Feedback report

Student: **acb16zje**. Total Score: 41/50

Feature	Classifier	Correction	Performance	Code	Total
9/10	8/10	6/10	9/10	9/10	41/50

Feature Extraction (Max 200 Words)

```
Steps:
1. Use the provided `process_training_data` function to extract features from
training data
2. Use PCA with 40 axes to reduce the perform dimensionality reduction on the
train data
3. Create a list of tuples called `char_compare_list` which contain all
possible comparision
   between unique labels, excluding self and duplicate comparision
4. Loop through `char_compare_list` and calculate the divergence of every pair
of labels
5. Pick the top 25 features with highest divergence using `np.argsort`
6. Loop through `char_compare_list` again and try to find the best overall 10
features using
   multidivergence in the list of 25 features. The best 10 features of each
pair of labels
   are stored in an array.
7. Count the top 10 most common features, and that will be our best features.
Initially I used divergence and correlation to find the top 10 features, but
the performance
was not good. In the multidivergence part, it requires a first feature to be
provided first in
order to find the best pair. So by trial and error, I have found out that
feature 1 gives the
best result.
```

- Sensible choice of PCA for the feature extraction.
- Credit for combining PCA with feature selection.
- An interesting and well described approach

Score: 9/10 (Features)

Classifier (Max 200 Words)

```
Originally I used the single nearest neighbour classifer and the result was
already impressive.
But the results on pages with high noise level can be improved. So I modified
the original
single nearest neighbour classifer to a k-nearest-neighbour classifier and used
`k = 3`.
If `k = 1`, then it will become a single nearest neighbour classifier. The
classifier will loop
through the test page, and classified the labels based on its `k` amount of
neighbours. For each
label in the test page, it looks for its `k` nearest neighbour and return the
most common label.
High amount of `k` makes the results on clear pages worse, but it does improve
the results on
pages with high noise level. By trial and error, 3 seems to be the optimum
value for `k`.
```

- Yes, a nearest-neighbour is a sensible choice for the classifier.
- Credit for experimenting with different values of K.
- Provide some experimental results to more fully justify your design choices.
- Some students fixed the performance trade-off problem you describe by estimating the noise level and setting the value of K accordingly.
- Credit for experimenting with noisy training data.

Score: 8/10 (Classifier)

Error Correction (Max 200 Words)

```
The test pages in the `dev` folder are using UK English. I have found a word
list online which
contains about 100k words sorted according to the word frequency. The word list
contain some
French and German words so I have to remove them and add some words manualluy.
Steps:
1. Declare two variables called `start_pos` and `end_pos` for recording the
starting and
   ending position of a word respectively.
2. Start by looping through all the labels returned by `classify_page`
function. Calculate the
   ending x-coordinate of the current label and the starting x-coordinate of
the next label.
   If the distance exceed a certain value, then join the labels starting from
`start_pos` to
   `end_pos + 1`.
3. The error correction will create a temporary dictionary for each word. It
will then
   look for the most similar word in the temporary dictionary. If the
similarity exceeds a
   threshold, then the word will be corrected.
4. After the error correction on the word, set the value of `start_pos` to
`end_pos + 1` and
   continue the loop.
5. Each successful loop will increase the value of `end_pos` by 1. The loop
continues until
   it reaches the end of the array.
```

- Well done for attempting this section.
- Sounds like a suitable approach.
- What do you do about punctuation etc?
- Provide test results to show whether this is improving the overall result or not.

Score: 6/10 (Correction)

Other information (Optional, Max 100 words)

```
In `process_training_data`, I tried adding some noise into the train data. The
results on page 4 to 6 were better.
But there are more performance drop on page 1 to page 3 than the improvement.
So I
commented it out.
In `load_test_page`, I tried to remove the noise from the pages. For each page,
it the "number of noise"
exceeds a certain threshold, then noise reduction will be performed on that
page. The noise reduction
will set pixels that are lower than a certain threshold to 0, and pixels that
are higher than a threshold
to 255.
```

Performance

```
The percentage errors (to 1 decimal place) for the development data are as
follows:
- Page 1: 97.9%
- Page 2: 99.0%
- Page 3: 97.9%
- Page 4: 85.4%
- Page 5: 67.7%
- Page 6: 54.3%
```

- Scores on test pages:
    - Page 1: 96.0%
    - Page 2: 96.1%
    - Page 3: 94.7%
    - Page 4: 78.9%
    - Page 5: 63.2%
    - Page 6: 49.7%
- Average correct = 79.8%
- Geometric mean error = 12.2%
- Percentiles: 93.8%, 94.6%

Excellent result. Among top 10%.

Score: 9/10 (Performance)

Code

```
"""OCR classification system.

PCA and k-Nearest Neighbour solution

version: v1.4
"""

import collections as coll
import Levenshtein
import numpy as np
import scipy.linalg
import utils.utils as utils

# Constants
MAX_DIMENSIONALITY = 10
NUM_PCA_AXES = 40


def feature_selection(pcatrain_data, labels_train):
    """
    Select the ten best features by using PCA train data and run
```

```
multidivergence

    :param pcatrain_data: The reduced dimension of train data
    :param labels_train: The correct labels for train data
    :return: The selected 10 best features
    """

    labels_train_unique = np.array(list(set(labels_train)))
    num_labels = len(labels_train_unique)

    # Create a list of labels tuple
    char_compare_list = [(labels_train_unique[c1], labels_train_unique[c2])
                         for c1 in range(num_labels)
                         for c2 in range(c1 + 1, num_labels)
                         if np.sum(labels_train == labels_train_unique[c1]) > 1
                         and np.sum(labels_train == labels_train_unique[c2]) >
1]

    # Compute divergence between every pair or characters
    dvg_list = []
    for char1, char2 in char_compare_list:
        char1_data = pcatrain_data[labels_train == char1, :]
        char2_data = pcatrain_data[labels_train == char2, :]
        d12 = divergence(char1_data, char2_data)
        dvg_list.append(np.array(d12))

    # Pick the top 25 features with highest divergence
    dvg_list = np.sum(np.array(dvg_list), axis=0)
    dvg_list = np.argsort(-dvg_list)[:25]

    # # Compute correlation and pick the top 25 (doesn't give better resuls)
    # corr = np.abs(np.corrcoef(pcatrain_data, rowvar=0))
    # corr = np.argsort(corr, axis=1)
    # corr = np.argsort(-corr, axis=1)[:, 0:25]
    # corr = np.ravel(corr)
    # best_corr = [bc[0] for bc in coll.Counter(corr).most_common()]
    # best_corr.sort()

    # Use multidivergence to get the overall best features,
    best_first_feature = 1
    features_list = []
    for char1, char2 in char_compare_list:
        char1_data = pcatrain_data[labels_train == char1, :]
        char2_data = pcatrain_data[labels_train == char2, :]
        features = [best_first_feature]
        nfeatures = [feature for feature in dvg_list if feature not in
features]

        # Get the best 10 overall features when comparing char1 and char2
        while len(features) < MAX_DIMENSIONALITY:
            multi_d = []
            for i in nfeatures:
```

```
                test_features = list(features)
                test_features.append(i)
                multi_d.append(multidivergence(char1_data, char2_data,
test_features))

            # Append the best testing features into the features
            index = np.argmax(multi_d)
            features.append(nfeatures[index])
            nfeatures.remove(nfeatures[index])

        features_list.append(features)

    # Get 10 overall features which appear the most
    overall_features_list = np.array(features_list).flatten()
    overall_features_count =
coll.Counter(overall_features_list).most_common(MAX_DIMENSIONALITY)
    best_overall_features = np.array([feature[0] for feature in
overall_features_count])

    return best_overall_features


def divergence(class1, class2):
    """compute a vector of 1-D divergences

    :param class1: data matrix for class 1, each row is a sample
    :param class2: data matrix for class 2
    :return: d12 - a vector of 1-D divergence scores
    """

    # Compute the mean and variance of each feature vector element
    m1 = np.mean(class1, axis=0)
    m2 = np.mean(class2, axis=0)
    v1 = np.var(class1, axis=0)
    v2 = np.var(class2, axis=0)

    # Plug mean and variances into the formula for 1-D divergence.
    # (Note that / and * are being used to compute multiple 1-D
    #  divergences without the need for a loop)
    d12 = 0.5 * (v1 / v2 + v2 / v1 - 2.0) + 0.5 * (m1 - m2) * (m1 - m2) * (1.0
/ v1 + 1.0 / v2)

    return d12


def multidivergence(class1, class2, features):
    """compute divergence between class1 and class2

    :param class1: data matrix for class 1, each row is a sample
    :param class2: data matrix for class 2
    :param features: the subset of features to use
    :return: d12 - a scalar divergence score
```

```
    """

    ndim = len(features)

    # compute mean vectors
    mu1 = np.mean(class1[:, features], axis=0)
    mu2 = np.mean(class2[:, features], axis=0)

    # compute distance between means
    dmu = mu1 - mu2

    # compute covariance and inverse covariance matrices
    cov1 = np.cov(class1[:, features], rowvar=0)
    cov2 = np.cov(class2[:, features], rowvar=0)

    icov1 = np.linalg.pinv(cov1)
    icov2 = np.linalg.pinv(cov2)

    # plug everything into the formula for multivariate gaussian divergence
    d12 = (0.5 * np.trace(np.dot(icov1, cov2) + np.dot(icov2, cov1) - 2 *
np.eye(ndim))
           + 0.5 * np.dot(np.dot(dmu, icov1 + icov2), dmu))

    return d12


def generate_pca_axes(fvectors_train):
    """Compute the principal components and the mean of the train data

    :param fvectors_train: The full feature vectors of training data
    :return: The principal components axes and the mean of the train data
    """

    covx = np.cov(fvectors_train, rowvar=0)
    n = covx.shape[0]
    w, v = scipy.linalg.eigh(covx, eigvals=(n - NUM_PCA_AXES, n - 1))
    v = np.fliplr(v)
    return v


def pca(fvectors, pca_axes):
    """
    Projecting the train or test data onto the
    principal components axes

    :param fvectors: The feature vectors of train or test data
    :param pca_axes: The principal components axes computed
    :return: The data with reduced dimension
    """
    pca_data = np.dot((fvectors - np.mean(fvectors)), pca_axes)
    return pca_data
```

```
def reduce_dimensions_train(feature_vectors_full, model):
    """Reduce the dimension of the train data by using PCA

    :param feature_vectors_full: feature vectors stored as rows in a matrix
    :param model: a dictionary storing the outputs of the model training stage
    :return: The train data with dimension reduced, and use the 10 best
features
    """

    labels_train = np.array(model['labels_train'])

    pca_axes = generate_pca_axes(feature_vectors_full)
    pcatrain_data = pca(feature_vectors_full, pca_axes)
    best_features = feature_selection(pcatrain_data, labels_train)

    # Store data into dictionary for reusing them in testing
    model['pca_axes'] = pca_axes.tolist()
    model['best_features'] = best_features.tolist()

    return pcatrain_data[:, model['best_features']]


def reduce_dimensions_test(feature_vectors_full, model):
    """Reduce the dimension of the test data by using PCA

    :param feature_vectors_full: feature vectors stored as rows in a matrix
    :param model: a dictionary storing the outputs of the model training stage
    :return: The test data with dimension reduced, and use the 10 best features
    """

    pcatest_data = pca(feature_vectors_full, model['pca_axes'])

    return pcatest_data[:, model['best_features']]


def get_bounding_box_size(images):
    """Compute bounding box size given list of images.

    :param images: A list of images stored as arrays
    :return: The height and width of the bounding box
    """

    height = max(image.shape[0] for image in images)
    width = max(image.shape[1] for image in images)

    return height, width


def images_to_feature_vectors(images, bbox_size=None):
    """Reformat characters into feature vectors.
```

```
    Takes a list of images stored as 2D-arrays and returns
    a matrix in which each row is a fixed length feature vector
    corresponding to the image.abs

    :param images: a list of images stored as arrays
    :param bbox_size: an optional fixed bounding box size for each image
    :return: The reformatted feature vectors
    """

    # 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


# The three functions below this point are called by train.py
# and evaluate.py and need to be provided.

def process_training_data(train_page_names):
    """Perform the training stage and return results in a dictionary.

    :param train_page_names: List of training page names
    :return: Dictionary storing the results
    """

    print('Reading data')
    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)

    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)

    # # Add some noise to the training data (not much overall improvement)
    # for i in range(fvectors_train_full.shape[0]):
```

```
    #     noise = np.random.randint(80, size=fvectors_train_full.shape[1])
    #     fvectors_train_full[i][:] = np.add(fvectors_train_full[i][:], noise)

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

    print('Reducing to 10 dimensions')
    fvectors_train = reduce_dimensions_train(fvectors_train_full, model_data)
    model_data['fvectors_train'] = fvectors_train.tolist()

    print('Loading the word lists')
    dictionary = []
    with open('data/train/dictionary.txt', 'r') as f:
        for line in f:
            dictionary.append(line.strip('\n'))

    model_data['dict'] = dictionary

    return model_data


def load_test_page(page_name, model):
    """Load test data page.

    This function must return each character as a 10-d feature
    vector with the vectors stored as rows of a matrix.

    :param page_name: name of page file
    :param model: dictionary storing data passed from training stage
    :return: The feature vector reduced to 10 dimensions
    """

    bbox_size = model['bbox_size']
    images_test = utils.load_char_images(page_name)
    fvectors_test = images_to_feature_vectors(images_test, bbox_size)

    # Remove noise from with high noise level
    for row in fvectors_test:
        col = row.flatten()
        noise_threshold = np.sum(col < 255) - np.sum(col == 0)

        # If there are a lot of noise detected in the character image, remove
the noise
        if noise_threshold > 75:
            row[row < 20] = 0
            row[row > 120] = 255

    # Perform the dimensionality reduction.
    fvectors_test_reduced = reduce_dimensions_test(fvectors_test, model)

    return fvectors_test_reduced
```

```
def get_corrected_word(word, model):
    """Perform the error correction and return the correct word

    :param word: The classified word
    :param model: dictionary storing data passed from training stage
    :return: The corrected word
    """

    temp_dict = [words for words in model['dict'] if len(words) == len(word)]
    words_score = []
    words_dist = []

    # Set the threshold for error correction
    ratio_threshold = 0.5 if len(word) < 4 else 0.82
    dist_threshold = 1

    # Get the scores for each of the word in the dictionary
    for words in temp_dict:
        words_score.append(Levenshtein.ratio(word, words))
        words_dist.append(Levenshtein.distance(word, words))

    # Perform error correction if the similarity ratio pass the threshold
    if max(words_score) >= ratio_threshold:
        best_match = np.argmax(words_score)
        corrected_word = temp_dict[best_match]

    # For case where it doesn't pass the ratio threshold, but there exist a
very close word
    elif min(words_dist) == dist_threshold:
        best_match = np.argmin(words_dist)
        corrected_word = temp_dict[best_match]
    else:
        corrected_word = word

    return corrected_word


def correct_errors(page, labels, bboxes, model):
    """
    Perform error correction on all words. Return the original word
    if it's correct

    :param page: 2d array, each row is a feature vector to be classified
    :param labels: the output classification label for each feature vector
    :param bboxes: 2d array, each row gives the 4 bounding box coords of the
character
    :param model: dictionary, stores the output of the training stage
    :return: The corrected word
    """
```

```
    spacing = 6
    start_pos = 0
    end_pos = 0
    corrected_labels = []

    for label in range(len(bboxes)):
        # If it is the last word on the page
        if label == len(bboxes) - 1:
            char_distance = spacing + 1
        else:
            char_distance = np.abs(bboxes[label + 1][0] - bboxes[label][2])

        if char_distance > spacing:
            word = "".join(labels[start_pos:end_pos + 1])

            # If the classified word is in the dictionary
            if word.lower() in model['dict']:
                corrected_word = word

            # If the last character of the word is a symbol
            elif not word[-1].isalpha() and word[-2:] != "l'":
                # Remove the last character
                temp_word = word[:-1]

                # If the word with symbol removed is in the dictionary
                if temp_word in model['dict']:
                    corrected_word = word
                else:
                    corrected_word = get_corrected_word(temp_word, model)

                # Add the symbol back if error correction if performed
                corrected_word = corrected_word + word[-1] if corrected_word !=
word else word

                # If the previous character of the word is one of ".?!"
                if labels[start_pos - 1] in '.?!':
                    corrected_word = corrected_word.capitalize()

            # If the previous character of the word is one of ".?!"
            elif labels[start_pos - 1] in '.?!':
                # Lower case the word
                temp_word = word.lower()

                if temp_word in model['dict']:
                    corrected_word = word
                else:
                    corrected_word = get_corrected_word(temp_word,
model).capitalize()

            # If the first two character is I' or l' (" in page)
            elif word[:2] == "I'" or word[:2] == "l'":
                # Sometimes l is classified as I
```

```
                word = "l" + word[1:] if word[:2] == "I'" else word

                temp_word = word[2:]
                if temp_word in model['dict']:
                    corrected_word = word
                else:
                    corrected_word = word[:2] + get_corrected_word(temp_word,
model)

            # If the last two character is l' (" in page)
            elif word[-2:] == "l'":
                # Check if the last character before l' is .?!
                temp_word = word[:-3] if word[-3] in '.?!' else word[:-2]

                # If the word with symbol removed is in the dictionary
                if temp_word in model['dict']:
                    corrected_word = word
                else:
                    corrected_word = get_corrected_word(temp_word, model)

                    if word[-3] in '.?!':
                        corrected_word += word[-3:]
                    else:
                        corrected_word += word[-2:]

            # If the word is not in the dictionary
            else:
                corrected_word = get_corrected_word(word, model)

            # Misclassified comma
            if corrected_word[-1] == "'" and corrected_word[-2:] != "l'":
                corrected_word = corrected_word[:-1] + ","

            # print("{:<20}".format(str(word)) + " to " + corrected_word)
            corrected_labels.extend(list(corrected_word))

            start_pos = end_pos + 1

        end_pos += 1

    return np.array(corrected_labels)


def classify_page(page, model):
    """Use nearest neighbour classification to find the correct label

    :param page: matrix, each row is a feature vector to be classified
    :param model: dictionary, stores the output of the training stage
    :return: The estimated correct label
    """

    fvectors_train = np.array(model['fvectors_train'])
```

```
    labels_train = np.array(model['labels_train'])
    k = 3

    # if only one test vector then make sure to cast into skinny matrix
    if page.ndim == 1:
        page = page[np.newaxis]

    # Super compact implementation of nearest neighbour
    x = np.dot(page, fvectors_train.transpose())
    modtest = np.sqrt(np.sum(page * page, axis=1))
    modtrain = np.sqrt(np.sum(fvectors_train * fvectors_train, axis=1))
    dist = x / np.outer(modtest, modtrain.transpose())  # cosine distance

    # If k = 1, it becomes a single nearest neighbour
    if k == 1:
        nearest = np.argmax(dist, axis=1)
        labels = labels_train[nearest]

    # Classify the label based on k nearest neighbour
    else:
        # Sort according to the closest neighbour
        neighbours = np.argsort(-dist, axis=1)
        labels = []

        # Count the most common labels
        for char in range(dist.shape[0]):
            nearest_list = coll.Counter(labels_train[neighbours[char]
[:k]]).most_common(1)
            (nearest, _) = np.array(nearest_list)[0]
            labels.append(nearest)

    return np.array(labels)
```

Pylint analysis

```
************* Module system
C: 99, 4: Invalid variable name "m1" (invalid-name)
C:100, 4: Invalid variable name "m2" (invalid-name)
C:101, 4: Invalid variable name "v1" (invalid-name)
C:102, 4: Invalid variable name "v2" (invalid-name)
C:152, 4: Invalid variable name "n" (invalid-name)
C:153, 4: Invalid variable name "w" (invalid-name)
C:153, 7: Invalid variable name "v" (invalid-name)
C:154, 4: Invalid variable name "v" (invalid-name)
C:240, 8: Invalid variable name "h" (invalid-name)
C:240,11: Invalid variable name "w" (invalid-name)
C:241, 8: Invalid variable name "h" (invalid-name)
C:242, 8: Invalid variable name "w" (invalid-name)
C:286,51: Invalid variable name "f" (invalid-name)
W:362,19: Unused argument 'page' (unused-argument)
C:483, 4: Invalid variable name "x" (invalid-name)

-------------------------------------------------------------------
Your code has been rated at 9.31/10 (previous run: 8.44/10, +0.87)
```

- Excellent, well presented code. Well done.- Can correct_errors be simplified?

Score: 9/10 (Code)