Hand In 1

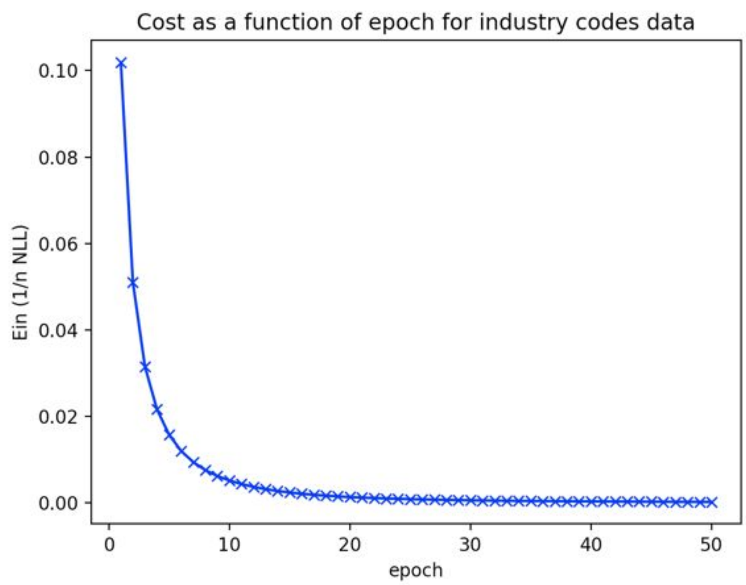
Zhuyun Zhou 201801015
Jing Yin Ong   201800935

PART I: Logistic Regression

Code

1. Summary and Results

As shown in the diagram below, the E_{in} decreases as the epoch increases because for each epoch, the w is updated to decrease the error.




```
vars args {'lr': -1, 'batch_size': -1, 'epochs': -1}
Logistic Regression Industri Codes Classifier
0.9743068391866914
In Sample Score:  0.9743068391866914
0.9628603104212861
Test Score:  0.9628603104212861
```

2. Actual Code

```
def cost_grad(self, X, y, w):
      cost = 0
      grad = np.zeros(w.shape)
       ### YOUR CODE HERE 5 - 15 lines
       for i, j in zip(X, y):
             cost = cost + np.sum([j * np.log(self.sigma(np.dot(w.T, i))), (1
- j) * np.log(1 - self.sigma(np.dot(w.T, i)))])
      cost = -cost / len(X)
      grad = -np.matmul(X.T, (y - self.sigma(np.matmul(X, w)))) / len(X)
       ### END CODE
       assert grad.shape == w.shape
       return cost, grad

def fit(self, X, y, w=None, lr=0.1, batch_size=16, epochs=10):
```

```
if w is None: w = np.zeros(X.shape[1])
history = []
### YOUR CODE HERE 14 - 20 lines
batches = int(len(y) / batch_size) + 1
for i in range(epochs):
        counter = 0
        for j in range(batches):
        batchX = X[counter:counter + batch_size]
        batchy = y[counter:counter + batch_size]
        cost,grad = self.cost_grad(batchX, batchy, w)
        w = w - lr * grad
        counter = counter + batch_size
       history.append(cost)
### END CODE
self.w = w
self.history = history
```

Theory

1. Runtime

Time to compute cost: O((d*1*d) * n * n) = $\mathbf{O(n^2 * d^2)}$
Time to compute gradient: O1(n*d*1) = O1(n*d), O(d*n*O1) = O(d*n*d) = $\mathbf{O(n * d^2)}$

2. Sanity Check

If we randomly permute the pixels in each image (with the same permutation) before we train the classifier, we will get a worse classifier than the one uses raw data.
Reason: The location of pixels relative to each other hold information of characteristics of cats and dogs, the base of our classification. A random permutation of all pixels' position affect and ruin this locality. The model we use exploit pixel locality, like the visualization of the softmax model applied to digits.

3. Linear Separable Data

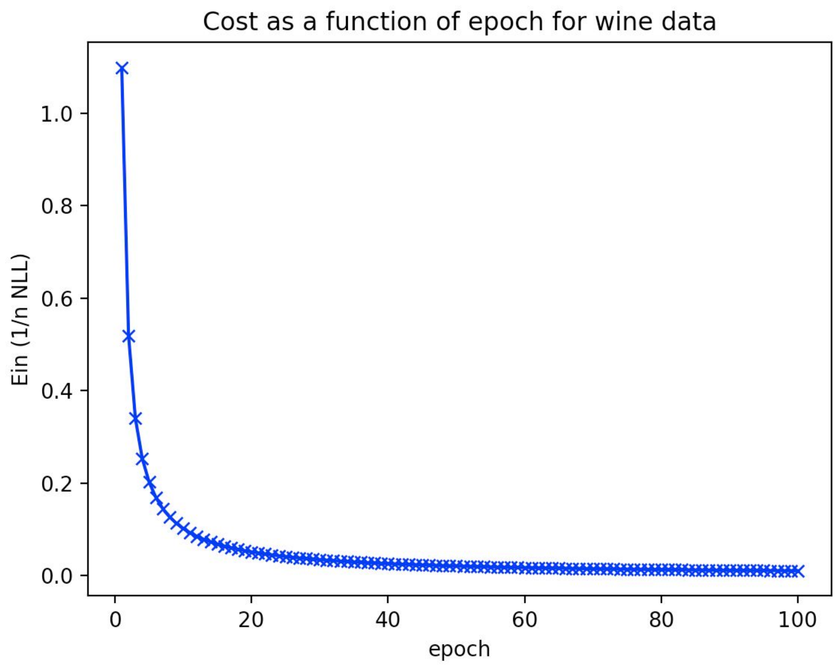
If the data is linearly separable, when we implement logistic regression with gradient descent, the weights that minimize the negative log likelihood tend to be more and more stable when we run the data.
They converge to some fixed number (fluctuate around it), instead of keeping increasing in magnitude (absolute value).
Reason: If data (features, X) is linearly separable, there exists a set of straight lines determined by weights (w) that can separate perfectly the predefined targets (y) into 2 groups. And when we implement logistic regression with gradient descent, every move from w(t) to w(t+1) is a move "in the right direction".

PART II: Softmax Regression

Code

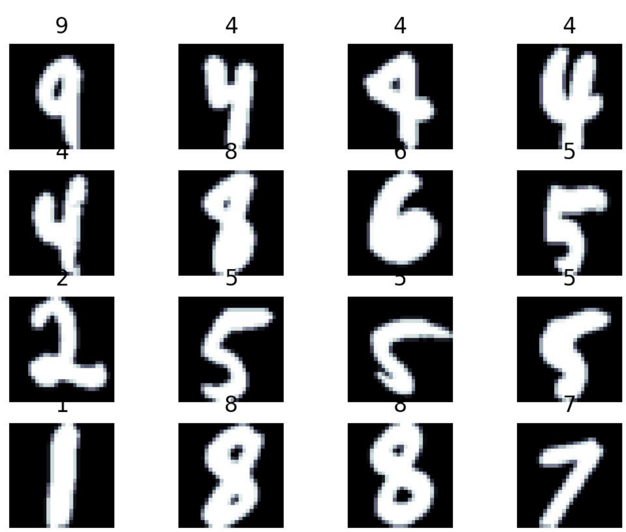
1. Summary and Results

    python softmax_test.py -wine -epochs 100 -lr 0.42 -bs 666



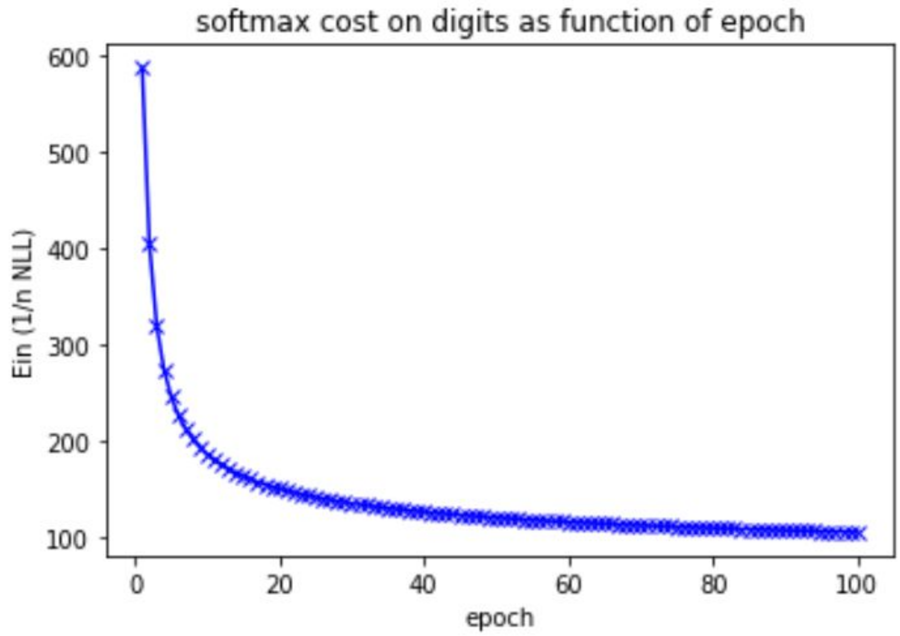

```
vars args {'wine': True, 'digits': False, 'visualize': False, 'show_digits': False, 'lr': 0.42,
'batch_size': 666, 'epochs': 100}
wine test: params - epochs 100, batch_size: 666, learning rate: 0.42
Softmax Wine Classifier
In Sample Score:  1.0
Test Score:  0.9192546583850931
```

python softmax_test.py -show_digits -epochs 100 -lr 0.42 -bs 666




```
vars args {'wine': False, 'digits': False, 'visualize': False, 'show_digits': True, 'lr': 0.42,
'batch_size': 666, 'epochs': 100}
shape of input data (10380, 784)
labels shape and type (10380,) int64
shape of input data (10380, 784)
labels shape and type (10380,) int64
```

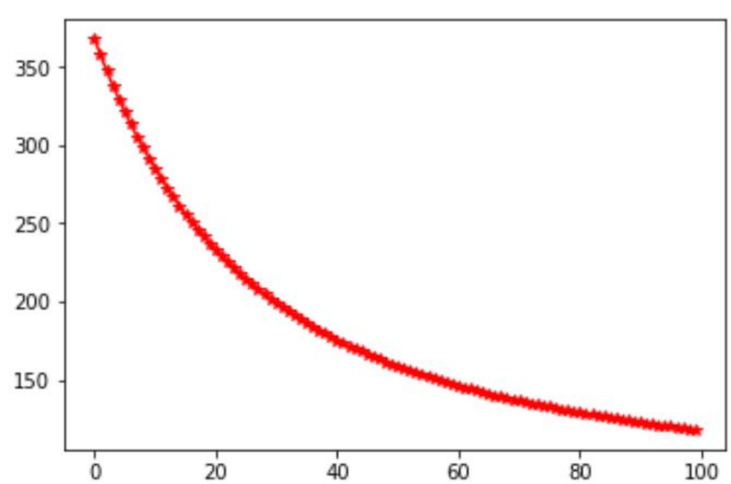
python softmax_test.py -digits -epochs 100 -lr 0.05 -bs 32



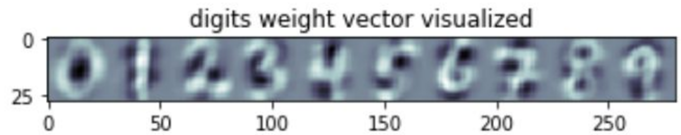

```
1  ## digits_test(epochs=10, batch_size=32, lr=0.05)
2  digits_test(epochs=100, batch_size=32, lr=0.05)
```

```
digits test: params - epochs 100, batch_size: 32, learning rate: 0.05
shape of input data (10380, 784)
labels shape and type (10380,) int64
shape of input data (2580, 784)
labels shape and type (2580,) int64
In Sample Score:  0.9017341040462428
Test Score:  0.9003875968992248
```

python softmax_test.py -visualize -epochs 100 -lr 0.01 -bs 64



```
shape of input data (10380, 784)
labels shape and type (10380,) int64
```



2. Actual Code

```
def cost_grad(self, X, y, W):
        cost = np.nan
        grad = np.zeros(W.shape)*np.nan
        Yk = one_in_k_encoding(y, self.num_classes) # may help - otherwise you may remove it
        ### YOUR CODE HERE
        cost = 0
        for i, j in zip(X, Yk):
            log_softmax = np.log(softmax(np.dot(i.T, W).reshape((1,W.shape[1]))))
            c = np.dot(log_softmax, j)
            cost += c
        cost = -cost / len(X)
        grad = -np.matmul(X.T, (Yk - softmax(np.matmul(X, W)))) / len(X)
        ### END CODE
        return cost, grad
```

```
    def fit(self, X, Y, W=None, lr=0.01, epochs=30, batch_size=16):
        if W is None: W = np.zeros((X.shape[1], self.num_classes))
        history = []
        ### YOUR CODE HERE
        batches = int(len(Y) / batch_size) + 1
        for i in range(epochs):
            counter = 0
            for j in range(batches):
                batchX = X[counter:counter + batch_size]
                batchy = Y[counter:counter + batch_size]
                cost,grad = self.cost_grad(batchX, batchy, W)
                W = W - lr * grad
                counter = counter + batch_size
            history.append(cost)
        ### END CODE
        self.W = W
        self.history = history
```

Theory

1. Runtime

Time to compute cost: O1(1*d*K) = O1(d*K), O2(K*1*O1) = O2(K*d*K) = O2(d*K^2)

O(O2 * n) = O(d*K^2*n) = **O(n*d*K^2)**

Time to compute gradient: O((d*n)*O1(n*d*K)) = O(d*n*d*K) = **O(n*d^2*K)**