인공지능 과제 리포트

과제 제목:KNN_MNIST

학번: B611092
이름: 서영진

1. 과제 개요

K-nearest-neighbor 알고리즘을 통해 mnist handwriting 데이터셋을 기반으로 test handwriting 데이터셋의 결과를 예측하는 과제

2. 구현 환경

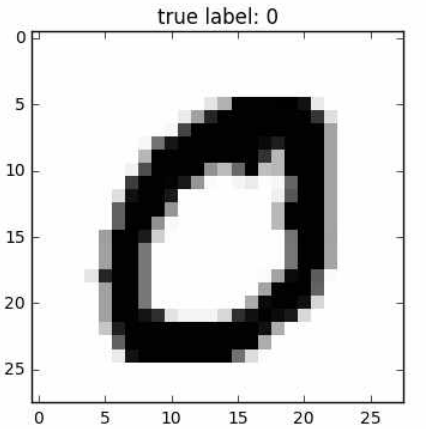
하드웨어 및 OS
Device :LG Electronics / 15ZD990-VX50K
CPU :Intel® Core™ i5-8265U @ 1.60Hz
OS : Microsoft Windows 10 Home
Integrated Development Environment : Pycharm

3. 알고리즘에 대한 설명

K-Neaeat-Neighbor algorithm (최근접 이웃 알고리즘)은 머신러닝에서 분류기법에 사용되는 알고리즘이다
지난 보고서 1에서 다루었던 KNN 과 같은 방식으로 추론해주면 된다
다만 지난 과제에서는 iris data 에 대하여 적용을 하였고
이번 과제의 대상은 Mnist_data 이다

Mnist_data 의 정의

Mnist dataset 에는 60,000개의 이미지 학습 세트와 10,000개의 이미지 테스트 세트로 나누어진 70,000개의 손글씨 숫자이미지가 포함되어있다
데이터 세트 작업의 용이성으로 인해 machine learning test를위한 고전적인 데이터 세트라고 한다.
이미지 자체는 단일 파일에 저장된 28x28 픽셀 이미지이다.



(Mnist data 의 예시)
28x28의 크기로 표현되고 실제값이 label 로주어져있다.

이번 과제에서는
1.feature 이 무엇인지
2.distance 를 어떻게 구할건지
3.계산시간을 어떻게 단축할것인지.
4.어떻게 feature 를 가공할 것인지.
에 대하여 분석을 해야할 필요가 있다.

지난 iris 과제와 mnist_handwriting 과제는 다음과 같은 차이점이 있다.

	Input feature data type	Input feature dimension	Training data size	Test data size
Iris	길이 (cm)	4	150 중 일부 (예: 140)	150 중 일부 (예: 10)
Mnist	이미지	784 (28x28)	60,000	10,000

Mnist 는 feature 가 이미지이고 차원도 매우많다. 또한 datasize 의 양도 만 단위로 오른 것을 확인할수 있다.

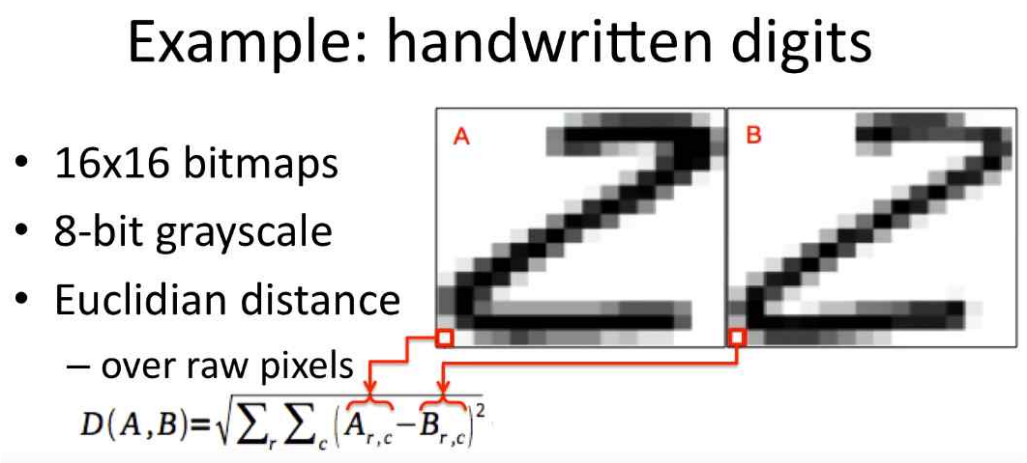
이번 과제의 feature 와 distance 계산방법

I).feature

```
(x_train, t_train), (x_test, t_test) = load_mnist(flatten=True, normalize=False)
```

x_train 에는 길이가 6만개의 배열이 있다.
배열의 원소도 배열인 2차원 배열의 구조이다.
각 원소에 해당하는 배열의 길이는 784의 0 부터 255의 값을 가지는 숫자가 담겨 있다.
우리가 사용하게될 데이터 자체는 손으로 쓰여진 글씨 이지만
784개(28 x 28)의 점으로 표현을 하고 모든 데이터들이 같은 784개의 점을 가지고 있는 데이터라면 다음과 같이 거리를 계산할수 있을 것이다.

II).distance 계산 방법



(출처 https://www.youtube.com/watch?v=ZD_tfNpKzHY)

현재 테스트해야되는 A라는 이미지가 있고 trainset 중 하나로 존재하는 B라는 이미지가 있을 경우
A의 i 번째 index 와 B의 i 번째 index 의 값을 빼준다음 제곱을 해준다
여기서 제곱을 해주는 이유는
음수차이와 양수차이를 모두 반영하기 위함이다.
여기서 i를 1부터 784번째 까지 모두 계산을 해준다음 그 합을 계산한다.
모든 제곱차의 합에 제곱근을 씌어주면 A 와 B 의 거리로 이용해줄수 있다.

III).계산시간
데이터셋의 길이가 6만개나 되기 때문에 하나의 test데이터에 대하여
60000 x 784 의 연산이 필요하다
모든 test데이터에 대해서 수행을 하려면 60000 x 784 x 10000 = 470,400,000,000 의 계산이 필요하게 된다

거리를 계산하고 새로운 배열을 만드는 모든연산은
배열의 모든원소의 접근과 추가를 반복적으로 수행하게 된다.
연산횟수가 매우 크므로 같은 계산이여도 시간이 작은 메소드를 사용해야 좋을 것이다.

```
In [2]: py_list = [i for i in range(10000)]
        start = process_time()
        py_list = [i+2 for i in py_list]
        end = process_time()
        round(end-start,5)
Out[2]: 0.00124
```

```
In [7]: np_arr = np.array([i for i in range(10000)])
        start = process_time()
        np_arr += 2
        end = process_time()
        round(end-start,5)
Out[7]: 0.00025
```

위의 자료를 보면 같은 연산이지만 파이썬과 넘파이의 연산차이는 뚜렷하게 나오는 것을 확인할수 있다.

게다가 우리가 하려는 모든연산은 리스트에 대한 순회연산이므로
넘파이가 제공하는 함수,메소드나 broadcast 성질을 적극적으로 사용해야될 필요가 있다.
또한 같은 양의 데이터여도 넘파이를 사용할 경우 메모리상으로 이점을 얻을수 있다.

IV)**feature 를 적절하게 변형**

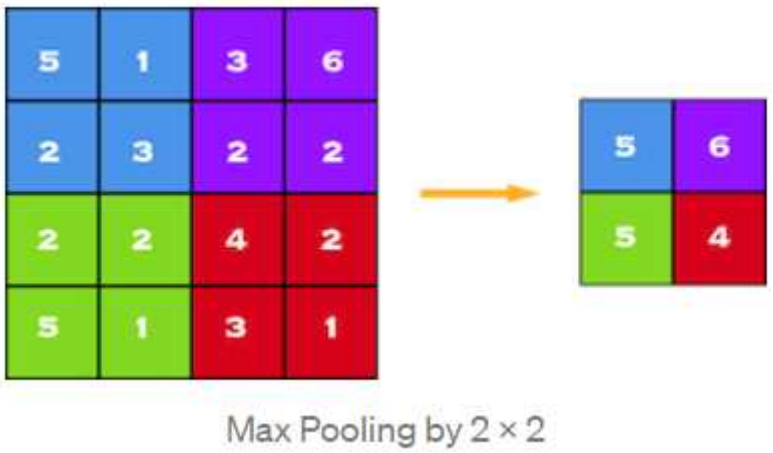
모든 데이터셋의 거리를 그대로 구하는 방식의 문제점.
1.결과를 얻는데 시간이 오래 걸린다.
대부분의 계산이 거리를 구하는데 대부분을 보내게 된다.
28 * 28 (784)개의 모든 데이터들을 비교한다면 시간이 오래 걸릴 것이다.
2.정확하게 픽셀단위로 거리를 비교하는게 최선의방법이 아닐수 있다.
숫자를 그릴 때 정확하게 같은 픽셀단위의 공간에 똑같이 표현될 수는 없을 것이다
하지만 위의 방식은 완전 똑같이 그리는게 아니라면 패널티를 부여하는 방식이기 때문에 인접한 공간과의 데이터는 관대하게 해석해야할 필요가 있다.

Max_polling 이라는 것을 이용하여 공간크기를 줄이고 계산복잡성을 줄이는 기법을 적용할 것이다.

Max_pooling 이란?
최대 풀링 또는 최대 풀링은 각 기능 맵의 각 패치에서 최대 또는 최대 값을 계산하는 풀링 작업입니다.

결과는 평균 풀링의 경우 기능의 평균 존재가 아니라 패치에서 가장 현재 기능을 강조하는 다운 샘플링 또는 풀링 된 기능 맵입니다. 이것은 이미지 분류와 같은 컴퓨터 비전 작업에 대한 평균 풀링보다 실제로 더 잘 작동하는 것으로 밝혀졌습니다.

요컨대, 그 이유는 피처가 피처 맵의 다른 타일 (따라서 피처 맵이라는 용어)에 대해 일부 패턴 또는 개념의 공간적 존재를 인코딩하는 경향이 있으며, 다른 피처의 최대 존재를 보는 것이 평균적인 존재보다 더 유익하기 때문입니다.



4. 데이터에 대한 설명

4.1 Input Feature

input feature의 차수, data type, 값의 의미 등

```
(x_train, t_train), (x_test, t_test) = load_mnist(flatten=True, normalize=False)
```

우리가 사용해야될 input data 들은 load_mnist 를 통하여 얻어 올수 있다.
Load_mnist 함수를 사용할때는 두가지 옵션을 사용한다.
Flatten = True
설정은 모든 데이터들을 1차원으로 받아오겠다는 설정이다.
거리를 계산하기 위해서 2차원배열(28 x 28) 로 바꿔줄 필요가 없다는 점을 이용하여 선형의 자료구조로 받으면 계산과 구현상의 이득을 볼수 있을 것이다.

Normalize = False

이미지안에서 픽셀의 진하기에 해당하는 데이터(748개의 숫자)를 0 부터 1사이의 실수값으로 받는 것이 아닌
0 부터 255까지의 (np.dtype = uint8)으로 표현된 데이터를 가져온다.

2개의 배열이 학습에 영향을 준다.
x_train : training data feature
t_train : training data label

X_train은 길이가 60000인 2차원 배열이다 각 원소에 해당하는 1차원 배열들은
길이가 748이며 모든 원소가 0 부터 255까지로 표현된다

T_train 은 길이가 60000이며 실제 이미지가 나타내는 숫자 (정답) 을 가지고 있다
이러한 데이터들을 lable 이라 부르기도 한다.

2개의 배열이 테스트에 이용이 된다.
x_test 는 test data feature
t_test 는 test data label

X_test 배열은 x_train 배열과 같은 구조이며 길이가 60000이 아닌 10000이라는것만 다르다.
t_test 배열도 t_train 배열과 같은 구조이며 길이가 60000이 아닌 10000이라는것만 다르다.

KNN 클래스를 이용하여 input feature , input lable , 현재 test_set 을 넘겨주는 방식으로 초기 train_set 에 대한 정보를 저장한다.

4.2 Target Output

KNN클래스의 weighted_majority_vote() 메소드를 사용하면 0부터 9사이의 가장 가중치가 높은 숫자를 반환한다.

5. 소스코드에 대한 설명

주석은 따로 소스코드 안에 작성

여기에서는 소스코드 중 중요한 부분에 대한 추가적인 설명 추가

KNN 클래스 내부 구현

```
class KNN:
    def __init__(self, feature, targets, target_names, k):
        self.feature = feature # 훈련에 사용되는 여러 비트맵의 특징들이 담겨있는 2차원 넘파이 배열
        self.targets = targets # 훈련에 사용되기 위한 테스트의 실제 숫자값 배열 (답안지)
        self.target_names = target_names # 0~9 까지 숫자를 기억하고 있는 배열
        self.k = k # knn 알고리즘에서 정의하는 k 값
        self.distances = [] # 테스트 입력이 주어질경우 모든 데이터와 거리비교를 하여 저장할 배열
        self.k_cnt_idx = [] # 거리에 따라 정렬을 한후 거리가 작은순서의 distances 의 원소의 index값 를 기억하기 위한 배열
```

KNN 클래스 생성자

feature :훈련에 사용되는 data_set 을 저장하는 변수

Target : 훈련에 사용되기 위한 lable을 저장하는 변수

Target_names:0부터 9까지 기억하고 있는 배열

k # knn 알고리즘에서 정의하는k 값

Distances 테스트 입력이 주어질경우 모든 데이터와 거리비교를 하여 저장할 배열

k_cnt_idx 거리에 따라 정렬을 한후 거리가 작은순서의distances 의 원소의index값 를 기억하기 위한 배열 (argsort() 넘파이 메소드를 사용하여 계산한다)

```
    def calculate_distance(self, p1, p2):
        # euclidian distance 를 구하기 위한 메소드
        # numpy의 braodcast 성질을 이용하여 계산
        return np.sqrt(((p1-p2)**2).sum(axis=1))
        #p1 ( 60000 길이의 배열) : 비교군 / p2 (feautre 길이의 배열) 실험군


    def k_nearest_neighbor(self, testcase):
        # calculate_distance 메소드를 이용하여 모든 훈련에 사용되는 분꽃들의 특징과 실험되는 분꽃 과의 계산한 거리값을
        # self.distances  라는 배열에다가 담는다
        self.distances = self.calculate_distance(self.feature, testcase)
        # boardcast 성질을 이용하여 쉽게 각거리의 값의 배열을 계산할수 있다.
        self.k_cnt_idx = (self.distances.argsort())[:self.k]
        # argsort()를 통해서 배열이 나오게 되는데
        # 모든 거리들중에서 작은원소 순서의 index를 k개만큼 기억하고 있는 배열이다.
        #numpy.array에서는 배열의 리스트의 index를 정하는 [ ] 안에 인덱스를 순서를 기억하는 배열을 인자로 넣어준는게 가능하다.
```

하나의 테스트케이스에 대하여 k_nearest_neighbor 메소드를 호출한다.

이 메소드를 호출할때는 예측을 하고싶은 테스트케이스를 인자로 넣어 실행을 시켜준다

k_nearest_neighbor 메소드 안에는 calculate_distance라는 메소드를 self.feature 과 testcase 라는 인자를 통해 호출한다

Self.feature 는 6만개의 train 데이터이고 testcase 는 현재 test 데이터 한 개이다

Calculate_distance 메소드내부에는 broadcast 성질을 이용하여 6만개의 데이터와 현재 test데이터를 뺴준후 제곱을 한후 모두 더한다 더한결과를 제곱근으로 반환한다

Calculate_distance 의 결과도 길이가 6만이고 해당 원소에는 float32 타입의 실수가 저장되어있다.

그리고 np.argsort() 라는 것을 이용하여 실제 self.distance 배열을 변형하지 않고 index 만 반환하여 기억한다
np.argsort() 함수의 반환값을 다시 리스트의 인덱스 [ ] 안에 넣어주면 해당 인덱스에 해당하는 원소들만 다시 np.array()배열로 반환이 되므로 아주 유용한 함수이다.

K개의 원소만 기억하면 되므로 배열 splice 를 이용하여 길이가 K가 되게끔 self.k_cnt_idx 를 초기화 해준다.

```
41    def weighted_majority_vote(self):
42        vote = np.zeros(len(self.target_names), dtype='float')
43        # k개의 특징점중 최대 weight를 가진 분포을 알기위해 숫자의종류만큼 (0~9) 배열을 선언해놓는다.
44        inverse_distance = np.array([float("inf") if (i == 0) else (1 / i) for i in self.distances[self.k_cnt_idx]],dtype=np.float64)
45        # 거리에 대해서 가까울수록 가중치를 두기 위하여 새로운 inverse_distance라는 배열을 선언하고
46        # 이 배열안에 모든 거리값들의 역수를 취해준다 (divide by zero exception 을 피하기 위해 삼항연산자 사용)
47        # 이미 distance 배열이 순서가 정해져있으므로 역수로 변환만 시켜준다
48        for target, w in zip(self.targets[self.k_cnt_idx], inverse_distance):
49            vote[target] += w
50        # 각각의 target 에 해당하는 index에 역수로 취해준 가중치를 더해준다
51        return np.argmax(vote) #가장 가중치가 높은 target_names 를 반환한다
52
```

42번째줄에서는 0부터 9까지의 가중치를 저장할 길이가 10인 vote 배열을 선언한다
넘파이 데이터타입은 실수형으로 하여 실수의 덧셈과 오버플로우를 해결한다.
Inverse_distance 배열은 기존에 계산하였던 self.distacne 의 모든 원소들 중에서 아까 계산하였던 K개의 인덱스들 (self.k_cnt_idx) 에 해당하는 원소들만 역수를 취하여 다시 저장한다.
여기서 완전 데이터가 같아서 거리가 0이 되는 경우를 방지하기 위해 삼항연산자를 이용하여 float(“inf”) 가 반환될수 있게끔 처리 하였다.
그다음 48번째 줄에서 실제 label 을 기억하고 있는 self.targets 의 K개의 데이터들
Self.targets[self.k_cnt_idx] 와 거리의 가중치 inverse_distance 를 zip을 이용하여 같이 순회를 하며 vote에 가중치를 더해준다

최종적으로 51번째줄에서 가장 값이 큰 인덱스를 반환해준다.

Main.py

```
12    x_train = x_train.astype(np.float32)
13    x_test = x_test.astype(np.float32)
14
15    label_name = [str(i) for i in range(10)]
16
17    test_counts = 10000
18    test_idxs = np.random.randint(0, t_test.shape[0], test_counts)
```

X_train 과 x_test 모두 KNN 클래스 내부의 calculate_distance메소드를 적용하기 위해서

타입을 float32로 변환해주어야 한다.
그렇지 않으면 오버플로우가 나기 때문에 12~13번줄 작업은 필수적이다.

15번줄 : KNN클래스 생성자에 들어갈 label_name [“0” ,…, “9”]을 초기화한다.
17번줄 : test 를 수행할 데이터의 개수를 정한다.
18번줄 : 0~10000중 임의의 test_counts개의 데이터를 뽑은 배열을 반환한다.
이 과제에서는 효율적인 연구를 위해 랜덤값을 이용하여 테스트를 진행할 것을 추천한다.

Main-pooling.py에 대한 설명

```
(x_train, t_train), (x_test, t_test) = load_mnist(flatten=True, normalize=False)

x_train = x_train.astype(np.float32).reshape(60000, 28, 28)
x_test = x_test.astype(np.float32).reshape(10000, 28, 28)

x_train = np.array([pooling_2d_arr(i) for i in x_train]).reshape((60000, 196))
x_test = np.array([pooling_2d_arr(i) for i in x_test]).reshape((10000, 196))
```

X_train 과 x_test 모두 784길이의 배열원소들을 28 * 28 로 reshape 해주어 3차원 배열로 만든다음
Pooling_2d_arr 함수를 이용하여 인접한 4개의 데이터의 max값만 가진 2차원 배열(14 * 14)로 변환한다.
그 다음 거리계산의 편의성을 위해 196길이의 1차원 넘파이 배열로 다시 변환한다.

```
12  def pooling_2d_arr(arr):
13      arr = np.pad(arr, 0, mode='constant')
14
15      output_size, target_size = (14, 14), (2, 2)
16      arr_w = as_strided(arr, shape=output_size + target_size,
17                         strides=(2 * arr.strides[0],
18                                  2 * arr.strides[1]) + arr.strides)
19      arr_w = arr_w.reshape(-1, *target_size)
20
21      return arr_w.max(axis=(1, 2)).reshape(output_size)
22
```

기존 배열의 차원이 작아지거나 변형이 되는 것을 방지하기 위해 패딩을 적용해주었다 14 *14 사이즈의 2차원 배열을 반환하길 원하므로 output_size 를 14,14로 설정해주고
2 x 2 사이즈안에서 최대값을 반영하길 희망하므로 target_size 를 (2,2)로 정해준다
다음 as_strided 를 이용하여 최적화된 배열변형과 분할을 한다.
19번째줄에서 한번 더 감싸여진 리스트를 해제한다.
반환할 때 1차원에 대하여 최대값으로 초기화해준후 2차원에 대해서도 최대값으로 초기화를 해준다 그다음 reshape 을 이용하여 14 * 14 로 반환하여 준다.

6. 학습 과정에 대한 설명

1.강의와 강의록을 이용

2.numpy 공식페이지를 통하여 메소드를 학습하였다.

https://numpy.org/

3.넘파이와 최적화 , pooling 에 대해서 인터넷검색을 통해 학습하였다.

http://rasbt.github.io/mlxtend/user_guide/data/mnist_data/

https://www.classes.cs.uchicago.edu/archive/2013/spring/12300-1/pa/pa1/

https://www.youtube.com/watch?v=ZD_tfNpKzHY

https://towardsdatascience.com/how-fast-numpy-really-is-e9111df44347

7. 결과 및 분석

결과 출력 및 결과에 대한 분석

테스트 데이터 1000개

1)784개 input을 모두 사용한 경우

```
k is :  3
tries :  1000
answer :  969 / wrong :  31
accuracy :  96.89999999999999 %
k is :  5
tries :  1000
answer :  973 / wrong :  27
accuracy :  97.3 %
k is :  7
tries :  1000
answer :  975 / wrong :  25
accuracy :  97.5 %
k is :  10
tries :  1000
answer :  972 / wrong :  28
accuracy :  97.2 %
```

2)pooling 방식의 Hand-crafted feature

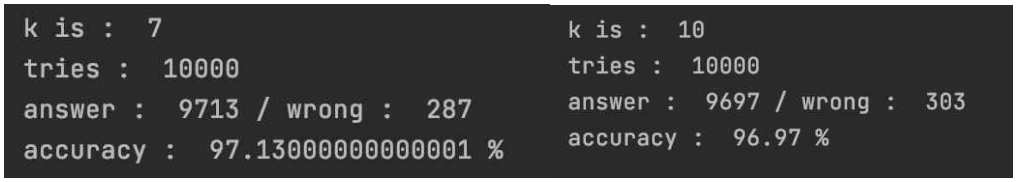
```
Python Console
k is :  3
tries :  1000
answer :  970 / wrong :  30
accuracy :  97.0 %
k is :  5
tries :  1000
answer :  975 / wrong :  25
accuracy :  97.5 %
k is :  7
tries :  1000
answer :  970 / wrong :  30
accuracy :  97.0 %
k is :  10
tries :  1000
answer :  970 / wrong :  30
accuracy :  97.0 %
```

테스트 데이터 10000개

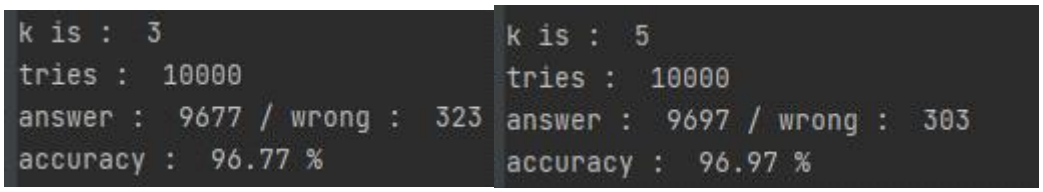
1.784개 input을 모두 사용한 경우

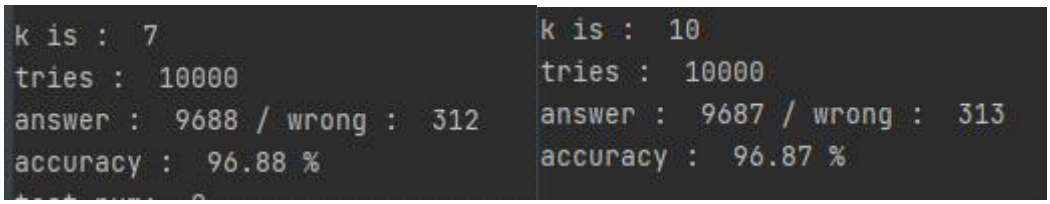
```
k is :  3
tries :  10000
answer :  9734 / wrong :  266
accuracy :  97.34 %
```

```
k is :  5
tries :  10000
answer :  9699 / wrong :  301
accuracy :  96.99 %
```



2.Pooling 방식을 이용하여 Hand-crafted feature 14 x 14 로 변환한 경우





분석 :
예측률 은 Polling 을 했을때보다 모든 input feature 를 그대로 이용해주었을 때가 조금더 잘 나오는 것을 확인할수 있다.
하지만 모든 데이터에서 같은 양상을 보이는 것은 아닌것같다.
K값또한 결과에 많은 영향을 주진 않아 보인다.
시간은 압도적으로 784개의 feature 를 사용하였을때가 pooling 방식을 사용하였을때보다 3,4배는 더 걸렸다.
정확도는 데이터의 수가 1000개 이상이면 비교적으로 비슷하게 나오는 것을 확인하였다 랜덤으로 테스트 데이터들을 추출한게 예측률 추론에 많은 도움이 되었던것같다.

예측결과가 1번과 2번이 많이 차이는 나지 않는 것으로 보다 적당하게 Hand-crafted feature 로 가공하여 추론에 이용하는 것이 훨씬 현실적인 대안이 될수있다고 생각한다.
또한 파이썬에서 기본으로 제공되는 list 의 append pop 메소드 대신 numpy 의 메소드를 쓰는 것이 계산시간단축에 많은 도움이 되는 것을 이번과제를 하면서 체감할수 있었다.