MITx_6.86x_Project_2_DigitRecognition_P1

July 7, 2019

1 Project 2: Digit recognition (Part 1)

1.1 1. Introduction

Alice, Bob, and Daniel are friends learning machine learning together. After watching a few lectures, they are very proud of having learned many useful tools, including linear and logistic regression, non-linear features, regularization, and kernel tricks. To see how these methods can be used to solve a real life problem, they decide to get their hands dirty with the famous digit recognition problem using the MNIST (Mixed National Institute of Standards and Technology) database.

Hearing that you are an excellent student in the MITx machine learning class with solid understanding of the material and great coding ability in Python, they decide to invite you to their team and help them with implementing these different algorithms.

The MNIST database contains binary images of handwritten digits commonly used to train image processing systems. The digits were collected from among Census Bureau employees and high school students. The database contains 60,000 training digits and 10,000 testing digits, all of which have been size-normalized and centered in a fixed-size image of 28 Œ 28 pixels. Many methods have been tested with this dataset and in this project, you will get a chance to experiment with the task of classifying these images into the correct digit using some of the methods you have learned so far.

1.1.1 Setup:

As with the last project, please use Python's NumPy numerical library for handling arrays and array operations; use matplotlib for producing figures and plots.

This project will be split in two parts. Project 2 (this project) consists in the first part and project 3 will cover the second part.

Note on software: For all the projects, we will use python 3.6 augmented with the NumPy numerical toolbox, the matplotlib plotting toolbox. In this project, we will also use the scikit-learn package, which you could install in the same way you installed other packages, as described in project 0, e.g. by conda install scikit-learn or pip install sklearn

Download mnist.tar.gz and untar it in to a working directory. The archive contains the various data files in the Dataset directory, along with the following python files:

part1/linear_regression.py where you will implement linear regression
part1/svm.py where you will implement support vector machine
part1/softmax.py where you will implement multinomial regression

part1/features.py where you will implement principal component analysis (PCA) CA dimensionality reduction

part1/kernel.py where you will implement polynomial and Gaussian RBF kernels

part1/main.py where you will use the code you write for this part of the project

Important: The archive also contains files for the second part of the MNIST project. For this project, you will only work with the part1 folder.

To get warmed up to the MNIST data set run python main.py. This file provides code that reads the data from mnist.pkl.gz by calling the function get_MNIST_data that is provided for you in utils.py. The call to get_MNIST_data returns Numpy arrays:

train_x : A matrix of the training data. Each row of train_x contains the features of one image, which are simply the raw pixel values flattened out into a vector of length $784 = 28^2$. The pixel values are float values between 0 and 1 (0 stands for black, 1 for white, and various shades of gray in-between).

train_y : The labels for each training datapoint, aka the digit shown in the corresponding image (a number between 0-9).

test_x : A matrix of the test data, formatted like train_x.

test_y : The labels for the test data, which should only be used to evaluate the accuracy of different classifiers in your report.

Next, we call the function plot_images to display the first 20 images of the training set. Look at these images and get a feel for the data (don't include these in your write-up).

Tip: Throughout the whole online grading system, you can assume the NumPy python library is already imported as np. In some problems you will also have access to python's random library, and other functions you've already implemented. Look out for the "Available Functions" Tip before the codebox, as you did in the last project.

This project will unfold both on MITx and on your local machine. However, we encourage you to first implement the functions locally and run the test scripts to validate basic funcitonality. Think of the online graders as a submission box to submit your code when it is ready. You should not have to use the online graders to debug your code.

1.1.2 utils.py

[ ]:
```
import pickle, gzip, numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import math


def plot_images(X):
    if X.ndim == 1:
        X = np.array([X])
    num_images = X.shape[0]
    num_rows = math.floor(math.sqrt(num_images))
    num_cols = math.ceil(num_images/num_rows)
    for i in range(num_images):
        reshaped_image = X[i,:].reshape(28,28)
        plt.subplot(num_rows, num_cols, i+1)
        plt.imshow(reshaped_image, cmap = cm.Greys_r)
```

```
        plt.axis('off')
    plt.show()


def pick_examples_of(X, Y, labels, total_count):
    bool_arr = None
    for label in labels:
        bool_arr_for_label = (Y == label)
        if bool_arr is None:
            bool_arr = bool_arr_for_label
        else:
            bool_arr |= bool_arr_for_label
    filtered_x = X[bool_arr]
    filtered_y = Y[bool_arr]
    return (filtered_x[:total_count], filtered_y[:total_count])


def extract_training_and_test_examples_with_labels(train_x, train_y, test_x,␣
 ↪test_y, labels, training_count, test_count):
    filtered_train_x, filtered_train_y = pick_examples_of(train_x, train_y,␣
 ↪labels, training_count)
    filtered_test_x, filtered_test_y = pick_examples_of(test_x, test_y, labels,␣
 ↪test_count)
    return (filtered_train_x, filtered_train_y, filtered_test_x,␣
 ↪filtered_test_y)

def write_pickle_data(data, file_name):
    f = gzip.open(file_name, 'wb')
    pickle.dump(data, f)
    f.close()

def read_pickle_data(file_name):
    f = gzip.open(file_name, 'rb')
    data = pickle.load(f, encoding='latin1')
    f.close()
    return data

def get_MNIST_data():
    """
    Reads mnist dataset from file

    Returns:
        train_x - 2D Numpy array (n, d) where each row is an image
        train_y - 1D Numpy array (n, ) where each row is a label
        test_x  - 2D Numpy array (n, d) where each row is an image
        test_y  - 1D Numpy array (n, ) where each row is a label
```

```
    """
    train_set, valid_set, test_set = read_pickle_data('../Datasets/mnist.pkl.
↪gz')
    train_x, train_y = train_set
    valid_x, valid_y = valid_set
    train_x = np.vstack((train_x, valid_x))
    train_y = np.append(train_y, valid_y)
    test_x, test_y = test_set
    return (train_x, train_y, test_x, test_y)

def load_train_and_test_pickle(file_name):
    train_x, train_y, test_x, test_y = read_pickle_data(file_name)
    return train_x, train_y, test_x, test_y

# returns the feature set in a numpy ndarray
def load_CSV(filename):
    stuff = np.asarray(np.loadtxt(open(filename, 'rb'), delimiter=','))
    return stuff
```

1.1.3 cubic_features_checker.py

[ ]:
```
import sys
sys.path.append("..")
import utils
from utils import *
import numpy as np
from features import cubic_features

def verify_cubic_features1D():
    X=np.array([[np.sqrt(3)],[0]])
    X_cube=np.sort(cubic_features(X))
    X_correct = np.array([[ 1., np.sqrt(9), np.sqrt(27), np.sqrt(27)],[0., 0.,␣
↪0., 1.]]);

    if np.all(np.absolute(X_cube-X_correct) < 1.0e-6):
        print ("Verifying cubic features of 1 dimension: Passed")
    else:
        print ("Verifying cubic features of 1 dimension: Failed")


def verify_cubic_features2D():
    X=np.array([[np.sqrt(3),np.sqrt(3)],[0,0]])
    X_cube=np.sort(cubic_features(X))
    X_correct = np.array([[1., 3., 3., 5.19615242, 5.19615242, 5.19615242, 5.
↪19615242, 7.34846923, 9., 9.],
                          [0., 0., 0., 0., 0., 0., 0., 0., 0., 1.]])
```

```
    if np.all(np.absolute(X_cube-X_correct) < 1.0e-6):
        print ("Verifying cubic features of 2 dimensions: Passed")
    else:
        print ("Verifying cubic features of 2 dimensions: Failed")


def verify_cubic_features2D2():
    X=np.array([[np.sqrt(3),0],[0,np.sqrt(3)]])
    X_cube=np.sort(cubic_features(X))
    X_correct = np.array([[0., 0., 0., 0., 0., 0., 1., 3., 5.19615242, 5.
 ↪19615242],
                          [0., 0., 0., 0., 0., 0., 1., 3., 5.19615242, 5.
 ↪19615242]])

    if np.all(np.absolute(X_cube-X_correct) < 1.0e-6):
        print ("Verifying cubic features of 2 dimensions asymmetric vectors:␣
 ↪Passed")
    else:
        print ("Verifying cubic features of 2 dimensions asymmetric vectors:␣
 ↪Failed")

verify_cubic_features1D()
verify_cubic_features2D()
verify_cubic_features2D2()
```

1.2 2. Linear Regression with Closed Form Solution

After seeing the problem, your classmate Alice immediately argues that we can apply a linear regression model, as the labels are numbers from 0-9, very similar to the example we learned from the lecture. Though being a little doubtful, you decide to have a try and start simple by using the raw pixel values of each image as features.

Alice wrote a skeleton code run_linear_regression_on_MNIST in main.py, but she needs your help to complete the code and make the model work.

[ ]:
```
import numpy as np
```

[ ]:
```
def compute_test_error_linear(test_x, Y, theta):
    test_y_predict = np.round(np.dot(test_x, theta))
    test_y_predict[test_y_predict < 0] = 0
    test_y_predict[test_y_predict > 9] = 9
    return 1 - np.mean(test_y_predict == Y)
```

1.2.1 Closed Form Solution of Linear Regression

To solve the linear regression problem, you recall the linear regression has a closed form solution:

$$\theta = (X^T X + \lambda I)^{-1} X^T Y$$

where I is the identity matrix.

Write a function closed_form that computes this closed form solution given the features X, labels Y and the regularization parameter .

Available Functions: You have access to the NumPy python library as np; No need to import anything.

[ ]:
```
def closed_form(X, Y, lambda_factor):
    """
    Computes the closed form solution of linear regression with L2 regularization

    Args:
        X - (n, d + 1) NumPy array (n datapoints each with d features plus the bias feature in the first dimension)
        Y - (n, ) NumPy array containing the labels (a number from 0-9) for each
            data point
        lambda_factor - the regularization constant (scalar)
    Returns:
        theta - (d + 1, ) NumPy array containing the weights of linear regression. Note that theta[0]
        represents the y-axis intercept of the model and therefore X[0] = 1
    """
    # YOUR CODE HERE
    n, d = X.shape
    return np.dot(np.linalg.inv(np.dot(np.transpose(X), X) + lambda_factor*np.eye(d,d)), np.dot(np.transpose(X), Y))
```

[ ]:
```
X = np.arange(1, 16).reshape(3, 5)
Y = np.arange(1, 4)
lambda_factor = 0.5
print(X, Y)
closed_form(X, Y, lambda_factor)
```

1.2.2 Test Error on Linear Regression

Apply the linear regression model on the test set. For classification purpose, you decide to round the predicted label into numbers 0-9.

Note: For this project we will be looking at the error rate defined as the fraction of labels that don't match the target labels, also known as the "gold labels" or ground truth. In other contexts, you might want to consider other performance measures we've learned about in this class, including precision and recall.

Please enter the test error of your linear regression algorithm for different (copy the output from the main.py run).

$Error|_{\lambda=1} = 0.744$

$Error|_{\lambda=0.1} = 0.7442$

$Error|_{\lambda=0.01} = 0.744$

1.2.3 What went Wrong?

Alice and you find that no matter what lambda factor you try, the test error is large. With some thinking, you realize that something is wrong with this approach.

× Gradient descent should be used instead of the closed form solution.

∨ The loss function related to the closed-form solution is inadequate for this problem.

× Regularization should not be used here.

1.3 3. Support Vector Machine

Bob thinks it is clearly not a regression problem, but a classification problem. He thinks that we can change it into a binary classification and use the support vector machine we learned in Lecture 4 to solve the problem. In order to do so, he suggests that we can build an one vs. rest model for every digit. For example, classifying the digits into two classes: 0 and not 0.

Bob wrote a function run_svm_one_vs_rest_on_MNIST where he changed the labels of digits 1-9 to 1 and keeps the label 0 for digit 0. He also found that sklearn package contains an SVM model that you can use directly. He gave you the link to this model and hopes you can tell him how to use that.

You will be working in the file part1/svm.py in this problem

Important: For this problem, you will need to use the scikit-learn library. If you don't have it, install it using pip install sklearn

1.3.1 One vs. Rest SVM

Use the sklearn package and build the SVM model on your local machine. Use random_state = 0, C=0.1 and default values for other parameters.

Available Functions: You have access to the sklearn's implementation of the linear SVM as LinearSVC; No need to import anything.

[ ]:
```
import numpy as np
from sklearn.svm import LinearSVC
```

[ ]:
```
# LinearSVC(C=0.1, class_weight=None, dual=True, fit_intercept=True,␣
↪intercept_scaling=1, loss='squared_hinge', max_iter=1000, multi_class='ovr',␣
↪penalty='l2', random_state=0, tol=1e-05, verbose=0)
def one_vs_rest_svm(train_x, train_y, test_x):
    clf = LinearSVC(random_state = 0, C=0.1)
    clf.fit(train_x, train_y)
    return clf.predict(test_x)
```

[ ]:
```
n, m, d = 5, 3, 7
train_x = np.random.random((n, d))
test_x = train_x[:m]
train_y = np.zeros(n)
train_y[-1] = 1
print(train_x, train_y)
```

[ ]:
```
one_vs_rest_svm(train_x, train_y, test_x)
```

1.3.2 Binary classification error

Report the test error by running run_svm_one_vs_rest_on_MNIST.
Error = 0.0075

1.3.3 Implement C-SVM

Play with the C parameter of SVM, what statement is true about the C parameter? (Choose all that apply.)
× Larger C gives larger tolerance of violation.
∨ Larger C gives smaller tolerance of violation.
× Larger C gives a larger-margin separating hyperplane.
∨ Larger C gives a smaller-margin separating hyperplane.

1.3.4 Multiclass SVM

In fact, sklearn already implements a multiclass SVM with a one-vs-rest strategy. Use LinearSVC to build a multiclass SVM model

Available Functions: You have access to the sklearn's implementation of the linear SVM as LinearSVC; No need to import anything.

[ ]:
```
def one_vs_rest_svm(train_x, train_y, test_x):
    """
    Trains a linear SVM for binary classifciation

    Args:
        train_x - (n, d) NumPy array (n datapoints each with d features)
        train_y - (n, ) NumPy array containing the labels (0 or 1) for each␣
↪training data point
        test_x - (m, d) NumPy array (m datapoints each with d features)
    Returns:
        pred_test_y - (m,) NumPy array containing the labels (0 or 1) for each␣
↪test data point
    """
    clf = LinearSVC(multi_class='ovr', random_state = 0, C=0.1)
    clf.fit(train_x, train_y)
    return clf.predict(test_x)

one_vs_rest_svm(train_x, train_y, test_x)
```

1.3.5 Multiclass SVM error

Report the overall test error by running run_multiclass_svm_on_MNIST.
Error = 0.0819

1.4 4. Multinomial (Softmax) Regression and Gradient Descent

Daniel suggests that instead of building ten models, we can expand a single logistic regression model into a multinomial regression and solve it with similar gradient descent algorithm.

The main function which you will call to run the code you will implement in this section is run_softmax_on_MNIST in main.py (already implemented). In the appendix at the bottom of this page, we describe a number of the methods that are already implemented for you in softmax.py that will be useful.

In order for the regression to work, you will need to implement three methods. Below we describe what the functions should do. We have included some test cases in test.py to help you verify that the methods you have implemented are behaving sensibly.

You will be working in the file part1/softmax.py in this problem

[ ]:
```
def augment_feature_vector(X):
    """
    Adds the x[i][0] = 1 feature for each data point x[i].

    Args:
        X - a NumPy matrix of n data points, each with d - 1 features

    Returns: X_augment, an (n, d) NumPy array with the added feature for each␣
↪datapoint
    """
    column_of_ones = np.zeros([len(X), 1]) + 1
    return np.hstack((column_of_ones, X))
```

[ ]:
```
def softmax_regression(X, Y, temp_parameter, alpha, lambda_factor, k,␣
↪num_iterations):
    """
    Runs batch gradient descent for a specified number of iterations on a␣
↪dataset
    with theta initialized to the all-zeros array. Here, theta is a k by d␣
↪NumPy array
    where row j represents the parameters of our model for label j for
    j = 0, 1, ..., k-1

    Args:
        X - (n, d - 1) NumPy array (n data points, each with d-1 features)
        Y - (n, ) NumPy array containing the labels (a number from 0-9) for␣
↪each
            data point
        temp_parameter - the temperature parameter of softmax function (scalar)
        alpha - the learning rate (scalar)
        lambda_factor - the regularization constant (scalar)
        k - the number of labels (scalar)
        num_iterations - the number of iterations to run gradient descent␣
↪(scalar)

    Returns:
        theta - (k, d) NumPy array that is the final value of parameters theta
        cost_function_progression - a Python list containing the cost␣
↪calculated at each step of gradient descent
```

```
    """
    X = augment_feature_vector(X)
    theta = np.zeros([k, X.shape[1]])
    cost_function_progression = []
    for i in range(num_iterations):
        cost_function_progression.append(compute_cost_function(X, Y, theta,␣
↪lambda_factor, temp_parameter))
        theta = run_gradient_descent_iteration(X, Y, theta, alpha,␣
↪lambda_factor, temp_parameter)
    return theta, cost_function_progression
```

[ ]:
```
def plot_cost_function_over_time(cost_function_history):
    plt.plot(range(len(cost_function_history)), cost_function_history)
    plt.ylabel('Cost Function')
    plt.xlabel('Iteration number')
    plt.show()
```

[ ]:
```
def compute_test_error(X, Y, theta, temp_parameter):
    error_count = 0.
    assigned_labels = get_classification(X, theta, temp_parameter)
    return 1 - np.mean(assigned_labels == Y)
```

1.4.1 Computing Probabilities for Softmax

Write a function compute_probabilities that computes, for each data point $x^{(i)}, the probability that $x^{(i)} is labeled as j for j=0,1,...,k1.

The softmax function h for a particular vector x requires computing

$$h(x) = \frac{1}{\sum_{j=1}^{k} e^{\theta_j x/\tau}} \begin{vmatrix} e^{\theta_1 x/\tau} \\ e^{\theta_2 x/\tau} \\ . \\ e^{\theta_k x/\tau} \end{vmatrix}$$

where >0 is the temperature parameter . When computing the output probabilities (they should always be in the range [0,1]), the terms $e^{\theta_j x/\tau}$ may be very large or very small, due to the use of the exponential function. This can cause numerical or overflow errors. To deal with this, we can simply subtract some fixed amount c from each exponent to keep the resulting number from getting too large. Since

$$h(x) = \frac{1}{\sum_{j=1}^{k} e^{[\theta_j x/\tau]-c}} \begin{vmatrix} e^{[\theta_1 x/\tau]-c} \\ e^{[\theta_2 x/\tau]-c} \\ . \\ e^{[\theta_k x/\tau]-c} \end{vmatrix}$$

subtracting some fixed amount c from each exponent will not change the final probabilities. A suitable choice for this fixed amount is $c = max_j \theta_j x/\tau$.

Reminder: You can implement this function locally first, and run python test.py in your project1 directory to validate basic functionality before checking against the online grader here.

Available Functions: You have access to the NumPy python library as np; No need to import anything.

[ ]:
```
def compute_probabilities(X, theta, temp_parameter):
    M = np.dot(theta, X.T)/temp_parameter
    # subtract max value by column
    H = np.exp(M-M.max(0))
    # probabiliy by column (devide by column sum)
    return H/H.sum(axis=0)[None,:]
```

[ ]:
```
n, d, k = 3, 5, 7
X = np.arange(0, n * d).reshape(n, d)
zeros = np.zeros((k, d))
temp_parameter = 0.2
theta = np.arange(0, k * d).reshape(k, d)
```

[ ]:
```
compute_probabilities(X, theta, temp_parameter)
```

1.4.2 Cost Function

Write a function compute_cost_function that computes the total cost over every data point.

The cost function J() is given by: (Use natural log)

$ $J(\theta) = -\frac{1}{n}\left[\sum_{i=1}^{n}\sum_{j=1}^{k}[[y^{(i)} == j]]log\frac{e^{\theta_j x^{(i)}/r}}{\sum_{l=1}^{k} e^{\theta_l x^{(i)}/r}}\right] + \frac{\lambda}{2}\sum$

Write a function run_gradient_descent_iteration that runs one step of batch gradient descent.

The partial derivative of J() wrt a particular $_j$ is given by:

$ ∇_{Θ}

Finally, report the final test error by running the main.py file, using the temperature parameter =1. If you have implemented everything correctly, the error on the test set should be around 0.1, which implies the linear softmax regression model is able to recognize MNIST digits with around 90 percent accuracy.

Note: For this project we will be looking at the error rate defined as the fraction of labels that don't match the target labels, also known as the "gold labels" or ground truth. In other contexts, you might want to consider other performance measures we've learned about in this class, including precision and recall.

Please enter the test error of your Softmax algorithm (copy the output from the main.py run).
0.1005

1.5 5. Temperature

We will now explore the effects of the temperature parameter in our algorithm.

You will be working in the files part1/main.py and part1/softmax.py in this problem

1.5.1 Effects of Adjusting Temperature

Explain how the temperature parameter affects the probability of a sample $x(i)$ being assigned a label that has a large . What about a small ?

× Larger temperature leads to less variance

$\vee$ Smaller temperature leads to less variance
$\times$ Smaller temperature makes the distribution more uniform

1.5.2 Reporting Error Rates

Set the temperature parameter to be 0.5, 1, and 2; re-run run_softmax_on_MNIST for each one of these (add your code to the specified part in main.py).
$Error|_{T=0.5} = 0.084$
$Error|_{T=1} = 0.1005$
$Error|_{T=2} = 0.1246$

1.6 6. Changing Labels

We now wish to classify the digits by their (mod 3) value, such that the new label $y^{(i)}$ of sample iis the old $y^{(i)}(mod 3)$. (Reminder: Return the temp_parameter to be 1 if you changed it for the last section)

You will be working in the file part1/main.py and part1/softmax.py in this problem

1.6.1 Using the Current Model - update target

Given that we already classified every $x^{(i)}$ as a digit, we could use the model we already trained and just calculate our estimations (mod 3).

Implement update_y function, which changes the old digit labels for the training and test set for the new (mod 3) labels.

Available Functions: You have access to the NumPy python library as np

[ ]:
```
def update_y(train_y, test_y):
    return train_y%3, test_y%3
```

1.6.2 Using the Current Model - compute test error

Implement compute_test_error_mod3 function, which takes the test points X, their correct labels Y (digits (mod 3) from 0-2), theta, and the temp_parameter, and returns the error.

Example:

x	Estimated Y	Estimated Y(mod 3)	Estimated X	Estimated X (mod 3)
x_1	9	0	8	2
x_2	6	0	6	0
x_3	5	3	8	2

The error of the regression with the original labels would be 0.66667

However, the error of the regression when comparing the (mod 3) of the labels would be 0.33333

Available Functions: You have access to the NumPy python library as np and to the

get_classification function from the project release

[ ]:
```
def get_classification(X, theta, temp_parameter):
    """
    Makes predictions by classifying a given dataset

    Args:
        X - (n, d - 1) NumPy array (n data points, each with d - 1 features)
        theta - (k, d) NumPy array where row j represents the parameters of our␣
↪model for
                label j
        temp_parameter - the temperature parameter of softmax function (scalar)

    Returns:
        Y - (n, ) NumPy array, containing the predicted label (a number between␣
↪0-9) for
            each data point
    """
    X = augment_feature_vector(X)
    probabilities = compute_probabilities(X, theta, temp_parameter)
    return np.argmax(probabilities, axis = 0)
```

[ ]:
```
def compute_test_error_mod3(X, Y, theta, temp_parameter):
    """
    Returns the error of these new labels when the classifier predicts the␣
↪digit. (mod 3)

    Args:
        X - (n, d - 1) NumPy array (n datapoints each with d - 1 features)
        Y - (n, ) NumPy array containing the labels (a number from 0-2) for␣
↪each
            data point
        theta - (k, d) NumPy array, where row j represents the parameters of␣
↪our
                model for label j
        temp_parameter - the temperature parameter of softmax function (scalar)

    Returns:
        test_error - the error rate of the classifier (scalar)
    """
    #YOUR CODE HERE
    predicted_Y = get_classification(X, theta, temp_parameter)%3
    return 1 - np.mean(predicted_Y == Y)
```

1.6.3 Using the Current Model - test error

Find the error rate of the new labels (call these two functions at the end of run_softmax_on_MNIST). See the functions' documentation for detailed explanations of the inputs and outputs.

Error rate for labels mod 3: 0.0768

1.6.4 Retrain with New Labels

Now suppose that instead we want to retrain our classifier with the new labels. In other words, rather than training the model to predict the original digits and then taking those predictions modulo 3, we explicitly train the model to predict the digits modulo 3 from the original image.

How do you expect the performance to change using the new labels?

$\times$ Increase

$\vee$ Decrease

$\times$ Stay the same

Implement run_softmax_on_MNIST_mod3 in main.py to perform this new training; report the new error rate.

Error rate when trained on labels mod 3: 0.1872

1.7 7. Classification Using Manually Crafted Features

The performance of most learning algorithms depends heavily on the representation of the training data. In this section, we will try representing each image using different features in place of the raw pixel values. Subsequently, we will investigate how well our regression model from the previous section performs when fed different representations of the data.

1.7.1 Dimensionality Reduction via PCA

Principal Components Analysis (PCA) is the most popular method for linear dimension reduction of data and is widely used in data analysis. For an in-depth exposition see: http://snobear.colorado.edu/Markw/BioMath/Otis/PCA/principal_components.ps.

Briefly, this method finds (orthogonal) directions of maximal variation in the data. By projecting an nŒd dataset X onto kd of these directions, we get a new dataset of lower dimension that reflects more variation in the original data than any other k-dimensional linear projection of X. By going through some linear algebra, it can be proven that these directions are equal to the k eigenvectors corresponding to the k largest eigenvalues of the covariance matrix $\widetilde{X}^T\widetilde{X}$, where $\widetilde{X}$ is a centered version of our original data.

Remark: The best implementations of PCA actually use the Singular Value Decomposition of $\widetilde{X}$ rather than the more straightforward approach outlined here, but these concepts are beyond the scope of this course.

1.7.2 Cubic Features

In this section, we will also work with a cubic feature mapping which maps an input vector $x = [x_1, \ldots, x_d]$ into a new feature vector (x), defined so that for any x, xR^d:

$$(x)^T(x) = (x^T x + 1)^3$$

1.8 8. Dimensionality Reduction Using PCA

PCA finds (orthogonal) directions of maximal variation in the data. In this problem we're going to project our data onto the principal components and explore the effects on performance.

You will be working in the files part1/main.py and part1/features.py in this problem

1.8.1 Project onto Principal Components

Fill in function project_onto_PC in features.py that implements PCA dimensionality reduction of dataset X.

Note that to project a given nŒd dataset X into its k-dimensional PCA representation, one can use matrix multiplication, after first centering X : $\widetilde{X}V$, where $\widetilde{X}$ is the centered version of the original data X and V is the dŒk matrix whose columns are the top k eigenvectors of $\widetilde{X}^T\widetilde{X}$. This is because the eigenvectors are of unit-norm, so there is no need to divide by their length.

You are given the full principal component matrix V as pcs in this function.

Available Functions: You have access to the NumPy python library as np and the function center_data which returns a centered version of the data, where each feature now has mean = 0

[ ]:
```
import numpy as np
from numpy import linalg
```

[ ]:
```
def cubic_features(X):
    """
    Returns a new dataset with features given by the mapping
    which corresponds to the cubic kernel.
    """
    n, d = X.shape  # dataset size, input dimension
    X_withones = np.ones((n, d + 1))
    X_withones[:, :-1] = X
    new_d = 0  # dimension of output
    new_d = int((d + 1) * (d + 2) * (d + 3) / 6)

    new_data = np.zeros((n, new_d))
    col_index = 0
    for x_i in range(n):
        X_i = X[x_i]
        X_i = X_i.reshape(1, X_i.size)

        if d > 2:
            comb_2 = np.matmul(np.transpose(X_i), X_i)
```

```
            unique_2 = comb_2[np.triu_indices(d, 1)]
            unique_2 = unique_2.reshape(unique_2.size, 1)
            comb_3 = np.matmul(unique_2, X_i)
            keep_m = np.zeros(comb_3.shape)
            index = 0
            for i in range(d - 1):
                keep_m[index + np.arange(d - 1 - i), i] = 0

                tri_keep = np.triu_indices(d - 1 - i, 1)

                correct_0 = tri_keep[0] + index
                correct_1 = tri_keep[1] + i + 1

                keep_m[correct_0, correct_1] = 1
                index += d - 1 - i

            unique_3 = np.sqrt(6) * comb_3[np.nonzero(keep_m)]

            new_data[x_i, np.arange(unique_3.size)] = unique_3
            col_index = unique_3.size

    for i in range(n):
        newdata_colindex = col_index
        for j in range(d + 1):
            new_data[i, newdata_colindex] = X_withones[i, j]**3
            newdata_colindex += 1
            for k in range(j + 1, d + 1):
                new_data[i, newdata_colindex] = X_withones[i, j]**2 *␣
↪X_withones[i, k] * (3**(0.5))
                newdata_colindex += 1

                new_data[i, newdata_colindex] = X_withones[i, j] *␣
↪X_withones[i, k]**2 * (3**(0.5))
                newdata_colindex += 1

                if k < d:
                    new_data[i, newdata_colindex] = X_withones[i, j] *␣
↪X_withones[i, k] * (6**(0.5))
                    newdata_colindex += 1

    return new_data
```

[ ]:
```
def center_data(X):
    feature_means = X.mean(axis=0)
    return(X - feature_means)
```

[ ]:
```
def principal_components(X):
    """
    Returns the principal component vectors of the data, sorted in decreasing␣
↪order
    of eigenvalue magnitude. This function first caluclates the covariance␣
↪matrix
    and then finds its eigenvectors.

    Args:
        X - n x d NumPy array of n data points, each with d features

    Returns:
        d x d NumPy array whose columns are the principal component directions␣
↪sorted
        in descending order by the amount of variation each direction (these␣
↪are
        equivalent to the d eigenvectors of the covariance matrix sorted in␣
↪descending
        order of eigenvalues, so the first column corresponds to the␣
↪eigenvector with
        the largest eigenvalue
    """
    centered_data = center_data(X)  # first center data
    scatter_matrix = np.dot(centered_data.transpose(), centered_data)
    eigen_values, eigedef plot_PC(X, pcs, labels):
    """
    Given the principal component vectors as the columns of matrix pcs,
    this function projects each sample in X onto the first two principal␣
↪components
    and produces a scatterplot where points are marked with the digit depicted␣
↪in
    the corresponding image.
    labels = a numpy array containing the digits corresponding to each image in␣
↪X.
    """
    pc_data = project_onto_PC(X, pcs, n_components=2)
    text_labels = [str(z) for z in labels.tolist()]
    fig, ax = plt.subplots()
    ax.scatter(pc_data[:, 0], pc_data[:, 1], alpha=0, marker=".")
    for i, txt in enumerate(text_labels):
        ax.annotate(txt, (pc_data[i, 0], pc_data[i, 1]))
    ax.set_xlabel('PC 1')
    ax.set_ylabel('PC 2')
    plt.show()n_vectors = np.linalg.eig(scatter_matrix)
    # Re-order eigenvectors by eigenvalue magnitude:
    idx = eigen_values.argsort()[::-1]
```

```
    eigen_values = eigen_values[idx]
    eigen_vectors = eigen_vectors[:, idx]
    return eigen_vectors
```

[ ]:
```
def reconstruct_PC(x_pca, pcs, n_components, X):
    """
    Given the principal component vectors as the columns of matrix pcs,
    this function reconstructs a single image from its principal component
    representation, x_pca.
    X = the original data to which PCA was applied to get pcs.
    """
    feature_means = X - center_data(X)
    feature_means = feature_means[0, :]
    x_reconstructed = np.dot(x_pca, pcs[:, range(n_components)].T) +␣
 ↪feature_means
    return x_reconstructed
```

[ ]:
```
def project_onto_PC(X, pcs, n_components):
    """
    Given principal component vectors pcs = principal_components(X)
    this function returns a new data array in which each sample in X
    has been projected onto the first n_components princicpal components.
    """
    # TODO: first center data using the centerData() function.
    # TODO: Return the projection of the centered dataset
    #       on the first n_components principal components.
    #       This should be an array with dimensions: n x n_components.
    # Hint: these principal components = first n_components columns
    #       of the eigenvectors returned by principal_components().
    #       Note that each eigenvector is already be a unit-vector,
    #       so the projection may be done using matrix multiplication.
    #n, d = X.shape
    #k = min(n, d)
    center_X = center_data(X)
    project_X = np.dot(center_X, pcs)[:, :n_components]
    return project_X
```

[ ]:
```
X = np.array([[0, 1, 2, 3], [1, 2, 3, 4], [2, 3, 4, 5]])
pcs = np.array([[-0.5       , -0.64641535, -0.8660254 , -0.8660254 ],
       [-0.5       , -0.32788993,  0.28867513,  0.28867513],
       [-0.5       ,  0.48715264,  0.28867513,  0.28867513],
       [-0.5       ,  0.48715264,  0.28867513,  0.28867513]])
n_components = 3
print(X.shape, pcs.shape)
np.array([[2, 0, 0], [0, 0, 0], [-2, 0, 0] ])
```

[ ]:
```
project_onto_PC(X, pcs, n_components)
```

Note: we only use the training dataset to determine the principal components. It is improper to use the test dataset for anything except evaluating the accuracy of our predictive model. If the

test data is used for other purposes such as selecting good features, it is possible to overfit the test set and obtain overconfident estimates of a model's performance.

1.8.2 Testing PCA

Use project_onto_PC to compute a 18-dimensional PCA representation of the MNIST training and test datasets, as illustrated in main.py.

Retrain your softmax regression model (using the original labels) on the MNIST training dataset and report its error on the test data, this time using these 18-dimensional PCA-representations rather than the raw pixel values.

If your PCA implementation is correct, the model should perform nearly as well when only given 18 numbers encoding each image as compared to the 784 in the original data (error on the test set using PCA features should be around 0.15). This is because PCA ensures these 18 feature values capture the maximal amount of variation from the original 784-dimensional data.

Error rate for 18-dimensional PCA features = 0.1483

1.8.3 Testing PCA (continued)

Use plot_PC in main.py to visualize the first 100 MNIST images, as represented in the space spanned by the first 2 principal components of the training data.

What does your PCA look like?

Use the calls to plot_images() and reconstruct_PC in main.py to plot the reconstructions of the first two MNIST images (from their 18-dimensional PCA-representations) alongside the originals.

Remark: Two dimensional PCA plots offer a nice way to visualize some global structure in high-dimensional data, although approaches based on nonlinear dimension reduction may be more insightful in certain cases. Notice that for our data, the first 2 principal components are insufficent for fully separating the different classes of MNIST digits.

1.9 9. Cubic Features

In this section, we will work with a cubic feature mapping which maps an input vector $x = [x_1, \ldots, x_d]$ into a new feature vector (x), defined so that for any x, xR^d:

$(x)^T(x) = (x^T x + 1)^3$

You will be working in the files part1/main.py and part1/features.py in this problem

1.9.1 Computing Cubic Features

In 2-D, let $x = [x_1, x_2]$. Write down the explicit cubic feature mapping (x)as a vector; i.e., $(x) = [f_1(x_1, x_2), \ldots, f_N(x_1, x_2)]$

Hint: (x) should be a 10-dimensional vector.

$$
\begin{aligned}
(x^T x + 1)^3 &= (\begin{vmatrix} x_1 \\ x_1^{'} \end{vmatrix} * \begin{vmatrix} x_2 \\ x_2^{'} \end{vmatrix} + 1)^3 \\
&= (1 + x_1 x_1^{'} + x_2 x_2^{'})^3 \\
&= \left(1 + (x_1 x_1^{'})^2 + (x_2 x_2^{'})^2 + 2(x_1 x_1^{'}) + 2(x_2 x_2^{'}) + 2(x_1 x_1^{'})(x_2 x_2^{'})\right)(1 + x_1 x_1^{'} + x_2 x_2^{'}) \\
&= 1 + 3(x_1 x_1^{'}) + 3(x_2 x_2^{'}) + 3(x_1 x_1^{'})^2 + 3(x_2 x_2^{'})^2 + (x_1 x_1^{'})^3 + (x_2 x_2^{'})^3 + 6(x_1 x_1^{'})(x_2 x_2^{'}) + 3(x_1 x_1^{'})(x_2 x_2^{'})^2 + \\
&= \begin{vmatrix} 1 \\ \sqrt{3}x_1 \\ \sqrt{3}x_2 \\ \sqrt{3}x_1^2 \\ \sqrt{3}x_2^2 \\ x_1^3 \\ x_2^3 \\ \sqrt{6}x_1 x_2 \\ \sqrt{3}x_1 x_2^2 \\ \sqrt{3}x_1^2 x_2 \end{vmatrix} * \begin{vmatrix} 1 \\ \sqrt{3}x_1^{'} \\ \sqrt{3}x_2^{'} \\ \sqrt{3}(x_1^{'})^2 \\ \sqrt{3}(x_2^{'})^2 \\ (x_1^{'})^3 \\ (x_2^{'})^3 \\ \sqrt{6}x_1^{'}x_2^{'} \\ \sqrt{3}x_1^{'}(x_2^{'})^2 \\ \sqrt{3}(x_1^{'})^2 x_2 \end{vmatrix}
\end{aligned}
$$

The cubic_features function in features.py is already implemented for you. That function can handle input with an arbitrary dimension and compute the corresponding features for the cubic Kernel. Note that here we don't leverage the kernel properties that allow us to do a more efficient computation with the kernel function (without computing the features themselves). Instead, here we do compute the cubic features explicitly and apply the PCA on the output features.

1.9.2 Applying to MNIST

If we explicitly apply the cubic feature mapping to the original 784-dimensional raw pixel features, the resulting representation would be of massive dimensionality. Instead, we will apply the quadratic feature mapping to the 10-dimensional PCA representation of our training data which we will have to calculate just as we calculated the 18-dimensional representation in the previous problem. After applying the cubic feature mapping to the PCA representations for both the train and test datasets, retrain the softmax regression model using these new features and report the resulting test set error below.

Important: You will probably get a runtime warning for getting the log of 0, ignore. Your code should still run and perform correctly.

Note: Use the same training parameters as the first softmax model given in main.py file and temperature 1.

If you have done everything correctly, softmax regression should perform better (on the test set) using these features than either the 18-dimensional principal components or raw pixels. The error on the test set using cubic features should only be around 0.08, demonstrating the power of nonlinear classification models.

Error rate for 10-dimensional cubic PCA features = 0.0867

1.10 10. Kernel Methods

As you can see, implementing a direct mapping to the high-dimensional features is a lot of work (imagine using an even higher dimensional feature mapping.) This is where the kernel trick becomes useful.

Recall the kernel perceptron algorithm we learned in the lecture. The weights can be represented by a linear combination of features:

$\theta = \sum_{i=1}^{n} \alpha^{(i)} y^{(i)} \phi(x^{(i)})$

In the softmax regression fomulation, we can also apply this representation of the weights:

$\theta_j = \sum_{i=1}^{n} \alpha_j^{(i)} y^{(i)} \phi(x^{(i)})$

$$h(x) = \frac{1}{\sum_{j=1}^{k} e^{[\theta_j \phi(x)/\tau] - c}} \begin{bmatrix} e^{[\theta_1 \phi(x)/\tau] - c} \\ e^{[\theta_2 \phi(x)/\tau] - c} \\ . \\ e^{[\theta_k \phi(x)/\tau] - c} \end{bmatrix}$$

$$= \frac{1}{\sum_{j=1}^{k} e^{[\sum_{i=1}^{n} \alpha_j^{(i)} y^{(i)} \phi(x^{(i)}) \phi(x)/\tau] - c}} \begin{bmatrix} e^{[\sum_{i=1}^{n} \alpha_1^{(i)} y^{(i)} \phi(x^{(i)}) \phi(x)/\tau] - c} \\ e^{[\sum_{i=1}^{n} \alpha_2^{(i)} y^{(i)} \phi(x^{(i)}) \phi(x)/\tau] - c} \\ . \\ e^{[\sum_{i=1}^{n} \alpha_k^{(i)} y^{(i)} \phi(x^{(i)}) \phi(x)/\tau] - c} \end{bmatrix}$$

We actually do not need the real mapping (x), but the inner product between two features after mapping: $(x_i)(x)$, where x_i is a point in the training set and x is the new data point for which we want to compute the probability. If we can create a kernel function $K(x, y) = (x)(y)$, for any two points x and y, we can then kernelize our softmax regression algorithm.

You will be working in the files part1/main.py and part1/kernel.py in this problem

1.10.1 Implementing Polynomial Kernel

In the last section, we explicitly created a cubic feature mapping. Now, suppose we want to map the features into d dimensional polynomial space,

$(x) = x_d^2, \ldots, x_1^2, \sqrt{2}x_d x_{d1}, \ldots, \sqrt{2}x_d x_1, \sqrt{2}x_{d1} x_{d2}, \ldots, \sqrt{2}x_{d1} x_1, \ldots, \sqrt{2}x_2 x_1, \sqrt{2c}x_d, \ldots, \sqrt{2c}x_1, c$

Write a function polynomial_kernel that takes in two matrix X and Y and computes the polynomial kernel K(x,y) for every pair of rows x in X and y in Y . Available Functions: You have access to the NumPy python library as np

[ ]:
```
import numpy as np
```

[ ]:
```
def polynomial_kernel(X, Y, c, p):
    """
        Compute the polynomial kernel between two matrices X and Y::
            K(x, y) = (<x, y> + c)^p
        for each pair of rows x in X and y in Y.

        Args:
            X - (n, d) NumPy array (n datapoints each with d features)
            Y - (m, d) NumPy array (m datapoints each with d features)
            c - a coefficient to trade off high-order and low-order terms␣
 ↪(scalar)
```

```
            p - the degree of the polynomial kernel

        Returns:
            kernel_matrix - (n, m) Numpy array containing the kernel matrix
    """
    # YOUR CODE HERE
    return (np.dot(X, np.transpose(Y)) + c)**p
```

[ ]:
```
n, m, d = 3, 5, 7
c = 1
p = 2
X = np.random.random((n, d))
Y = np.random.random((m, d))
expect = np.array([[47.27173974362664,2468.2223462654674, 592.9316296853342,␣
↪2472.839712680877, 3683.09728317986],
[138.481779223827, 1743.668766562655, 3078.2980365011017, 4133.1119714528595,␣
↪1906.575031585209],
[104.17032579535437, 1145.306633695184, 839.7998782098495, 2197.7060334184293,␣
↪2951.6746714623982]])
expect
```

[ ]:
```
polynomial_kernel(X, Y, c, d)
```

1.10.2 Gaussian RBF Kernel

Another commonly used kernel is the Gaussian RBF kenel. Similarly, write a function rbf_kernel that takes in two matrices X and Y and computes the RBF kernel K(x,y) for every pair of rows x in X and y in Y.

Available Functions: You have access to the NumPy python library as np

[ ]:
```
def rbf_kernel(X, Y, gamma):
    """
        Compute the Gaussian RBF kernel between two matrices X and Y::
            K(x, y) = exp(-gamma ||x-y||^2)
        for each pair of rows x in X and y in Y.

        Args:
            X - (n, d) NumPy array (n datapoints each with d features)
            Y - (m, d) NumPy array (m datapoints each with d features)
            gamma - the gamma parameter of gaussian function (scalar)

        Returns:
            kernel_matrix - (n, m) Numpy array containing the kernel matrix
    """
    # YOUR CODE HERE
    kernel_matrix = np.exp(-gamma * np.array([np.linalg.norm(rowX - Y,␣
↪axis=1)**2 for rowX in X]))
    return kernel_matrix
```

[ ]:
```
n, m, d = 3, 5, 7
gamma = 0.5
X = np.random.random((n, d))
Y = np.random.random((m, d))
```

[ ]:
```
rbf_kernel(X, Y, gamma):
```

[ ]:
```
np.array([np.exp(-gamma * (np.linalg.norm(rowX - rowY) ** 2)) for rowX in X for␣
 ↪rowY in Y]).reshape(n, m)
```

[ ]:
```
np.exp(-gamma * np.array([np.linalg.norm(rowX - Y, axis=1)**2 for rowX in X]))
```

Now, try implementing the softmax regression using kernelized features. You will have to rewrite the softmax_regression function in softmax.py, as well as the auxiliary functions compute_cost_function, compute_probabilities, run_gradient_descent_iteration.

How does the test error change?

In this project, you have been familiarized with the MNIST dataset for digit recognition, a popular task in computer vision.

You have implemented a linear regression which turned out to be inadequate for this task. You have also learned how to use scikit-learn's SVM for binary classification and multiclass classification.

Then, you have implemented your own softmax regression using gradient descent.

Finally, you experimented with different hyperparameters, different labels and different features, including kernelized features.

In the next project, you will apply neural networks to this task.

1.11 Attachment

1.11.1 linear_regression.py

[ ]:
```
import numpy as np

### Functions for you to fill in ###

def closed_form(X, Y, lambda_factor):
    """
    Computes the closed form solution of linear regression with L2␣
 ↪regularization

    Args:
        X - (n, d + 1) NumPy array (n datapoints each with d features plus the␣
 ↪bias feature in the first dimension)
        Y - (n, ) NumPy array containing the labels (a number from 0-9) for␣
 ↪each
            data point
        lambda_factor - the regularization constant (scalar)
    Returns:
        theta - (d + 1, ) NumPy array containing the weights of linear␣
 ↪regression. Note that theta[0]
```

```
        represents the y-axis intercept of the model and therefore X[0] = 1
    """
    # YOUR CODE HERE
    n, d = X.shape
    return np.dot(np.linalg.inv(np.dot(np.transpose(X), X) + lambda_factor*np.
 ↪eye(d,d)), np.dot(np.transpose(X), Y))
    raise NotImplementedError

### Functions which are already complete, for you to use ###

def compute_test_error_linear(test_x, Y, theta):
    test_y_predict = np.round(np.dot(test_x, theta))
    test_y_predict[test_y_predict < 0] = 0
    test_y_predict[test_y_predict > 9] = 9
    return 1 - np.mean(test_y_predict == Y)
```

1.11.2 svm.py

[ ]:
```
import numpy as np
from sklearn.svm import LinearSVC


### Functions for you to fill in ###

def one_vs_rest_svm(train_x, train_y, test_x):
    """
    Trains a linear SVM for binary classifciation

    Args:
        train_x - (n, d) NumPy array (n datapoints each with d features)
        train_y - (n, ) NumPy array containing the labels (0 or 1) for each␣
 ↪training data point
        test_x - (m, d) NumPy array (m datapoints each with d features)
    Returns:
        pred_test_y - (m,) NumPy array containing the labels (0 or 1) for each␣
 ↪test data point
    """
    # import pdb; pdb.set_trace()
    clf = LinearSVC(random_state = 0, C=0.1)
    clf.fit(train_x, train_y)
    return clf.predict(test_x)

def one_vs_rest_svm_c(train_x, train_y, test_x, C):
    clf = LinearSVC(random_state = 0, C=C)
    clf.fit(train_x, train_y)
    return clf.predict(test_x)
```

```
def multi_class_svm(train_x, train_y, test_x):
    """
    Trains a linear SVM for multiclass classifciation using a one-vs-rest␣
 ↪strategy

    Args:
        train_x - (n, d) NumPy array (n datapoints each with d features)
        train_y - (n, ) NumPy array containing the labels (int) for each␣
 ↪training data point
        test_x - (m, d) NumPy array (m datapoints each with d features)
    Returns:
        pred_test_y - (m,) NumPy array containing the labels (int) for each␣
 ↪test data point
    """
    clf = LinearSVC(multi_class='ovr', random_state = 0, C=0.1)
    clf.fit(train_x, train_y)
    return clf.predict(test_x)


def compute_test_error_svm(test_y, pred_test_y):
    return 1 - np.mean(pred_test_y == test_y)
```

1.11.3 softmax.py

[ ]:
```
import sys
sys.path.append("..")
import utils
from utils import *
import numpy as np
import matplotlib.pyplot as plt
import scipy.sparse as sparse


def augment_feature_vector(X):
    """
    Adds the x[i][0] = 1 feature for each data point x[i].

    Args:
        X - a NumPy matrix of n data points, each with d - 1 features

    Returns: X_augment, an (n, d) NumPy array with the added feature for each␣
 ↪datapoint
    """
    column_of_ones = np.zeros([len(X), 1]) + 1
    return np.hstack((column_of_ones, X))
```

```
def compute_probabilities(X, theta, temp_parameter):
    """
    Computes, for each datapoint X[i], the probability that X[i] is labeled as␣
 ↪j
    for j = 0, 1, ..., k-1

    Args:
        X - (n, d) NumPy array (n datapoints each with d features)
        theta - (k, d) NumPy array, where row j represents the parameters of␣
 ↪our model for label j
        temp_parameter - the temperature parameter of softmax function (scalar)
    Returns:
        H - (k, n) NumPy array, where each entry H[j][i] is the probability␣
 ↪that X[i] is labeled as j
    """
    #YOUR CODE HERE
    M = np.dot(theta, X.T)/temp_parameter
    c = M.max(0)
    H = np.exp(M-c)
    return H/H.sum(axis=0)[None,:]


def compute_cost_function(X, Y, theta, lambda_factor, temp_parameter):
    """
    Computes the total cost over every datapoint.

    Args:
        X - (n, d) NumPy array (n datapoints each with d features)
        Y - (n, ) NumPy array containing the labels (a number from 0-9) for␣
 ↪each
            data point
        theta - (k, d) NumPy array, where row j represents the parameters of␣
 ↪our
                model for label j
        lambda_factor - the regularization constant (scalar)
        temp_parameter - the temperature parameter of softmax function (scalar)

    Returns
        c - the cost value (scalar)
    """
    #YOUR CODE HERE
    n, d = X.shape
    k, _ = theta.shape
    reg_term = lambda_factor/2*np.sum(theta**2)

    # P = np.log(compute_probabilities(X, theta, temp_parameter))
```

```
    P = np.log(np.clip(compute_probabilities(X, theta, temp_parameter), 1e-15,␣
 ↪1-1e-15))
    L = np.zeros((n, k))
    for i in range(n):
       for j in range(k):
           if Y[i] == j:
               L[i,j] = 1
    #L = sparse.coo_matrix(([1]*n, (Y, range(n))), shape=(k,n)).toarray()
    loss_term = -1/n*sum(np.dot(L[i:i+1,:], P[:,i:i+1]) for i in range(n))

    return np.asscalar(loss_term) + reg_term


def run_gradient_descent_iteration(X, Y, theta, alpha, lambda_factor,␣
 ↪temp_parameter):
    """
    Runs one step of batch gradient descent

    Args:
        X - (n, d) NumPy array (n datapoints each with d features)
        Y - (n, ) NumPy array containing the labels (a number from 0-9) for␣
 ↪each
            data point
        theta - (k, d) NumPy array, where row j represents the parameters of␣
 ↪our
                model for label j
        alpha - the learning rate (scalar)
        lambda_factor - the regularization constant (scalar)
        temp_parameter - the temperature parameter of softmax function (scalar)

    Returns:
        theta - (k, d) NumPy array that is the final value of parameters theta
    """
    #YOUR CODE HERE
    n, d = X.shape
    k, _ = theta.shape
    # get probability matrix
    P = compute_probabilities(X, theta, temp_parameter)
    # get y == j matrix
    M = sparse.coo_matrix(([1]*n, (Y, range(n))), shape=(k,n)).toarray()
    L = M - P
    delta_j = np.array([ -1/(temp_parameter * n ) * sum( X[i,:]*L[j,i] for i in␣
 ↪range(n) ) + lambda_factor * theta[j] for j in range(k) ])
    updated_theta = theta - alpha * delta_j
    return updated_theta

def update_y(train_y, test_y):
```

```
    """
    Changes the old digit labels for the training and test set for the new (mod 3)
    labels.

    Args:
        train_y - (n, ) NumPy array containing the labels (a number between 0-9)
                  for each datapoint in the training set
        test_y - (n, ) NumPy array containing the labels (a number between 0-9)
                 for each datapoint in the test set

    Returns:
        train_y_mod3 - (n, ) NumPy array containing the new labels (a number between 0-2)
                       for each datapoint in the training set
        test_y_mod3 - (n, ) NumPy array containing the new labels (a number between 0-2)
                      for each datapoint in the test set
    """
    #YOUR CODE HERE
    return train_y%3, test_y%3

def compute_test_error_mod3(X, Y, theta, temp_parameter):
    """
    Returns the error of these new labels when the classifier predicts the digit. (mod 3)

    Args:
        X - (n, d - 1) NumPy array (n datapoints each with d - 1 features)
        Y - (n, ) NumPy array containing the labels (a number from 0-2) for each
            data point
        theta - (k, d) NumPy array, where row j represents the parameters of our
                model for label j
        temp_parameter - the temperature parameter of softmax function (scalar)

    Returns:
        test_error - the error rate of the classifier (scalar)
    """
    #YOUR CODE HERE
    predicted_Y = get_classification(X, theta, temp_parameter)%3
    return 1 - np.mean(predicted_Y == Y)
```

```
def softmax_regression(X, Y, temp_parameter, alpha, lambda_factor, k,␣
↪num_iterations):
    """
    Runs batch gradient descent for a specified number of iterations on a␣
↪dataset
    with theta initialized to the all-zeros array. Here, theta is a k by d␣
↪NumPy array
    where row j represents the parameters of our model for label j for
    j = 0, 1, ..., k-1

    Args:
        X - (n, d - 1) NumPy array (n data points, each with d-1 features)
        Y - (n, ) NumPy array containing the labels (a number from 0-9) for␣
↪each
            data point
        temp_parameter - the temperature parameter of softmax function (scalar)
        alpha - the learning rate (scalar)
        lambda_factor - the regularization constant (scalar)
        k - the number of labels (scalar)
        num_iterations - the number of iterations to run gradient descent␣
↪(scalar)

    Returns:
        theta - (k, d) NumPy array that is the final value of parameters theta
        cost_function_progression - a Python list containing the cost␣
↪calculated at each step of gradient descent
    """
    X = augment_feature_vector(X)
    theta = np.zeros([k, X.shape[1]])
    cost_function_progression = []
    for i in range(num_iterations):
        cost_function_progression.append(compute_cost_function(X, Y, theta,␣
↪lambda_factor, temp_parameter))
        theta = run_gradient_descent_iteration(X, Y, theta, alpha,␣
↪lambda_factor, temp_parameter)
    return theta, cost_function_progression

def get_classification(X, theta, temp_parameter):
    """
    Makes predictions by classifying a given dataset

    Args:
        X - (n, d - 1) NumPy array (n data points, each with d - 1 features)
        theta - (k, d) NumPy array where row j represents the parameters of our␣
↪model for
                label j
```

```
        temp_parameter - the temperature parameter of softmax function (scalar)

    Returns:
        Y - (n, ) NumPy array, containing the predicted label (a number between␣
 ↪0-9) for
            each data point
    """
    X = augment_feature_vector(X)
    probabilities = compute_probabilities(X, theta, temp_parameter)
    return np.argmax(probabilities, axis = 0)

def plot_cost_function_over_time(cost_function_history):
    plt.plot(range(len(cost_function_history)), cost_function_history)
    plt.ylabel('Cost Function')
    plt.xlabel('Iteration number')
    plt.show()

def compute_test_error(X, Y, theta, temp_parameter):
    error_count = 0.
    assigned_labels = get_classification(X, theta, temp_parameter)
    return 1 - np.mean(assigned_labels == Y)
```

1.11.4 features.py

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt

def project_onto_PC(X, pcs, n_components):
    """
    Given principal component vectors pcs = principal_components(X)
    this function returns a new data array in which each sample in X
    has been projected onto the first n_components principcal components.
    """
    # TODO: first center data using the centerData() function.
    # TODO: Return the projection of the centered dataset
    #       on the first n_components principal components.
    #       This should be an array with dimensions: n x n_components.
    # Hint: these principal components = first n_components columns
    #       of the eigenvectors returned by principal_components().
    #       Note that each eigenvector is already be a unit-vector,
    #       so the projection may be done using matrix multiplication.
    #import pdb; pdb.set_trace()
    center_X = center_data(X)
    project_X = np.dot(center_X, pcs)[:, :n_components]
    #n_components = pcs[:k,:k]
    return project_X
```

```
### Functions which are already complete, for you to use ###

def cubic_features(X):
    """
    Returns a new dataset with features given by the mapping
    which corresponds to the cubic kernel.
    """
    n, d = X.shape  # dataset size, input dimension
    X_withones = np.ones((n, d + 1))
    X_withones[:, :-1] = X
    new_d = 0  # dimension of output
    new_d = int((d + 1) * (d + 2) * (d + 3) / 6)

    new_data = np.zeros((n, new_d))
    col_index = 0
    for x_i in range(n):
        X_i = X[x_i]
        X_i = X_i.reshape(1, X_i.size)

        if d > 2:
            comb_2 = np.matmul(np.transpose(X_i), X_i)

            unique_2 = comb_2[np.triu_indices(d, 1)]
            unique_2 = unique_2.reshape(unique_2.size, 1)
            comb_3 = np.matmul(unique_2, X_i)
            keep_m = np.zeros(comb_3.shape)
            index = 0
            for i in range(d - 1):
                keep_m[index + np.arange(d - 1 - i), i] = 0

                tri_keep = np.triu_indices(d - 1 - i, 1)

                correct_0 = tri_keep[0] + index
                correct_1 = tri_keep[1] + i + 1

                keep_m[correct_0, correct_1] = 1
                index += d - 1 - i

            unique_3 = np.sqrt(6) * comb_3[np.nonzero(keep_m)]

            new_data[x_i, np.arange(unique_3.size)] = unique_3
            col_index = unique_3.size

    for i in range(n):
        newdata_colindex = col_index
```

```
        for j in range(d + 1):
            new_data[i, newdata_colindex] = X_withones[i, j]**3
            newdata_colindex += 1
            for k in range(j + 1, d + 1):
                new_data[i, newdata_colindex] = X_withones[i, j]**2 *␣
↪X_withones[i, k] * (3**(0.5))
                newdata_colindex += 1

                new_data[i, newdata_colindex] = X_withones[i, j] *␣
↪X_withones[i, k]**2 * (3**(0.5))
                newdata_colindex += 1

                if k < d:
                    new_data[i, newdata_colindex] = X_withones[i, j] *␣
↪X_withones[i, k] * (6**(0.5))
                    newdata_colindex += 1

    return new_data


def center_data(X):
    """
    Returns a centered version of the data, where each feature now has mean = 0

    Args:
        X - n x d NumPy array of n data points, each with d features

    Returns:
        n x d NumPy array X' where for each i = 1, ..., n and j = 1, ..., d:
        X'[i][j] = X[i][j] - means[j]
    """
    feature_means = X.mean(axis=0)
    return(X - feature_means)


def principal_components(X):
    """
    Returns the principal component vectors of the data, sorted in decreasing␣
↪order
    of eigenvalue magnitude. This function first caluclates the covariance␣
↪matrix
    and then finds its eigenvectors.

    Args:
        X - n x d NumPy array of n data points, each with d features

    Returns:
```

```
        d x d NumPy array whose columns are the principal component directions␣
↪sorted
        in descending order by the amount of variation each direction (these␣
↪are
        equivalent to the d eigenvectors of the covariance matrix sorted in␣
↪descending
        order of eigenvalues, so the first column corresponds to the␣
↪eigenvector with
        the largest eigenvalue
    """
    centered_data = center_data(X)  # first center data
    scatter_matrix = np.dot(centered_data.transpose(), centered_data)
    eigen_values, eigen_vectors = np.linalg.eig(scatter_matrix)
    # Re-order eigenvectors by eigenvalue magnitude:
    idx = eigen_values.argsort()[::-1]
    eigen_values = eigen_values[idx]
    eigen_vectors = eigen_vectors[:, idx]
    return eigen_vectors


def plot_PC(X, pcs, labels):
    """
    Given the principal component vectors as the columns of matrix pcs,
    this function projects each sample in X onto the first two principal␣
↪components
    and produces a scatterplot where points are marked with the digit depicted␣
↪in
    the corresponding image.
    labels = a numpy array containing the digits corresponding to each image in␣
↪X.
    """
    pc_data = project_onto_PC(X, pcs, n_components=2)
    text_labels = [str(z) for z in labels.tolist()]
    fig, ax = plt.subplots()
    ax.scatter(pc_data[:, 0], pc_data[:, 1], alpha=0, marker=".")
    for i, txt in enumerate(text_labels):
        ax.annotate(txt, (pc_data[i, 0], pc_data[i, 1]))
    ax.set_xlabel('PC 1')
    ax.set_ylabel('PC 2')
    plt.show()


def reconstruct_PC(x_pca, pcs, n_components, X):
    """
    Given the principal component vectors as the columns of matrix pcs,
    this function reconstructs a single image from its principal component
```

```
    representation, x_pca.
    X = the original data to which PCA was applied to get pcs.
    """
    feature_means = X - center_data(X)
    feature_means = feature_means[0, :]
    x_reconstructed = np.dot(x_pca, pcs[:, range(n_components)].T) +␣
↪feature_means
    return x_reconstructed
```

1.11.5 kerrel.py

[ ]:
```
import numpy as np

### Functions for you to fill in ###



def polynomial_kernel(X, Y, c, p):
    """
        Compute the polynomial kernel between two matrices X and Y::
            K(x, y) = (<x, y> + c)^p
        for each pair of rows x in X and y in Y.

        Args:
            X - (n, d) NumPy array (n datapoints each with d features)
            Y - (m, d) NumPy array (m datapoints each with d features)
            c - an coefficient to trade off high-order and low-order terms␣
↪(scalar)
            p - the degree of the polynomial kernel

        Returns:
            kernel_matrix - (n, m) Numpy array containing the kernel matrix
    """
    # YOUR CODE HERE
    return (np.dot(X, np.transpose(Y)) + c)**p



def rbf_kernel(X, Y, gamma):
    """
        Compute the Gaussian RBF kernel between two matrices X and Y::
            K(x, y) = exp(-gamma ||x-y||^2)
        for each pair of rows x in X and y in Y.

        Args:
            X - (n, d) NumPy array (n datapoints each with d features)
```

```
            Y - (m, d) NumPy array (m datapoints each with d features)
            gamma - the gamma parameter of gaussian function (scalar)

        Returns:
            kernel_matrix - (n, m) Numpy array containing the kernel matrix
    """
    # YOUR CODE HERE
    kernel_matrix = np.exp(-gamma * np.array([np.linalg.norm(rowX - Y,␣
↪axis=1)**2 for rowX in X]))
    return kernel_matrix
```

1.11.6 main.py

[ ]:
```
import sys
import numpy as np
import matplotlib.pyplot as plt
sys.path.append("..")
from utils import *
from linear_regression import *
from svm import *
from softmax import *
from features import *
from kernel import *

#####################################################################
# 1. Introduction
#####################################################################

# Load MNIST data:
train_x, train_y, test_x, test_y = get_MNIST_data()
# Plot the first 20 images of the training set.
plot_images(train_x[0:20, :])

#####################################################################
# 2. Linear Regression with Closed Form Solution
#####################################################################

# TODO: first fill out functions in linear_regression.py, or the below␣
↪functions will not work

def run_linear_regression_on_MNIST(lambda_factor=1):
    """
    Trains linear regression, classifies test data, computes test error on test␣
↪set

    Returns:
```

```
        Final test error
    """
    train_x, train_y, test_x, test_y = get_MNIST_data()
    train_x_bias = np.hstack([np.ones([train_x.shape[0], 1]), train_x])
    test_x_bias = np.hstack([np.ones([test_x.shape[0], 1]), test_x])
    theta = closed_form(train_x, train_y, lambda_factor)
    test_error = compute_test_error_linear(test_x, test_y, theta)
    return test_error


# Don't run this until the relevant functions in linear_regression.py have been␣
 ↪fully implemented.
#print('Linear Regression test_error =',␣
 ↪run_linear_regression_on_MNIST(lambda_factor=1))


#####################################################################
# 3. Support Vector Machine
#####################################################################

# TODO: first fill out functions in svm.py, or the below functions will not␣
 ↪work

def run_svm_one_vs_rest_on_MNIST():
    """
    Trains svm, classifies test data, computes test error on test set

    Returns:
        Test error for the binary svm
    """
    train_x, train_y, test_x, test_y = get_MNIST_data()
    train_y[train_y != 0] = 1
    test_y[test_y != 0] = 1
    pred_test_y = one_vs_rest_svm(train_x, train_y, test_x)
    test_error = compute_test_error_svm(test_y, pred_test_y)
    return test_error


#print('SVM one vs. rest test_error:', run_svm_one_vs_rest_on_MNIST())


def run_multiclass_svm_on_MNIST():
    """
    Trains svm, classifies test data, computes test error on test set

    Returns:
        Test error for the binary svm
```

```
    """
    train_x, train_y, test_x, test_y = get_MNIST_data()
    pred_test_y = multi_class_svm(train_x, train_y, test_x)
    test_error = compute_test_error_svm(test_y, pred_test_y)
    return test_error


#print('Multiclass SVM test_error:', run_multiclass_svm_on_MNIST())

#############################################################################
# 4. Multinomial (Softmax) Regression and Gradient Descent
#############################################################################

# TODO: first fill out functions in softmax.py, or run_softmax_on_MNIST will␣
 ↪not work

def run_softmax_on_MNIST(temp_parameter=1):
    """
    Trains softmax, classifies test data, computes test error, and plots cost␣
 ↪function

    Runs softmax_regression on the MNIST training set and computes the test␣
 ↪error using
    the test set. It uses the following values for parameters:
    alpha = 0.3
    lambda = 1e-4
    num_iterations = 150

    Saves the final theta to ./theta.pkl.gz

    Returns:
        Final test error
    """
    train_x, train_y, test_x, test_y = get_MNIST_data()
    theta, cost_function_history = softmax_regression(train_x, train_y,␣
 ↪temp_parameter, alpha= 0.3, lambda_factor = 1.0e-4, k = 10, num_iterations =␣
 ↪150)
    plot_cost_function_over_time(cost_function_history)
    test_error = compute_test_error(test_x, test_y, theta, temp_parameter)
    # Save the model parameters theta obtained from calling softmax_regression␣
 ↪to disk.
    write_pickle_data(theta, "./theta.pkl.gz")

    # TODO: add your code here for the "Using the Current Model" question in␣
 ↪tab 4.
    #      and print the test_error_mod3
```

```
    return test_error

#print('softmax test_error=', run_softmax_on_MNIST(temp_parameter=1))

# TODO: Find the error rate for temp_parameter = [.5, 1.0, 2.0]
#       Remember to return the tempParameter to 1, and re-run␣
 ↪run_softmax_on_MNIST

############################################################################
# 6A. Changing Labels using original digits
############################################################################

def run_softmax_on_MNIST_mod3a(temp_parameter=1):
    """
    Trains Softmax regression on digit (mod 3) classifications.

    See run_softmax_on_MNIST for more info.
    """
    #YOUR CODE HERE
    train_x, train_y, test_x, test_y = get_MNIST_data()
    theta, cost_function_history = softmax_regression(train_x, train_y,␣
 ↪temp_parameter, alpha= 0.3, lambda_factor = 1.0e-4, k = 10, num_iterations =␣
 ↪150)
    plot_cost_function_over_time(cost_function_history)

    train_y, test_y = update_y(train_y, test_y)
    test_error = compute_test_error_mod3(test_x, test_y, theta, temp_parameter)
    # Save the model parameters theta obtained from calling softmax_regression␣
 ↪to disk.
    write_pickle_data(theta, "./theta_mod3.pkl.gz")

    # TODO: add your code here for the "Using the Current Model" question in␣
 ↪tab 4.
    #       and print the test_error_mod3
    return test_error

# TODO: Run run_softmax_on_MNIST_mod3(), report the error rate
#print('softmax test_error=', run_softmax_on_MNIST_mod3())


############################################################################
# 6. Changing Labels using mod3
############################################################################

def run_softmax_on_MNIST_mod3(temp_parameter=1):
    """
    Trains Softmax regression on digit (mod 3) classifications.
```

```
    See run_softmax_on_MNIST for more info.
    """
    #YOUR CODE HERE
    train_x, train_y, test_x, test_y = get_MNIST_data()

    # change label using mod3
    train_x, test_x = update_y(train_x, test_x)
    train_y, test_y = update_y(train_y, test_y)

    theta, cost_function_history = softmax_regression(train_x, train_y,␣
 ↪temp_parameter, alpha= 0.3, lambda_factor = 1.0e-4, k = 10, num_iterations =␣
 ↪150)

    test_error = compute_test_error(test_x, test_y, theta, temp_parameter)

    # TODO: add your code here for the "Using the Current Model" question in␣
 ↪tab 4.
    #       and print the test_error_mod3
    return test_error

# TODO: Run run_softmax_on_MNIST_mod3(), report the error rate
#print('softmax test_error=', run_softmax_on_MNIST_mod3())


#############################################################################
# 7. Classification Using Manually Crafted Features
#############################################################################

## Dimensionality reduction via PCA ##

# TODO: First fill out the PCA functions in features.py as the below code␣
 ↪depends on them.

# n_components = 18
# pcs = principal_components(train_x)
# train_pca = project_onto_PC(train_x, pcs, n_components)
# test_pca = project_onto_PC(test_x, pcs, n_components)
# # train_pca (and test_pca) is a representation of our training (and test)␣
 ↪data
# # after projecting each example onto the first 18 principal components.

# # TODO: Train your softmax regression model using (train_pca, train_y)
# #       and evaluate its accuracy on (test_pca, test_y).

def run_softmax_on_MNIST_PCA18(temp_parameter=1):
```

```
    train_x, train_y, test_x, test_y = get_MNIST_data()

    n_components = 18
    pcs = principal_components(train_x)
    train_pca = project_onto_PC(train_x, pcs, n_components)
    test_pca = project_onto_PC(test_x, pcs, n_components)

    theta, cost_function_history = softmax_regression(train_pca, train_y,␣
 ↪temp_parameter, alpha= 0.3, lambda_factor = 1.0e-4, k = 10, num_iterations =␣
 ↪150)
    test_error = compute_test_error(test_pca, test_y, theta, temp_parameter)

    return test_error

#print('run_softmax_on_MNIST_PCA18 test_error=', run_softmax_on_MNIST_PCA18())


train_x, train_y, test_x, test_y = get_MNIST_data()

n_components = 18
pcs = principal_components(train_x)
train_pca = project_onto_PC(train_x, pcs, n_components)
test_pca = project_onto_PC(test_x, pcs, n_components)

# TODO: Use the plot_PC function in features.py to produce scatterplot
#       of the first 100 MNIST images, as represented in the space spanned by␣
 ↪the
#       first 2 principal components found above.
plot_PC(train_x[range(100),], pcs, train_y[range(100)])


# TODO: Use the reconstruct_PC function in features.py to show
#       the first and second MNIST images as reconstructed solely from
#       their 18-dimensional principal component representation.
#       Compare the reconstructed images with the originals.
firstimage_reconstructed = reconstruct_PC(train_pca[0, ], pcs, n_components,␣
 ↪train_x)
plot_images(firstimage_reconstructed)
plot_images(train_x[0,])

secondimage_reconstructed = reconstruct_PC(train_pca[1, ], pcs, n_components,␣
 ↪train_x)
plot_images(secondimage_reconstructed)
plot_images(train_x[1,])
```

```
## Cubic Kernel ##
# TODO: Find the 10-dimensional PCA representation of the training and test set

# TODO: First fill out cubicFeatures() function in features.py as the below␣
 ↪code requires it.

# train_cube = cubic_features(train_pca10)
# test_cube = cubic_features(test_pca10)
# train_cube (and test_cube) is a representation of our training (and test)␣
 ↪data
# after applying the cubic kernel feature mapping to the 10-dimensional PCA␣
 ↪representations.


# TODO: Train your softmax regression model using (train_cube, train_y)
#       and evaluate its accuracy on (test_cube, test_y).

def run_softmax_on_MNIST_PCA_cubic(temp_parameter=1):

    train_x, train_y, test_x, test_y = get_MNIST_data()

    n_components = 10
    pcs = principal_components(train_x)
    train_pca10 = project_onto_PC(train_x, pcs, n_components)
    test_pca10 = project_onto_PC(test_x, pcs, n_components)

    train_cube = cubic_features(train_pca10)
    test_cube = cubic_features(test_pca10)

    theta, cost_function_history = softmax_regression(train_cube, train_y,␣
 ↪temp_parameter, alpha= 0.3, lambda_factor = 1.0e-4, k = 10, num_iterations =␣
 ↪150)
    test_error = compute_test_error(test_cube, test_y, theta, temp_parameter)

    return test_error

print('run_softmax_on_MNIST_PCA_cubic test_error=',␣
 ↪run_softmax_on_MNIST_PCA_cubic())
```

1.11.7 test.py

[ ]:
```
import os
import sys
import time
import traceback
import numpy as np
```

```
import linear_regression
import svm
import softmax
import features
import kernel

sys.path.append("..")
import utils

verbose = False

epsilon = 1e-6

def green(s):
    return '\033[1;32m%s\033[m' % s

def yellow(s):
    return '\033[1;33m%s\033[m' % s

def red(s):
    return '\033[1;31m%s\033[m' % s

def log(*m):
    print(" ".join(map(str, m)))

def log_exit(*m):
    log(red("ERROR:"), *m)
    exit(1)


def check_real(ex_name, f, exp_res, *args):
    try:
        res = f(*args)
    except NotImplementedError:
        log(red("FAIL"), ex_name, ": not implemented")
        return True
    if not np.isreal(res):
        log(red("FAIL"), ex_name, ": does not return a real number, type: ",␣
 ↪type(res))
        return True
    if not -epsilon < res - exp_res < epsilon:
        log(red("FAIL"), ex_name, ": incorrect answer. Expected", exp_res, ",␣
 ↪got: ", res)
        return True


def equals(x, y):
```

```
    if type(y) == np.ndarray:
        return (np.abs(x - y) < epsilon).all()
    return -epsilon < x - y < epsilon

def check_tuple(ex_name, f, exp_res, *args, **kwargs):
    try:
        res = f(*args, **kwargs)
    except NotImplementedError:
        log(red("FAIL"), ex_name, ": not implemented")
        return True
    if not type(res) == tuple:
        log(red("FAIL"), ex_name, ": does not return a tuple, type: ",␣
↪type(res))
        return True
    if not len(res) == len(exp_res):
        log(red("FAIL"), ex_name, ": expected a tuple of size ", len(exp_res),␣
↪" but got tuple of size", len(res))
        return True
    if not all(equals(x, y) for x, y in zip(res, exp_res)):
        log(red("FAIL"), ex_name, ": incorrect answer. Expected", exp_res, ",␣
↪got: ", res)
        return True

def check_array(ex_name, f, exp_res, *args):
    try:
        res = f(*args)
    except NotImplementedError:
        log(red("FAIL"), ex_name, ": not implemented")
        return True
    if not type(res) == np.ndarray:
        log(red("FAIL"), ex_name, ": does not return a numpy array, type: ",␣
↪type(res))
        return True
    if not len(res) == len(exp_res):
        log(red("FAIL"), ex_name, ": expected an array of shape ", exp_res.
↪shape, " but got array of shape", res.shape)
        return True
    if not equals(res, exp_res):
        log(red("FAIL"), ex_name, ": incorrect answer. Expected", exp_res, ",␣
↪got: ", res)

        return True

def check_list(ex_name, f, exp_res, *args):
    try:
        res = f(*args)
```

```
    except NotImplementedError:
        log(red("FAIL"), ex_name, ": not implemented")
        return True
    if not type(res) == list:
        log(red("FAIL"), ex_name, ": does not return a list, type: ", type(res))
        return True
    if not len(res) == len(exp_res):
        log(red("FAIL"), ex_name, ": expected a list of size ", len(exp_res), "␣
 ↪but got list of size", len(res))
        return True
    if not all(equals(x, y) for x, y in zip(res, exp_res)):
        log(red("FAIL"), ex_name, ": incorrect answer. Expected", exp_res, ",␣
 ↪got: ", res)
        return True

def check_get_mnist():
    ex_name = "Get MNIST data"
    train_x, train_y, test_x, test_y = utils.get_MNIST_data()
    log(green("PASS"), ex_name, "")


def check_closed_form():
    ex_name = "Closed form"
    X = np.arange(1, 16).reshape(3, 5)
    Y = np.arange(1, 4)
    lambda_factor = 0.5
    exp_res = np.array([-0.03411225,  0.00320187,  0.04051599,  0.07783012,  0.
 ↪11514424])
    if check_array(
            ex_name, linear_regression.closed_form,
            exp_res, X, Y, lambda_factor):
        return

    log(green("PASS"), ex_name, "")

def check_svm():
    ex_name = "One vs rest SVM"
    n, m, d = 5, 3, 7
    train_x = np.random.random((n, d))
    test_x = train_x[:m]
    train_y = np.zeros(n)
    train_y[-1] = 1
    exp_res = np.zeros(m)

    if check_array(
            ex_name, svm.one_vs_rest_svm,
            exp_res, train_x, train_y, test_x):
```

```
        return

    train_y = np.ones(n)
    train_y[-1] = 0
    exp_res = np.ones(m)

    if check_array(
            ex_name, svm.one_vs_rest_svm,
            exp_res, train_x, train_y, test_x):
        return

    log(green("PASS"), ex_name, "")


def check_compute_probabilities():
    ex_name = "Compute probabilities"
    n, d, k = 3, 5, 7
    X = np.arange(0, n * d).reshape(n, d)
    zeros = np.zeros((k, d))
    temp = 0.2
    exp_res = np.ones((k, n)) / k
    if check_array(
            ex_name, softmax.compute_probabilities,
            exp_res, X, zeros, temp):
        return

    theta = np.arange(0, k * d).reshape(k, d)
    softmax.compute_probabilities(X, theta, temp)
    exp_res = np.zeros((k, n))
    exp_res[-1] = 1
    if check_array(
            ex_name, softmax.compute_probabilities,
            exp_res, X, theta, temp):
        return

    log(green("PASS"), ex_name, "")

def check_compute_cost_function():
    ex_name = "Compute cost function"
    n, d, k = 3, 5, 7
    X = np.arange(0, n * d).reshape(n, d)
    Y = np.arange(0, n).reshape(n, 1)
    zeros = np.zeros((k, d))
    temp = 0.2
    lambda_factor = 0.5
    # exp_res = 5.83773044716594  this is not correct
    exp_res = 1.9459101490553135
```

```
    if check_real(
            ex_name, softmax.compute_cost_function,
            exp_res, X, Y, zeros, lambda_factor, temp):
        return
    log(green("PASS"), ex_name, "")

def check_run_gradient_descent_iteration():
    ex_name = "Run gradient descent iteration"
    n, d, k = 3, 5, 7
    X = np.arange(0, n * d).reshape(n, d)
    Y = np.arange(0, n)
    zeros = np.zeros((k, d))
    alpha = 2
    temp = 0.2
    lambda_factor = 0.5
    exp_res = np.zeros((k, d))
    exp_res = np.array([
       [ -7.14285714,  -5.23809524,  -3.33333333,  -1.42857143, 0.47619048],
       [  9.52380952,  11.42857143,  13.33333333,  15.23809524, 17.14285714],
       [ 26.19047619,  28.0952381 ,  30.        ,  31.9047619 , 33.80952381],
       [ -7.14285714,  -8.57142857, -10.        , -11.42857143, -12.85714286],
       [ -7.14285714,  -8.57142857, -10.        , -11.42857143, -12.85714286],
       [ -7.14285714,  -8.57142857, -10.        , -11.42857143, -12.85714286],
       [ -7.14285714,  -8.57142857, -10.        , -11.42857143, -12.85714286]
    ])

    if check_array(
            ex_name, softmax.run_gradient_descent_iteration,
            exp_res, X, Y, zeros, alpha, lambda_factor, temp):
        return
    softmax.run_gradient_descent_iteration(X, Y, zeros, alpha, lambda_factor,␣
 ↪temp)
    log(green("PASS"), ex_name, "")

def check_update_y():
    ex_name = "Update y"
    train_y = np.arange(0, 10)
    test_y = np.arange(9, -1, -1)
    exp_res = (
            np.array([0, 1, 2, 0, 1, 2, 0, 1, 2, 0]),
            np.array([0, 2, 1, 0, 2, 1, 0, 2, 1, 0])
            )
    if check_tuple(
            ex_name, softmax.update_y,
            exp_res, train_y, test_y):
        return
    log(green("PASS"), ex_name, "")
```

```
def check_project_onto_PC():
    ex_name = "Project onto PC"
    X = np.array([
        [0, 1, 2, 3 ],
        [1, 2, 3, 4 ],
        [2, 3, 4, 5 ],
    ]);
    pcs = features.principal_components(X)
    exp_res = np.array([
        [2, 0, 0],
        [0, 0, 0],
        [-2, 0, 0]
    ])
    n_components = 3
    if check_array(
            ex_name, features.project_onto_PC,
            exp_res, X, pcs, n_components):
        return
    log(green("PASS"), ex_name, "")


def check_polynomial_kernel():
    ex_name = "Polynomial kernel"
    n, m, d = 3, 5, 7
    c = 1
    p = 2
    X = np.random.random((n, d))
    Y = np.random.random((m, d))
    try:
        K = kernel.polynomial_kernel(X, Y, c, d)
    except NotImplementedError:
        log(red("FAIL"), ex_name, ": not implemented")
        return True
    for i in range(n):
        for j in range(m):
            exp = (X[i] @ Y[j] + c) ** d
            got = K[i][j]
            if (not equals(exp, got)):
                log(
                    red("FAIL"), ex_name,
                    ": values at ({}, {}) do not match. Expected {}, got {}"
                    .format(i, j, exp, got)
                )
    log(green("PASS"), ex_name, "")
```

```
def check_rbf_kernel():
    ex_name = "RBF kernel"
    n, m, d = 3, 5, 7
    gamma = 0.5
    X = np.random.random((n, d))
    Y = np.random.random((m, d))
    try:
        K = kernel.rbf_kernel(X, Y, gamma)
    except NotImplementedError:
        log(red("FAIL"), ex_name, ": not implemented")
        return True
    for i in range(n):
        for j in range(m):
            exp = np.exp(-gamma * (np.linalg.norm(X[i] - Y[j]) ** 2))
            got = K[i][j]
            if (not equals(exp, got)):
                log(
                    red("FAIL"), ex_name,
                    ": values at ({}, {}) do not match. Expected {}, got {}"
                    .format(i, j, exp, got)
                )
    log(green("PASS"), ex_name, "")


def main():
    log(green("PASS"), "Import mnist project")
    try:
        check_get_mnist()
        check_closed_form()
        check_svm()
        check_compute_probabilities()
        check_compute_cost_function()
        check_run_gradient_descent_iteration()
        check_update_y()
        check_project_onto_PC()
        check_polynomial_kernel()
        check_rbf_kernel()
    except Exception:
        log_exit(traceback.format_exc())

if __name__ == "__main__":
    main()
```