Kernelized Logistic Regression

November 3, 2018

1 Logistic Regression

Given a set of inputs X, we want to assign them to one of two possible categories (0 or 1). Logistic regression models the probability that each input belongs to a particular category.

1.0.1 Hypothesis

Logistic regression uses the sigmoid function that gives probability output between 0 and 1 for all values of X.

$h(\theta, X) = sigmoid(\theta^T X) = \frac{1}{1+e^{-\theta^T X}}$

1.0.2 Loss function

in logistic regression we learn the weights θ and we want to find the best values for them. To start we pick random values and we need a way to measure how well the algorithm performs using those random weights. That measure is computed using the loss function, defined as:

$J(\theta) = \frac{1}{m}(-y^T \log(h) - (1-y)^T \log(1-h))$

1.0.3 Gradient descent

The goal in logistic regression is to minimize the loss function by increasing/decreasing the weights θ fitting them with the data, to do that we use the derivative of the loss function with respect to each weight (i.e. the gradient). It tells us how loss would change if we modified the parameters. Then we update the weights by substracting to them the derivative times the learning rate α.

$\frac{\partial J(\theta)}{\partial \theta} = \frac{1}{m} X^T (h - y)$

$\theta = \theta - \alpha \frac{\partial J(\theta)}{\partial \theta}$

1.0.4 Prediction

to predict on new data we just need to check if the sigmoid probability is greater than 0.5 to predict class 1 otherwise 0.

1.0.5 Implementation

```
In [1]: import numpy as np
        from time import time

        def sigmoid(z):
            return 1 / (1 + np.exp(-z))

        def loss(h, y):
            return (-y * np.log(h) - (1 - y) * np.log(1 - h)).mean()

        # append new feature with value of ones (represents the bias for theta zero)
        def add_ones_feature(X):
            _X = np.ones((X.shape[0], X.shape[1] + 1))
            _X[:, 1:] = X
            return _X

        def fit(X, y, learn_rate=0.01, num_iter=100000):
            X = add_ones_feature(X)
            m = X.shape[0]
            d = X.shape[1]
            theta = np.zeros(d)

            for iteration in range(num_iter):
                h = sigmoid(np.dot(X, theta))
                gradient = np.dot(X.T, (h - y)) / m
                theta -= learn_rate * gradient

                if iteration % (num_iter // 10) == 0:
                    print(f'loss: {loss(h, y)}')

            return theta

        def predict(X, theta, threshold=0.5):
            X = add_ones_feature(X)
            return sigmoid(np.dot(X, theta)) >= threshold
```

1.0.6 Stochastic gradient decend

In stochastic gradient decend we calculate the gradient depending on one example at a time, the algorithm is altered as follows

```
In [2]: def fit_stochastic(X, y, learn_rate=0.01, num_iter=100000):
            X = add_ones_feature(X)
            m = X.shape[0]
            d = X.shape[1]
            theta = np.zeros(d)

            # reduce the number of iteration according to size of examples
```

```
        num_iter = num_iter // m
        for iteration in range(num_iter):
            for i in range(m): # loop through examples
                h = sigmoid(np.dot(X, theta))
                # calcualate the gradient depending on example i
                gradient = np.dot(X[i].T, (h[i] - y[i])) / m
                # decend
                theta -= learn_rate * gradient

            if iteration % (num_iter // 10) == 0:
                print(f'loss: {loss(h, y)}')

        return theta
```

1.0.7 Kernelized Logistic Regression

Logistic regression can be kernelized in the formula of the gradient as follows:

$Kmat = K(X_{train}, X_{train})$

$\frac{\partial J(\theta)}{\partial \theta} = \frac{1}{m} Kmat^T (h - y)$

$\theta = \theta - \alpha \frac{\partial J(\theta)}{\partial \theta}$

$h(\theta, X) = sigmoid(\theta^T K(X, X_{train})) = \frac{1}{1+e^{-\theta^T K(X, X_{train})}}$

In [3]:
```
def rbf_kernel(v1, v2, sigma=1.0):
    return np.exp((-1 * np.linalg.norm(v1 - v2) ** 2) / (2 * sigma ** 2))

def polynomial_kernel(v1, v2, c=1, d=2):
    return (v1.T.dot(v2) + c) ** d

def sigmoid_kernel(v1, v2, alpha=2, c=1):
    return np.tanh(alpha * v1.T.dot(v2) + c)

def compute_kmat(X, kernel):
    m = X.shape[0]
    kmat = np.zeros((m, m))
    for i in range(m):
        for j in range(m//2 + 1):
            kmat[i, j] = kmat[j, i] = kernel(X[i], X[j])
    return kmat

def compute_kmat2(X, train, kernel):
    m = X.shape[0]
    d = train.shape[0]
    kmat = np.zeros((m, d))
    for i in range(m):
        for j in range(d):
            kmat[i, j] = kernel(X[i], train[j])
    return kmat
```

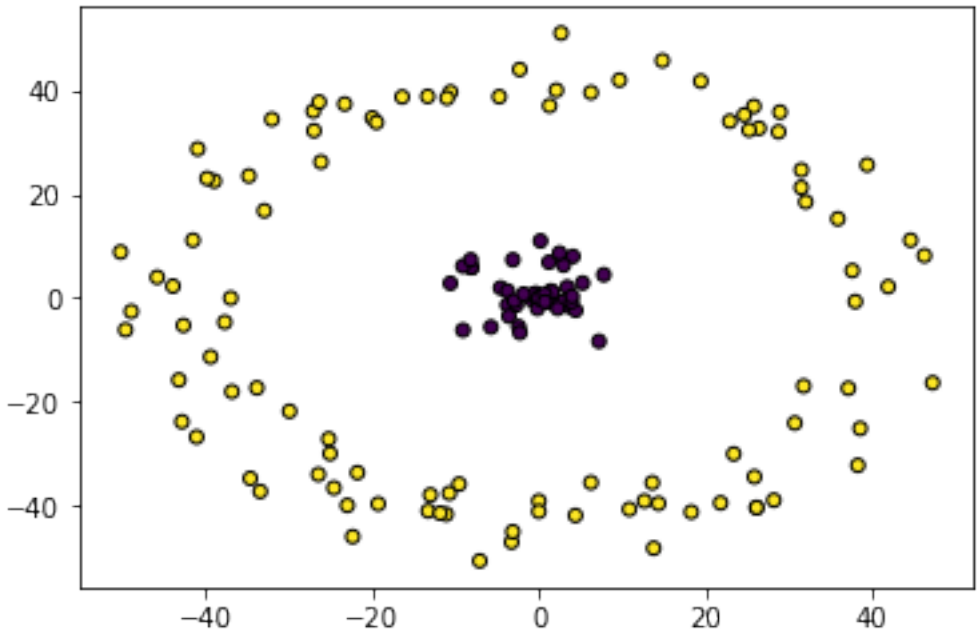
```
def k_fit(X, y, learn_rate=0.01, num_iter=100000, kernel=rbf_kernel):
    kmat = compute_kmat(X, kernel)
    kmat = add_ones_feature(kmat)
    m = kmat.shape[0]
    d = kmat.shape[1]
    theta = np.zeros(d)

    for iteration in range(num_iter):
        h = sigmoid(np.dot(kmat, theta))
        gradient = np.dot(kmat.T, (h - y)) / m
        theta -= learn_rate * gradient

        if iteration % (num_iter // 10) == 0:
            print(f'loss: {loss(h, y)}')

    return theta

def k_fit_stochastic(X, y, learn_rate=0.01, num_iter=100000, kernel=rbf_kernel):
    kmat = compute_kmat(X, kernel)
    kmat = add_ones_feature(kmat)
    m = kmat.shape[0]
    d = kmat.shape[1]
    theta = np.zeros(d)
    num_iter = num_iter // m
    for iteration in range(num_iter):
        for i in range(m):
            h = sigmoid(np.dot(kmat, theta))
            gradient = np.dot(kmat[i].T, (h[i] - y[i]))
            theta -= learn_rate * gradient

        if iteration % (num_iter // 10) == 0:
            print(f'loss: {loss(h, y)}')

    return theta

def k_predict(X, train, theta, threshold=0.5, kernel=rbf_kernel):
    kmat = compute_kmat2(X, train, kernel)
    kmat = add_ones_feature(kmat)
    return sigmoid(np.dot(kmat, theta)) >= threshold
```

1.0.8 Performance and accuracy tests

for testing we use a non linearly seperable data as follows: since the data is non linearly seperable, we gain better accuracy using the kernel method but more execution time since for calculating the kernel matrix $kmat$ and learning the weights θ of size m in the new feature space

```
In [6]: import matplotlib.pyplot as plt
        data = np.loadtxt('../data2.data')
        X = data[:,:2]
        y = data[:,2]
        plt.scatter(X[:,0], X[:,1], c=y , s=25, edgecolor='k')
```

```
Out[6]: <matplotlib.collections.PathCollection at 0x7f0ffa911e48>
```




```
In [5]: if __name__ == '__main__':
            data = np.genfromtxt('../data2.data', delimiter=' ')
            np.random.shuffle(data) # shuffle the examples
            train_size = len(data) * 60 // 100 # split into train/test
            X_train = data[:train_size,:2]
            y_train = data[:train_size,2]
            X_test = data[train_size:,:2]
            y_test = data[train_size:,2]

            print("logistic regression")
            start = time()
            theta = fit(X_train, y_train)
            p = predict(X_test, theta)
            print(f'accuracy: {int((p == y_test).mean() * 100)}%')
            print(f'execution time: {time() - start} sec')

            print("\nkernelized logistic regression")
            start = time()
```

```
theta = k_fit(X_train, y_train)
p = k_predict(X_test, X_train, theta)
print(f'accuracy: {int((p == y_test).mean() * 100)}%')
print(f'execution time: {time() - start} sec')

print("\nlogistic regression (stochastic GD)")
start = time()
theta = fit_stochastic(X_train, y_train)
p = predict(X_test, theta)
print(f'accuracy: {int((p == y_test).mean() * 100)}%')
print(f'execution time: {time() - start} sec')

print("\nkernelized logistic regression (stochastic GD)")
start = time()
theta = k_fit_stochastic(X_train, y_train)
p = k_predict(X_test, X_train, theta)
print(f'accuracy: {int((p == y_test).mean() * 100)}%')
print(f'execution time: {time() - start} sec')
```

```
logistic regression
loss: 0.6931471805599454
loss: 0.619326320925166
loss: 0.619326320925166
loss: 0.619326320925166
loss: 0.619326320925166
loss: 0.619326320925166
loss: 0.619326320925166
loss: 0.619326320925166
loss: 0.619326320925166
loss: 0.619326320925166
accuracy: 63%
execution time: 3.0068016052246094 sec

kernelized logistic regression
loss: 0.6931471805599454
loss: 0.3557513106520843
loss: 0.3096036916085649
loss: 0.28634628609401674
loss: 0.2722585317916767
loss: 0.2627847679921432
loss: 0.25591848284368135
loss: 0.25065718472643855
loss: 0.24645382637279642
loss: 0.24298765662718153
accuracy: 90%
execution time: 5.7770607471466064 sec

logistic regression (stochastic GD)
```

```
loss: 0.6940528211005405
loss: 0.664342294738323
loss: 0.6463898368070594
loss: 0.6359956563432866
loss: 0.6299235146407501
loss: 0.626345621706906
loss: 0.6242220036959032
loss: 0.622954423544964
loss: 0.6221948321324273
loss: 0.6217386409496074
loss: 0.6214645316844505
accuracy: 63%
execution time: 2.7731752395629883 sec

kernelized logistic regression (stochastic GD)
loss: 0.6581890258247205
loss: 0.3550137394351634
loss: 0.3092470475276124
loss: 0.2861286940092524
loss: 0.2721096464247646
loss: 0.26267413155629904
loss: 0.2558309386026887
loss: 0.2505845471955586
loss: 0.2463913830660107
loss: 0.24293254250062946
loss: 0.2400092685665866
accuracy: 90%
execution time: 3.9141945838928223 sec
```