Project2

Welcome to the Project2! It consists of the **necessary part** and additional part.

Necessary part tasks are obligatory to implement in our course. You can get 100% for the Project2 if you complete all the tasks of the necessary part. Additional part tasks are not obligatory and you will receive no grade for them, but you're free to work with it since the material is really great.

Necessary part:

- Linear Regression with Closed Form Solution:
    - Implement closed_form(X, Y, lambda_factor) function
    - Answer "Test Error on Linear Regression" question
    - Answer "What went Wrong?" question
- Support Vector Machine:
    - Implement one_vs_rest_svm(train_x, train_y, test_x) function
    - Answer "Binary classification error" question
    - Answer "Implement C-SVM" question
    - Implement multi_class_svm(train_x, train_y, test_x) function
    - Answer "Multiclass SVM error"
- Multinomial (Softmax) Regression and Gradient Descent:
    - Implement compute_probabilities(X, theta, temp_parameter) function
    - Implement compute_cost_function(X, Y, theta, lambda_factor, temp_parameter) function
    - Implement run_gradient_descent_iteration(X, Y, theta, alpha, lambda_factor, temp_parameter) function
    - Answer "Test Error on Softmax Regression" question

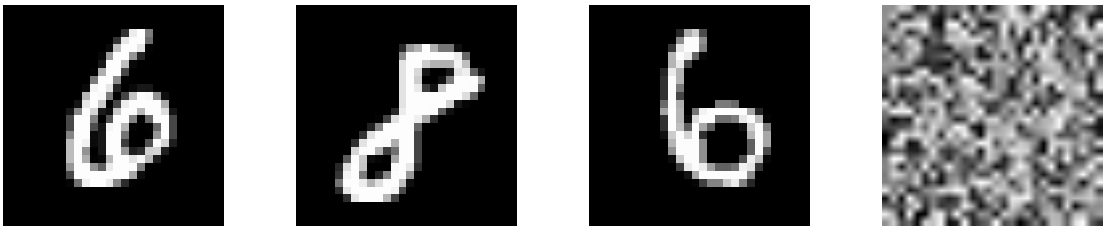
All the other tasks in this project are considered as an additional part and are not required to get 100% for Project2.

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 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 × 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.



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.

1. *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.

2. 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) 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:

1. `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).
2. `train_y` : The labels for each training datapoint, aka the digit shown in the corresponding image (a number between 0-9).
3. `test_x` : A matrix of the test data, formatted like `train_x`.
4. `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: You may find the tutorial on Scikit-learn in Introduction to ML Packages (Part 1) (in the resource section) helpful for this project.

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.

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 λ.
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
    """
```

 # TASK: - FIND THIS FUCTION IN linear_regression.py AND COMPLETE THE CODE

CHECK IT BY RUNNING test.py

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.

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

Error|λ=1=

Error|λ=0.1=

Error|λ=0.01=

What went Wrong?

Alice and you find that no matter what λ 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.

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. 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`

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.

```
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
    """
```

 # TASK: - FIND THIS FUCTION IN svm.py AND COMPLETE THE CODE

CHECK IT BY RUNNING test.py

Binary classification error

Report the test error by running `run_svm_one_vs_rest_on_MNIST`.

Error=

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.

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 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
    """
```

 # TASK: - FIND THIS FUCTION IN svm.py AND COMPLETE THE CODE

CHECK IT BY RUNNING test.py

Multiclass SVM error

Report the overall test error by running `run_multiclass_svm_on_MNIST`.

Error=

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

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 $j=0,1,\ldots,k-1$.

The softmax function h for a particular vector x requires computing

$$h(x) = \frac{1}{\sum_{j=0}^{k-1} e^{\theta_j \cdot x/\tau}} \begin{bmatrix} e^{\theta_0 \cdot x/\tau} \\ e^{\theta_1 \cdot x/\tau} \\ \vdots \\ e^{\theta_{k-1} \cdot x/\tau} \end{bmatrix},$$

where $\tau > 0$ is the **temperature parameter** . When computing the output probabilities (they should always be in the range $[0,1]$), the terms $e^{\theta_j \cdot x/\tau}$ 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

$$\begin{aligned} h(x) &= \frac{e^{-c}}{e^{-c} \sum_{j=0}^{k-1} e^{\theta_j \cdot x/\tau}} \begin{bmatrix} e^{\theta_0 \cdot x/\tau} \\ e^{\theta_1 \cdot x/\tau} \\ \vdots \\ e^{\theta_{k-1} \cdot x/\tau} \end{bmatrix} \\ &= \frac{1}{\sum_{j=0}^{k-1} e^{[\theta_j \cdot x/\tau]-c}} \begin{bmatrix} e^{[\theta_0 \cdot x/\tau]-c} \\ e^{[\theta_1 \cdot x/\tau]-c} \\ \vdots \\ e^{[\theta_{k-1} \cdot x/\tau]-c} \end{bmatrix}, \end{aligned}$$

subtracting some fixed amount c from each exponent will not change the final probabilities. A suitable choice for this fixed amount is

$$c = \max_j \theta_j \cdot x/\tau.$$

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

```
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
    """
```

 # TASK: - FIND THIS FUCTION IN softmax.py AND COMPLETE THE CODE

CHECK IT BY RUNNING test.py

Cost Function

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

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

$$J(\theta) = -\frac{1}{n}\left[\sum_{i=1}^{n}\sum_{j=0}^{k-1}[[y^{(i)} == j]]\log\frac{e^{\theta_j \cdot x^{(i)}/\tau}}{\sum_{l=0}^{k-1} e^{\theta_l \cdot x^{(i)}/\tau}}\right] + \frac{\lambda}{2}\sum_{j=0}^{k-1}\sum_{i=0}^{d-1}\theta_{ji}^2$$

Available Functions: You have access to the NumPy python library as `np` and the previous function as `compute_probabilities`

```
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)
    """
```

 # TASK: - FIND THIS FUCTION IN softmax.py AND COMPLETE THE CODE

CHECK IT BY RUNNING test.py

Gradient Descent

Now, in order to run the gradient descent algorithm to minimize the cost function, we need to take the derivative of $J(\theta)$ wrt a particular θ_m.

Notice that within $J(\theta)$, we have:

$$\frac{e^{\theta_j \cdot x^{(i)}/\tau}}{\sum_{l=0}^{k-1} e^{\theta_l \cdot x^{(i)}/\tau}} = p\left(y^{(i)} = j | x^{(i)}, \theta\right)$$

so we first compute: $\frac{\partial p(y^{(i)}=j|x^{(i)},\theta)}{\partial \theta_m}$,
when $m = j$,

$$\frac{\partial p\left(y^{(i)} = j | x^{(i)}, \theta\right)}{\partial \theta_m} = \frac{x^{(i)}}{\tau} p\left(y^{(i)} = m | x^{(i)}, \theta\right)\left[1 - p\left(y^{(i)} = m | x^{(i)}, \theta\right)\right]$$

when $m \neq j$,

$$\frac{\partial p\left(y^{(i)} = j | x^{(i)}, \theta\right)}{\partial \theta_m} = -\frac{x^{(i)}}{\tau} p\left(y^{(i)} = m | x^{(i)}, \theta\right) p\left(y^{(i)} = j | x^{(i)}, \theta\right)$$

Now we compute

$$\begin{aligned}
\frac{\partial}{\partial \theta_m}\left[\sum_{j=0}^{k-1}[[y^{(i)} == j]] \log \frac{e^{\theta_j \cdot x^{(i)}/\tau}}{\sum_{l=0}^{k-1} e^{\theta_l \cdot x^{(i)}/\tau}}\right] &= \sum_{j=0, j\neq m}^{k-1}\left[[[y^{(i)} == j]]\left[-\frac{x^{(i)}}{\tau} p\left(y^{(i)} = m | x^{(i)}, \theta\right)\right]\right] \\
&+ [[y^{(i)} == m]] \frac{x^{(i)}}{\tau}\left[1 - p\left(y^{(i)} = m | x^{(i)}, \theta\right)\right] \\
&= \frac{x^{(i)}}{\tau}\left[[[y^{(i)} == m]] - p\left(y^{(i)} = m | x^{(i)}, \theta\right) \sum_{j=0}^{k-1}[[y^{(i)} == j]]\right] \\
&= \frac{x^{(i)}}{\tau}\left[[[y^{(i)} == m]] - p\left(y^{(i)} = m | x^{(i)}, \theta\right)\right]
\end{aligned}$$

Plug this into the derivatite of $J(\theta)$, we have

$$\begin{aligned}
\frac{\partial J(\theta)}{\partial \theta_m} &= \frac{\partial}{\partial \theta_m}\left[-\frac{1}{n}\left[\sum_{i=1}^{n}\sum_{j=0}^{k-1}[[y^{(i)} == j]] \log p\left(y^{(i)} = j | x^{(i)}, \theta\right)\right] + \frac{\lambda}{2}\sum_{j=0}^{k-1}\sum_{i=0}^{d-1}\theta_{ji}^2\right] \\
&= -\frac{1}{\tau n}\sum_{i=1}^{n}\left[x^{(i)}\left([[y^{(i)} == m]] - p\left(y^{(i)} = m | x^{(i)}, \theta\right)\right)\right] + \lambda\theta_m
\end{aligned}$$

Plug this into the derivatite of $J(\theta)$, we have

$$\begin{aligned}\frac{\partial J(\theta)}{\partial \theta_m} &= \frac{\partial}{\partial \theta_m}\left[-\frac{1}{n}\left[\sum_{i=1}^{n}\sum_{j=0}^{k-1}[[y^{(i)}==j]]\log p(y^{(i)}=j|x^{(i)},\theta)\right]+\frac{\lambda}{2}\sum_{j=0}^{k-1}\sum_{i=0}^{d-1}\theta_{ji}^2\right] \\ &= -\frac{1}{\tau n}\sum_{i=1}^{n}\left[x^{(i)}\left([[y^{(i)}==m]]-p(y^{(i)}=m|x^{(i)},\theta)\right)\right]+\lambda\theta_m\end{aligned}$$

To run gradient descent, we will update θ at each step with $\theta \leftarrow \theta - \alpha \nabla_\theta J(\theta)$, where α is the learning rate.

Write a function `run_gradient_descent_iteration` that runs one step of the gradient descent algorithm.

Available Functions: You have access to the NumPy python library as `np`, `compute_probabilities` which you previously implemented and `scipy.sparse` as `sparse`

You should use `sparse.coo_martix` so that your function can handle larger matrices efficiently. The sparse matrix representation can handle sparse matrices efficiently.

Hint:

This is how to use `scipy`'s `sparse.coo_matrix` function to create a sparse matrix of 0's and 1's:

```
M = sparse.coo_matrix(([1]*n, (Y, range(n))), shape=(k,n)).toarray()
```

This will create a normal numpy array with 1s and 0s.

On larger inputs (i.e. MNIST), this is 10x faster than using a naive for loop. (See example code if interested).

Note: As a personal challenge, try to see if you can use special numpy functions to add 1 in-place. This would be even faster.

```
import time
import numpy as np
import scipy.sparse as sparse

ITER = 100
K = 10
N = 10000

def naive(indices, k):
    mat = [[1 if i == j else 0 for j in range(k)] for i in indices]
    return np.array(mat).T
```

```
def with_sparse(indices, k):
    n = len(indices)
    M = sparse.coo_matrix(([1]*n, (Y, range(n))),
shape=(k,n)).toarray()
    return M


Y = np.random.randint(0, K, size=N)

t0 = time.time()
for i in range(ITER):
    naive(Y, K)
print(time.time() - t0)


t0 = time.time()
for i in range(ITER):
    with_sparse(Y, K)
print(time.time() - t0)
```

---

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

"""

 # TASK: - FIND THIS FUCTION IN softmax.py AND COMPLETE THE CODE

CHECK IT BY RUNNING test.py

Test Error on Softmax Regression

Finally, report the final test error by running the `main.py` file, using the temperature parameter $\tau=1$. 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 such as precision and recall, which we have not discussed in this class.

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

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

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 θ. What about a small θ?

- [x] Larger temperature leads to less variance
- [ ] Smaller temperature leads to less variance
- [ ] Smaller temperature makes the distribution more uniform

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

$Error|_{T=2}=$

0.126

6. Changing Labels

We now wish to classify the digits by their (mod 3) value, such that the new label $y^{(i)}$ of sample i is the old $y^{(i)} (\mathrm{mod} 3)$. (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

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):
    """
    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
    """
```

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:

	Estimated Y	Estimated Y (mod 3)	Correct Y	Correct Y (mod 3)
x_1	9	0	8	2
x_2	6	0	6	0
x_3	5	2	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 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)
    """
```

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

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?

- Increase
- Decrease
- 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:

0.187

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.

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 $n \times d$ dataset X onto $k \leq d$ 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 $\tilde{X}^T\tilde{X}$, where $\tilde{X}$ is a centered version of our original data.

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

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 $\phi(x)$, defined so that for any $x, x' \in R^d$:

$$\phi(x)^T \phi(x') = (x^T x' + 1)^3$$

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

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 $n \times d$ dataset X into its k-dimensional PCA representation, one can use matrix multiplication, after first centering X:
$\tilde{X}V$
where $\tilde{X}$ is the centered version of the original data X and V is the $d \times k$ matrix whose columns are the top k eigenvectors of $\tilde{X}^T\tilde{X}$. 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

```
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.
```

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.

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.

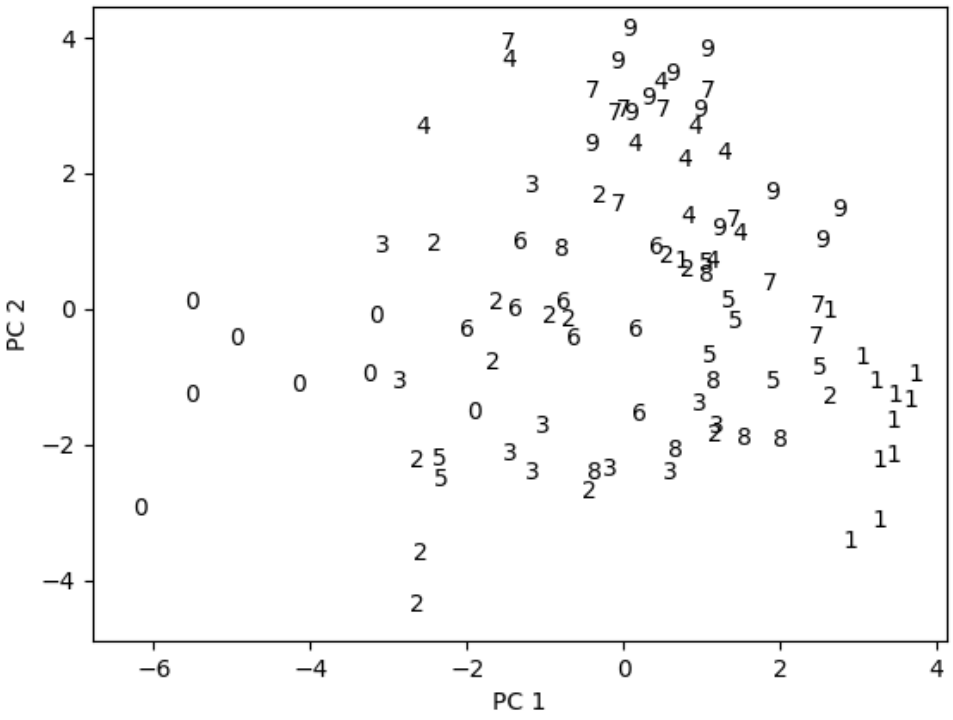
Error rate for 18-dimensional PCA features =

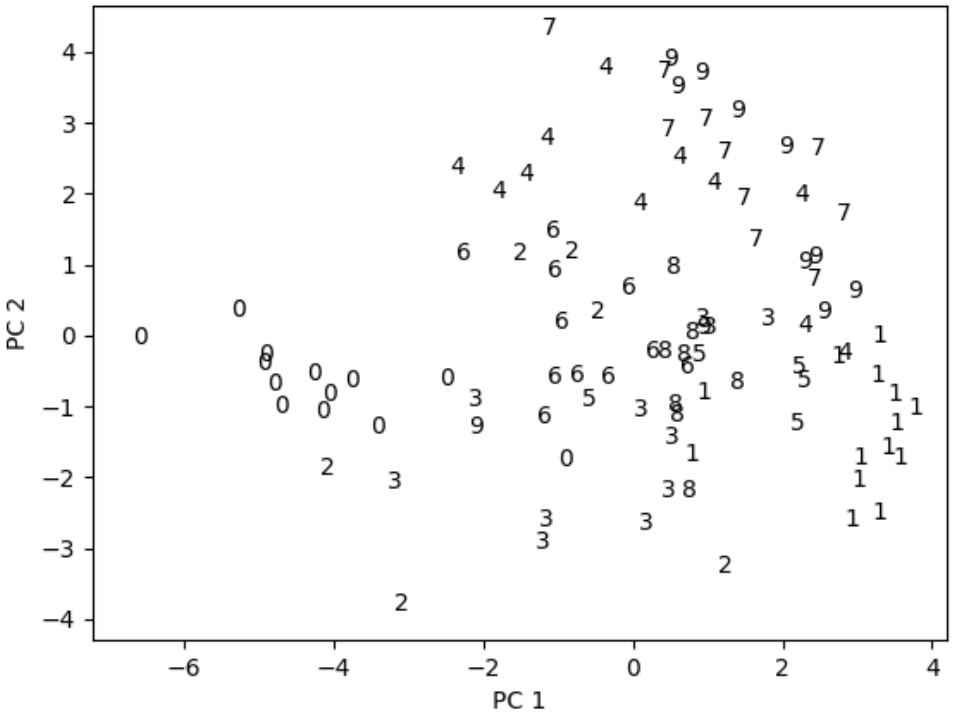
0.1491

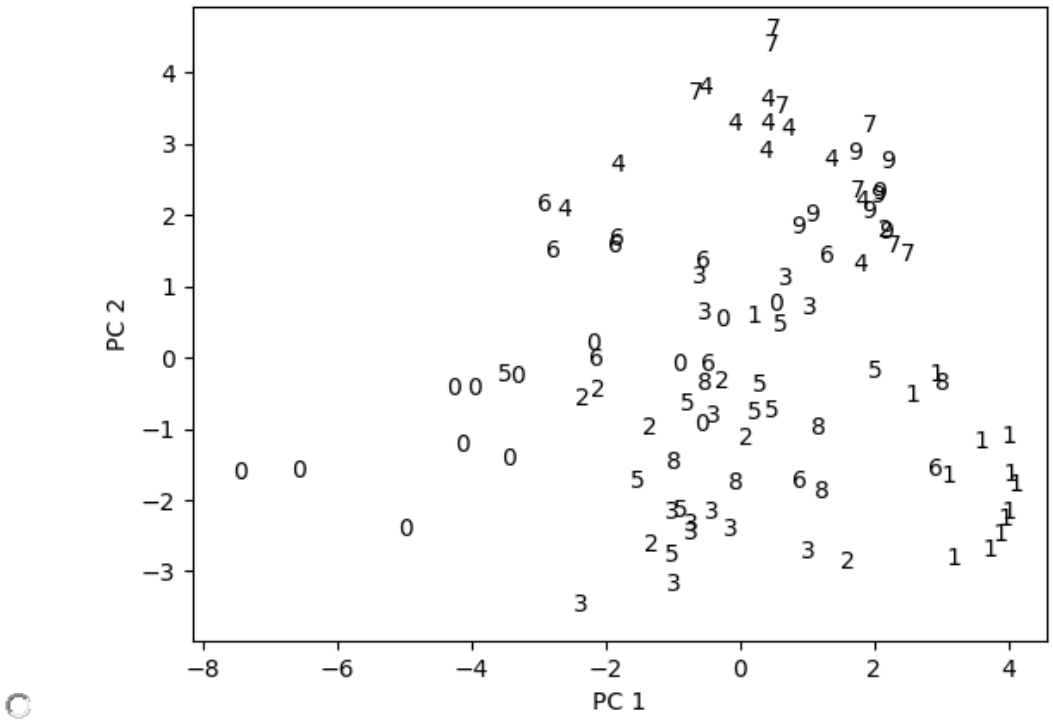
Testing PCA (continued)

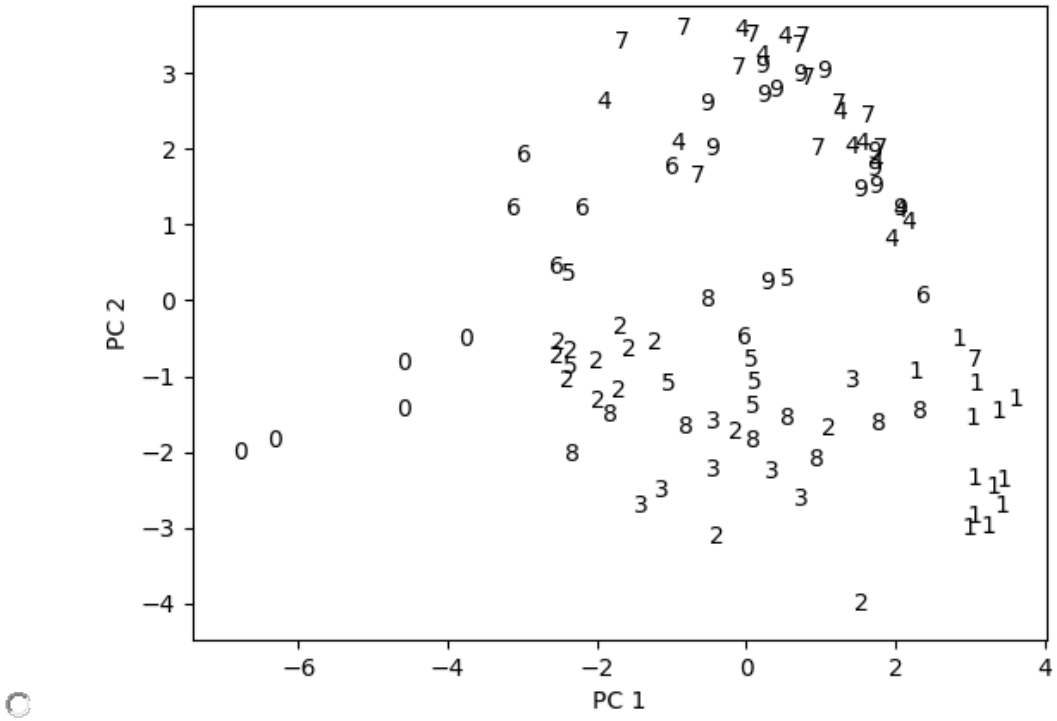
Use `plot_PC` in **main.py** to visualize the first 100 MNIST images, as represented in the space spanned by the first 2 principal components of the training data.

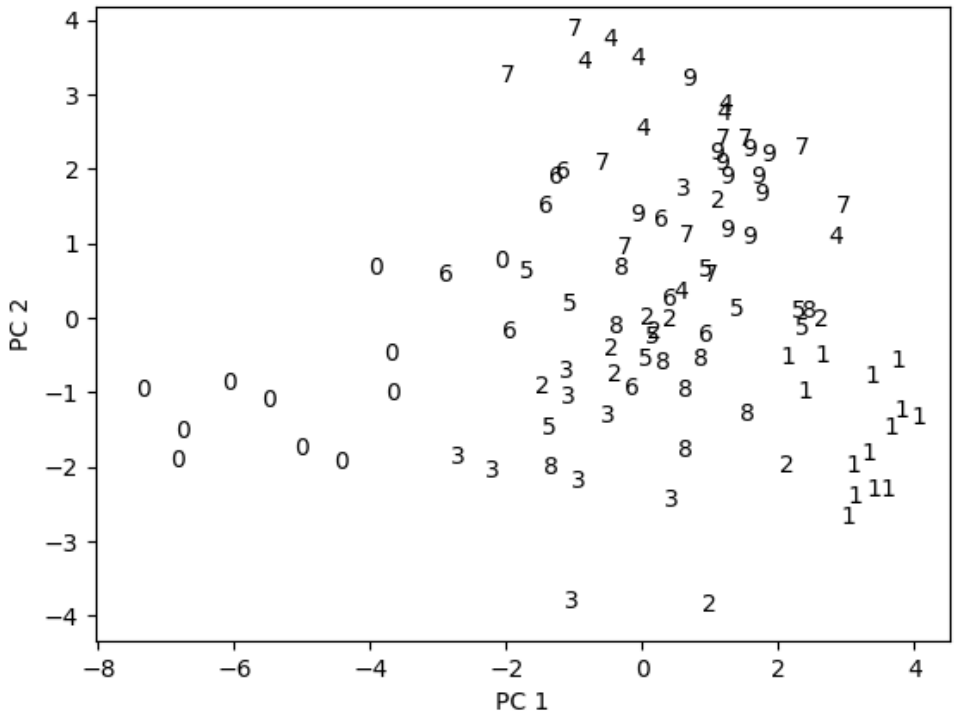
What does your PCA look like?


PC 2
PC 1


PC 2
PC 1


PC 2
PC 1


PC 2
PC 1



Please, choose one of four.

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.

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 $\phi(x)$, defined so that for any $x,x' \in \mathbb{R}^d$:

$$\phi(x)^T \phi(x') = (x^T x' + 1)^3$$

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

Computing Cubic Features

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

$$\phi(x) =$$

Hint: it must be 10-dimensional vector

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.

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 =

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 θ can be represented by a linear combination of features:

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

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

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

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

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

We actually do not need the real mapping $\phi(x)$, but the inner product between two features after mapping: $\phi(x_i) \cdot \phi(x)$, 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 $K(x,y) = \phi(x) \cdot \phi(y)$, 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

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,

$$\phi(x) = \langle x_d^2, \ldots, x_1^2, \sqrt{2}x_d x_{d-1}, \ldots, \sqrt{2}x_d x_1, \sqrt{2}x_{d-1}x_{d-2}, \ldots, \sqrt{2}x_{d-1}x_1, \ldots, \sqrt{2}x_2 x_1, \sqrt{2c}x_d, \ldots, \sqrt{2c}x_1, c \rangle$$

Write a function `polynomial_kernel` that takes in two matrix X and Y and computes the polynomial kernel $\mathrm{K(x,y)}$ 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 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
    """
```

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 $\mathrm{K(x,y)}$ 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
"""
```

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….

But this is the end of MechMath’s 2019 Introduction to Machine Learning Using Python course. Neural networks are coming little bit later ;-)