배정밀도)라는 것을 명시적으로 컴파일러에게 제공하였다. 따라서 예제 2.1은 eager compilation이 진행되며, 예제 2.2는 데코레이터 뒤에 데이터 타입을 명시하지 않았으므로 lazy compilation이 진행된다. Lazy compilation은 입력 인수와 출력값의 타입을 컴파일러에게 추론하도록 하여 코드를 간결하게 할 수 있지만, eager compilation에 비해 데이터 타입을 추론하는 과정이 추가적으로 진행됨으로써 컴파일 시간이 느려진다. 또한 위 예제는 앞에서 언급한 조건문이 필요한 벡터 연산으로 numpy보다 numba를 사용하는 것이 효율적이다. numpy 라이브러리에도 예제 2.1, 2.2와 같은 계산을 할 수 있는 maximum( ) 함수가 제공되지만 위 함수와 비교해보면 numba 함수의 eager compilation의 경우가 더 빠른 것을 확인할 수 있었다. 예제 데이터로는 정규분포에서 랜덤으로 100,000개의 난수를 추출하여 생성한 표본을 사용하였으며 이 데이터에 대해 크기를 비교하는 과정을 10,000번 반복하였다. 이 때 numpy.maiximum( ) 함수의 실행시간은 약 483.13 μs(microsecond)였으며, vectorize의 경우 약 25.42 μs로 numpy 라이브러리가 약 19배 정도의 시간이 더 소요되었다. 위의 예제에서는 numba를 이용하는 것이 numpy를 이용하는 것보다 빠르게 나타났으나 속도의 차이는 구현 방법과 연산에 따라 달라질 수 있다. 위의 예제에서 계산 소요 시간에 큰 차이가 나타나는 것은 매우 간단한 연산인 점과 numpy의 경우에 자체 멤버 변수와 멤버 함수 처리에 따른 추가 절차와 참조 메모리의 인접성(contiguity) 등의 영향을 받은 것으로 판단된다.

Table 2.1. Data types used in numba

데이터 타입	약어	설명
boolean	b1	8비트 논리형 데이터 타입
uint8	u1	8비트 부호가 없는 정수형
uint16	u2	16비트 부호가 없는 정수형
uint32	u4	32비트 부호가 없는 정수형
uint64	u8	64비트 부호가 없는 정수형
int8, char	i1	8비트 정수형, 문자
int16	i2	16비트 정수형
int32	i4	32비트 정수형
int64	i8	64비트 정수형
intc	-	C 타입의 정수형
uintc	-	C 타입의 부호가 없는 정수형
intp	-	정수형을 가리키는 포인터
uintp	-	부호가 없는 정수형을 가리키는 포인터
float32	f4	single-precision 부동 소수점
float64, double	f8	double-precision 부동 소수점
complex64	c8	single-precision 복소수
complex128	c16	double-precision 복소수

numba에 사용할 수 있는 데이터 타입은 Table 2.1에 요약하였다. 일부 데이터 타입의 뒤에 숫자는 각 데이터 타입의 비트 단위의 메모리 크기를 의미한다. 예를 들어 float32는 32비트 메모리 크기를 가지는 부동 소수점 형식 데이터 타입을 의미한다.

vectorize 데코레이터는 입력 벡터와 출력 벡터의 길이, 즉 원소의 수가 동일한 경우에만 사용할 수 있고 길이가 다른 벡터의 연산을 위해서는 guvectorize 데코레이터를 사용하여야 한다. guvectorize 데코레이터는 입력 및 출력 벡터의 크기가 같을 것을 요구하지 않지만 결과를 반환하지 않으므로 입출력 벡터를 모두 함수의 인수로 전달하고 데이터 타입을 명시해야 한다. 이러한 방식은 R에서의 .C() 함수를 이용한 C 언어의 연동 및 CUDA의 커널(kernel) 함수를 사용하는 방식과 유사하다. guvectorize 데코레이터를 적용하기 위해서는 첫 번째 인수로 파이썬 함수의 입출력 인수의 데이터 타입과 배열의 차원을 입력해야 한다. 두 번째 인수로는 1차원 또는 2차원 배열의 원소의 수를 문자 형태로 소괄호 안에 표현하며, 원소의 수가 같은 경우에만 같은 문자로 표현할 수 있다. 스칼라 값일 경우에는 빈 소괄호 '( )'로 표현한다. 아래 예제 2.3을 통하여 guvectorize의 사용법에 대하여 살펴보고자 한다. 예제 2.3에서는 LASSO 회귀모형(Tibshirani, 1996)의 알고리즘에서 많이 적용되는 소프트 임계화(soft-thresholding)를 guvectorize 데코레이터를 이용하여 구현하였다.

예제 2.3: 소프트 임계화 (soft-thresholding)

```
@guvectorize([(float64[:], float64, float64[:])], '(n),()->(n)', target = "parallel")
def soft_thresh(x, y, res):
    for i in range(x.shape[0]):
        abs_x = numpy.abs(x[i])
        if abs_x >= y:
```

```
            res[i] = numpy.sign(x[i]) * (abs_x-y)
        else:
            res[i] = 0
```

vectorize와 guvectorize 데코레이터의 경우 기본적으로 하나의 CPU 코어에서 연산이 이루어지기 때문에 만약 멀티코어 또는 GPU 연산을 하기 위해서는 target = "parallel" (멀티코어) 또는 target = "cuda" (GPU)를 인수로 전달하여 연산을 수행할 하드웨어 방식을 변환할 수 있다.

2.2. Just-in-time 컴파일

jit 데코레이터는 파이썬 함수를 JIT 컴파일하며, numba에서는 nopython mode와 object mode를 제공한다. Nopython mode는 파이썬이 아닌 저수준 언어로 번역하여 컴파일함으로써 효율성을 올릴 수 있는 반면 파이썬 함수 중에서 저수준 언어로 번역할 수 있는 함수들이 제한적이다. 그에 반하여 파이썬 언어로 컴파일하는 object mode의 경우에는 nopython mode에 비해 속도는 느리지만 파이썬 함수의 사용에 있어 제한이 적다. 여기서 object mode는 numba의 컴파일러가 Python C API에 접근해 Python object(파이썬 객체)를 다룰 수 있게 데코레이터에 제공되는 옵션이다. jit 데코레이터도 vectorize와 마찬가지로 eager compilation과 lazy compilation 컴파일 방식을 제공한다.

jit 데코레이터의 사용법을 소개하기 위하여 벡터의 원소의 합을 구하는 함수를 아래의 예제 2.4와 예제 2.5에 나타내었다.

예제 2.4: 벡터의 원소의 합 (nopython mode)

```
@jit('float64(float64[:])', nopython = True)
def njit_sum(x):
    n = x.shape[0]
    res = 0.0
    for i in range(n):
        res += x[i]
    return res
```

예제 2.5: 벡터의 원소의 합 (object mode)

```
@jit(nopython = False)
def objit_sum(x):
    n = x.shape[0]
    res = 0.0
    for i in range(n):
        res += x[i]
    return res
```

예제 2.4는 jit 데코레이터에 vectorize의 eager compilation과 같이 input과 output 데이터형과 배열의 차

원을 명시적으로 컴파일러에게 알려줌으로써 eager compilation을 실행하며, "nopython = True"로 nopython mode로 저수준의 기계어로 번역하여 컴파일한다. 이와 반대로 예제 2.5의 경우는 "nopython = False"로 object mode로 컴파일하며, 데이터형을 제공하지 않았기 때문에 lazy compilation이 진행된다. 예제 2.4와 2.5의 실행시간을 비교하기 위해 표준정규분포에서 랜덤으로 1,000,000개의 난수를 추출하여 표본을 생성하였으며 원소들의 합을 구하는 과정을 1,000번 반복하였다. 시간 측정 결과 예제 2.5의 경우 약 8.0672 ms (millisecond)가 소요되었으며, 예제 2.4의 경우 7.4379 ms가 소요되었다. 참고로 파이썬의 sum 함수는 동일한 조건으로 측정한 결과 약 158.7999 ms가 소요되었다. 위 예제의 경우 간단한 벡터의 원소의 합을 계산하는 단순한 구문이기 때문에 object mode와 nopython mode의 차이가 적지만 복잡한 연산을 진행할수록 두 mode의 실행시간의 차이는 더 크게 벌어진다.

numba가 제공하는 다른 유용한 기능 중 하나로 파이썬의 문법 형식으로 작성된 코드를 C 언어 코드로 재정의하여 이를 기반으로 컴파일된 함수를 제공하는 것을 들 수 있다. 해당 기능을 소개하기 위하여 $m \times n$ 행렬의 각 원소 값을 2배하여 반환하는 함수의 예제를 준비하였다. 해당 기능의 사용을 위해서는 numba의 cfunc, types, carray와 ctypes 라이브러리를 이용해야 한다.

예제 2.6: 행렬 원소별 연산 예제

```
from numba import cfunc, types, carray
import numpy as np
from ctypes import *
c_sig = types.void(types.CPointer(types.double), types.CPointer(types.double), types.intc, types.intc)
@cfunc(c_sig)
def callback(in_, out, m, n):
    in_arr = carray(in_, (m,n))
    out_arr = carray(out, (m,n))
    for i in range(m):
        for j in range(n):
            out_arr[i, j] = 2 * in_arr[i, j]
input_data = np.random.random(10).astype(c_double)
output_data = np.zeros(10).astype(c_double)
addr_input = input_data.ctypes.data_as(POINTER(c_double))
addr_output = output_data.ctypes.data_as(POINTER(c_double))
callback.ctypes(addr_input, addr_output, 5, 2)
```

cfunc 데코레이터는 jit 데코레이터와 유사한 방법으로 사용할 수 있지만 차이점이 존재한다. cfunc 데코레이터는 guvectorize처럼 반드시 signature 변수를 입력 인수로 전달해야 한다. signature 변수는 C 언어에서 함수 정의를 할 때 함수의 선언부에 해당하는 부분으로 각 인수들과 인수들의 데이터타입을 types 객체를 이용하여 표현한다. 예를 들어, 위의 예제 2.6의 4행의 c_sig 변수를 정의할 때 numba의 types 클래스를 이용하며, "출력 타입 (인수 타입)" 형태로 정의한다. 함수의 반환 타입으로 void 타입으로 정의한 것을 참고하기 바란다. 포인터 변수의 경우 types.CPointer 멤버함수를 사용하여 해당하는 데이터 타입

Table 2.2. Summary of key decorators in numba

numba 데코레이터	주요 기능
@vectorize	벡터들의 원소별 연산에 대한 함수를 컴파일하는 데코레이터, 입력 인수와 출력 인수의 차원이 동일한 함수만 컴파일 가능, CPU 병렬 처리 및 CUDA 지원
@guvectorize	vectorize 데코레이터와 유사하게 벡터 연산에 대한 함수 컴파일 가능 입력 인수와 출력 인수의 차원이 다른 함수 적용 가능 CPU 병렬 처리 및 CUDA 지원
@cfunc	외부의 C 언어 기반의 라이브러리와 호환되어 호출될 수 있도록 파이썬 함수를 컴파일 해주는 데코레이터 (object mode의 경우 컴파일 불가)
@jit	파이썬 함수를 JIT 컴파일 하는 데코레이터, CPU 병렬 처리 지원
@njit	nopython mode = True인 @jit 데코레이터, CPU 병렬 처리 지원

의 포인터 변수를 만들 수 있다. 이와 같은 방식으로 함수의 매개변수와 반환 타입에 대한 정보를 하나의 signature로 만들어 cfunc 데코레이터의 입력 인수로 사용하는 것이 일반적이다.

지금까지 numba의 주요 데코레이터를 예제와 함께 설명하였다. 이 절에서 살펴본 numba 데코레이터와 각 데코레이터의 주요 기능은 Table 2.2와 같다.

3. 통계 계산 적용 예제

본 절에서는 numba 라이브러리가 효율적으로 사용될 수 있는 두 가지 통계 계산 문제에 대하여 소개하고 이를 순수한 파이썬 구문을 이용하여 구현한 코드와 numba를 이용하여 구현한 코드를 제시하였다. numba를 이용한 구현에서 단순 반복문에 대한 JIT 컴파일 적용의 경우에는 데코레이터만 적용하면 되므로 순수한 파이썬 코드만 제시하고 데코레이터에 대한 코드는 본문에서 서술하였다. 본 절에서 제시한 예제들은 연산의 효율성을 위해 벡터화(vectorization)를 고려하여도 반복문을 피할 수 없는 예제로 구성되어 있으며 통계학 연구자들이 모의 실험을 진행할 경우에 도움이 될 수 있도록 간단한 치환의 반복과 여러 선형대수적 연산들의 반복을 포함하고 있다. 먼저 3.1절에서 이표본 순열검정에 대한 예제를 통하여 간단한 치환 반복의 예제에 대하여 살펴 보고 3.2절에서는 선형대수적 연산이 반복되는 EM 알고리즘 기반의 정규 혼합 모형(Gaussian mixture model) 모수 추정 예제를 살펴 본다. 정규 혼합 모형 예제에서는 단변량(univariate)과 다변량(multivariate)의 경우 구현의 차이가 존재하여 두 가지 경우를 구분하여 구현하였다. 앞서 언급했듯이 numba는 반복문이 많이 사용되는 경우 일반적인 파이썬 함수에 비해 효율적이다. 본 예제에서는 효율성을 수치적으로 보이기 위해 각 구현에 따른 계산 시간을 측정하고 이를 기반으로 구현 방법들을 비교하였다. 계산 시간은 AMD Ryzen 3950X (CPU)와 64 GB RAM의 워크스테이션을 이용하여 측정되었다.

3.1. 이표본 순열검정

가설 검정을 수행하기 위해서는 일반적으로 귀무가설이 참일 때의 검정 통계량의 분포(귀무 분포, null distribution)가 요구된다. 보통 관측 데이터의 분포 가정을 통하여 귀무 분포가 유도되나 데이터가 부족하거나 관측 데이터가 기존의 분포 가정을 만족한다고 볼 수 없는 경우, 귀무 분포의 유도에 어려움이 따른다. 순열 검정(permutation test)은 이러한 상황에서 유용하게 적용할 수 있는 비모수적 가설검정 방법으로 데이터의 순서를 무작위로 바꾸며 검정 통계량을 반복적으로 계산하여 이를 바탕으로 귀무 분포를 추정

Code 1 Estimate the null distribution implemented by naive python

```
import numpy as np
def est_null(x, y, z, iteration = 10000):
    n = x.shape[0]
    null_dist = np.zeros(iteration)
    np.random.seed(42)
    for i in range(iteration):
        temp = np.random.permutation(z)
        temp_x, temp_y = temp[:n], temp[n:]
        null_dist[i] = np.mean(temp_x) - np.mean(temp_y)
    return null_dist
```

하고 가설검정을 수행하는 방법이다 (Pitman, 1937).

본 예제에서는 순열 검정을 이용한 이표본의 평균 검정에 대하여 다루고자 한다. 순열 검정의 절차를 간략히 소개하면, 먼저 관측된 두 표본의 표본 평균의 차이를 검정 통계량으로 정의하고 이를 계산한다. 이후 주어진 데이터로부터 귀무 분포를 추정하기 위해 두 표본의 데이터를 하나로 합하여 순서를 무작위로 치환한 뒤, 첫 번째 표본의 수만큼 앞에서부터 선택하여 새로운 첫 번째 표본으로 만들고 나머지 데이터를 새로운 두 번째 표본으로 정의한다. 새롭게 만들어진 두 표본에 대한 평균의 차로 계산된 검정 통계량을 저장한다. 위 작업을 반복하여 저장된 검정통계량을 기반으로 귀무 분포를 추정하면 원 데이터로부터 계산된 검정 통계량에 대한 유의확률(p-value)을 계산할 수 있다. 보다 상세한 서술을 위하여 먼저 이표본 데이터가 $\mathbf{x} = (x_1, \ldots, x_{n_1})$, $\mathbf{y} = (y_1, \ldots, y_{n_2})$로 관측되었다고 가정하고 두 벡터가 하나로 합쳐진 벡터 $\mathbf{z}$를 아래와 같이 정의한다.

$$\mathbf{z} = (z_1, \ldots, z_{n_1+n_2}) = (\mathbf{x}, \mathbf{y}) = (x_1, \ldots, x_{n_1}, y_1, \ldots, y_{n_2})$$

$\mathbf{z}$ 벡터의 순서를 무작위로 바꾸어 새로운 벡터 $\mathbf{z}^{(i)}$를 정의한 후 원래의 표본의 길이와 동일한 $\mathbf{x}^{(i)}$와 $\mathbf{y}^{(i)}$를 다음과 같이 정의한다.

$$\begin{aligned}
\mathbf{z}^{(i)} &= \left(z_1^{(i)}, z_2^{(i)}, \ldots, z_{n_1+n_2}^{(i)}\right) \\
\mathbf{x}^{(i)} &= \left(z_1^{(i)}, z_2^{(i)}, \ldots, z_{n_1}^{(i)}\right) \\
\mathbf{y}^{(i)} &= \left(z_{n_1+1}^{(i)}, z_{n_1+2}^{(i)}, \ldots, z_{n_1+n_2}^{(i)}\right)
\end{aligned}$$

무작위 치환에 의해 새롭게 정의된 $\mathbf{x}^{(i)}$와 $\mathbf{y}^{(i)}$의 평균의 차를 t_i로 정의하고 위의 절차를 k번 반복하여 $\mathbf{t} = (t_1, t_2, \ldots, t_k)$를 계산한다. 이렇게 계산된 $\mathbf{t}$를 바탕으로 아래와 같이 순열 검정의 p-value를 계산하여 주어진 유의수준 α에 대한 가설 검정을 진행하면 된다.

$$P(|T| > |t_0|) \approx \frac{1}{k}\sum_{i=1}^{k} I(|t_i| > |t_0|)\,,$$

여기서 $t_0 = (1/n_1)\sum_{i=1}^{n_1} x_i - (1/n_2)\sum_{i=1}^{n_2} y_i$, $t_j = (1/n_1)\sum_{i=1}^{n_1} x_i^{(j)} - (1/n_2)\sum_{i=1}^{n_2} y_i^{(j)}$ for $j = 1, \ldots, k$이다.

앞서 상술한 바와 같이 순열 검정을 진행할 경우, 전체 자료를 무작위로 치환하고 나누어 평균차이를 계산하는 절차의 반복을 통하여 귀무 분포를 추정하게 되는데 이 과정에서 반복문의 사용을 피할 수 없다. 본 절에서는 `numba`의 효율성을 보이기 위하여 귀무 분포 추정 절차를 순수한 파이썬(naive python) 구문으로

Table 3.1. Summary of computation times (sec.) for permutation test. Numbers in parenthesis denote the standard errors

n	δ	Rejection	Naive	JIT-wec	JIT-aec	JIT-wlc	JIT-alc	mlxtend
50	0	0.0778	0.1410 (0.0019)	0.0087	0.0084 (0.00006)	0.2234	0.0074 (0.00005)	0.1267 (0.0005)
50	0.25	0.2889	0.1325 (0.0021)	0.0077	0.0076 (0.00003)	0.1277	0.0076 (0.000007)	0.1273 (0.0002)
50	0.5	0.7248	0.1307 (0.0014)	0.0076	0.0076 (0.00002)	0.1354	0.0075 (0.000009)	0.1326 (0.0001)
100	0	0.0726	0.1457 (0.0018)	0.0155	0.0154 (0.00004)	0.1863	0.0148 (0.00003)	0.1411 (0.0004)
100	0.25	0.4638	0.1495 (0.0018)	0.0156	0.0156 (0.0006)	0.1367	0.0152 (0.00005)	0.1396 (0.0001)
100	0.5	0.9461	0.1447 (0.0019)	0.0156	0.0155 (0.00005)	0.1352	0.0151 (0.00002)	0.1380 (0.0004)

구현한 코드와 numba를 활용하여 구현한 코드를 이용하여 계산을 수행하고 계산 시간을 비교하고자 한다. Code 1에서 귀무 분포 추정의 절차를 순수한 파이썬 함수로 구현한 코드를 제시하였다.

Code 1에서 제시한 함수의 절차는 numba의 njit 데코레이터를 이용하여 매우 쉽게 JIT 컴파일을 적용할 수 있다. 자세히 설명하면, JIT 컴파일을 진행하기 위하여 Code 1의 Line 1과 Line 2의 사이에

```
import numba
@numba.njit
```

의 두 줄을 추가하면 된다. 만약 2절에서 설명한 eager compilation을 적용하고자 할 경우 njit 데코레이터의 인수를

```
@numba.njit("float64[:](float64[:],float64[:],float64[:],int64)")
```

와 같이 추가하여 데이터 타입을 명시하면 된다. 데이터 타입을 명시하기 위해서는 필요로 하는 데이터 타입을 numba로 부터 import하여야 한다. 배정밀도가 아닌 단정밀도 데이터 타입을 사용하고 싶으면 float64 대신 float32로 바꾸어 사용할 수도 있다. 다만, 위의 구현에서 JIT 컴파일 적용을 위해 한 가지 기억해야하는 부분이 있다. Code 1에서 **z** 벡터를 내부에서 정의한 것이 아니라 외부에서 입력 변수로 전달 받은 점이다. 보통 단순히 x와 y를 결합하는 연산은 함수 내부에서 진행하지만 numba.njit에서는 numpy.r_[ ] 함수에 대한 JIT 컴파일을 지원하지 않는다. 따라서 구현 상 조금 번거롭더라도 함수의 외부에서 전처리를 한 후 입력하여야 한다.

Table 3.1은 이표본의 각 표본의 수를 동일하게 가정하였을 때 ($n = n_1 = n_2$), 이표본의 평균의 차(δ)에 따라 순수한 파이썬과 numba를 이용한 두 구현 코드의 속도를 비교한 결과이다. 결과는 초(second) 단위로 측정하였으며 귀무가설의 분포를 추정하기 위한 무작위 치환 횟수는 10,000번으로 설정하였다. Table 3.1에서 Rejection은 귀무가설을 기각한 비율을 나타내며, $\delta = 0$일 때는 검정의 크기(size of test), $\delta \neq 0$인 경우는 검정력(power of test)을 의미한다. Naive는 순수한 파이썬 코드로 구현한 함수이며, JIT-wec (JIT with eager compilation)와 JIT-aec (JIT after eager compilation)는 각각 eager compilation을 포함한 실행 시간과 컴파일을 마친 후의 함수의 실행 시간 (컴파일에 필요한 시간 제외)을 의미한다. JIT-wlc (JIT with lazy compilation)와 JIT-alc (JIT after lazy compilation)는 앞에서 JIT-wec와 JIT-aec에서 eager compilation 대신 lazy compilation을 사용했을 경우이다. 마지막으로 mlxtend는 파이썬의 외부 라이브러리인 mlxtend 의 순열 검정 방법 함수의 실행 시간을 의미한다. 함수 실행 시간의 변동성을 고려하여 Naive와 mlxtend는 각각

10번, JIT-aec와 JIT-alc는 각각 9번을 반복적으로 측정하여 평균을 낸 결과이다. JIT-aec와 JIT-alc는 9번 반복 측정한 것은 JIT 컴파일의 경우 함수의 첫 실행에서 코드를 해석하여 기계어로 바꾼뒤 실행파일로 만드는 컴파일 과정의 시간이 포함되어 있어 이를 JIT-wec와 JIT-wlc로 분리하였기 때문이다. 결과를 살펴보면, 순수한 파이썬 코드로 만든 함수의 경우 0.1307초($n = 50, \delta = 0.5$)인 반면 eager compilation을 사용하여 컴파일된 함수의 실행시간을 의미하는 JIT-aec는 0.0076초($n = 50, \delta = 0.5$)이다. Naive의 경우 JIT-aec에 비해 시간이 약 17배 정도 걸리는 것을 확인할 수 있다. 또한 JIT 컴파일을 사용한 경우 `mlxtend` 라이브러리에 비해서도 빠른 결과를 보인다. 추가적으로 eager compilation을 사용할 때와 lazy compilation을 사용할 때의 컴파일 시간은 위와 동일한 조건 하에서 0.0087초, 0.2234초로 데이터의 타입을 추론해야 하는 lazy compilation의 경우 데이터의 타입을 명시하는 eager compilation 방법에 비해 계산시간이 약 26배 더 소요되는 것으로 보인다. lazy compilation은 구현의 편의성과 데이터 타입에 대한 유연성을 지니고 있지만, 데이터의 타입을 정확히 명시할 수 있는 상황이라면 eager compilation을 사용하는 것이 보다 효율적이다.

3.2. EM 알고리즘

Expectation-Maximization (EM) (Dempster 등, 1977) 알고리즘은 고려하는 모형에 관찰되지 않은 잠재변수가 있다고 가정하는 상황에서 반복법(iterative method)으로 가능도함수를 최대화하는 모수를 찾는 방법 중 하나로 다양한 모형에 적용할 수 있다. 본 절에서는 모형 기반의 군집 분석에 활용되는 정규 혼합 모형(Gaussian mixture model)의 가정 하에서 EM 알고리즘에 대하여 설명하고 numba를 이용한 구현에 대하여 살펴보고자 한다. 먼저 혼합 모형에 대하여 살펴 보기 위하여, $x_i \in \mathbb{R}^p$는 관측 데이터, $Z_i \in \{1,2,\dots,K\}$는 관측되지 않은 군집 구분자(cluster index), $\tau_k = \Pr(Z = k)$는 k번째 군집의 비율에 대한 사전 분포, $\Theta = \{(\theta_k, \tau_k), k = 1,2,\dots,K\}$, θ_k는 k번째 군집에 대한 모수로 표현한다. EM 알고리즘은 먼저 관측되지 않은 군집 구분자가 관측되었다고 가정한 완전 자료(complete data) 하에서의 완전 가능도 함수(complete likelihood function)을 정의한다. n개의 표본이 K개의 서로 다른 분포에서 생성 되었을 때, 완전 가능도 함수는 아래와 같이 표현할 수 있다.

$$CL(\Theta; x_1, z_1, \dots, x_n, z_n) = \prod_{i=1}^{n} \prod_{k=1}^{K} (\tau_k f_{X|Z}(x_i|Z_i = k))^{I(Z_i=k)}. \tag{3.1}$$

특히 데이터가 정규분포에서 생성되었다고 가정했을 때의 혼합 모형을 정규 혼합 모형이라 부른다. 정규 혼합 모형에 대한 로그 완전 가능도 함수를 나타내면 다음과 같다.

$$\log CL(\Theta; x_1, z_1, \dots, x_n, z_n) = \sum_{i=1}^{n} \sum_{k=1}^{K} I(Z_i = k) \left\{ \log\ \tau_k - \frac{1}{2} \log|2\pi\Sigma_k| - \frac{1}{2}(x_i - \mu_k)^T \Sigma_k^{-1} (x_i - \mu_k) \right\}.$$

EM 알고리즘의 절차는 크게 Expectation (E)과 Maximization (M)의 두 단계로 이루어져있으며, E-step에서는 관측된 데이터와 현재 모수 추정량이 주어졌을 때의 완전 가능도 함수 (또는 로그 완전 가능도 함수)의 기댓값을 계산하고 M-step에서는 E-step에서 계산된 완전 로그 가능도 함수의 기댓값을 최대화하는 모수를 찾는다. EM 알고리즘은 E-step과 M-step을 추정하고자 하는 모수 Θ가 수렴할 때까지 반복한다. 정규 혼합 모형의 모수 추정의 예제를 보다 자세히 살펴 보면, 현재 주어진 모수 추정량을 $\theta^{(0)}$라 할 때, E-step은 아래와 같이 로그 완전 가능도 함수에 대한 기댓값을 계산한다.

$$E_{\theta^{(0)}}\left(\log CL(\Theta)\right)|x_1,\ldots,x_n)=\sum_{i=1}^{n}\sum_{k=1}^{K}Q_{ik}^{(0)}\left\{\log\tau_k-\frac{1}{2}\log|2\pi\Sigma_k|-\frac{1}{2}(x_i-\mu_k)^T\Sigma_k^{-1}(x_i-\mu_k)\right\}, \tag{3.2}$$

여기서 $Q_{ik}^{(0)}=P_{\theta^{(0)}}(Z_i=k|x_1,\ldots,x_n)$이다.

M-step에서는 식 (3.2)를 최대화하는 Q,μ,Σ,τ를 구한다. 각 모수는 1차 미분을 기준으로 유도된 아래의 식을 통해 갱신한다.

$$\begin{aligned}
Q_{ik}^{(n)} &= \frac{f\left(x_i,z_i=k|\theta_k^{(n)}\right)}{f_X\left(x_i|\theta_k^{(n)}\right)},\\
\mu_k^{(n+1)} &= \frac{\sum_{i=1}^{n}Q_{ik}^{(n)}x_i}{\sum_{i=1}^{(n)}Q_{ik}^{(n)}},\\
\Sigma_k^{(n+1)} &= \frac{1}{\sum_{i=1}^{n}Q_{ik}^{(n)}}\sum_{i=1}^{n}Q_{ik}^{(n)}\left(x_i-\mu_k^{(n+1)}\right)\left(x_i-\mu_k^{(n+1)}\right)^T,\\
\tau_k^{(n+1)} &= \frac{1}{n}\sum_{i=1}^{n}Q_{ik}^{(n)}.
\end{aligned}$$

위 식들을 바탕으로 하여 E-step과 M-step을 반복하면서 수렴조건이 만족할 때까지 반복추정한다. 본 논문에서는 EM 알고리즘의 수렴 조건으로 $\|\Theta^{(k+1)}-\Theta^{(k)}\|_\infty<\epsilon$을 적용하였다. 여기서 $\|x\|_\infty=\max_i x_i$로 L_∞-노음(norm)을 의미하며 행렬 형식의 모수는 벡터화(vectorization)한 후 수렴 조건을 계산하였다 (즉, 원소별 차이 최댓값을 수렴 조건에 적용).

Code 2와 Code 3는 각각 단변량 정규분포와 다변량 정규분포에 대한 혼합 모형의 모수 추정에 대한 EM 알고리즘을 순수한 파이썬 구문을 이용하여 구현한 코드이다. Code 2와 Code 3의 처음 부분에 정규 분포의 밀도 함수 계산을 위한 파이썬 함수를 정의하였는데 외부 라이브러리에서 제공하는 함수를 이용할 수 있으나 본 논문에서는 JIT 컴파일을 원활하게 진행하기 위하여 직접 함수를 구현하였다. 단변량 정규분포에 해당하는 Code 2의 경우에는 단순히 `njit` 데코레이터를 함수 정의 바로 위에 추가하여 쉽게 JIT 컴파일을 진행 할 수 있다. 보다 자세히 설명하면, Code 2의 Line 1과 Line 3의 위에 `@njit` 코드를 추가하면 JIT with lazy compilation을 바로 적용할 수 있다. 컴파일의 속도를 향상 시키기 위해서는 아래와 같이 코드를 추가하여 JIT with eager compilation을 진행할 수 있다.

Line 1의 위:

```
@njit("float64[:](float64[:],float64,float64)")
```

Line 3의 위:

```
r_sig = types.Tuple([float64[:],float64[:],float64[:],int64])
sig = r_sig(float64[:],float64[:],float64[:],int64,float64[:],float64[:,:],float64)

@njit(sig)
```

위의 코드에서 `r_sig`는 함수 `GMM_EM()`의 반환 변수 4개에 해당하는 데이터 타입을 하나의 튜플 형태로 만드는 코드이다. `sig` 변수는 반환하는 변수에 대한 데이터 타입과 입력 인수들의 데이터 타입을 묶어 하나의 signature로 만들어 준다. 반환하는 변수들과 입력 인수들이 많은 경우에는 위와 같이 하나의 변수로 만

Code 2 Naive python for the univariate mixture

```
1:  def normal_ll(X, mu, sigma):
2:      return np.exp(-(X-mu)**2 / (2*sigma)) / np.sqrt(2*np.pi*sigma)
3:  def GMM_EM(X,mu,sigma,max_iter,tau,q,tol = 1e-15):
4:      K = len(mu)
5:      n = len(X)
6:      for iteration in range(max_iter):
7:          for k in range(K):
8:              ll = normal_ll(X,mu[k],sigma[k])
9:              q[:, k] = tau[k] * ll
10:         for i in range(n):
11:             q[i,:] /= np.sum(q[i,:])
12:         mu_before = mu
13:         sigma_before = sigma
14:         tau_before = tau
15:         for k in range(K):
16:             q_k = np.sum(q[:,k])
17:             mu[k] = np.sum(q[:, k] * X) / q_k
18:             sigma[k] = np.sum(q[:, k] * (X-mu[k])**2) / q_k
19:             tau[k] = q_k / n
20:         mu_diff = np.max(np.abs(mu - mu_before))
21:         sigma_diff = np.max(np.abs(sigma - sigma_before))
22:         tau_diff = np.max(np.abs(tau - tau_before))
23:         diff = np.max(np.array([np.abs(mu_diff), np.abs(sigma_diff), np.abs(tau_diff)]))
24:         if ( (iteration > 1) & (diff < tol) ): break
25:     return mu, sigma, tau, iteration
```

들어 사용하는 것이 가독성 및 유지 보수에 이점이 있다. 이렇게 정의한 signature 변수를 `@njit(sig)`와 같이 데코레이터의 인수로 컴파일러에게 전달하여 데이터 타입을 추론하는 시간을 줄일 수 있다.

다변량 정규분포의 경우, 데이터의 차원이 벡터에서 행렬로 바뀌어 단순히 데코레이터만을 추가하여 컴파일을 할 경우 문제가 발생할 수 있다. `numba`의 JIT 컴파일은 배열의 `shape` 정의에 명확한 표현을 요구하는데 `numpy` 라이브러리의 1차원 벡터 타입은 이와 호환되지 않아 오류가 발생한다. 조금 더 정확히 표현하면 `numpy` 라이브러리는 1차원 벡터의 shape을 임의의 차원 n에 대하여 "`(n,)`" 형태 또는 "`(n,1)`" 형태로 정의할 수 있는데 이 중 전자의 형태는 `numba`의 JIT 컴파일에서 사용할 수 없다. 따라서 벡터의 차원을 재정의하는 작업이 추가적으로 필요하다. 벡터의 차원을 재정의하는 방법 중 하나로 `numpy.ascontiguousarray` 함수를 이용할 수 있다. 이 함수를 이용하여 Code 3의 21행과 22행 사이에 `q1 = np.ascontiguousarray(q[:,k])`를 추가하고 22–23행의 `q[:,k]`를 `q1`으로 치환하면 `njit` 데코레이터를 이용하여 JIT 컴파일 할 수 있다. 따라서 다변량 정규분포의 경우에는 데코레이터를 사용하는 것 이외에 차원에 대한 호환성을 확인하고 차원을 재정의하는 코드만 추가하면 JIT 컴파일을 사용할 수 있다. 전체 예제에 대하여 구현된 코드는 저자의 깃허브(https://github.com/YounsangCho/Numba)에서 확인할 수 있다. Code 2와 Code 3에서의 모수의 초깃값은 임의로 생성한 데이터의 근방의 값으로 하였으며, 초기화 방법은 Cho (2018)의 방법을 적용할 수도 있다.

Code 2와 Code 3를 기반으로 구현된 알고리즘의 효율성을 비교하기 위하여 순열 검정의 예제와 동일하게 Naive, JIT-wec, JIT-aec, JIT-wlc, JIT-alc를 고려하여 계산 시간을 비교하였다. 정규 혼합 모형의 추정에 적용할 수 있는 `scikit-learn`의 `GaussianMixture` 함수를 고려하여 기존에 구현된 라이브러리의 함수와도

Code 3 Naive python for the multivariate mixture

```
def multi_ll(X, mu, cov):
    n = X.shape[0]
    res = np.zeros(n)
    for i in range(n):
        exp_inter = np.dot(np.dot( (X[i,:] - mu).T, np.linalg.inv(cov) ), (X[i, :] - mu) / 2.)
        res[i] = (2*np.pi)**(-p/2) * np.linalg.det(cov)**(-0.5) * np.exp(-exp_inter)
    return res
def GMM_EM_multi(X,mu,cov,max_iter,tau,q,tol = 1e-08):
    n = X.shape[0]
    K = mu.shape[0]
    for iteration in range(max_iter):
        for k in range(K):
            ll = mulit_ll(X,mu[k,:],cov[:,:,k])
            q[:, k] = tau[k] * ll
        for i in range(n):
            q[i,:] /= np.sum(q[i,:])
        mu_before = mu
        cov_before = cov
        tau_before = tau
        for k in range(K):
            q_k = np.sum(q[:,k])
            mu[k,:] = np.sum(q[:, k].reshape(n,1) * X, axis = 0) / q_k
            cov[:,:,k] = np.dot((q[:,k].reshape(n,1) * (X-mu[k,:])).T, (X-mu[k,:])) / q_k
            tau[k] = q_k / n
        mu_diff = np.max(np.abs(mu - mu_before))
        sigma_diff = np.max(np.abs(cov - cov_before))
        tau_diff = np.max(np.abs(tau - tau_before))
        diff = np.max(np.array([np.abs(mu_diff), np.abs(sigma_diff), np.abs(tau_diff)]))
        if ( (iteration > 1) & (diff < tol) ): break
    return mu, cov, tau, iteration
```

계산 시간 비교를 진행하였다. 단변량의 경우, 정규 혼합 모형은 3개의 군집($K = 3$)을 가정하고 각 모수는 아래와 같이 고려하였다.

$$(\mu_1, \sigma_1^2) = (20, 3.1^2), \quad (\mu_2, \sigma_2^2) = (3, 2.3^2), \quad (\mu_3, \sigma_3^2) = (-5, 1.4^2),$$
$$(\tau_1, \tau_2, \tau_3) = \left(\frac{1}{3}, \frac{1}{3}, \frac{1}{3}\right).$$

다변량의 경우, 정규 혼합 모형은 이변량 정규 분포를 따르는 2개의 군집($K = 2$)을 가정하고 각 모수는 아래와 같이 설정하였다.

$$\mu_1 = (1.5, 2.5), \quad \mu_2 = (7.3, 10.2),$$
$$\Sigma_1 = \begin{pmatrix} 1.2 & 0.4 \\ 0.4 & 1.1 \end{pmatrix}, \quad \Sigma_2 = \begin{pmatrix} 1.5 & 0.5 \\ 0.5 & 2.1 \end{pmatrix},$$
$$(\tau_1, \tau_2) = \left(\frac{1}{2}, \frac{1}{2}\right).$$

계산 속도 비교를 위하여 표본 크기(n)를 600, 30,000으로 고려하여 구현한 각 알고리즘의 계산 시간을 측정하였다. EM 알고리즘의 경우, 생성된 표본에 따라 수렴까지의 반복수가 변하므로 보다 명확한 비교를 위하여 시간 측정을 위한 반복 계산에서 난수 생성 seed 번호를 고정한 Fixed와 표본을 반복마다 다르게

Table 3.2. Summary of computation times (ms) for EM algorithm. Numbers in parenthesis denote the standard errors.

Type	Seed	Size	K	Naive (ms)	JIT-wec (ms)	JIT-aec (ms)	JIT-wlc (ms)	JIT-alc (ms)	Scikit learn (ms)
Uni	Fixed	600	3	10.4611 (0.2250)	0.3443	0.1509 (0.0003)	872.1120	0.1515 (0.0005)	3.2307 (0.0982)
Uni	Changed	600	3	9.9916 (0.0452)	0.1602	0.1510 (0.0002)	858.0930	0.1504 (0.0004)	3.2692 (0.0413)
Uni	Fixed	30000	3	460.8467 (2.8338)	6.7878	6.9091 (0.1097)	840.2407	6.8434 (0.1134)	326.0081 (11.7890)
Uni	Changed	30000	3	454.6624 (0.7028)	6.7961	6.8017 (0.0019)	838.5603	6.7479 (0.7126)	354.0168 (4.2034)
Multi	Fixed	600	2	75.3585 (0.5279)	4.9205	4.6459 (0.0050)	1970.5291	4.4564 (0.0030)	3.3439 (0.2363)
Multi	Changed	600	2	74.9992 (0.0469)	4.6721	4.3353 (0.0141)	1949.0967	4.2113 (0.0051)	2.7690 (0.0491)
Multi	Fixed	30000	2	3830.0113 (7.7058)	227.1440	226.7110 (0.1644)	2247.5245	226.6806 (0.1467)	140.5398 (3.9624)
Multi	Changed	30000	2	3711.7020 (11.7747)	211.6017	209.6154 (0.0956)	2141.0129	209.1743 (0.0797)	142.0074 (1.9507)

생성한 Changed로 구분하여 계산 시간을 측정하였다. 각 경우마다 10회씩 계산을 반복하고 평균 계산 시간 및 표준 오차에 대하여 Table 3.2에 정리하였다. Nopython mode의 JIT 컴파일은 Table 3.1과 마찬가지 방식으로 함수의 첫 실행에 컴파일을 포함한 시간과 나머지 실행에 대한 부분을 나누어 정리하였다. 추가적으로 Table 3.2의 경우 실행시간의 단위는 밀리세컨드(ms) 를 사용하였다. 결과를 살펴보면, 데이터를 단변량 정규분포에서 생성했을 경우 순수한 파이썬 구문으로 구현한 코드는 `njit` 데코레이터를 사용한 경우에 비해 최대 약 69배 정도의 시간이 소비된다. `scikit-learn`의 `GaussianMixture` 함수의 경우 `numba`에 비해 최대 약 52배까지 시간이 걸리는 것을 확인할 수 있다. 다변량 정규분포의 경우에는 순수한 파이썬 코드에 비해 계산 속도가 약 18배 정도 빠르지만, `scikit-learn`과 비교하여서는 오히려 느려지는 경향이 있다. 이는 본 논문에서 구현한 알고리즘은 다변량 정규분포의 공분산 행렬을 사용하지만, `GaussianMixture` 함수의 내부에서는 공분산 행렬의 역행렬인 정밀 행렬(precision matrix)을 이용하여 계산함에 따르는 차이라고 생각된다.

4. 결론

본 논문에서는 파이썬의 함수를 저수준의 언어로 JIT 컴파일하는 라이브러리인 `numba`의 사용법에 대하여 소개하고 JIT 컴파일 적용의 효율성에 대하여 수치적으로 살펴보았다. 본 논문에서 다루지는 않았지만 `numba`에서는 GPU 병렬 연산을 수행하는 CUDA와의 호환도 가능하다. 하지만 GPU에서 CPU로의 메모리 복사과정이 원활하지 않아 현재까지는 `pycuda`와 같이 GPU 연산을 위한 라이브러리에 비해 효율적이지 못한 결과를 보인다. `numba`는 많은 수의 반복문을 실행해야 되는 함수를 직접 만들어야 하는 경우 간단히 데코레이터만을 추가함으로써 일반적인 파이썬 함수를 효율적으로 변환할 수 있다. 만약 파이썬에 익숙한 독자들이라면, 위와 같은 상황을 마주했을 때 `numba`를 사용해 보는 것을 추천한다. 특히 `numba`의 효율성은 반복문이 자주 사용되는 파이썬 구문에 적용할 때 높아진다. 예를 들어 차원의 크기가 큰 행렬의 곱 연산과 같이 한 번의 연산 자체가 시간이 걸리는 함수보다는 본 논문의 예제들과 같은 간단한 연산에 반복문을 많이 사용해야 하는 함수들의 구현에 있어 효율적이다. 또한 파이썬의 라이브러리 중 하나인 `numpy`와 호환이 좋기 때문에, 함수를 구성할 때 `numpy`의 함수나 파이썬의 기본 라이브러리 함수를 쓰는

것을 추천한다.

하지만 코드 구현 시 상기해야 하는 부분이 있다. numba는 파이썬의 기본 함수에서 오류 발생 시 제공하는 오류 문구와 다르게 오류에 대한 부분을 상세히 설명해주지 않기 때문에, 함수를 만드는 작업에 있어 많은 디버깅이 요구될 수 있다. 또한 numba는 수치 연산을 효율적으로 하기 위한 라이브러리로 pandas와 같이 데이터프레임을 다루는 고수준의 라이브러리와는 호환이 되지 않는다. 만약 C 또는 C++와 같은 저수준의 언어에 익숙한 독자들이 파이썬을 통해 함수를 최적화하기를 원한다면 numba도 쉽게 익숙해질 수 있지만, 보다 유연한 구현과 실행을 위해서는 본 논문의 서론에서 언급한 파이썬과 C를 연동하는 방법을 사용하기를 권장한다.

References

Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D. S., and Smith, K. (2010). Cython: the best of both worlds, *Computing in Science & Engineering*, **13**, 31–39.

Cho, H. (2018). Initalizing method of finite mixture model using kernel density estimation and application on model-based clustering, *Journal of the Korean Data & Information Science Society*, **29**, 327–338.

Dempster, A. P., Laird, N. M., and Rubin, D. B. (1977). Maximum likelihood from incomplete data via the EM algorithm, *Journal of the Royal Statistical Society: Series B*, **39**, 1–38.

Eddelbuettel, D. and Francosis, R. (2011). Rcpp: Seamless R and C++ Integration, *Journal of Statistical Software*, **40**, 1–18.

Lam, S. K., Pitrou, A., and Seibert, S. (2015). Numba: A LLVM-based Python JIT compiler, *Proc. 2nd Workshop LLVM Compiler Infrastructure HPC*, **7**, 1–6.

Lees, J. A., Harris, S. R., Tonkin-Hill, G., Gladstone, R. A., Lo, S. W., Weiser, J. N., Corander, J., Bentley, S. D., and Croucher, N. J. (2019). Fast and flexible bacterial genomic epidemiology with PopPUNK. *Genome Research*, **29**, 304–316.

McInnes, L., Healy, J., Saul, N., and Großberger, L. (2018). UMAP: Uniform Manifold Approximation and Projection. *Journal of Open Source Software*, **3**, 861.

Pitman, E. J. G. (1937). Significance tests which may be applied to samples from any populations, *Journal of the Royal Statistical Society*, **4**, 119–130.

Stone, J. E., Gohara, D., and Shi, G. (2010). OpenCL: a parallel programming standard for heterogeneous computing systems, *Computing in Science & Engineering*, **12**, 66–73.

Tibshirani, R. (1996). Regression shrinkage and selection via the lasso, *Journal of the Royal Statistical Society: Seires B*, **58**, 267–288.

효율적인 통계 계산을 위한 파이썬 numba 라이브러리의 소개

조윤상[a] · 유동현[a] · 손 원[b,1] · 박선철[c]

[a]인하대학교 통계학과, [b]단국대학교 정보통계학과,
[c]Pacific Climate Impacts Consortium, University of Victoria

(2020년 9월 22일 접수, 2020년 10월 15일 수정, 2020년 10월 16일 채택)

요 약

본 논문은 순수하게 파이썬 언어로 작성된 연산에 대하여 just-in-time (JIT) 컴파일을 적용하여 전체 계산 속도를 향상시킬 수 있는 numba 라이브러리에 대한 사용법과 응용에 대하여 소개한다. 실제 통계 계산 문제에 대한 numba 라이브러리의 적용에 대한 예제로 반복문 사용이 요구되는 통계 계산 문제들 중 순열 검정과 정규 혼합 분포의 모수 추정의 EM 알고리즘을 고려하였으며 순수한 파이썬 구문 및 반복문을 활용한 계산 시간과 numba를 활용한 계산 시간을 비교하여 numba 라이브러리 활용의 효율성을 수치적으로 제시하였다.

주요용어: 통계 계산, 파이썬, numba, JIT 컴파일

이 성과는 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임 (No. NRF-2018R1C1B 6001108).

[1] 교신저자: (16890) 경기도 용인시 수지구 죽전로 152, 단국대학교 정보통계학과. E-mail: son.won@dankook.ac.kr