Week 3 Assessment: Orthogonal Projections

Learning Objectives

In this week, we will write functions which perform orthogonal projections.

By the end of this week, you should be able to

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

We highlight some tips and tricks which would be useful when you implement numerical algorithms that you have never encountered before. You are invited to think about these concepts when you write your program.

The important thing is to learn to map from mathematical equations to code. It is not always easy to do so, but you will get better at it with more practice.

We will apply this to project high-dimensional face images onto lower dimensional basis which we call "eigenfaces". We will also revisit the problem of linear regression, but from the perspective of solving normal equations, the concept which you apply to derive the formula for orthogonal projections. We will apply this to predict housing prices for the Boston housing dataset, which is a classic example in machine learning.

In [1]:
```
# PACKAGE: DO NOT EDIT
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight')
import numpy as np
```

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
```

```
downloading Olivetti faces from http://cs.nyu.edu/~roweis/data/olive
ttifaces.mat (http://cs.nyu.edu/~roweis/data/olivettifaces.mat) to /
home/jovyan/scikit_learn_data
```

Advice for testing numerical algorithms

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?

When you have a set of invariants, you can generate random inputs and make assertions about these invariants. This is sometimes known as fuzzing (https://en.wikipedia.org/wiki/Fuzzing), which has proven to be a very effective technique for identifying bugs in programs.

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.

1. Orthogonal Projections

Recall that for projection of a vector x onto a 1-dimensional subspace 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 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

$$\boldsymbol{B} = (\boldsymbol{b}_1 | \dots | \boldsymbol{b}_M)$$

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 $\pi_U(\boldsymbol{x}) = \boldsymbol{P}\boldsymbol{x}$.

Note that for orthogonal projections, we have the following invariants:

```
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(np.dot(p-x, p), 0)
```

In [16]:
```python
# 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,), the basis for the subspace

    Returns:
        P: the projection matrix
    """
    D, = b.shape
    P=np.zeros([D,D])
    for i in range(D):
        m=b[i]
        for j in range(D):
            n=b[j]
            k=m*n
            P[i,j]=k
    P=P/(np.transpose(b)@b)
    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,), the basis for the subspace

    Returns:
        y: projection of x in space spanned by b
    """
    p = projection_matrix_1d(b)@x
    return p

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

    Returns:
        P: the projection matrix
    """
    P = B@(np.linalg.inv(np.transpose(B)@B))@np.transpose(B)
    return P

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

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

```
In [11]: # 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)
```

```
In [17]: # Test 1D
         np_test.assert_almost_equal(projection_matrix_1d(np.array([1, 2, 2])),
                                     np.array([[1,  2,  2],
                                               [2,  4,  4],
                                               [2,  4,  4]]) / 9)

         np_test.assert_almost_equal(project_1d(np.ones(3),
                                                np.array([1, 2, 2])),
                                     np.array([5, 10, 10]) / 9)

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

         # Test General
         np_test.assert_almost_equal(projection_matrix_general(B),
                                     np.array([[5,  2, -1],
                                               [2,  2,  2],
                                               [-1, 2,  5]]) / 6)

         np_test.assert_almost_equal(project_general(np.array([6, 0, 0]), B),
                                     np.array([5, 2, -1]))
         print('correct')
```

```
correct
```

```
In [ ]: # Write your own test cases here, use random inputs, utilize the invar
```

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".

```
In [ ]: 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
```

In [ ]:
```
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.py`, first we load it into the variable B.

In [ ]:
```
B = np.load('eigenfaces.npy')[:50] # we use the first 50 dimensions of
print("the eigenfaces have shape {}".format(B.shape))
```

Along the first dimension of B, each instance is a `64x64` image, an "eigenface". Let's visualize a few of them.

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

Take a look at what happens if we project our faces onto the basis spanned by these "eigenfaces". This requires us to reshape B into the same shape as the matrix representing the basis as we have done earlier. 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 [ ]:
```
@interact(i=(0, 10))
def show_eigenface_reconstruction(i):
    original_face = faces_normalized[i].reshape(64, 64)
    # project original_face onto the vector space spanned by B_basis,
    # you should take advantage of the functions you have implemented
    # to perform the projection. First, reshape B such that it represe
    # for the eigenfaces. Then perform orthogonal projection which wou
    # `face_reconstruction`.
    B_basis = ???
    face_reconstruction = ???
    plt.figure()
    plt.imshow(np.hstack([original_face, face_reconstruction]), cmap='
    plt.show()
```

Question:

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

Modify the code above to visualize it.

3. Least square for predicting Boston housing prices (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 collect the features in 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 of n datapoints $\boldsymbol{x}_i$ in a data matrix $\boldsymbol{X}$, we can write down our model like this:

$$\begin{bmatrix} \boldsymbol{x}_1^T \\ \vdots \\ \boldsymbol{x}_n^T \end{bmatrix} \boldsymbol{\theta} = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}.$$

That is,

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

where $\boldsymbol{y}$ collects all house prices $y_1, \ldots, y_n$ of the training set.

Our goal is to find the best $\boldsymbol{\theta}$ that minimizes the following (least squares) objective:

$$\begin{aligned} &\sum_{i=1}^{n} \|\boldsymbol{\theta}^T\boldsymbol{x}_i - y_i\|^2 \\ &= (\boldsymbol{X}\boldsymbol{\theta} - \boldsymbol{y})^T(\boldsymbol{X}\boldsymbol{\theta} - \boldsymbol{y}). \end{aligned}$$

Note that we aim to minimize the squared error between the prediction $\boldsymbol{\theta}^T\boldsymbol{x}_i$ of the model and the observed data point y_i in the training set.

To find the optimal (maximum likelihood) parameters $\boldsymbol{\theta}^*$, we set the gradient of the least-squares objective to $\mathbf{0}$:

$$\begin{aligned} \nabla_{\boldsymbol{\theta}}(\boldsymbol{X}\boldsymbol{\theta} - \boldsymbol{y})^T(\boldsymbol{X}\boldsymbol{\theta} - \boldsymbol{y}) &= \mathbf{0} \\ \iff \nabla_{\boldsymbol{\theta}}(\boldsymbol{\theta}^T\boldsymbol{X}^T - \boldsymbol{y}^T)(\boldsymbol{X}\boldsymbol{\theta} - \boldsymbol{y}) &= \mathbf{0} \\ \iff \nabla_{\boldsymbol{\theta}}(\boldsymbol{\theta}^T\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{\theta} - \boldsymbol{y}^T\boldsymbol{X}\boldsymbol{\theta} - \boldsymbol{\theta}^T\boldsymbol{X}^T\boldsymbol{y} + \boldsymbol{y}^T\boldsymbol{y}) &= \mathbf{0} \\ \iff 2\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{\theta} - 2\boldsymbol{X}^T\boldsymbol{y} &= \mathbf{0} \\ \iff \boldsymbol{X}^T\boldsymbol{X}\boldsymbol{\theta} &= \boldsymbol{X}^T\boldsymbol{y}. \end{aligned}$$

The solution,\boldsymbol which gives zero gradient solves the **normal equation**

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

If you recall from the lecture on projection onto n-dimensional subspace, this is exactly the same as the normal equation we have for projection (take a look at the notes here (https://www.coursera.org/teach/mathematics-machine-learning-pca/content/edit/supplement/fQq8T/content) if you don't remember them).

Let's put things into perspective and try to find the best parameter θ^* of the line $y = \theta x$, where $x, \theta \in \mathbb{R}$ for a given a training set $\boldsymbol{X} \in \mathbb{R}^n$ and $\boldsymbol{y} \in \mathbb{R}^n$.

Note that in our example, the features x_i are only scalar, such that the parameter θ is also only a scalar. The derivation above holds for general parameter vectors (not only for scalars).

Note: This is exactly the same problem as linear regression which was discussed in Mathematics for Machine Learning: Multivariate Calculus (https://www.coursera.org/teach/multivariate-calculus-machine-learning/content/edit/lecture/74ryq/video-subtitles). However, rather than finding the optimimal θ^* with gradient descent, we can solve this using the normal equation.

In [ ]:
```
x = np.linspace(0, 10, num=50)

random = np.random.RandomState(42)  # we use the same random seed so w
theta = random.randn()              # we use a random theta, our goal
y = theta * x + random.rand(len(x)) # our theta is corrupted by some n

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

In [ ]:
```
X = x.reshape(-1,1)
Y = y.reshape(-1,1)

theta_hat = np.linalg.solve(X.T @ X,
                            X.T @ Y)
```

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

In [ ]:
```
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)
```

What would happen to $\|\theta^* - \theta\|$ if we increased the number of datapoints?

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

In [ ]:
```
N = np.arange(10, 10000, step=10)
# Your code here which calculates θ* for different sample size.
```

We see how we can find the best θ. In fact, we can extend our methodology to higher dimensional dataset. Let's now try applying the same methodology to the boston housing prices dataset.

In [ ]:
```
from sklearn.datasets import load_boston
boston = load_boston()
boston_X, boston_y = boston.data, boston.target
print("The housing dataset has size {}".format(boston_X.shape))
print("The prices has size {}".format(boston_X.shape))

boston_theta_hat = np.zeros(3) ## EDIT THIS to predict boston_theta_ha
```