Assignment 2

Introduction:

How to run:

I couldn't get it to run from terminal using wordsearch(test1, words, train_data, train_labels), but in the file, on lines 384-389 is the corresponding code to run the wordsearch function for each of the trials. So you can uncomment the line corresponding to the that trial and run.

Preprocessing:

To preprocess the data into 225 subarrays of 30x30 pixels, I defined the function "preprocessing" which takes in the 450x450 array and reshapes it into 225 30x30. Where each 30x30 block represents a character from the image. This array is then reshaped to form the 225x900 element matrix, called "preprocessed_wordsearch". I took this approach as it seemed more efficient than the nested for loop I had created previously to preprocess the data.

Classification:

Firstly, I state that all 699 elements of the train_data and train_labels should be used to train the classifier, this will provide us with the most accurate classification of letters.

Then, to classify the data, giving labels (representing the letter) to each of the 225x30x30 elements, I compute the principal components using either 900 dimensions or 10, depending on what trial is being tested. This occurs in the "computing_principal_components" function or the "learn_pca" and "reduce"pca" function.

I use the cosine distance to classify my data as this produced results upwards of 90%+ correct, meaning that more words were found.

Perform the search:

My function "find_word" starts the searching process; it starts by going through each [i][j] element and seeing if the letter found at that position is the same as the first letter of the word to be found. If it is, another function "find_direction" is called which looks at the surrounding 8 letters and checks to see if any of them are the second letter in the word to be found, if it is, a final function is called "possible_words" which takes the [i][j] co-ordinate of the start letter and goes the distance = length of the word in that direction. The function then checks if the letter found at this new position is the final letter of the word to be found, and if it is, then these co-ordinates are sent to the "draw_line" function which draws a line over top the word, indicating its position.

Display the result:

I have a function called "draw_line" that receives the [i] and [j] co-ordinates for both the start letter and end letter. The function then converts the row and column number into co-ordinates by * them by 30.

To make sure that the line goes over the word and not to the side, I either + or - 15 from the co-ordinate to centre the line over the word.

The function uses plt.plot() to draw the line, and I removed the colour parameter to make each line a different colour (it helps to show where one of my misclassifications is so that it doesn't look like one long line)

Loop over all words:

My function “word_loop” takes the list of 24 words and the classified wordsearch as parameters, and iterates over each word, converting it to a list of ints and calling my “find_word” function of the list of ints and classified wordsearch.

Dimensionality Reduction:

To reduce dimensionality, I used a different approach for when all (900) features were to be used (Trail1) and when only 10 features needed to be used (Trial2 & 3). For Trial1, I used the “compute_principal_components” which is an adaption from the PCA function from lab7. I adapted it to not take in test_labels, and it no longer computes an efficiency score or confusion matrix. For Trial2 and 3, I used the “learn_pca” and “reduce_pca” functions.

I have done it this way, as I found that the learn_pca and reduce_pca technique worked best for Trial2 and 3, where the train data for the classifier was much less suited to classifying images of such low resolution.

Noise robustness:

To help with the noise, I found that the learn_pca and reduce_pca method from lab10 was better in classifying the letters on the pictures for the more noisy test. Meaning that my searching method was able to find more words (up to 7 from 4).

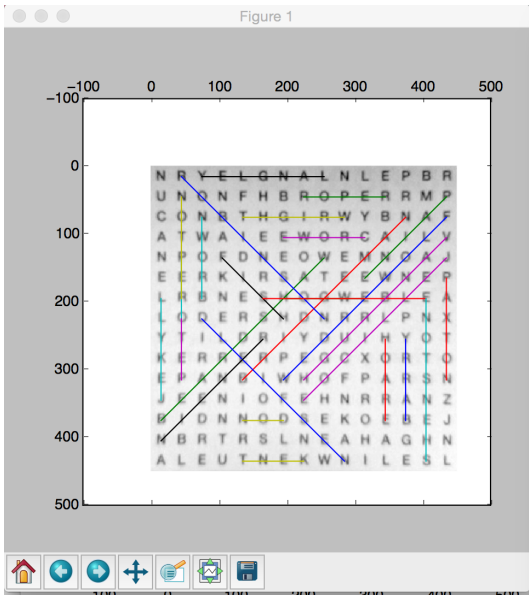
General Performance:

Using time.time(): for Trial1 I recorded a time of: 0.25 seconds
for Trial2 I recorded a time of: 0.18 seconds
for Trial3 I recorded a time of: 016 seconds

Trial 1:

Words Found: 23, missing “Cane”.
“Peto” and “Shenstone” had duplicates.

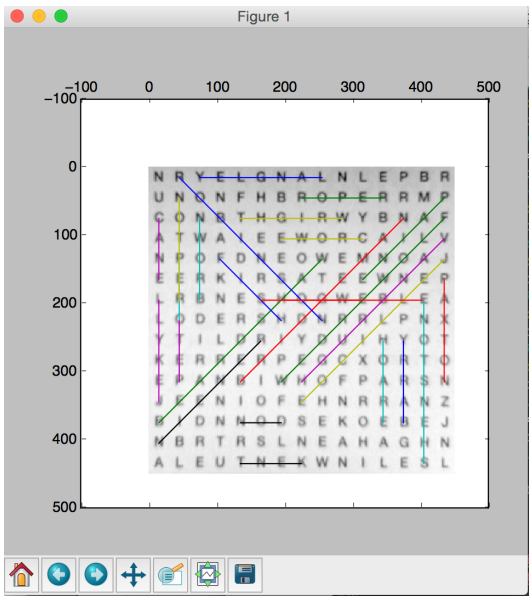
wordsearch(test1, words, train_data, train_labels



Trial 2:

Words Found: 23, missing “Nesfield”.
“Peto” and “Shenstone” had duplicates.

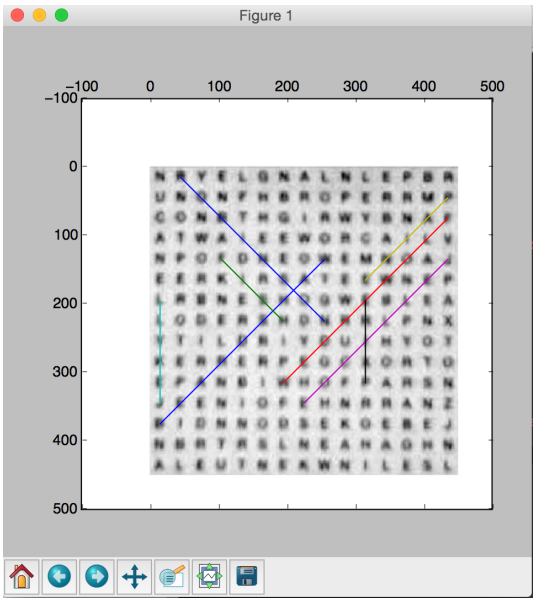
wordsearch(test1, words, train_data, train_labels, range(1,11))



Trial 3:

Words Found: 7, words found: “Fish”, “Paxton”, “Vanbrugh”, “Jekyll”, “Paine” and “Flowerdew”.

wordsearch(test2, words, train_data, train_labels, range(1,11))



Easily-readable listing of the final version of code:

```
import pickle
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import numpy as np
import scipy
import time
from scipy import linalg


def wordsearch(test, pwords, ptrain_data, ptrain_labels, dimensions=None):
    """The main function that runs the program.
    Returning the complete wordsearch.

    :param test: The image of the wordsearch; either test1 or test2.
    :param pwords: The list of words to be found.
    :param ptrain_data: The data used to train the classifier.
    :param ptrain_labels: The labels that correspond to the train_data.
    :param dimensions: The number of dimensions to be used.
    :return: An image of the wordsearch with lines over top indicating
            word positions.
    """

    wordsearch_array = np.array(test)
    # Displays the wordsearch image being used.
    plt.matshow(test, cmap=cm.gray)

    # Converts the 450x450 array into 225x30x30, with each 30x30 block a letter.
    wordsearch_subarray = preprocessing(wordsearch_array)
    # Converts the 225x30x30 array into 225x900.
    preprocessed_wordsearch = np.reshape(
        wordsearch_subarray, (225, 900), order='F')

    # Use all 699 of the train_data to train the classifier.
    training_data = ptrain_data[0:, :]
    training_labels = ptrain_labels[0:]

    # Classify the test set to get an array of guessed labels
    # Use all features if no feature parameter has been supplied.
    if dimensions is None:
        test_guessed = computing_principal_components(
            training_data, training_labels, preprocessed_wordsearch, None)
    # Uses 10 features if a feature parameter has been supplied.
    else:
        [v, mean_vector] = learn_pca(training_data, 40)
        a = reduce_pca(training_data, v, mean_vector)
        b = reduce_pca(preprocessed_wordsearch, v, mean_vector)
        test_guessed = classify(a, training_labels, b, dimensions)


    # Converts the classified wordsearch into a 15x15 block,
    # representing the actual wordsearch.
    word_search = test_guessed.reshape(15, 15)

    # Loops over all 24 words,
    # attempting to find them in the classified wordsearch.
    word_loop(pwords, word_search)
```

```
def preprocessing(array):
    """Processes the data into individual characters of size 30x30.
    It reshapes the array into smaller subblocks, each containg
    a letter, and also maintaining the size of the array.

    :param array: the 450x450 array of pixels to be reshaped.
    :return: an array of subarrays of size 30x30.
    """

    height, width = array.shape
    reshaped_array = array.reshape(height//30, 30, -1, 30)
    axes_swapped = reshaped_array.swapaxes(1,2)
    preprocessed_subarrays = axes_swapped.reshape(-1, 30, 30)
    return preprocessed_subarrays


def computing_principal_components(
        ptrain_data, ptrain_labels, test_data, features=None):
    """Computing principal components returns the labels for each of the
     225 letters.

    :param ptrain_data: The data used to train the classifier.
    :param ptrain_labels: The labels corresponding to the train data.
    :param test_data: The data that will be classified.
    :param features: The number of dimensions to use
    :return: The classified wordsearch
    """

    # Computes the covariance matrix from the training data.
    covx = np.cov(ptrain_data, rowvar=0)
    # The number of rows in the covarance matrix.
    N = covx.shape[0]
    # The principal components are the eigenvectors of the covariance matrix.
    w, v = scipy.linalg.eigh(covx, eigvals=(N - 900, N - 1))
    # A column vector of principal component axes
    v = np.fliplr(v)

    # Centering the pca data by subtracting the mean letter vector.
    pcatrain_data = np.dot((ptrain_data - np.mean(ptrain_data)), v)
    pcatest_data = np.dot((test_data - np.mean(ptrain_data)), v)

    return classify(pcatrain_data, ptrain_labels, pcatest_data, features)


def learn_pca(data, N):
    """Learn PCA dimensionality reduction transform.

    data - the data to learn the transform from
    N - the desired number of dimensions.

    Returns:
    pca_axes - the PCA transform matrix
    data_mean - the data mean vector
    """
    # Calculate the number of rows in the data matrix
    ndata = data.shape[0]
    # Calculate and display the mean face
    mean_vector = np.mean(data, axis=0)
```

```
    # Calculate the mean normalised data
    delta_data = data - mean_vector

    # (You'd expect the next line to be deltadata'*deltadata but
    # a computational shortcut is being employed...)
    u = np.dot(delta_data, delta_data.transpose())
    # The principal components are the eigenvectors of the covariance matrix
    d, pc = scipy.linalg.eigh(u, eigvals=(ndata - N, ndata - 1))
    # A column vector of principal component axes
    pc = np.fliplr(pc)
    pca_axes = np.dot(pc.transpose(), delta_data)
    # compute the mean vector
    mean_vector = np.mean(data, axis=0)

    return pca_axes, mean_vector


def reduce_pca(data, pca_axes, mean_vector):
    """Apply the PCA dimensionality reduction transform. Returns the matrix"""

    # subtract mean from all data points
    centered = data - mean_vector
    # project points onto PCA axes
    reduced_data = np.dot(centered, pca_axes.transpose())

    return reduced_data


def classify(train_data, train_labels, test_data, features=None):
    """Nearest Neighbour Classifier

    :param train_data: The data used to train the classifier.
    :param train_labels: The labels that correspond to train_data.
    :param test_data: The data matrix which stores the test data.
    :param features: A vector that indicates the features to be used
    :return: The labels for the word search.
    """

    # Use all feature if no feature parameter has been supplied
    if features is None:
        features = np.arange(0, train_data.shape[1])

    # Select the desired features from the training and test data
    train = train_data[:, features]
    test = test_data[:, features]

    # Super compact implementation of nearest neighbour
    # Computes the inner product on the transposed word search matrix.
    x = np.dot(test, train.transpose())
    # Computes the distance matrix for each pair in test data
    modtest = np.sqrt(np.sum(test * test, axis=1))
    # Computes the distance matrix for each pair in train data
    modtrain = np.sqrt(np.sum(train * train, axis=1))
    # cosine distance
    dist = x / np.outer(modtest, modtrain.transpose())
    nearest = np.argmax(dist, axis=1)
    # Computes the labels for each [i][j] index in the matrix.
    label = train_labels[nearest]
    return label
```

```
def word_loop(words, wordsearch):
    """Converts the words into their corresponding numbers and calls
    the find_word function to try and find the word within the
    wordsearch. If successful, outputs the wordsearch with lines
    drawn over top, indicating the position of the words.

    :param words: The list of names that need to be found in the wordsearch.
    :return: The wordsearch with the words found.
    """

    for word in words:
        int_word = convert_name_to_int(word)
        find_word(int_word, wordsearch)


def convert_name_to_int(name):
    """Takes the name given and converts each letter into its corresponding
    index in the alphabet. e.g. "A" would be converted into 1.
    Does this by finding the ASCI number of the letter and - 96
    to get the number.

    :param name: The name to be found
    :return: The name to be found in number form
    """

    output = []
    for letters in name:
        number = ord(letters) - 96
        output.append(number)
    return output


def find_word(int_word, wordsearch):
    """Searches through the word search looking for potential word matches.
    Works by finding all occurrences of the first letter in the word,
    then seeing if any of the surrounding characters are the second
    letter in the word (this gives a direction). If this is found,
    it will lastly see if the last letter of the word is the same
    as found on the word search.

    :param int_word: The word to be found in integer form.
    :param wordsearch: The classified word search.
    :return: The start and end co-ordinates of the word
             and a line drawn between them.
    """

    for i in range (15):
        for j in range (15):
            # If the first letter of the word to be found is equal to
            # the letter at co-ordinates [i][j], the "find_direction"
            # function is used to determine the direction in which
            # the word is written.
            if int_word[0] == wordsearch[i][j]:
                direction = find_direction(i, j, int_word[1], wordsearch)
                # If no directions are found, indicates that the letter is
                # on its own (not surrounded by the second letter in the word)
                if len(direction) > 0:
                    # If there is a direction, the "possible_words" function
                    # determines if it's a possible word by seeing if the
```

```
                # last letter of the word is equal to the letter found
                # at the end co-ordinates.
                possible_tripple = possible_words(
                    i, j, int_word, wordsearch, direction[0])
                if possible_tripple != None:
                    end_letter = possible_tripple[0]
                    end_i = possible_tripple[1]
                    end_j = possible_tripple[2]
                    # If the last letter in the word to be found is
                    # equal to the letter at co-ordinate end_i, end_j
                    # we plot a line between the start co-ordinates and
                    # end co-ordinates.
                    if int_word[len(int_word)-1] == end_letter:
                        draw_line(i, j, end_i, end_j)


def find_direction(i, j, int_letter, wordsearch):
    """Takes the co-ordinates of the position of the first letter in the word
    and checks the surrounding 8 co-ordinates to see if any of them are the
    second letter to be found. If it finds one, it returns the direction.
    There is also logic to make sure that if the start letter is on the side
    of the wordsearch, the function wont look in directions the word obviously
    wont be in.

    :param i: The start co-ordinate of the potential word.
    :param j: The start co-ordinate of the potential word.
    :param int_letter: The word to be found in integer form.
    :param wordsearch: The classified word search.
    :return: An int indicating the direction in which the word is facing.
    """

    result = []
    # Direction = Up
    if i > 0 and int_letter == wordsearch[i - 1][j]:
        result.append(1)
    # Direction = Right
    if j < 14 and int_letter == wordsearch[i][j + 1]:
        result.append(4)
    # Direction = Left
    if j > 0 and int_letter == wordsearch[i][j - 1]:
        result.append(3)
    # Direction = Down
    if i < 14 and int_letter == wordsearch[i + 1][j]:
        result.append(2)
    # Direction = Down Right
    if j < 14 and i < 14 and int_letter == wordsearch[i + 1][j + 1]:
        result.append(8)
    # Direction = Up Right
    if j < 14 and i > 0 and int_letter == wordsearch[i - 1][j + 1]:
        result.append(7)
    # Direction = Down Left
    if i < 14 and j > 0 and int_letter == wordsearch[i + 1][j - 1]:
        result.append(6)
    # Direction = Up Left
    if j > 0 and i > 0 and int_letter == wordsearch[i - 1][j - 1]:
        result.append(5)

    return result
```

```
def possible_words(i, j, int_word, wordsearch, direction):
    """The function goes in the direction of the word for
    the length of the word, and returns the letter at that position
    and the co-ordinates of that letter.

    :param i: The start co-ordinate of the potential word.
    :param j: The start co-ordinate of the potential word.
    :param int_word: The word to be found in integer form.
    :param wordsearch: The classified word search.
    :param direction: The direction the word is facing.
    :return: An array containing the end letter, and the end co-ordinates.
    """

    word_length = len(int_word)

    if direction == 1 and i-(word_length-1) >= 0:
        return result_append(
            wordsearch[i-(word_length-1)][j], i-(word_length-1), j)
    if direction == 2 and i+(word_length-1) < 15:
        return result_append(
            wordsearch[i+(word_length-1)][j], i+(word_length-1), j)
    if direction == 3 and j-(word_length-1) >= 0:
        return result_append(
            wordsearch[i][j-(word_length-1)], i, j-(word_length-1))
    if direction == 4 and j+(word_length-1) < 15:
        return result_append(
            wordsearch[i][j+(word_length-1)], i, j+(word_length-1))
    if direction == 5 and i-(word_length-1) >=0 and j-(word_length-1) >= 0:
        return result_append(
            wordsearch[i-(word_length-1)][j-(word_length-1)],
            i-(word_length-1), j-(word_length-1))
    if direction == 6 and i+(word_length-1) < 15 and j-(word_length-1) >= 0:
        return result_append(
            wordsearch[i+(word_length-1)][j-(word_length-1)],
            i+(word_length-1), j-(word_length-1))
    if direction == 8 and i+(word_length-1) <15 and j+(word_length-1) < 15:
        return result_append(
            wordsearch[i+(word_length-1)][j+(word_length-1)],
            i+(word_length-1), j+(word_length-1))
    if direction == 7 and i-(word_length-1) >= 0 and j+(word_length-1) < 15:
        return result_append(
            wordsearch[i-(word_length-1)][j+(word_length-1)],
            i-(word_length-1), j+(word_length-1))


def result_append(letter, i, j):
    """Receives the location and type of the possible end of the word and
    adds it to an array.

    :param letter: the end letter of the word
    :param i: the co-ordinate of the end letter in the wordsearch
    :param j: the co-ordinate of the end letter in the wordsearch
    :return: an array of 3 elements, the end letter, and the co-ordinates.
    """

    result = [letter, i, j]
    return result


def draw_line(i, j, end_i, end_j):
```

```
    """Draws a coloured line over where it thinks a word may be.

    :param i: Start co-ordinate for the word
    :param j: Start co-ordinate for the word
    :param end_i: End co-ordinate for the word
    :param end_j: End co-ordinate for the word
    :return: A line indicating the position of the word.
    """

    plt.plot(
        ((j * 30) + 15, (end_j * 30) + 15),
        ((i * 30) + 15, (end_i * 30) + 15))


# Loading the data and assigning the corresponding parts to variables.
data = pickle.load(open("assignment2.pkl", "rb"))
train_data = data['train_data']
train_labels = data['train_labels']
test1 = data['test1']
test2 = data['test2']
words = data['words']

start = time.time()

# Calling the main function to run the program, can remove the "range(1, 11)"
# to use all 900 features.

# De-comment to run Trial1
wordsearch(test1, words, train_data, train_labels)
# De-comment to run Trial2
# wordsearch(test1, words, train_data, train_labels, range(1,11)
# De-comment to run Trial3
# wordsearch(test2, words, train_data, train_labels, range(1,11)

end = time.time()
print(end-start)

# Plots the final image, showing the lines on the wordsearch.
plt.show()
```