# Assignment1 Report

## K Means

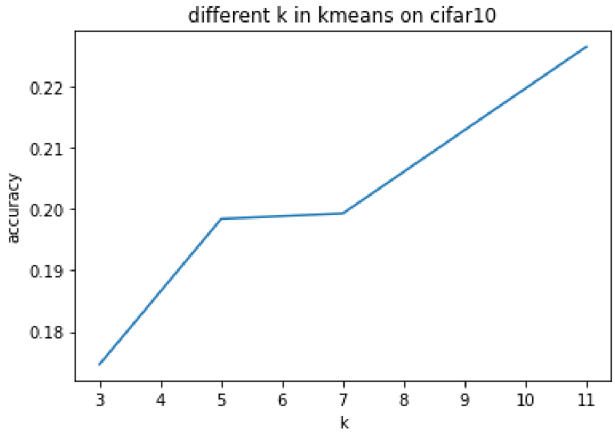
I use matrix of numpy instead of loop to make my code more efficient

```
class kmeans:
    def __init__(self):
        pass

    def fit(self, X_train, y_train, k, normalize=False, limit=20,
verbose=False):
        if normalize:
            stats = (X_train.mean(axis=0), X_train.std(axis=0))
            X_train = (X_train - stats[0]) / stats[1]

        self.centers = X_train[:k]
        self.k = k

        for i in range(limit):
            self.classifications = np.argmin(((X_train[:, :, None] -
self.centers.T[None, :, :])**2).sum(axis=1), axis=1)
            new_centers = np.array([X_train[self.classifications == j,
:].mean(axis=0) for j in range(k)])
            if (new_centers == self.centers).all():
                break
            else:
                self.centers = new_centers
                if verbose:
                    print("finish iter {}/{}".format(i + 1, limit))

        if normalize:
            self.centers = self.centers * stats[1] + stats[0]

        cluster_labels = np.zeros((k, len(np.unique(y_train))))

        for i, y in enumerate(y_train):
            cluster_labels[self.classifications[i], y] += 1
        self.cluster_labels = np.argmax(cluster_labels, axis=1)

    def predict(self, X_test):
        test_classifications = np.argmin(((X_test[:, :, None] -
self.centers.T[None, :, :])**2).sum(axis=1), axis=1)

        test_labels = np.zeros(len(X_test))
        for i, l in enumerate(test_classifications):
            test_labels[i] = self.cluster_labels[l]

        return test_labels
```

| | k=3 | k=5 | k=7 | k=11 |
|---|---|---|---|---|
| train accuracy(mean of cross validation) | 0.175 | 0.198 | 0.199 | 0.227 |
| test accuracy | - | - | - | 0.225 |

The best k is 11 and the accuracy of test data is 0.225

The following shows how the k of k means infect the accuracy




## KNN

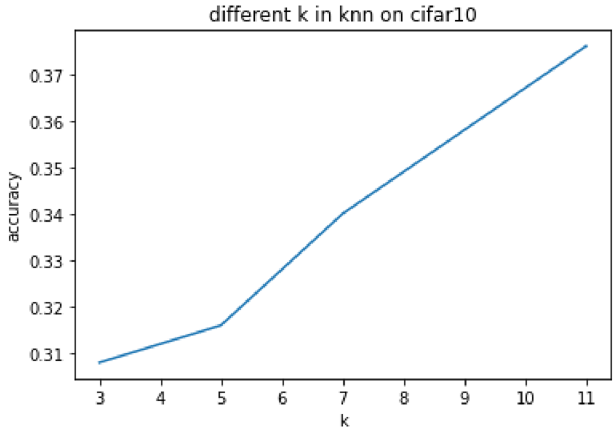
Same as k means, I also use matrix

```
class knn:
    def __init__(self):
        pass

    def fit(self, X_train, y_train):
        self.x = X_train
        self.y = y_train

    def predict(self, X_test, k, verbose=False):
        total = len(X_test)
        y_pred = np.zeros(total)
        for i, x in enumerate(X_test):
            dist = np.linalg.norm(self.x - x, axis=1)
            topk = np.argsort(dist)[: k]
            y_pred[i] = np.bincount(self.y[topk]).argmax()
            if verbose and i % 50 == 0:
                print("Finish {}/{}".format(i + 1, total))
        return y_pred
```

| | k=3 | k=5 | k=7 | k=11 |
|---|---|---|---|---|
| train accuracy(mean of cross validation) | 0308 | 0.316 | 0.34 | 0.376 |
| test accuracy | - | - | - | 0.341 |

The best k is 11 and the accuracy of test data is 0.341

The following shows how the k of KNN infect the accuracy




## Softmax

Since when I ran the softmax algorithm, I found numpy warning "exp function overflow", after reading some materials, I use the following softmax function to avoid overflow

$$
\begin{aligned}
\log[f(x_i)] &= \log\left(\frac{e^{x_i}}{e^{x_1} + e^{x_2} + \cdots e^{x_n}}\right) \\
&= \log\left(\frac{\frac{e^{x_i}}{e^M}}{\frac{e^{x_1}}{e^M} + \frac{e^{x_2}}{e^M} + \cdots \frac{e^{x_n}}{e^M}}\right) \\
&= \log\left(\frac{e^{(x_i - M)}}{\sum_j^n e^{(x_j - M)}}\right) \\
&= \log\left(e^{(x_i - M)}\right) - \log\left(\sum_j^n e^{(x_j - M)}\right) \\
&= (x_i - M) - \log\left(\sum_j^n e^{(x_j - M)}\right)
\end{aligned}
$$

For softmax, I written learning rate schedule, l2 regulation, early stop

For learning rate schedule, I use formula $lr = lr_{base} \cdot lr_{decay}^{iteration/lr_{step}}$

```
class softmax_regression:
    def __init__(self):
        self.stats = np.array([0, 1])

    def softmax(self, z):
        maximum = np.max(z, axis=1).reshape(-1, 1)
        return np.exp(z - maximum) / np.sum(np. exp(z - maximum),
axis=1).reshape(z.shape[0], 1)

    def fit(self, X_train, y_train, lr=0.0001, limit=5000, normalize=False,
reg=0, lr_schedule=[1, 1], early_stop=False, verbose=False):
        """
        lr_schedule: first is LR_DECAY, second is LR_STEP
        """
        if normalize:
            self.stats = (X_train.mean(axis=0), X_train.std(axis=0))
            X_train = (X_train - self.stats[0]) / self.stats[1]

        m = X_train.shape[0]
        X = np.hstack((np.ones((m, 1)), X_train))
        k = len(np.unique(y_train))
        Y = np.zeros((m, k))
```

```
        lr_base = lr
        best_loss = np.inf
        best_count = 0

        for cls in np.unique(y_train).astype(int):
            Y[np.where(y_train[:] == cls), cls] = 1

        i = 0
        self.loss_arr = []

        self.theta = np.zeros((k, X.shape[1]))
        total = X.shape[0]
        for i in range(limit):
            if i % lr_schedule[1] == 0:
                lr = lr_base * lr_schedule[0] ** (i // lr_schedule[1])
            lineq = np.dot(X, self.theta.T)
            h = self.softmax(lineq)

            #Cost function
            epsilon = 1e-5
            loss = -np.sum(Y * np.log(h + epsilon)) / m + 0.5 * reg *
np.sum(self.theta * self.theta)
            if loss < best_loss:
                best_loss = loss
            else:
                best_count += 1
            self.loss_arr.append(loss)

            #gradient descent
            delta = (lr / m) * np.dot((h - Y).T, X) + reg * self.theta
            self.theta -=  delta
            if verbose and i % 50 == 0:
                print("Finish {}/{}, loss = {}, lr = {}".format(i + 1, limit,
loss, lr))
            if early_stop and best_count > 10:
                print("Early stop success at iteration {} with loss =
{}".format(i + 1, loss))
                break
            i = i + 1

        return self.loss_arr

    def predict(self, X_test):
        m_test = X_test.shape[0]
        X_test = (X_test - self.stats[0]) / self.stats[1]
        X_test = np.hstack((np.ones((m_test,1)),X_test))

        probab = self.softmax(np.dot(X_test,self.theta.T))
        predict = np.argmax(probab, axis=1)

        return predict
```

I tried the following parameters and finally found the best parameters are

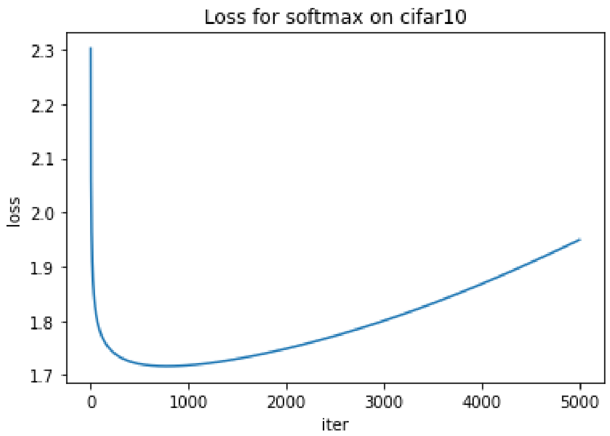
- lr = 0.01

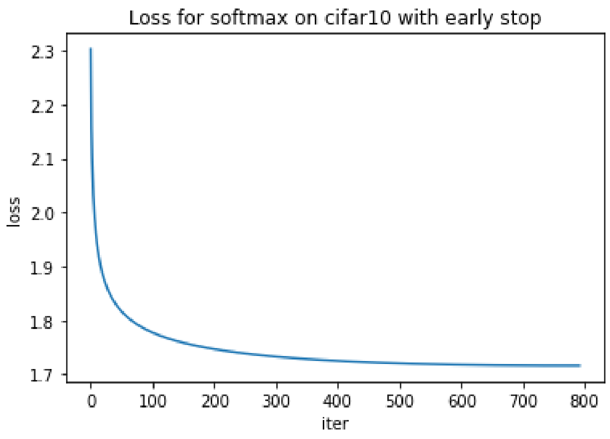
- lr schedule = No
- regulation = 0.0001

And the test accuracy is 0.4166

| lr | lr schedule | regulation | early stop | train loss | test accuracy | iteration |
|---|---|---|---|---|---|---|
| 0.01 | [0.99, 10] | 0.001 | No | 1.945 | 0.3575 | 5000 |
| 0.01 | [0.99, 10] | 0.001 | Yes | 1.716 | 0.4078 | 792 |
| 0.01 | No | 0.001 | Yes | 1.679 | 0.4136 | 5000 |
| 0.01 | No | No | Yes | 1.608 | 0.4155 | 5000 |
| 0.01 | No | 0.0001 | Yes | 1.619 | 0.4166 | 5000 |
| 0.01 | No | 0.01 | Yes | 1.794 | 0.3926 | 2507 |
| 0.01 | [0.99, 100] | No | Yes | 1.616 | 0.416 | 5000 |
| 0.02 | [0.99, 1] | No | Yes | 1.734 | 0.4042 | 2787 |

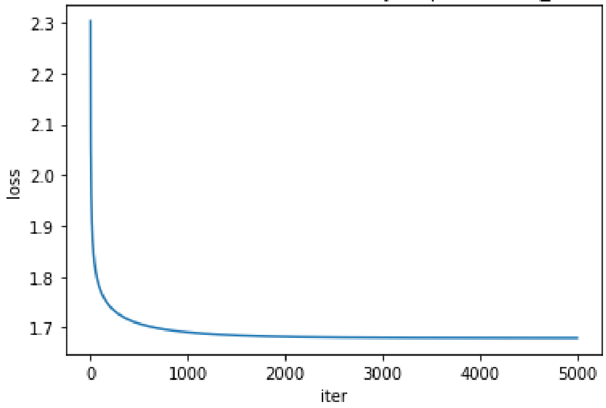
Then the following I shows the loss v.s. iteration in the order of the parameters listed behind



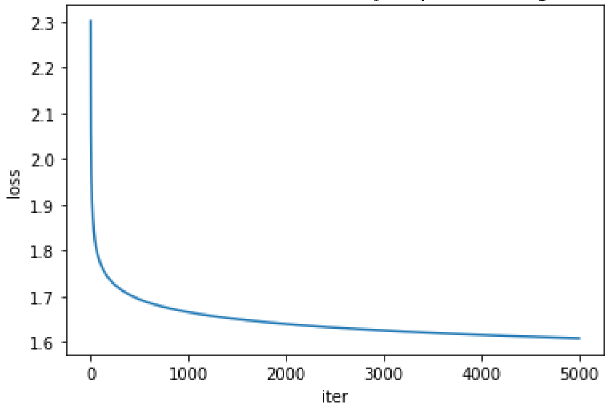




From the first picture we can know that the model is overfitted, so I wrote early stop to prevent this

Loss for softmax on cifar10 with early stop without lr_schedule

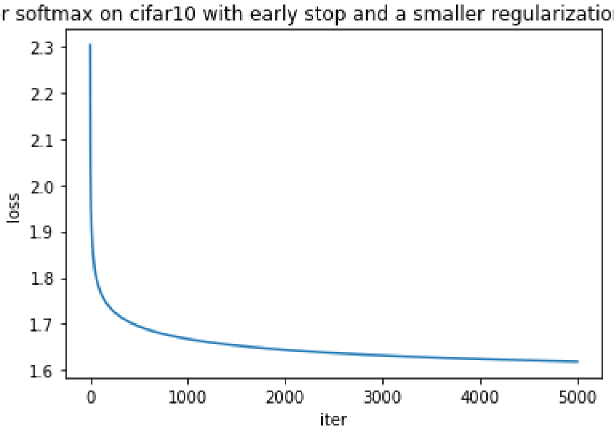



Loss for softmax on cifar10 with early stop without regularization

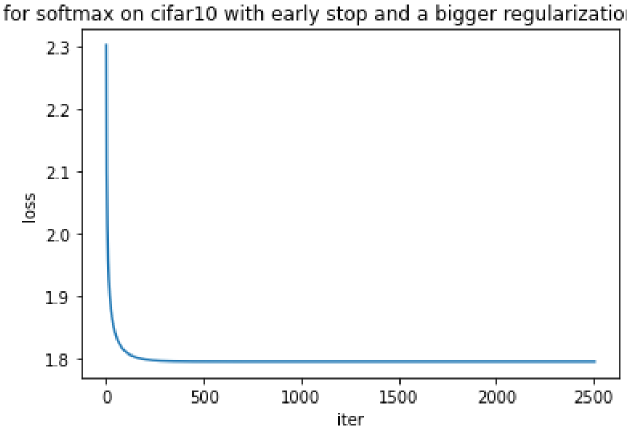


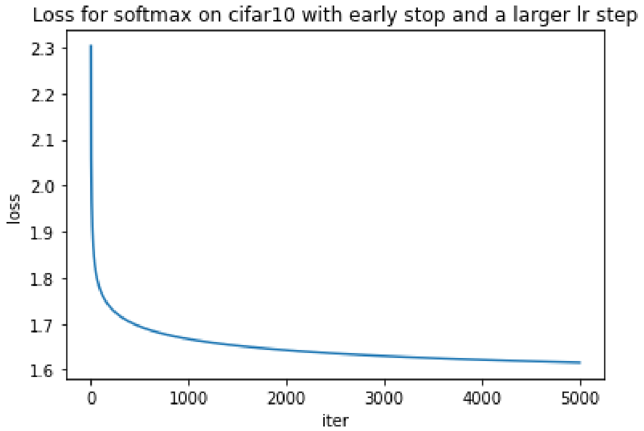

We can find that if we don't use learning rate schedule, the model will not be overfitted early as before, I think this may caused by the smaller learning rate

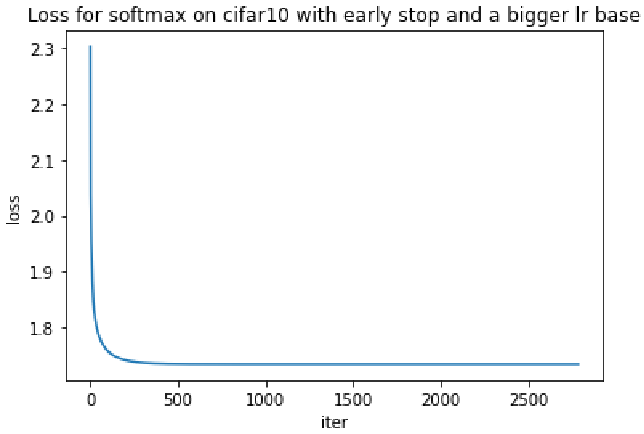
Loss for softmax on cifar10 with early stop and a smaller regularization parameter




Loss for softmax on cifar10 with early stop and a bigger regularization parameter




Loss for softmax on cifar10 with early stop and a larger lr step
loss
iter
2.3
2.2
2.1
2.0
1.9
1.8
1.7
1.6
0
1000
2000
3000
4000
5000


Loss for softmax on cifar10 with early stop and a bigger lr base
loss
iter
2.3
2.2
2.1
2.0
1.9
1.8
0
500
1000
1500
2000
2500