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 numpy as np
import matplotlib
import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight')
%matplotlib inline
```

Next, we will retrieve the Olivetti faces dataset.

In [2]:
```
from sklearn.datasets import fetch_olivetti_faces
from ipywidgets import interact
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.

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 $\boldsymbol{U}$ with basis vector $\boldsymbol{b}$ we have

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

And for the general projection onto an M-dimensional subspace $\boldsymbol{U}$ with basis vectors $\boldsymbol{b}_1, \dots, \boldsymbol{b}_M$ we have

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

where

$$\boldsymbol{B} = [\boldsymbol{b}_1, \ldots, \boldsymbol{b}_M]$$

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 $\boldsymbol{U}$.
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`.

Projection (1d)

Recall that you can use `np.dot(a, b)` or `a@b` to perform matrix-matrix or matrix-vector multiplication. `a*b` shall compute the element-wise product of $\boldsymbol{a}$ and $\boldsymbol{b}$.

You may find the function np.outer() useful.

Remember that the transpose operation does not do anything for 1 dimensional arrays. So you cannot compute $\boldsymbol{b}\boldsymbol{b}^T$ using `np.dot(b, b.T)`.

In [4]:
```python
# GRADED FUNCTION: DO NOT EDIT THIS LINE
def projection_matrix_1d(b):
    """Compute the projection matrix onto the space spanned by `b`
    Args:
        b: ndarray of dimension (D,), the basis for the subspace

    Returns:
        P: the projection matrix
    """
    # YOUR CODE HERE
    ### Uncomment and modify the code below
    D, = b.shape
    ### Edit the code below to compute a projection matrix of shape (D,D)
    P = np.zeros((D, D)) # <-- EDIT THIS
    # You may be tempted to follow the formula and implement bb^T as b @ b.T in numpy.
    # However, notice that this b is a 1D ndarray, so b.T is an no-op. Use np.outer instead
    # to implement the outer product.
    P = np.outer(b, b) / np.square(np.linalg.norm(b))
    return P
```

With the help of the function `projection_matrix_1d`, you should be able to implement `project_1d`.

In [5]:
```python
# GRADED FUNCTION: DO NOT EDIT THIS LINE
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,), the basis for the subspace

    Returns:
        y: ndarray of shape (D,) projection of x in space spanned by b
    """
    # YOUR CODE HERE
    ### Uncomment and modify the code below
    # <-- EDIT THIS
    P = projection_matrix_1d(b) @ x
    return P
```

In [6]:
```python
# Test 1D
# Test that we computed the correct projection matrix
from numpy.testing import assert_allclose

assert_allclose(
    projection_matrix_1d(np.array([1, 2, 2])),
    np.array([[1,  2,  2],
              [2,  4,  4],
              [2,  4,  4]]) / 9
)

# Some hidden tests below
```

In [7]:
```python
# Test that we project x on to the 1d subspace correctly
assert_allclose(
    project_1d(np.ones(3), np.array([1, 2, 2])),
    np.array([5, 10, 10]) / 9
)

# Some hidden tests below
```

Projection (ND)

You may find the function np.linalg.inv() useful.

In [9]:
```python
# GRADED FUNCTION: DO NOT EDIT THIS LINE
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
    """
    # YOUR CODE HERE
    ### Uncomment and modify the code below
    P = np.eye(B.shape[0])
    P = B @ (np.linalg.inv(B.T @ B)) @ B.T
    return P
```

In [10]:
```python
# GRADED FUNCTION: DO NOT EDIT THIS LINE
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)
    """
    # YOUR CODE HERE
    # Uncomment and modify the code below
    # <-- EDIT THIS
    P = projection_matrix_general(B) @ x
    return P
```

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 [11]:
```python
from numpy.testing import assert_allclose

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

assert_allclose(
    projection_matrix_general(B),
    np.array([[5,  2, -1],
              [2,  2,  2],
              [-1, 2,  5]]) / 6
)

# Some hidden tests below
```

In [12]:
```
# Test 2D
# Test that we computed the correct projection matrix

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

# Some hidden tests below
```

2. Eigenfaces (optional)

Next, you will see what happens if you project a dataset consisting of human faces onto a subset of a basis we call the "eigenfaces". This technique of projecting the data onto a subspace with smaller dimension, and thus reducing the number of coordinates required to represent the data, is widely used in machine learning. It can help identify trends and relationships between variables that are otherwise hidden away.

We have already prepared the eigenfaces basis using Principle Component Analysis (PCA), which finds the best possible basis to represent this smaller subspace. Next week you will derive PCA and implement the PCA algorithm, but for now you'll simply see the results.

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

In [13]:
```
from sklearn.datasets import fetch_olivetti_faces
from ipywidgets import interact
%matplotlib inline
image_shape = (64, 64)
# Load faces data
dataset = fetch_olivetti_faces(data_home='./')
faces = dataset.data
```

Each face of the dataset is a gray scale image of size (64, 64). Let's visualize some faces in the dataset.

In [14]:
```
plt.figure(figsize=(10,10))
plt.imshow(np.hstack(faces[:5].reshape(5,64,64)), cmap='gray');
```