Feedback report

Student: acb19rx. Total Score: 42/50

Feature	Classifier	Correction	Performance	Code	Total
8/10	10/10	7/10	9/10	8/10	42/50

Feature Extraction (Max 200 Words)

[In both training and testing phases, I use Principal Component Analysis (PCA) algorithm to conduct the feature dimension reduction which is simple but very effective. In PCA, I try different values of reduced dimensions (n = 10, 11, ... 20) and pick 10 most distinguishable features from them by comparing their divergence. Extensive experiments show that best results can be obtained by setting n = 12 and 13. To save the computational time, I have n = 12 in my model settings.]

- Yes. PCA is a sensible approach for the feature design.
- Credit for combining PCA with feature selection.
- It would have been worth including some of the results of the extensive experiments in the report, and using them to justify the final choices.

Score: 8/10 (Features)

Classifier (Max 200 Words)

[OCR is a multi-classification task by nature, I implement two classification models using numpy, i.e., k-Nearest Neighbours (KNN) and Support Vector Machines (SVM). KNN is simple and no training is required. It compares the extracted feature of a test sample with that of all training samples, then uses the majority voting of its top-k similar training sample's labels as the classified label. As the setting of different k has large impact on the classification results, I conduct numerous tests to find a series of appropriate k values for images with different noise levels. k is set to a small value if the background noise is low while k is set to a bigger value when much more noise is artificially added to the test images.
SVM is a supervised machine learning model with the aim to find a hyperplane in an N-dimensional space (N features) that distinctly classifies the data points into two or multiple categories. SVM is more complicated and requires training. The training process is very computational expensive which takes several hours for training the training data. The experiments show in small sample dataset, the classification accuracy can achieve around 80%, but for the large OCR dataset, it is hard for the model to get converged.]

- Excellent work.
- Credit for trying alternative approaches and implementing them yourself.
- Setting k according to the noise level is also an excellent idea. A few students tried this and all got results in the top 10%.

Score: 10/10 (Classifier)

Error Correction (Max 200 Words)

[KNN classifier labels every test sample k most likely labels, say, if a word has a length of m, there are k^m combinations of characters. In error correction, the objective is to find the most probable combination of characters then check whether it is contained in the dictionary. Concretely, I use the Dijkstra algorithm, a popular algorithm to search the shortest path. If the word with the shortest path does not appear in the dictionary, I use the most similar word in the dictionary to replace it. Experiments show it reaches 5%-10% performance gain, especially for high noise images, after the error correction module is equipped.]

- Well done for completing this challenging section.
- It is not necessary to use dynamic programming because the OCR system can only make substitutions so the two strings being compared are already aligned.

Score: 7/10 (Correction)

Other information (Optional, Max 100 words)

```
[OCR Model Architecture:
-----------------        ------------------        ------------------
|                |      |                 |      |                 |
Input  |   Dimension    |      |   Multi-Class    |      | Error Correction |      Recognized
=====> |    Reduction    | ====> |  Classification  | ====> |    (Dijkstra       | ====>  Characters
|      (PCA)       |      |  (KNN & SVM)    |      |  Shortest Path)  |
|-----------------|      |------------------|      |------------------|]
```

Performance

[The percentage errors (to 1 decimal place) using KNN classifier for the development data are as follows:
- Page 1: score = 97.7% correct
- Page 2: score = 98.2% correct
- Page 3: score = 92.0% correct
- Page 4: score = 77.3% correct
- Page 5: score = 65.0% correct
- Page 6: score = 53.0% correct]

- Scores on test pages:
    - Page 1: 97.9%
    - Page 2: 95.6%
    - Page 3: 83.4%
    - Page 4: 69.3%
    - Page 5: 58.5%
    - Page 6: 46.8%
- Average correct = 75.2% (93.6% percentile)

Excellent result. Among top 10%.

- No comment

Score: 9/10 (Performance)

Code

```
import numpy as np
import scipy.linalg
import random
import operator


import utils.utils as utils

if_print_info = False
pca_reduced_dimensions = 12 # dimensionality reduction parameter in pca function
feature_selection_dimensions = 10 # dimension of selected features
k_nearest_neighbour = 5 # k in nn classifier
n_nearest_neighbour = 3 # number of outputs of knn
error_correction_cost_limit = 10
error_correction_nodes_limit = 800
average_image_gray_value_list = [] #average image gray value
classification_method = 'knn'   # we implement knn and svm classifiers


#original
def get_bounding_box_size(images):
    """Compute bounding box size given list of images."""
    height = max(image.shape[0] for image in images)
    width = max(image.shape[1] for image in images)
    return height, width

#original
def images_to_feature_vectors(images, bbox_size=None):
    """
    Reformat characters into feature vectors.
    Parameters
    ----------
    images : 2D-arrays
        a list of images
    bbox_size : int, optional
        an optional fixed bounding box size for each image
    Returns
    -------
    fvectors : array
        a matrix in which each row is a fixed length feature vector
        corresponding to the image.abs
    """
    print(" ### Enter the Image to Feature Vectors Module ... ... ...")
    # If no bounding box size is supplied then compute a suitable
    # bounding box by examining sizes of the supplied images.
    if bbox_size is None:
        bbox_size = get_bounding_box_size(images)

    bbox_h, bbox_w = bbox_size
    nfeatures = bbox_h * bbox_w
    fvectors = np.empty((len(images), nfeatures))
```

```
    for i, image in enumerate(images):
        padded_image = np.ones(bbox_size) * 255
        h, w = image.shape
        h = min(h, bbox_h)
        w = min(w, bbox_w)
        padded_image[0:h, 0:w] = image[0:h, 0:w]
        fvectors[i, :] = padded_image.reshape(1, nfeatures)
    return fvectors

#original but already fixed
def process_training_data(train_page_names):
    """
    Perform the training stage and return results in a dictionary.

    Parameters
    ----------
    train_page_names : list
        training page names

    Returns
    -------
    model_data : dict
        stored training data
    """
    print(" ### Enter the Process Training Data Module ... ... ...")

    if if_print_info:
        print(' *** *** Load word dictionary ...')
    word_list = list()
    with open('word_dictionary.txt','r') as f:
        for line in f:
            word_list.append(line.strip('\n'))
    word_list.extend(['i\'ll', 'i\'m', 'i\'d', 'you\'re', 'don\'t', 'didn\'t', 'haven\'t', 'who\'s', 'there\'s', 'it\'s'])

    if if_print_info:
        print(' *** *** Reading character data of all pages ...')
    images_train = []
    labels_train = []
    for page_name in train_page_names:
        images_train = utils.load_char_images(page_name, images_train)
        labels_train = utils.load_labels(page_name, labels_train)
    labels_train = np.array(labels_train)

    if if_print_info:
        print(' *** *** Extracting features from training data ...')
    bbox_size = get_bounding_box_size(images_train)
    fvectors_train_full = images_to_feature_vectors(images_train, bbox_size)

    model_data = dict()
    model_data['word_list'] = word_list
    model_data['labels_train'] = labels_train.tolist()
    model_data['bbox_size'] = bbox_size
```

```
    model_data['training_phase'] = True
    model_data['testing_phase'] = False

    if if_print_info:
        print(' *** *** Perform dimension reduction ...')
    feature_vectors_train = reduce_dimensions(fvectors_train_full, model_data)
    model_data['fvectors_train'] = feature_vectors_train.tolist()

    if classification_method == 'svm':
        numerical_labels_train, char_label_set_train, number_label_set_train =
convert_training_char_label_to_numbers(labels_train)
        numerical_labels_train = np.array(numerical_labels_train)
        num_of_classes = len(number_label_set_train)
        onehot_labels_train = change_seq_label_to_onehot(numerical_labels_train, num_of_classes)
        model_dict = train_svm_model(feature_vectors_train, onehot_labels_train, num_of_classes)
        model_data['numerical_labels_train'] = numerical_labels_train.tolist()
        model_data['char_label_set_train'] = char_label_set_train
        model_data['number_label_set_train'] = number_label_set_train
        model_data['model_dict'] = model_dict

    return model_data

#original but already fixed
def load_test_page(page_name, model):
    """
    Parameters
    ----------
    page_name : name of page file
    model : dictionary
        storing data passed from training stage

    Returns
    -------
    feature_vectors_test_reduced : each character as a 10-d feature
    vector with the vectors stored as rows of a matrix.

    """
    print(" ### Enter Load Test Page Module ... ... ...")

    bbox_size = model['bbox_size']
    images_test = utils.load_char_images(page_name)
    calculate_average_gray_value(images_test)
    feature_vectors_test = images_to_feature_vectors(images_test, bbox_size)
    feature_vectors_test_reduced = reduce_dimensions(feature_vectors_test, model)
    return feature_vectors_test_reduced

#already fixed
def calculate_average_gray_value(images):
    """
    Caculate the average gray value in images for nosiy detection

    Parameters
```

```
    ----------
    images : array
        images in data file

    Returns
    -------
    None.
    """
    print(" ### Enter the Calculate Average Image Gray Value Module ... ... ...")
    gray_mean = 0
    pixel_count = 0
    for i in range(len(images)):
        gray_mean = gray_mean + np.sum(images[i])
        pixel_count = pixel_count + images[i].shape[0] * images[i].shape[1]
    gray_mean = gray_mean / pixel_count
    normalized_gray_mean = gray_mean / 255
    average_image_gray_value_list.append(normalized_gray_mean)


#already fixed
def reduce_dimensions(feature_vectors_full, model):
    """
    Reduce 10 dimensions in feature vector

    Parameters
    ----------
    feature_vectors_full : array
        feature vector matrix in array
    model : dictionary
        which stored the model training outputs

    Returns
    -------
    array
        trained feature vectors
    """
    print(" ### Enter the Dimension Reduction Module ... ... ...")

    if model['training_phase'] is True and model['testing_phase'] is False:
        print(' *** *** Dimension Reduction for the Training Phase')
        eigenvectors = pca(feature_vectors_full, pca_reduced_dimensions)
        feature_vectors_train = np.dot((feature_vectors_full - np.mean(feature_vectors_full)), eigenvectors)
        model['eigenvectors'] = eigenvectors.tolist()

        # Feature selection if required
        if(feature_selection_dimensions < pca_reduced_dimensions):
            print(' *** *** Select Useful Features...')
            labels_array_train = np.array(model['labels_train'])
            selected_features = select_features(feature_vectors_train, labels_array_train,
feature_selection_dimensions)
            feature_vectors_train = feature_vectors_train[:, selected_features]
            model['selected_features'] = selected_features.tolist()
```

```
        model['training_phase'] = False
        model['testing_phase'] = True
        return feature_vectors_train

    if model['training_phase'] is False and model['testing_phase'] is True:
        print(' *** *** Dimension Reduction for the Testing Phase!')
        eigenvectors = np.array(model['eigenvectors'])
        feature_vectors_test = np.dot((feature_vectors_full - np.mean(feature_vectors_full)), eigenvectors)

        if('selected_features' in model.keys()):
            if if_print_info:
                print(' *** *** Select Useful Features...')
            selected_features = np.array(model['selected_features'])
            feature_vectors_test = feature_vectors_test[:, selected_features]
        return feature_vectors_test

    else:
        print(' *** *** No Dimension Reduction is performed!')
        return feature_vectors_full

#from lab
def pca(X, reduced_dimensions):
    """
    PCA constructs features that are the linear combination of
    the original feature values that best preserves the spread of the data

    Parameters
    ----------
    X : array
        feature vector
    reduced_dimensions : int
        reduced dimension number

    Returns
    -------
    v : array
        feature vector after reduced its dimensions

    """
    print(" ### Enter the PCA Module ... ... ...")

    covx = np.cov(X, rowvar=0)
    N = covx.shape[0]
    w, v = scipy.linalg.eigh(covx, eigvals=(N-reduced_dimensions, N-1))
    v = np.fliplr(v)
    return v

def select_features(feature_vectors, labels, dimensions):
    """
    summing divergences over pairs of classes.
    rank the PCA features according to their 1-D divergence and pick the best 10
```

```
    """
    nfeatures = feature_vectors.shape[1]
    classes_label_list = list(set(labels))
    classes_label = np.array(classes_label_list)
    classes_count = classes_label.shape[0]
    divergences = np.zeros(nfeatures)

    for i in range(classes_count):
        if if_print_info:
            print('\r', 'Selecting features...', i+1, '/', classes_count, end='')
        for j in range(i+1, classes_count):
            data1 = feature_vectors[labels == classes_label[i], :]
            data2 = feature_vectors[labels == classes_label[j], :]
            if (not (data1.shape[0] < 2)) and (not (data2.shape[0] < 2)):
                mean1 = np.mean(data1, axis=0)
                mean2 = np.mean(data2, axis=0)
                var1 = np.var(data1, axis=0)
                var2 = np.var(data2, axis=0)
                combine_data = 0.5 * (var1 / var2 + var2 / var1 - 2) + 0.5 * ( mean1 - mean2 ) * (mean1 - mean2) *
(1.0 / var1 + 1.0 / var2)
                divergences = combine_data + divergences

    sorted_indexes = np.argsort(-divergences)
    features = sorted_indexes[0:dimensions]

    return features

#already fixed
def classify_page(page, model):
    """
    classify each page

    Parameters
    ----------
    page : array
        matrix, each row is a feature vector to be classified
    model : dictionary
        which stores the output of the training stage

    Returns
    -------
    label : array
        labels which are classified by knn

    """
    print(" ### Enter the Classify Page Module ... ... ...")

    if classification_method == 'knn':
        avg_img_gray_value = average_image_gray_value_list.pop(0)
        k_nearest_neighbour = int(round(-32.321 * 100*avg_img_gray_value + 2058.1))
        if k_nearest_neighbour <= 1:
```

```
        k_nearest_neighbour = 1

    elif k_nearest_neighbour > 200:
        k_nearest_neighbour = 200

    if if_print_info:
        print('k =', k_nearest_neighbour)

    feature_vectors_train = np.array(model['fvectors_train'])
    labels_array_train = np.array(model['labels_train'])
    pred_label_list = knn_classifier(feature_vectors_train, labels_array_train, page, k_nearest_neighbour,
n_nearest_neighbour)
    #pred_label_list = knn_classifier(np.array(model['fvectors_train']), np.array(model['labels_train']), page,
k_nearest_neighbour, n_nearest_neighbour)

  if classification_method == 'svm':
    feature_vectors_train = np.array(model['fvectors_train'])
    labels_array_train = np.array(model['labels_train'])
    pred_label_list = svm_classification(feature_vectors_train, page, model)

  return pred_label_list

def knn_classifier(train, train_labels, test, k, n):
  """
  It compares the extracted feature of a test sample with that of all
  training samples, then uses the majority voting of its
  top-k similar training sample's labels as the classified label.
  As the setting of different k has large impact on the classification results,
  I conduct numerous tests to find a series of appropriate k values for
  images with different noise levels. k is set to a small value
  if the background noise is low while k is set to a bigger value
  when much more noise is artificially added to the test images.
  """

  print(" ### Enter the kNN Classifier Module ... ... ...")

  # Super compact implementation of nearest neighbour
  AB = np.dot(test, train.transpose())
  mod_A = np.sqrt(np.sum(test * test, axis=1))
  mod_B = np.sqrt(np.sum(train * train, axis=1))
  cos_distance = AB / np.outer(mod_A, mod_B.transpose());

  if k == 1:
    nearest_cos_dist = np.argmax(cos_distance, axis=1)
    pred_label = train_labels[nearest_cos_dist]
    return pred_label
  else:
    k_nearest_cos_dist = np.argsort(-cos_distance, axis=1)[:, :k]
    k_labels = train_labels[k_nearest_cos_dist]
    all_classes_label_set = np.array(list(set(train_labels)))
    pred_labels = []
```

```
    for i in range(k_nearest_cos_dist.shape[0]):
        if if_print_info:
            print('\r', i, '/', k_nearest_cos_dist.shape[0], end='')
        label_sum = np.zeros(all_classes_label_set.shape[0])
        for j in range(k_nearest_cos_dist.shape[1]):
            label_index = np.argwhere(all_classes_label_set == k_labels[i, j])
            label_sum[label_index] += cos_distance[i, k_nearest_cos_dist[i, j]]
        pred_labels.append(all_classes_label_set[np.argsort(-label_sum)[:n]])
    if if_print_info:
        print('\r', k_nearest_cos_dist.shape[0], '/', k_nearest_cos_dist.shape[0], end='')
    return np.array(pred_labels)

def correct_errors(page, labels, bboxes, model):
    """
    KNN classifier labels every test sample k most likely labels, say,
    if a word has a length of m, there are k^m combinations of characters.
    In error correction, the objective is to find the most probable
    combination of characters then check whether it is
    contained in the dictionary.
    """
    print(" ### Enter Correct Errors Module ... ... ... ")

    if(len(labels.shape) == 1):
        print('Error correction skipped (k=1)')
        return labels

    print('Processing error correction...')
    # Make a word set
    word_dict = set(model['word_list'])

    # Ready for error correction
    total_length = labels.shape[0]
    start_index = 0
    output_labels = []

    # Try to split words based on border boxes
    for i in range(bboxes.shape[0]):
        if(i == total_length-1):
            dijkstra_correct(labels, start_index, i, word_dict, output_labels)
            start_index = i+1
        elif(abs(bboxes[i+1][0] - bboxes[i][2]) > 6):
            dijkstra_correct(labels, start_index, i, word_dict, output_labels)
            start_index = i+1
        if if_print_info:
            print('\r', start_index, '/', total_length, end='')

    output_labels_array = np.array(output_labels)
    return output_labels_array


class Node:
    def __init__(self, pos, path, cost):
```

```
        self.pos = pos
        self.path = path
        self.cost = cost

def dijkstra_correct(labels, start, end, wordset, output_labels):
    """
    Concretely, I use the Dijkstra algorithm,
    a popular algorithm to search the shortest path.
    If the word with the shortest path does not appear in the dictionary,
    I use the most similar word in the dictionary to replace it.
    Experiments show it reaches 5%-10% performance gain,
    especially for high noise images, after the error correction module is equipped.
    """
    predictions = labels.shape[1]
    word_length = end - start
    node_list = []
    closed_count = 0

    init_node = Node(-1, [], 0)
    node_list.append(init_node)

    while len(node_list) > 0:
        node = node_list[0]

        if((node.cost > error_correction_cost_limit) or (closed_count > error_correction_nodes_limit)):
            # Over limits, give up
            output_labels.extend(labels[start:end+1, 0])
            return

        if(node.pos == word_length):
            # Length matched, verify
            predict_word = ''.join(node.path)
            if(predict_word.replace('\\', '').strip(',.?!') in wordset):
                # Matched, success
                output_labels.extend(node.path)
                return

        # Finish the node, find successor
        node_list.remove(node)
        closed_count += 1
        next_pos = node.pos+1
        if(next_pos <= word_length):
            for i in range(predictions):
                insert_node(node_list, Node(next_pos, node.path+[labels[start+next_pos, i]], node.cost+i+1))

    output_labels.extend(labels[start:end+1, 0])
    return

def insert_node(node_list, node):
    for j in range(len(node_list)):
        if node.cost < node_list[j].cost:
            node_list.insert(j, node)
```

```
        return
    node_list.append(node)




###SVM Classifier###


def train_svm_model(X_train, y_train, num_of_classes):
    print(" ### Enter Train SVM Model Module ... ... ...")
    model_dict = {}
    for i in range (num_of_classes):
        y_train_ovr = y_train[:, i]
        model_dict[str(i)] = svm_model(X_train, y_train_ovr, i)
    return model_dict


def convert_training_char_label_to_numbers(label_list):
    print(" ### Enter Convert Training Char Label to Number Label Module ... ... ...")
    char_label_set = list(set(label_list))
    char_label_set.sort()
    num_of_char_labels = len(char_label_set)
    number_label_set = list(range(num_of_char_labels))
    combine_dict = dict(zip(char_label_set, number_label_set))
    new_label_list = []
    for item in label_list:
        temp_list = combine_dict[item]
        new_label_list.append(temp_list)
    return new_label_list, char_label_set, number_label_set

def change_seq_label_to_onehot(y, num_labels):
    print(" ### Enter Change Seqential Label to Onehot Label Module ... ... ...")
    num_samples = len(y)
    y_onehot = np.zeros((num_samples, num_labels))
    for i in range(len(y)):
        y_onehot[i, (y[i]-1)] = 1
    return y_onehot

def svm_model(X, Y, character):
    """
    SVM is a supervised machine learning model with the aim to find
    a hyperplane in an N-dimensional space (N features) that distinctly classifies
    the data points into two or multiple categories. SVM is more complicated and requires training.
    The training process is very computational expensive which takes several hours for training the training
data.
    The experiments show in small sample dataset, the classification accuracy can achieve around 80%,
    but for the large OCR dataset, it is hard for the model to get converged.

    """
    print(" ### Enter SVM Model Module ... ... ...")
    # Map 0 to -1
```

```
Y[Y==0] = -1
# Variables
tol = 0.01 #1e-3
max_passes = 5 ### 5
passes = 0
C = 1
sigma = 0.01
alphas = np.zeros(len(X))
b = 0
E = np.zeros(len(X))
eta = 0
L = 0
H = 0
K = construct_RBF_kernel_matrix(X, sigma)
m = Y.shape[0]

while passes < max_passes:
    num_changed_alphas = 0
    for i in range(m):
        print('character = %d, passes = %d, sample_index = %d' % (character, passes, i) )
        E[i] = b + np.sum(alphas*Y*K[:,i]) - Y[i]

        if (Y[i]*E[i] < -tol and alphas[i] < C) or (Y[i]*E[i] > tol and alphas[i] > 0):
            j = random.randint(0, m-1)
            while j == i:
                j = random.randint(0, m-1)

            E[j] = b + sum(alphas*Y*K[:,j]) - Y[j]
            alpha_i_old = alphas[i]
            alpha_j_old = alphas[j]
            if Y[i] == Y[j]:
                L = max(0, alphas[j] + alphas[i] - C)
                H = min(C, alphas[j] + alphas[i])
            else:
                L = max(0, alphas[j] - alphas[i])
                H = min(C, C + alphas[j] - alphas[i])

            if L == H:
                continue

            eta = 2 * K[i,j] - K[i,i] - K[j,j]
            if eta >= 0:
                continue

            alphas[j] = alphas[j] - (Y[j] * (E[i] - E[j])) / eta
            alphas[j] = min(H, alphas[j])
            alphas[j] = max(L, alphas[j])

            if abs(alphas[j] - alpha_j_old) < tol:
                alphas[j] = alpha_j_old
                continue
```

```
            alphas[i] = alphas[i] + Y[i]*Y[j]*(alpha_j_old - alphas[j])
            b1 = b - E[i] - Y[i] * (alphas[i] - alpha_i_old) *  K[i,i] - Y[j] * (alphas[j] - alpha_j_old) *  K[i,j]
            b2 = b - E[j] - Y[i] * (alphas[i] - alpha_i_old) *  K[i,j] - Y[j] * (alphas[j] - alpha_j_old) *  K[j,j]

            if 0 < alphas[i] and alphas[i] < C:
                b = b1
            elif 0 < alphas[j] and alphas[j] < C:
                b = b2
            else:
                b = (b1+b2)/2

            num_changed_alphas = num_changed_alphas + 1;
        if num_changed_alphas == 0:
            passes = passes + 1;
        else:
            passes = 0
    #save model dict
    model = {}
    idx = alphas > 0
    model['X'] = X[idx, :]
    model['y'] = Y[idx]
    model['K'] = K
    model['b'] = b
    model['alphas'] = alphas[idx]
    model['w'] = np.dot(alphas*Y, X).T
    return model

def construct_RBF_kernel_matrix(X, sigma):
    print(" ### Enter Construct RBF Kernel Matrix Module ... ... ...")
    num_samples = len(X)
    X_square = np.sum(np.square(X), 1)
    X_square = np.expand_dims(X_square, 0).repeat(num_samples, axis=0)
    X_square_T = X_square.T
    coefficient = -1 / (2 * sigma * sigma)
    square_diff = X_square - 2 * np.dot(X, X.T) + X_square_T
    RBF_Matrix = np.exp(coefficient * square_diff)
    return RBF_Matrix

def gaussian_kernel(X1, X2, sigma=0.1):
    sim = 0
    diff = X1 - X2
    sim = np.sum(diff*diff)
    sim = sim / (-2 * (sigma*sigma))
    sim = np.exp(sim)
    return sim

def svm_classification(X_train, X_test, model):
    svm_model_dict = model['model_dict']
    char_label_list_train = model['char_label_set_train']
    number_label_list_train = model['number_label_set_train']
    pred_label_list = []
    for i in range(len(X_test)):
```

```
        test_sample = X_test[i, :]
        pred_numerical_label = svm_classifier(test_sample, X_train, svm_model_dict)
        pred_char_label = convert_numerical_label_to_char(pred_numerical_label, number_label_list_train,
char_label_list_train)
        pred_label_list.append(pred_char_label)
    pred_label_list = np.array(pred_label_list)
    return pred_label_list

def convert_numerical_label_to_char(pred_number, number_label_list, char_label_list):

    index = number_label_list.index(pred_number)
    pred_char = char_label_list[index]
    return pred_char

def svm_classifier(test_input, X_train, model_dict):
    recorded_matrix = np.zeros(len(X_train))
    sigma = 0.1
    num_models = len(model_dict.keys())
    for i in range(len(X_train)):
        recorded_matrix[i] = gaussian_kernel(test_input, X_train[i, :], sigma)

    confidence_vector = np.zeros(num_models)
    for i in range(num_models):
        svm_model = model_dict[str(i)]
        alphas = svm_model['alphas']
        confidence_vector[i] = np.dot(alphas.T, recorded_matrix.T)
    max_index = 0

    for i in range(len(confidence_vector)):
        if confidence_vector[i] > confidence_vector[max_index]:
            max_index = i
    pred_result = max_index
    return pred_result
```

Pylint analysis

```
************* Module system
system.py:84:0: C0301: Line too long (122/100) (line-too-long)
system.py:113:0: C0301: Line too long (131/100) (line-too-long)
system.py:196:0: C0301: Line too long (108/100) (line-too-long)
system.py:200:0: C0325: Unnecessary parens after 'if' keyword (superfluous-parens)
system.py:203:0: C0301: Line too long (120/100) (line-too-long)
system.py:214:0: C0301: Line too long (107/100) (line-too-long)
system.py:216:0: C0325: Unnecessary parens after 'if' keyword (superfluous-parens)
system.py:272:0: C0325: Unnecessary parens after 'not' keyword (superfluous-parens)
system.py:272:0: C0325: Unnecessary parens after 'not' keyword (superfluous-parens)
system.py:277:0: C0301: Line too long (140/100) (line-too-long)
system.py:319:0: C0301: Line too long (131/100) (line-too-long)
system.py:320:0: C0301: Line too long (157/100) (line-too-long)
system.py:347:0: W0301: Unnecessary semicolon (unnecessary-semicolon)
system.py:381:0: C0325: Unnecessary parens after 'if' keyword (superfluous-parens)
system.py:396:0: C0325: Unnecessary parens after 'if' keyword (superfluous-parens)
```

```
system.py:399:0: C0325: Unnecessary parens after 'elif' keyword (superfluous-parens)
system.py:435:0: C0301: Line too long (103/100) (line-too-long)
system.py:440:0: C0325: Unnecessary parens after 'if' keyword (superfluous-parens)
system.py:443:0: C0325: Unnecessary parens after 'if' keyword (superfluous-parens)
system.py:452:0: C0325: Unnecessary parens after 'if' keyword (superfluous-parens)
system.py:454:0: C0301: Line too long (108/100) (line-too-long)
system.py:507:0: C0301: Line too long (114/100) (line-too-long)
system.py:508:0: C0301: Line too long (101/100) (line-too-long)
system.py:567:0: C0301: Line too long (119/100) (line-too-long)
system.py:568:0: C0301: Line too long (119/100) (line-too-long)
system.py:577:0: W0301: Unnecessary semicolon (unnecessary-semicolon)
system.py:579:0: W0301: Unnecessary semicolon (unnecessary-semicolon)
system.py:620:0: C0301: Line too long (127/100) (line-too-long)
system.py:649:0: C0304: Final newline missing (missing-final-newline)
system.py:1:0: C0114: Missing module docstring (missing-module-docstring)
system.py:7:0: E0401: Unable to import 'utils.utils' (import-error)
system.py:62:0: R0914: Too many local variables (17/15) (too-many-locals)
system.py:166:4: C0200: Consider using enumerate instead of iterating with range and len (consider-using-enumerate)
system.py:211:4: R1705: Unnecessary "else" after "return" (no-else-return)
system.py:250:4: W0612: Unused variable 'w' (unused-variable)
system.py:254:0: R0914: Too many local variables (19/15) (too-many-locals)
system.py:307:8: W0621: Redefining name 'k_nearest_neighbour' from outer scope (line 12) (redefined-outer-name)
system.py:329:0: R0914: Too many local variables (19/15) (too-many-locals)
system.py:349:4: R1705: Unnecessary "else" after "return" (no-else-return)
system.py:371:19: W0613: Unused argument 'page' (unused-argument)
system.py:409:0: C0115: Missing class docstring (missing-class-docstring)
system.py:409:0: R0903: Too few public methods (0/2) (too-few-public-methods)
system.py:459:0: C0116: Missing function or method docstring (missing-function-docstring)
system.py:460:4: C0200: Consider using enumerate instead of iterating with range and len (consider-using-enumerate)
system.py:472:0: C0116: Missing function or method docstring (missing-function-docstring)
system.py:481:0: C0116: Missing function or method docstring (missing-function-docstring)
system.py:494:0: C0116: Missing function or method docstring (missing-function-docstring)
system.py:498:4: C0200: Consider using enumerate instead of iterating with range and len (consider-using-enumerate)
system.py:502:0: R0914: Too many local variables (25/15) (too-many-locals)
system.py:570:19: C0122: Comparison should be alphas[i] > 0 (misplaced-comparison-constant)
system.py:572:21: C0122: Comparison should be alphas[j] > 0 (misplaced-comparison-constant)
system.py:502:0: R0912: Too many branches (14/12) (too-many-branches)
system.py:502:0: R0915: Too many statements (64/50) (too-many-statements)
system.py:593:0: C0116: Missing function or method docstring (missing-function-docstring)
system.py:604:0: C0116: Missing function or method docstring (missing-function-docstring)
system.py:612:0: C0116: Missing function or method docstring (missing-function-docstring)
system.py:625:0: C0116: Missing function or method docstring (missing-function-docstring)
system.py:640:8: W0621: Redefining name 'svm_model' from outer scope (line 502) (redefined-outer-name)
system.py:631:0: C0116: Missing function or method docstring (missing-function-docstring)
system.py:645:4: C0200: Consider using enumerate instead of iterating with range and len (consider-using-enumerate)
system.py:4:0: W0611: Unused import operator (unused-import)
system.py:3:0: C0411: standard import "import random" should be placed before "import numpy as np"
```

```
(wrong-import-order)
system.py:4:0: C0411: standard import "import operator" should be placed before "import numpy as np"
(wrong-import-order)
```

- Great code. No major issues. Well done.
- Use all caps for constants and declare at head of code with clear documentation.
- Avoid using short or cryptic variable names where it makes the code unclear.
- Some functions have no description. Would be good to comment all aspects of your code.
- Some clear, well-written code.

Score: 8/10 (Code)