**Assignment 1 Report**

심명진

Q1: k-Nearest Neighbor classifier

Q2: Training a Support Vector Machine

Q3: Implement a Softmax classifier

Q4: Two-Layer Neural Network

Q5: Higher Level Representations: Image Features

**[FULL CODE]** assignment 1

<https://github.com/myeongmy/deepLearning/tree/master/assignment1>

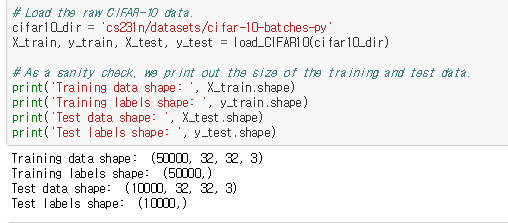
Q1 : k-Nearest Neighbor classifier

Knn 분류기는 다른 선형 분류기나 신경망 네트워크, CNN과는 다르게 데이터 중심 접근방법을 이용한다. 각 카테고리 별로 이미지들을 수집하여 이미지와 라벨 자체를 모델에 훈련시킨다. 훈련 시간은 적게 드는 반면, 새로운 이미지에 대한 prediction이 오래 걸린다는 단점이 있다.

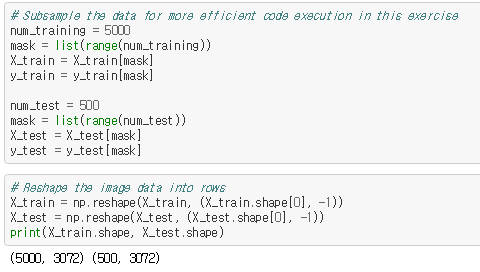
* 이 과제에서 구현해야 할 것

1. Image classification 과정 이해
2. 하이퍼파라미터인 k를 구하기 위한 cross-validation 시행

<knn.ipynb>



먼저 데이터셋(CIFAR-10)을 로드하여 training data와 그에 대한 라벨 데이터, testing data와 그에 대한 라벨 데이터를 마련한다. Training data는 50000개, testing data는 10000개에 해당한다.

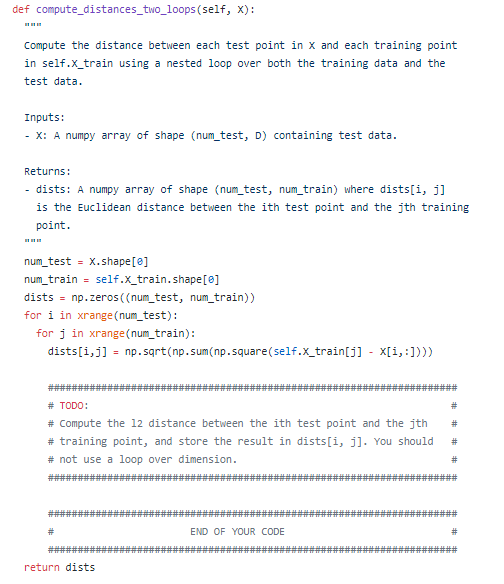


이 예제에서는 training data로 5000개, testing data로 500개만 이용하기로 한다. 그리고 각 이미지 데이터를 32\*32\*3 열로 길게 펼친다.

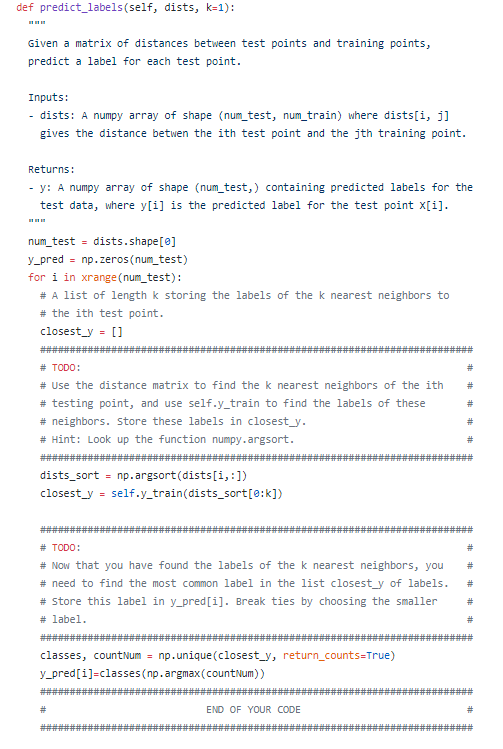
<k\_nearest\_neighbor.py>

Training data를 k\_nearest\_neighbor.py의 train 함수로 훈련시킨 뒤, 500개의 testing data와 5000개의 training data 간의 거리 비교를 하려고 한다.

먼저 두 개의 반복 loop를 이용한 방법이다.

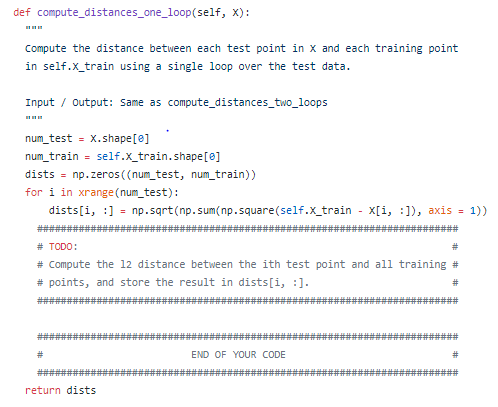


input으로 testing data X를 받아 testing data의 개수를 num\_test 변수에 저장하고 인스턴스 변수인 X\_train을 통해 training data의 개수를 num\_train에 저장한다. Dists 배열을 (500, 5000) shape의 모든 원소가 0인 배열로 초기화를 한 후, 각 for문을 돌면서 인덱스 하나하나씩 채워간다. 우리는 L2 distance를 구해야 하므로 500개의 testing data를 모든 training data와 한 번씩 비교를 해나가는데 각 셀 값의 차를 제곱하고 그 값들을 전부 더하여 루트 연산을 적용하면 해당 training data와 해당 testing data의 거리 차이가 계산된다. 그리고 그 dists 배열을 리턴한다.



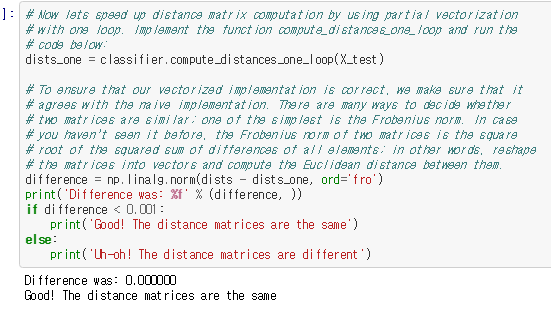
위에서 구한 dists 배열을 가지고 predict\_labels 함수를 이용해 testing data에 대한 라벨 값을 예측한다. Testing data 500개 하나하나 for문을 돌면서 예측되는 라벨을 구해 y\_pred 배열을 완성한다. Closest\_y에는 5000개의 training data와의 거리 데이터 중 가장 작은 거리에 해당하는 k개를 골라 그에 해당하는 라벨들을 저장한다. 그리고 그 closest\_y에 unique 연산을 실시해 각 클래스(라벨)별로 개수를 구한다. 그래서 개수가 가장 큰 클래스(라벨)을 해당 testing data의 라벨 값으로 최종 결정한다.

이제는 거리 측정 연산의 속도를 높이기 위해 loop를 한 번만 사용하기로 한다.

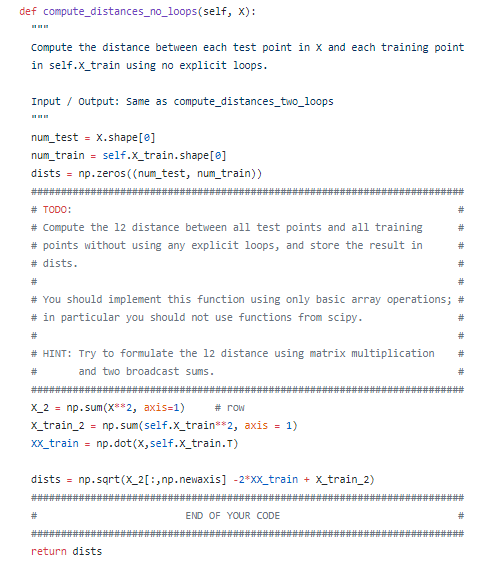


Loop는 testing data 500개에 대해서만 하나 돌며 dists[i]를 각각 구한다. Numpy는 vectorized 연산을 제공하므로 self.X\_train – X[I]를 하면 각 행별로 뺄셈을 한 결과가 나오게 되고 그것을 제곱하여 axis = 1로 덧셈을 하면 각 행별로 덧셈이 이루어진다. 그래서 그것에 루트 연산을 해주게 되면 하나의 testing data에 대한 거리 측정값을 구하게 된다.

* 결과 : loop를 두 개를 이용하나 하나를 이용하나 거리 측정 값에는 변화가 없음을 알 수 있다.

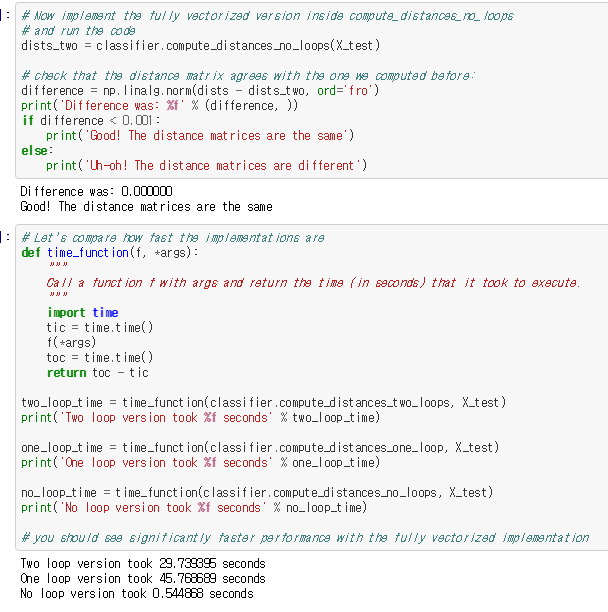


마지막으로 loop를 아예 사용하지 않고 fully vectorized하게 거리 측정 함수를 구현해 보면



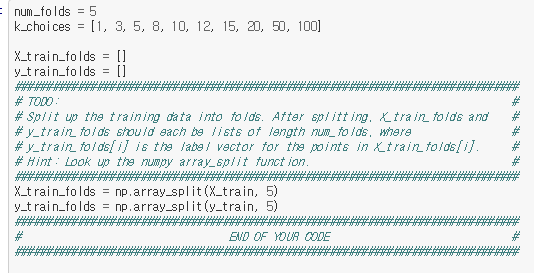
Loop를 쓰지 않고 구현하기 위해 l2 distance를 구하는 공식, 즉 완전제곱식의 형태를 풀어서 계산해보았다. 따라서, testing data의 제곱 값을 따로 구하고, training data의 제곱 값을 따로 구하고, 그 둘의 내적 곱을 따로 구해 마지막에 덧/뺄셈을 통해 dists를 구하였다.

* 결과 : 이것도 마찬가지로 앞에서 구한 two loop를 이용해서 구하는 방식과 결과 값은 다르지 않았다. 하지만 실행 시간 면에서는 훨씬 더 빠른 속도를 보였다. (왜 one loop가 더 오래 걸리는지 모르겠습니다..)

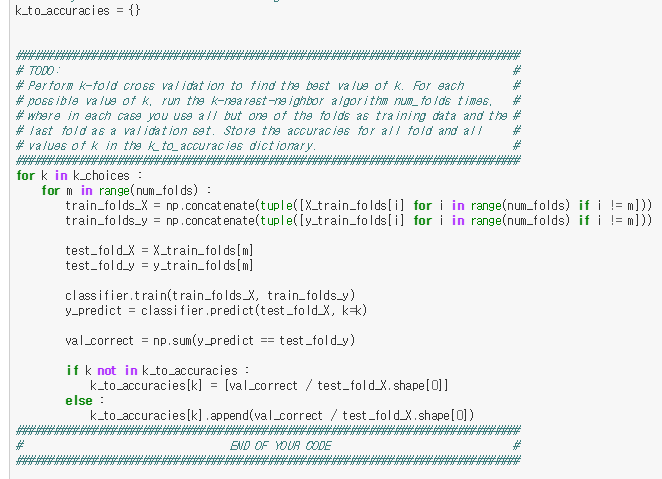


Cross-validation (좋은 k를 선정하기 위함)

<knn.ipynb>

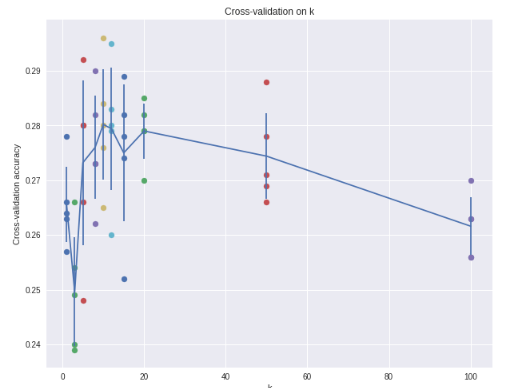


Fold의 개수는 5개라고 하고 training data를 다섯 등분해준다. Array\_split 연산을 이용하여 나눠주고 각각 X\_train\_folds, y\_train\_folds 배열에 담는다.

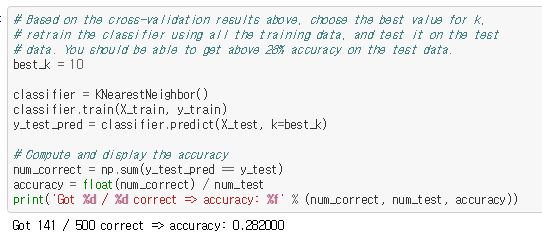


모든 k의 후보에 대해 knn 알고리즘을 적용하고 그 중 정확도가 제일 좋았던 k를 선택한다. 각 k마다 cross-validation 기법으로 훈련시키고 테스트 하기 때문에 하나의 k에 대하여 정확도 값은 5개가 나온다. (5개의 fold로 나누었으므로 각 fold가 한 번씩 testing data가 되는 것이다)

* 결과 : k=10일 때 정확도가 가장 좋아보인다.



따라서, k=10이라 하고 최종적으로 훈련시키고 정확도를 다시 계산하여보면 28.2%가 나온다.



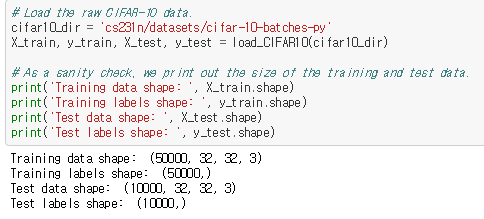
Q2: Training a Support Vector Machine

SVM classifier는 앞의 knn classifier와 다르게 데이터 중심의 접근 방법이 아닌 parametric 접근 방식이다. Training data를 이용하여 가중치 w를 최적화하여 이미지 분류를 가능하게 한다.

* 이 과제에서 구현해야 할 것

1. Loss function과 그에 대한 gradient 구하기
2. 최적의 learning rate와 regularization strength 찾기
3. SGD 방법을 이용하여 손실 함수 최적화

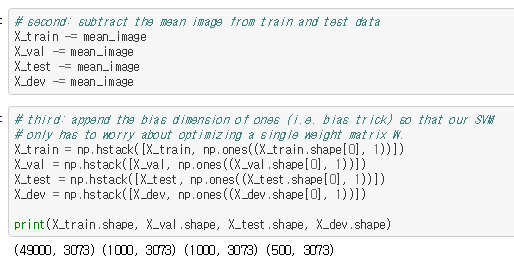
<svm.ipynb>



먼저 데이터셋(CIFAR-10)을 로드하여 training data와 그에 대한 라벨 데이터, testing data와 그에 대한 라벨 데이터를 마련한다. Training data는 50000개, testing data는 10000개에 해당한다.



데이터를 training, validation, testing set 세 가지로 나눈다. 그리고 training data 중 일부를 development set으로 가진다. 그리고 각 데이터셋들을 32\*32\*3 열로 길게 펼쳐준다.

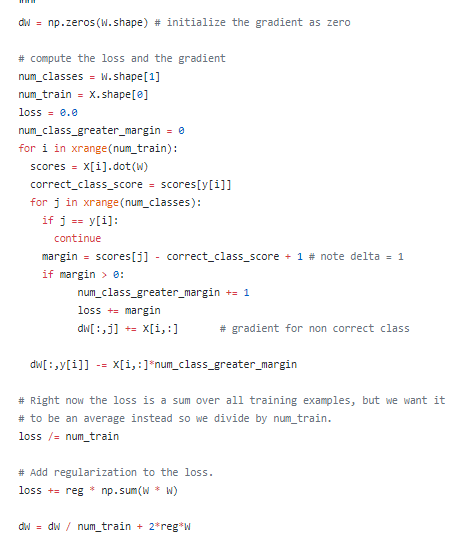


Training data를 모델에 훈련시키기 전에 데이터 전처리 과정이 필요하다. Training data와 testing data로부터 mean image를 추출하여 빼주고 bias 항을 추가해준다.

데이터 전처리가 완료되면 이제 svm loss function을 구현해야한다. 첫 번째는 loss function을 vectorized 연산을 이용하지 않고 naive하게 구현한다. 가중치 W는 작은 수로 초기화시켜두고 loss를 계산한다.

<linear\_svm.py>

Def svm\_loss\_naive



함수의 입력으로 가중치 w와 데이터와 데이터 라벨 배열을 받으면 각 이미지 데이터 별로 데이터 X와 W의 내적 곱을 구하여 각 클래스에 대한 score를 얻어낸다. 그리고 해당 이미지의 정답 클래스에 해당하는 점수를 따로 저장해둔 뒤, 모든 다른 클래스와 해당 점수를 비교하여 마진 값을 구한다. 이 때, 모든 클래스를 for문을 돌면서 정답 클래스에 해당하는 경우는 따로 마진 값을 계산하지 않고 넘어가고 그 이외의 클래스에 대해서는 마진 값을 계산한 뒤 해당 마진 값이 0보다 큰 경우에만 loss에 그 값을 더해준다. 이것이 svm loss를 구하는 방식이다. 이렇게 총 loss를 구해준 뒤 training data의 개수로 나눠주어 평균 loss를 얻고 마지막에 l2 regularization을 한다는 가정하에 regularization 항을 추가해준다.

Gradient 같은 경우에는 우리가 가중치 W에 대한 gradient가 필요한 것이므로 loss 계산해주었던 수식을 토대로 차례차례 편미분을 통해 식을 얻어낼 수 있다. 앞서 loss를 구할 때 마진 값이 0보다 큰 경우에만 loss에 해당 마진 값을 더해주었으므로 dW에도 그 경우에만 X[I] 값을 더해주고 정답 클래스의 경우에는 마진 값이 0 이상이었던 클래스의 개수만큼 오히려 빼준다. 그리고 전체 loss를 training data의 개수로 나누어주었던 것처럼 dW도 똑같이 나누어주고, reg\*np.sum(W\*W)의 미분 값인 2\*reg\*W도 추가로 더해준다.

다음은 똑같은 svm loss를 vectorized하게 구현한다.