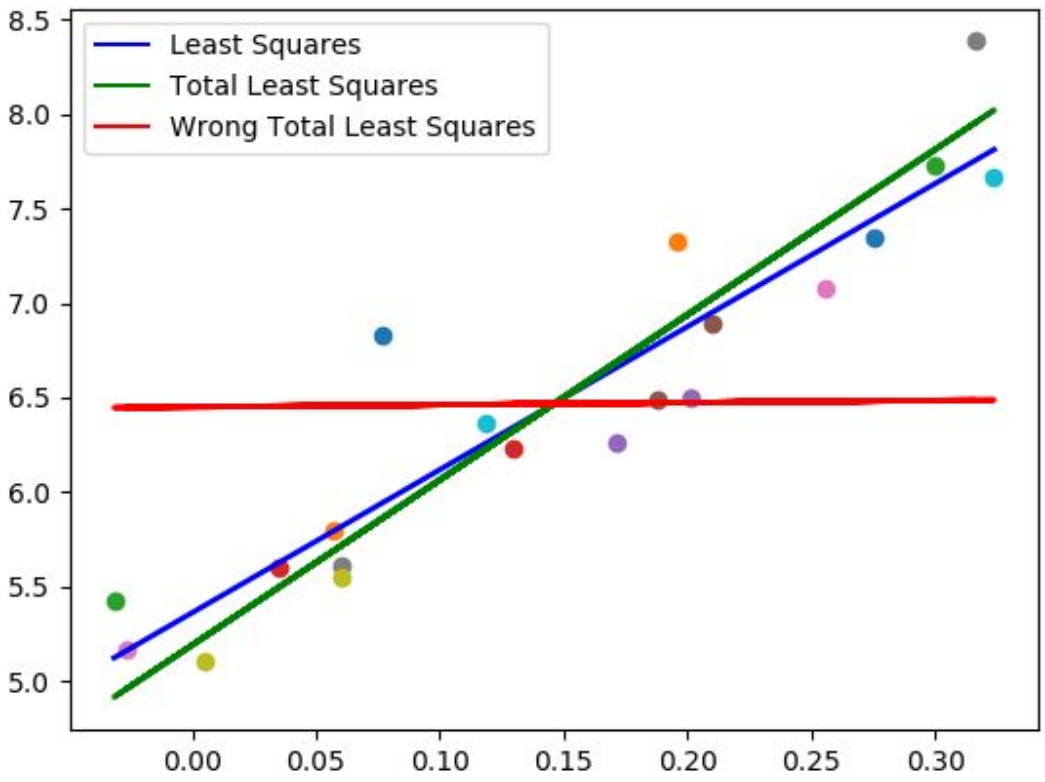
Image Analysis Assignment 3
Olof Harrysson

1.

```
def least_squares(x, y):
  A = np.append(x, np.ones(x.shape), axis=0).T
  p = np.linalg.lstsq(A, y.T)
  k, c = p[0]
  least_square_error = p[1]
  print("Least square error is {}".format(least_square_error))

  return k, c


def total_least_squares(x, y):
  x = x.flatten()
  y = y.flatten()
  n = x.size
  x2 = sum([a**2 for a in np.nditer(x)])
  y2 = sum([a**2 for a in np.nditer(y)])
  xy = np.sum([a*b for a, b in zip(x, y)])
  x_sum = np.sum(x)
  y_sum = np.sum(y)
  n_xy = xy - 1 / n * x_sum * y_sum

  lag_m = [[x2 - 1 / n * x_sum ** 2, n_xy],
           [n_xy, y2 - 1 / n * y_sum ** 2]]

  eig_vals, eig_vecs = np.linalg.eig(lag_m)

  a1 = eig_vecs[0][0]
  b1 = eig_vecs[1][0]

  a2 = eig_vecs[1][0]
  b2 = eig_vecs[1][1]

  c1 = -(1 / n) * (a1 * x_sum + b1 * y_sum)
  c2 = -(1 / n) * (a2 * x_sum + b2 * y_sum)

  y1 = (-c1 - a1 * x) / b1
  y2 = (-c2 - a2 * x) / b2;

  return y1, y2
```

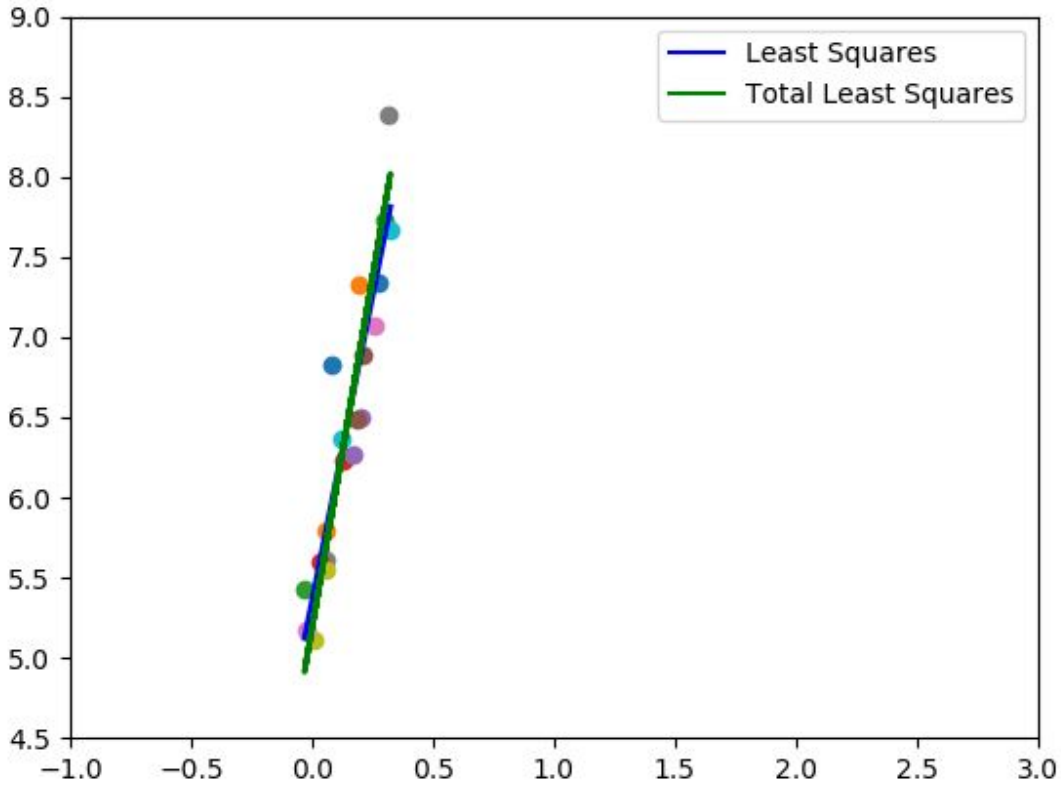


As is evident from the figure above both the total least square and the least square methods produces a line that approximates the points. The total least square method minimizes the total squared distance between the points and the line whilst the least square looks at the total squared distance in the y-direction. In this case there isn't much of a difference between the methods but there are cases where the difference would be more evident.

Errors:

```
(deep_learn) Olofs-MacBook-Pro:3 olof$ python linefit.py
Least square error for the least_square line is 2.224433330996876
Least square error for the total_least_square line is 2.5637105269531335

Total least square error for the least_square line is 0.03816492597247226
Total least square error for the total_least_square line is 0.0331243265145014
```

Comparing the errors we see that the least square errors are bigger. This is because for a steep line the vertical distance is much bigger than the orthogonal distance. This can be seen in the graph below.



2.

I implemented a nearest neighbour classifier. It compares the norms of pictures.

```
import scipy.io
import numpy as np
import sys
import numpy.linalg as LA
import matplotlib.pyplot as plt

def nearest_neighbour(train_x, train_y, test_x, test_y, n_img_to_plot=0):
  nbr_correct = 0

  for test_x_i, test_y_i in zip(test_x, test_y):
    if n_img_to_plot > 0:
      plot(classify_nn(train_x, train_y, test_x_i), test_y_i, test_x_i) # Plots image and
classification
      n_img_to_plot -= 1

    nbr_correct += 1 if classify_nn(train_x, train_y, test_x_i) == test_y_i else 0

  return nbr_correct / test_x.shape[0]
```

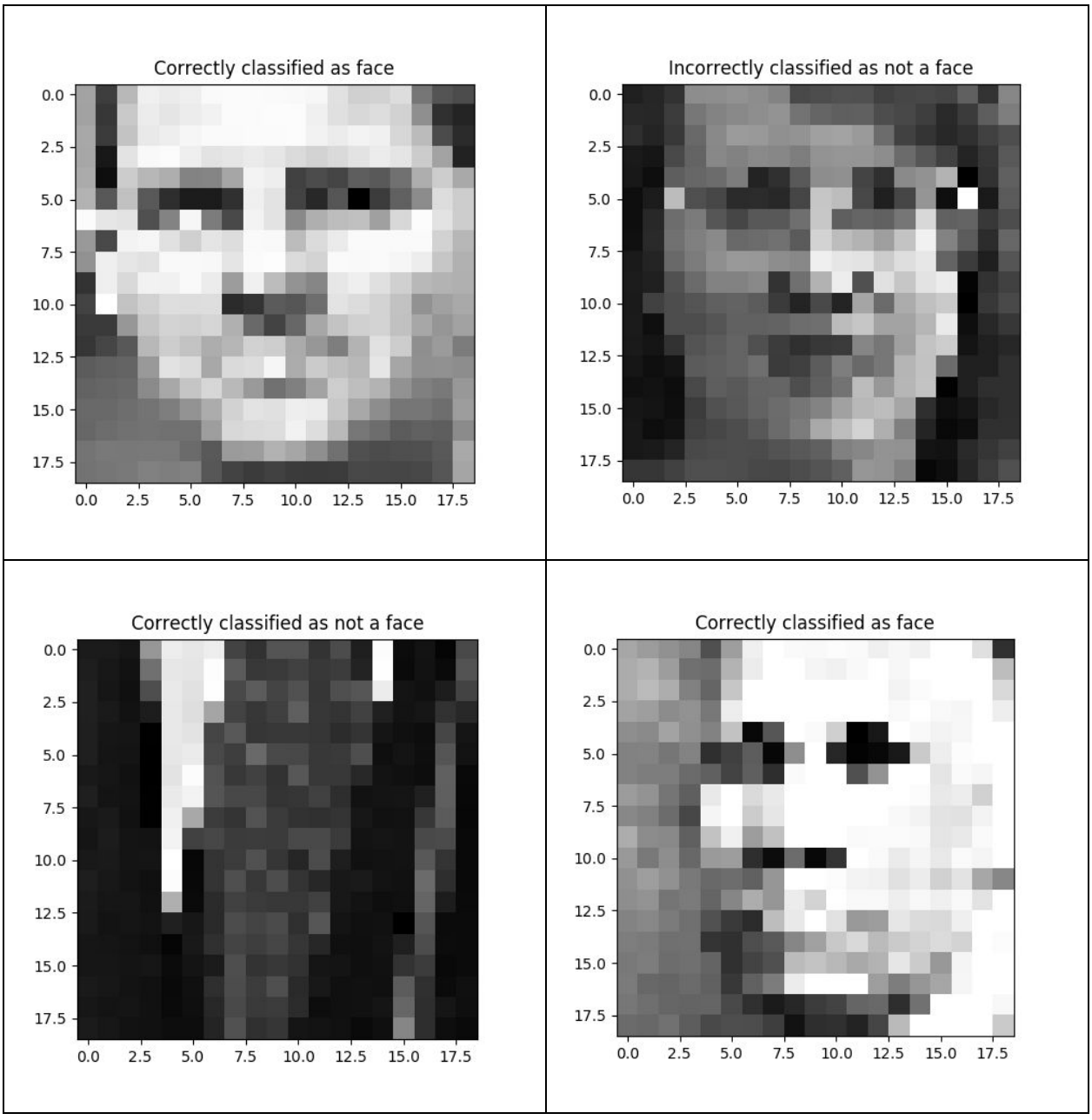
```
def classify_nn(train_x, train_y, test_x):
  norms = LA.norm(train_x - test_x, ord=2, axis=1)
  return train_y[np.argmin(norms)]


def plot(prediction, ground_truth, img):
  img = np.reshape(img, (19, 19), order='F')
  fig = plt.figure()
  nbr_to_face = lambda x: 'face' if x==[1] else 'not a face'

  if prediction == ground_truth:
    plt.title('Correctly classified as {}'.format(nbr_to_face(prediction)))
  else:
    plt.title('Incorrectly classified as {}'.format(nbr_to_face(prediction)))

  imgplot = plt.imshow(img, cmap='gray')
  plt.show()

################################### START ###################################

mat = scipy.io.loadmat('FaceNonFace.mat')
x = mat['X'].T.astype(np.int16)
y = mat['Y'].T

accuracies = []
n_loops = 100
for i in range(n_loops):
  msk = np.random.rand(len(x)) < 0.8

  train_x = x[msk]
  train_y = y[msk]

  test_x = x[~msk]
  test_y = y[~msk]

  accuracies.append(nearest_neighbour(train_x, train_y, test_x, test_y, n_img_to_plot=3))

print("Mean accuracy of {} loops: {}".format(n_loops, np.mean(accuracies)))
```

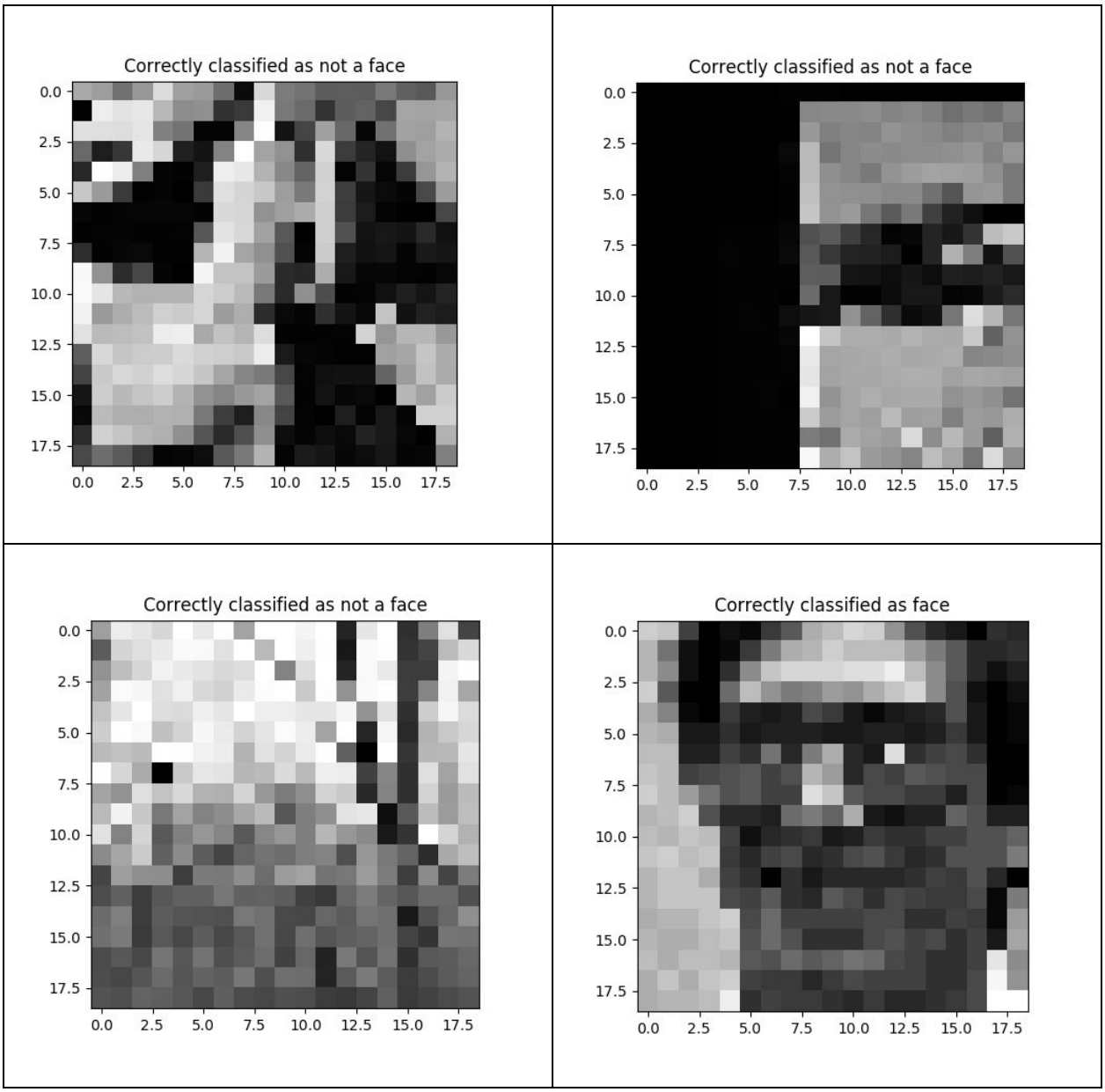
When running the training and testing phase 100 times with different train/test data the mean accuracy for the test data is

```
Mean accuracy of 100 loops: 0.8352746908038595
Mean accuracy for the training data doing 100 loops: 1.0
```

The program never makes a mistake when it's testing on the training data. This is because the nearest neighbour always is itself, having a norm of 0. It's considered a big nono to test on training data ;)

To verify that the program is working it could also be helpful to manually inspect some of its decisions.





3.

```
load('../../ocrfeaturestrain');
X = transpose(X);
Y = transpose(Y);

letters_to_compare = [['p' 't'];
                      ['o' 'q']
                      ['i' 'l']
                      ['w' 'm']
                      ['y' 'v']
```

```
                    ['j' 'l']
                    ['e' 'f']
                    ];
n_pairs = size(letters_to_compare,1);

A = [];
for i= 1:n_pairs
    l1 = letters_to_compare(i,1);
    l2 = letters_to_compare(i,2);
    A = [A; premade_train_and_predict(X, Y, l1, l2)];
end;

A
mean = sum(A, 1) / n_pairs


--------------------------
function [ accuracy ] = premade_train_and_predict(X, Y, letter1, letter2 )

letter_to_position = @(x) x - 'a' + 1;
letter1 = letter_to_position(letter1);
letter2 = letter_to_position(letter2);

l1_or_l2 = (Y == letter1 | Y == letter2);

X_filtered = X(l1_or_l2, :);
Y_filtered = Y(l1_or_l2, :);

n_data = size(X_filtered, 1);
part = cvpartition(n_data, 'HoldOut', 0.20);

x_train = X_filtered(part.training);
y_train = Y_filtered(part.training);

x_test = X_filtered(part.test);
y_test = Y_filtered(part.test);


tree = fitctree(x_train, y_train); % Regression tree classifier
SVMModel = fitcsvm(x_train, y_train); % Support Vector Machine
mdl = fitcknn(x_train, y_train); % Nearest Neighbour Classifier


tree_p = tree.predict(x_test);
SVM_p = SVMModel.predict(x_test);
mdl_p = mdl.predict(x_test);

n_test = size(x_test, 1);

tree_acc = sum(tree_p == y_test) / n_test;
SVM_acc = sum(SVM_p == y_test) / n_test;
```

```
mdl_acc = sum(mdl_p == y_test) / n_test;

accuracy = [tree_acc SVM_acc mdl_acc];

end
```

Running the script gives the following accuracies.

Letters	Regression Tree Classifier	Support Vector Machine	Nearest Neighbour Classifier
P & T	0.8333	0.8333	1.0000
O & Q	0.8571	1.0000	1.0000
I & L	0.6000	0.4000	0.6000
W & M	1.0000	1.0000	1.0000
Y & V	0.8000	0.7000	0.5000
J & L	0.5000	0.5000	0.6667
E & F	0.5000	0.7500	0.3750

With a mean of 0.7272 0.7405 0.7345 for the techniques in the table above.

4.
I decided to use matlabs already implemented Nearest Neighour classifier over my own since I wrote mine in python. It seemed like a headache to somehow connect my python script with the test and benchmark script in matlab. I wanted to write my classifier in python as I'm trying to get as much exposure to numpy and python as possible.

My features2class where I fit and predict

```
function y = features2class(x,classification_data)
x_train = cell2mat(classification_data(1));
y_train = cell2mat(classification_data(2));

mdl = fitcknn(x_train, y_train); % Nearest Neighbour Classifier

y = mdl.predict(transpose(x));
```

My inl3_stub where I acquire my features

```
load('../../ocrsegments.mat')

X = cellfun(@segment2features, S ,'uniformoutput',false);
X = transpose(cell2mat(X));

classification_data = {X,y};
```

```
save('classification_data.mat', 'classification_data')
```

Running the test and benchmark gives a hit rate of 0.98 on the short1 dataset and NaN on the other datasets. I think that the problem lies in the segmentation. I tested this and due to noisier images the segmenter finds small noisy pixels and not the whole letter. I corrected this for the fourth assignment as I only pick the largest segments.