CS 422/622 Project 3

Due 11/16 11:59pm

Logistics: You must implement everything stated in this project description that is marked with an **implement** tag. Whenever you see the **write-up** tag, that is something that must be addressed in the project README.txt. Graduate students are required to implement everything including items tagged with **622**. Students in 422 do not need to complete these extra elements. **You cannot import any packages unless specifically noted by the instructor.** In this project, you may import the numpy and sys packages. You are welcome to ask about particular packages as they come up. The idea is to implement everything from scratch.

Deliverables: Each student should submit a single ZIP file, containing their project code (*.py files) and your writeup (README.txt). You should create a folder called Project3 (exactly like this with the capital P) and put all your code and your writeup in this folder. Then zip this folder up. That way when I extract your folder I can run my tests from outside the directory without having to change anything. Your zip file should be named lastname_firstname_project2.zip. Your code should run without errors in a linux environment. If your code does not run for a particular problem, you will lose 50% on that problem. You should submit only one py file for each problem. If your file does not match the filename provided exactly, you will lose 50% on that problem.

Grading: I have provided you with a test script (test_script.py) you can use to test out your functions. I will be using a similar test script, so if your code works with this script it should work with mine! The output of the test script should look something like this if you have implemented everything correctly. **Each student must work independently. You are to submit your own original work.**

1 PCA (50 points)

File name: **pca.py**

Implement: You will implement four functions listed here and detailed below.

```
def compute_Z(X, centering=True, scaling=False)
def compute_covariance_matrix(Z)
def find_pcs(COV)
def project_data(Z, PCS, L, k, var)
```

Write-Up: Describe your implementation concisely.

`def compute_Z(X, centering=True, scaling=False)`

The above function will take the data matrix `X`, and boolean variables `centering` and `scaling`. `X` has one sample per row. Remember there are no labels in PCA. If `centering` is True, you will subtract the mean from each feature. If `scaling` is True, you will divide each feature by its standard deviation. This function returns the `Z` matrix (numpy array), which is the same size as `X`.

`def compute_covariance_matrix(Z)`

The above function will take the standardized data matrix `Z` and return the covariance matrix $Z^T Z$=`COV` (a numpy array).

`def find_pcs(COV)`

The above function will take the covariance matrix `COV` and return the ordered (largest to smallest) principal components `PCS` (a numpy array where each column is an eigenvector) and corresponding eigenvalues `L` (a numpy array). You will want to use `np.linalg.eig` for this.

`def project_data(Z, PCS, L, k, var)`

The above function will take the standardized data matrix `Z`, the principal components `PCS`, and corresponding eigenvalues `L`, as well as a `k` integer value and a `var` floating point value. `k` is the number of principal components you wish to maintain when projecting the data into the new space. $0 \leq$ `k` $\leq D$. If `k`= 0, then we use the cumulative variance to determine the projection dimension. `var` is the desired cumulative variance explained by the projection. $0 \leq$`v`≤ 1. If `v`= 0, then `k` is used instead. Assume they are never both 0 or both > 0. This function will return `Z_star`, the projected data.

The result of the test script on the PCA tests should give the following results:

```
PCA Test 1:
 [[4. 0.]
 [0. 4.]]
 [4. 4.]
 [[0. 1.]
 [1. 0.]]
 [[-1.]
 [ 1.]
 [-1.]
 [ 1.]]

PCA Test 2:
[[5.5489993 5.539    ]
 [5.539     6.449    ]]
[11.556249    0.44175014]
[[-0.6778734  -0.73517865]
 [-0.73517865  0.6778734 ]]
[[-0.82797021]
 [ 1.77758027]
 [-0.99219756]
 [-0.27421041]
 [-1.67580129]
 [-0.91294906]
 [ 0.09910945]
 [ 1.14457214]
 [ 0.43804613]
 [ 1.22382056]]
```

2 SVM (50 points)

File name: **binary_classification.py**

You will implement a linear classifier with a decision boundary that maximizes the margin between positive and negative samples (SVM). The details of SVM can be quite complex, and so here you will implement a simpler version of it for 2D data.

We provide you with a function for generating data (features and labels):

```
data = helpers.generate_training_data_binary(num)
```

`num` indicates the training set. There are four different datasets `(num=1, num=2, num=3, num=4)` which you will need to train on and provide results for. All the datasets are linearly separable. `data` is a numpy array containing 2D features and their corresponding label in each row. So `data[0]` will have the form `[x1, x2, y]` where `[x1, x2]` is the feature for the first training sample and it has the label `y`. `y` can be -1 or 1.

Implement: You must write the following function:

```
[w,b,S] = svm_train_brute(training_data)
```

This function will return the decision boundary for your classifier and a numpy array of support vectors `S`. That is, it will return the line characterized by `w` and `b` that separates the training data with the largest margin between the positive and negative class.

Since the decision boundary is a line in the case of separable 2D data, you can find the best decision boundary (the one with the maximum margin between positive and negative samples) by looking at lines that separate the data and choosing the best one (the one with the maximum margin). The maximum margin separator depends only on the support vectors. So you can find the maximum margin separator via a brute force search over possible support vectors. In 2D there can only be either 2 or 3 support vectors. *THINK ABOUT THIS.*

In order to implement your training function, you will need to write some helper functions:

Implement: Write a function that compute the distance between a point and a hyperplane. In 2D, the hyperplane is a line, defined by `w` and `b`. Your code must work for hyperplanes in 2D. Do not worry about higher dimensions.

```
dist = distance_point_to_hyperplane(pt, w, b)
```

Implement: Write a function that takes a set of data, and a separator, and computes the margin.

```
margin = compute_margin(data, w, b)
```

Implement: You will need to write the following function to test new data given a decision boundary. The following function will output the class `y` for a given test point `x`.

```
y = svm_test_brute(w,b,x)
```

Use the above helper functions when writing your training function.

Write-Up: Describe your implementation concisely.

The result of the test script on the SVM Binary Classification tests should give the following results:

```
SVM Binary Classification Test 1:
[-1.  0.] 0.0 [[-1.  0.  1.]
 [ 1.  0. -1.]]

SVM Binary Classification Test 2:
[ 0. -1.] 0.0 [[ 0. -1.  1.]
 [ 0.  1. -1.]]

SVM Binary Classification Test 3:
[0.25 0.25] -0.24999999999999956 [[ 3.  2.  1.]
 [-1. -2. -1.]]

SVM Binary Classification Test 4:
[-0.5  0.5] 8.326672684688674e-17 [[-1.  1.  1.]
 [ 0. -2. -1.]
 [-3. -1.  1.]]
```

622 The remainder of the project description is for **622** students.

622 File name: multiclass_classification.py

Now, let's assume we're working with data that comes from more than two classes. You can still use an SVM to classify data into one of the Y classes by using multiple SVMs in a one-vs-all fashion. Instead of thinking about it as class 1 versus class 2 versus class 3... you think of it as class 1 versus not class 1, and class 2 versus not class 2. So you would have one binary classifier for each class to distinguish that class from the rest.

We will provide you with a function to generate training data:

```
[data, Y] = generate_training_data_multi(num)
```

Similarly to the binary data generation function, `num` indicates the particular set of data, and there are 2. Each dataset has a different number of classes. The classes are identified from 1 to `Y`, where `Y` is the number of classes.

622 Implement: Implement the following function:

```
[W, B] = svm_train_multiclass(training_data)
```

This function should use your previously implemented `svm_train_brute` to train one binary classifier for each class. It will return the `Y` decision boundaries, one for each class (one-vs-rest). `W` is an array of `w`s and `B` is an array of `b`s where $w_i x + b_i$ is the decision boundary for the i^{th} class.

622 Implement: You will also implement a test function:

```
y = svm_test_multiclass(W,B,x)
```

This function will take the C decision boundaries as input and a test point x, and will return the predicted class, c. There are two special cases you will need to account for. You may find that a test point is in the "rest" for all of your one-vs-rest classifiers, and so it belongs to no class. You can just return -1 (null) when this happens. You may also find that a test point belongs to two classes. In this case you may choose the class for which the test point is the farthest from the decision boundary.

622 Write-Up: Describe your implementation concisely.

The result of the test script on the SVM Multi-Class Classification test should give the following results:

```
SVM Multi-Class Classification Test:
[array([-2., -0.]), array([ 2., -0.]), array([-0., -2.]), array([-0.,
    2.])] [-1.0, -1.0, -1.0, -1.0]
```