Documentation

Logistic Regression Models

1. Logistic Regression

```
class LogisticRegression:
    """
    This class implements Logistic Regression using both
    Batch Gradient Descent and Stochastic Gradient descent.
    Class attributes:
        W        : current set of weights
        W_arr    : list of weights at each iteration
        Cost     : current cost
        cost_arr : list of costs at each iteration
    """

    def sigmoid(self,x):
        """
        Sigmoid activation function.
        returns the sigmoid of the given input.
        """
        return 1/(1+np.exp(-x))

    def get_cost(self, X, y, W):
        """
        This function returns the cost with the given set of weights
        using the formula
```

$$J = \frac{1}{2m} \sum_{i=0}^{m} (y^i \log(h_w(x^i)) + (1 - y^i)(1 - \log(1 - h_w(x^i)))$$

```
        """
        total_cost = 0
        for i in range(X.shape[0]):
            total_cost += y[i]*np.log(self.get_h_i(X, i, W)) + (1-y[i])*np.log(1-self.get_h_i(X, i, W))
        return (0.5/X.shape[0])*total_cost

    def get_h_i(self, X, i, W):
        """
        This function returns the hypothesis of ith feature vector
        with the given weights W.
```

$$h_w(x^i) = sigmoid(\sum_{j=0}^{n} w_j {x^i}_j) = sigmoid(x^i w)$$

```
        """
        h_i = np.matmul(X[i].reshape(1,-1),W)
        return self.sigmoid(h_i[0][0])
```

```
def batch_grad_descent(self, X, y, alpha, max_iter):
    """
    This function implements the Batch Gradient Descent algorithm.
    It runs for multiple iterations until either the weights converge or
    iterations reach max_iter. At each iteration the weights are updated using
    the following rule
        repeat until convergence{
```

$$w_j^{t+1} = w_j^t - \alpha \sum_{i=1}^{m} (y^i(1 - h_w(x^i)) - (1 - y^i)h_w(x^i))x_j^i$$

```
    """
    W_new = self.W.copy()
    for _ in range(max_iter):
        grad = np.zeros((X.shape[0],1))
        for i in range(X.shape[0]):
            grad[i] = (-y[i]*(1-self.get_h_i(X, i, self.W)) + (1-y[i])*self.get_h_i(X, i, self.W))
        for j in range(X.shape[1]):
            W_new[j][0] = self.W[j][0] - (alpha/X.shape[0])*np.sum(grad*X[:,j:j+1].reshape(-1,1))
        self.W = W_new.copy()
        self.cost_arr.append(self.get_cost(X, y, self.W))
        self.W_arr.append(self.W)
        if len(self.W_arr)>1:
            if sum(abs(self.W_arr[-2]-self.W_arr[-1]))<0.0001:
                break
    return W_new
```

```
def stochastic_grad_descent(self, X, y, alpha, max_iter):
    """
    This function implements the Stochastic Gradient Descent algorithm.
    It runs for multiple iterations until either the weights converge or
    iterations reach max_iter. Weights are updated for every row of the
    training set.

        repeat until convergence{
            randomly shuffle the feature matrix rows
            for each feature vector x^i {
                update all weights j -> 0 to n+1
```

$$w_j^{t+1} = w_j^t - \alpha(y^i(1 - h_w(x^i)) - (1 - y^i)h_w(x^i))x_j^i$$

```
            }
        }
    """
    mat = np.concatenate((X,y.reshape(-1,1)), axis=1)
    for _ in range(max_iter):
        W_new = self.W.copy()
        np.random.shuffle(mat)
        X = mat[:,0:3]
        y = mat[:,3]
        for i in range(X.shape[0]):
            grad = (-y[i]*(1-self.get_h_i(X, i, self.W)) + (1-y[i])*self.get_h_i(X, i, self.W))
            for j in range(X.shape[1]):
                W_new[j][0] = self.W[j][0] - (alpha)*(grad[0]*X[i,j])
            self.W = W_new.copy()
        self.cost_arr.append(self.get_cost(X, y, self.W))
        self.W_arr.append(self.W)
        if len(self.W_arr)>1:
            if sum(abs(self.W_arr[-2]-self.W_arr[-1]))<0.0001:
                break
    return self.W
```

```python
if __name__ == "__main__":
    model = LogisticRegression()

    # data input
    data = pd.read_excel("./data3.xlsx",header=None)
    data = data.sample(frac=1).reset_index(drop=True)
    X = data[[0,1,2,3]]
    y = data[4]-1

    # data preprocessing (Normal scaling)
    mscaler = NormalScaler()
    for j in range(X.shape[1]):
        mscaler.fit(X.loc[:,j])
        X.loc[:,j] = mscaler.transform(X.loc[:,j])

    # holdout cross validation split
    train_percent = 0.6
    X_train = X[:int(train_percent*X.shape[0])]
    y_train = y[:int(train_percent*X.shape[0])]
    X_test = X[int(train_percent*X.shape[0]):]
    y_test = y[int(train_percent*X.shape[0]):]
```

```python
    # Training the model by choosing alpha and max_iter values.
    # gradient descent algorithm can be set as either ‘batch’ or ‘stochastic’
    # in this function call.
    alpha = 0.26
    max_iter = 100
    algo = 'batch'
    model.train(X_train.values,y_train.values,alpha,max_iter,algo)

# Testing on train set
    print("\nTraining..")
    y_pred = model.test(X_train.values)
    for i in range(y_pred.shape[0]):
        y_pred[i] = 0 if y_pred[i]<0.5 else 1

    print('\n',y_pred)
    print("\nTraining set accuracy: ",sum(y_pred==y_train)/y_train.shape[0])
    print("Training set sensitivity: ",sum((y_pred==1) & (y_train==1))/sum(y_train==1))
    print("Training set specificity: ",sum((y_pred==0) & (y_train==0))/sum(y_train==0))

# Testing on test set
    print("\nTesting...")
    y_pred = model.test(X_test.values)
    for i in range(y_pred.shape[0]):
        y_pred[i] = 0 if y_pred[i]<0.5 else 1

    print('\n',y_pred)
    print("\nTesting set accuracy: ",sum(y_pred==y_test)/y_test.shape[0])
    print("Training set sensitivity: ",sum(y_pred*y_test)/sum(y_test))
    print("Training set specificity: ",sum((y_pred==0) & (y_test==0))/sum(y_test==0))
```

Results:

Training data

[0. 1. 0. 0. 0. 0. 1. 0. 0. 0. 1. 0. 1. 1. 0. 0. 0. 0. 0. 1. 1. 1. 1. 0. 1. 1. 0. 0. 1. 1. 1. 0. 0. 1. 1. 1. 0. 0. 1. 1. 0. 0. 0. 0. 1. 1. 0. 1. 1. 1. 1. 1. 1. 0. 1. 0. 1. 0. 0. 1.]

Training set accuracy: 1.0

Training set sensitivity: 1.0

Training set specificity: 1.0

Testing predictions

[0. 1. 0. 0. 1. 0. 1. 1. 0. 1. 1. 0. 0. 0. 1. 0. 1. 0. 1. 1. 0. 1. 1. 0. 1. 1. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1. 1. 0. 1. 1.]

Testing set accuracy: 0.975

Training set sensitivity: 1.0

Training set specificity: 0.95

2. One vs All Classifier

```
from LogisticRegression import LogisticRegression,NormalScaler

if __name__ == "__main__":
    # data input
    data = pd.read_excel("./data4.xlsx",header=None)
    data = data.sample(frac=1).reset_index(drop=True)

    X = data[[i for i in range(7)]]
    y = data[7]

    unique_classes = np.unique(y)
    num_classes = len(unique_classes)

    # data preprocessing
    mscaler = NormalScaler()
    for j in range(X.shape[1]):
        mscaler.fit(X[j])
        X[j] = mscaler.transform(X[j])
```

```
    y_cat = (y==unique_classes[0]).astype('int').values.reshape(-1,1)
    for i in unique_classes[1:]:
        y_cat = np.concatenate((y_cat,(y==i).astype('int').values.reshape(-1,1)),axis=1)

    # splitting data using holdout cross validation
    train_percent = 0.6
    X_train = X[:int(train_percent*X.shape[0])]
    y_train = y[:int(train_percent*X.shape[0])]
    y_cat_train = y_cat[:int(train_percent*X.shape[0])]
    X_test = X[int(train_percent*X.shape[0]):]
    y_test = y[int(train_percent*X.shape[0]):]
    y_cat_test = y_cat[int(train_percent*X.shape[0]):]
```

```python
# creating a Logistic regression model for each class
models = [LogisticRegression() for i in unique_classes]

y_train_pred = np.ndarray((y_train.shape[0],num_classes))
y_test_pred = np.ndarray((y_test.shape[0],num_classes))
for c in range(num_classes):
    # training
    models[c].train(X_train,y_cat_train[:,c],0.26,100,'batch')
    y_train_pred[:,c] = models[c].test(X_train)

    # testing
    y_test_pred[:,c] = models[c].test(X_test)
    y_p = (y_test_pred[:,c]>0.5)
    print("Class ",unique_classes[c]," Accuracy = ", sum(y_p==(y_test==unique_classes[c]))/(X_test.shape[0]))
```

```python
y_train_t = np.argmax(y_train_pred, axis=1)+1
y_test_t = np.argmax(y_test_pred, axis=1)+1

print("Train Accuracy : ",sum(y_train_t==y_train)/y_train.shape[0])
print("Test Accuracy : ",sum(y_test_t==y_test)/y_test.shape[0])

# Confusion Matrix
conf_mat = np.ndarray((num_classes, num_classes))
for i in range(num_classes):
    for j in range(num_classes):
        conf_mat[i][j] = sum((y_test_t==unique_classes[i]) & (y_test==unique_classes[j]))

print(conf_mat)
```

Results:

Class 1 Accuracy = 1.0

Class 2 Accuracy = 0.783

Class 3 Accuracy = 0.833

Train Accuracy : 0.933

Test Accuracy : 0.8167

Confusion Matrix

```
[[17.  0.  0.]
 [ 0. 13.  2.]
 [ 0.  9. 19.]]
```

3. One vs One Classifier

```
from LogisticRegression import LogisticRegression,NormalScaler

if __name__ == "__main__":
    model = LogisticRegression()

    # data input
    data = pd.read_excel("./data4.xlsx",header=None)
    data = data.sample(frac=1).reset_index(drop=True)

    X = data[[i for i in range(7)]]
    y = data[7]

    # data preprocessing
    mscaler = NormalScaler()
    for j in range(X.shape[1]):
        mscaler.fit(X.loc[:,j])
        X.loc[:,j] = mscaler.transform(X.loc[:,j])

    unique_classes = np.unique(y)
    num_classes = len(unique_classes)
    num_models = (int)(num_classes*(num_classes-1)/2)
```

```
    # splitting data using holdout cross validation
    train_percent = 0.6
    X_train = X[:int(train_percent*X.shape[0])]
    y_train = y[:int(train_percent*X.shape[0])]
    X_test = X[int(train_percent*X.shape[0]):]
    y_test = y[int(train_percent*X.shape[0]):]

    models = [[0 for j in range(num_classes)] for i in range(num_classes)]

    y_test_pred = np.ndarray((y_test.shape[0], num_models))
    k = 0
    # training and testing n(n-1)/2 models
    for i in range(num_classes-1):
        for j in range(i+1, num_classes):
            class_i = unique_classes[i]
            class_j = unique_classes[j]

            models[i][j] = LogisticRegression()
            tmp = (y_train==class_i) | (y_train==class_j)
            y_train_i_j = (y_train[tmp]==class_i).astype('int').values
            models[i][j].train(X_train[tmp], y_train_i_j, 0.1, 100, 'batch')

            y_test_pred[:,k] = models[i][j].test(X_test)
            y_test_pred[:,k][y_test_pred[:,k]>=0.5] = class_i
            y_test_pred[:,k][y_test_pred[:,k]<0.5] = class_j
            acc = sum(y_test_pred[:,k]==y_test)/y_test.shape[0]
            print("{0} vs {1} Accuracy: {2}".format(i+1,j+1,acc))
            k+=1
```

```python
    # calculating overall accuracy
    y_test_t = np.ndarray((y_test.shape[0],))
    for i in range(y_test.shape[0]):
        uniqu,counts = np.unique(y_test_pred[i],return_counts=True)
        y_test_t[i] = uniqu[np.argmax(counts)]
    print("\nOverall Accuracy: ", sum(y_test_t==y_test)/y_test.shape[0])
```

Results:

1 vs 2 Accuracy: 0.63

1 vs 3 Accuracy: 0.63

2 vs 3 Accuracy: 0.65

Overall Accuracy: 0.933

4. One vs All using K-Fold Cross Validation

```python
from LogisticRegression import LogisticRegression,NormalScaler

def predictOneVsAll(X_train, y_train, X_test, y_test, unique_classes):
    num_classes = len(unique_classes)
    models = [LogisticRegression() for i in unique_classes]

    y_train_pred = np.ndarray((y_train.shape[0],num_classes))
    y_test_pred = np.ndarray((y_test.shape[0],num_classes))
    for c in range(num_classes):
        models[c].train(X_train,y_cat_train[:,c],0.26,100,'batch')
        y_train_pred[:,c] = models[c].test(X_train)
        y_test_pred[:,c] = models[c].test(X_test)
        y_p = (y_test_pred[:,c]>0.5)
        print("Class ",unique_classes[c]," Accuracy = ", sum(y_p==(y_test==unique_classes[c]))/(X_test.shape[0]))

    y_train_t = np.argmax(y_train_pred, axis=1)+1
    y_test_t = np.argmax(y_test_pred, axis=1)+1

    test_acc = sum(y_test_t==y_test)/y_test.shape[0]
    print("Train Accuracy : ",sum(y_train_t==y_train)/y_train.shape[0])
    print("Test Accuracy : \n",test_acc)

    # Confusion Matrix
    conf_mat = np.ndarray((num_classes, num_classes))
    for i in range(num_classes):
        for j in range(num_classes):
            conf_mat[i][j] = sum((y_test_t==unique_classes[i]) & (y_test==unique_classes[j]))

    print(conf_mat,"\n")
    return test_acc
```

```python
if __name__ == "__main__":
    # data input
    data = pd.read_excel("./data4.xlsx",header=None)
    data = data.sample(frac=1).reset_index(drop=True)

    X = data[[i for i in range(7)]]
    y = data[7]

    unique_classes = np.unique(y)
    num_classes = len(unique_classes)

    # data preprocessing
    mscaler = NormalScaler()
    for j in range(X.shape[1]):
        mscaler.fit(X[j])
        X[j] = mscaler.transform(X[j])

    y_cat = (y==unique_classes[0]).astype('int').values.reshape(-1,1)
    for i in unique_classes[1:]:
        y_cat = np.concatenate((y_cat,(y==i).astype('int').values.reshape(-1,1)),axis=1)
```

```python
    k = 5
    N = X.shape[0]
    j = 0
    acc = 0
    # splitting data using k fold cross validation approach
    for i in range(0,k):
        X_train = np.concatenate((X[:i*(N//k)],X[(i+1)*(N//k):]))
        y_train = np.concatenate((y[:i*(N//k)],y[(i+1)*(N//k):]))
        y_cat_train = np.concatenate((y_cat[:i*(N//k)],y_cat[(i+1)*(N//k):]))
        X_test = X[i*(N//k):(i+1)*(N//k)]
        y_test = y[i*(N//k):(i+1)*(N//k)]
        y_cat_test = y_cat[i*(N//k):(i+1)*(N//k)]
        acc += predictOneVsAll(X_train, y_train, X_test, y_test, unique_classes)
    print("Average Accuracy: \n", acc/k)
```

Results:

<u>K-fold-1</u>

Class 1 Accuracy = 1.0

Class 2 Accuracy = 0.9333333333333333

Class 3 Accuracy = 0.8333333333333334

Train Accuracy : 0.8916666666666667

Test Accuracy : 0.9333333333333333

Confusion Matrix

[[9. 0. 0.]
 [0. 9. 0.]
 [0. 2. 10.]]

K-fold-2

Class 1 Accuracy = 1.0

Class 2 Accuracy = 0.6666666666666666

Class 3 Accuracy = 0.8666666666666667

Train Accuracy : 0.9166666666666666

Test Accuracy : 0.8666666666666667

Confusion Matrix

[[10. 0. 0.]
 [1. 7. 1.]
 [0. 2. 9.]]

K-fold-3

Class 1 Accuracy = 1.0

Class 2 Accuracy = 0.6333333333333333

Class 3 Accuracy = 0.8333333333333334

Train Accuracy : 0.9166666666666666

Test Accuracy : 0.8

Confusion Matrix

[[9. 0. 0.]
 [0. 7. 2.]
 [0. 4. 8.]]

K-fold-4

Class 1 Accuracy = 1.0

Class 2 Accuracy = 0.7666666666666667

Class 3 Accuracy = 0.9666666666666667

Train Accuracy : 0.9083333333333333

Test Accuracy : 0.9333333333333333

Confusion Matrix

[[14. 0. 0.]
 [0. 5. 1.]
 [0. 1. 9.]]

K-fold-5

Class 1 Accuracy = 1.0

Class 2 Accuracy = 0.8666666666666667

Class 3 Accuracy = 0.9333333333333333

Train Accuracy : 0.9083333333333333

Test Accuracy : 0.9666666666666667

Confusion Matrix

[[7. 0. 0.]
 [0. 12. 0.]
 [0. 1. 10.]]

Average Accuracy: 0.9