hw1-solution

February 9, 2019

1 Cody Fizette

1.1 Homework 1

```
In [1]: from hw1_code import *
        import pandas as pd
        import numpy as np
        import matplotlib.pyplot as plt
        import matplotlib
        from sklearn.linear_model import LinearRegression
        from sklearn.model_selection import train_test_split
```

```
In [2]: matplotlib.rcParams['figure.figsize'] = [15, 10]
        np.random.seed(1337)
```

2 2. Mathamatical Fundamentals

2.1 2.1 Probability

Let $(X_1, X_2, \cdots, X_d)$ have a d-dimensional multivariate Gaussian distribution, with mean vector $\mu \in \mathbb{R}^{\mathrm{d}}$ and covariance matrix $\Sigma \in \mathbb{R}^{\mathrm{d}\times\mathrm{d}}$, i.e. $(X_1, X_2, \cdots, X_d) \sim \mathcal{N}(\mu, \Sigma)$. Use μ_i to denote the i^{th} element of μ and Σ_{ij} to denote the element at the i^{th} row and j^{th} column of Σ.

2.1.1 2.1.1

Let $x, y \in R^d$ be two independent samples drawn from $\mathcal{N}(\mu, \Sigma)$. Give expression for $E\|x\|_2^2$ and $E\|x-y\|_2^2$. Express your answer as a function of μ and Σ. $\|x\|_2$ represents the ℓ_2-norm of vector x.

$$\begin{aligned}
E\|x\|_2^2 &= E\left(\sum_{i=1}^{d} x_i^2\right), && (1)\\
&= \sum_{i=1}^{d} E(x_i^2), && (2)\\
&= \sum_{i=1}^{d} \left(\Sigma_{i,i} + \mu_i^2\right). && (3)
\end{aligned}$$

And

$$E\|x-y\|_2^2 = E\Big(\sum_{i=1}^{d}(x_i-y_i)^2\Big), \tag{4}$$
$$= \sum_{i=1}^{d}\Big(E(x_i)^2 - 2E(x_* y_i) + E(y_i)^2\Big), \tag{5}$$
$$= \sum_{i=1}^{d}\Big(E(x_i^2) - 2E(x_i^2) + E(x_i^2)\Big), \tag{6}$$
$$= 0. \tag{7}$$

2.1.2 2.1.2

Find the distribution of $Z = \alpha_i X_i + \alpha_j X_j$, for $i \neq j$ and $1 \leq i, j \leq d$. The answer will belong to a familiar class of distribution. Report the answer by identifying this class of distribution and specifying the parameters.

Z is normally distributed.

$Z \sim \mathcal{N}(\alpha_i\mu_i + \alpha_j\mu_j,\ \alpha_i^2\Sigma_{i,i} + \alpha_j^2\Sigma_{j,j} + 2\alpha_i\alpha_j\Sigma_{i,j})$

2.1.3 2.1.3

Assume W and R are two Gaussian distributed random variables. Is $W + R$ still Gaussian?

No. Proof:

Let $W \sim \mathcal{N}(\mu, \Sigma)$ Now let $R = -W$. Then it follows that $R \sim \mathcal{N}(\mu, \Sigma)$. However,

$$W + R = W - W,$$
$$= 0.$$

Thus $W + R$ is not Gaussian.

2.2 2.2 Linear Algebra

2.2.1 2.2.1

Let A be a $d \times d$ matrix with rank k. Consider the set $S_A := \{x \in R^d | Ax = 0\}$. What is the dimension of S_A?

$dim(S_A) = d - k$.

2.2.2 2.2.2

Assume S_v is a k dimensional subspace in R^d and $v_1, v_2, \cdots, v_k$ form an orthonormal basis of S_v. Let w be an arbitrary vector in R^d. Find

$$x^* = \underset{x \in S_v}{\operatorname{argmin}} \|w - x\|_2, \tag{8}$$

where $\|w - x\|_2$ is the Euclidean distance between w and x. Express x^* as a function of $v_1, v_2, \ldots, v_k$ and w

Solution

$$x^* = \underset{x \in S_v}{\operatorname{argmin}} \|w - x\|_2, \tag{9}$$
$$= proj_{s_v}(w), \tag{10}$$
$$= \sum_{i=1}^{d} \frac{w \cdot v_i}{v_i \cdot v_i} w_i \tag{11}$$

3 3. Linear Regression

3.1 3.1 Feature Normalization

```
########################################
### Feature normalization
def feature_normalization(train, test):
    """Rescale the data so that each feature in the training set is in
    the interval [0,1], and apply the same transformations to the test
    set, using the statistics computed on the training set.

    Args:
        train - training set, a 2D numpy array of size (num_instances, num_features)
        test - test set, a 2D numpy array of size (num_instances, num_features)

    Returns:
        train_normalized - training set after normalization
        test_normalized - test set after normalization
    """
    # Remove columns with constant values
    cols_to_delete = np.all(train==train[0,:], axis=0)
    cols_to_delete = np.argwhere(cols_to_delete==True)
    train = np.delete(train, cols_to_delete, 1)
    test = np.delete(test, cols_to_delete, 1)

    min_arr = np.amin(train, axis=0)
    range_arr = np.amax(train, axis=0) - min_arr

    train_normalized = (train-min_arr)/range_arr
    test_normalized = (test-min_arr)/range_arr

    return train_normalized, test_normalized
```

3.2 3.2 Gradient Descent Setup

3.2.1 3.2.1

$$J(\theta) = \frac{1}{m}(X^T\theta - y)^T(X^T\theta - y) \tag{12}$$

3.2.2 3.2.2

$$\nabla J(\theta) = \frac{2}{m}(X^T\theta - y)^T X \quad (13)$$

3.2.3 3.2.3

$$J(\theta + \eta h) - J(\theta) \approx J(\theta) + \eta h^T \nabla J(\theta) \quad (14)$$

3.2.4 3.2.4

$$\theta \leftarrow \theta - \eta \nabla J(\theta) \quad (15)$$

3.2.5 3.2.5

```python
#######################################
### The square loss function
def compute_square_loss(X, y, theta):
    """
    Given a set of X, y, theta, compute the average square loss for predicting y with X*theta.

    Args:
        X - the feature vector, 2D numpy array of size (num_instances, num_features)
        y - the label vector, 1D numpy array of size (num_instances)
        theta - the parameter vector, 1D array of size (num_features)

    Returns:
        loss - the average square loss, scalar
    """
    y_pred = np.matmul(X,theta)

    #return (1/len(y)) * np.matmul((y_pred - y).T, (y_pred - y))

    return np.mean(np.square(y - y_pred))
```

3.2.6 3.2.6

"'python ####################################### ### The gradient of the square loss function def compute_square_loss_gradient(X, y, theta): """ Compute the gradient of the average square loss (as defined in compute_square_loss), at the point theta.

```
Args:
    X - the feature vector, 2D numpy array of size (num_instances, num_features)
    y - the label vector, 1D numpy array of size (num_instances)
    theta - the parameter vector, 1D numpy array of size (num_features)

Returns:
```

```
    grad - gradient vector, 1D numpy array of size (num_features)
"""
y_pred = np.matmul(X, theta)
n = len(y)
return (2/n) * np.matmul(X.T, y_pred - y)
```

3.3 3.3 Gradient Checker

"'python ## ### Gradient checker def grad_checker(X, y, theta, epsilon=0.01, tolerance=1e-4): """Implement Gradient Checker Check that the function compute_square_loss_gradient returns the correct gradient for the given X, y, and theta.

```
Let d be the number of features. Here we numerically estimate the
gradient by approximating the directional derivative in each of
the d coordinate directions:
(e_1 = (1,0,0,...,0), e_2 = (0,1,0,...,0), ..., e_d = (0,...,0,1))

The approximation for the directional derivative of J at the point
theta in the direction e_i is given by:
( J(theta + epsilon * e_i) - J(theta - epsilon * e_i) ) / (2*epsilon).

We then look at the Euclidean distance between the gradient
computed using this approximation and the gradient computed by
compute_square_loss_gradient(X, y, theta).  If the Euclidean
distance exceeds tolerance, we say the gradient is incorrect.

Args:
    X - the feature vector, 2D numpy array of size (num_instances, num_features)
    y - the label vector, 1D numpy array of size (num_instances)
    theta - the parameter vector, 1D numpy array of size (num_features)
    epsilon - the epsilon used in approximation
    tolerance - the tolerance error

Return:
    A boolean value indicating whether the gradient is correct or not
"""
true_gradient = compute_square_loss_gradient(X, y, theta) #The true gradient
num_features = theta.shape[0]
approx_grad = np.zeros(num_features) #Initialize the gradient we approximate

hs = np.eye(num_features)

for i, h in enumerate(hs):
    approx_grad[i] = (compute_square_loss(X, y, theta + epsilon*h) - compute_square_loss(X, y, t

dist = np.linalg.norm(approx_grad - true_gradient)
```

```
return dist <= tolerance
```

```

“‘python ######################################## ### Generic gradient checker def generic_gradient_checker(X, y, theta, objective_func, gradient_func, epsilon=0.01, tolerance=1e-4): """ The functions takes objective_func and gradient_func as parameters. And check whether gradient_func(X, y, theta) returned the true gradient for objective_func(X, y, theta). Eg: In LSR, the objective_func = compute_square_loss, and gradient_func = compute_square_loss_gradient """ true_gradient = gradient_func(X, y, theta) #The true gradient num_features = theta.shape[0] approx_grad = np.zeros(num_features) #Initialize the gradient we approximate

```
hs = np.eye(num_features)

for i, h in enumerate(hs):
    approx_grad[i] = (objective_func(X, y, theta + epsilon*h) - objective_func(X, y, theta - eps

dist = np.linalg.norm(approx_grad - true_gradient)

return dist <= tolerance
```

```

3.3.1 Some helper functions

```
In [3]: def add_bias(X, b=1):
            n = X.shape[0]
            bias = b*np.ones((n,1))
            return np.hstack((bias, X))
```

```
In [4]: def load_data(train_size=0.8):
            df = pd.read_csv('data.csv').values
            train, test = train_test_split(df, train_size=train_size)
            train, test = feature_normalization(train, test)
            X = train[:, :-1]
            y = train[:,-1]
            X_test = test[:,:-1]
            y_test = test[:,-1]
            return X, y, X_test, y_test
```

```
In [5]: X, y, _, __ = load_data(train_size=0.99)
```

```
/home/cfizette/anaconda3/envs/ml/lib/python3.6/site-packages/sklearn/model_selection/_split.py:2
  FutureWarning)
```

```
In [6]: X = add_bias(X)
```

3.4 3.4 Batch Gradient Descent

3.4.1 3.4.1

“‘python ## ### Batch gradient descent def batch_grad_descent(X, y, alpha=0.1, num_step=1000, grad_check=False): """ In this question you will implement batch gradient descent to minimize the average square loss objective.

```
Args:
    X - the feature vector, 2D numpy array of size (num_instances, num_features)
    y - the label vector, 1D numpy array of size (num_instances)
    alpha - step size in gradient descent
    num_step - number of steps to run
    grad_check - a boolean value indicating whether checking the gradient when updating

Returns:
    theta_hist - the history of parameter vector, 2D numpy array of size (num_step+1, num_featur
                 for instance, theta in step 0 should be theta_hist[0], theta in step (num_step)
    loss_hist - the history of average square loss on the data, 1D numpy array, (num_step+1)
"""
num_instances, num_features = X.shape[0], X.shape[1]
theta_hist = np.zeros((num_step+1, num_features)) #Initialize theta_hist
loss_hist = np.zeros(num_step+1) #Initialize loss_hist
theta = np.zeros(num_features) #Initialize theta

for i in range(num_step+1):
    loss_hist[i] = compute_square_loss(X, y, theta)
    theta_hist[i] = theta

    grad = compute_square_loss_gradient(X, y, theta)

    if grad_check:
        if not grad_checker(X, y, theta):
            warnings.warn('Error computing gradient on iteration {}'.format(i))
            return theta_hist, loss_hist

    theta -= grad*alpha

return theta_hist, loss_hist
```

```
In [7]: batch_alphas = [0.5, 0.1, 0.05, 0.01, 0.00005]
        theta_hists = []
        loss_hists = []
        for alpha in batch_alphas:
            theta_hist, loss_hist = batch_grad_descent(X, y, alpha=alpha, num_step=1000, grad_ch
            theta_hists.append((theta_hist, alpha))
            loss_hists.append((loss_hist, alpha))
```