CS 559 Homework 3

Stephen Szemis

December 11, 2020

I pledge my honor that I have abided by the Stevens honor system.

Problem 1: K-Mean

1.

```
iteration :  1
center of RED is [5.171 3.171]
```

2.

```
iteration :  2
center of GREEN is [5.3 4.0]
```

3.

```
iteration :  3
center of BLUE is [6.2   3.025]
```

4. The centers converge after 2 iterations.

Problem 2: K-Means and gradient descent

1. Starting with our loss function.

$$L = \sum_{j=1}^{k} \sum_{x_i \in S_j} \parallel x_i - u_j \parallel^2$$

We then take the gradient with respect to u_1 in order to derive the update formula. Remember that since we are using batch gradient decent, we will make use of all of our data.

$$\frac{\partial L}{\partial u_1} = \frac{\partial}{\partial u_1} \sum_{x_i \in S_j} \parallel x_i - u_1 \parallel^2 + \sum_{x_i \in S_j} \parallel x_i - u_2 \parallel^2 + \cdots + \sum_{x_i \in S_j} \parallel x_i - u_i \parallel^2$$

$$\frac{\partial L}{\partial u_1} = 2 \sum_{x_i \in S_j} (x_i - u_1) + 0 + \cdots + 0$$

$$\frac{\partial L}{\partial u_1} = -2 \sum_{x_i \in S_j} (x_i - u_1)$$

So the update rule is.

$$\hat{\mu_1} = \mu_1 + 2\epsilon \sum_{x_i \in S_j} (x_i - u_1)$$

2. This is very similar to batch gradient decent, except we now only use one piece of data randomly selected. The gradient is the same as before, but our update rule changes to.

$$\hat{\mu_1} = \mu_1 + 2\epsilon (x_i - u_1)$$

For some x_i randomly selected at each update step.

3. In order to create the standard update step, we need to rearrange our batch gradient decent update rule as follows.

$$\hat{\mu_1} = \mu_1 + 2\epsilon \sum_{x_i \in S_j} (x_i - u_1)$$

$$\hat{\mu_1} = \mu_1 + 2\epsilon \left(\sum_{x_i \in S_j} (x_i) - (\parallel S_j \parallel) \mu_1 \right)$$

Make $\epsilon = \frac{1}{2\|S_j\|}$ and we get.

$$\hat{\mu_1} = \mu_1 + 2\frac{1}{2 \parallel S_j \parallel} \left(\sum_{x_i \in S_j} (x_i) - (\parallel S_j \parallel)\mu_1 \right)$$

Simplify:

$$\hat{\mu_1} = \mu_1 - \mu_1 + \frac{1}{\parallel S_j \parallel} \sum_{x_i \in S_j} (x_i)$$

$$\hat{\mu_1} = \frac{1}{\parallel S_j \parallel} \sum_{x_i \in S_j} (x_i)$$

Which is the exact k-means update rule.

Problem 3: Latent variable model and GMM

1.

$$p(\mathbf{z}) = \prod_{k=1}^{K} \pi_k^{z_k}$$

$$p(\mathbf{x}|\mathbf{z}) = \prod_{k=1}^{K} \mathcal{N}(\mathbf{x}|\mu_k, \Sigma_k)^{z_k}$$

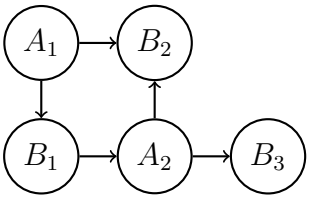
2. We notice that because z is using a 1-of-K representation, we can say that only 1 value of k in the above products will be non-zero. Therefore...

$$p(\mathbf{x}) = \sum_{\mathbf{z}} p(\mathbf{z})p(\mathbf{x}|\mathbf{z}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}|\mu_k, \Sigma_k)$$

3. We can use the Expectation Maximization Algorithm in order to solve our latent variable model. This is different from k-means because it uses a soft assignment, that is, each data point is not associated uniquely with one cluster. Instead they are associated with multiple clusters according to there posterior probabilities. This allows the Algorithm to be more accurate in some cases, however it can also be slower and harder to implement.

Problem 4: Bayesian Network

1. The graph looks like the following.



2.

$$p(A_1, A_2, B_1, B_2, B_3) = p(B_3|A_2)p(A_2|B_1)p(B_1|A_1)p(B_2|A_1, A_2)p(A_1)$$

3. We need one independent parameter in order to fully specify the joint probability. Since ultimately every is dependent on A_1.

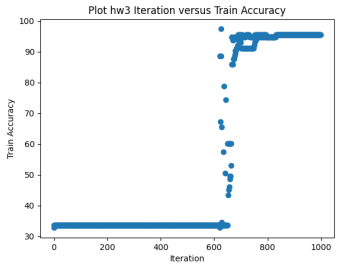
4. In this example we will use this factorization.

$$p(A_1, A_2, B_1, B_2, B_3) = p(B_3|B_1)p(B_2|B_1)p(B_1|A_1, A_2)p(A_1)p(A_2)$$

And therefore we need 2 independent parameters.

Problem 5: Neural Network

1. You can view the full code at the end of this document. I had two sigmoid activated layers with a hidden node number of 6. This gave okay results, but it seemed very dependent on the initial values of our weights. Often the model would be stuck in what seemed to be a steady state of some sort, unable to correctly "learn" from our data. I have a feeling this is because of the Sigmoid activation, which really isn't all that good at distinguishing between multiple classes. It's good for a binary classification problem, but we had three classes to learn. I ended up with accuracy in the area of 0.9. You can view the graphs of our accuracy below.



2. I used Keras from tensorflow to create my learning network. I'll explain the reasoning behind my final code. First I used three different layers, a RELU layer with 32 nodes, another RELU layer with 16 nodes, and a softmax layer with 3 nodes. The RELU layers are mostly there to give "space" to learn. I found that adding more nodes did increase my accuracy up to a point, this seemed like a good number. I choose RELU because it's easy to compute and it give the benefits of a linear activation layer without the issues of negative weights. The softmax layer simply gives a percent chance for our three classes. So it output a vector of probabilities that our input is either class A, B, or C. This is very helpful when doing multi-classification problems.

We also use a keras function which turns our "class labels" into a binary vectors. So...

$$0 => [1,0,0] 1 => [0,1,0] 2 => [0,0,1]$$

This really just so that our keras function can easily map our final predictions easily. We got around 97 percent accuracy, so it's much MUCH better then my simple network.

Code for part 1:

```
# Author: Stephen Szemis
# Pledge: I pledge my honor that I have abided by the Stevens honor system. - St
# Date: December, 9, 2020

import numpy as np

np.set_printoptions(precision=3)

data = np.array([[5.9, 3.2], [4.6, 2.9], [6.2, 2.8], [4.7, 3.2], [5.5, 4.2], [5.0,
                 [4.9, 3.1], [6.7, 3.1], [5.1, 3.8], [6.0, 3.0]])

cluster_colors = ["RED", "GREEN", "BLUE"]
cluster_centers = np.array([[6.2, 3.2], [6.6, 3.7], [6.5, 3.0]])

def update_centers():
    k = np.shape(cluster_centers)[0]
    distances = np.zeros((len(data), len(cluster_centers)))
    # Get sets for each cluster
```

```
    for i, x in enumerate(data):
        for j, y in enumerate(cluster_centers):
            distances[i][j] = np.sqrt(np.sum((x - y) ** 2))
    cluster_assignments = np.argmin(distances, axis=1)

    # Recompute Cluster means
    for i in range(k):
        t = data[cluster_assignments == i]
        cluster_centers[i] = np.sum(t, axis=0) / len(t)

for i in range(4):
    update_centers()
    print("iteration : ", i+1)
    print("center of RED is", cluster_centers[0, :])
    print("center of GREEN is", cluster_centers[1, :])
    print("center of BLUE is", cluster_centers[2, :])
    print("-------------------------------------------------")
```

Code for part 5:

```
# Author: Stephen Szemis
# Pledge: I pledge my honor that I have abided by the Stevens honor system. - St
# Date: December, 9, 2020

# Note: This code is based of off the code we saw and discussed in class.
# See link: https://dev.to/shamdasani/build-a-flexible-neural-network-with-backp

import numpy as np
import matplotlib.pyplot as plt
import random as rand

# Used for our final question
from keras import models
from keras import layers
from keras.utils import to_categorical

def split_data(X, Y, test_percent=0.3):
    testX = []
```

```
    trainX = []
    testY = []
    trainY = []
    for i, y in enumerate(Y[0]):
        num = rand.random()
        if (num < test_percent):
            testX.append(X[i])
            testY.append(y)
        else:
            trainX.append(X[i])
            trainY.append(y)
    return np.array(testX), np.atleast_2d(np.array(testY)), np.array(trainX), np.a

# Grab our iris data
def get_data():
    X = []
    Y = []
    f = open('iris.data')
    for line in f:
        z = line.strip().split(',')
        temp = [float(x) for x in z[:-1]]
        if (temp != []):
            X.append(temp)
            if (z[-1] == 'Iris-virginica'):
                Y.append(1)
            elif (z[-1] == 'Iris-versicolor'):
                Y.append(0.5)
            else:
                Y.append(0)
    return np.array(X), np.atleast_2d(np.array(Y))

class Neural_Network(object):
    def __init__(self):
        # parameters
        self.inputSize = 4
        self.outputSize = 1
        self.hiddenSize = 6
```

```
        # weights
        # (4x6) weight matrix from input to hidden layer
        self.W1 = np.random.randn(self.inputSize, self.hiddenSize)
        # (6x1) weight matrix from hidden to output layer
        self.W2 = np.random.randn(self.hiddenSize, self.outputSize)

    def forward(self, X):
        # forward propagation through our network
        # dot product of X (input) and first set of 3x2 weights
        self.z = np.dot(X, self.W1)
        self.z2 = self.sigmoid(self.z)  # activation function
        # dot product of hidden layer (z2) and second set of 3x1 weights
        self.z3 = np.dot(self.z2, self.W2)
        o = self.sigmoid(self.z3)  # final activation function
        return o

    def sigmoid(self, s):
        # activation function
        return 1/(1+np.exp(-s))

    def sigmoidPrime(self, s):
        # derivative of sigmoid
        return s * (1 - s)

    def backward(self, X, y, o):
        # backward propgate through the network

        self.o_error = np.subtract(y.T, o)  # error in output

        # applying derivative of sigmoid to error
        self.o_delta = self.o_error*self.sigmoidPrime(o)

        # z2 error: how much our hidden layer weights contributed to output erro
        self.z2_error = self.o_delta.dot(self.W2.T)
        # applying derivative of sigmoid to z2 error
        self.z2_delta = self.z2_error*self.sigmoidPrime(self.z2)

        # adjusting first set (input --> hidden) weights
```

```
        self.W1 += X.T.dot(self.z2_delta)
        # adjusting second set (hidden --> output) weights
        self.W2 += self.z2.T.dot(self.o_delta)

    def train(self, X, y):
        o = self.forward(X)
        self.backward(X, y, o)


def accuracy(o, Y):
    acc = 0
    for i, sample in enumerate(o):
        if (sample < 0.33) and (Y[0][i] == 0):
            acc += 1
        elif (sample > 0.66) and (Y[0][i] == 1):
            acc += 1
        elif (sample < 0.66) and (sample > 0.33 ) and (Y[0][i] == 0.5):
            acc += 1
    return (acc / len(o)) * 100

def run_NN():
    X, Y = get_data()
    X = X/np.amax(X, axis=0)  # maximum of X array
    testX, testY, trainX, trainY = split_data(X, Y)

    NN = Neural_Network()
    loss = []
    test_accuracy = []
    train_accuracy = []
    # Test and Train loop
    for i in range(1000):
        # mean sum squared loss
        train_accuracy.append(accuracy(NN.forward(trainX), trainY))
        test_accuracy.append(accuracy(NN.forward(testX), testY))
        NN.train(trainX, trainY)

    fig = plt.figure()
    ax = fig.add_subplot(111)
```

```
    ax.scatter(range(len(train_accuracy)), train_accuracy)

    # Produce Graph
    ax.set_title("Plot hw3 Iteration versus Train Accuracy")
    ax.set_xlabel('Iteration')
    ax.set_ylabel('Train Accuracy')
    fig.savefig('hw3_train.png')

    fig = plt.figure()
    ax = fig.add_subplot(111)
    ax.scatter(range(len(test_accuracy)), test_accuracy)

    # Produce Graph
    ax.set_title("Plot hw3 Iteration versus Test Accuracy")
    ax.set_xlabel('Iteration')
    ax.set_ylabel('Test Accuracy')
    fig.savefig('hw3_test.png')

def run_keras_model():
    X, Y = get_data()
    X = X/np.amax(X, axis=0)  # maximum of X array
    testX, testY, trainX, trainY = split_data(X, Y)

    network = models.Sequential()
    network.add(layers.Dense(32, activation='relu', input_shape=(4,)))
    network.add(layers.Dense(16, activation='relu'))
    # network.add(layers.Dense(16, activation='relu'))
    network.add(layers.Dense(3, activation='softmax'))

    network.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=

    # Transform our output into binary vectors
    train_labels = to_categorical(trainY[0], num_classes=3)
    test_labels = to_categorical(testY[0], num_classes=3)

    network.fit(trainX, train_labels, epochs=50)
    test_loss, test_acc = network.evaluate(testX, test_labels)
    print('test_acc:', test_acc)
```

```
    print('test_loss:', test_loss)

run_keras_model()
```