1.

1 Honor Code

Declare and sign the following statement (Mac Preview, PDF Expert, and FoxIt PDF Reader, among others, have tools to let you sign a PDF file):

"I certify that all solutions are entirely my own and that I have not looked at anyone else's solution. I have given credit to all external sources I consulted."

Signature: Vbaga

No collaborators

2.1.

$$\nabla_w J(w) = (\nabla_w S)(\nabla_S J(w))$$

Let $\sigma(z) = \frac{1}{1+e^{-z}}$ $\quad \sigma'(z) = \sigma(z)(1-\sigma(z))$

$$\nabla_w \sigma(x_i \cdot w) = \nabla_w (x_i \cdot w) \cdot \sigma'(x_i \cdot w)$$

$$= x_i \cdot \sigma(x_i^T w)(1 - \sigma(x_i^T w))$$

$$\nabla_w S = X^T (S \cdot (1-S))$$

(elementwise multiplication)

$$\nabla_S J(w) = \nabla_S (-y \cdot \ln S - (1-y) \cdot \ln(1-S))$$

$$= \nabla_S (-y \cdot \ln S) - \nabla_S (1-y) \cdot \ln(1-S)$$

$$= \frac{-y}{S} + \frac{1-y}{1-S}$$

(elementwise division)

$$\nabla_w J(w) = X^T \left(S \cdot (1-S)\left(\frac{-y}{S} + \frac{1-y}{1-S}\right)\right)$$

$$= X^T(-(1-S)y + S \cdot (1-y))$$

$$= X^T(-y + \cancel{Sy} + S - \cancel{Sy}) = X^T(S-y)$$

2.2. $\nabla_w^2 J(w) = \nabla_w (X^T(s-y))$

$$= \nabla_w (X^T s)$$

$$= (\nabla_w s) X$$

$$= (\nabla_w \sigma(Xw)) X$$

$$= X^T S X$$

where $S = \text{diag}(s_i(1-s_i))$

2.3.

$$w^{(x)} = w^{(x-1)} - \left(\nabla_w^2 J(w_{(x-1)})\right)^{-1} \nabla_w J(w_{(x-1)})$$

$$= w - \left(X^T S X\right)^{-1} \left(X^T (S^{(x-1)} - y)\right)$$

2.4.

$$X_1 = \begin{bmatrix} 0.2 \\ 3.1 \\ 1 \end{bmatrix} \quad X_2 = \begin{bmatrix} 1 \\ 3 \\ 1 \end{bmatrix} \quad X_3 = \begin{bmatrix} -0.2 \\ 1.2 \\ 1 \end{bmatrix}$$

$$X_4 = \begin{bmatrix} 1 \\ 1.1 \\ 1 \end{bmatrix}$$

$$y_{1,2,3,4} = 1, 1, 0, 0$$

a) $$S^{(0)} = \begin{bmatrix} \sigma(X_1 \cdot w^0) \\ \sigma(X_2 \cdot w^0) \\ \sigma(X_3 \cdot w^0) \\ \sigma(X_4 \cdot w^0) \end{bmatrix} = \begin{bmatrix} \sigma(2.9) \\ \sigma(2) \\ \sigma(1.4) \\ \sigma(0.1) \end{bmatrix} = \begin{bmatrix} 0.95 \\ 0.88 \\ 0.80 \\ 0.53 \end{bmatrix}$$

2.4.b

$$W^{(1)} = \begin{bmatrix} 1.32 \\ 3.05 \\ -6.83 \end{bmatrix}$$

2.4.c

$$S^{(1)} = \begin{bmatrix} 0.95 \\ 0.97 \\ 0.31 \\ 0.10 \end{bmatrix}$$

2.4.d

$$W^{(2)} = \begin{bmatrix} 1.37 \\ 4.16 \\ -9.2 \end{bmatrix}$$

3.1

$$J(w) = -\sum_{i=1}^{n} \left(y_i \ln s(X_i \cdot w) + (1 - y_i) \ln (1 - s(X_i \cdot w)) \right) + \lambda \|w\|^2$$

$$\nabla_w J(w) = X^T (s - y) + \lambda 2w$$

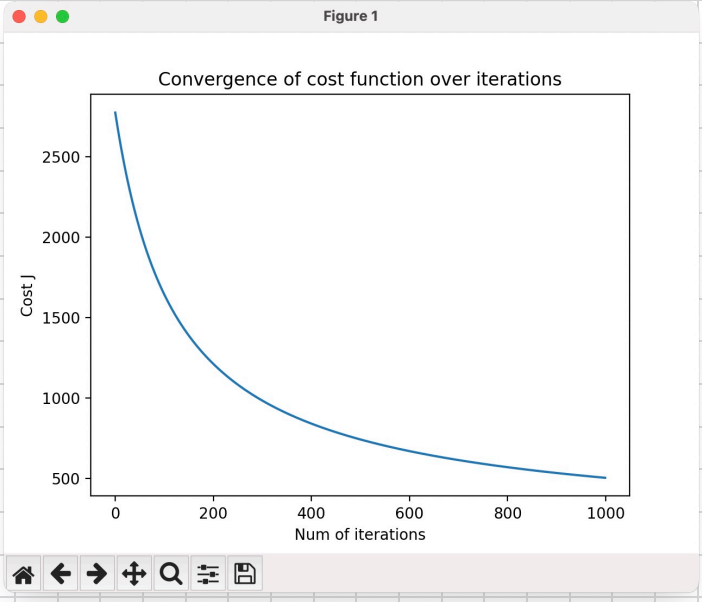
Update rule

$$w^{(i)} = w^{(i-1)} - \alpha \left(X^T (s - y) + \lambda 2 w^{(i-1)} \right)$$

$$= w^{(i-1)} - \alpha \left(X^T (s(X w^{(i-1)}) - y) + \lambda 2 w^{(i-1)} \right)$$

3.2.

```python
import numpy as np
from scipy.io import loadmat
from scipy.special import expit
import matplotlib.pyplot as plt

np.random.seed(44)

data = loadmat('data.mat')
X = data['X']
y = data['y'].flatten()
mean = np.mean(X, axis=0)
std = np.std(X, axis=0)
X_norm = (X - mean) / std

def split_data(X, y, test_size=0.2, random_state=42):
    m = X.shape[0]
    shuffled_indices = np.random.permutation(m)
    test_set_size = int(m * test_size)
    test_indices = shuffled_indices[:test_set_size]
    train_indices = shuffled_indices[test_set_size:]
    return X[train_indices], X[test_indices], y[train_indices], y[test_indices]


alpha = 0.01 / 5000
num_iters = 1000
X_with_intercept = np.concatenate([np.ones((X_norm.shape[0], 1)), X_norm], axis=1)
X_train, X_val, y_train, y_val = split_data(X_with_intercept, y, test_size=0.2)
lambda_ = 1 / 5000

def logistic_regression(X, y, w, alpha, lambda_, num_iters):
    J_history = []
    for _ in range(num_iters):
        z = X.dot(w)
        s = expit(z)
        error = s - y
        gradient = np.dot(X.T, error)
        gradient[1:] += 2 * (lambda_) * w[1:]
        w -= alpha * gradient
        cost = -np.sum(y * np.log(s) + (1 - y) * np.log(1 - s))
        cost += (lambda_) * np.sum(np.square(w[1:]))
        J_history.append(cost)
    return J_history

initial_w = np.zeros(X_with_intercept.shape[1])
J_history = logistic_regression(X_train, y_train, initial_w, alpha, lambda_, num_iters)
plt.plot(J_history)
plt.xlabel('Num of iterations')
plt.ylabel('Cost J')
plt.title('Convergence of cost function over iterations')
plt.show()
```



$\alpha = 0.01 / 5000$

$\lambda = 1 / 5000$

random seed = 44

ex3_2.py

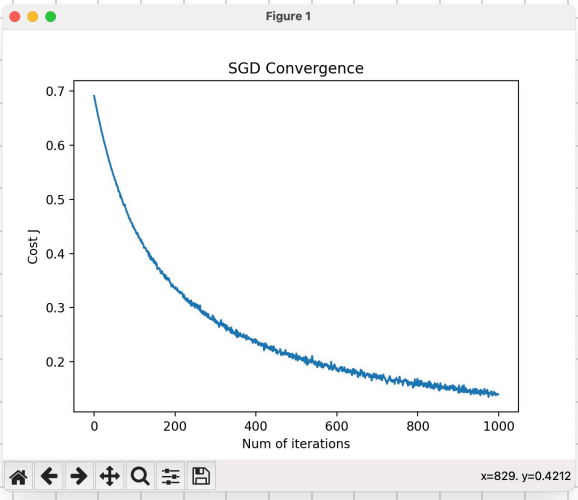
3.3.

$$w^{(t)} = w^{(t-1)} - \alpha \left(X_j^T (s - y)_j + \lambda 2w \right)$$

where j is random in $[0, \ldots, \dim(w)]$

3.4.

```
s > bayelasyibekov > Downloads > hw4 2 > ex3_4.py > stochastic_gradient_descent
import numpy as np
from scipy.io import loadmat
from scipy.special import expit
import matplotlib.pyplot as plt

np.random.seed(42)
data = loadmat('data.mat')
X = data['X']
y = data['y'].flatten()
mean = np.mean(X, axis=0)
std = np.std(X, axis=0)
X_norm = (X - mean) / std
X_with_intercept = np.concatenate([np.ones((X_norm.shape[0], 1)), X_norm], axis=1)
shuffle_indices = np.random.permutation(np.arange(X_with_intercept.shape[0]))
train_indices = shuffle_indices[:int(0.8 * len(shuffle_indices))]
val_indices = shuffle_indices[int(0.8 * len(shuffle_indices)):]
X_train, X_val = X_with_intercept[train_indices], X_with_intercept[val_indices]
y_train, y_val = y[train_indices], y[val_indices]
n = X_train.shape[1]
initial_w = np.zeros(n)
def stochastic_gradient_descent(X, y, initial_w, alpha, lambda_, num_iters):
    m, n = X.shape
    w = initial_w.copy()
    J_history = []
    for iter in range(num_iters):
        cost = 0
        for i in range(m):
            random_index = np.random.randint(m)
            xi = X[random_index, :].reshape(1, -1)
            yi = y[random_index]
            prediction = expit(xi @ w)
            error = prediction - yi
            gradient = xi.T @ error + (lambda_/m) * np.r_[[0], w[1:]]
            w -= alpha * gradient.flatten()
            cost += - yi * np.log(prediction) - (1 - yi) * np.log(1 - prediction)
        cost /= m
        reg_term = (lambda_ / (2 * m)) * np.sum(w[1:]**2)
        J_history.append(cost + reg_term)
    return J_history
alpha = 0.01 / 5000
lambda_ = 1 / 5000
num_iters = 1000
J_history_sgd = stochastic_gradient_descent(X_train, y_train, initial_w, alpha, lambda_, num_iters)
plt.plot(J_history_sgd)
plt.xlabel('Num of iterations')
plt.ylabel('Cost J')
plt.title('SGD Convergence')
plt.show()
```



ex3_4.py

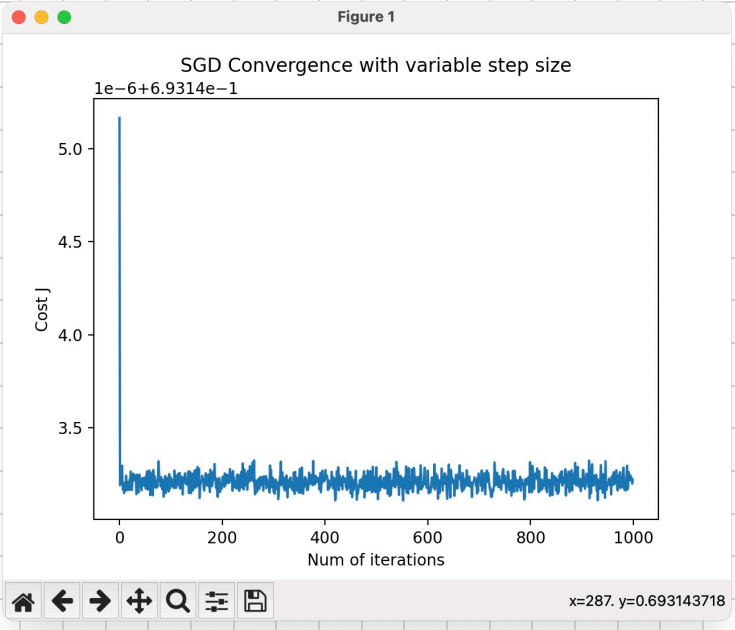
$\alpha = 0.01 / 5000$

$\lambda = 1 / 5000$

random seed = 42

stockastic method converges more quickly because the cost is smaller.

3.5.



$\delta = \alpha = 1/5000$

$\lambda = 1/5000$

rand seed = 42

It converges faster but the cost seems to settle at a higher value.

```python
import numpy as np
from scipy.io import loadmat
from scipy.special import expit
import matplotlib.pyplot as plt

np.random.seed(42)
data = loadmat('data.mat')
X = data['X']
y = data['y'].flatten()
mean = np.mean(X, axis=0)
std = np.std(X, axis=0)
X_norm = (X - mean) / std
X_with_intercept = np.concatenate([np.ones((X_norm.shape[0], 1)), X_norm], axis=1)
shuffle_indices = np.random.permutation(np.arange(X_with_intercept.shape[0]))
train_indices = shuffle_indices[:int(0.8 * len(shuffle_indices))]
val_indices = shuffle_indices[int(0.8 * len(shuffle_indices)):]
X_train, X_val = X_with_intercept[train_indices], X_with_intercept[val_indices]
y_train, y_val = y[train_indices], y[val_indices]
n = X_train.shape[1]
initial_w = np.zeros(n)
def stochastic_gradient_descent(X, y, initial_w, alpha, lambda_, num_iters):
    m, n = X.shape
    w = initial_w.copy()
    J_history = []
    for iter in range(num_iters):
        cost = 0
        alpha /= num_iters
        for i in range(m):
            random_index = np.random.randint(m)
            xi = X[random_index, :].reshape(1, -1)
            yi = y[random_index]
            prediction = expit(xi @ w)
            error = prediction - yi
            gradient = xi.T @ error + (lambda_/m) * np.r_[[0], w[1:]]
            w -= alpha * gradient.flatten()
            cost += - yi * np.log(prediction) - (1 - yi) * np.log(1 - prediction)
        cost /= m
        reg_term = (lambda_ / (2 * m)) * np.sum(w[1:]**2)
        J_history.append(cost + reg_term)
    return J_history
alpha = 1 / 5000
lambda_ = 1 / 5000
num_iters = 1000
J_history_sgd = stochastic_gradient_descent(X_train, y_train, initial_w, alpha, lambda_, num_iters)
plt.plot(J_history_sgd)
plt.xlabel('Num of iterations')
plt.ylabel('Cost J')
plt.title('SGD Convergence with variable step size')
plt.show()
```

ex3_5.py

3.6.

ex3_6.py

```python
import numpy as np
from scipy.io import loadmat
from scipy.special import expit
from sklearn.svm import SVC
import csv
np.random.seed(42)
data = loadmat('data.mat')
X = data['X']
y = data['y'].flatten()
X_test = data['X_test']
mean = np.mean(X, axis=0)
std = np.std(X, axis=0)
X_norm = (X - mean) / std
X_test_norm = (X_test - mean) / std
X_with_intercept = np.concatenate([np.ones((X_norm.shape[0], 1)), X_norm], axis=1)
X_test_with_intercept = np.concatenate([np.ones((X_test_norm.shape[0], 1)), X_test_norm], axis=1)
shuffle_indices = np.random.permutation(np.arange(X_with_intercept.shape[0]))
train_indices = shuffle_indices[:int(0.8 * len(shuffle_indices))]
val_indices = shuffle_indices[int(0.8 * len(shuffle_indices)):]
X_train, X_val = X_with_intercept[train_indices], X_with_intercept[val_indices]
y_train, y_val = y[train_indices], y[val_indices]
n = X_train.shape[1]
initial_theta = np.zeros(n)
def stochastic_gradient_descent(X, y, initial_w, alpha, lambda_, num_iters):
    m, n = X.shape
    w = initial_w.copy()
    J_history = []
    iter_number = 1
    for iter in range(num_iters):
        cost = 0
        for i in range(m):
            alpha = alpha/(iter + 1)
            random_index = np.random.randint(m)
            xi = X[random_index, :].reshape(1, -1)
            yi = y[random_index]
            prediction = expit(xi @ w)
            error = prediction - yi
            gradient = xi.T @ error + (lambda_/m) * np.r_[[0], w[1:]]
            w -= alpha * gradient.flatten()
            cost += - yi * np.log(prediction) - (1 - yi) * np.log(1 - prediction)
            iter_number += 1
        cost /= m
        reg_term = (lambda_ / (2 * m)) * np.sum(w[1:]**2)
        J_history.append(cost + reg_term)
    return w, J_history
alpha = 0.1
lambda_ = 1
num_iters = 100
theta_sgd, J_history_sgd = stochastic_gradient_descent(X_train, y_train, initial_theta, alpha, lambda_, num_iters)
def predict(theta, X):
    probabilities = expit(X.dot(theta))
    return [1 if x >= 0.5 else 0 for x in probabilities]
predictions = predict(theta_sgd, X_test_with_intercept)
with open('example.csv', mode='w', newline='\n') as file:
    writer = csv.writer(file)
    writer.writerow(["Id", "Category"])
    id = 1
    for categ in predictions:
        writer.writerow([id, categ])
        id += 1
```

alpha = 0.1/5000

lambda = 1/5000

my best classifier is a Stochastic gradient descent with decreasing stepsize.

Keggle : Bayelle

Score: 0.993

Private score:

4.1.

$$f(w \mid (x_i, y_i)) \propto f(y_i \mid x_i, w) \cdot f(w)$$

$$f(y_i \mid x_i, w) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(y_i - w \cdot x_i)^2 / 2\sigma^2}$$

$$f(w) = \prod_{j=1}^{d} f(w_j) = \prod_{j=1}^{d} \frac{1}{2b} e^{-|w_j|/b}$$

$$f(w \mid (x_i, y_i)_{i \in [1,\dots,n]}) \propto \prod_{i=1}^{n} \frac{1}{\sigma\sqrt{2\pi}} e^{-(y_i - w \cdot x_i)^2 / 2\sigma^2} \cdot \prod_{j=1}^{d} \frac{1}{2b} e^{-|w_j|/b}$$

$$\propto e^{\sum_{i=1}^{n} -(y_i - w \cdot x_i)^2 / 2\sigma^2 + \sum_{j=1}^{d} \frac{-|w_j|}{b}}$$

4.2.

$$f(w) = \ln f(w, x_{i\in[n]}, y_{i\in[n]})$$

$$\propto \ln e^{\sum_{i=1}^{n} -(y_i - w\cdot x_i)^2/2\sigma^2 + \sum_{j=1}^{d} \frac{-|w_j|}{b}}$$

$$= \sum_{i=1}^{n} -(y_i - w\cdot x_i)^2/2\sigma^2 + \sum_{j=1}^{d} \frac{-|w_j|}{b}$$

maximizing this is equal to minimizing this

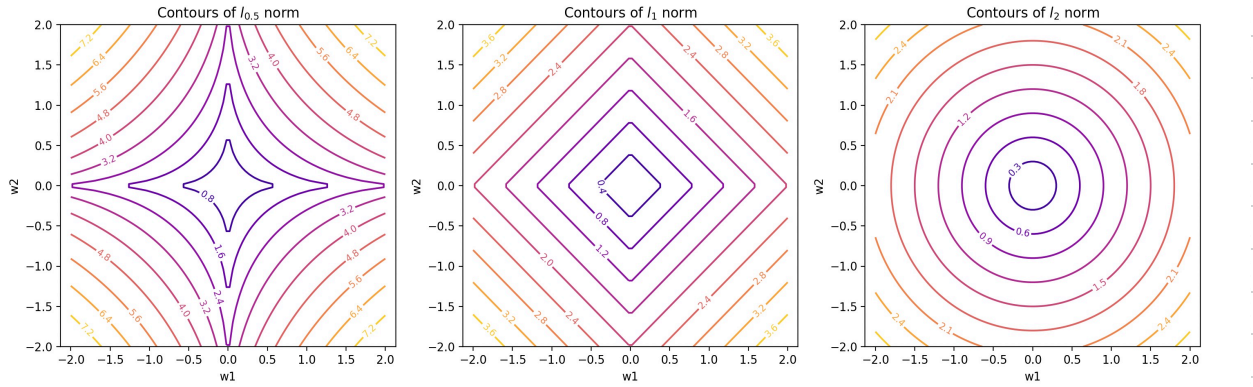
$$\sum_{i=1}^{n} (y_i - w\cdot x_i)^2/2\sigma^2 + \sum_{j=1}^{d} |w_j|/b$$

$$\propto \sum_{i=1}^{n} (y_i - w\cdot x_i)^2 + \frac{2\sigma^2}{b} \sum_{j=1}^{d} |w_j|$$

Set $\lambda = \frac{2\sigma^2}{b}$

$$= \sum_{i=1}^{n} (y_i - w\cdot x_i)^2 + \lambda \|w\|_1$$

5.1.




```python
import numpy as np
import matplotlib.pyplot as plt

def plot_lp_contours(p, ax, title, range_lim=2.0, num_points=100):
    x = np.linspace(-range_lim, range_lim, num_points)
    y = np.linspace(-range_lim, range_lim, num_points)
    X, Y = np.meshgrid(x, y)

    Z = (np.abs(X)**p + np.abs(Y)**p)**(1/p)

    contours = ax.contour(X, Y, Z, levels=10, cmap='plasma')
    ax.clabel(contours, inline=True, fontsize=8)
    ax.set_title(f'Contours of ${title}$ norm')
    ax.set_xlabel('w1')
    ax.set_ylabel('w2')
    ax.axis('equal')

fig, axs = plt.subplots(1, 3, figsize=(18, 5))
plot_lp_contours(0.5, axs[0], 'l_{0.5}')
plot_lp_contours(1, axs[1], 'l_1')
plot_lp_contours(2, axs[2], 'l_2')
plt.show()
```

ex5_1.pg

5.2.

$$J_1(w) = \|Xw - y\|^2 + \lambda \sum_{i=1}^{d} |w_i|$$

$$= (Xw-y)^T(Xw-y) + \lambda \sum_{i=1}^{d} |w_i|$$

$$= w^T X^T X w - y^T X w - w^T X^T y + y^T y + \lambda \sum_{i=1}^{d} |w_i|$$

$$= \|y\|^2 + n\, w^T w - 2 w X^T y + \sum_{i=1}^{d} \lambda |w_i|$$

$$= \|y\|^2 + \sum_{i=1}^{d} w_i^2 - \sum_{i=1}^{d} 2(w_i \cdot X_{*i}^T y) + \sum_{i=1}^{d} \lambda |w_i|$$

$$= \|y\|^2 + \sum_{i=1}^{d} \underbrace{w_i^2 - 2(w_i \cdot X_{*i}^T y) + \lambda |w_i|}_{f(x_{*i}, w_i)}$$

5.3.

$w_i^* > 0$

$$\frac{\partial}{\partial w_i} J_1(w) = 2w_i - 2X_{*i}^T y + \lambda = 0$$

$$\Rightarrow \lambda = 2X_{*i}^T y - 2w_i$$

$w_i^* < 0$

$$\frac{\partial}{\partial w_i} J_1(w) = 2w_i - 2X_{*i}^T y - \lambda = 0$$

$$\Rightarrow \lambda = 2w_i - 2X_{*i}^T y$$

$w_i^* = 0$

if $\quad -\lambda < 2(w_i - 2X_{*i}^T y) < \lambda$

5.4.

$$\nabla_w J_2(w) = 2(X^T X w - X^T y) + 2\lambda w$$

$$= 2(nw - X^T y) + 2\lambda w$$

for i^{th} component

$$2(nw_i - (X^T y)_i) + 2\lambda w_i = 0$$

if $w_i^* = 0$

then $(X^T y)_i = 0$

if $(X^T y)_i = 0$

then $2nw_i + 2\lambda w_i = 0 \Rightarrow w_i = 0$

since $n, \lambda > 0$

thus

nessasary and sufficient $(X^T y)_i = 0$

5.5 w^* is more likely to be sparse because for an ℓ_1 norm to get zero there is a whole range of values for λ, while for $w^\#$ we need $(X^T y)_j = 0$.

3.2

```python
import numpy as np
from scipy.io import loadmat
from scipy.special import expit
import matplotlib.pyplot as plt

np.random.seed(44)

data = loadmat('data.mat')
X = data['X']
y = data['y'].flatten()
mean = np.mean(X, axis=0)
std = np.std(X, axis=0)
X_norm = (X - mean) / std

def split_data(X, y, test_size=0.2, random_state=42):
    m = X.shape[0]
    shuffled_indices = np.random.permutation(m)
    test_set_size = int(m * test_size)
    test_indices = shuffled_indices[:test_set_size]
    train_indices = shuffled_indices[test_set_size:]
    return X[train_indices], X[test_indices], y[train_indices], y[test_indices]


alpha = 0.01 / 5000
num_iters = 1000
X_with_intercept = np.concatenate([np.ones((X_norm.shape[0], 1)), X_norm], axis=1)
X_train, X_val, y_train, y_val = split_data(X_with_intercept, y, test_size=0.2)
lambda_ = 1 / 5000

def logistic_regression(X, y, w, alpha, lambda_, num_iters):
    J_history = []
    for _ in range(num_iters):
        z = X.dot(w)
        s = expit(z)
        error = s - y
        gradient = np.dot(X.T, error)
        gradient[1:] += 2 * (lambda_) * w[1:]
        w -= alpha * gradient
        cost = -np.sum(y * np.log(s) + (1 - y) * np.log(1 - s))
        cost += (lambda_) * np.sum(np.square(w[1:]))
        J_history.append(cost)
    return J_history

initial_w = np.zeros(X_with_intercept.shape[1])
J_history = logistic_regression(X_train, y_train, initial_w, alpha, lambda_, num_iters)
plt.plot(J_history)
plt.xlabel('Num of iterations')
plt.ylabel('Cost J')
plt.title('Convergence of cost function over iterations')
plt.show()
```

3.4

s > bayelasylbekov > Downloads > hw4 2 > ex3_4.py > stochastic_gradient_descent

```python
import numpy as np
from scipy.io import loadmat
from scipy.special import expit
import matplotlib.pyplot as plt

np.random.seed(42)
data = loadmat('data.mat')
X = data['X']
y = data['y'].flatten()
mean = np.mean(X, axis=0)
std = np.std(X, axis=0)
X_norm = (X - mean) / std
X_with_intercept = np.concatenate([np.ones((X_norm.shape[0], 1)), X_norm], axis=1)
shuffle_indices = np.random.permutation(np.arange(X_with_intercept.shape[0]))
train_indices = shuffle_indices[:int(0.8 * len(shuffle_indices))]
val_indices = shuffle_indices[int(0.8 * len(shuffle_indices)):]
X_train, X_val = X_with_intercept[train_indices], X_with_intercept[val_indices]
y_train, y_val = y[train_indices], y[val_indices]
n = X_train.shape[1]
initial_w = np.zeros(n)
def stochastic_gradient_descent(X, y, initial_w, alpha, lambda_, num_iters):
    m, n = X.shape
    w = initial_w.copy()
    J_history = []
    for iter in range(num_iters):
        cost = 0
        for i in range(m):
            random_index = np.random.randint(m)
            xi = X[random_index, :].reshape(1, -1)
            yi = y[random_index]
            prediction = expit(xi @ w)
            error = prediction - yi
            gradient = xi.T @ error + (lambda_/m) * np.r_[[0], w[1:]]
            w -= alpha * gradient.flatten()
            cost += - yi * np.log(prediction) - (1 - yi) * np.log(1 - prediction)
        cost /= m
        reg_term = (lambda_ / (2 * m)) * np.sum(w[1:]**2)
        J_history.append(cost + reg_term)
    return J_history
alpha = 0.01 / 5000
lambda_ = 1 / 5000
num_iters = 1000
J_history_sgd = stochastic_gradient_descent(X_train, y_train, initial_w, alpha, lambda_, num_iters)
plt.plot(J_history_sgd)
plt.xlabel('Num of iterations')
plt.ylabel('Cost J')
plt.title('SGD Convergence')
plt.show()
```

3.5

```python
import numpy as np
from scipy.io import loadmat
from scipy.special import expit
import matplotlib.pyplot as plt

np.random.seed(42)
data = loadmat('data.mat')
X = data['X']
y = data['y'].flatten()
mean = np.mean(X, axis=0)
std = np.std(X, axis=0)
X_norm = (X - mean) / std
X_with_intercept = np.concatenate([np.ones((X_norm.shape[0], 1)), X_norm], axis=1)
shuffle_indices = np.random.permutation(np.arange(X_with_intercept.shape[0]))
train_indices = shuffle_indices[:int(0.8 * len(shuffle_indices))]
val_indices = shuffle_indices[int(0.8 * len(shuffle_indices)):]
X_train, X_val = X_with_intercept[train_indices], X_with_intercept[val_indices]
y_train, y_val = y[train_indices], y[val_indices]
n = X_train.shape[1]
initial_w = np.zeros(n)
def stochastic_gradient_descent(X, y, initial_w, alpha, lambda_, num_iters):
    m, n = X.shape
    w = initial_w.copy()
    J_history = []
    for iter in range(num_iters):
        cost = 0
        alpha /= num_iters
        for i in range(m):
            random_index = np.random.randint(m)
            xi = X[random_index, :].reshape(1, -1)
            yi = y[random_index]
            prediction = expit(xi @ w)
            error = prediction - yi
            gradient = xi.T @ error + (lambda_/m) * np.r_[[0], w[1:]]
            w -= alpha * gradient.flatten()
            cost += - yi * np.log(prediction) - (1 - yi) * np.log(1 - prediction)
        cost /= m
        reg_term = (lambda_ / (2 * m)) * np.sum(w[1:]**2)
        J_history.append(cost + reg_term)
    return J_history
alpha = 1 / 5000
lambda_ = 1 / 5000
num_iters = 1000
J_history_sgd = stochastic_gradient_descent(X_train, y_train, initial_w, alpha, lambda_, num_iters)
plt.plot(J_history_sgd)
plt.xlabel('Num of iterations')
plt.ylabel('Cost J')
plt.title('SGD Convergence with variable step size')
plt.show()
```

3.6

```python
import numpy as np
from scipy.io import loadmat
from scipy.special import expit
from sklearn.svm import SVC
import csv
np.random.seed(42)
data = loadmat('data.mat')
X = data['X']
y = data['y'].flatten()
X_test = data['X_test']
mean = np.mean(X, axis=0)
std = np.std(X, axis=0)
X_norm = (X - mean) / std
X_test_norm = (X_test - mean) / std
X_with_intercept = np.concatenate([np.ones((X_norm.shape[0], 1)), X_norm], axis=1)
X_test_with_intercept = np.concatenate([np.ones((X_test_norm.shape[0], 1)), X_test_norm], axis=1)
shuffle_indices = np.random.permutation(np.arange(X_with_intercept.shape[0]))
train_indices = shuffle_indices[:int(0.8 * len(shuffle_indices))]
val_indices = shuffle_indices[int(0.8 * len(shuffle_indices)):]
X_train, X_val = X_with_intercept[train_indices], X_with_intercept[val_indices]
y_train, y_val = y[train_indices], y[val_indices]
n = X_train.shape[1]
initial_theta = np.zeros(n)
def stochastic_gradient_descent(X, y, initial_w, alpha, lambda_, num_iters):
    m, n = X.shape
    w = initial_w.copy()
    J_history = []
    iter_number = 1
    for iter in range(num_iters):
        cost = 0
        for i in range(m):
            alpha = alpha/(iter + 1)
            random_index = np.random.randint(m)
            xi = X[random_index, :].reshape(1, -1)
            yi = y[random_index]
            prediction = expit(xi @ w)
            error = prediction - yi
            gradient = xi.T @ error + (lambda_/m) * np.r_[[0], w[1:]]
            w -= alpha * gradient.flatten()
            cost += - yi * np.log(prediction) - (1 - yi) * np.log(1 - prediction)
            iter_number += 1
        cost /= m
        reg_term = (lambda_ / (2 * m)) * np.sum(w[1:]**2)
        J_history.append(cost + reg_term)
    return w, J_history
alpha = 0.1
lambda_ = 1
num_iters = 100
theta_sgd, J_history_sgd = stochastic_gradient_descent(X_train, y_train, initial_theta, alpha, lambda_, num_iters)
def predict(theta, X):
    probabilities = expit(X.dot(theta))
    return [1 if x >= 0.5 else 0 for x in probabilities]
predictions = predict(theta_sgd, X_test_with_intercept)
with open('example.csv', mode='w', newline='\n') as file:
    writer = csv.writer(file)
    writer.writerow(["Id", "Category"])
    id = 1
    for categ in predictions:
        writer.writerow([id, categ])
        id += 1
```

5.1.

```python
import numpy as np
import matplotlib.pyplot as plt

def plot_lp_contours(p, ax, title, range_lim=2.0, num_points=100):
    x = np.linspace(-range_lim, range_lim, num_points)
    y = np.linspace(-range_lim, range_lim, num_points)
    X, Y = np.meshgrid(x, y)

    Z = (np.abs(X)**p + np.abs(Y)**p)**(1/p)

    contours = ax.contour(X, Y, Z, levels=10, cmap='plasma')
    ax.clabel(contours, inline=True, fontsize=8)
    ax.set_title(f'Contours of ${title}$ norm')
    ax.set_xlabel('w1')
    ax.set_ylabel('w2')
    ax.axis('equal')

fig, axs = plt.subplots(1, 3, figsize=(18, 5))
plot_lp_contours(0.5, axs[0], 'l_{0.5}')
plot_lp_contours(1, axs[1], 'l_1')
plot_lp_contours(2, axs[2], 'l_2')
plt.show()
```