bayesian-analysis (/github/christianhbye/bayesian-analysis/tree/main)
/ homeworks (/github/christianhbye/bayesian-analysis/tree/main/homeworks)

Open in Colab
(https://colab.research.google.com/github/christianhbye/bayesian-analysis/blob/main/homeworks/HW3_288.ipynb)

Homework 3

Linear Algebra - Gaussian Elimination, SVD, Polynomial Regression, PCA, KNN, 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.

Imports

In [1]:

```
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
```

Mounting Google Drive locally

Mount your Google Drive on your runtime using an authorization code.

Note: When using the 'Mount Drive' button in the file browser, no authentication codes are necessary for notebooks that have only been edited by the current user.

In [ ]:

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

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. (Hint - https://matplotlib.org/api/_as_gen /matplotlib.axes.Axes.errorbar.html)

Again, in this and all future assignments, you are expected to show error bars in all figures if the data include uncertainties. You will lose points if error bars are not shown.

In [ ]:

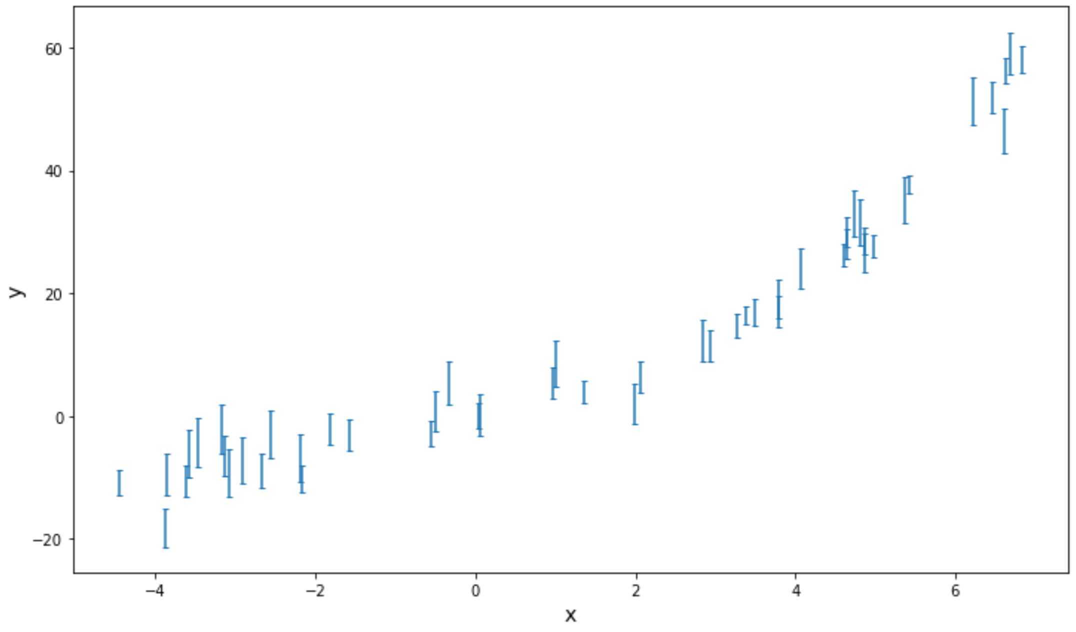
```
hw_path = "drive/MyDrive/P188_288/P188_288_HW3/"
```

In [ ]:

```
# Load a given 2D data
data = np.loadtxt(hw_path + "Problem1_data.dat")
x = data[:,0]
y = data[:,1]
sig_y = data[:,2]
```

In [ ]:

```
plt.figure(figsize=(12,7))
plt.errorbar(x, y, yerr=sig_y, marker="o", fmt="none", capsize=2
plt.xlabel("x", fontsize=14)
plt.ylabel("y", fontsize=14)
plt.show()
```



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. Print the first row of A.

In [ ]:
```
A = x.reshape(-1, 1) ** np.arange(4).reshape(1, -1)  # numerator
A /= sig_y.reshape(-1, 1)  # divide by y error

b = y / sig_y

print(A.shape)
print(A[0])
```

```
(50, 4)
[0.30368985 0.60162612 1.1918541  2.36112786]
```

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 $\boldsymbol{\alpha} = \mathbf{A^T} \cdot \mathbf{A}$ and $\boldsymbol{\beta} = \mathbf{A^T} \cdot \mathbf{b}$ ($\boldsymbol{\alpha}$ is an $M \times M$ matrix, and $\boldsymbol{\beta}$ 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:

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

This can be solved for the vector of parameters $\mathbf{a}$ by linear algebra numerical methods.

3. Define the matrix alpha and vector beta. Print both alpha and beta.

In [ ]:
```
alpha = A.T @ A
beta = A.T @ b

print("alpha:")
print(alpha)
print("\n\nbeta:")
print(beta)
```

```
alpha:
[[7.57344292e+00 1.59581405e+01 1.20838371e+02 4.80208969e+02]
 [1.59581405e+01 1.20838371e+02 4.80208969e+02 3.20704253e+03]
 [1.20838371e+02 4.80208969e+02 3.20704253e+03 1.62887892e+04]
 [4.80208969e+02 3.20704253e+03 1.62887892e+04 1.05149671e+05]]


beta:
[  118.53904396   727.88040211  3581.30337095 22023.93157276]
```

4. We have $\boldsymbol{\alpha} \cdot \mathbf{a} = \boldsymbol{\beta}.$ Solve for $\mathbf{a}$ using (1) GaussianElimination_pivot with pivoting (See undergrad version for hints) (2) LU decomposition and forward subsitution and backsubstitution. (https://docs.scipy.org/doc/scipy-0.14.0/reference/generated /scipy.linalg.lu_factor.html) Plot the best-fit line on top of the data.

Hint: You can use scipy.linalg.lu to do the LU decomposition. After you do "L, U = lu(A, permute_l=True)," print L and U matrices. Note that L is not a lower triangle matrix. Swap rows of L (and B) and make it a lower triangular matrix. And then, solve for y in Ly = B.

e.g. If your L matrix is the following:

[[0.01577114 0.10593754 0.41569921 1.] [0.03323166 -0.04364428 1. 0.] [0.25163705 1. 0. 0.] [1. 0. 0. 0.]],

you can change it to this:

[[1. 0. 0. 0.] [0.25163705 1. 0. 0.] [0.03323166 -0.04364428 1. 0.] [0.01577114 0.10593754 0.41569921 1.]]

Then, you should also change B from

[118.53904396 727.88040211 3581.30337095 22023.93157276]

to

[22023.93157276 3581.30337095 727.88040211 118.53904396].

In [ ]:

```
# Using the Gaussian elimination with partial pivoting

# Pivoting: move row with largest element on diagonal to the top
def gauss_elimination(A, b):
  """
  Solve the system Ax = b using Gaussian elimination with partia
  """

  # sort rows from largest to smallest
  order = np.argsort(np.diag(A))[::-1]
  A = A[order]
  b = b[order]

  nrows, ncols = A.shape

  for i in range(nrows):  # loop over rows

    # we want 1s along the diagonal, so divide by diagonal eleme
    d = A[i, i]
    A[i] /= d
    b[i] /= d

    # subtract this row from all the rows below
    for j in range(i+1, nrows):
      m = A[j, i]  # the factor to multiply the row by, this ens
      A[j] -= m * A[i]
      b[j] -= m * b[i]

  # Initialize the solution vector x as a copy of b
  x = b.copy()

  # backsubstitution: start at the bottom and go up
  for i in range(nrows-1, -1, -1):
    for j in range(i+1, ncols):
      x[i] -= A[i, j] * x[j]  # remove the variables we already

  return x


# "lu" does LU decomposition with pivot. Reference - https://doc
from scipy.linalg import lu

"""
We will do permute_l=False. This returns matrices P, L, U where
permutation matrix (reorders the rows of A to make the LU decomp
The hint concerns the case permute_l=True, in which case the pro
returned and we have to manually figure out the inversion of P.

In this case, we have A = PLU or equivalently P^-1 A = LU. Our s
Ax = b --> PLU x = b --> LU x = P^-1 b. We will thus simply solv
Ly = P^-1 b and Ux = y.
"""


def lu_pivot(A, b):
  P, L, U = lu(A, permute_l=False)

  # solve Ly = P^-1 b with forward substitution
  # note that L has 1s along the diagonal already
  y = np.linalg.inv(P) @ b
  for i in range(len(y)):
    for j in range(i):
      y[i] -= L[i, j] * y[j]  # subtract the known variables (ro

  # solve Ux = y with backward substitution
  # U does not necessarily have 1s along the diagonal
  x = y.copy()

  for i in range(len(y)-1, -1, -1):
    for j in range(i+1, len(y)):
      x[i] -= U[i, j] * x[j]
    x[i] /= U[i, i]  # divide by diagonal element

  return x
```

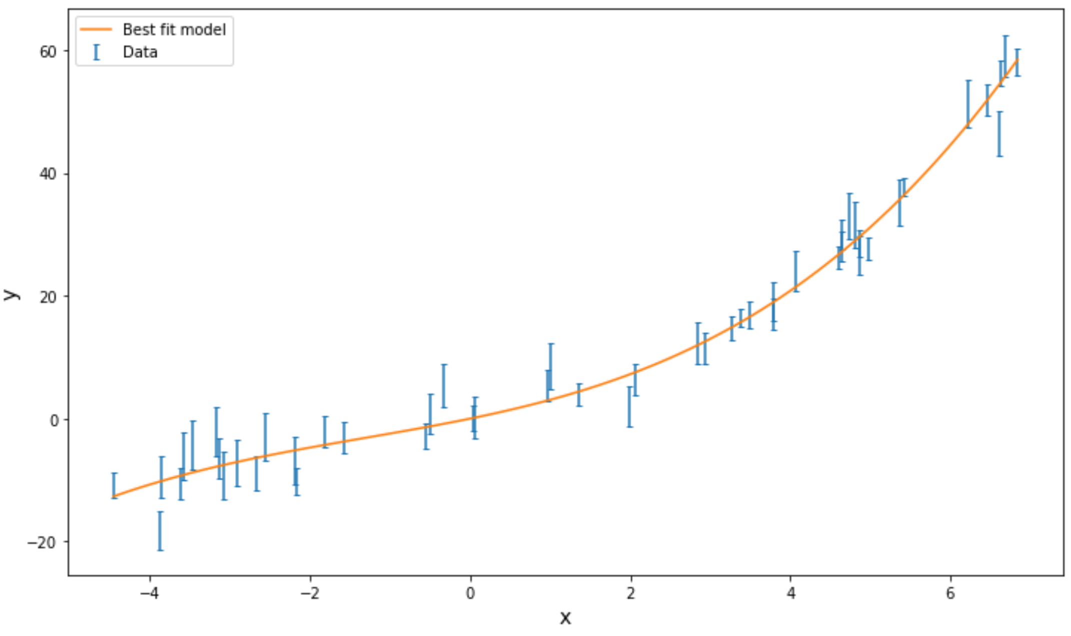
In [ ]:

```
a_ge = gauss_elimination(alpha, beta)
print("Gaussian elimination:")
for i in range(len(a_ge)):
  print(f"a_{i} = {a_ge[i]}")

print("\n")

a_lu = lu_pivot(alpha, beta)
print("LU decomposition:")
for i in range(len(a_lu)):
  print(f"a_{i} = {a_lu[i]}")
```

```
Gaussian elimination:
a_0 = -0.03081629537274111
a_1 = 2.66764608224988
a_2 = 0.3148392700785302
a_3 = 0.07945935335134485


LU decomposition:
a_0 = -0.030816295372740405
a_1 = 2.667646082249884
a_2 = 0.31483927007852985
a_3 = 0.07945935335134478
```

In [ ]:

```
# the two methods agree and are the correct solution
assert np.allclose(a_ge, a_lu)
best_params = a_ge
assert np.allclose(alpha @ best_params, beta)


def polynomial(x, a):
  """
  Evaluate a polynomial of degree N = len(a)-1 at the values x,
  polynomial coefficients
  """
  N = len(a)
  y = a.reshape(1, -1) * x.reshape(-1, 1) ** np.arange(N).reshap
  return y.sum(axis=1)

x_smooth = np.linspace(x.min(), x.max(), num=100)
best_fit = polynomial(x_smooth, best_params)

plt.figure(figsize=(12,7))
plt.errorbar(x, y, yerr=sig_y, marker="o", fmt="none", capsize=2
plt.plot(x_smooth, best_fit, c="C1", label="Best fit model")
plt.legend()
plt.xlabel("x", fontsize=14)
plt.ylabel("y", fontsize=14)
plt.show()
```



The inverse matrix $\mathbf{C} = \boldsymbol{\alpha}^{-1}$ 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 using the covariance matrix.

In [ ]:

```
cov = np.linalg.inv(alpha)  # covariance matrix
var = np.diag(cov)  # variance
err = np.sqrt(var)  # standard deviation

print("Errors on best fit parameters:")
for i in range(len(err)):
  print(f"Error on a_{i} = {err[i]:.1g}")
```

```
Errors on best fit parameters:
Error on a_0 = 0.7
Error on a_1 = 0.2
Error on a_2 = 0.06
Error on a_3 = 0.01
```

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 $\chi^2 = |\mathbf{A} \cdot \mathbf{a} - \mathbf{b}|^{\mathbf{2}}$.

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 $\mathbf{V}$. ($\mathbf{A} = \mathbf{U}\mathbf{S}\mathbf{V}^{\mathbf{T}}$).
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:

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

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/
from scipy.linalg import svd

U, S, Vt = svd(A)
V = Vt.T  # recover V

M = len(S)
a_from_SVD = np.zeros(M)
for i in range(M):
  a_from_SVD += (U[:, i] @ b / S[i]) * V[:, i]


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

# Error on a
s = (V / S) ** 2   # summand
sigma_a_SVD = np.sqrt(s.sum(axis=1))

print('Error: on a0 =', sigma_a_SVD[0], ', on a1 =', sigma_a_SVD


# check that this agrees with previous methods
assert np.allclose(best_params, a_from_SVD)
assert np.allclose(err, sigma_a_SVD)
```

```
Using SVD:
a0 = -0.030816295372704805 , a1 = 2.667646082249882 , a2 = 0.314
Error: on a0 = 0.7139418927087802 , on a1 = 0.22390274023290271
```

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:

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

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:

$$\theta = \frac{1}{2}\arctan\Big(\frac{2 \cdot \mathrm{Cov}(a_0, a_1)}{\sigma(a_0)^2 - \sigma(a_1)^2}\Big) = \arctan(\frac{\overrightarrow{v_1}(y)}{\overrightarrow{v_1}(x)})$$

where $\overrightarrow{v_1}$ is the eigenvector with the largest eigenvalue. So we calculate the angle of the largest eigenvector towards the x-axis to obtain the orientation of the ellipse.

Then, we multiply the axis lengths by some factor depending on the confidence level we are interested in. For 68%, this scale factor is $\sqrt{\Delta\chi^2} \approx 1.52$. For 95%, it is ≈ 2.48.

7. Compute the covariance between a_0 and a_1. Plot the 68% and 95% confidence region of the parameter a_0 and a_1.

In [ ]:

```
from matplotlib.patches import Ellipse
import matplotlib as mpl
from numpy.linalg import eigvals
```

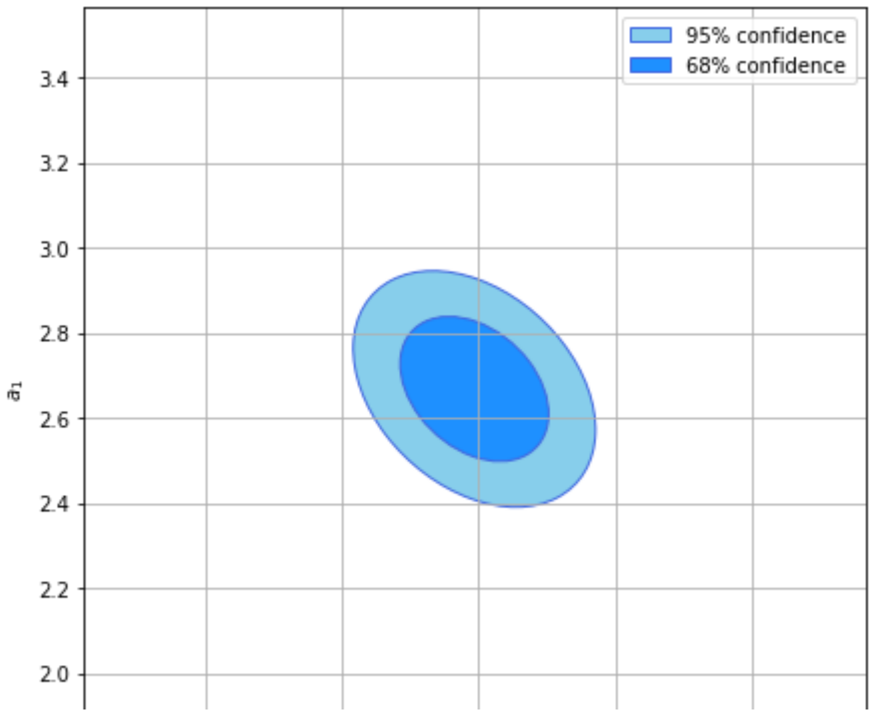
In [ ]:

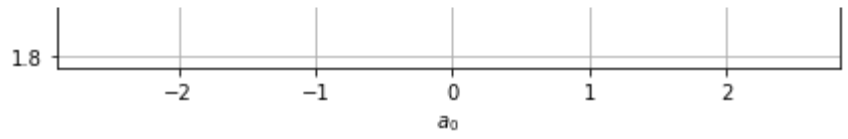
```
# Build the covariance matrix
CovM = np.empty((2, 2))

CovM[0, 0] = sigma_a_SVD[0] ** 2
CovM[1, 1] = sigma_a_SVD[1] ** 2

# compute covariance between a0 and a1
cov_a0a1 = np.sum((V[0] * V[1] / S** 2))

CovM[0, 1] = CovM[1, 0] = cov_a0a1

print(CovM)
```

```
[[ 0.50971303 -0.05496243]
 [-0.05496243  0.05013244]]
```

In [ ]:

```
# Plot the confidence region (https://stackoverflow.com/question

eigvec, eigval, u = np.linalg.svd(CovM)

# Semimajor axis (diameter)
semimaj = np.sqrt(eigval[0])
# Semiminor axis (diameter)
semimin = np.sqrt(eigval[1])

theta = np.arctan2(eigvec[0, 1], eigvec[0, 0])

# Plot 1-sig confidence region
ell = mpl.patches.Ellipse(xy=[a_from_SVD[0], a_from_SVD[1]], wid
                          height=1.52*semimin, angle = theta*180
                          facecolor = 'dodgerblue', edgecolor =
                          label = '68% confidence')
# Plot 2-sig confidence region
ell2 = mpl.patches.Ellipse(xy=[a_from_SVD[0], a_from_SVD[1]], wi
                           height=2.48*semimin, angle = theta*18
                           facecolor = 'skyblue', edgecolor = 'r
                           label = '95% confidence')

fig, ax = plt.subplots(figsize=(7,7))

ax.add_patch(ell2)
ax.add_patch(ell)


# Set bounds for x,y axes
bounds = np.sqrt(CovM.diagonal())
plt.xlim(a_from_SVD[0]-4*bounds[0], a_from_SVD[0]+4*bounds[0])
plt.ylim(a_from_SVD[1]-4*bounds[1], a_from_SVD[1]+4*bounds[1])

plt.grid(True)
plt.xlabel('$a_0$')
plt.ylabel('$a_1$')
plt.legend()
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)

In [ ]:

```
# we shuffle the indices
shuffled_ix = np.arange(len(x))
np.random.shuffle(shuffled_ix)

x_shuffled = x[shuffled_ix]
y_shuffled = y[shuffled_ix]
sig_y_shuffled = sig_y[shuffled_ix]

N_train = int(len(x) * 0.8)  # number of data points in training

x_train, x_test = x_shuffled[:N_train], x_shuffled[N_train:]
y_train, y_test = y_shuffled[:N_train], y_shuffled[N_train:]
sig_y_train, sig_y_test = sig_y_shuffled[:N_train], sig_y_shuffl
```

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

# deg = 1
model1 = PolynomialFeatures(degree=1)
X_model1 = model1.fit_transform(x_train[:, np.newaxis])

# deg = 3
model3 = PolynomialFeatures(degree=3)
X_model3 = model3.fit_transform(x_train[:, np.newaxis])

# deg = 10
model10 = PolynomialFeatures(degree=10)
X_model10 = model10.fit_transform(x_train[:, np.newaxis])
```

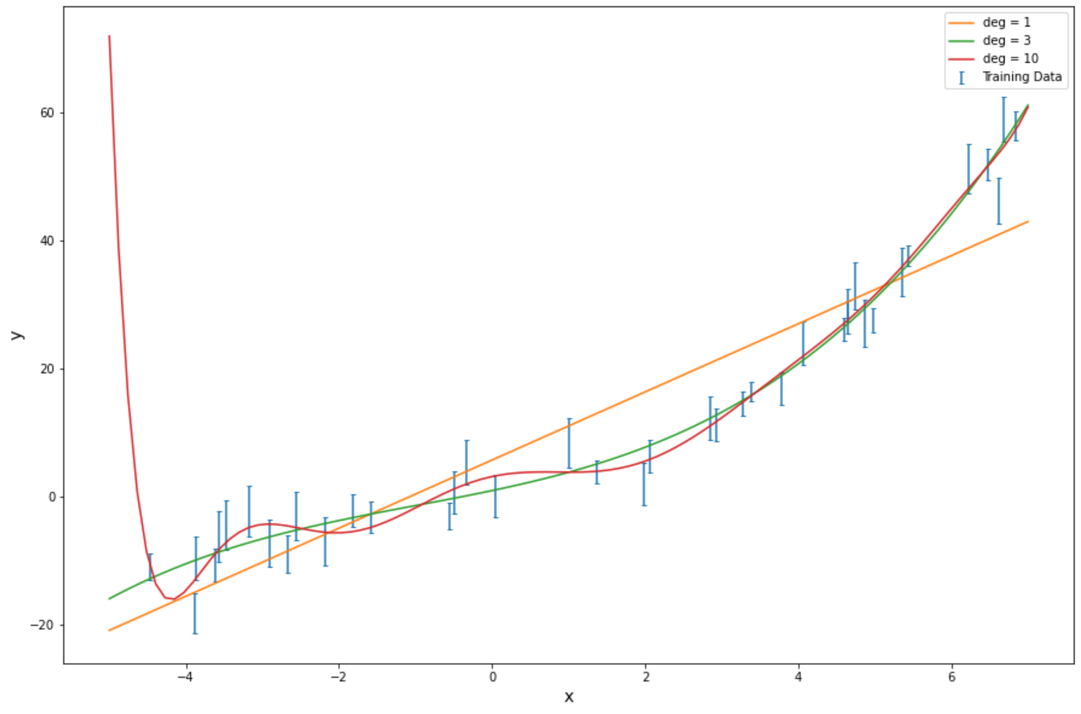
Then, do the least squares linear regression. (http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html#sklearn.linear_model.LinearRegression.fit (http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html#sklearn.linear_model.LinearRegression.fit))

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 [ ]:

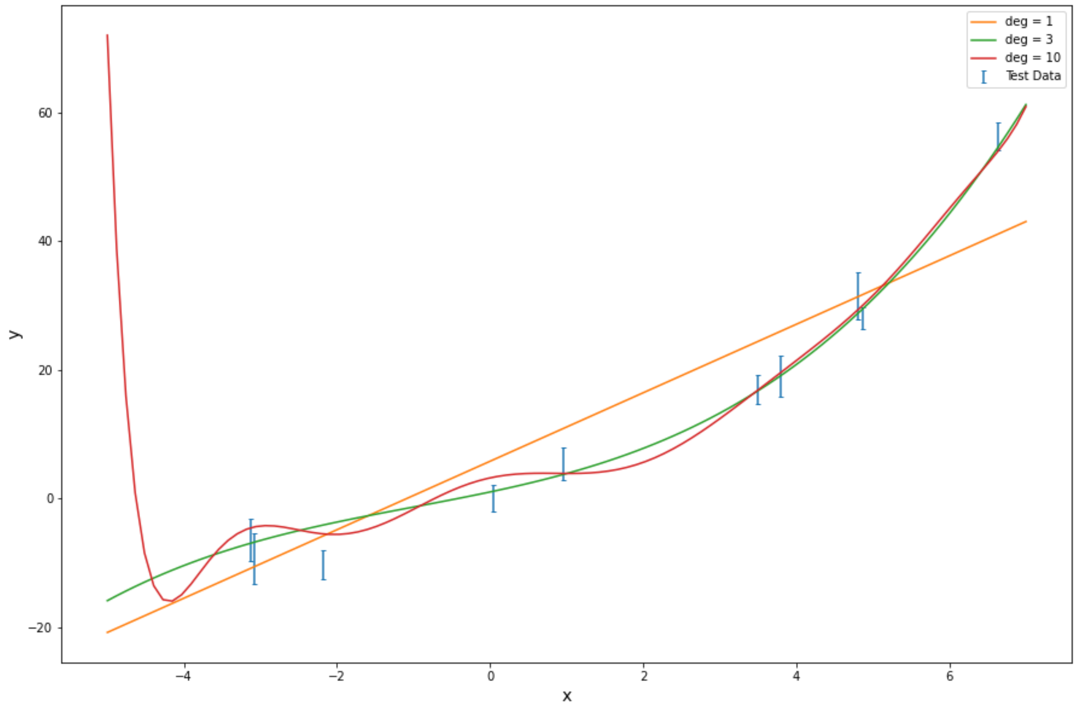
```
plt.figure(figsize=(15,10))
plt.errorbar(x_train, y_train, yerr=sig_y_train, marker="o", fmt
             capsize=2, label="Training Data")

X_sample = np.linspace(-5, 7, 100)

# deg = 1
LR1 = linear_model.LinearRegression()
LR1.fit(X_model1, y_train)
X_sample_transform1 = model1.fit_transform(X_sample[:,np.newaxis
Y_sample1 = LR1.predict(X_sample_transform1)
plt.plot(X_sample, Y_sample1, label="deg = 1")

# deg = 3
LR3 = linear_model.LinearRegression()
LR3.fit(X_model3, y_train)
X_sample_transform3 = model3.fit_transform(X_sample[:,np.newaxis
Y_sample3 = LR3.predict(X_sample_transform3)
plt.plot(X_sample, Y_sample3, label="deg = 3")

# deg = 10
LR10 = linear_model.LinearRegression()
LR10.fit(X_model10, y_train)
X_sample_transform10 = model10.fit_transform(X_sample[:,np.newax
Y_sample10 = LR10.predict(X_sample_transform10)
plt.plot(X_sample, Y_sample10, label="deg = 10")

plt.legend()
plt.xlabel("x", fontsize=14)
plt.ylabel("y", fontsize=14)
plt.show()
```



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

In [ ]:

```
plt.figure(figsize=(15,10))
plt.errorbar(x_test, y_test, yerr=sig_y_test, marker="o", fmt="n
             capsize=2, label="Test Data")


# deg = 1
plt.plot(X_sample, Y_sample1, label="deg = 1")

# deg = 3
plt.plot(X_sample, Y_sample3, label="deg = 3")

# deg = 10
plt.plot(X_sample, Y_sample10, label="deg = 10")

plt.legend()
plt.xlabel("x", fontsize=14)
plt.ylabel("y", fontsize=14)
plt.show()
```



You can obtain the estimated linear coefficients using linear*model.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 [ ]:

```
print("The linear coefficients")

# deg = 1
print("\n")
print("deg = 1:")
coeffs = LR1.coef_
for i in range(len(coeffs)):
  print(f"a_{i} = {coeffs[i]:.3g}")

# deg = 3
print("\n")
print("deg = 3:")
coeffs = LR3.coef_
for i in range(len(coeffs)):
  print(f"a_{i} = {coeffs[i]:.3g}")

# deg = 10
print("\n")
print("deg = 10:")
coeffs = LR10.coef_
for i in range(len(coeffs)):
  print(f"a_{i} = {coeffs[i]:.3g}")
```

```
The linear coefficients


deg = 1:
a_0 = 0
a_1 = 5.32


deg = 3:
a_0 = 0
a_1 = 2.52
a_2 = 0.261
a_3 = 0.0868


deg = 10:
a_0 = 0
a_1 = 2.57
a_2 = -2.69
a_3 = 0.29
a_4 = 0.631
a_5 = -0.0906
a_6 = -0.0425
a_7 = 0.00902
a_8 = 0.000604
a_9 = -0.000251
a_10 = 1.53e-05
```

The high-order coefficients of the degree 10 polynomial are very small, effectively making it similar to a lower-order polynomial.

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 $1340 < \lambda < 1620\ \mathring{A}$.

In [ ]:
```
# Load data
wavelength = np.loadtxt("/content/drive/My Drive/P188_288/P188_2
flux = np.loadtxt("/content/drive/My Drive/P188_288/P188_288_HW3
```

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

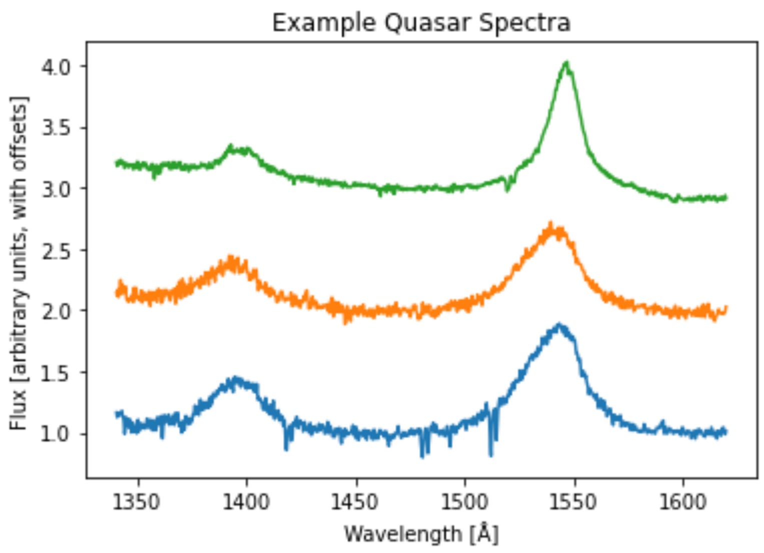
```
(824,)
(18, 824)
```

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 [ ]:
```
plt.figure()
plt.plot(wavelength, flux[0])
plt.plot(wavelength, flux[1]+1)
plt.plot(wavelength, flux[2]+2)
plt.xlabel("Wavelength [Å]")
plt.ylabel("Flux [arbitrary units, with offsets]")
plt.title("Example Quasar Spectra")
plt.show()
```



"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.

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

where $x_{m,n}$ denote the flux of mth QSO in nth wavelength bin, and $\overline{x}_k$ 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 [ ]:
```
Xc = flux - flux.mean(axis=0)[None, :]
C = Xc.T @ Xc / (len(Xc) - 1)
```

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).*

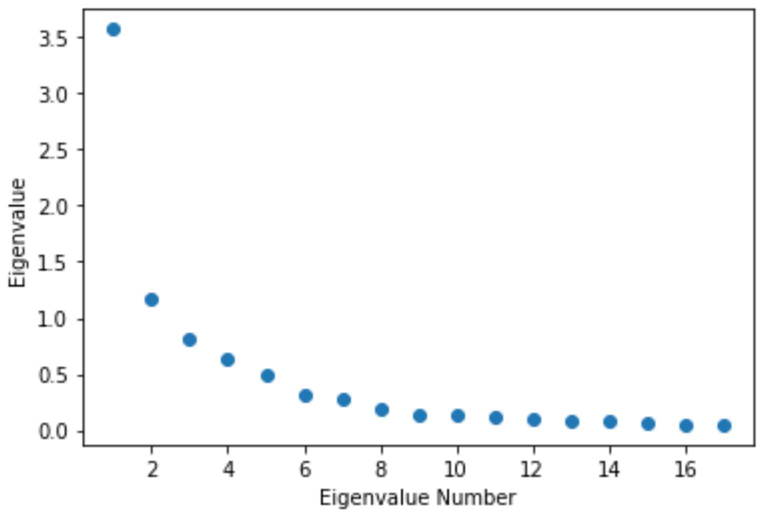
Here, by ranking the eigenvalues based on their magnitudes, you basically rank them in order of significance. You should show that the first few components are dominant, accounting for most of the variability in the data. So you can plot eigenvalues as a function of component number (1,2,3,...,17)

In [ ]:

```
# we use svd since it returns the eigenvalues sorted

from numpy.linalg import svd
u, s, vt = np.linalg.svd(C)

# select the 17 greatest magnitude eigenvalues
evals = s[:17]
evecs = vt[:17]
```

In [ ]:

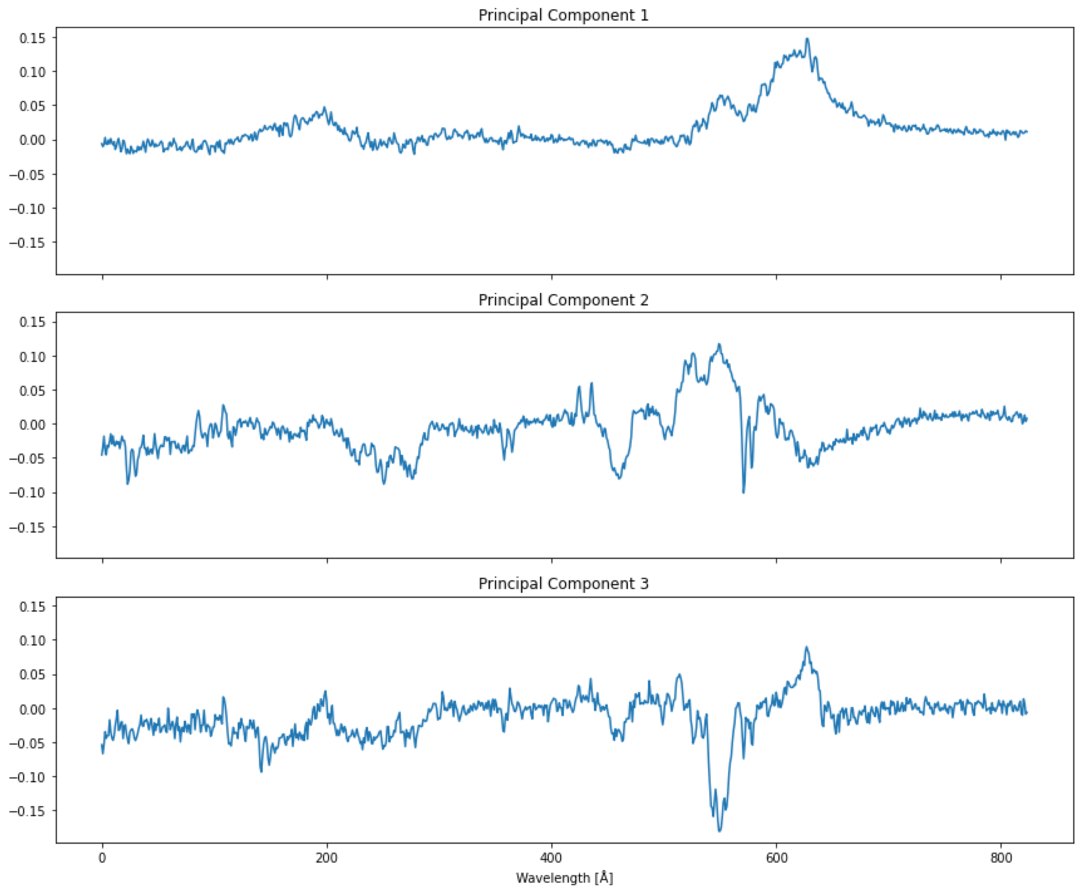
```
# Make plot
plt.figure()
plt.scatter(np.arange(len(evals))+1, evals)
plt.xlabel("Eigenvalue Number")
plt.ylabel("Eigenvalue")
plt.show()
```



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

In [ ]:

```
fig, axs = plt.subplots(figsize=(12,10), nrows=3, sharex=True, s
for i, ax in enumerate(axs.ravel()):
  ax.plot(evecs[i])
  ax.set_title(f"Principal Component {i+1}")
ax.set_xlabel("Wavelength [Å]")
plt.tight_layout()
plt.show()
```



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 $\mathbf{V^T}$:

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

where $\boldsymbol{\mu}$ is the vector of mean QSO flux.

Now, comparing the original data with the reconstructed data, we can calculate the residuals. Let $\mathbf{X_{(i)}, \hat{X}_{(i)}}$ denote the rows of $\mathbf{X, \hat{X}}$ 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 $\frac{1}{N}\sum_{i=1}^{N}|\mathbf{\hat{X}_{(i)}} - \mathbf{X_{(i)}}|^\mathbf{2}$ where N is the total number of QSOs (NOTE: $|\mathbf{\hat{X}_{(i)}} - \mathbf{X_{(i)}}|$ is the magnitude of the difference between two vectors $\mathbf{\hat{X}_{(i)}}$ and $\mathbf{X_{(i)}}$.)

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

In [ ]:

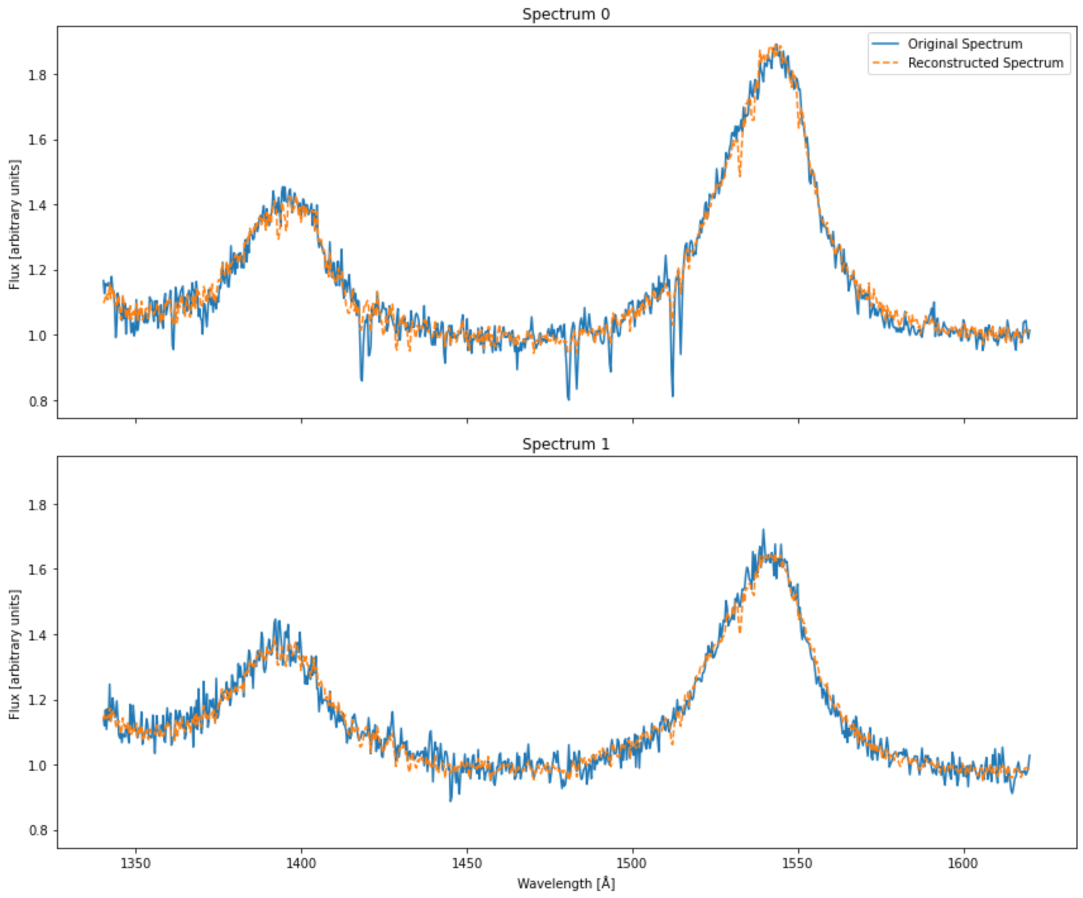
```
def rec(x, n_components, evecs):
  """
  Compute reconstructed spectra using n PCA components. Evecs ar
  principal components.
  """
  mu = x.mean(axis=0, keepdims=True)  # mu
  if n_components == 0:
    xhat = mu
  else:  # using at least one component
    V = evecs[:n_components].T  # the evecs are ordered by row,
    Xc = x - x.mean(axis=0)[None, :]  # center x
    Z = Xc @ V
    xhat = mu + Z @ V.T
  return xhat


def res(x, n_components, evecs):
  """
  Compute the residuals according to the formula above using n P
  Evecs are the sorted principal components.
  """
  xhat = rec(x, n_components, evecs)
  # compute residuals
  diff = np.sum((xhat - x) ** 2, axis=1)  # vector difference sq
  r = np.sum(diff) / len(x)
  return r

print("Residuals with N components:")
for N in [0, 2, 6]:
  r = res(flux, N, evecs)
  print(f"N = {N}: residual = {r:.3g}")
```

```
Residuals with N components:
N = 0: residual = 7.84
N = 2: residual = 3.36
N = 6: residual = 1.22
```

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

In [ ]:

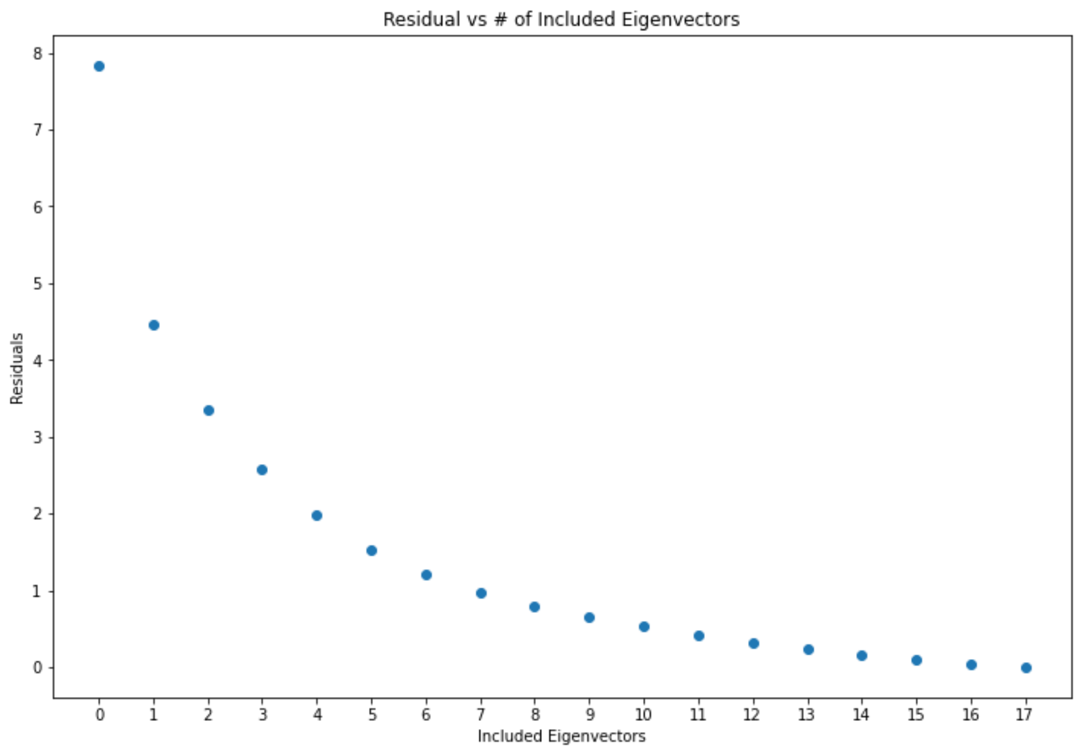
```
rec_spectra = rec(flux, 6, evecs)

fig, axs = plt.subplots(figsize=(12, 10), nrows=2, sharex=True,
for i, ax in enumerate(axs.ravel()):
  ax.plot(wavelength, flux[i], label="Original Spectrum")
  ax.plot(wavelength, rec_spectra[i], label="Reconstructed Spect
  ax.set_ylabel("Flux [arbitrary units]")
  ax.set_title(f"Spectrum {i}")
ax.set_xlabel("Wavelength [Å]")
axs[0].legend()
plt.tight_layout()
plt.show()
```



7. Plot the residual as a function of the number of included eigenvectors (1,2,3,...,17).

In [ ]:

```
all_res = []
for n in range(len(evals) + 1):
  r = res(flux, n, evecs)  # compute residuals
  all_res.append(r)

plt.figure(figsize=(12, 8))
plt.scatter(np.arange(len(evals)+1), all_res)
plt.title("Residual vs # of Included Eigenvectors")
plt.ylabel("Residuals")
plt.xlabel("Included Eigenvectors")
plt.xticks(np.arange(len(evals)+1))
plt.show()
```



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 $\mathbf{X_c}$.

In [ ]:

```
from scipy.linalg import svd

S = svd(Xc, compute_uv=False)

# Print Eigenvalues
evals_svd = S**2 / (len(Xc) - 1)
print(evals_svd)

assert np.allclose(evals_svd[:17], evals)
```

```
[3.57183360e+00 1.16921314e+00 8.19491399e-01 6.43886056e-01
 4.87138879e-01 3.16043520e-01 2.72246202e-01 1.83227778e-01
 1.41340818e-01 1.35417557e-01 1.24349547e-01 9.68536857e-02
 8.91735508e-02 8.05492370e-02 6.53953675e-02 5.46083371e-02
 4.85022745e-02 2.72391755e-30]
```

Problem 3 - Back to MNIST

In Assignment 2, we used the UMAP module to reduce the MNIST dataset to 2 dimensions (from 784) for easy visualization and observed that different classes (10 digits - "0", "1", ..., "9") got separated nicely into clusters when the MNIST data are embedded into lower dimensions by UMAP.

In this exercise, instead of the UMAP module, we use the PCA method for dimensionality reduction.

As mentioned in Problem 2, PCA is a technique for reducing the number of dimensions in a dataset whilst retaining most information. It is using the correlation between some dimensions and tries to provide a minimum number of variables that keeps the maximum amount of variation or information about how the original data is distributed. It does not do this using guesswork but using hard mathematics and it uses something known as the eigenvalues and eigenvectors of the data-matrix. These eigenvectors of the covariance matrix have the property that they point along the major directions of variation in the data. These are the directions of maximum variation in a dataset. Here, we use the scikit-learn implementation of PCA: https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html (https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html)

First, load the MNIST data:

In [2]:

```
from sklearn.datasets import fetch_openml
mnist = fetch_openml('mnist_784')
X = mnist.data
Y = mnist.target

X = X.to_numpy()
Y = Y.to_numpy()
```

"X" contains information about the given MNIST digits. We have a 28x28 pixel grid, so each image is a vector of length 784; we have 70,000 images (digits), so X is a

70,000x784 matrix. "Y" is a label (0-9; the category to which each image belongs) vector of length 70,000.

1. Do the following:

(1) Randomly shuffle data (i.e. randomize the order)

(Note: The label Y_1 corresponds to a vector X_{1j}, and even after shuffling, Y_1 should still correspond to X_{1j}.)

(2) Select 1/3 of the data. (You are free to work with a larger set of the data, but it will take much longer time to train.)

(3) Split data into training and test samples using train_test_split (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html)). Set train_size = 0.8. (80% of X is our training samples.) Print the dimension of training and test samples. </i></span>

In [3]:

```
from sklearn.model_selection import train_test_split
from sklearn.utils import check_random_state

# select 1/3 of the data to work with (discard the "test data" =
X, _, Y, _ = train_test_split(X, Y, train_size=1/3, random_state

X_train, X_test, Y_train, Y_test = train_test_split(
    X, Y, train_size=0.8, random_state=42, shuffle=True
)

print(X_train.shape)
print(X_test.shape)
print(Y_train.shape)
print(Y_test.shape)
```

```
(18666, 784)
(4667, 784)
(18666,)
(4667,)
```

Many machine learning 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)) PCA is a prime example of when such normalization is important; if the variables are not measured on the same scale, then each principal component can be dominated by a single variable.

In this exercise, the MNIST pixel values in images should also be scaled prior to providing the images as an input to PCA. There are three main types of pixel scaling techniques: normalization (scaling pixel to the range 0-1), centering (scale pixel values to have a zero-mean), and standardization (scale pixel values to have a zero-mean and unit-variance).

First, let us try normalization. Each pixel contains a greyscale value quantified by an integer between 0 and 255. To standardize the dataset, we normalize the "X" data in the interval [0, 1].

2. Normalize the X data (both training and test).

In [4]:
```
X_train /= 255
X_test /= 255
```

Next, using scikit-learn's PCA module, we can select the first two principal components from the original 784 dimensions.

In [5]:
```
from sklearn.decomposition import PCA
```

(1) Define the PCA model with the first 2 principal components:

pca = PCA(n_components=2)

(2) Using "fit_transform," fit the model with the training X data and apply the dimensionality reduction on it.

X_train_PCA = pca.fit_transform(training X data)

(3) With the same model, apply the dimensionality reduction on the test X data.

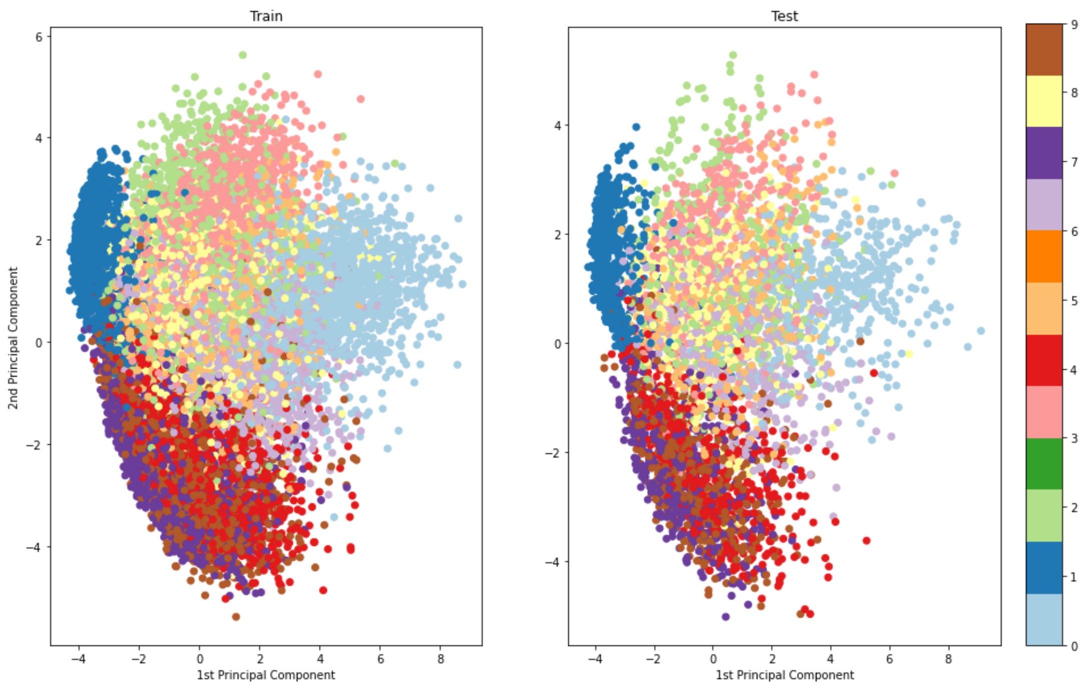
X_test_PCA = pca.transform(test X data)

3. This problem is similar to HW2-Q4-Part3. For both training and test samples, create a scatterplot of the first and second principal component and color each of the different types of digits with a different color. Label each axis (e.g. x-axis: 1st principal component, y-axis: 2nd principal component). How does it compare to the UMAP results?

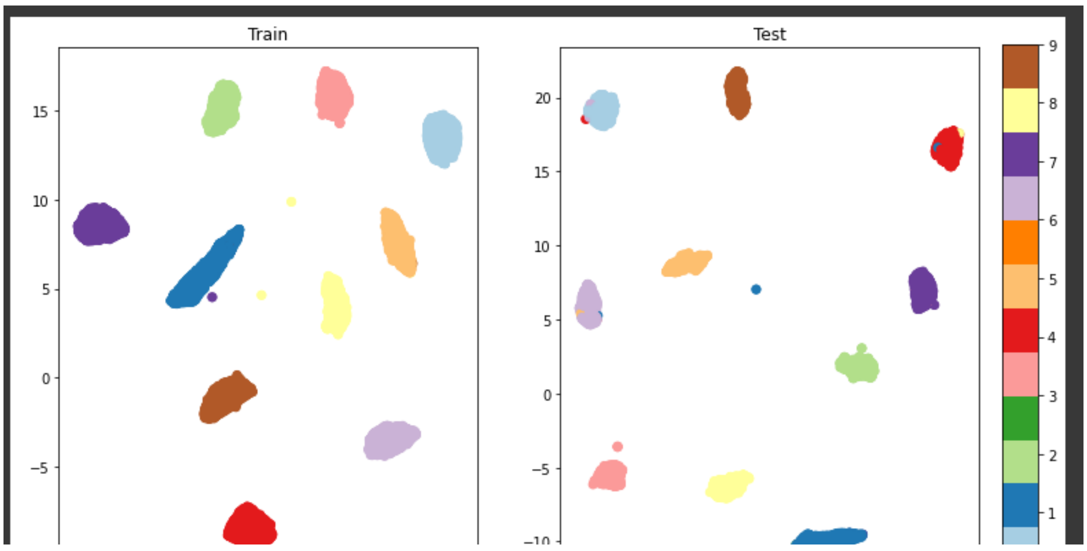
In [6]:
```
pca = PCA(n_components=2)
X_train_PCA = pca.fit_transform(X_train)
X_test_PCA = pca.transform(X_test)
```

In [7]:

```
from matplotlib.colors import Normalize
from matplotlib.cm import ScalarMappable

norm = Normalize(vmin=0, vmax=9)
sm = ScalarMappable(norm=norm, cmap="Paired")

fig, axs = plt.subplots(figsize=(15, 10), ncols=2)
axs[0].scatter(*X_train_PCA.T, c=Y_train.astype(int), cmap="Pair
axs[0].set_title("Train")
axs[1].scatter(*X_test_PCA.T, c=Y_test.astype(int), cmap="Paired
axs[1].set_title("Test")
cax = fig.add_axes([0.92, 0.125, 0.03, 0.76])
fig.colorbar(sm, cax=cax)
plt.setp(axs, xlabel="1st Principal Component")
axs[0].set_ylabel("2nd Principal Component")
plt.show()
```

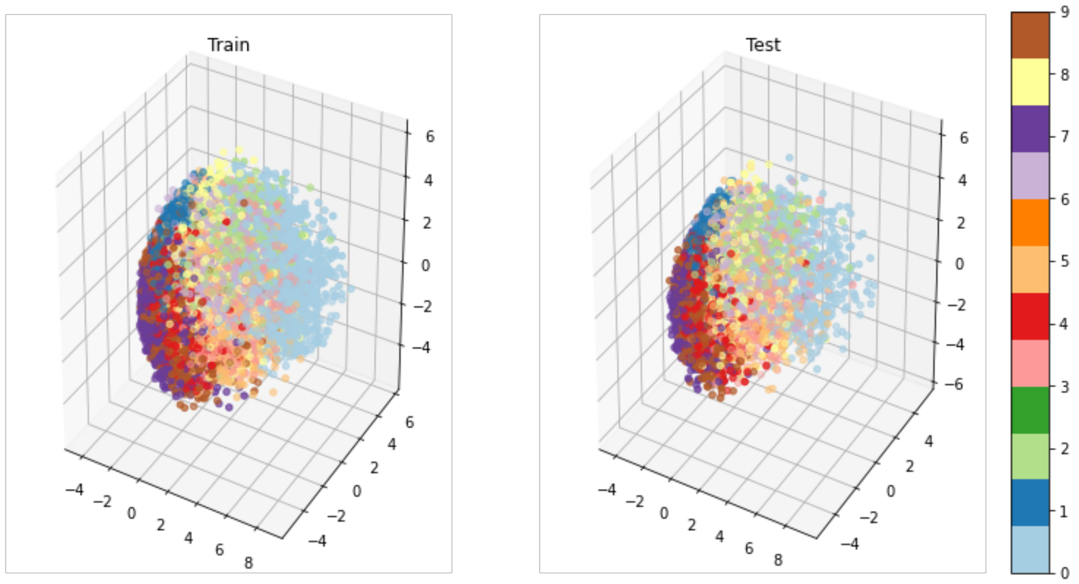


We attach a screenshot of the results from last week below. Clearly, UMAP separates the digits much more than PCA with 2 components. PCA still has clusters, but there is much more overlap than in the UMAP case. This is true both for Train and Test data.



4. Select the first three principal components and make 3D scatterplot on the training data. (similar to HW2-Q4-Part5)

In [8]:
```
pca = PCA(n_components=3)
X_train_PCA = pca.fit_transform(X_train)
X_test_PCA = pca.transform(X_test)

fig = plt.figure(figsize=(12, 7))
ax1 = fig.add_subplot(121, projection='3d')
ax1.scatter(*X_train_PCA.T, c=Y_train.astype(int), cmap="Paired"
ax1.set_title("Train")

ax2 = fig.add_subplot(122, projection='3d')
ax2.scatter(*X_test_PCA.T, c=Y_test.astype(int), cmap="Paired")
ax2.set_title("Test")

cax = fig.add_axes([0.92, 0.125, 0.03, 0.76])
fig.colorbar(sm, cax=cax)
plt.show()
```



From the graph we can see the two or three components definitely hold some information, especially for specific digits, but clearly not enough to set all of them apart. There are other techniques, such as UMAP module or t-SNE (t-Distributed Stochastic Neighbouring Entities), which can better reduce the dimensions for visualisation.

In [9]:
```
from sklearn.neighbors import KNeighborsClassifier as knn
```

Now, we will introduce K-nearest neighbors (KNN), one of the most widely used machine learning classification techniques. We use scikit-learn implementation of KNN:

https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsRegressor.html (https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsRegressor.html)

Ideally, we should tune KNN hyperparameters by doing a grid search using k-fold cross validation, but in this exercise we simply use default parameters with n_neighbors = 6.

(1) Define the knn classifier

clf = knn(n_neighbors=6)

(2) Fit the model

clf.fit(training X data, training Y/target data)

(3) Get the classification accuracy on the test data

clf.score(test X data, test Y/target data)

5. Evaluate the classification accuracy on the test data using a KNN classifier.

In [10]:

```
clf = knn(n_neighbors=6)
clf.fit(X_train, Y_train)
score = clf.score(X_test, Y_test)
print(f"The score is {score:.3g}.")
```

```
The score is 0.958.
```

The above KNN classifier considers all 784 features for each image when making its decisions. What if you do not need that many? It is possible that a lot of those features do not really affect our predictions that much. Or worse, KNN could be considering feature anomalies that are unique to our training data, resulting in overfitting. One way to deal with this is by removing features that aren't contributing much.

Now, suppose you take the first two principal components from PCA and fit your model using those two components.

pca = PCA(n_components=2)

X_train_PCA = pca.fit_transform(training X data)

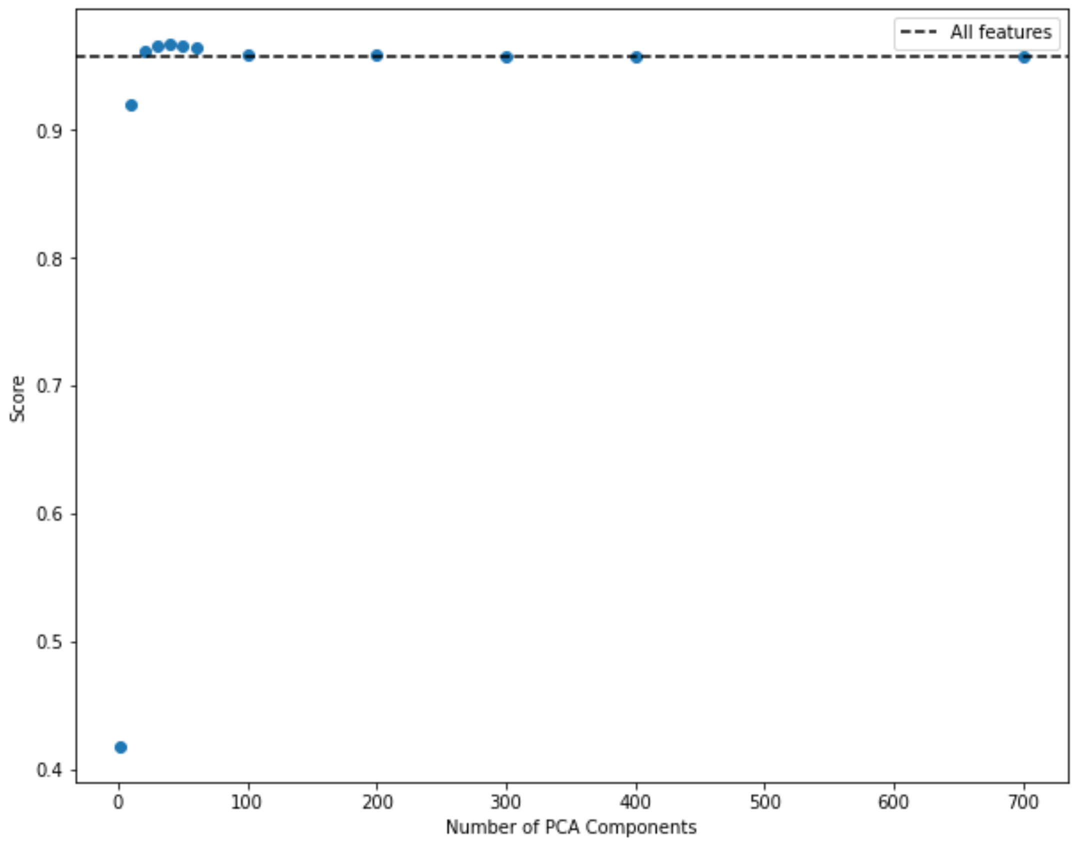
X_test_PCA = pca.transform(test X data)

Now you can take X_train_PCA, along with training Y data, to fit the KNN model and evaluate the classification accuracy.

6. Evaluate the classification accuracy with different number of PCA componenets. Let N_PCA_component = [2, 10, 20, 30, 40, 50, 60, 100, 200, 300, 400, 700]. Plot classification accuracy vs. number of PCA components. How does it compare to the accuracy in Part 5? Draw a horizontal line for the accuracy with all 784 features.

In [12]:
```
N_PCA = [2, 10, 20, 30, 40, 50, 60, 100, 200, 300, 400, 700]
scores = []

for n in N_PCA:
  pca = PCA(n_components=n)
  X_train_PCA = pca.fit_transform(X_train)
  X_test_PCA = pca.transform(X_test)

  clf = knn(n_neighbors=6)
  clf.fit(X_train_PCA, Y_train)
  s = clf.score(X_test_PCA, Y_test)
  scores.append(s)
```

In [13]:
```
plt.figure(figsize=(10, 8))
plt.scatter(N_PCA, scores)
plt.axhline(score, ls="--", c="k", label="All features")
plt.xlabel("Number of PCA Components")
plt.ylabel("Score")
plt.legend()
plt.show()
```



7. Instead of the PCA method, fit the UMAP model with the training data and do unsupervised learning. Reduce data to 2 dimensions (embed to 2 dimensions) and train the KNN model on the embedded training data. Compared to Part 6, does it give you a higher classification accuracy even with n_component = 2?

In [11]:

```
!pip install umap-learn
import umap
```

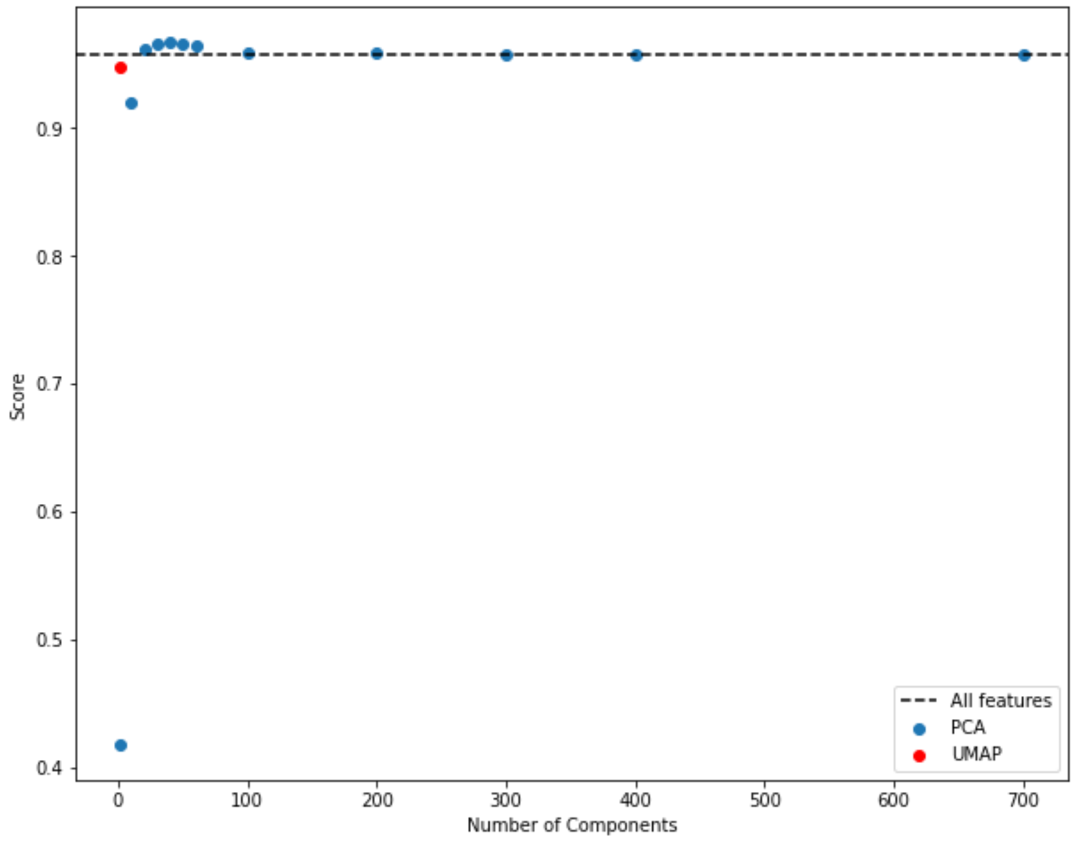
```
Looking in indexes: https://pypi.org/simple, https://us-python.p
Collecting umap-learn
  Downloading umap-learn-0.5.3.tar.gz (88 kB)
     |████████████████████████████████| 88 kB 3.2 MB/s
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/pyt
Requirement already satisfied: scikit-learn>=0.22 in /usr/local/
Requirement already satisfied: scipy>=1.0 in /usr/local/lib/pyth
Requirement already satisfied: numba>=0.49 in /usr/local/lib/pyt
Collecting pynndescent>=0.5
  Downloading pynndescent-0.5.7.tar.gz (1.1 MB)
     |████████████████████████████████| 1.1 MB 44.1 MB/s
Requirement already satisfied: tqdm in /usr/local/lib/python3.7/
Requirement already satisfied: llvmlite<0.40,>=0.39.0dev0 in /us
Requirement already satisfied: setuptools<60 in /usr/local/lib/p
Requirement already satisfied: importlib-metadata in /usr/local/
Requirement already satisfied: joblib>=0.11 in /usr/local/lib/py
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/loca
Requirement already satisfied: typing-extensions>=3.6.4 in /usr/
Requirement already satisfied: zipp>=0.5 in /usr/local/lib/pytho
Building wheels for collected packages: umap-learn, pynndescent
  Building wheel for umap-learn (setup.py) ... done
  Created wheel for umap-learn: filename=umap_learn-0.5.3-py3-no
  Stored in directory: /root/.cache/pip/wheels/b3/52/a5/1fd9e3e7
  Building wheel for pynndescent (setup.py) ... done
  Created wheel for pynndescent: filename=pynndescent-0.5.7-py3-
  Stored in directory: /root/.cache/pip/wheels/7f/2a/f8/7bd5dcec
Successfully built umap-learn pynndescent
Installing collected packages: pynndescent, umap-learn
Successfully installed pynndescent-0.5.7 umap-learn-0.5.3
```

In [20]:

```
umap_model = umap.UMAP(n_components=2)
embedding_train = umap_model.fit_transform(X_train, Y_train)
embedding_test = umap_model.transform(X_test)

clf = knn(n_neighbors=6)
clf.fit(embedding_train, Y_train)
umap_score = clf.score(embedding_test, Y_test)
```

In [23]:

```
plt.figure(figsize=(10, 8))
plt.scatter(N_PCA, scores, label="PCA")
plt.axhline(score, ls="--", c="k", label="All features")
plt.scatter(2, umap_score, color="red", label="UMAP")
plt.xlabel("Number of Components")
plt.ylabel("Score")
plt.legend()
plt.show()
```



UMAP is almost as good with 2 features as PCA is with 20 features (the third blue point). It was expected that UMAP would do better given how much more it separates the clusters in the above scatter plots.

Instead of pixel normalization, we can also try feature rescaling through standardization (rescaling the features such that they have the properties of a standard normal distribution with a mean of zero and a standard deviation of one). We can use sklearn.preprocessing.StandardScaler for this job.

In [16]:

```
from sklearn.preprocessing import StandardScaler
```

(1) Define the StandardScaler

sc = StandardScaler()

(2) Fit the training X data and then transform it.

X_train = sc.fit_transform(training X data)

(3) Perform standardization on the test X data.

X_test = sc.transform(test X data)

8. Re-load the MNIST data (Repeat Part 1) and try standardization on both training and test X data following the above steps. Evaluate the classification accuracy using a KNN classifier. How does it compare to Part 5?

In [22]:

```
sc = StandardScaler()
X_train = sc.fit_transform(X_train * 255)  # undo the pixel norm
X_test = sc.transform(X_test * 255)

clf = knn(n_neighbors=6)
clf.fit(X_train, Y_train)
score2 = clf.score(X_test, Y_test)
print(f"The new score is {score2:.3g}.")
print(f"The old score was {score:.3g}.")
```

```
The new score is 0.926.
The old score was 0.958.
```

The StandardScaler was actually not as good as pixel normalization. Thus could be due to the test and training set having slightly different distributions (the rescaled test data has standard deviation 1.5, not 1.)

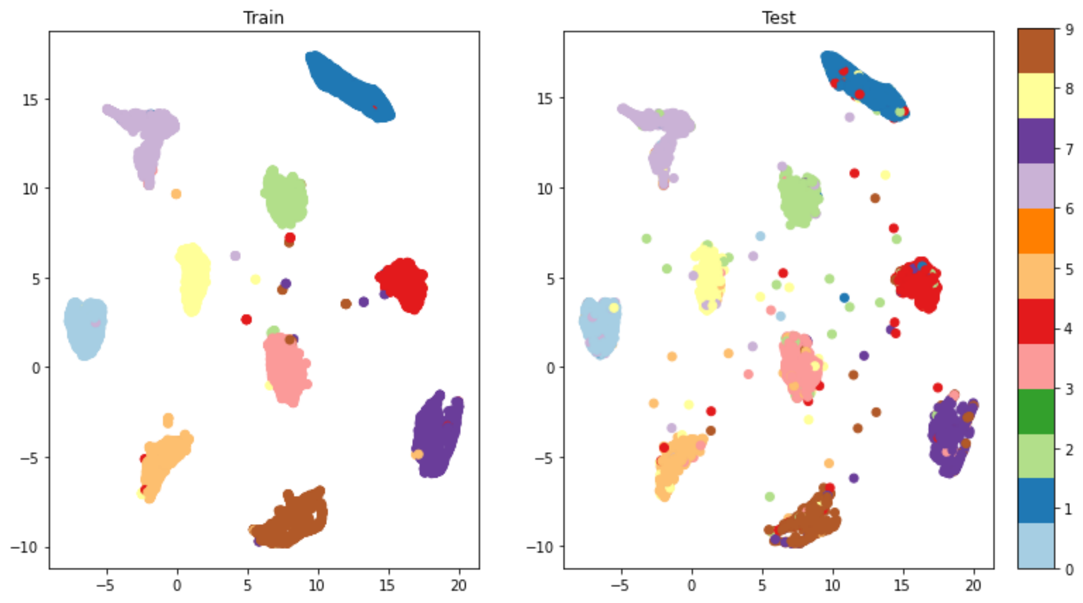
In [30]:

```
print(X_train.mean())
print(X_test.mean())

print(X_train.std())
print(X_test.std())
```

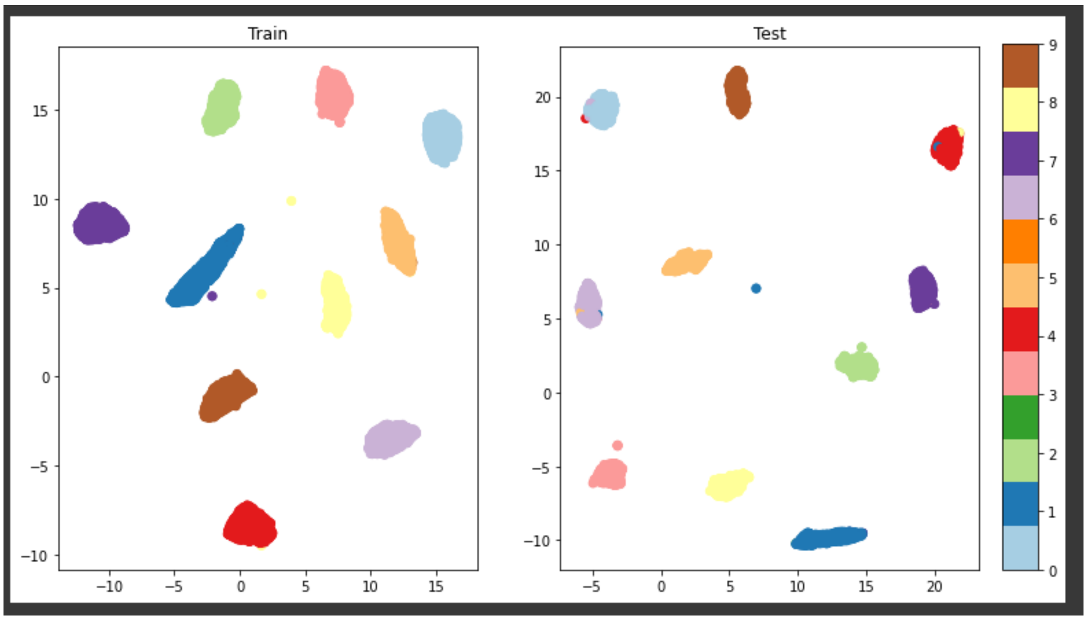
```
5.311781494848962e-19
0.005641247349771385
0.9455858782052696
1.5221657914335163
```

9. Again, take the data from Part 8 (standardized X data) and do unsupervised learning using the UMAP module. Reduce the data to 2 dimensions, and plot the embedding as a scatterplot (for the training data) and color by the target array. How does it compare to HW2-Q4-Part6? Which pixel rescaling method do you think works better?

In [31]:
```
umap_model = umap.UMAP(n_components=2)
embedding_train = umap_model.fit_transform(X_train, Y_train)
embedding_test = umap_model.transform(X_test)

fig, axs = plt.subplots(figsize=(12, 7), ncols=2)
axs[0].scatter(*embedding_train.T, c=Y_train.astype(int), cmap="
axs[0].set_title("Train")
axs[1].scatter(*embedding_test.T, c=Y_test.astype(int), cmap="Pa
axs[1].set_title("Test")
cax = fig.add_axes([0.92, 0.125, 0.03, 0.76])
fig.colorbar(sm, cax=cax)
plt.show()
```



We again attach the screenshot from last week. It again seems that the standard scaling (mean = 0 and unit variance) is the best choice in this case.


Train
Test