Rizu Jain
UIN 430000753
Date: 08 September 2020

CSCE 636: Deep Learning (Fall 2020)

Assignment #1: Linear Models for Handwritten Digits Classification

Report

1. Data Preprocessing

    a. The function *train_valid_split* splits the data into training set and validation set. We need this to tune the hyperparameters of our model

    b. Yes, before testing we can train the model on all the data. The purpose of a separate validation set is to find out the optimal hyperparameters. Once we have them, we can train the model with all the data.

    c. The features have been implemented in the code:

```python
def prepare_X(raw_X):
    """Extract features from raw_X as required.

    Args:
        raw_X: An array of shape [n_samples, 256].

    Returns:
        X: An array of shape [n_samples, n_features].
    """
    raw_image = raw_X.reshape((-1, 16, 16))

    # Feature 1: Measure of Symmetry
    ### YOUR CODE HERE
    flip_X = np.flip(raw_image,2)
    pixel_sym_diff = raw_image - flip_X
    abs_sym_diff = np.sum(np.abs(pixel_sym_diff),(1,2))
    f_sym = -1 * abs_sym_diff / 256
    ### END YOUR CODE

    # Feature 2: Measure of Intensity
    ### YOUR CODE HERE
    f_intensity = np.sum(raw_image,(1,2))/256
    ### END YOUR CODE

    # Feature 3: Bias Term. Always 1.
    ### YOUR CODE HERE
    f_bias = np.ones(raw_image.shape[0])
    ### END YOUR CODE

    # Stack features together in the following order.
    # [Feature 3, Feature 1, Feature 2]
    ### YOUR CODE HERE
    X = np.stack((f_bias,f_sym,f_intensity),1)
    ### END YOUR CODE
    return X
```

d. Logistic Regression is modeled using the equation:

$$h(x) = \theta \sum_{i=1}^{d} w_i x_i + w_0 x_0$$

intercept value to make model general $(x_0 = 1)$

$$\equiv \quad y = \theta\left(\sum_{i=0}^{d} w_i x_i\right)$$

$$= \theta(w^T x)$$

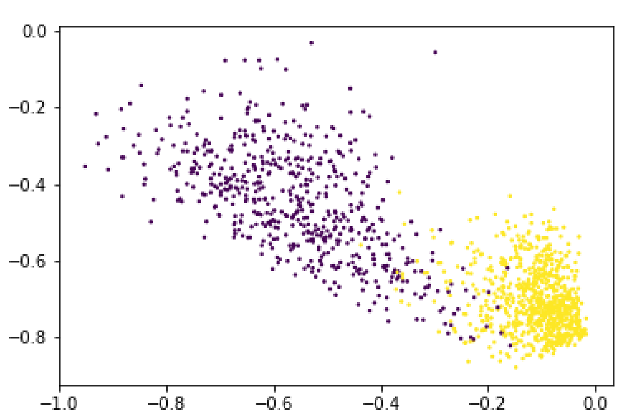
By adding the extra bias unit (always equal to 1) we are able to write the equation using dot product between w^T and x.

e. Noted.

f. The code for *visualize_features* has been implemented:

```
def visualize_features(X, y):
    '''This function is used to plot a 2-D scatter plot of training features.

    Args:
        X: An array of shape [n_samples, 2].
        y: An array of shape [n_samples,]. Only contains 1 or -1.

    Returns:
        No return. Save the plot to 'train_features.*' and include it
        in submission.
    '''
    ### YOUR CODE HERE
    plt.scatter(X[:,0],X[:,1],2,y)
    ### END YOUR CODE
```

The 2-D scatter plot (not including the third feature) looks like below:



Rizu Jain
UIN 430000753 Date: 08 September 2020

2. Cross-entropy loss

a.

For one training sample (x, y)

$$\boxed{E(w) = \ln\left(1 + e^{-y w^T x}\right)}$$

b.

Gradient $\nabla E(w)$:

$$= \frac{\partial}{\partial w}\left[\ln\left(1 + e^{-y w^T x}\right)\right]$$

$$= \frac{1}{1 + e^{-y w^T x}} \cdot \frac{\partial}{\partial W}\left(1 + e^{-y w^T x}\right)$$

$$= -(yx)\frac{1}{1 + e^{-A}} \cdot e^{-A}$$

where
$A = y w^T x$

$$= -(yx)\frac{1}{1 + e^{A}}$$

$$= \boxed{\frac{-yx}{1 + e^{y w^T x}}}$$

c. Sigmoid function is used to transform the output into a range of [0,1] so that we can model the certainty of an event. The output of a logistic regression model is interpreted as probability of the event.
We use this sigmoid function because it has a soft threshold which is easier for gradient calculation used in cross entropy loss minimization to optimize the model.

d.

Prediction Rule :

$$\theta(w^T x) \geq 0.9 \qquad \text{(for class 1)}$$

$$\theta(w^T x) < 0.9 \qquad \text{(for class -1)}$$

Here 0.9 is the threshold.

$$\theta(w^T x) - 0.9 = 0$$

$$\theta(w^T x) = 0.9$$

Since $\theta(s) = \dfrac{1}{1+e^{-s}}$,

$$\theta(w^T x) = \frac{1}{1+e^{-w^T x}} = 0.9$$

$$\Rightarrow \quad w^T x = -\ln(0.11)$$

This would still represent a linear decision boundary.

e. In the feature space, decision boundary can separate the input features that belong to one class from the others that do not belong to that class. This makes the feature space linearly separable.
It is evident by the fact that the hypothesis function still uses a linear combination of features, which makes the decision boundary ending up being linear.

3. Sigmoid logistic regression

a. *_gradient* function implementation:

```
    def _gradient(self, _x, _y):
        """Compute the gradient of cross-entropy with respect to self.W
        for one training sample (_x, _y). This function is used in fit_*.

        Args:
            _x: An array of shape [n_features,].
            _y: An integer. 1 or -1.

        Returns:
            _g: An array of shape [n_features,]. The gradient of
                cross-entropy with respect to self.W.
        """
        ### YOUR CODE HERE
        _g = - _y * _x / ( 1 + math.exp(_y * np.dot(self.W,_x)))
        return _g
        ### END YOUR CODE
```

b. *fit_GD* implementation:

```
def fit_GD(self, X, y):
    """Train perceptron model on data (X,y) with GD.

    Args:
        X: An array of shape [n_samples, n_features].
        y: An array of shape [n_samples,]. Only contains 1 or -1.

    Returns:
        self: Returns an instance of self.
    """
    n_samples, n_features = X.shape

    ### YOUR CODE HERE
    self.W = np.zeros(n_features)

    for _ in range(self.max_iter):
        grad = 0
        for i in range(0,n_samples):
            grad += self._gradient(X[i],y[i])

        grad = grad/n_samples
        self.W -= self.learning_rate * grad

    ### END YOUR CODE
    return self
```

fit_SGD implementation:

```
def fit_SGD(self, X, y):
    """Train perceptron model on data (X,y) with SGD.

    Args:
        X: An array of shape [n_samples, n_features].
        y: An array of shape [n_samples,]. Only contains 1 or -1.

    Returns:
        self: Returns an instance of self.
    """
    ### YOUR CODE HERE
    n_samples, n_features = X.shape
    self.W = np.zeros(n_features)
    for _ in range(self.max_iter):
        for i in range(0,n_samples):
            grad = self._gradient(X[i],y[i])
            self.W -= self.learning_rate * grad
    ### END YOUR CODE
    return self
```

fit_BGD implementation:

```
def fit_BGD(self, X, y, batch_size):
    """Train perceptron model on data (X,y) with BGD.

    Args:
        X: An array of shape [n_samples, n_features].
        y: An array of shape [n_samples,]. Only contains 1 or -1.
        batch_size: An integer.

    Returns:
        self: Returns an instance of self.
    """
    ### YOUR CODE HERE
    n_samples, n_features = X.shape
    self.W = np.zeros(n_features)
    for _ in range(self.max_iter):
        for i in range(0, n_samples//batch_size):
            grad = 0
            for j in range(i * batch_size, (i+1) * batch_size):
                if j >= n_samples:
                    break
                grad += self._gradient(X[j],y[j])

            grad = grad/batch_size
            # print("LR gradients:",grad)
            self.W -= self.learning_rate * grad

    ### END YOUR CODE
    return self
```

c. *predict* implementation:

```
def predict(self, X):
    """Predict class labels for samples in X.

    Args:
        X: An array of shape [n_samples, n_features].

    Returns:
        preds: An array of shape [n_samples,]. Only contains 1 or -1.
    """
    ### YOUR CODE HERE
    z = np.dot(X, self.W)

    def sigmoid(p):
        return 1/(1+ math.exp(-p))

    ans = np.vectorize(sigmoid)(z)

    ans[ans>=0.5] = 1
    ans[ans<0.5] = -1

    return ans
```

score implementation:

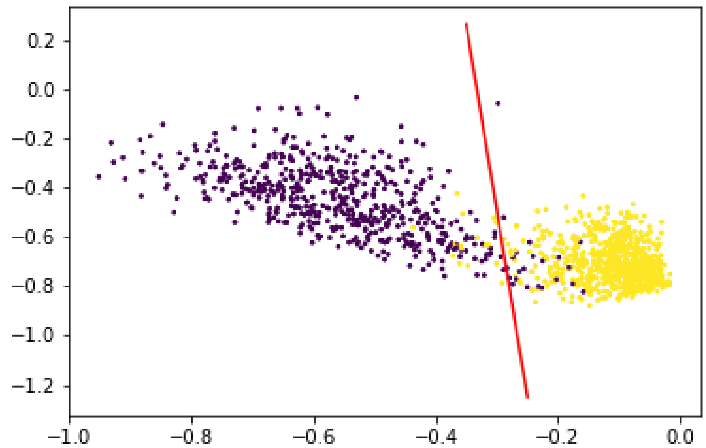
```
def score(self, X, y):
    """Returns the mean accuracy on the given test data and labels.

    Args:
        X: An array of shape [n_samples, n_features].
        y: An array of shape [n_samples,]. Only contains 1 or -1.

    Returns:
        score: An float. Mean accuracy of self.predict(X) wrt. y.
    """
    ### YOUR CODE HERE
    pred_y = self.predict(X)
    acc = sum(pred_y == y) / y.shape[0]
    return acc
    ### END YOUR CODE
```

predict_proba implementation:

```
def predict_proba(self, X):
    """Predict class probabilities for samples in X.

    Args:
        X: An array of shape [n_samples, n_features].

    Returns:
        preds_proba: An array of shape [n_samples, 2].
            Only contains floats between [0,1].
    """
    ### YOUR CODE HERE

    z = np.dot(X, self.W)

    def sigmoid(p):
        return 1/(1+ math.exp(-p))

    ans = np.vectorize(sigmoid)(z)
    return np.concatenate((np.reshape(ans,(-1,1)), np.reshape(1-ans,(-1,1))), axis = 1)

    ### END YOUR CODE
```

d. *visualize_results* implementation:

```
def visualize_result(X, y, W):
    '''This function is used to plot the sigmoid model after training.

    Args:
        X: An array of shape [n_samples, 2].
        y: An array of shape [n_samples,]. Only contains 1 or -1.
        W: An array of shape [n_features,].

    Returns:
        No return. Save the plot to 'train_result_sigmoid.*' and include it
        in submission.
    '''
    ### YOUR CODE HERE
    plt.scatter(X[:,0],X[:,1],2,y)
    x_points = np.linspace(-0.35,-0.25,100)
    y_points = - (W[0]/(1e-10 + W[2])) - x_points * (W[1]/(1e-10 + W[2]))
    plt.plot(x_points,y_points,'-r')
    plt.savefig('1_f_2D_Scatter_plot.png')

    ### END YOUR CODE
```



e. Testing process has been implemented and best logreg model's test accuracy reported:

```
# Use the 'best' model above to do testing. Note that the test data should be loaded and pro
### YOUR CODE HERE
test_raw_data, test_labels = load_data(os.path.join(data_dir, test_filename))
test_X_all = prepare_X(test_raw_data)
test_y_all, test_idx = prepare_y(test_labels)
test_X = test_X_all[test_idx]
test_y = test_y_all[test_idx]
test_y[np.where(test_y == 2)] = -1

print("----------------------------------------------")
print("Best model testing accuracy logistic regression: ", best_model.score(test_X,test_y))
print("----------------------------------------------")

### END YOUR CODE
```

```
----------------------------------------------
Best model testing accuracy logistic regression:
0.9264069264069265
----------------------------------------------
```

4. Softmax logistic regression

a. *_gradient* implementation:

```
def _gradient(self, _x, _y):
    """Compute the gradient of cross-entropy with respect to self.W
    for one training sample (_x, _y). This function is used in fit_*.

    Args:
        _x: An array of shape [n_features,].
        _y: One_hot vector.

    Returns:
        _g: An array of shape [n_features,]. The gradient of
            cross-entropy with respect to self.W.
    """
    ### YOUR CODE HERE
    h = self.softmax(np.dot(_x, self.W))
    grad = _x.reshape(-1,1) * (h - _y).reshape(1,-1)
    return grad
    ### END YOUR CODE
```

b. *Fit_BGD* implementation

```
def fit_BGD(self, X, labels, batch_size):
    """Train perceptron model on data (X,y) with BGD.

    Args:
        X: An array of shape [n_samples, n_features].
        labels: An array of shape [n_samples,].  Only contains 0,..,k-1.
        batch_size: An integer.

    Returns:
        self: Returns an instance of self.

    Hint: the labels should be converted to one-hot vectors, for example: 1--
    """

    ### YOUR CODE HERE
    import pandas as pd

    n_samples, n_features = X.shape
    y = pd.get_dummies(labels).values

    self.W = np.zeros((n_features, self.k))

    for _ in range(self.max_iter):
        for i in range(0, n_samples//batch_size):
            grad = 0
            for j in range(i * batch_size, (i+1) * batch_size):
                if j >= n_samples:
                    break
                grad += self._gradient(X[j],y[j])

            grad = grad/batch_size
            # print("LRM gradients:",grad)
            self.W -= self.learning_rate * grad

    ### END YOUR CODE
```

c. _predict implementation:

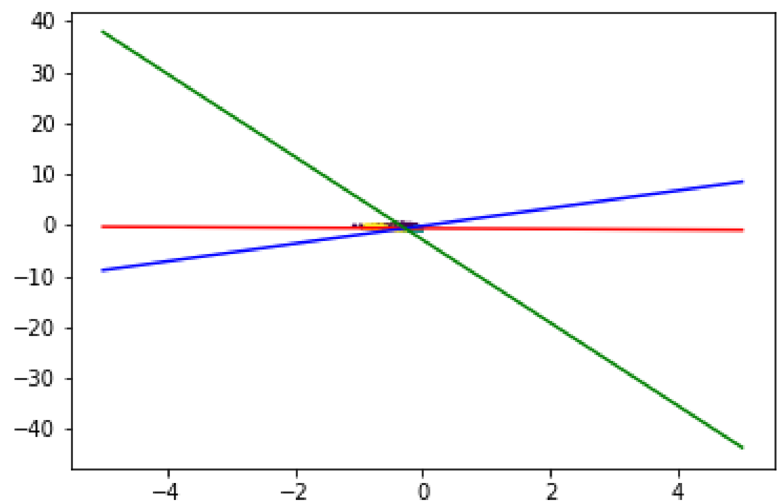
```
def predict(self, X):
    """Predict class labels for samples in X.

    Args:
        X: An array of shape [n_samples, n_features].

    Returns:
        preds: An array of shape [n_samples,]. Only contains 0,..,k-1.
    """
    ### YOUR CODE HERE
    pred_one_hot = np.dot(X,self.W)
    pred = np.argmax(pred_one_hot,axis=1)
    return pred
    ### END YOUR CODE
```

_*score* implementation:

```
def score(self, X, labels):
    """Returns the mean accuracy on the given test data and labels.

    Args:
        X: An array of shape [n_samples, n_features].
        labels: An array of shape [n_samples,]. Only contains 0,..,k-1.

    Returns:
        score: An float. Mean accuracy of self.predict(X) wrt. labels.
    """
    ### YOUR CODE HERE
    pred_y = self.predict(X)
    acc = sum(pred_y == labels) / labels.shape[0]
    return acc
    ### END YOUR CODE
```

d. *visualize_result_multi* implementation:

```
def visualize_result_multi(X, y, W):
    '''This function is used to plot the softmax model after training.

    Args:
        X: An array of shape [n_samples, 2].
        y: An array of shape [n_samples,]. Only contains 0,1,2.
        W: An array of shape [n_features, 3].

    Returns:
        No return. Save the plot to 'train_result_softmax.*' and include it
        in submission.
    '''
    ### YOUR CODE HERE
    W = W.T
    plt.figure()
    plt.scatter(X[:,0],X[:,1],2,y)
    x0_points = np.linspace(-5,5,100)
    y0_points = - (W[0,0]/W[0,2]) - x0_points * (W[0,1]/W[0,2])
    plt.plot(x0_points, y0_points,'-r')
    x1_points = np.linspace(-5,5,100)
    y1_points = - (W[1,0]/W[1,2]) - x1_points * (W[1,1]/W[1,2])
    plt.plot(x1_points, y1_points,'-b')
    x2_points = np.linspace(-5,5,100)
    y2_points = - (W[2,0]/W[2,2]) - x2_points * (W[2,1]/W[2,2])
    plt.plot(x2_points, y2_points,'-g')
    ### END YOUR CODE
```



e. Best logreg multiclass model test accuracy:

```
-----------------------------------------------
Best model testing accuracy multiclass :  0.8672350791717418
-----------------------------------------------
```

5. Softmax logistic vs Sigmoid logistic

   a. We have added functions **'fit_BGD_Convergence'** functions in LogisticRegression.py and LRM.py for comparing the results at convergence.

      i. Softmax implementation

```
###### First, fit softmax classifer until convergence, and evaluate
##### Hint: we suggest to set the convergence condition as "np.linalg.norm(gradients*1./batch_size) < 0.0005" c
### YOUR CODE HERE
logisticR_classifier_multiclass = logistic_regression_multiclass(learning_rate=0.5, max_iter=10000, k = 2)
logisticR_classifier_multiclass.fit_BGD_Convergence(train_X,train_y,10)
print(logisticR_classifier_multiclass.get_params())
print(logisticR_classifier_multiclass.score(train_X, train_y))

### END YOUR CODE
```

      ii. Sigmoid implementation

```
### YOUR CODE HERE
logisticR_classifier = logistic_regression(learning_rate=0.5, max_iter=10000)

logisticR_classifier.fit_BGD_Convergence(train_X, train_y, 10)
print(logisticR_classifier.get_params())
print(logisticR_classifier.score(train_X, train_y))

### END YOUR CODE
```

Results:

- Softmax:
  - Weights:
    [[-0.6960125 0.6960125], [-7.43512852 7.43512852], [2.86312768 -2.86312768]]
  - Accuracy:
    0.9666666666666667
- Sigmoid:
  - Weights:
    [1.01655638 14.83693967 -5.42109992]
  - Accuracy:
    0.9688888888888889

Observation:
Both the classifiers have given approximately similar results. However, the results are not the same. The softmax classifier has the same decision boundary for both the classes but in opposite directions. The sigmoid classifier's boundary is close to that.

b. Comparison of learning between sigmoid and softmax:

For this, I have trained the models for 3 iterations and observed the gradients and the final weights. Here are the results:

Logistic Regression: Learning rate = 1

Gradients

Epoch 1**: [-0.08074074, -0.08442919, 0.11061595]**

Epoch 2: **[-0.05034237, -0.09129272, 0.09091999]**

Epoch 3: **[-0.03113024, -0.09491502, 0.07801487]**

Weights:

[0.16221335 0.27063692 -0.27955081]

Accuracy:

0.5866666666666667

Softmax regression: Learning rate = 0.5

Gradients

Epcoh 1: [[0.08074074 -0.08074074], [0.08442919 -0.08442919], [-0.11061595 0.11061595]]

Epcoh 2: [[0.05034237 -0.05034237], [0.09129272 -0.09129272], [-0.09091999 0.09091999]]

Epoch 3: [[0.03113024 -0.03113024], [0.09491502 -0.09491502], [-0.07801487 0.07801487]]

Weights [w1 w2]:

[[-0.08110667 0.08110667], [-0.13531846 0.13531846], [0.13977541 -0.13977541]]

Weights [w2 – w1]:

[0.16221335 0.27063692 -0.27955081]

Accuracy

0.5866666666666667

Conclusion:
The softmax regression has the same gradients as the sigmoid regression. However, softmax applies it to both the set of weights and hence has twice the impact than that of sigmoid.
If we counter that with the learning rate i.e. if we train sigmoid logistic regression at twice the learning rate of softmax logistic regression model's learning rate, we get the same gradients and same weights after same number of epochs.
Thus, to maintain w = w2 – w1, learning rate for sigmoid should be twice that of softmax.