COMP 6636 Mini-project 1

Dennis Brown, dgb0028@auburn.edu

March 3, 2021

1 Overview

This Mini-project looks at classification error using k-Nearest Neighbors (kNN) and Perceptron, plus attribute importance using Perceptron. The project was implemented in Python 3 and the code is included in section 6 of this report. The output log from running the code is in section 7.

The project considers two data sets:

- a4a, obtained from https://www.csie.ntu.edu.tw/ cjlin/libsvmtools/datasets/binary.html#a4a
- iris, obtained from https://www.csie.ntu.edu.tw/ cjlin/libsvmtools/datasets/multiclass.html#iris

The data was provided in the "libsvm" format. Although there is a Python wrapper for libsvm that imports this data, that seems too heavyweight for this project, so instead a simple custom data importer reads the data into numpy arrays where each row is a sample and each column is a feature.

The a4a data included an "a4a" data set used for training and an "a4a.t" data set used for testing. The "a4a.t" data set had one more feature than the "a4a" data set, so the "a4a" data set was augmented by an extra column of zeros (which is implied by the libsvm data file format).

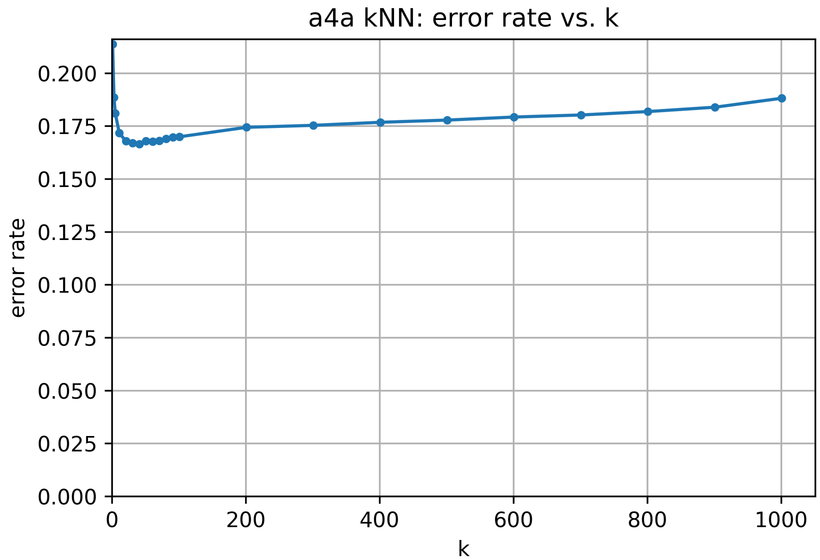
The iris data set didn't include separate train and test data sets. It consists of 150 samples that are assigned to one of three classes and is sorted by class. Therefore, the data was randomly shuffled and partitioned into 50 training and 100 test samples (the shuffling ensures the three classes are represented in both train and test data sets).

2 Classification Error for a4a

2.1 kNN

a4a data was classified using kNN as described in the overview. There are 4781 samples in the training set and 27780 samples in the test set. The test data was classified for 22 values of k between 1 and 1001. No data reduction was performed and this testing took a few hours. The classification error rate improves until $k = 41$ (error rate 0.167), then it rises consistently for the remaining values of k.

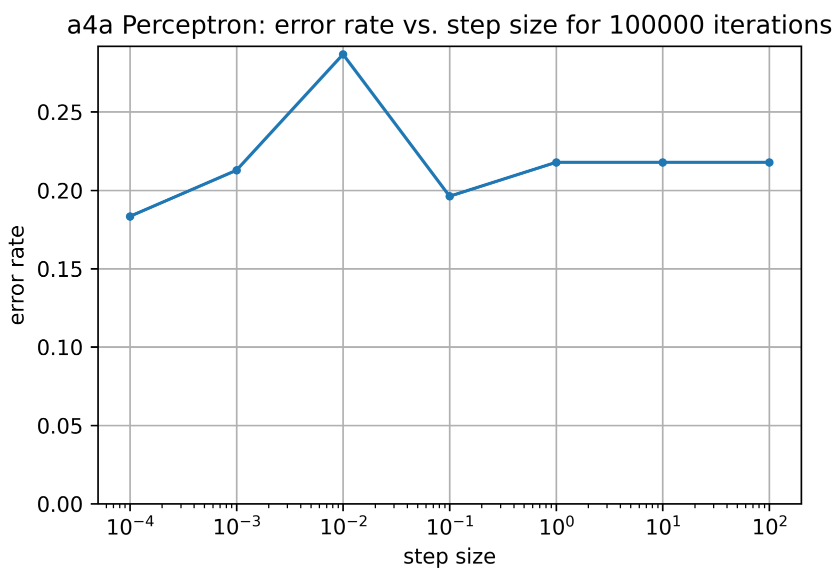
Figure 1: kNN classification error for a4a



2.2 Perceptron

a4a data was classified using Perceptron as described in the overview. There are 4781 samples in the training set and 27780 samples in the test set. The test data was classified for 7 values of β (step size) between .0001 and 100. The algorithm never converged with a limit of 100000 steps. No data reduction was performed. The lowest classification error rate is 0.183 with step size = 0.0001.

Figure 2: Perceptron classification error for a4a

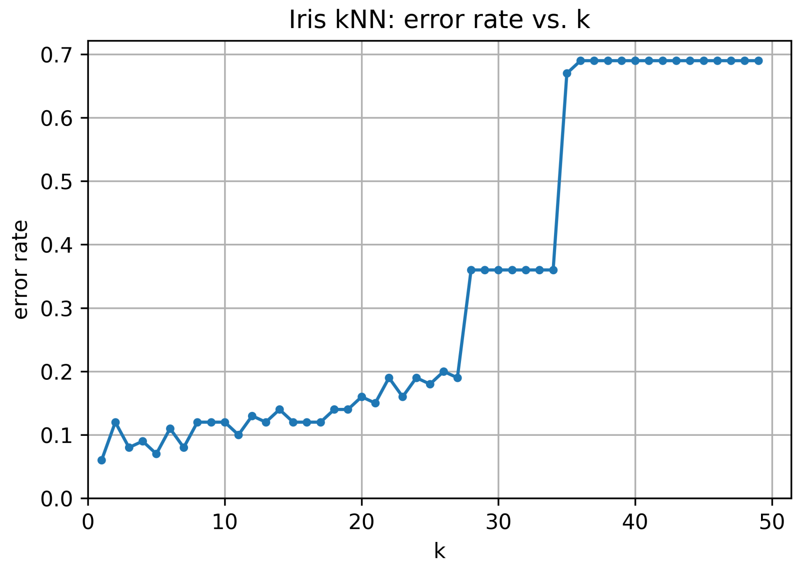


3 Classification Error for iris

3.1 kNN

iris data was classified using kNN as described in the overview. There are 50 samples in the training set and 100 samples in the test set. The minimum error is 0.06 at $k = 1$ and rises somewhat consistently from there. I don't have a good explanation for the stair-step effect starting at $k = 28$ and $k = 36$, other than it's an anomaly of how a small data set was randomly selected.

Figure 3: kNN classification error for iris



3.2 Perceptron

Since *iris* data has 3 classes, the One vs. All (OvA) approach was used to classify the data. Each of the three classes was trained and tested individually, where in each case the class being tested was considered "1" and the other classes were considered "-1". Changing the step size β or the step limit didn't make any difference, so only one configuration is presented here with $\beta = 0.1$ and step limit = 100000. Multi-class perceptron provided 3 weight vectors as visualized in section 5.

Testing the three weight vectors give the following results:

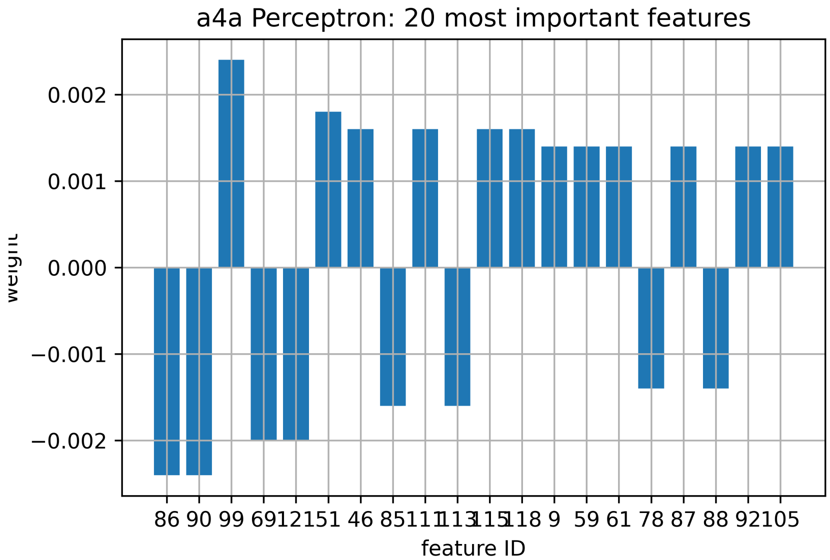
- Class 1: 0.00 error rate (it converged; class 1 is linearly separable from classes 2 and 3)
- Class 2: 0.68 error rate (did not converge)
- Class 3: 0.58 error rate (did not converge)
- Average: 0.42 error rate

I didn't graph the error rates because there are only three data points.

4 Importance of Attributes for a4a

The graph shows the top 20 feature ids after taking the weight vector associated with the lowest error rate and sorting the feature weights by magnitude. I could not find an explanation of what each feature id means. Feature 86 is the most important, whatever that is. I chose to visualize the actual values rather than absolute values so the reader can see what features are positively and negatively weighted.

Figure 4: a4a: Most important feature weights



5 Importance of Attributes for iris

Again, multi-class Perceptron was used to find a weight vector for each of the three classes of *iris* data. Since there are only four features, they are not shown here sorted for importance. It is more informative to look at the bar graph of each unsorted weight vector and see how the relative importance of each feature changes for the three classes:

- the signs of features 2 and 3 are reverse for classes 1 and 2;
- the sign of feature 4 is reverse between classes 2 and 3;
- the signs of features 2, 3, and 4 are reverse between classes 1 and 3; and
- feature 1 is relatively unimportant universally.

Figure 5: Feature weights for class = 1

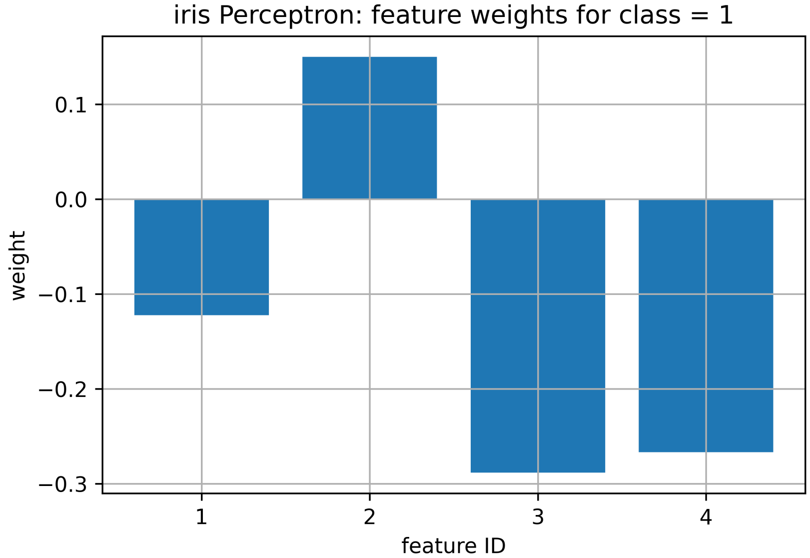


Figure 6: Feature weights for class = 2

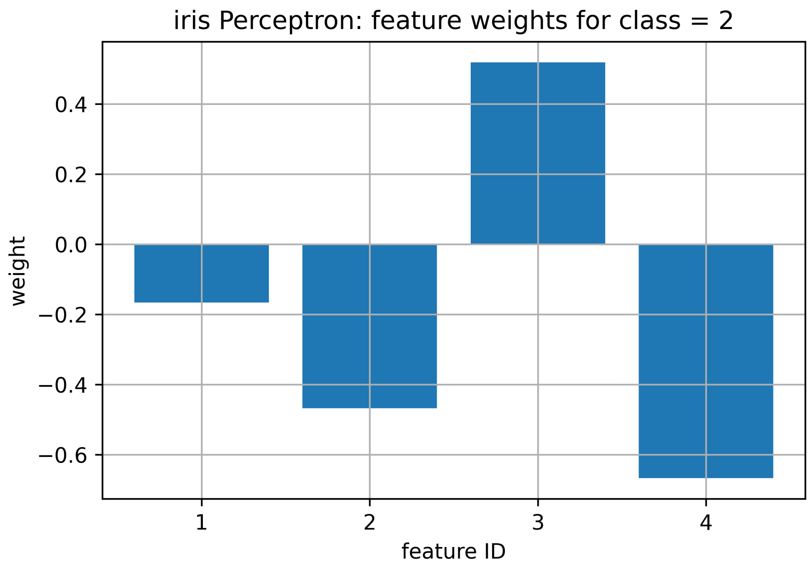
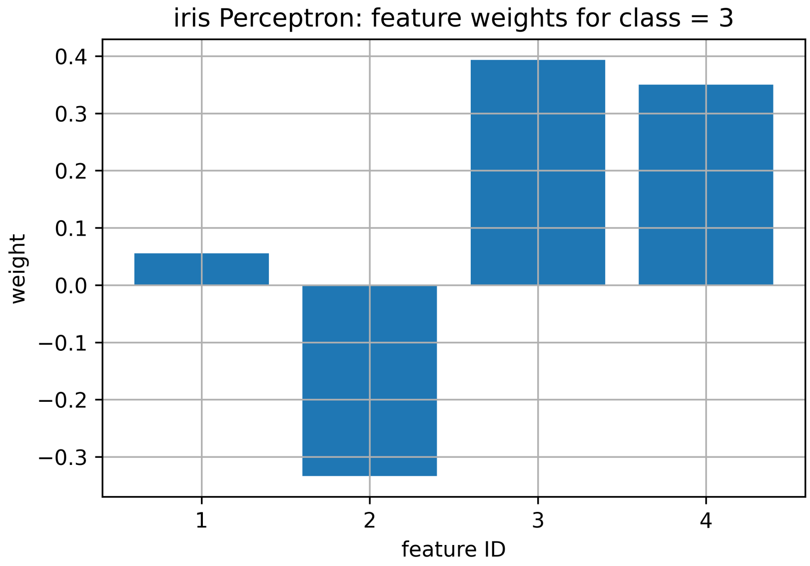


Figure 7: Feature weights for class = 3



6 Code

```
# -*- coding: utf-8 -*-
"""
Mini project 1

Dennis Brown, COMP6636, 03 MAR 2021
"""

import numpy as np
import copy
import matplotlib.pyplot as plt


def libsvm_scale_import(filename):
    """
    Read data from a libsvm .scale file
    """
    datafile = open(filename, 'r')

    # First pass: get dimensions of data
    num_samples = 0
    max_feature_id = 0
    for line in datafile:
        num_samples += 1
        tokens = line.split()
        for feature in tokens[1:]:
            feature_id = int(feature.split(':')[0])
            max_feature_id = max(feature_id, max_feature_id)

    # Second pass: read data into array
    data = np.zeros((num_samples, max_feature_id + 1))
    curr_sample = 0
    datafile.seek(0)
    for line in datafile:
        tokens = line.split()
        data[curr_sample][0] = float(tokens[0])
        for feature in tokens[1:]:
            feature_id = int(feature.split(':')[0])
```

```python
            feature_val = float(feature.split(':')[1])
            data[curr_sample][feature_id] = feature_val
        curr_sample += 1
    datafile.close()

    print('LOADED:', filename, ':', data.shape)

    return data


def get_neighbors(data, test_sample, num_neighbors):
    """
    Given training data, a test sample, and a number of
    neighbors, return the closest neighbors.
    """
    # Calculate all distances from the training samples
    # to this test sample. Collect index, distance into a list.
    indices_and_distances = list()
    for i in range(len(data)):
        dist = np.linalg.norm(test_sample[1:] - (data[i])[1:]) # leave out classification at pos 0
        indices_and_distances.append([i, dist])

    # Sort list by distance
    indices_and_distances.sort(key=lambda _: _[1])

    # Make a list of requested number of closest neighbors from sorted
    # list of indices+distances
    neighbors = list()
    for i in range(num_neighbors):
        neighbors.append(indices_and_distances[i][0])

    return neighbors


def classify_one_sample(data, test_sample, num_neighbors):
    """
    Given training data, a test sample, and a number of neighbors,
    predict which classification the test sample belongs to.
    """
    # Get closest neighbors
    neighbors = get_neighbors(data, test_sample, num_neighbors)

    # Create list of classifications of the neighbors
    classifications = list()
    for i in range(len(neighbors)):
        classifications.append(data[neighbors[i]][0]) # 0 = classification

    # Return the most common classification of the neighbors
    prediction = max(set(classifications), key = classifications.count)
    return prediction


def k_nearest_neighbors(data, test_samples, num_neighbors):
    """
    Given sample data (samples are rows, columns
    features, and samples have classifications in position 0),
    test data, and a number of neighbors, predict which classification
    each test sample belongs to.
    """
    classifications = list()
    for i in range(len(test_samples)):
        output = classify_one_sample(data, test_samples[i], num_neighbors)
        classifications.append(output)
        if ((i % 20) == 0):
            print('\rknn test sample', i, end='')
    print()

```

```python
    return(classifications)


def check_knn_classifications(y, y_hat):
    """
    Given actual values y and classiciations y_hat,
    return the number of errors
    """
    errors = 0
    for i in range(len(y)):
        if (y[i] != y_hat[i]):
            errors += 1

    return errors


def train_perceptron(data, beta, step_limit):
    """
    Perceptron. Given a set of data (samples are rows, columns
    features, and samples have classifications in position 0),
    a step size (beta), and a step limit, train and return a
    weight vector that can be used to classify the given data.
    """

    # Initialize the weight vector including bias element
    w = np.zeros(len(data[0]))

    # Initialize y_hat
    y_hat = np.zeros(len(data))

    # Slice off y
    y = data[:,0]

    # Repeat the main loop until we have convergence or reach the
    # iteration limit
    steps = 0
    converged = False
    while(not(converged) and (steps < step_limit)):
        converged = True

        # For each sample in the data, calculate w's classification error
        # and update w.
        for i in range(len(data)):
            # Replace classification in sample[0] with a 1 to allow
            # for a biased weight vector
            biased_sample = np.copy(data[i])
            biased_sample[0] = 1

            # Get prediction and error, then update weight vector
            y_hat[i] = 1 if (np.matmul(w.T, biased_sample) > 0) else -1
            error = y[i] - y_hat[i]
            w += biased_sample * error * beta
            steps += 1

            # If error on this element is > a very small value, we have
            # not converged.
            if (abs(error) > 0.000001):
                converged = False

    print('Perceptron:' ,steps, 'steps; converged?', converged)

    return w


def multiclass_train_perceptron(data, beta, step_limit):
    """
    Perceptron. Given a set of data (samples are rows, columns
    features, and samples have classifications in position 0),
```

```
    a step size (beta), and a step limit, train and return a
    weight vector that can be used to classify the given data.

    This version works on data with multiple classes by one-vs-rest.
    """
    # Find unique classes
    classes = []
    for i in range(data.shape[0]):
        if (not(data[i][0] in classes)):
            classes.append(data[i][0])

    # For each classification, train perceptron on current class vs.
    # rest of the untrained classes.
    ws = []
    curr_data = copy.deepcopy(data)
    for curr_class in range(len(classes)):

        # Save original classification data
        orig_classes = copy.deepcopy(curr_data[:,0])

        # Reset classification data to 1 (for current class) or -1 for other
        for i in range(curr_data.shape[0]):
            if (curr_data[i][0] == classes[curr_class]):
                curr_data[i][0] = 1
            else:
                curr_data[i][0] = -1

        # Train and find weights
        ws.append(train_perceptron(curr_data, beta, step_limit))

        # Put original classifications back
        for i in range(curr_data.shape[0]):
            curr_data[i][0] = orig_classes[i]

    return ws


def test_perceptron(data, w):
    """
    Given test data and a weight vector w, return number of
    num_misclass when classifying the test data using the
    weights.
    """
    errors = 0

    # Initialize y_hat
    y_hat = np.zeros(len(data))

    # Slice off y
    y = data[:,0]

    # Determine how weights classify each test sample and count
    # num_misclass
    for i in range(len(data)):
        biased_sample = np.copy(data[i])
        biased_sample[0] = 1
        y_hat[i] = 1 if (np.matmul(w.T, biased_sample) > 0) else -1
        if (y[i] != y_hat[i]):
            errors += 1

    return errors


def multiclass_test_perceptron(data, ws):
    """
    Given test data and a weight vector w, return number of
    num_misclass when classifying the test data using the
    weights.
```

```python

    This version works on data with multiple classes by One vs. All (OVA).
    """
    # Find unique classes
    classes = []
    for i in range(data.shape[0]):
        if (not(data[i][0] in classes)):
            classes.append(data[i][0])

    # For each classification, test perceptron on current class vs.
    # rest of the untested classes.
    errors = []
    curr_data = copy.deepcopy(data)
    for curr_class in range(len(classes)):

        # Save original classification data
        orig_classes = copy.deepcopy(curr_data[:,0])

        # Reset classification data to 1 (for current class) or -1 for other
        for i in range(curr_data.shape[0]):
            if (curr_data[i][0] == classes[curr_class]):
                curr_data[i][0] = 1
            else:
                curr_data[i][0] = -1

        # Train and find weights
        errors.append(test_perceptron(curr_data, ws[curr_class]))

        # Put original classifications back
        for i in range(curr_data.shape[0]):
            curr_data[i][0] = orig_classes[i]

    return errors


def iris_knn():
    """
    Run kNN on the iris dataset for the various numbers of neighbors.
    """
    print("----------\niris kNN")

    # Load data
    data = libsvm_scale_import('data/iris.scale')

    # Shuffle the data because we want to split it into train & test,
    # and it is pre-sorted (we would test against classes we didn't
    # see in training)
    np.random.seed(1) # ensure consistent shuffling
    np.random.shuffle(data)

    # Split up data into training and test data based on split value
    split = 50
    train_data = data[:split]
    test_data = data[split:]

    # Test multiple values of k
    test_ks = np.arange(1, split)
    error_rates = np.zeros(test_ks.shape[0])
    for i in range(len(test_ks)):
        # Classify the test data
        print('Classify with k =', test_ks[i])
        classifications = k_nearest_neighbors(train_data, test_data,
                                              test_ks[i])
        # Check accuracy
        errors = check_knn_classifications(test_data[:,0], classifications)
        error_rates[i] = errors / test_data.shape[0]
        print(errors, 'errors in', test_data.shape[0], 'samples')

```

```python
    print('ks:', test_ks)
    print('error rates:', error_rates)
    plt.clf()
    plt.plot(test_ks, error_rates, marker='.')
    plt.title('Iris kNN: error rate vs. k')
    plt.xlabel('k')
    plt.ylabel('error rate')
    plt.xlim(left = 0)
    plt.ylim(bottom = 0)
    plt.grid(True)
    plt.savefig('iris_knn.png', dpi = 600)


def iris_perceptron():
    """
    Run Perceptron on the iris dataset in various ways.
    """
    print("--------------------\niris Perceptron")

    # Load data
    data = libsvm_scale_import('data/iris.scale')

    # Shuffle the data because we want to split it into train & test,
    # and it is pre-sorted (we would test against classes we didn't
    # see in training)
    np.random.seed(1) # ensure consistent shuffling
    np.random.shuffle(data)

    # Split up data into training and test data based on split value
    split = 50
    train_data = data[:split]
    test_data = data[split:]

    # Perform multi-class training and test and collect
    # a weight vector and number of errors for each class
    ws = multiclass_train_perceptron(train_data, 0.1, 100000)
    errors = multiclass_test_perceptron(test_data, ws)

    # Report errors
    print(errors, 'errors in', test_data.shape[0], 'samples')

    # Show sorted weights for every class
    for i in range(len(ws)):

        # Sort weights to find most important
        w = list(ws[i][1:])
        feature_ids = range(1, len(w) + 1)
        print('W:', w)
        labels = []
        for id in feature_ids:
            labels.append(str(int(id)))

        # Report top weights
        plt.clf()
        plt.bar(labels, w)
        plt.title('iris Perceptron: feature weights for class = ' + str(i+1))
        plt.xlabel('feature ID')
        plt.ylabel('weight')
        plt.grid(True)
        plt.savefig('iris_weights' + str(i+1) + '.png', dpi = 600)


def a4a_knn():
    """
    Run kNN on the a4a dataset for various numbers of neighbors.
    """
    print("--------------------\na4a kNN")

```

```
    # Load data
    train_data = libsvm_scale_import('data/a4a')
    test_data = libsvm_scale_import('data/a4a.t')

    # Training data has 1 fewer feature than test data, so add a column
    # of zeros to it so samples have same number of features in train and test
    zero_col = np.zeros((len(train_data), 1))
    train_data = np.hstack((train_data, zero_col))

    # Test multiple values of k
    # This takes over 3 hours to run on my fastest computer.
    test_ks = np.array([1, 3, 5, 11, 21, 31, 41, 51, 61, 71, 81, 91, 101, 201, 301, 401, 501, 601, 701, 801, 901, 1001])
    error_rates = np.zeros(len(test_ks))
    for i in range(len(test_ks)):
        print('Classify with k =', test_ks[i])
        # Classify the test data
        classifications = k_nearest_neighbors(train_data, test_data,
                                              test_ks[i])
        # Check accuracy
        errors = check_knn_classifications(test_data[:,0], classifications)
        error_rates[i] = errors / test_data.shape[0]
        print(errors, 'errors in', test_data.shape[0], 'samples')

    print('ks:', test_ks)
    print('error rates:', error_rates)
    plt.clf()
    plt.plot(test_ks, error_rates, marker='.')
    plt.title('a4a kNN: error rate vs. k')
    plt.xlabel('k')
    plt.ylabel('error rate')
    plt.xlim(left = 0)
    plt.ylim(bottom = 0)
    plt.grid(True)
    plt.savefig('a4a_knn.png', dpi = 600)


def a4a_perceptron():
    """
    Run Perceptron on the a4a dataset in various ways.
    """
    print("----------\na4a Perceptron")

    # Load data
    train_data = libsvm_scale_import('data/a4a')
    test_data = libsvm_scale_import('data/a4a.t')

    # Training data has 1 fewer feature than test data, so add a column
    # of zeros to it so samples have same number of features in train and test
    zero_col = np.zeros((len(train_data), 1))
    train_data = np.hstack((train_data, zero_col))

    # Test multiple values of beta
    test_betas = np.array([0.0001, 0.001, 0.01, 0.1, 1.0, 10.0, 100.0])
    error_rates = np.zeros(test_betas.shape[0])
    ws = []
    best_beta = -1
    best_error_rate = 999999
    for i in range(len(test_betas)):
        print('Classify with beta =', test_betas[i])

        # Train and find weights
        ws.append(train_perceptron(train_data, test_betas[i], 100000))

        # Check accuracy
        errors = test_perceptron(test_data, ws[i])
        error_rates[i] = errors / test_data.shape[0]
        if (error_rates[i] < best_error_rate):
```

```
                best_error_rate = error_rates[i]
                best_beta = i
        print(errors, 'errors in', test_data.shape[0], 'samples')

    # Report error rates
    print('betas:', test_betas)
    print('error rates:', error_rates)
    plt.clf()
    plt.plot(test_betas, error_rates, marker='.')
    plt.title('a4a Perceptron: error rate vs. step size for 100000 iterations')
    plt.xscale('log')
    plt.xlabel('step size')
    plt.ylabel('error rate')
    plt.ylim(bottom = 0)
    plt.grid(True)
    plt.savefig('a4a_perceptron.png', dpi = 600)

    # Sort weights to find most important
    w = list(ws[best_beta][1:])
    feature_ids = range(1, len(w) + 1)
    bar_data = list(zip(feature_ids, w))
    bar_data.sort(key = lambda _: abs(_[1]), reverse = True)
    bar_data = np.array(bar_data[:20])
    labels = []
    for id in bar_data[:,0]:
        labels.append(str(int(id)))

    # Report top weights
    plt.clf()
    plt.bar(labels, bar_data[:,1])
    plt.title('a4a Perceptron: 20 most important features')
    plt.xlabel('feature ID')
    plt.ylabel('weight')
    plt.grid(True)
    plt.savefig('a4a_weights.png', dpi = 600)


def main():
    iris_knn()
    iris_perceptron()
    a4a_knn()
    a4a_perceptron()


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

MiniProj1.py

7 Run Log

```
——————————
iris kNN
LOADED: data/iris.scale : (150, 5)
 Classify with k = 1
knn test sample 80
6 errors in 100 samples
 Classify with k = 2
knn test sample 80
12 errors in 100 samples
 Classify with k = 3
knn test sample 80
8 errors in 100 samples
 Classify with k = 4
knn test sample 80
9 errors in 100 samples
```

```
Classify with k = 5
knn test sample 80
7 errors in 100 samples
Classify with k = 6
knn test sample 80
11 errors in 100 samples
Classify with k = 7
knn test sample 80
8 errors in 100 samples
Classify with k = 8
knn test sample 80
12 errors in 100 samples
Classify with k = 9
knn test sample 80
12 errors in 100 samples
Classify with k = 10
knn test sample 80
12 errors in 100 samples
Classify with k = 11
knn test sample 80
10 errors in 100 samples
Classify with k = 12
knn test sample 80
13 errors in 100 samples
Classify with k = 13
knn test sample 80
12 errors in 100 samples
Classify with k = 14
knn test sample 80 40
14 errors in 100 samples
Classify with k = 15
knn test sample 80
12 errors in 100 samples
Classify with k = 16
knn test sample 80
12 errors in 100 samples
Classify with k = 17
knn test sample 80
12 errors in 100 samples
Classify with k = 18
knn test sample 80
14 errors in 100 samples
Classify with k = 19
knn test sample 80
14 errors in 100 samples
Classify with k = 20
knn test sample 80
16 errors in 100 samples
Classify with k = 21
knn test sample 80
15 errors in 100 samples
Classify with k = 22
knn test sample 80
19 errors in 100 samples
Classify with k = 23
knn test sample 80
16 errors in 100 samples
Classify with k = 24
knn test sample 80
19 errors in 100 samples
Classify with k = 25
knn test sample 80
18 errors in 100 samples
Classify with k = 26
knn test sample 80
20 errors in 100 samples
Classify with k = 27
knn test sample 80
```

```
19 errors in 100 samples
Classify with k = 28
knn test sample 80
36 errors in 100 samples
Classify with k = 29
knn test sample 80
36 errors in 100 samples
Classify with k = 30
knn test sample 80
36 errors in 100 samples
Classify with k = 31
knn test sample 80
36 errors in 100 samples
Classify with k = 32
knn test sample 80 40
36 errors in 100 samples
Classify with k = 33
knn test sample 80
36 errors in 100 samples
Classify with k = 34
knn test sample 80
36 errors in 100 samples
Classify with k = 35
knn test sample 80
67 errors in 100 samples
Classify with k = 36
knn test sample 80
69 errors in 100 samples
Classify with k = 37
knn test sample 80
69 errors in 100 samples
Classify with k = 38
knn test sample 80
69 errors in 100 samples
Classify with k = 39
knn test sample 80
69 errors in 100 samples
Classify with k = 40
knn test sample 60 80
69 errors in 100 samples
Classify with k = 41
knn test sample 80
69 errors in 100 samples
Classify with k = 42
knn test sample 80
69 errors in 100 samples
Classify with k = 43
knn test sample 80
69 errors in 100 samples
Classify with k = 44
knn test sample 80
69 errors in 100 samples
Classify with k = 45
knn test sample 80
69 errors in 100 samples
Classify with k = 46
knn test sample 80
69 errors in 100 samples
Classify with k = 47
knn test sample 80
69 errors in 100 samples
Classify with k = 48
knn test sample 8020
69 errors in 100 samples
Classify with k = 49
knn test sample 80
69 errors in 100 samples
ks: [ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
```

```
 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48
 49]
error rates: [0.06 0.12 0.08 0.09 0.07 0.11 0.08 0.12 0.12 0.12 0.1  0.13 0.12 0.14
 0.12 0.12 0.12 0.14 0.14 0.16 0.15 0.19 0.16 0.19 0.18 0.2  0.19 0.36
 0.36 0.36 0.36 0.36 0.36 0.36 0.67 0.69 0.69 0.69 0.69 0.69 0.69 0.69
 0.69 0.69 0.69 0.69 0.69 0.69 0.69]
--------------------
iris Perceptron
LOADED: data/iris.scale : (150, 5)
Perceptron: 100 steps; converged? True
Perceptron: 100000 steps; converged? False
Perceptron: 250 steps; converged? True
[0, 68, 58] errors in 100 samples
W: [-0.1222222, 0.15, -0.28813540000000004, -0.2666666]
W: [-0.16614598199997474, -0.46739085999993474, 0.5187340000001649, -0.6666498142802026]
W: [0.05555557298200006, -0.33333285999999995, 0.39322, 0.35000042421438]
--------------------
a4a kNN
LOADED: data/a4a : (4781, 123)
LOADED: data/a4a.t : (27780, 124)
Classify with k = 1
knn test sample 27760
5937 errors in 27780 samples
Classify with k = 3
knn test sample 27760
5238 errors in 27780 samples
Classify with k = 5
knn test sample 27760
5028 errors in 27780 samples
Classify with k = 11
knn test sample 27760
4769 errors in 27780 samples
Classify with k = 21
knn test sample 27760
4663 errors in 27780 samples
Classify with k = 31
knn test sample 27760
4639 errors in 27780 samples
Classify with k = 41
knn test sample 27760
4626 errors in 27780 samples
Classify with k = 51
knn test sample 27760
4664 errors in 27780 samples
Classify with k = 61
knn test sample 27760
4656 errors in 27780 samples
Classify with k = 71
knn test sample 27760
4668 errors in 27780 samples
Classify with k = 81
knn test sample 27760
4695 errors in 27780 samples
Classify with k = 91
knn test sample 27760
4714 errors in 27780 samples
Classify with k = 101
knn test sample 27760
4721 errors in 27780 samples
Classify with k = 201
knn test sample 27760
4845 errors in 27780 samples
Classify with k = 301
knn test sample 27760
4871 errors in 27780 samples
Classify with k = 401
knn test sample 27760
4911 errors in 27780 samples
```

```
Classify with k = 501
knn test sample 27760
4940 errors in 27780 samples
Classify with k = 601
knn test sample 27760
4980 errors in 27780 samples
Classify with k = 701
knn test sample 27760
5007 errors in 27780 samples
Classify with k = 801
knn test sample 27760
5052 errors in 27780 samples
Classify with k = 901
knn test sample 27760
5109 errors in 27780 samples
Classify with k = 1001
knn test sample 27760
5227 errors in 27780 samples
ks: [    1    3    5   11   21   31   41   51   61   71   81   91  101  201
  301  401  501  601  701  801  901 1001]
error rates: [0.2137149  0.18855292 0.18099352 0.17167027 0.16785457 0.16699064
 0.16652268 0.16789057 0.16760259 0.16803456 0.16900648 0.16969042
 0.1699424  0.17440605 0.17534197 0.17678186 0.17782577 0.17926566
 0.18023758 0.18185745 0.18390929 0.18815695]
--------------
a4a Perceptron
LOADED: data/a4a : (4781, 123)
LOADED: data/a4a.t : (27780, 124)
Classify with beta = 0.0001
Perceptron: 100401 steps; converged? False
5094 errors in 27780 samples
Classify with beta = 0.001
Perceptron: 100401 steps; converged? False
5911 errors in 27780 samples
Classify with beta = 0.01
Perceptron: 100401 steps; converged? False
7966 errors in 27780 samples
Classify with beta = 0.1
Perceptron: 100401 steps; converged? False
5450 errors in 27780 samples
Classify with beta = 1.0
Perceptron: 100401 steps; converged? False
6052 errors in 27780 samples
Classify with beta = 10.0
Perceptron: 100401 steps; converged? False
6052 errors in 27780 samples
Classify with beta = 100.0
Perceptron: 100401 steps; converged? False
6052 errors in 27780 samples
betas: [1.e-04 1.e-03 1.e-02 1.e-01 1.e+00 1.e+01 1.e+02]
error rates: [0.18336933 0.21277898 0.28675306 0.19618431 0.21785457 0.21785457
 0.21785457]
```