Homework 3

Collaborators:
Name: Jiang Shibiao
Student ID: 3170102587

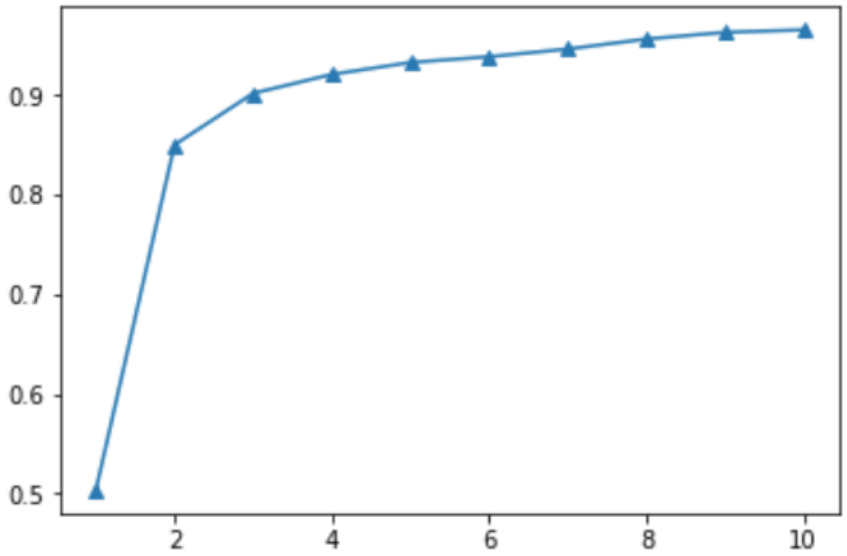
Problem 3-1. Neural Networks

In this problem, we will implement the feedforward and backpropagation process of the neural networks.

(a) Answer: After completing the files, execute *run.ipynb*. Final test error is 92.7%.

```
# TODO
import matplotlib.pyplot as plt

plt.figure()
plt.plot(XX, YY, "-^")
plt.show()
loss, accuracy, _ = feedforward_backprop(test_data, test_label, weights)
print('loss:{:.3}, accuracy:{}'.format(loss, accuracy))
```




```
loss:0.241, accuracy:0.927
```

Problem 3-2. K-Nearest Neighbor

In this problem, we will play with K-Nearest Neighbor (KNN) algorithm and try it on real-world data. Implement KNN algorithm (in *knn.m*/*knn.py*), then answer the following questions.

(a) Try KNN with different K and plot the decision boundary.

Answer:

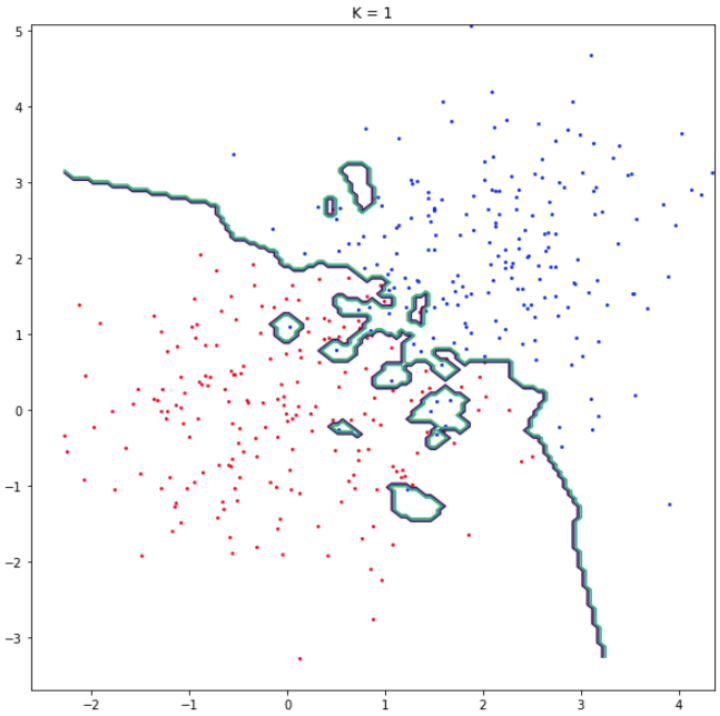
I implemented the direct algorithm at first, but it ran so slow (No results after 5 min).

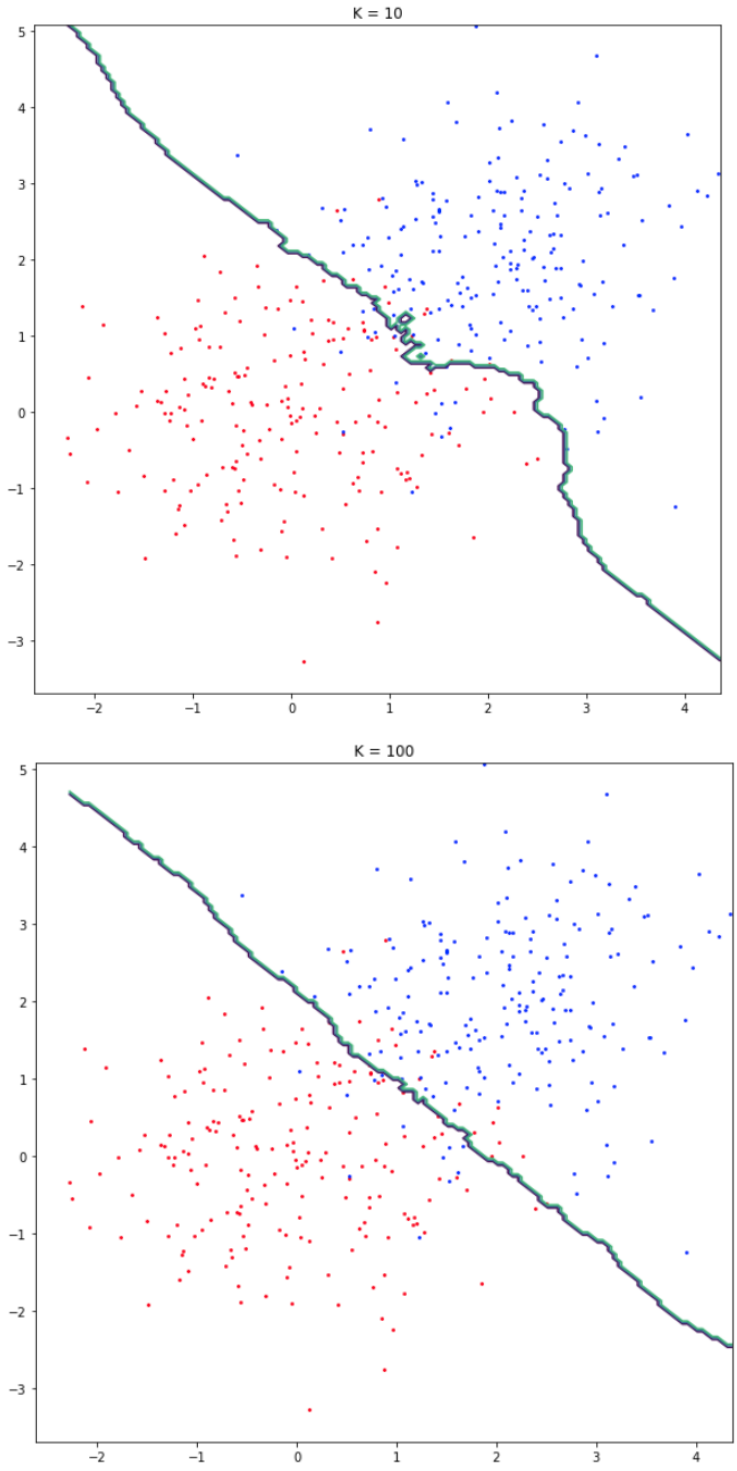
```
ret = []
for x_ask in x:
    neighbors = np.argsort([distance(x, x_train_example) for x_train_example in x_train])[:k]
    label_dict = {}
    for nei in neighbors:
        label_dict[y_train[nei]] = label_dict.get(y_train[nei], 0) + 1
    ret.append(max(label_dict, key = label_dict.get))
# The above solution is too slow
```

Then I wrote a matrix version and the results came out in few seconds.

```
xy = np.matmul(x, x_train.T)
xx = np.tile(np.sum(x ** 2, axis = 1), (x_train.shape[0], 1)).T
yy = np.tile(np.sum(x_train ** 2, axis = 1), (x.shape[0], 1))
#print (xy.shape, x.shape, xx.shape, x_train.shape, yy.shape)
dxy = xx + yy - 2 * xy
lbs = y_train[np.argsort(dxy, axis = 1)[:, :k]]
ret = scipy.stats.mode(lbs, axis = 1)[0].flatten()
```

The figures are as follow.





(b) We have seen the effects of different choices of K. How can you choose a proper K when dealing with real-world data ?

Answer: We can use cross validation to try different K and find the best one.

(c) Finish *hack.m*/*hack.py* to recognize the CAPTCHA image using KNN algorithm.

Answer:

First I download some CAPTCHA pictures and make a label-data file **hack_data.npz**. There are 400 annotated cases totally.

To choose a suitable K, I split some valid data from the whole data, and iterate K from 1 to 100 and call the method **KNN** in **hack.py**. The codes are as follows.

```
x_train = data['x_train']
y_train = data['y_train']
number = len(x_train)

# begin answer
x_train, x_valid = x_train[ : number // 2], x_train[number // 2:]
y_train, y_valid = y_train[ : number // 2], y_train[number // 2:]

best_acc, best_k = 0.0, 1
for k in range(1, 101):
    y = knn.knn(x_valid, x_train, y_train, k)
    acc = np.sum(y == y_valid) / len(y)
    print ("K =", k, " ACC =", acc)
    if acc > best_acc:
        best_acc = acc
        best_k = k
print ("Choose", best_k, "as K.")
digits = knn.knn(x, x_train, y_train, best_k)
# end answer
return digits
```

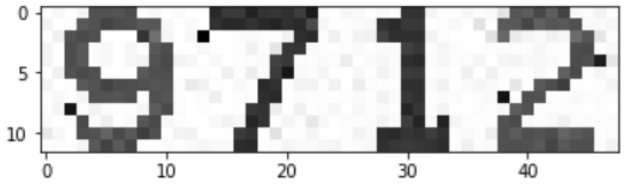
It seems that **smaller K will reach better accuracy**. Finally I set $K = 5$.

```
K = 1   ACC = 1.0      K = 21  ACC = 0.75    K = 81  ACC = 0.24
K = 2   ACC = 1.0      K = 22  ACC = 0.75    K = 82  ACC = 0.22
K = 3   ACC = 1.0      K = 23  ACC = 0.75    K = 83  ACC = 0.21
K = 4   ACC = 1.0      K = 24  ACC = 0.74    K = 84  ACC = 0.2
K = 5   ACC = 1.0      K = 25  ACC = 0.7     K = 85  ACC = 0.19
K = 6   ACC = 1.0      K = 26  ACC = 0.69    K = 86  ACC = 0.18
K = 7   ACC = 1.0      K = 27  ACC = 0.64    K = 87  ACC = 0.13
K = 8   ACC = 1.0      K = 28  ACC = 0.63    K = 88  ACC = 0.12
K = 9   ACC = 1.0      K = 29  ACC = 0.62    K = 89  ACC = 0.12
K = 10  ACC = 1.0      K = 30  ACC = 0.62    K = 90  ACC = 0.12
K = 11  ACC = 1.0      K = 31  ACC = 0.62    K = 91  ACC = 0.12
K = 12  ACC = 0.98     K = 32  ACC = 0.61    K = 92  ACC = 0.12
K = 13  ACC = 0.8      K = 33  ACC = 0.59    K = 93  ACC = 0.12
K = 14  ACC = 0.78     K = 34  ACC = 0.56    K = 94  ACC = 0.12
K = 15  ACC = 0.77     K = 35  ACC = 0.54    K = 95  ACC = 0.12
K = 16  ACC = 0.77     K = 36  ACC = 0.54    K = 96  ACC = 0.12
K = 17  ACC = 0.76     K = 37  ACC = 0.54    K = 97  ACC = 0.12
K = 18  ACC = 0.76     K = 38  ACC = 0.53    K = 98  ACC = 0.12
K = 19  ACC = 0.76     K = 39  ACC = 0.53    K = 99  ACC = 0.12
K = 20  ACC = 0.75     K = 40  ACC = 0.53    K = 100  ACC = 0.12
```

Then I successfully hack the CAPTCHA.

```
Choose 1 as K.

Out[37]: array([9., 7., 1., 2.])
```



Problem 3-3. Decision Tree and ID3

Consider the scholarship evaluation problem: selecting scholarship recipients based on gender and GPA. Given the following training data:

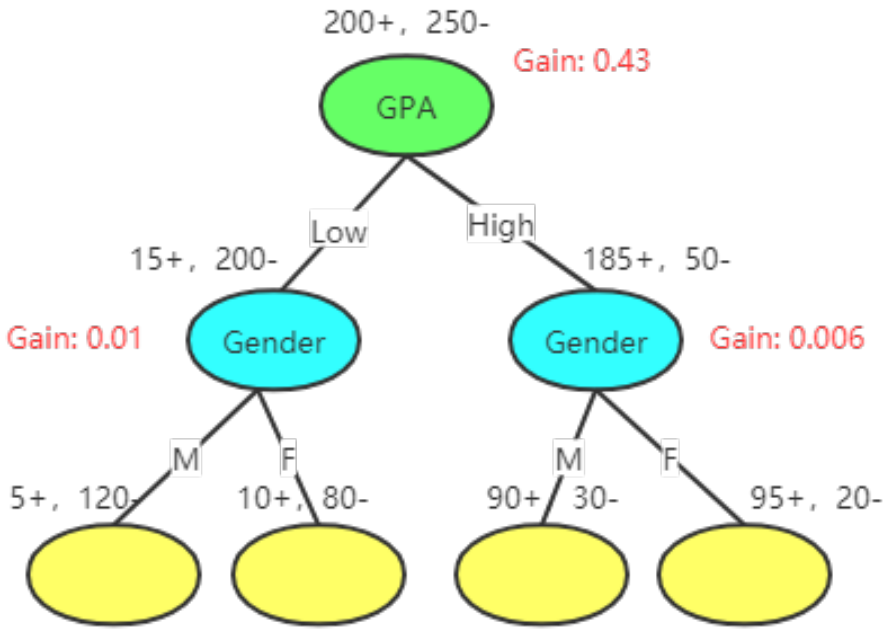
Answer: We should determine which property should be chosen in the first branch.

$$Ent(D) = -\sum_{k=1}^{|y|} p_k \log_2(p_k) = -\frac{200}{450}\log_2(\frac{200}{450}) - \frac{250}{450}\log_2(\frac{250}{450}) = 0.9911$$

$$Gain(D, Gender) = Ent(D) - \frac{205}{450}Ent(D_F) - \frac{245}{450}Ent(D_M) = 0.0112$$

$$Gain(D, GPA) = Ent(D) - \frac{215}{450}Ent(D_{Low}) - \frac{235}{450}Ent(D_{High}) = 0.4267$$

So first choose GPA as the division property.



Problem 3-4. K-Means Clustering

Finally, we will run our first unsupervised algorithm k-means clustering.

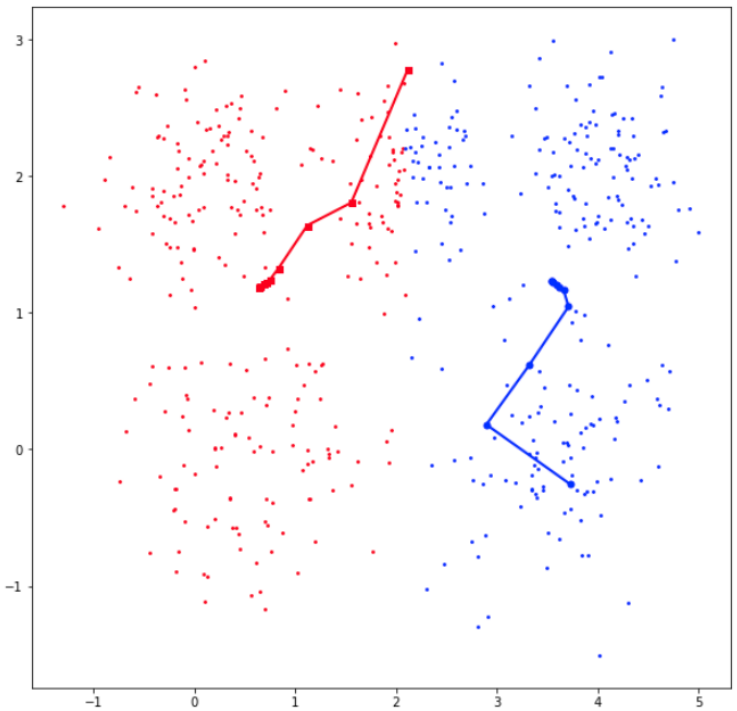
(a) Visualize the process of k-means algorithm for the two trials.

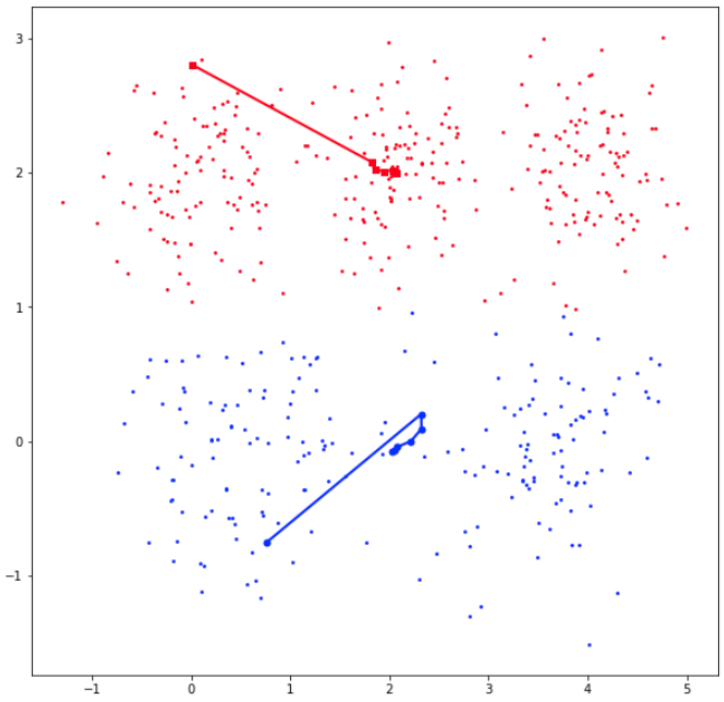
Answer: I set **max_iter=10**, and do the same trick as KNN (matrix notation) to acclerate the calculation towards distance.

Repeat 1000 times to find the min and max SD.

```
for re in range(1000):
    idx, ctrs, iter_ctrs = kmeans(x, k)
    sumdist = 0
    for i in range(idx.shape[0]):
        sumdist += np.sqrt(np.sum((ctrs[idx[i]] - x[i])**2))
    if sumdist < minSD:
        minSD = sumdist
        record[0] = [idx, ctrs, iter_ctrs]
    if sumdist > maxSD:
        maxSD = sumdist
        record[1] = [idx, ctrs, iter_ctrs]
print (minSD, maxSD)
kmeans_plot(x, *record[0])
kmeans_plot(x, *record[1])
```

```
637.4200237561756 771.7458654811335
```



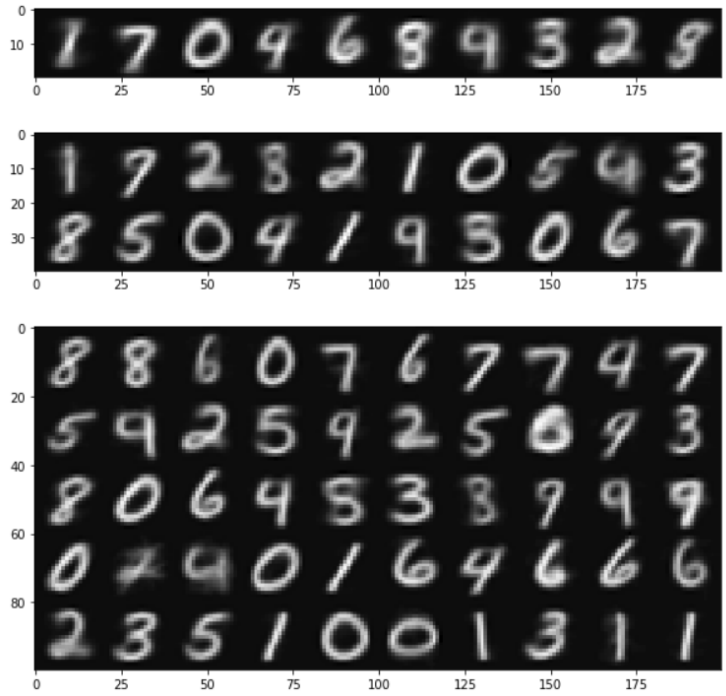


(b) How can we get a stable result using k-means?

Answer: repeat some times to get a relative stable soltuion.

(c) Visualize the centroids.

Answer:



(d) Vector quantization.

Answer: We can simply call **kmeans** method to reach the target.

```
for k in [8, 16, 32, 64]:
    idx, ctrs, _ = kmeans.kmeans(fea.copy(), k)
    new_img = ctrs[idx]
    plt.imshow(new_img.reshape(img.shape).astype(np.uint8))
    plt.show()
```

When I was running the code, I found that in some case the program will output **nan** so that the picture can not be generated correctly. It is because a centroid may loses all its points during the kmeans' iteration. So I added the following special judge.

```
for j in range(k):
    if np.sum(idx == j) > 0:
        ctrs[j] = np.mean(x[idx == j], axis = 0)
```

For $k = 64$, compression ratio is $\frac{HW \cdot \log(64)}{HW \cdot \log(256) \cdot 3} = \frac{1}{4}$.

Then I test for **sample0.jpg** and **sample1.jpg**.

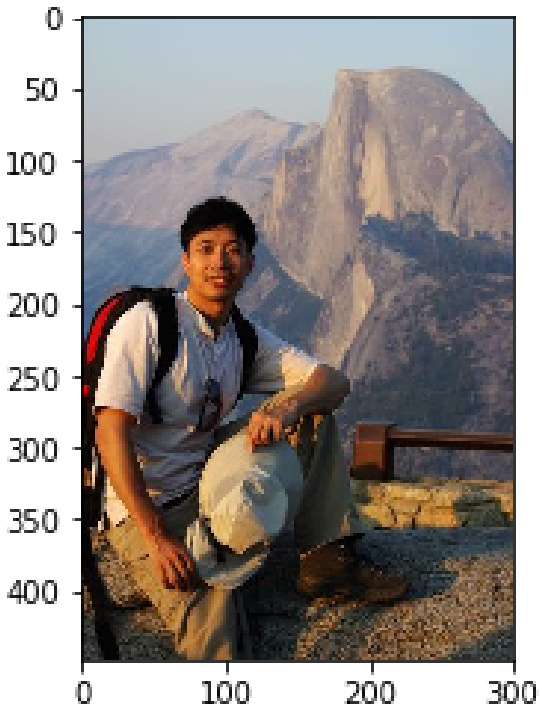
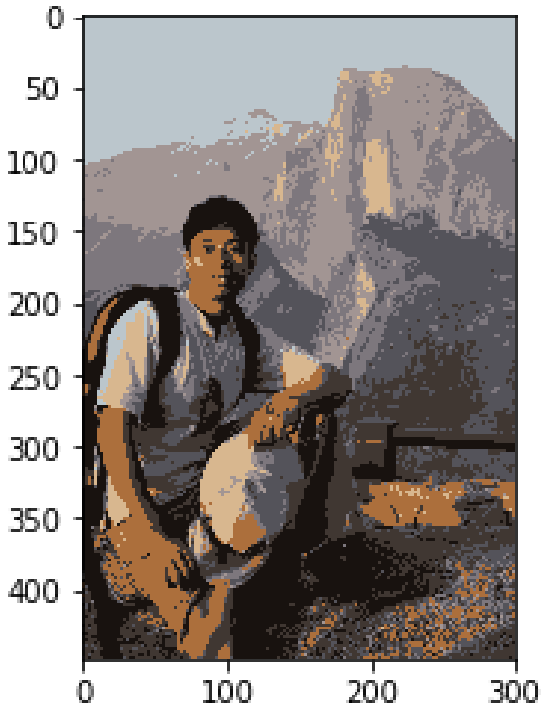
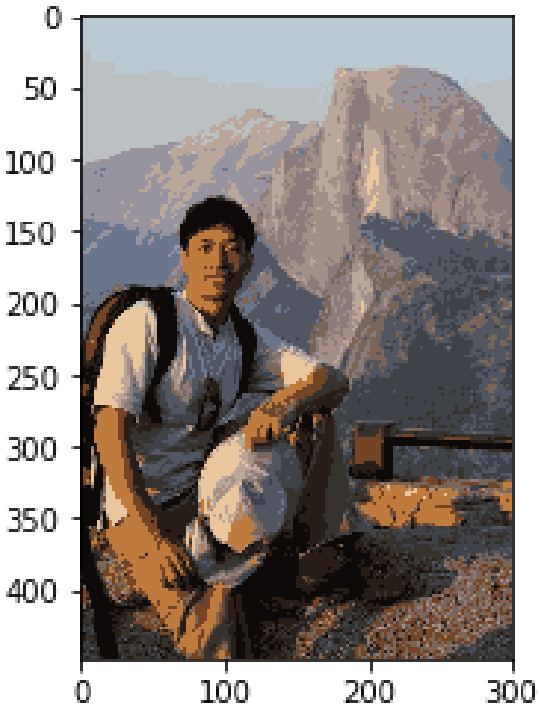


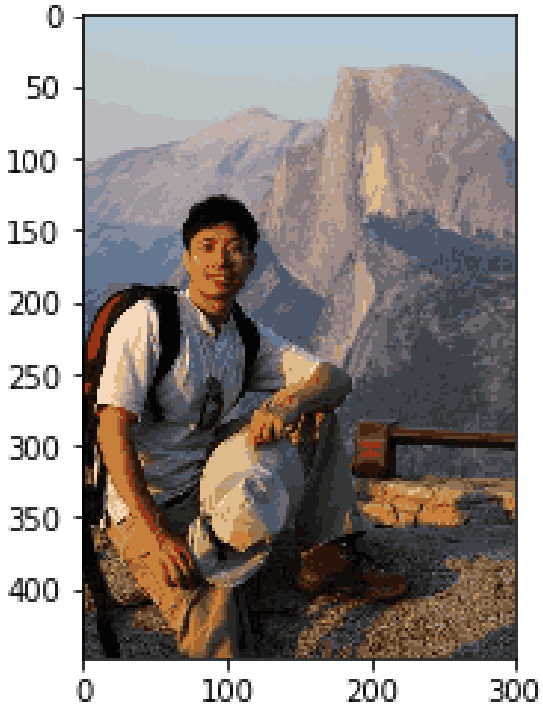
Figure 1: Original picture for sample0.jpg



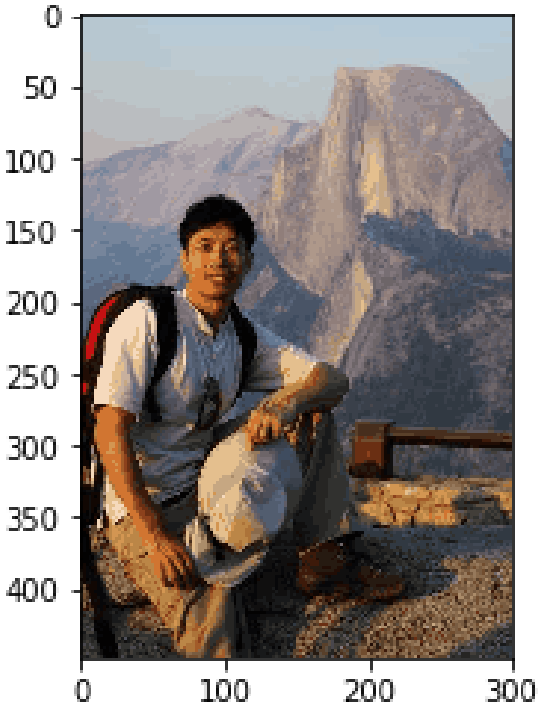
(a) K=8



(b) K=16

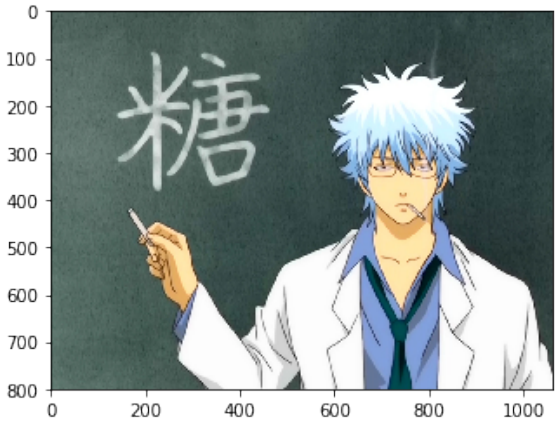


(c) K=32

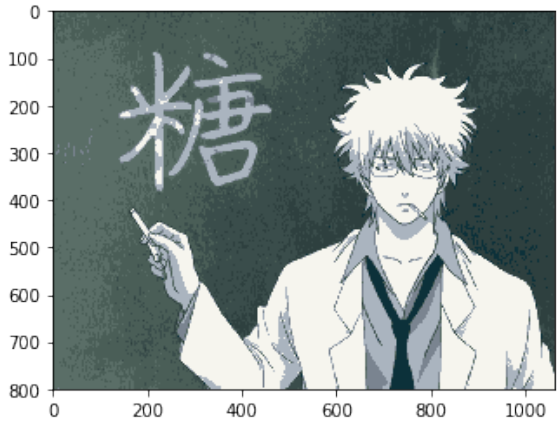


(d) K=64

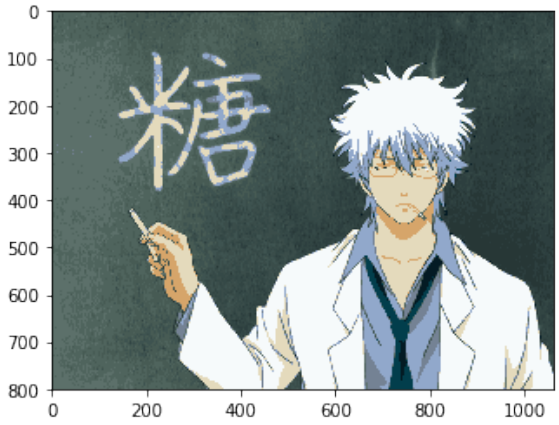
Figure 2: Test for sample0.jpg



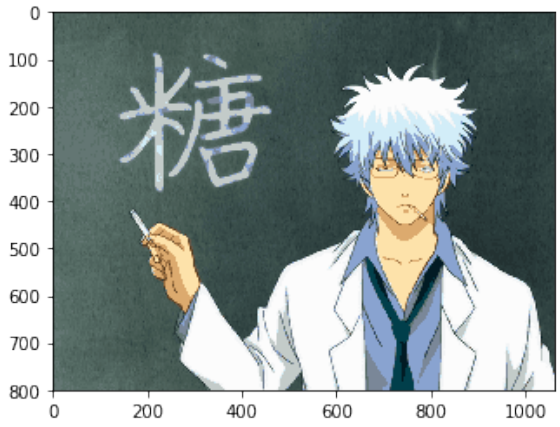
(a) Original Picture



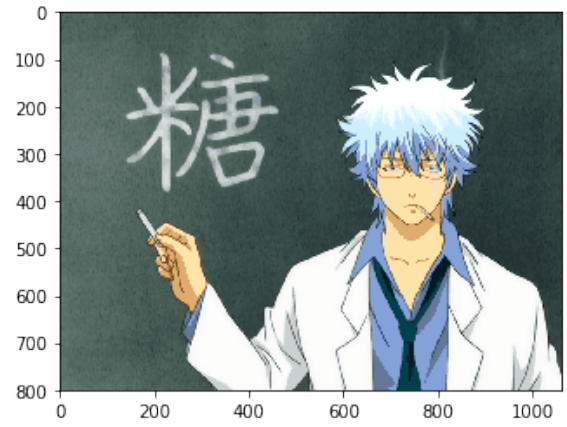
(b) K=8



(c) K=16



(d) K=32



(e) K=64

Figure 3: Test for sample1.jpg