Orthogonal Projections

We will write functions that will implement orthogonal projections.

Learning objectives

1. Write code that projects data onto lower-dimensional subspaces.
2. Understand the real world applications of projections.

As always, we will first import the packages that we need for this assignment.

In [1]:

```
# PACKAGE: DO NOT EDIT THIS CELL
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight')
import numpy as np
```

Next, we will retrieve the Olivetti faces dataset.

In [2]:

```
from sklearn.datasets import fetch_olivetti_faces, fetch_lfw_people
from ipywidgets import interact
%matplotlib inline
image_shape = (64, 64)
# Load faces data
dataset = fetch_olivetti_faces('./')
faces = dataset.data
```

Advice for testing numerical algorithms

Before we begin this week's assignment, there are some advice that we would like to give for writing functions that work with numerical data. They are useful for finding bugs in your implementation.

Testing machine learning algorithms (or numerical algorithms in general) is sometimes really hard as it depends on the dataset to produce an answer, and you will never be able to test your algorithm on all the datasets we have in the world. Nevertheless, we have some tips for you to help you identify bugs in your implementations.

1. Test on small dataset

Test your algorithms on small dataset: datasets of size 1 or 2 sometimes will suffice. This is useful because you can (if necessary) compute the answers by hand and compare them with the answers produced by the computer program you wrote. In fact, these small datasets can even have special numbers, which will allow you to compute the answers by hand easily.

2. Find invariants

Invariants refer to properties of your algorithm and functions that are maintained regardless of the input. We will highlight this point later in this notebook where you will see functions, which will check invariants for some of the answers you produce.

Invariants you may want to look for:

1. Does your algorithm always produce a positive/negative answer, or a positive definite matrix?
2. If the algorithm is iterative, do the intermediate results increase/decrease monotonically?
3. Does your solution relate with your input in some interesting way, e.g. orthogonality?

Finding invariants is hard, and sometimes there simply isn't any invariant. However, DO take advantage of them if you can find them. They are the most powerful checks when you have them.

We can find some invariants for projections. In the cell below, we have written two functions which check for invariants of projections. See the docstrings which explain what each of them does. You should use these functions to test your code

See the docstrings which explain what each of them does. You should use these functions to test your code.

In [3]:

```
import numpy.testing as np_test
def test_property_projection_matrix(P):
    """Test if the projection matrix satisfies certain properties.
    In particular, we should have P @ P = P, and P = P^T
    """
    np_test.assert_almost_equal(P, P @ P)
    np_test.assert_almost_equal(P, P.T)

def test_property_projection(x, p):
    """Test orthogonality of x and its projection p."""
    np_test.assert_almost_equal(p.T @ (p-x), 0)
```

1. Orthogonal Projections

Recall that for projection of a vector $\boldsymbol x$ onto a 1-dimensional subspace U with basis vector $\boldsymbol b$ we have

$${\pi_U}(\boldsymbol x) = \frac{\boldsymbol b\boldsymbol b^T}{{\lVert\boldsymbol b \rVert}^2}\boldsymbol x$$

And for the general projection onto an M-dimensional subspace U with basis vectors $\boldsymbol b_1,\dotsc, \boldsymbol b_M$ we have

$${\pi_U}(\boldsymbol x) = \boldsymbol B(\boldsymbol B^T\boldsymbol B)^{-1}\boldsymbol B^T\boldsymbol x$$

where

$$\boldsymbol B = [\boldsymbol b_1,...,\boldsymbol b_M]$$

Your task is to implement orthogonal projections. We can split this into two steps

1. Find the projection matrix $\boldsymbol P$ that projects any $\boldsymbol x$ onto U.
2. The projected vector $\pi_U(\boldsymbol x)$ of $\boldsymbol x$ can then be written as $\pi_U(\boldsymbol x) = \boldsymbol P\boldsymbol x$.

To perform step 1, you need to complete the function `projection_matrix_1d` and `projection_matrix_general`. To perform step 2, complete `project_1d` and `project_general`.

In [4]:

```
# GRADED FUNCTION: DO NOT EDIT THIS LINE

# Projection 1d

# ===YOU SHOULD EDIT THIS FUNCTION===
def projection_matrix_1d(b):
    """Compute the projection matrix onto the space spanned by `b`
    Args:
        b: ndarray of dimension (D, 1), the basis for the subspace

    Returns:
        P: the projection matrix
    """
    D, _ = b.shape
    ### Edit the code below to compute a projection matrix of shape (D,D)
    P = (b @ b.T)/(np.linalg.norm(b)**2) # <-- EDIT THIS
    return P
    ###

# ===YOU SHOULD EDIT THIS FUNCTION===
def project_1d(x, b):
    """Compute the projection matrix onto the space spanned by `b`
    Args:
        x: the vector to be projected
        b: ndarray of dimension (D, 1), the basis for the subspace

    Returns:
        y: ndarray of shape (D, 1) projection of x in space spanned by b
    """
    p = (b @ b.T)/(np.linalg.norm(b)**2) @ x # <-- EDIT THIS
    return p
```

```
# Projection onto a general (higher-dimensional) subspace
# ===YOU SHOULD EDIT THIS FUNCTION===
def projection_matrix_general(B):
    """Compute the projection matrix onto the space spanned by the columns of `B`
    Args:
        B: ndarray of dimension (D, M), the basis for the subspace

    Returns:
        P: the projection matrix
    """
    P = (B@(np.linalg.inv(B.T@B))@B.T) # <-- EDIT THIS
    return P

# ===YOU SHOULD EDIT THIS FUNCTION===
def project_general(x, B):
    """Compute the projection matrix onto the space spanned by the columns of `B`
    Args:
        x: ndarray of dimension (D, 1), the vector to be projected
        B: ndarray of dimension (D, M), the basis for the subspace

    Returns:
        p: projection of x onto the subspac spanned by the columns of B; size (D, 1)
    """
    p = (B@(np.linalg.inv(B.T@B))@B.T)@x # <-- EDIT THIS
    return p
```

We have included some unittest for you to test your implementation.

In [5]:

```
# Orthogonal projection in 2d
# define basis vector for subspace
b = np.array([2,1]).reshape(-1, 1)
# point to be projected later
x = np.array([1,2]).reshape(-1, 1)
```

Remember our discussion earlier about invariants? In the next cell, we will check that these invariants hold for the functions that you have implemented earlier.

In [6]:

```
# Test 1D
# Test that we computed the correct projection matrix
np_test.assert_almost_equal(projection_matrix_1d(np.array([1, 2, 2]).reshape(-1,1)),
                            np.array([[1,  2,  2],
                                      [2,  4,  4],
                                      [2,  4,  4]]) / 9)

# Test that we project x on to the 1d subspace correctly
np_test.assert_almost_equal(project_1d(np.ones((3,1)),
                                       np.array([1, 2, 2]).reshape(-1,1)),
                            np.array([5, 10, 10]).reshape(-1,1) / 9)

B = np.array([[1, 0],
              [1, 1],
              [1, 2]])

# Test 2D
# Test that we computed the correct projection matrix
np_test.assert_almost_equal(projection_matrix_general(B),
                            np.array([[5,  2, -1],
                                      [2,  2,  2],
                                      [-1, 2,  5]]) / 6)

# Test that we project x on to the 2d subspace correctly
np_test.assert_almost_equal(project_general(np.array([6, 0, 0]).reshape(-1,1), B),
                            np.array([5, 2, -1]).reshape(-1,1))
```

It is always good practice to create your own test cases. Create some test cases of your own below!

In [7]:

In [7]:

```
# Write your own test cases here, use random inputs, utilize the invariants we have!
```

2. Eigenfaces (optional)

Next, we will take a look at what happens if we project some dataset consisting of human faces onto some basis we call the "eigenfaces". You do not need to know what `eigenfaces` are for now but you will know what they are towards the end of the course!

As always, let's import the packages that we need.

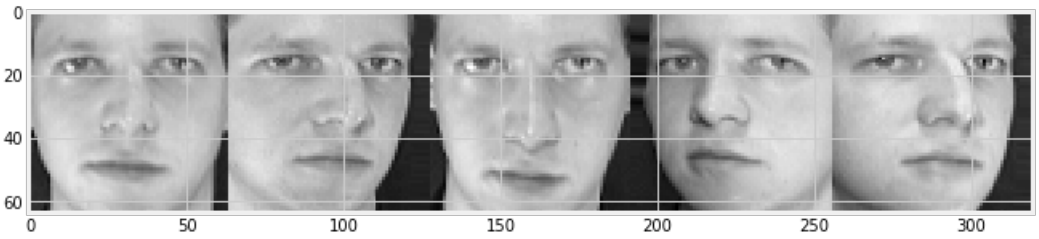
In [8]:

```
from sklearn.datasets import fetch_olivetti_faces, fetch_lfw_people
from ipywidgets import interact
%matplotlib inline
image_shape = (64, 64)
# Load faces data
dataset = fetch_olivetti_faces('./')
faces = dataset.data
```

Let's visualize some faces in the dataset.

In [9]:

```
plt.figure(figsize=(10,10))
plt.imshow(np.hstack(faces[:5].reshape(5,64,64)), cmap='gray');
```



In [10]:

```
# for numerical reasons we normalize the dataset
mean = faces.mean(axis=0)
std = faces.std(axis=0)
faces_normalized = (faces - mean) / std
```

The data for the basis has been saved in a file named `eigenfaces.npy`, first we load it into the variable B.

In [11]:

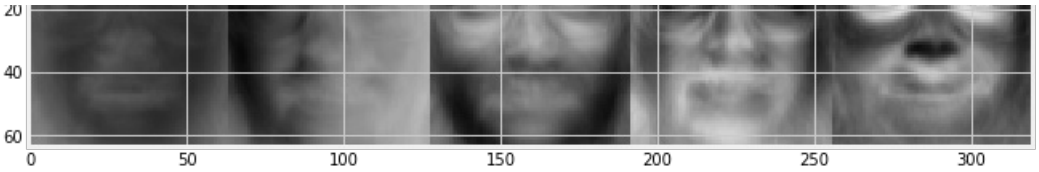
```
B = np.load('eigenfaces.npy')[:50] # we use the first 50 basis vectors --- you should play around
with this.
print("the eigenfaces have shape {}".format(B.shape))
```

```
the eigenfaces have shape (50, 64, 64)
```

Each instance in \boldsymbol B is a `64x64' image, an "eigenface", which we determined using an algorithm called Principal Component Analysis. Let's visualize a few of those "eigenfaces".

In [12]:

```
plt.figure(figsize=(10,10))
plt.imshow(np.hstack(B[:5].reshape(-1, 64, 64)), cmap='gray');
```



Take a look at what happens if we project our faces onto the basis $\boldsymbol B$ spanned by these 50 "eigenfaces". In order to do this, we need to reshape $\boldsymbol B$ from above, which is of size (50, 64, 64), into the same shape as the matrix representing the basis as we have done earlier, which is of size (4096, 50). Here 4096 is the dimensionality of the data and 50 is the number of data points.

Then we can reuse the functions we implemented earlier to compute the projection matrix and the projection. Complete the code below to visualize the reconstructed faces that lie on the subspace spanned by the "eigenfaces".

In [13]:

```
# EDIT THIS FUNCTION
@interact(i=(0, 10))
def show_face_face_reconstruction(i):
    original_face = faces_normalized[i].reshape(64, 64)
    # reshape the data we loaded in variable `B`
    # so that we have a matrix representing the basis.
    B_basis = np.random.normal(size=(4096,50)) # <-- EDIT THIS
    face_reconstruction = project_general(faces_normalized[i], B_basis).reshape(64, 64)
    plt.figure()
    plt.imshow(np.hstack([original_face, face_reconstruction]), cmap='gray')
    plt.show()
```

What would happen to the reconstruction as we increase the dimension of our basis?

Modify the code above to visualize it.

3. Least squares regression (optional)

Consider the case where we have a linear model for predicting housing prices. We are predicting the housing prices based on features in the housing dataset. If we denote the features as $\boldsymbol x_0, \dotsc, \boldsymbol x_n$ and collect them into a vector $\boldsymbol {x}$, and the price of the houses as y. Assuming that we have a prediction model in the way such that $\hat{y}_i = f(\boldsymbol {x}_i) = \boldsymbol \theta^T\boldsymbol {x}_i$.

If we collect the dataset into a (N,D) data matrix $\boldsymbol X$, we can write down our model like this:

$$\begin{bmatrix} \boldsymbol{x}_1^T \\ \vdots \\ \boldsymbol{x}_N^T \end{bmatrix} \boldsymbol{\theta} = \begin{bmatrix} y_1 \\ \vdots \\ y_2 \end{bmatrix},$$

i.e.,

$$\boldsymbol X\boldsymbol{\theta} = \boldsymbol{y}.$$

Note that the data points are the *rows* of the data matrix, i.e., every column is a dimension of the data.

Our goal is to find the best $\boldsymbol\theta$ such that we minimize the following objective (least square).

$$\begin{eqnarray} & \sum^n_{i=1}{\lVert \bar{y_i} - y_i \rVert^2} \\ &= \sum^n_{i=1}{\lVert \boldsymbol \theta^T\boldsymbol{x}_i - y_i \rVert^2} \\ &= (\boldsymbol X\boldsymbol {\theta} - \boldsymbol y)^T(\boldsymbol X\boldsymbol {\theta} - \boldsymbol y). \end{eqnarray}$$

If we set the gradient of the above objective to $\boldsymbol 0$, we have $\begin{eqnarray} \nabla_\theta(\boldsymbol X\boldsymbol {\theta} - \boldsymbol y)^T(\boldsymbol X\boldsymbol {\theta} - \boldsymbol y) &=& \boldsymbol 0 \\ \nabla_\theta(\boldsymbol {\theta}^T\boldsymbol X^T - \boldsymbol y^T)(\boldsymbol X\boldsymbol {\theta} - \boldsymbol y) &=& \boldsymbol 0 \\ \nabla_\theta(\boldsymbol {\theta}^T\boldsymbol X^T\boldsymbol X\boldsymbol {\theta} - \boldsymbol y^T\boldsymbol X\boldsymbol \theta - \boldsymbol \theta^T\boldsymbol X^T\boldsymbol y + \boldsymbol y^T\boldsymbol y) &=& \boldsymbol 0 \\ 2\boldsymbol X^T\boldsymbol X\theta - 2\boldsymbol X^T\boldsymbol y &=& \boldsymbol 0 \\ \boldsymbol X^T\boldsymbol X\boldsymbol \theta &=& \boldsymbol X^T\boldsymbol y. \end{eqnarray}$

The solution that gives zero gradient solves (which we call the maximum likelihood estimator) the following equation:

$$\boldsymbol X^T\boldsymbol X\boldsymbol \theta = \boldsymbol X^T\boldsymbol y$$

This is exactly the same as the normal equation we have for projections.

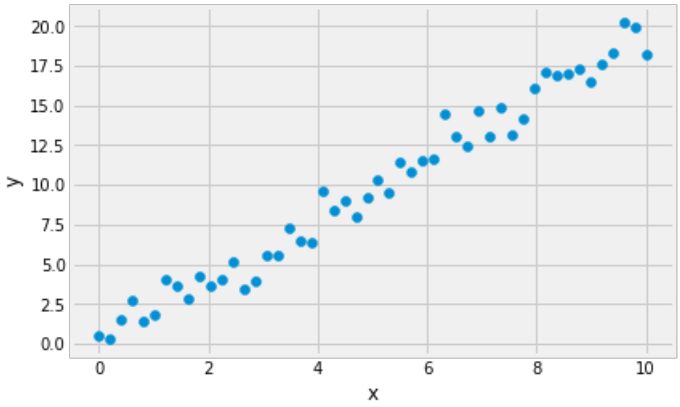
This means that if we solve for $\boldsymbol X^T\boldsymbol X\boldsymbol \theta = \boldsymbol X^T\boldsymbol y$. we would find the best $\boldsymbol \theta = (\boldsymbol X^T\boldsymbol X)^{-1}\boldsymbol X^T\boldsymbol y$, i.e. the $\boldsymbol \theta$ which minimizes our objective.

Let's put things into perspective. Consider that we want to predict the true coefficient $\boldsymbol \theta$ of the line $\boldsymbol y = \boldsymbol \theta^T \boldsymbol x$ given only $\boldsymbol X$ and $\boldsymbol y$. We do not know the true value of $\boldsymbol \theta$.

Note: In this particular example, $\boldsymbol \theta$ is a number. Still, we can represent it as an $\mathbb{R}^1$ vector.

In [14]:

```
x = np.linspace(0, 10, num=50)
theta = 2
def f(x):
    random = np.random.RandomState(42) # we use the same random seed so we get deterministic output
    return theta * x + random.normal(scale=1.0, size=len(x)) # our observations are corrupted by so
me noise, so that we do not get (x,y) on a line

y = f(x)
plt.scatter(x, y);
plt.xlabel('x');
plt.ylabel('y');
```



In [15]:

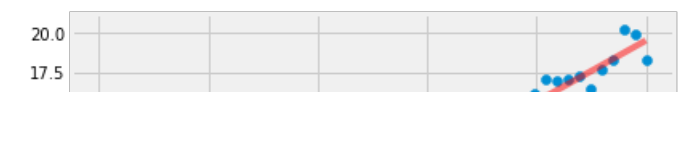
```
X = x.reshape(-1,1) # size N x 1
Y = y.reshape(-1,1) # size N x 1

# maximum likelihood estimator
theta_hat = np.linalg.solve(X.T @ X, X.T @ Y)
```

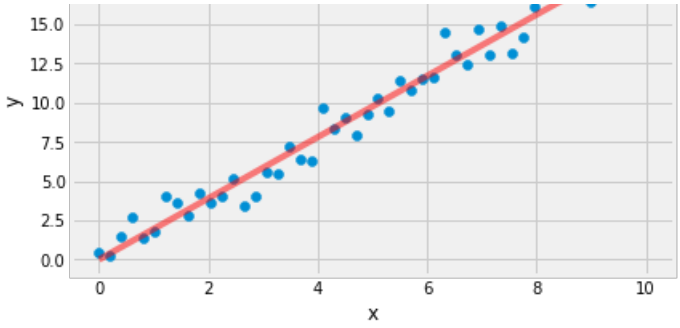
We can show how our $\hat{\boldsymbol \theta}$ fits the line.

In [16]:

```
fig, ax = plt.subplots()
ax.scatter(x, y);
xx = [0, 10]
yy = [0, 10 * theta_hat[0,0]]
ax.plot(xx, yy, 'red', alpha=.5);
ax.set(xlabel='x', ylabel='y');
print("theta = %f" % theta)
print("theta_hat = %f" % theta_hat)
```

```
theta = 2.000000
theta_hat = 1.951585
```





What would happend to $\lVert \hat{\boldsymbol \theta} - \boldsymbol \theta \rVert$ if we increase the number of datapoints?

Make your hypothesis, and write a small program to confirm it!

In [17]:

```
N = np.arange(2, 10000, step=10)
# Your code comes here, which calculates \hat{\theta} for different dataset sizes.

theta_error = np.zeros(N.shape)

theta_error = np.ones(N.shape) # <-- EDIT THIS


plt.plot(theta_error)
plt.xlabel("dataset size")
plt.ylabel("parameter error");
```