COMP 6636 Mini-project 2

Dennis Brown, dgb0028@auburn.edu

April 23, 2021

1 Overview

This Mini-project looks at training accuracy and classification error using 2-layer Neural Network, Support Vector Machine, and Kernel Perceptron. The project was implemented in Python 3 and the code is included in section 6 of this report.

The project uses the Modified National Institute of Standards and Technology (MNIST) database of 60,000 handwritten digits 0 to 9, where the samples are 780-pixel bitmaps of the digit images. 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. For all runs, the data was split into 70% training and 30% testing data.

For each of the three classifiers:

- First, the classifier was trained and tested with many variations of hyperparameters in order to find the optimal (or near-optimal) values of each hyperparameter. In the interest of time, these trials were run on a small subset of the data (1000 samples split 70/30).
- Second, the classifer was trained and tested on the entire dataset (60000 samples split 70/30) using the best hyperparameters identified in the first step.

2 2-layer Neural Network

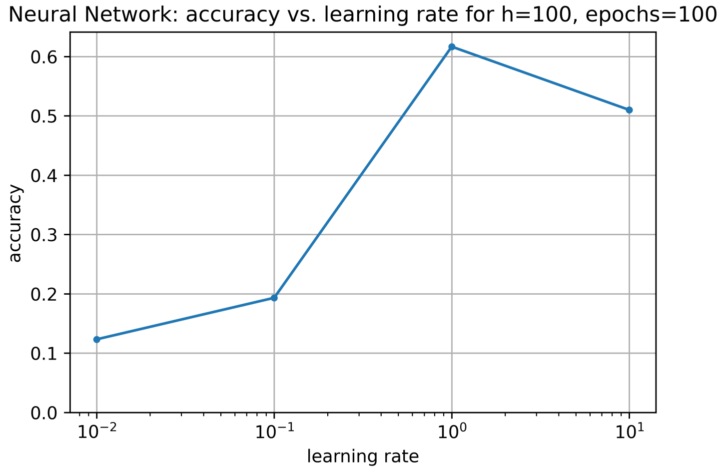
The 2-layer Neural Network uses a hidden layer of 100 units (as specified), although other numbers of hidden units were tested as well. The link/activation function is Sigmoid. A bias term was added to each input sample and output of the hidden layer. The decimal training labels were converted to 4-digit binary numbers and the network has 4 inputs and 4 outputs. The outputs are then converted back to decimal.

2.1 Finding optimal parameters

2.1.1 Learning Rate

The learning rate was varied from 0.01 to 10.0 with fixed values of 100 hidden layers and 100 epochs. 1.0 performed best.

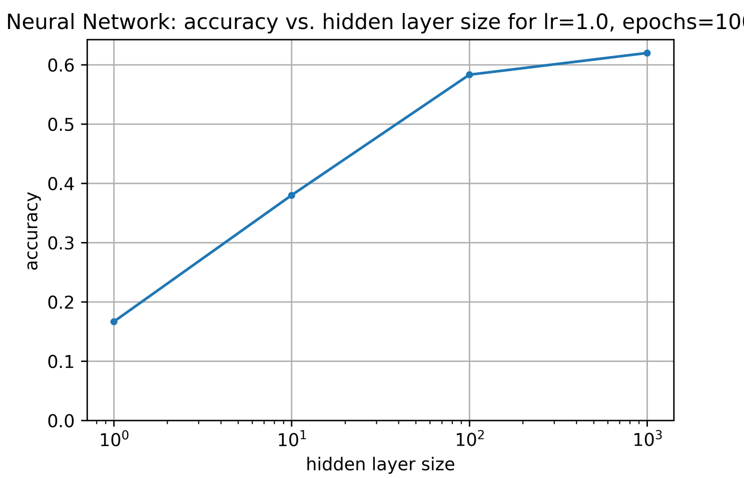
Figure 1: Neural Network: accuracy vs. learning rate



2.1.2 Hidden Units

The size of the hidden layer was varied from 1 to 1000 with fixed values of 1.0 learning rate and 100 epochs. 1000 units performed best, but only slightly better than 100 and with a much higher computational cost. 100 hidden units were used in all other Neural Network trials.

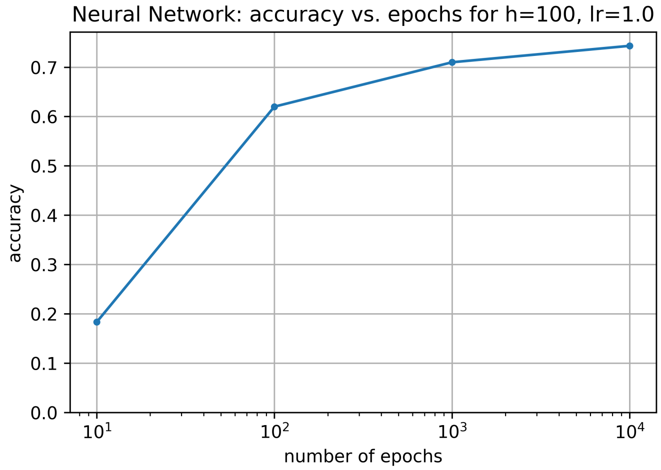
Figure 2: Neural Network: accuracy vs. hidden units



2.1.3 Epochs

The number of epochs was varied from 10 to 10000 with fixed values of 1.0 learning rate and 100 hidden units. 10000 epochs performed best, but only slightly better than 1000 and with a much higher computational cost. 1000 epochs were used in all other Neural Network trials.

Figure 3: Neural Network: accuracy vs. epochs



2.2 Performance on entire dataset

Using the entire dataset, the Neural Network was trained on 42000 samples and tested on 18000 samples using 100 hidden layers, 1.0 learning rate, and 1000 epochs.

The accuracy was 15489/18000 (86.05%).

The confusion matrix is shown below. One can observe "hotspots" where the digit 5 was misclassified as 1, 8 as 9, 8 as 0, and 3 as 1 and 2. The author isn't quite sure why that would happen other than noting those digits can look similar when handwritten.

Figure 4: Confusion Matrix: NN on entire dataset

Actual

Predicted	0	1	2	3	4	5	6	7	8	9
0	1663	19	66	15	41	32	18	4	99	23
1	17	1879	4	100	16	137	1	24	63	71
2	26	4	1579	93	1	11	42	13	25	2
3	5	12	40	1503	1	30	0	68	20	19
4	27	2	17	2	1473	72	55	14	22	15
5	4	14	3	35	64	1211	4	83	9	55
6	8	0	56	2	36	23	1609	18	2	0
7	0	2	19	50	2	82	4	1680	10	22
8	25	6	18	10	47	5	3	0	1440	80
9	5	5	9	53	62	34	2	31	121	1452

3 Linear Support Vector Machine

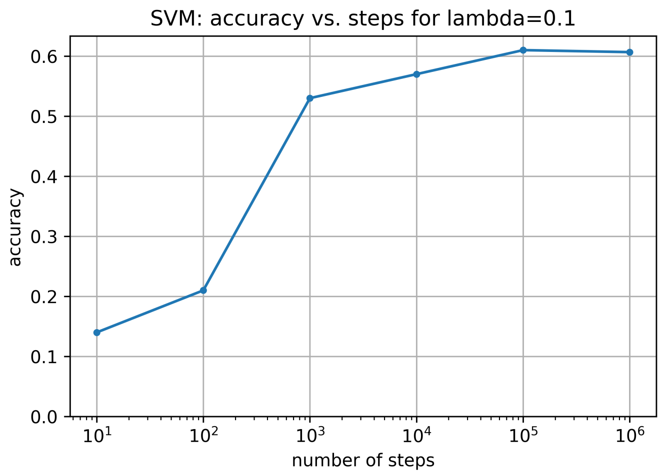
The linear SVM is a binary classifier. Employing a form of Error-Correcting Output Code (ECOC), the decimal training labels were converted to 4-digit binary numbers and fed into four instances of the linear SVM. The output of those 4 instances were then combined and converted back to decimal.

3.1 Finding optimal parameters

3.1.1 Steps

The number of steps was varied from 10 to 1000000 with a fixed value of 0.1 lambda. 100000 steps performed best.

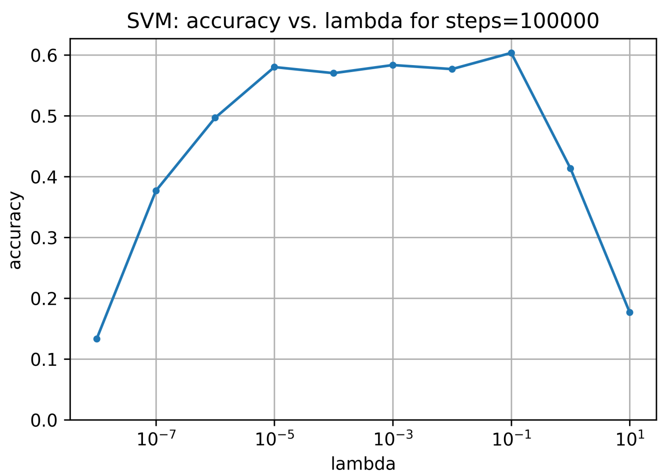
Figure 5: SVM: accuracy vs. steps



3.1.2 Lambda

The value of lambda was varied from 1e-8 to 10 with a fixed value of 100,000 steps. Lambda = 0.1 performed best.

Figure 6: SVM: accuracy vs. lambda



3.2 Performance on entire dataset

Using the entire dataset, the four SVMs were trained on 42000 samples and tested on 18000 samples using 100000 steps and lambda = 0.1.

The accuracy was 11473/18000 (63.74%).

The confusion matrix is shown below. One should note a lot of confusion here, as would be indicated by the medicre performance in accuracy. The worst offenses seem to be misclassifying 8 as 0, 5 as 1, 7 as 5, and 3 as 1.

Figure 7: Confusion Matrix: SVM on entire dataset

		Actual									
		0	1	2	3	4	5	6	7	8	9
Predicted	0	1402	43	152	44	39	165	88	11	461	120
	1	69	1834	30	376	48	445	4	110	337	186
	2	175	4	1384	121	6	40	94	21	52	20
	3	5	22	92	1155	1	75	1	246	36	28
	4	79	2	24	7	1144	176	122	23	51	51
	5	4	8	5	16	309	515	1	424	45	231
	6	36	0	71	18	62	51	1419	16	11	5
	7	0	4	20	49	5	63	4	1008	4	30
	8	8	25	23	13	9	31	4	5	695	151
	9	2	1	10	64	120	76	1	71	119	917

4 Kernel Perceptron

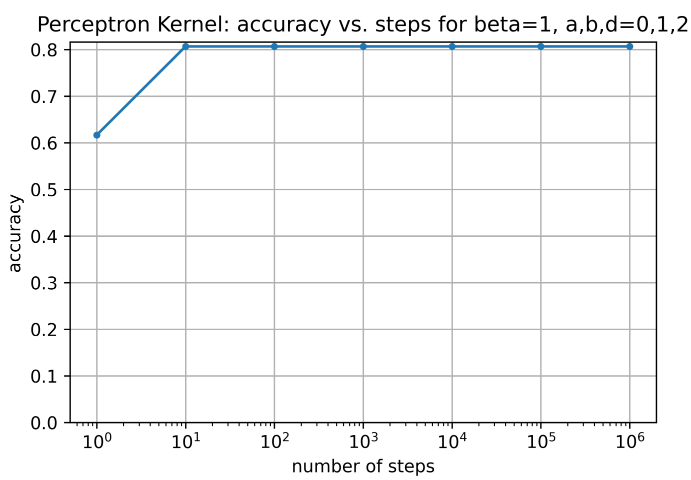
The Kernel Perceptron was implemented with a polynomial kernel $K^d_{poly}(x, z) = (a + bx^T z)^d$. It is a binary classifier. Employing a form of Error-Correcting Output Code (ECOC), the decimal training labels were converted to 4-digit binary numbers and fed into four instances of Kernel Perceptron. The output of those 4 instances were then combined and converted back to decimal.

4.1 Finding optimal parameters

4.1.1 Steps

The number of steps was varied from 1 to 1000000 with fixed values of beta = 1.0 and the polynomial kernel variables a, b, d = 0.0, 1.0, 2.0. There was no improvement after 10 steps.

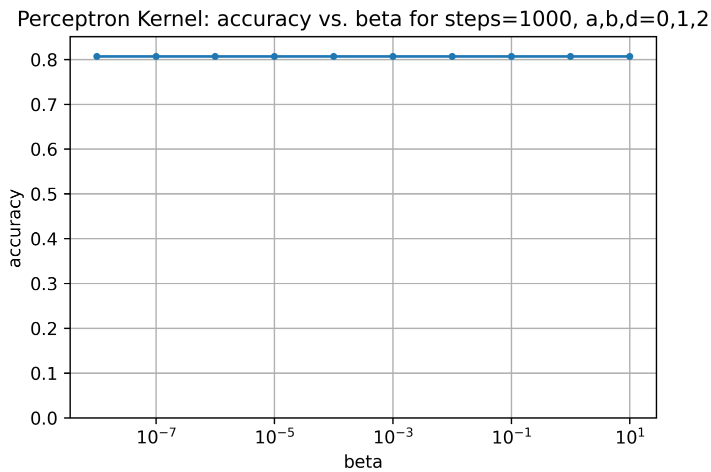
Figure 8: KP: accuracy vs. steps



4.1.2 Beta

The value of beta was varied from 1e-8 to 10 with fixed values of 1000 steps and the polynomial kernel variables a, b, d = 0.0, 1.0, 2.0. The value of beta does not appear to matter. I double-checked and my code does use it.

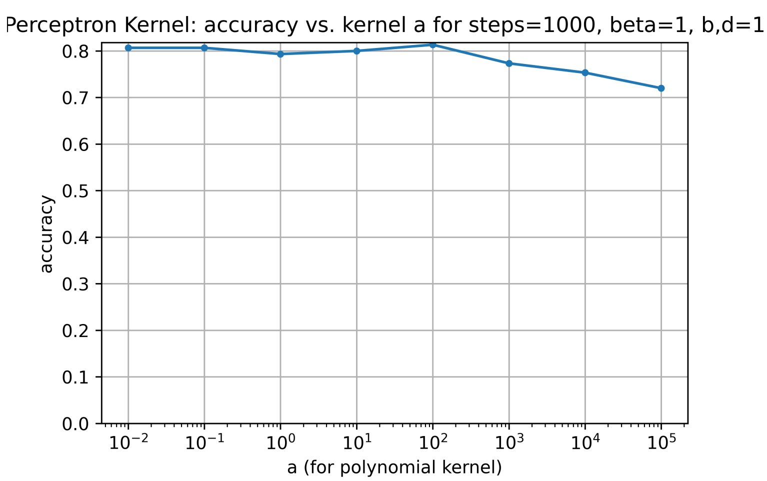
Figure 9: KP: accuracy vs beta



4.1.3 Kernel a

The value of a in the polynomial kernel was varied from 0.01 to 100000 with fixed values of 1000 steps, beta = 1.0, and the polynomial kernel variables b, d = 1.0, 2.0. a = 100 performed best.

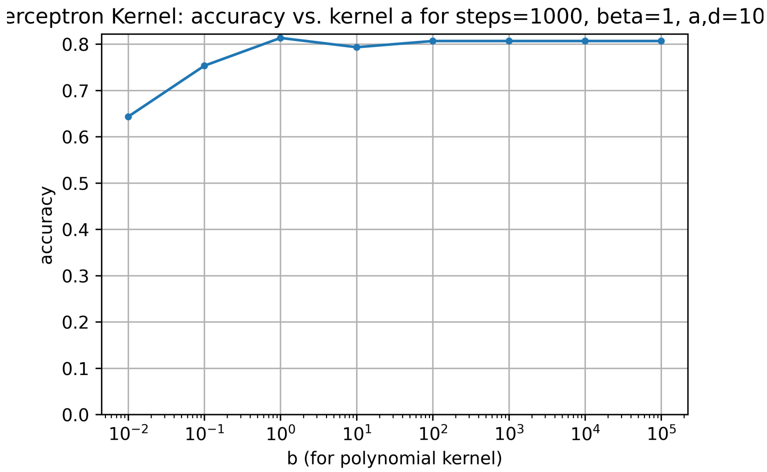
Figure 10: KP: accuracy vs kernel a



4.1.4 Kernel b

The value of b in the polynomial kernel was varied from 0.01 to 100000 with fixed values of 1000 steps, beta = 1.0, and the polynomial kernel variables a, d = 100, 2.0. b = 1.0 performed best.

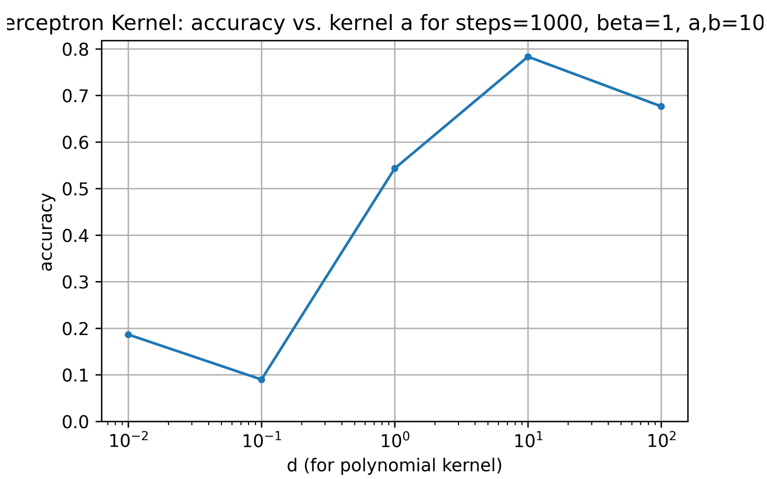
Figure 11: KP: accuracy vs kernel b



4.1.5 Kernel d

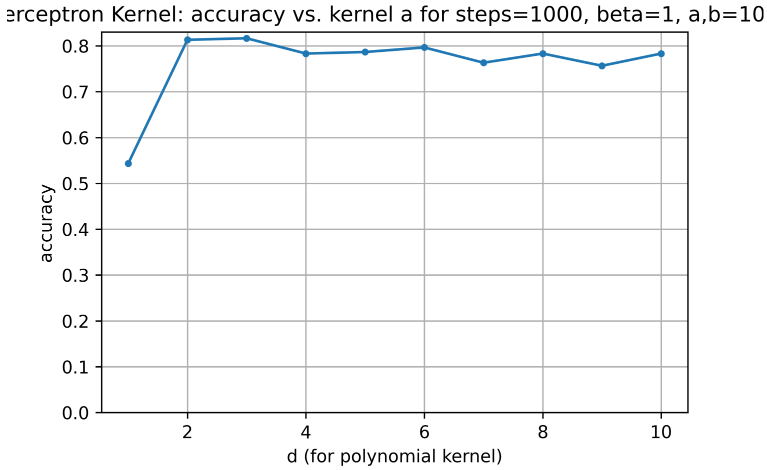
The value of d in the polynomial kernel was first varied from 0.01 to 100 with fixed values of 1000 steps, beta = 1.0, and the polynomial kernel variables a, b = 100, 1.0. d = 10 performed best.

Figure 12: KP: accuracy vs kernel d



However, prior runs showed there was reason to further explore d in the range between 0 and 10. In the end, d = 3 performed best.

Figure 13: KP: accuracy vs kernel d part 2



4.2 Performance on entire dataset

Using the entire dataset, the four Kernel Perceptrons were trained on 42000 samples and tested on 18000 samples using 10 steps, beta = 1.0, and polynomial kernel values a = 100, b = 1.0, and d = 3.0.

The accuracy was 17013/18000 (94.52%).

The confusion matrix is shown below. The worst misclassifications are well under 100 each (much less than SVM) and include misclassifying 8 as 9, 5 as 7, 7 as 3, and 4 as 0.

Figure 14: Confusion Matrix: Kernel Perceptron on entire dataset

Predicted \ Actual	0	1	2	3	4	5	6	7	8	9
0	1740	9	22	1	47	8	4	1	43	10
1	9	1912	1	27	7	40	0	18	11	30
2	14	4	1742	33	3	3	23	5	18	0
3	0	7	16	1748	0	11	0	54	5	9
4	6	0	0	0	1618	18	9	5	2	6
5	0	2	0	5	13	1461	0	18	2	12
6	1	2	10	0	18	12	1693	16	0	0
7	0	1	1	10	2	60	4	1801	0	7
8	8	1	5	8	14	3	1	1	1667	34
9	2	5	14	31	21	21	4	16	63	1631

5 Conclusion

Three classifiers were presented along with an investigation of their hyperparameters. The best-configured Kernel Perceptron had the best accuracy on the entire MNIST dataset of the three classifiers and took about 5 hours of computing time to train and test. The best-configured 2-layer Neural Network followed as a close second in accuracy, but took about 12 hours to train and test. The best-configured SVM had very mediocre accuracy, but only took about 1 hour to train and test.

6 Code

```
# -*- coding: utf-8 -*-
"""
Mini project 2

Dennis Brown, COMP6636, 23 APR 2021
"""

import numpy as np
import random
import sys
import matplotlib.pyplot as plt


###############################################
#
# MNIST data set functions
#
###############################################

def libsvm_scale_import(filename, limit = 0):
    """
    Read data from a libsvm .scale file. Set 'limit' to limit the import
    to a certain number of samples.
    """
    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
    # If the limit is set, import up to the limit, otherwise import all
    import_samples = min(num_samples, limit) if limit else num_samples
    features = np.zeros((import_samples, max_feature_id))
    classes = np.zeros((import_samples, 1))
    curr_sample = 0
    # Read the samples
    datafile.seek(0)
    for line in datafile:
        # Stop at the limit if it's set
        if (limit and (curr_sample >= limit)): break
        tokens = line.split()
        # Read the classification
        classes[curr_sample][0] = float(tokens[0])
        # Read the features
        for feature in tokens[1:]:
            feature_id = int(feature.split(':')[0])
            feature_val = float(feature.split(':')[1])
            features[curr_sample][feature_id - 1] = feature_val
        curr_sample += 1
    datafile.close()

    print('LOADED:', filename, ':', classes.shape, features.shape)

    return classes, features


def convert_mnist_classes_to_binary(classes):
    """
    Given a list of integer MNIST classes, return an array where each class is
```

```
    converted to binary. e.g., 5 -> [0. 1. 0. 1.]
    """
    binary_classes = np.zeros((classes.shape[0], 4))
    for i in range(classes.shape[0]):
        boolver = bin(int(classes[i][0]))[2:].zfill(4)
        for bit in range(len(boolver)): binary_classes[i][bit] = float(boolver[bit])
        # print(classes[i][0], binary_classes[i])
    return binary_classes


def convert_mnist_classes_to_integer(binary_classes):
    """
    Given a list of binary MNIST classes, return an array where each class is
    converted to integer. e.g., [0. 1. 0. 1.] -> 5
    """
    classes = np.zeros((binary_classes.shape[0], 1))
    for i in range(binary_classes.shape[0]):
        bins = binary_classes[i]
        # Not very elegant, but it works -- convert to integer and cap at 9
        classes[i][0] = min((bins[0] * 8) + (bins[1] * 4) + (bins[2] * 2) + bins[3], 9)
        # print(classes[i][0], binary_classes[i])
    return classes


###############################################
#
# Neural Network functions
#
###############################################

def sigmoid(x):
    """
    Sigmoid Function
    """
    return 1 / (1 + np.exp(-x))


def train_neural_network(X, y, H_size, learning_rate, epochs):
    """
    2-Layer Neural Network: Given a 2-D set of data X (samples are rows,
    columns features), a vector y of classifications, a number of
    hidden-layer neurons, a learning rate, and number of epochs,
    train a 2-layer neural network. Return weight matrices.
    """
    # Randomly initialize the weights for the input -> hidden layer
    xh = (np.random.random((X.shape[1] + 1, H_size))) * 2 - 1

    # Randomly initialize the weights for the hidden layer -> output
    hy = (np.random.random((H_size + 1, y.shape[1]))) * 2 - 1

    for epoch in range(epochs):
        # print(str(epoch) + ' ', end = '')
        # sys.stdout.flush()

        # ---------------------
        # Forward Propagation
        # ---------------------

        # Add bias terms to X
        X_bias = np.hstack((X, np.ones([X.shape[0], 1])))

        # Calculate hidden layer outputs
        H_output = sigmoid(np.dot(X_bias, xh))

        # Add bias terms to H_output
        H_output_bias = np.hstack((H_output, (np.ones([H_output.shape[0], 1]))))

        y_hat = sigmoid(np.dot(H_output_bias, hy))
```

```
        # --------------------
        # Backward Propagation
        # --------------------

        # Find error
        y_error = y_hat - y

        # Calculate hidden layer error (remove bias from hy)
        H_error = H_output * (1 - H_output) * np.dot(y_error, hy.T[:, 1:])

        # Calculate partial derivatives
        H_pd = X_bias[:, :, np.newaxis] * H_error[: , np.newaxis, :]
        y_pd = H_output_bias[:, :, np.newaxis] * y_error[:, np.newaxis, :]

        # Calculate total gradients for hidden and output layers
        # (find average of each column)
        H_gradient = np.average(H_pd, axis = 0)
        y_gradient = np.average(y_pd, axis = 0)

        # Update weights using learning rate and gradients
        xh -= (learning_rate * H_gradient)
        hy -= (learning_rate * y_gradient)

    # print()

    # Return weight matrices when finished
    return xh, hy


def test_neural_network(X, xh, hy):
    """
    2-layer Neural Network: Given a 2-D set of data X (samples are rows,
    columns features) and weight matrices for the 2-layer network,
    return the predicted output.
    """
    X_bias = np.hstack((X, np.ones([X.shape[0], 1])))
    H_output = sigmoid(np.dot(X_bias, xh))
    H_output_bias = np.hstack((H_output, (np.ones([H_output.shape[0], 1]))))
    y_hat = sigmoid(np.dot(H_output_bias, hy))
    return y_hat


def mnist_neural_network(train_classes, test_classes, train_features,
                         test_features, h_size, learning_rate, num_epochs):
    """
    Given MNIST features and classes split into training and testing data,
    train and evaluate Neural Network.
    """
    # Convert classifications to binary
    binary_train_classes = convert_mnist_classes_to_binary(train_classes)

    # Train
    xh, hy = train_neural_network(train_features, binary_train_classes,
                                  h_size, learning_rate, num_epochs)

    # Test
    binary_pred_classes = test_neural_network(test_features, xh, hy)
    binary_pred_classes = 1.0 * (binary_pred_classes > 0.5)
    pred_classes = convert_mnist_classes_to_integer(binary_pred_classes)

    # Create label for this evaluation
    label = str(train_features.shape[0] + test_features.shape[0])
    label += '_' + str(h_size)
    label += '_' + str(learning_rate)
    label += '_' + str(num_epochs)

    # Calculate number correct
```

```python
    correct = 0
    cm = np.zeros((10, 10))
    for i in range(test_classes.shape[0]):
        if (pred_classes[i][0] == test_classes[i][0]): correct += 1
        cm[int(pred_classes[i][0])][int(test_classes[i][0])] += 1
    print('Correct:', correct, '/', test_classes.shape[0], 'for', label)
    print(cm)

    np.savetxt('./data/confusion_nn_' + label + '.csv', cm, delimiter=',', fmt='%10.0f')

    return correct / test_classes.shape[0]


##################################################
#
# Support Vector Machine functions
#
##################################################

def train_svm(X, y, lam, limit):
    """
    Support Vector Machine. Given a sample matrix X,
    a vector Y of classifications, a regularization parameter lam,
    and a step limit, train and return a weight vector that
    can be used to classify the given data.
    """
    # Convert (1, 0) to (1, -1)
    y = y * 2 - 1

    # Initialize the weight vector
    w = np.zeros(X.shape[1])

    # Pegasos algorithm
    # Repeat the main loop until we reach the iteration limit
    t = 1
    while(t <= limit):
        i = random.randint(0, X.shape[0] - 1)
        eta = 1.0 / (lam * t)
        y_hat = y[i][0] * np.matmul(w, X[i])
        if (y_hat < 1.0):
            w = ((1 - (eta * lam)) * w) + (eta * y[i][0] * X[i])
        else:
            w = ((1 - (eta * lam)) * w)
        if (np.linalg.norm(w) > 0.0):
            w = min(1.0, ((1.0 / np.sqrt(lam)) / (np.linalg.norm(w)))) * w
        t += 1

    return w


def test_svm(X, w):
    """
    Support Vector Machine. Given a sample matrix X
    and a weight vector, predict the classes of X.
    """
    # Calculate predictions
    y_hat = np.zeros((X.shape[0], 1))
    for i in range(X.shape[0]):
        y_hat[i][0] = np.matmul(w, X[i])

    # Convert to (1, -1)
    y_hat = np.sign(y_hat)

    # Convert (1, -1) to (1, 0)
    y_hat = (y_hat + 1) / 2

    return y_hat

```

```python

def mnist_svm(train_classes, test_classes, train_features, test_features,
              limit, lam):
    """
    Given MNIST features and classes split into training and testing data,
    train and evaluate Support Vector Machine.
    """
    # Convert classes to four binary y vectors
    binary_train_classes = convert_mnist_classes_to_binary(train_classes)
    y1 = binary_train_classes[:,[0]]
    y2 = binary_train_classes[:,[1]]
    y3 = binary_train_classes[:,[2]]
    y4 = binary_train_classes[:,[3]]

    # Train on the four y vectors
    w1 = train_svm(train_features, y1, lam, limit)
    w2 = train_svm(train_features, y2, lam, limit)
    w3 = train_svm(train_features, y3, lam, limit)
    w4 = train_svm(train_features, y4, lam, limit)

    # Get binary predictions from the four perceptrons
    y_hat1 = test_svm(test_features, w1)
    y_hat2 = test_svm(test_features, w2)
    y_hat3 = test_svm(test_features, w3)
    y_hat4 = test_svm(test_features, w4)

    # Convert binary predictions back to decimal
    binary_pred_classes = np.hstack((y_hat1, y_hat2, y_hat3, y_hat4))
    pred_classes = convert_mnist_classes_to_integer(binary_pred_classes)

    # Create label for this evaluation
    label = str(train_features.shape[0] + test_features.shape[0])
    label += '_' + str(limit)
    label += '_' + str(lam)

    # Calculate number correct
    correct = 0
    cm = np.zeros((10, 10))
    for i in range(test_classes.shape[0]):
        if (pred_classes[i][0] == test_classes[i][0]): correct += 1
        cm[int(pred_classes[i][0])][int(test_classes[i][0])] += 1
    print('Correct:', correct, '/', test_classes.shape[0], 'for', label)
    print(cm)

    np.savetxt('./data/confusion_svm_' + label + '.csv', cm, delimiter=',', fmt='%10.0f')

    return correct / test_classes.shape[0]


############################################
#
# Perceptron Kernel functions
#
############################################


def poly_kernel(x, z, a, b, d):
    """
    Calculate polynomial kernel for samples x and z.
    a, b, and d are hyperparameters.
    """
    return (a + (b * (np.matmul(x.T, z))) ** d)


def gram(X, ka, kb, kd):
    """
    Calculate Gram Matrix given X and parameters for poly kernel
    """
```

```python
    G = np.zeros((X.shape[0], X.shape[0]))
    for i in range(X.shape[0]):
        for j in range(X.shape[0]):
            G[i][j] = poly_kernel(X[i], X[j], ka, kb, kd)

    return G


def train_perceptron_kernel(G, y, beta, step_limit):
    """
    Perceptron with a kernel. Given a Gram matrix G,
    a vector Y of classifications, a learning rate (beta),
    and a step limit, train and return a weight vector that
    can be used to classify the given data.
    """
    # Convert (1, 0) to (1, -1)
    y = y * 2 - 1

    # Initialize the alpha vector
    a = np.zeros(G.shape[0])

    # Initialize y_hat
    y_hat = np.zeros((G.shape[0], 1))

    # 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 # assume converged until we determine otherwise

        # For each sample in X, calculate alpha's classification error
        # and update alpha.
        for i in range(G.shape[0]):

            # Find current prediction based on kernel
            y_hat[i][0] = np.sign(np.matmul(G[i,:], a))

            # If error on this element is > a very small value (is not
            # effectively 0), we need to update alpha, and have not converged.
            error = y[i][0] - y_hat[i][0]
            if (abs(error) > 0.000001):
                a[i] += beta * y[i][0]
                converged = False
        steps += 1

    return a


def test_perceptron_kernel(Xtrain, Xtest, a, ka, kb, kd):
    """
    Perceptron with a kernel. Given a sample matrices Xtrain and Xtest,
    and vector a, return predicted classes.
    """
    y_hat = np.zeros((Xtest.shape[0], 1))

    for i in range(Xtest.shape[0]):
        for j in range(a.shape[0]):
            y_hat[i][0] += a[j] * poly_kernel(Xtrain[j], Xtest[i], ka, kb, kd)

    # Convert to (1, -1)
    y_hat = np.sign(y_hat)

    # Convert (1, -1) to (1, 0)
    y_hat = (y_hat + 1) / 2

    return y_hat
```

```python

def mnist_perceptron_kernel(train_classes, test_classes, train_features,
                            test_features, limit, beta, ka, kb, kd):
    """
    Given MNIST features and classes split into training and testing data,
    train and evaluate Kernel Perceptron. ka, kb, and kd are for poly kernel.
    """
    # Convert classes to four binary y vectors
    binary_train_classes = convert_mnist_classes_to_binary(train_classes)
    y1 = binary_train_classes[:,[0]]
    y2 = binary_train_classes[:,[1]]
    y3 = binary_train_classes[:,[2]]
    y4 = binary_train_classes[:,[3]]

    # Train on the four y vectors
    G = gram(train_features, ka, kb, kd)
    a1 = train_perceptron_kernel(G, y1, beta, limit)
    a2 = train_perceptron_kernel(G, y2, beta, limit)
    a3 = train_perceptron_kernel(G, y3, beta, limit)
    a4 = train_perceptron_kernel(G, y4, beta, limit)

    # Get binary predictions from the four perceptrons
    y_hat1 = test_perceptron_kernel(train_features, test_features, a1, ka, kb, kd)
    y_hat2 = test_perceptron_kernel(train_features, test_features, a2, ka, kb, kd)
    y_hat3 = test_perceptron_kernel(train_features, test_features, a3, ka, kb, kd)
    y_hat4 = test_perceptron_kernel(train_features, test_features, a4, ka, kb, kd)

    # Convert binary predictions back to decimal
    binary_pred_classes = np.hstack((y_hat1, y_hat2, y_hat3, y_hat4))
    pred_classes = convert_mnist_classes_to_integer(binary_pred_classes)

    # Create label for this evaluation
    label = str(train_features.shape[0] + test_features.shape[0])
    label += '_' + str(limit)
    label += '_' + str(beta)
    label += '_' + str(ka)
    label += '_' + str(kb)
    label += '_' + str(kd)

    # Calculate number correct
    correct = 0
    cm = np.zeros((10, 10))
    for i in range(test_classes.shape[0]):
        if (pred_classes[i][0] == test_classes[i][0]): correct += 1
        cm[int(pred_classes[i][0])][int(test_classes[i][0])] += 1
    print('Correct:', correct, '/', test_classes.shape[0], 'for', label)
    print(cm)

    np.savetxt('./data/confusion_kp_' + label + '.csv', cm, delimiter=',', fmt='%10.0f')

    return correct / test_classes.shape[0]


###############################################
#
# Run it all
#
###############################################

def main():

    ###############################################
    #
    # PART 1: VARIATIONS OF HYPERPARAMETERS ON A SMALL DATA SET
    #
    ###############################################

    # Load small data set for variation tests
```

```
    sample_limit = 1000
    classes, features = libsvm_scale_import('data/mnist.scale', limit = sample_limit)
    split = int(len(classes) * 0.70)
    train_classes = classes[:split]
    test_classes = classes[split:]
    train_features = features[:split]
    test_features = features[split:]
    print('training data =', train_features.shape, train_classes.shape)
    print('test_data =', test_features.shape, test_classes.shape)

    # Test decimal-binary-decimal conversion
    binary_train_classes = convert_mnist_classes_to_binary(train_classes)
    decimal_train_classes = convert_mnist_classes_to_integer(binary_train_classes)
    print(train_classes - decimal_train_classes)

    # Execute Neural Network testing
    print('\nNeural Network Variations')

    # Vary learning rate
    nn_lrs = np.array([0.01, 0.1, 1.0, 10.0])
    nn_lr_results = np.zeros(nn_lrs.shape)
    for i in range(nn_lrs.shape[0]):
        nn_lr_results[i] = mnist_neural_network(train_classes, test_classes, train_features,
                                                test_features, 100, nn_lrs[i], 100)
    plt.clf()
    plt.plot(nn_lrs, nn_lr_results, marker='.')
    plt.title('Neural Network: accuracy vs. learning rate for h=100, epochs=100')
    plt.xscale('log')
    plt.xlabel('learning rate')
    plt.ylabel('accuracy')
    plt.ylim(bottom = 0)
    plt.grid(True)
    plt.savefig('./plots/nn_accuracy_learning_rate.png', dpi = 600)

    # Vary size of hidden layer
    nn_hs = np.array([1, 10, 100, 1000])
    nn_h_results = np.zeros(nn_hs.shape)
    for i in range(nn_hs.shape[0]):
        nn_h_results[i] = mnist_neural_network(train_classes, test_classes, train_features,
                                               test_features, nn_hs[i], 1.0, 100)
    plt.clf()
    plt.plot(nn_hs, nn_h_results, marker='.')
    plt.title('Neural Network: accuracy vs. hidden layer size for lr=1.0, epochs=100')
    plt.xscale('log')
    plt.xlabel('hidden layer size')
    plt.ylabel('accuracy')
    plt.ylim(bottom = 0)
    plt.grid(True)
    plt.savefig('./plots/nn_accuracy_hsize.png', dpi = 600)

    # Vary number of epochs
    nn_epochs = np.array([10, 100, 1000, 10000])
    nn_epoch_results = np.zeros(nn_epochs.shape)
    for i in range(nn_epochs.shape[0]):
        nn_epoch_results[i] = mnist_neural_network(train_classes, test_classes,
    train_features,
                                                   test_features, 100, 1.0, nn_epochs[i])
    plt.clf()
    plt.plot(nn_epochs, nn_epoch_results, marker='.')
    plt.title('Neural Network: accuracy vs. epochs for h=100, lr=1.0')
    plt.xscale('log')
    plt.xlabel('number of epochs')
    plt.ylabel('accuracy')
    plt.ylim(bottom = 0)
    plt.grid(True)
    plt.savefig('./plots/nn_accuracy_epochs.png', dpi = 600)

    # Execute SVM testing
```

```python
    print('\nSupport Vector Machine Variations')

    # Vary number of steps
    svm_steps = np.array([10, 100, 1000, 10000, 100000, 1000000])
    svm_step_results = np.zeros(svm_steps.shape)
    for i in range(svm_steps.shape[0]):
        svm_step_results[i] = mnist_svm(train_classes, test_classes, train_features,
                                        test_features, svm_steps[i], 0.1)
    plt.clf()
    plt.plot(svm_steps, svm_step_results, marker='.')
    plt.title('SVM: accuracy vs. steps for lambda=0.1')
    plt.xscale('log')
    plt.xlabel('number of steps')
    plt.ylabel('accuracy')
    plt.ylim(bottom = 0)
    plt.grid(True)
    plt.savefig('./plots/svm_accuracy_step.png', dpi = 600)

    # Vary lambda
    svm_lams = np.array([1e-8, 1e-7, 1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1.0, 10.0])
    svm_lam_results = np.zeros(svm_lams.shape)
    for i in range(svm_lams.shape[0]):
        svm_lam_results[i] = mnist_svm(train_classes, test_classes, train_features,
                                       test_features, 100000, svm_lams[i])
    plt.clf()
    plt.plot(svm_lams, svm_lam_results, marker='.')
    plt.title('SVM: accuracy vs. lambda for steps=100000')
    plt.xscale('log')
    plt.xlabel('lambda')
    plt.ylabel('accuracy')
    plt.ylim(bottom = 0)
    plt.grid(True)
    plt.savefig('./plots/svm_accuracy_lambda.png', dpi = 600)

    # Execute Kernel Perceptron testing
    print('\nKernel Perceptron Variations')

    # Vary number of steps
    kp_steps = np.array([1, 10, 100, 1000, 10000, 100000, 1000000])
    kp_step_results = np.zeros(kp_steps.shape)
    for i in range(kp_steps.shape[0]):
        kp_step_results[i] = mnist_perceptron_kernel(train_classes, test_classes,
                                                     train_features, test_features,
                                                     kp_steps[i], 1,
                                                     0.0, 1.0, 2.0)
    plt.clf()
    plt.plot(kp_steps, kp_step_results, marker='.')
    plt.title('Perceptron Kernel: accuracy vs. steps for beta=1, a,b,d=0,1,2')
    plt.xscale('log')
    plt.xlabel('number of steps')
    plt.ylabel('accuracy')
    plt.ylim(bottom = 0)
    plt.grid(True)
    plt.savefig('./plots/kp_accuracy_step.png', dpi = 600)

    # Vary beta
    kp_betas = np.array([1e-8, 1e-7, 1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1.0, 10.0])
    kp_beta_results = np.zeros(kp_betas.shape)
    for i in range(kp_betas.shape[0]):
        kp_beta_results[i] = mnist_perceptron_kernel(train_classes, test_classes,
                                                     train_features, test_features,
                                                     1000, kp_betas[i],
                                                     0.0, 1.0, 2.0)
    plt.clf()
    plt.plot(kp_betas, kp_beta_results, marker='.')
    plt.title('Perceptron Kernel: accuracy vs. beta for steps=1000, a,b,d=0,1,2')
    plt.xscale('log')
    plt.xlabel('beta')
```

```python
    plt.ylabel('accuracy')
    plt.ylim(bottom = 0)
    plt.grid(True)
    plt.savefig('./plots/kp_accuracy_beta.png', dpi = 600)

    # Vary a
    kp_kas = np.array([1e-2, 1e-1, 1.0, 10.0, 100, 1000, 10000, 100000])
    kp_ka_results = np.zeros(kp_kas.shape)
    for i in range(kp_kas.shape[0]):
        kp_ka_results[i] = mnist_perceptron_kernel(train_classes, test_classes,
                                                   train_features, test_features,
                                                   1000, 1.0,
                                                   kp_kas[i], 1.0, 2.0)
    plt.clf()
    plt.plot(kp_kas, kp_ka_results, marker='.')
    plt.title('Perceptron Kernel: accuracy vs. kernel a for steps=1000, beta=1, b,d=1,2')
    plt.xscale('log')
    plt.xlabel('a (for polynomial kernel)')
    plt.ylabel('accuracy')
    plt.ylim(bottom = 0)
    plt.grid(True)
    plt.savefig('./plots/kp_accuracy_ka.png', dpi = 600)

    # Vary b
    kp_kbs = np.array([1e-2, 1e-1, 1.0, 10.0, 100, 1000, 10000, 100000])
    kp_kb_results = np.zeros(kp_kbs.shape)
    for i in range(kp_kbs.shape[0]):
        kp_kb_results[i] = mnist_perceptron_kernel(train_classes, test_classes,
                                                   train_features, test_features,
                                                   1000, 1.0,
                                                   100, kp_kbs[i], 2.0)
    plt.clf()
    plt.plot(kp_kbs, kp_kb_results, marker='.')
    plt.title('Perceptron Kernel: accuracy vs. kernel a for steps=1000, beta=1, a,d=100,2')
    plt.xscale('log')
    plt.xlabel('b (for polynomial kernel)')
    plt.ylabel('accuracy')
    plt.ylim(bottom = 0)
    plt.grid(True)
    plt.savefig('./plots/kp_accuracy_kb.png', dpi = 600)

    # Vary d
    kp_kds = np.array([1e-2, 1e-1, 1.0, 10.0, 100])
    kp_kd_results = np.zeros(kp_kds.shape)
    for i in range(kp_kds.shape[0]):
        kp_kd_results[i] = mnist_perceptron_kernel(train_classes, test_classes,
                                                   train_features, test_features,
                                                   1000, 1.0,
                                                   100, 1.0, kp_kds[i])
    plt.clf()
    plt.plot(kp_kds, kp_kd_results, marker='.')
    plt.title('Perceptron Kernel: accuracy vs. kernel a for steps=1000, beta=1, a,b=100,1')
    plt.xscale('log')
    plt.xlabel('d (for polynomial kernel)')
    plt.ylabel('accuracy')
    plt.ylim(bottom = 0)
    plt.grid(True)
    plt.savefig('./plots/kp_accuracy_kd.png', dpi = 600)

    # Vary d again (different and linear range)
    kp_kds = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
    kp_kd_results = np.zeros(kp_kds.shape)
    for i in range(kp_kds.shape[0]):
        kp_kd_results[i] = mnist_perceptron_kernel(train_classes, test_classes,
                                                   train_features, test_features,
                                                   1000, 1.0,
                                                   100, 1.0, kp_kds[i])
    plt.clf()
```

```python
    plt.plot(kp_kds, kp_kd_results, marker='.')
    plt.title('Perceptron Kernel: accuracy vs. kernel a for steps=1000, beta=1, a,b=100,1')
    plt.xlabel('d (for polynomial kernel)')
    plt.ylabel('accuracy')
    plt.ylim(bottom = 0)
    plt.grid(True)
    plt.savefig('./plots/kp_accuracy_kd_part2.png', dpi = 600)


    ##############################################
    #
    # PART 2: TEST ENTIRE DATA SET ON OPTIMAL-ISH PARAMETERS
    #
    ##############################################

    # Load all data
    classes, features = libsvm_scale_import('data/mnist.scale')
    split = int(len(classes) * 0.70)
    train_classes = classes[:split]
    test_classes = classes[split:]
    train_features = features[:split]
    test_features = features[split:]
    print('training data =', train_features.shape, train_classes.shape)
    print('test_data =', test_features.shape, test_classes.shape)

    mnist_neural_network(train_classes, test_classes, train_features,
                         test_features, 100, 1.0, 1000)  # this takes 12 hours

    mnist_svm(train_classes, test_classes, train_features, test_features, 6000000, 0.1)

    mnist_perceptron_kernel(train_classes, test_classes,
                            train_features, test_features,
                            10, 1.0, 100, 1.0, 3.0)

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

MiniProj2.py