```
#!/usr/bin/env python3

# Luke Weber, 11398889
# CptS 580, HW #1
# Created 02/08/2017

"""
Structured Perceptron with Randomized Greedy Search to
make inferences; running with varying joint-feature
representations, such as: unary, pairwise, third-order,
and fourth-order.

NOTE: Currently, we have a problem with this program not
converging, which is likely due to a small, language-
dependent error.
"""

import pyqtgraph as pg
import numpy as np
import random
import signal
import string
import math
import time
import copy
import sys
import re
import os

# Running on Windows?
if os.name != "posix":
    windows = True
    import winsound
else: windows = False

# Debugging
verbose = False          # Debug output control
sig = False              # See if a signal is already being handled

# Model
alphabet = set()         # Set of given dataset's alphabet of labels

# Phi-related
len_x = -1
phi_dimen = -1
pairwise_base_index = -1
triplet_base_index = -1
quadruplet_base_index = -1
pairs = []
triplets = []
quadruplets = []

# Scoring function (weights)
weights = []
weights_dir = "weights/"

def main():
    """
    Main: Driver function

    Data: Handwritten words and text-to-speech

    Setup (for both handwriting and text-to-speech mapping) problems:
        0. Establish model parameters
        1. Parse training and testing data
        2. Train structured perceptron on training data
        3. Test on testing data

    Assumptions:
        0. All data in form "000001010101010101..." "[label]"
    """

    global len_x, phi_dimen
```

```
    # Save weights on Ctrl-C
    signal.signal(signal.SIGINT, signal_handler)

    # Set non-truncated printing of numpy arrays
    np.set_printoptions(threshold = np.inf)

    # Perceptron training params
    R = 20
    eta = 0.01
    MAX = 100
    L = None

    # Raw training and testing data
    data_dir = "data/"
    raw_train_test = [(data_dir + "nettalk_stress_train.txt",
                       data_dir + "nettalk_stress_test.txt"),
                      (data_dir + "ocr_fold0_sm_train.txt",
                       data_dir + "ocr_fold0_sm_test.txt")]
    data_limit = len(raw_train_test)

    for raw_train, raw_test in raw_train_test[:data_limit]:

        print()
        print("Parsing training and testing data:")
        print("\t" + raw_train)
        print("\t" + raw_test)

        # Parse train & test data
        train, len_x, len_y = parse_data_file(raw_train)
        test, *_ = parse_data_file(raw_test)

        print("Done parsing!")
        print()

        # From data -> joint feature function; detect and
        # set phi_dimen dynamically
        phi = [phi_unary, phi_pairwise, phi_third_order, phi_fourth_order][3]
        phi_dimen = len(phi(train[0][0], train[0][1], len_x, len_y))

        # NOTE: We can either train for weights, or load them
        load_w = False
        if load_w: w = load_w()
        else: w = ospt(train, phi, R, eta, MAX, L)

        # Test
        ospt(test, phi, R, 1, 1, L, w)

        # Clear proverbial canvas
        reset_data_vars()

    return


"""
STRUCTURED PERCEPTRON

Methods immediately relevant to the concept of a perceptron
"""

def ospt(D, phi, R, eta, MAX, L, w = None):
    """
    Online structured perceptron training/testing:
        1. If weight vector w is not supplied, we're training;
        2. Else, we're testing
    """

    # See if we're training or testing
    training = w is None
    if training: work_word = "Train"
    else: work_word = "Test"

    # Check data limit
```

```
    if L is None: L = len(D)

    # Display heading
    print()
    print("<" + work_word + "> ", end = "")
    print("Structured Perceptron:")
    print()
    print("\tData length (with limitation) = " + str(L))
    print("\tNumber of restarts = " + str(R))
    print("\tLearning rate = " + str(eta))
    print("\tMax iteration count = " + str(MAX))
    print("\tNumber of joint-features = " + str(phi_dimen))
    print("\tAlphabet length = " + str(len(alphabet)))
    print()

    # Record model's progress w.r.t. accuracy (and iteration improvment)
    it_improvement = np.zeros((L))
    acc_progress = []
    if training: pw = pg.plot()

    # Setup weights of scoring function to 0, if weight vector is
    # not supplied
    if training: w = np.zeros((phi_dimen))

    # Iterate until max iterations or convergence
    for it in range(MAX):

        # Time each iteration
        it_start = time.clock()

        print("[Iteration " + str(it) + "]\n")

        # Essential iteration-related vars
        train_num = 0
        num_mistakes = 0
        num_correct = 0

        # Go through training examples
        for x, y in D[:L]:

            # Skip empty data points
            if len(x) < 1: continue

            # Predict
            y_hat = rgs(x, phi, w, R, len(y))
            num_right_chars = len(y) - list_diff(y_hat, y)

            # If error, update weights
            instance_str = (work_word + " instance " + str(it)
                            + "." + str(train_num))
            if y_hat != y:
                instance_str = ("\t[-]\t" + instance_str + "\t("
                                + str(num_right_chars) + "/" + str(len(y)) + ")")
                if training:
                    # TODO: Uncomment the weight update!
                    w = np.add(w, np.dot(eta, (np.subtract(phi(x, y),
                                                           phi(x, y_hat)))))
                    # HACK: Setting "weights" each time might be too much
                    weights = w
                    num_mistakes += 1
            else:
                instance_str = ("\t[+]\t" + instance_str + "\t(" + str(len(y))
                                + "/" + str(len(y)) + ")")
                num_correct += 1

            instance_str += ("\t[" + str(num_correct) + "/"
                             + str(train_num + 1) + "]")

            # Measure iteration improvment (compared to last)
            if (it > 0):
                improvement = num_right_chars - it_improvement[train_num]
                if improvement != 0:
                    instance_str += "\t" + give_sign(int(improvement))
```

```
            it_improvement[train_num] = num_right_chars

            # Print instance details and update training number
            print(instance_str)
            train_num += 1

        # Determine accuracy
        accuracy = num_correct / (num_correct + num_mistakes)
        acc_progress.append(accuracy)

        # Plot accuracy timeline
        if len(acc_progress) > 1:
            pw.plot(acc_progress, clear = True)
            pg.QtGui.QApplication.processEvents()

        # Report iteration stats
        print()
        print("\t| Accuracy = " + str(accuracy * 100) + "%")
        print("\t| Number correct = " + str(num_correct))
        print("\t| Number of mistakes = " + str(num_mistakes))
        print("\t| Time = ~" + str(round((time.clock() - it_start) / 60))
              + " min")
        print()

        # Check for convergence
        if len(acc_progress) > 1:
            if acc_progress[-1] == acc_progress[-2]:
                print("Model has converged:")
                print("\tAccuracy = " + str(accuracy * 100) + "%")
                print("\tIteration = " + str(it))
                break

    # Return and save weights!
    if training: return save_w(w)
    return

def rgs(x, phi, w, R, len_y):
    """
    Randomized Greedy Search (RGS) inference:
    Try and use the current weights to arrive at the correct label;
    we will always return our best guess
    """

    for i in range(R):

        # Initialize best scoring output randomly
        y_hat = get_random_y(len_y)

        # Until convergence
        while True:

            # Get max char
            y_max = get_max_one_char(w, phi, x, y_hat)
            if y_max == y_hat: break
            y_hat = y_max

    return y_hat

def phi_unary(x, y, len_x = None, len_y = None):
    """ Unary joint-feature function """

    if len_x is None: dimen = phi_dimen
    else: dimen = len_x * len_y
    vect = np.zeros((dimen))

    for i in range(len(x)):

        x_i = x[i]
        y_i = y[i]

        # Sorting keeps consistency of indices with respect to
        # all prior and following phi(x, y) vectors
        alpha_list = list(alphabet)
```

```
        alpha_list.sort()
        index = alpha_list.index(y_i)
        x_vect = np.array(x_i)

        # Manual insertion of x into standard vector
        # NOTE: Holy fuck, had "= x_vect[j]" before, not "+="
        y_target = len(x_i) * index
        for j in range(len(x_i)): vect[j + y_target] += x_vect[j]

    return vect

def phi_pairwise(x, y, len_x = None, len_y = None):
    """
    Pairwise joint-feature function:
        0. Do unary features
        1. Capture all two-char permutations
        2. Assign these permuations consistent indices
        3. Count frequencies of each permuation and update vector
           at that index
    """

    global pairwise_base_index, pairs

    # NOTE: len_y = len(alphabet)
    # Initial setting of phi dimensions
    if len_x is None: dimen = phi_dimen
    else:
        pairwise_base_index = len_x * len_y
        dimen = (len_x * len_y) + (len_y ** 2)

    vect = np.zeros((dimen))
    alpha_list = list(alphabet)
    alpha_list.sort()

    # (One-time) Generate pair-index object
    if len(pairs) == 0:
        for a in alpha_list:
            for b in alpha_list:
                p = a + b
                pairs.append(p)

    # Unary features
    for i in range(len(x)):

        x_i = x[i]
        y_i = y[i]

        # Unary features
        index = alpha_list.index(y_i)
        x_vect = np.array(x_i)
        y_target = len(x_i) * index
        for j in range(len(x_i)): vect[j + y_target] += x_vect[j]

    # Pairwise features
    for i in range(len(y) - 1):

        # Get pair index
        a = y[i]
        b = y[i + 1]
        p = a + b
        comb_index = pairs.index(p)
        vect_index = pairwise_base_index + comb_index

        # Update occurace of pair
        vect[vect_index] += 1

    return vect

def phi_third_order(x, y, len_x = None, len_y = None):
    """
    Third-order joint-feature function:
        0. Do unary features
        1. Do pairwise features
```

```
        2. Capture all three-char permutations
        3. Assign these permuations consistent indices
        4. Count frequencies of each permuation and update vector
           at that index
    """

    global pairwise_base_index, triplet_base_index, pairs, triplets

    # NOTE: len_y = len(alphabet)
    # Initial setting of phi dimensions
    if len_x is None: dimen = phi_dimen
    else:
        pairwise_base_index = len_x * len_y
        triplet_base_index = pairwise_base_index + (len_y ** 2)
        dimen = triplet_base_index + (len_y ** 3)

    vect = np.zeros((dimen))
    alpha_list = list(alphabet)
    alpha_list.sort()

    # (One-time) Generate pair and triplet lists
    if len(triplets) == 0:
        for a in alpha_list:
            for b in alpha_list:
                # Grab pair
                p = a + b
                pairs.append(p)

                for c in alpha_list:
                    # Grab triplet
                    t = a + b + c
                    triplets.append(t)

    # Unary features
    for i in range(len(x)):

        x_i = x[i]
        y_i = y[i]

        # Unary features
        index = alpha_list.index(y_i)
        x_vect = np.array(x_i)
        y_target = len(x_i) * index
        for j in range(len(x_i)): vect[j + y_target] += x_vect[j]

    # Pairwise features
    for i in range(len(y) - 1):

        # Get pair index
        a = y[i]
        b = y[i + 1]
        p = a + b
        comb_index = pairs.index(p)
        vect_index = pairwise_base_index + comb_index

        # Update occurace of pair
        #print(p, "occurs at", vect_index)
        vect[vect_index] += 1

    # Third-order features
    for i in range(len(y) - 2):

        # Get pair index
        a = y[i]
        b = y[i + 1]
        c = y[i + 2]
        t = a + b + c
        comb_index = triplets.index(t)
        vect_index = triplet_base_index + comb_index

        # Update occurace of pair
        #print(t, "occurs at", vect_index)
        vect[vect_index] += 1
```

```
    return vect

def phi_fourth_order(x, y, len_x = None, len_y = None):
    """
    Fourth-order joint-feature function:
        0. Do unary features
        1. Do pairwise features
        2. Do third-order features
        3. Capture all four-char permutations
        4. Assign these permuations consistent indices
        5. Count frequencies of each permuation and update vector
           at that index
    """

    global pairwise_base_index, triplet_base_index, quadruplet_base_index
    global pairs, triplets

    # NOTE: len_y = len(alphabet)
    # Initial setting of phi dimensions
    if len_x is None: dimen = phi_dimen
    else:
        pairwise_base_index = len_x * len_y
        triplet_base_index = pairwise_base_index + (len_y ** 2)
        quadruplet_base_index = triplet_base_index + (len_y ** 3)
        dimen = quadruplet_base_index + (len_y ** 4)

    vect = np.zeros((dimen))
    alpha_list = list(alphabet)
    alpha_list.sort()

    # (One-time) Generate pair, triplet, and quadruplet lists
    if len(quadruplets) == 0:
        for a in alpha_list:
            for b in alpha_list:
                # Grab pair
                p = a + b
                pairs.append(p)

                for c in alpha_list:
                    # Grab triplet
                    t = a + b + c
                    triplets.append(t)

                    for d in alpha_list:
                        # Grab quadruplet
                        q = a + b + c + d
                        quadruplets.append(q)

    # Unary features
    for i in range(len(x)):

        x_i = x[i]
        y_i = y[i]

        # Unary features
        index = alpha_list.index(y_i)
        x_vect = np.array(x_i)
        y_target = len(x_i) * index
        for j in range(len(x_i)): vect[j + y_target] += x_vect[j]

    # Pairwise features
    for i in range(len(y) - 1):

        # Get pair index
        a = y[i]
        b = y[i + 1]
        p = a + b
        comb_index = pairs.index(p)
        vect_index = pairwise_base_index + comb_index

        # Update occurace of pair
        #print(p, "occurs at", vect_index)
```

```
            vect[vect_index] += 1

        # Third-order features
        for i in range(len(y) - 2):

            # Get pair index
            a = y[i]
            b = y[i + 1]
            c = y[i + 2]
            t = a + b + c
            comb_index = triplets.index(t)
            vect_index = triplet_base_index + comb_index

            # Update occurace of pair
            #print(t, "occurs at", vect_index)
            vect[vect_index] += 1

        # Fourth-order features
        for i in range(len(y) - 3):

            # Get pair index
            a = y[i]
            b = y[i + 1]
            c = y[i + 2]
            d = y[i + 3]
            q = a + b + c + d
            comb_index = quadruplets.index(q)
            vect_index = quadruplet_base_index + comb_index

            # Update occurace of pair
            #print(q, "occurs at", vect_index)
            vect[vect_index] += 1

        return vect

    def get_score(w, phi, x, y_hat):
        return np.dot(w, phi(x, y_hat))

    def get_random_y(len_y):
        """ Return random array of alphabet characters of given length """

        rand_y = []

        # If no length passed
        if len_y == None:
            min_word_len = 2
            max_word_len = 6
            rand_word_len = math.floor(random.uniform(min_word_len,
                                                      max_word_len + 1))
        else: rand_word_len = len_y

        for i in range(rand_word_len):
            rand_char = random.sample(alphabet, 1)[0]
            rand_y.append(rand_char)

        return rand_y

    def get_max_one_char(w, phi, x, y_hat):
        """
        Make one-character changes to y_hat, finding which
        single change produces the best score; we return
        that resultant y_max
        """

        # Initialize variables to max
        s_max = get_score(w, phi, x, y_hat)
        y_max = y_hat

        for i in range(len(y_hat)):

            # Copy of y_hat to play with
            y_temp = copy.deepcopy(y_hat)
```

```
        # Go through a-z at i-th index
        for c in alphabet:

            # Get score of 1-char change
            y_temp[i] = c
            s_new = get_score(w, phi, x, y_temp)

            # Capture highest-scoring change
            if s_new > s_max:
                s_max = s_new
                y_max = y_temp

    return y_max


"""
UTILITY FUNCTIONS

Methods not directly relevant to the concept of the structured
percpetron, but more to the maintainance and assistance of
more basic computation in program
"""

def reset_data_vars():
    global alphabet
    alphabet = set()

def dprint(s):
    if verbose: print(s)

def setify(num):
    """
    Set-ify that number (i.e. remove trailing zeros, as is
    automatically done in the alphabet set) for consistency
    """

    return list(set([num]))[0]

def beep():
    """ Give absent-minded programmer a notification """

    freq = 700  # Hz
    dur = 1000  # ms
    for i in range(10):
        winsound.Beep(freq, dur)
        freq += 100
    winsound.Beep(freq * 2, dur * 4) # RRRREEEEEEEEEE!

def list_diff(a, b):
    """ Show's degree of difference of list a from b """

    if len(a) != len(b):
        raise ValueError("Lists of different length.")
    return sum(i != j for i, j in zip(a, b))

def give_sign(n):
    if n < 0: return str(n) # Number will already be negative
    if n > 0: return "+" + str(n)
    return str(n) # We'll say 0 has no sign

def save_w(w):
    """ Serialize the weights of perceptron into local file """

    # TODO: Encode filenames with the following:
    #   0. Number of iterations (until stopped or converged)
    #   1. Degree of phi
    #   2. Type of data (e.g. nettalk or ocr)
    #   3. Version (i.e. the number of weight files)

    print("-" * 35)
    print("Saving weights to local file...")

    # Get user's attention
```

```
    if windows: beep()

    # Ask if they want to save
    if (input("Proceed? (y/n): ").strip() == "n"): exit(0)

    # List weight files
    files = [f for f in os.listdir(weights_dir)
         if os.path.isfile(f)
         and f.split(".")[-1] == "npy"]
    print("\tCurrent weight files (to avoid conflicts):", files)

    # Enter filename and save
    w_file_name = input("\tPlease enter filename: ")
    np.save(weights_dir + w_file_name, w)
    print("Saved!")
    print("-" * 35)

    return w

def load_w():
    """ Deserialize weights of perceptron from local file """

    print("-" * 35)
    print("Loading weights from local file:")

    # Show available weight files
    files = [f for f in os.listdir(weights_dir)
             if os.path.isfile(f)
             and f.split(".")[-1] == "npy"]
    print("\tAvailable weight files:", files)

    # User selects file (looping until valid file) -> we load
    w_file_name = ""
    while w_file_name not in files:
        w_file_name = input("\tPlease enter filename: ")
    w = np.load(weights_dir + w_file_name)
    print("File loaded!")
    print("-" * 35)

    return w

def signal_handler(signal, frame):
    """ Save weights on Ctrl+C """

    global sig

    # Check for the double Ctrl+C, which means exit
    if sig is True: exit(0)

    sig = True
    print('[Ctrl+C pressed]')
    save_w(weights)
    sig = False
    exit(0)

def parse_data_file(file_loc):
    """ Parse raw data into form of [(x_0, y_0), ..., (x_n, y_n)] """

    global alphabet

    data_arr = []
    len_x_vect = -1

    with open(file_loc) as f:

        x = []
        y = []

        # Take collection of examples (e.g. collection of pairs of
        # character data x_i matched with the actual character
        # class y_i) and push into data array
        for line in f:
```

```
            # Push single collection of examples onto data array
            # when newline is encountered
            if not line.strip():
                data_arr.append((x, y))
                x = []
                y = []
                continue

            # Parse one example (x_i, y_i)
            l_toks = line.split("\t")
            x_i_str = l_toks[1][2:] # Trim leading "im" tag
            x_i = [int(c) for c in x_i_str]
            y_i = setify(l_toks[2])

            # Collect length of all x_i's
            if len_x_vect < 0: len_x_vect = len(x_i)

            # Take note of all possible labels (i.e. the set Y)
            # NOTE: Listifying y_i is necessary to keep leading
            # zeroes, e.g. maintaining '04' rather than '4'
            alphabet.update([y_i])

            # Add single example to collection
            x.append(x_i)
            y.append(y_i)

    num_labels = len(alphabet)

    return data_arr, len_x_vect, num_labels

# Party = started
main()
```