Homework 4

Linear Algebra and Data Modeling

This notebook is arranged in cells. Texts are usually written in the markdown cells, and here you can use html tags (make it bold, italic, colored, etc). You can double click on this cell to see the formatting.

The ellipsis (...) are provided where you are expected to write your solution but feel free to change the template (not over much) in case this style is not to your taste.

Hit "Shift-Enter" on a code cell to evaluate it. Double click a Markdown cell to edit.

Link Okpy

In [ ]:
```
from client.api.notebook import Notebook
ok = Notebook('hw4_U.ok')
_ = ok.auth(inline = True)
```

Imports

In [3]:
```
import numpy as np
from scipy.integrate import quad
import sklearn as sk
from sklearn import datasets, linear_model
from sklearn.preprocessing import PolynomialFeatures
#For plotting
import matplotlib.pyplot as plt
%matplotlib inline
```

Problem 1 - Solving Least Squares Using Normal Equations and SVD

(Reference - NR 15.4) We fit a set of 50 data points (x_i, y_i) to a polynomial $y(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3$. (Note that this problem is linear in a_i but nonlinear in x_i). The uncertainty σ_i associated with each measurement y_i is known, and we assume that the x_i's are known exactly. To measure how well the model agrees with the data, we use the chi-square merit function:

$$\chi^2 = \sum_{i=0}^{N-1} \left(\frac{y_i - \sum_{k=0}^{M-1} a_k x^k}{\sigma_i} \right)^2.$$

where N = 50 and M = 4. Here, $1, x, \ldots, x^3$ are the basis functions.

1. Plot data (make sure to include error bars). (Hint - https://matplotlib.org/api/_as_gen/matplotlib.axes.Axes.errorbar.html (https://matplotlib.org/api/_as_gen/matplotlib.axes.Axes.errorbar.html))

In [ ]:
```
# Load a given 2D data
data = np.loadtxt("HW4_Problem2_data.dat")
x = data[:,0]
y = data[:,1]
sig_y = data[:,2]
```

In [ ]:
```
# Make plot
plt.figure(figsize = (10, 7))
# Scatter plot
...
```

We will pick as best parameters those that minimize χ^2.

First, let $\mathbf{A}$ be a matrix whose $N \times M$ components are constructed from the M basis functions evaluated at the N abscissas x_i, and from the N measurement errors σ_i, by the prescription

$$A_{ij} = \frac{X_j(x_i)}{\sigma_i}$$

where $X_0(x) = 1,\ X_1(x) = x,\ X_2(x) = x^2,\ X_3(x) = x^3$. We call this matrix $\mathbf{A}$ the design matrix.

Also, define a vector $\mathbf{b}$ of length N by

$$b_i = \frac{y_i}{\sigma_i}$$

and denote the M vector whose components are the parameters to be fitted (a_0, a_1, a_2, a_3) by $\mathbf{a}$.

2. Define the design matrix A. (Hint: Its dimension should be NxM = 50x4.) Also, define the vector b.

In [ ]:
```
# Define A
A = ...
# Define b
b = ...
```

Minimize χ^2 by differentiating it with respect to all M parameters a_k vaishes. This condition yields the matrix equation

$$\sum_{j=0}^{M-1} \alpha_{kj} a_j = \beta_k$$

where $\alpha = \mathbf{A}^T \cdot \mathbf{A}$ and $\beta = \mathbf{A}^T \cdot \mathbf{b}$ (α is an $M \times M$ matrix, and β is a vector of length M). This is the normal equation of the least squares problem. In matrix form, the normal equations can be written as:

$$\alpha \cdot \mathbf{a} = \beta.$$

This can be solved for the vector of parameters a by linear algebra numerical methods.

3. Define the matrix alpha and vector beta.

In [ ]:
```
# Transpose of the matrix A
A_transpose = ...

# alpha matrix
alpha = ...
# beta vector
beta = ...
```

4. We have $\alpha \cdot a = \beta$. Solve for a using (1) "GaussianElimination_pivot" defined below (2) LU decomposition and forward subsitution and backsubstitution. Plot the best-fit line on top of the data.

In [ ]:
```
def GaussianElimination_pivot(A, b):

    N = len(b)

    for m in range(N):

        # Check if A[m,m] is the largest value from elements bellow and perform swapping
        for i in range(m+1,N):
            if A[m,m] < A[i,m]:
                A[[m,i],:] = A[[i,m],:]
                b[[m,i]] = b[[i,m]]

        # Divide by the diagonal element
        div = A[m,m]
        A[m,:] /= div
        b[m] /= div

        # Now subtract from the lower rows
        for i in range(m+1,N):
            mult = A[i,m]
            A[i,:] -= mult*A[m,:]
            b[i] -= mult*b[m]

    # Backsubstitution
    x = np.empty(N,float)
    for m in range(N-1,-1,-1):
        x[m] = b[m]
        for i in range(m+1,N):
            x[m] -= A[m,i]*x[i]

    return x
```

In [ ]:
```
# Using the Gaussian elimination with partial pivoting
a = ...

print('Using Gaussian Elimination:')
print('a0 =', a[0], ', a1 =', a[1], ', a2 =', a[2], ', a3 =', a[3])
```

In [ ]:
```
# "lu" does LU decomposition with pivot. Reference - https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.linalg.lu_factor.html
from scipy.linalg import lu

def solve_lu_pivot(A, B):
    # LU decomposition with pivot
    L, U = lu(A, permute_l=True)

    N = len(B)

    # forward substitution: We have Ly = B. Solve for y
    ...

    # backward substitution: We have y = Ux. Solve for x.
    ...

    return ...

a = ...

print('Using LU Decomposition:')
print('a0 =', a[0], ', a1 =', a[1], ', a2 =', a[2], ', a3 =', a[3])
```

In [ ]:
```
# Make plot
...
```

The inverse matrix $\mathbf{C} = \alpha^{-1}$ is called the covariance matrix, which is closely related to the probable uncertainties of the estimated parameters $\mathbf{a}$. To estimate these uncertainties, we compute the variance associated with the estimate a_j. Following NR p.790, we obtain:

$$\sigma^2(a_j) = \sum_{k=0}^{M-1} \sum_{l=0}^{M-1} C_{jk} C_{jl} \alpha_{kl} = C_{jj}$$

5. Compute the error (standard deviation - square root of the variance) on the fitted parameters.

In [ ]:
```
from scipy.linalg import inv
# Covariance matrix

...

sigma_a0 = ...
sigma_a1 = ...
sigma_a2 = ...
sigma_a3 = ...

print('Error: on a0 =', sigma_a0, ', on a1 =', sigma_a1, ', on a2 =', sigma_a2, ', on a3 =', sigma_a3)
```

Now, instead of using the normal equations, we use singular value decomposition (SVD) to find the solution of least squares. Please read Ch. 15 of NR for more details. Remember that we have the $N \times M$ design matrix $\mathbf{A}$ and the vector $\mathbf{b}$ of length N. We wish to mind $\mathbf{a}$ which minimizes $\chi^2 = |\mathbf{A} \cdot \mathbf{a} - \mathbf{b}|^2$.

Using SVD, we can decompose $\mathbf{A}$ as the product of an $N \times M$ column-orthogonal matrix $\mathbf{U}$, an $M \times M$ diagonal matrix $\mathbf{S}$ (with positive or zero elements - the "singular" values), and the transpose of an $M \times M$ orthogonal matrix $\mathbf{V}$. ($\mathbf{A} = \mathbf{U}\mathbf{S}\mathbf{V}^T$).
Let $\mathbf{U}_{(i)}$ and $\mathbf{V}_{(i)}$ denote the columns of $\mathbf{U}$ and $\mathbf{V}$ respectively (Note: We get M number of vectors of length M.) $\mathbf{S}_{(i,i)}$ are the ith diagonal elements (singular values) of $\mathbf{S}$. Then, the solution of the above least squares problem can be written as:

$$\mathbf{a} = \sum_{i=1}^{M} \left(\frac{\mathbf{U_{(i)}} \cdot \mathbf{b}}{\mathbf{S_{(i,i)}}}\right)\mathbf{V_{(i)}}.$$

The variance in the estimate of a parameter a_j is given by:

$$\sigma^2(a_j) = \sum_{i=1}^{M} \left(\frac{V_{ji}}{S_{ii}}\right)^2$$

and the covariance:

$$\mathrm{Cov}(a_j, a_k) = \sum_{i=1}^{M} \left(\frac{V_{ji}V_{ki}}{S_{ii}^2}\right).$$

6. Decompose the design matrix A using SVD. Estimate the parameter a_i's and its variance.

In [ ]:
```
# Reference - https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.linalg.svd.html
from scipy.linalg import svd

# Decompose A
# Note: S, in this case, is a vector of length M, which contains the singular values.
U, S, VT = svd(A, full_matrices=False)
V = VT.T

# Solve for a
...
a_from_SVD = ...

print('Using SVD:')
print('a0 =', a_from_SVD[0], ', a1 =', a_from_SVD[1], ', a2 =', a_from_SVD[2], ', a3 =', a_from_SVD[3])
```

In [ ]:
```
# Error on a
...
sigma_a_SVD = ...


print('Error: on a0 =', sigma_a_SVD[0], ', on a1 =', sigma_a_SVD[1], ', on a2 =', sigma_a_SVD[2], ', on a3 =', sigma_a_SVD[3])
```

Suppose that you are only interested in the parameters a_0 and a_1. We can plot the 2-dimensional confidence region ellipse for these parameters by building the covariance matrix:

$$C' = \begin{pmatrix} \sigma(a_0)^2 & \mathrm{Cov}(a_0, a_1) \\ \mathrm{Cov}(a_0, a_1) & \sigma(a_1)^2 \end{pmatrix}$$

The lengths of the ellipse axes are the square root of the eigenvalues of the covariance matrix, and we can calculate the counter-clockwise rotation of the ellipse with the rotation angle:

$$\theta = \frac{1}{2}\arctan\left(\frac{2 \cdot \mathrm{Cov}(a_0, a_1)}{\sigma(a_0)^2 - \sigma(a_1)^2}\right)$$

Then, we multiply the axis lengths by some factor depending on the confidence level we are interested in. For 68%, this scale factor is $\sqrt{\Delta\chi^2} \approx 1.52$.

7. Compute the covariance between a_0 and a_1. Plot the 68% confidence region of the parameter a_0 and a_1.

In [ ]:
```
# Compute the covariance
...
sigma_01 = ...

# Build the covariance matrix
CovM = np.array([[sigma_a_SVD[0], sigma_01],[sigma_01, sigma_a_SVD[1]]])

from numpy.linalg import eigvals
axis1 = 1.52*eigvals(CovM)[0]
axis2 = 1.52*eigvals(CovM)[1]

theta = np.arctan(2*sigma_01/(sigma_a_SVD[0]**2-sigma_a_SVD[1]**2))/2.
```

In [ ]:
```
# Plot the 1-sigma confidence region (https://stackoverflow.com/questions/32371996/python-matplotlib-how-to-plot-ellipse)
from matplotlib.patches import Ellipse
import matplotlib as mpl

ell = mpl.patches.Ellipse(xy=[a[0], a[1]], width=axis1, height=axis2, angle = theta*180/np.pi)
fig, ax = plt.subplots(figsize=(7,7))

ax.add_patch(ell)
ax.set_aspect('equal')
ax.autoscale()
plt.grid(True)
plt.xlabel('$a_0$')
plt.ylabel('$a_1$')
plt.show()
```

In lecture, we discussed that we fit the existing data to obtain model parameters in data analysis, while in machine learning we use the model derived from the existing data to make prediction for new data.

Next, let us take the given data and do the polynomial regression.

First, split the sample into training data and the testing data. Keep 80% data as training data and uses the remaining 20% data for testing.

8. Often, the data can be ordered in a specific manner, hence shuffle the data prior to splitting it into training and testing samples. (Use https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.shuffle.html (https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.shuffle.html))

In [ ]:
```
...

train_x = ...
train_y = ...
train_sigy = ...

test_x = ...
test_y = ...
test_sigy = ...
```

Many algorithms are also not scale invariant, and hence we need to scale the data (different features to a uniform scale). All this comes under preprocessing the data. http://scikit-learn.org/stable/modules/preprocessing.html#preprocessing (http://scikit-learn.org/stable/modules/preprocessing.html#preprocessing) We can use StandardScaler from sklearn (or write your own routine) to center the data to 0 mean and 1 variance. Note that you only center the training data and then use its mean and variance to scale the testing data before using it.

In the case of polynomial regression, we need to generate polynomial features (http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.PolynomialFeatures.html (http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.PolynomialFeatures.html)) for preprocessing. Note that we call each term in the polynomial as a "feature" in our model, and here we generate features' high-order (and interaction) terms. For example, suppose we set the degree of the polynomial to be 3. Then, the features of X is transformed from (X) to $(1, X, X^2, X^3)$. We can do this transform using PolynomialFeatures.fit_transform(train_x). But fit_transform() takes the numpy array of shape [n_samples, n_features]. So you need to re-define our training set as train_set_prep = train_x[:,np.newaxis] so that it has the shape [40,1].

9. Define three different polynomial models with degree of 1, 3, 10. (e.g. model = PolynomialFeatures(degree=...)) Then, fit to data and transform it using "fit_transform"

In [1]:
```
# e.g.
# model = PolynomialFeatures(degree = ...)
# X_model = model.fit_transform(train_x[:,np.newaxis])

...
```

Then, do the least squares linear regression. (http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html#sklearn.linear_model.LinearRegression.fit (http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html#sklearn.linear_model.LinearRegression.fit))

1. define the object for linear regression: LR = linear_model.LinearRegression()
2. Fit the linear model to the training data: LR.fit(transformed x data, y data)
3. Define new x samples for plotting: X_sample = np.linspace(-5, 7, 100)
4. Transform x sample: X_sample_transform = model.fit_transform(X_sample[:,np.newaxis])
5. Predict using the linear model: Y_sample = LR.predict(X_sample_transform)
6. Plot the fit: plt.plot(X_sample, Y_sample)

10. Do the linear regression for three different polynomial models defined in Part 9. Plot the fit on top of the training data (Label each curve).

In [ ]:
```
...
```

In [ ]:
```
# Make plot
plt.figure(figsize = (10, 7))

...

plt.xlabel('$x$')
plt.ylabel('$y$')
plt.xlim(-5, 7)
plt.ylim(-12, 65)
plt.legend()
plt.show()
```

11. Plot the fit on top of the test data (Label each curve).

In [ ]:
```
# Make plot
plt.figure(figsize = (10, 7))

...

plt.xlabel('$x$')
plt.ylabel('$y$')
plt.xlim(-5, 7)
plt.ylim(-12, 65)
plt.legend()
plt.show()
```

You can obtain the estimated linear coefficients using linearmodel.LinearRegression.coef (http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html#sklearn.linear_model.LinearRegression (http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html#sklearn.linear_model.LinearRegression))

12. Print the linear coefficients of three polynomial models you used. For the polynomial of degree 10, do you see that high-order coefficients are very small?

In [ ]:
```
...
```

Problem 2 - Applying the PCA Method on Quasar Spectra

The following analysis is based on https://arxiv.org/pdf/1208.4122.pdf (https://arxiv.org/pdf/1208.4122.pdf).

"Principal Component Analysis (PCA) is a powerful and widely used technique to analyze data by forming a custom set of “principal component” eigenvectors that are optimized to describe the most data variance with the fewest number of components. With the full set of eigenvectors the data may be reproduced exactly, i.e., PCA is a transformation which can lend insight by identifying which variations in a complex dataset are most significant and how they are correlated. Alternately, since the eigenvectors are optimized and sorted by their ability to describe variance in the data, PCA may be used to simplify a complex dataset into a few eigenvectors plus coefficients, under the approximation that higher-order eigenvectors are predominantly describing fine tuned noise or otherwise less important features of the data." (S. Bailey, arxiv: 1208.4122)

In this problem, we take the quasar (QSO) spectra from the Sloan Digital Sky Survey (SDSS) and apply PCA to them. Filtering for high S/N in order to apply the standard PCA, we select 18 high-S/N spectra of QSOs with redshift 2.0 < z < 2.1, trimmed to $1340 < \lambda < 1620$ Å.

In [ ]:
```
# Load data
wavelength = np.loadtxt("HW5_Problem2_wavelength.txt")
flux = np.loadtxt("HW5_Problem2_QSOspectra.txt")
```

In [ ]:
```
# Data dimension
print( np.shape(wavelength) )
print( np.shape(flux) )
```

In the above cell, we load the following data: wavelength in Angstroms ("wavelength") and 2D array of spectra x fluxes ("flux").

We have 824 wavelength bins, so "flux" is 18 $\times$ 824 matrix, each row containing fluxes of different QSO spectra.

1. Plot any three QSO spectra flux as a function of wavelength. (In order to better see the features of QSO spectra, you may plot them with some offsets.)

In [ ]:
```
...
```

"Flux" is the data matrix of order 18 $\times$ 824. Call this matrix $\mathbf{X}$.

We can construct the covariance matrix $\mathbf{C}$ using the mean-centered data matrix. First, calculate the mean of each column and subtracts this from the column. Let $\mathbf{X_c}$ denote the mean-centered data matrix.

$$\mathbf{X_c} = \begin{bmatrix} x_{(1,1)} - \bar{x}_1 & x_{(1,2)} - \bar{x}_2 & \cdots & x_{(1,824)} - \bar{x}_{824} \\ x_{(2,1)} - \bar{x}_1 & x_{(2,2)} - \bar{x}_2 & \cdots & x_{(2,824)} - \bar{x}_{824} \\ \vdots & \vdots & \vdots & \vdots \\ x_{(18,1)} - \bar{x}_1 & x_{(18,2)} - \bar{x}_2 & \cdots & x_{(18,824)} - \bar{x}_{824} \end{bmatrix}$$

where $x_{m,n}$ denote the flux of mth QSO in nth wavelength bin, and $\bar{x}_k$ is the mean flux in kth wavelength bin.

Then, the covariance matrix is: $\mathbf{C} = \frac{1}{N-1}\mathbf{X_c^T X_c}$. ($N$ is the number of QSOs.)

2. Find the covariance matrix C using the data matrix flux.

In [ ]:
```
C =
...
```

3. Using numpy.linalg, find eigenvalues and eigenvectors of the covariance matrix. Order the eigenvalues from largest to smallest and then plot them as a function of the number of eigenvalues. (Remember that the eigenvector with the highest eigenvalue is the principle component of the data set.) In this case, we find that our covariance matrix is rank-17 matrix, so we only select the first 17 highest eigenvalues and corresponding eigenvectors (other eigenvalues are close to zero).

In [ ]:
```
from numpy.linalg import eig
...
```

In [ ]:
```
# Make plot
...
```

4. Plot the first three eigenvectors. These eigenvectors represent the principal variations of the spectra with respect to that mean spectrum.

In [ ]:
```
...
```

The eigenvectors indicate the direction of the principal components, so we can re-orient the data onto the new zes by multiplying the original mean-centered data by the eigenvectors. We call the re-oriented data "PC scores." (Call the PC score matrix $\mathbf{Z}$) Suppose that we have k eigenvectors. Construct the matrix of eigenvectors $\mathbf{V} = [\mathbf{v_1 v_2 ... v_k}]$, with $\mathbf{v_i}$ the ith highest eigenvector. Then, we can get 18 $\times$ k PC score matrix by multiplying the 18 $\times$ 824 data matrix with the 824 $\times$ k eigenvector matrix:

$$\mathbf{Z} = \mathbf{X_c V}$$

Then, we can reconstruct the data by mapping it back to 824 dimensions with $\mathbf{V^T}$:

$$\hat{\mathbf{X}} = \mu + \mathbf{Z V^T}$$

where μ is the vector of mean QSO flux.

Now, comparing the original data with the reconstructed data, we can calculate the residuals. Let $\mathbf{X_{(i)}}, \hat{\mathbf{X}}_{(i)}$ denote the rows of $\mathbf{X}, \hat{\mathbf{X}}$ respectively. Remember that the data matrix has the dimension 18 $\times$ 824, so each row $\mathbf{X_{(i)}}$ corresponding the spectra of one particular QSO. (For example, if you wish to see the QSO spectra in row 7, you can plot $\mathbf{X_{(7)}}$ as a function of wavelength.). Then, we can simply calculate the residual as $\frac{1}{N}\sum_{i=1}^{N} |\hat{\mathbf{X}}_{(i)} - \mathbf{X_{(i)}}|^2$ where N is the total number of QSOs (NOTE: $|\hat{\mathbf{X}}_{(i)} - \mathbf{X_{(i)}}|$ is the magnitude of the difference between two vectors $\hat{\mathbf{X}}_{(i)}$ and $\mathbf{X_{(i)}}$.)

5. First, start with only mean flux value μ (in this case $\hat{X} = \mu, V = 0$) and calculate the residual. Then, do the reconstruction using the first two principal eigenvectors $V = [v_1 v_2]$ and calculate the residual. Finally, let $V = [v_1 v_2 ... v_6]$ (the first six principal eigenvectors) and compute the residual.

In [ ]:
```
...
```

6. For any two QSO spectra, plot the original and reconstructed spectra using the first six principal eigenvectors.

In [ ]:
```
...
```

7. Plot the residual as a function of the number of included eigenvectors.

In [ ]:
```
...
```

In this problem, we only have 18 QSO spectra, so the idea of using PCA may seem silly. We can also use SVD to find eigenvalues and eigenvectors. With SVD, we get $\mathbf{X_c} = \mathbf{USV^T}$. Then, the covariance matrix is $\mathbf{C} = \frac{1}{N-1}\mathbf{X_c^T X_c} = \frac{1}{N-1}\mathbf{VS^2V^T}$. Then, the eigenvalues are the squared singular values scaled by the factor $\frac{1}{N-1}$ and the eigenvectors are the columns of $\mathbf{V}$.

8. Find the eigenvalues applying SVD to the mean-centered data matrix X_c.

In [ ]:
```
from scipy.linalg import svd

...

# Print Eigenvalues
...
```

To Submit

Execute the following cell to submit. If you make changes, execute the cell again to resubmit the final copy of the notebook, they do not get updated automatically.
We recommend that all the above cells should be executed (their output visible) in the notebook at the time of submission.
Only the final submission before the deadline will be graded.

In [ ]:
```
1 _ = ok.submit()
```