Edward-Kuhn / **ml_Assaginment** Public

generated from Thayer-ENGS108/Assignment_5_Fall2022

Code Issues Pull requests Actions Projects Wiki Security Insig

main

ml_Assaginment / **Assignment_2_Fall2022.Dianhao_Liu.ipynb**

Edward-Kuhn Add files via upload **History**

1 contributor

1 lines (1 sloc) | 398 KB

ENGS 108 Fall 2022 Assignment 2

Due September 30, 2022 at 11:59PM on Github

Instructors: George Cybenko

TAs: Chase Yakaboski and Clement Nyanhongo

Rules and Requirements

1. You are only allowed to use Python packages that are explicity imported in

the assignment notebook or are standard (bultin) python libraries like random, os, sys, etc, (Standard Bultin Python libraries will have a Python.org documentation). For this assignment you may use:

- numpy
- pandas
- scikit-learn
- matplotlib

2. All code must be fit into the designated code or text blocks in the assignment notebook. They are indentified by a **TODO** qualifier.

3. For analytical questions that don't require code, type your answer cleanly in Markdown. For help, see the Google Colab Markdown Guide.

In [ ]:

```
''' Import Statements '''
import numpy as np
import pandas as pd
import sklearn
import matplotlib.pyplot as plt
from copy import deepcopy
from sklearn.linear_model import LinearRegression
from scipy import signal
import matplotlib.collections as collections

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

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

Data Loading

Upload the red and synthetic datasets to your google colab session using Google

Drive. Read the following tutorial for how to get setup.

In [ ]:

```
#TODO: Set your base datasets path. This is my base path, you will need to
dataset_base_path = '/content/drive/MyDrive/ml_Assignment_2/datasets/'
```

In [ ]:

```
#-- Everything else you should not need to change.
import os
import pickle

#-- Gather paths
synth_data_path = os.path.join(dataset_base_path, 'assign_2_synth_data.pk')
red_train_path = os.path.join(dataset_base_path, 'red_train.csv')
red_valid_path = os.path.join(dataset_base_path, 'red_valid.csv')
red_test_path = os.path.join(dataset_base_path, 'red_test.csv')
synth_train_path = os.path.join(dataset_base_path, 'synth_train.csv')
synth_valid_path = os.path.join(dataset_base_path, 'synth_valid.csv')
synth_test_path = os.path.join(dataset_base_path, 'synth_test.csv')

#-- Load Synth_Data
with open(synth_data_path, 'rb') as f_:
  synth_data = pickle.load(f_)

#-- Load Red Wine Data
red_train_df = pd.read_csv(red_train_path)
red_valid_df = pd.read_csv(red_valid_path)
red_test_df = pd.read_csv(red_test_path)
synth_train_df = pd.read_csv(synth_train_path)
synth_valid_df = pd.read_csv(synth_valid_path)
synth_test_df = pd.read_csv(synth_test_path)

#-- Data is stored in a tuple of format (X, y) and are already converted to
red_train = (red_train_df.drop('quality', axis=1).to_numpy(), red_train_df[
red_valid = (red_valid_df.drop('quality', axis=1).to_numpy(), red_valid_df[
red_test = (red_test_df.drop('quality', axis=1).to_numpy(), red_test_df['qu

#-- Load in Synth train, valid, test data with tuple format (X, y)
synth_train = (synth_train_df.drop('y', axis=1).to_numpy(), synth_train_df[
synth_valid = (synth_valid_df.drop('y', axis=1).to_numpy(), synth_valid_df[
synth_test = (synth_test_df.drop('y', axis=1).to_numpy(), synth_test_df['y'
```

Problem 1: K-Means Clustering

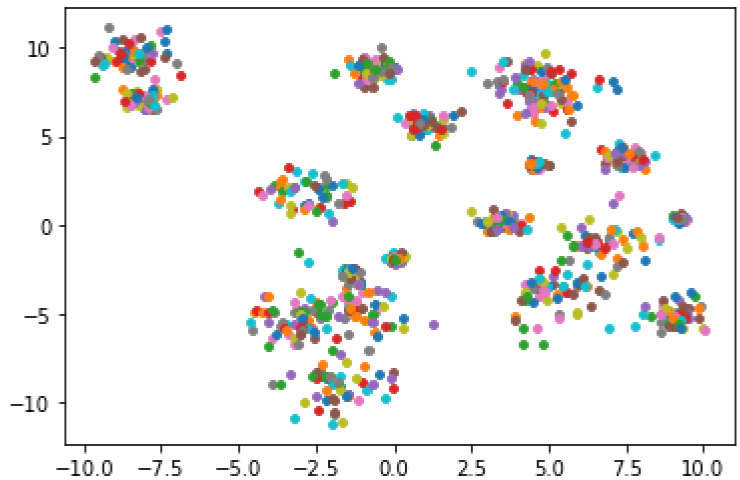
In this problem, you will solve a clustering task using the k-means algorithm and an associated classification task using k nearest neighbors algorithm, both of which you learned in class. The dataset for this problem is a synthetic two-dimensional dataset *synth_data*. Each entry has two features (x_1, x_2).

> **Part 1** A reasonable first step in every machine learning task is to understand the dataset at hand. Proceed to explore this problem's dataset by addressing the following:
>
> > **(a)** Choose a suitable type of plot and visualize the training data

training data.

In [ ]:

```
#TODO: Write your code here. Use matplotlib for visualization.
fig, ax = plt.subplots()

for i in range(len(synth_data)):
  x = synth_data[i][0]
  y = synth_data[i][1]

  ax.scatter(x, y, s=15)

plt.show()
```



> **(b)** From your plot, how many clusters, k, would you estimate are represented in the dataset?

TODO: *Type your answer in Markdown here.*

k=14

> **Part 2** Build a model.
>
> > **(a)** Using the k-Means algorithm, implement a clustering model. *Hint: Use scikit-learn's K-means library.*

In [ ]:

```
#TODO: Write your code here. Hint: Just define a model, don't train yet.
from sklearn.cluster import KMeans

kmeans = KMeans(n_clusters = 14, random_state = True).fit(synth_data)
```

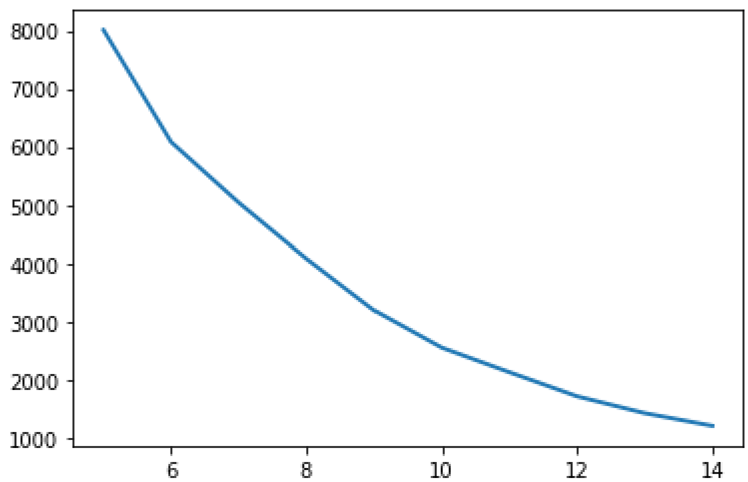
> **(b)** Train the clustering model on several reasonable values of k, taking into account your visual inspection from 1b. Plot the sum of distance (SSE) from each data

> point and its respective cluster for 10 different values of k.

In [ ]:
```
def train1(k, dataset):
  ''' Using your model above, implement a function that will train your K-m
  for different values of k on your dataset and return the trained model'''

  model = KMeans(n_clusters = k, random_state = True).fit(dataset)
  return model
```

In [ ]:
```
def calculateSSE(model):
  ''' Using a trained model calculate the SSE for the model '''

  sse = model.inertia_
  return sse
```

In [ ]:
```
#TODO: Choose 10 different values of k based on your inspection and plot th
fig, ax = plt.subplots()

x = []
y = []

for i in range(5,15):
  x.append(i)
  y.append(calculateSSE(train1(i, synth_data)))
ax.plot(x, y, linewidth=2.0)
plt.show()
```



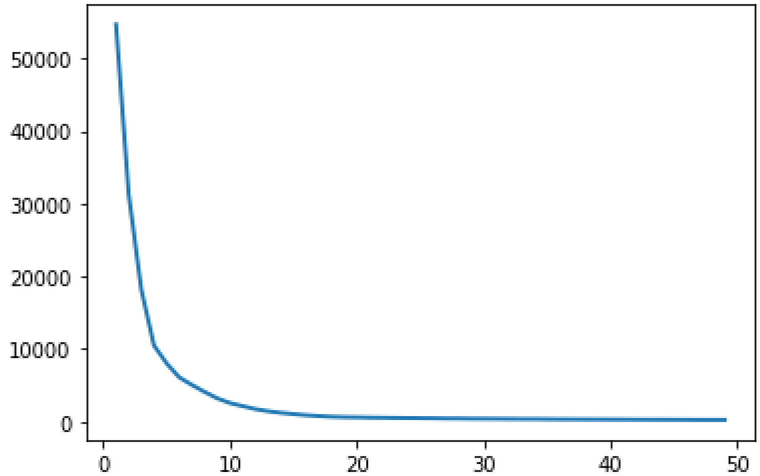
> **(c)** What value of k is optimal? How does it compare to your visual inspection?

TODO: *Type your answer in Markdown here.*

Best k value is 8.

In [ ]:
```
#TODO: Write code and plot a graph showing the optimal value of k.
```

```
fig, ax = plt.subplots()

x = []
y = []

for i in range(1,50):
  x.append(i)
  y.append(calculateSSE(train1(i, synth_data)))
ax.plot(x, y, linewidth=2.0)
plt.show()
```



Problem 2: k-NN Classification

In this problem, you will utilize data deriving from the same synthetic dataset as above. This time, the data has been separated into *synth_train*, *synth_valid* and *synth_test* arrays. Furthermore, each sample now includes a class label found in the y column. These class labels come from the set $\{1, 2, \ldots, 31\}$. *Note: These are not the same datasets as Problem 1.*

> **Part 1** Train an implementation of the k-Nearest Neighbors algorithm on the training dataset. Note that k here refers to the number of neighbors, not clusters.

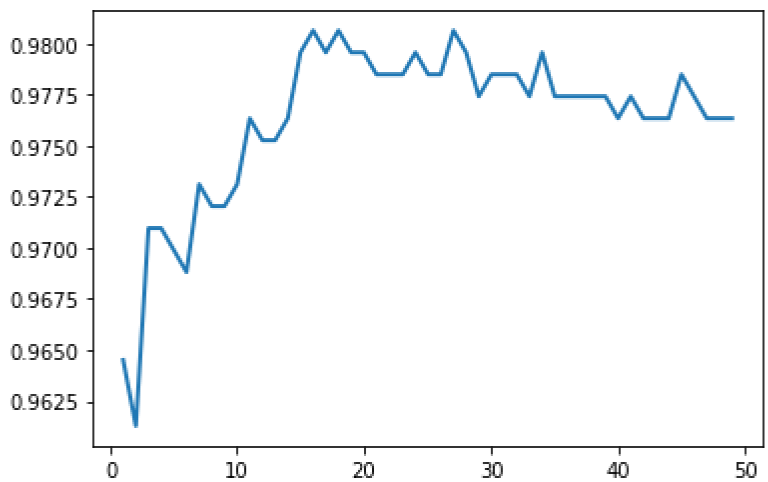
In [ ]:

```
from sklearn.neighbors import KNeighborsClassifier
def train2(k, dataset):
  ''' Implement a function that will train a k-NN
  for different values of k on your dataset and return the trained model'''

  model = KNeighborsClassifier(n_neighbors=k).fit(dataset[0],dataset[1])
  return model
```

> **Part 2** Report the classification accuracy of this model on the validation set for different values for k. Plot these accuracies against k and report the optimal value for k.

In [ ]:

```
#TODO: Write your code here.
ig, ax = plt.subplots()

x = []
y = []

for i in range(1,50):
  train_k = train2(i, synth_train)

  x.append(i)
  y.append(train_k.score(synth_test[0], synth_test[1]))

ax.plot(x, y, linewidth=2.0)
plt.show()
```



In [ ]:

```
"""
ig, ax = plt.subplots()

accuracy_k = 0
best_k = 0
x = []
y = []

for i in range(1,50):
  counter_k = 0
  predict_k = []
  train_k = train2(i, synth_train)

  for j in range(len(synth_test[1])):
    predict_k.append(train_k.predict([synth_test[0][j]]))

    if (synth_test[1][j] == predict_k[j]):
      counter_k += 1

  if (counter_k > accuracy_k):
    accuracy_k = counter_k
    best_k = i

  x.append(i)
  y.append(counter_k/len(synth_test[1]))

ax.plot(x, y, linewidth=2.0)
```

```
ax.plot(x, y, linewidth=2.0)
plt.show()

print("the optimal value for k is:",best_k)
"""
```

> **Part 3** Report the classification precision, recall and F1-score of this model on the data in synth test.csv using the optimal value of k that you found in Part 2.

In [ ]:

```
#TODO: Write your code here.
train_k = train2(16, synth_train)
print(train_k.score(synth_test[0], synth_test[1]))
```

```
0.9806451612903225
```

Problem 3: Decision Tree Classification

In this problem you will use decision trees to classify the quality of red vinho verde wine samples based on their physicochemical properties. The dataset has been separated into *red_train, red_valid and red_test* arrays. For all of these files, the rightmost column ("quality") is the target label for each datapoint. All other columns are features.

> **Part 1** First let's explore the datasets through the following exercises. Note that we cannot plot the data in a meaningful way given that number of features exceed the physical dimensions.
>
> > **(a)** How many datapoints are in the training, validation, and testing sets?

In [ ]:

```
#TODO: Write your code here.
len(red_train[0]), len(synth_valid[0]), len(synth_test[0])
```

Out[ ]: (895, 434, 930)

> > **(b)** How many features are available for each datapoint?

In [ ]:

```
#TODO: Write your code here.
len(red_train[0][0])
```

Out[ ]: 11

> > **(c)** What are the average *alcohol* and *pH* values for *training* samples?

In [ ]:
```
#TODO: Write your code here.

red_average = np.average(red_train[0], axis=0)
print("average alcohol", red_average[10], "pH values", red_average[8])
```

```
average alcohol 10.397951582867744 pH values 3.309541899441341
```

> **Part 2** Decision Trees.

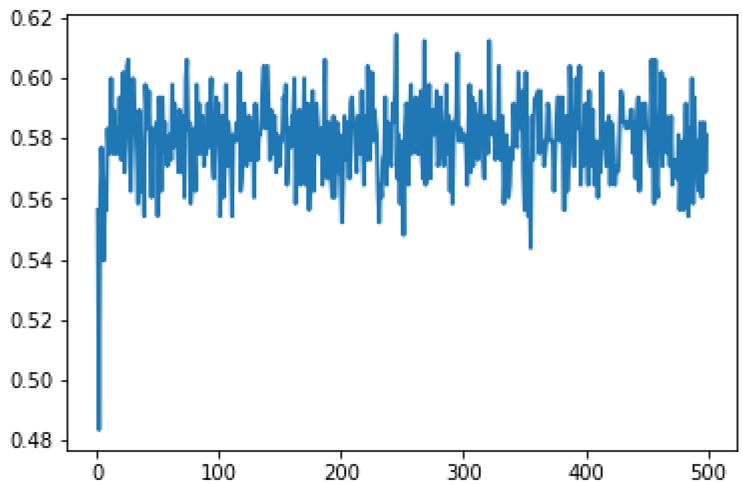
> > **(a)** Implement a binary decision tree model for the training data. *Hint: Try looking at the [scikit-learn decision tree library]().*

In [ ]:
```
from sklearn.tree import DecisionTreeClassifier
def train3(dataset, max_depth=None):
  ''' Implement a function that will train a decision tree model
  on your dataset and return the trained model'''

  model = DecisionTreeClassifier(max_depth=max_depth).fit(dataset[0],datase

  return model
```

> > **(b)** There are a number of hyperparameters that can be tuned to improve your model, one of which is the criteria for ending the splitting process. Two common ways of terminating the splitting process are *maximum depth* of the tree or *minimum number of samples* left. Tune the *maximum depth* of the tree by reporting the accuracy of the classifier in 2a on the validation set for different settings of *maximum depth*. Plot your findings.

In [ ]:
```
#TODO: Write your code here and plot your results.
ig, ax = plt.subplots()

x = []
y = []

accuracy_red = 0
best_red = 0

for i in range(1,500):
  red = train3(red_train, i)

  x.append(i)
  y.append(red.score(red_test[0], red_test[1]))
```

```
  if (red.score(red_test[0], red_test[1]) > accuracy_red):
    accuracy_red = red.score(red_test[0], red_test[1])
    best_red = i

ax.plot(x, y, linewidth=2.0)
plt.show()

print("maximum depth is", best_red)
```




```
maximum depth is 245
```

> **(c)** Use the optimum setting of *maximum depth* found in 2b to report the accuracy of the classifier on the *test* dataset.

In [ ]:

```
#TODO: Write
red = train3(red_train, best_red)
print(red.score(red_test[0], red_test[1], sample_weight=None))
```

```
0.5666666666666667
```

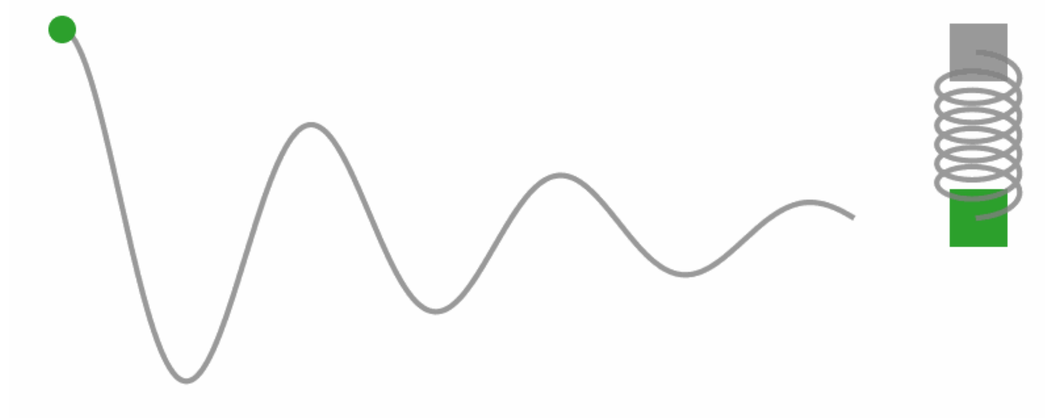
Problem 4: Systems - Estimating ODE Parameters

Many real-world systems can be modelled by linear diffferential equations. Some of the most common examples are mechanical and electrical oscillations (see mass-spring example below) which can be described by the solution of an initial value problem of the form:

$$ax'' + bx' + cx = g(t) \tag{1}$$

, where initial condition are given by: $x(0) = x0$, $x'(0) = x'0$

For our problems, we will assume that $g(t) = 0$, no external force (for spring system etc)



> **Part 1** Lets generate some synthetic data using an ODE for a vibration with no damping in chapter 3.7 Example 4 (Source: Elementary Differential Equations and Boundary Value Problems by Boyce & DiPrima, Wiley 2017).

In this system,

$$x'' + 0.125x' + x = 0 \tag{2}$$

and the analytical solution is the function below:

In [ ]:

```
t = np.linspace(0, 30*np.pi, 1000)   # time
x_funct = lambda t: (32/np.sqrt(255))*np.exp(-1*t/16)*np.cos((np.sqrt(255)/

# analytic function x given t
x_analytic = x_funct(t)
```

> > **(a)** Now lets assume we have observed a noisy sample composed of the first 20% of x_analytic. Create noisy data for the first 20% of x_analytic

In [ ]:

```
# create t_noisy (time) to record time for the first 20% of t
NOISY_FACTOR = 5 # controls the threshold for adding noise

len_t = int(0.2*len(t))
t_noisy = t[:len_t]
```

In [ ]:

```
# TODO: Compute x for the corresponding t_noisy
x = x_funct(t_noisy)
```

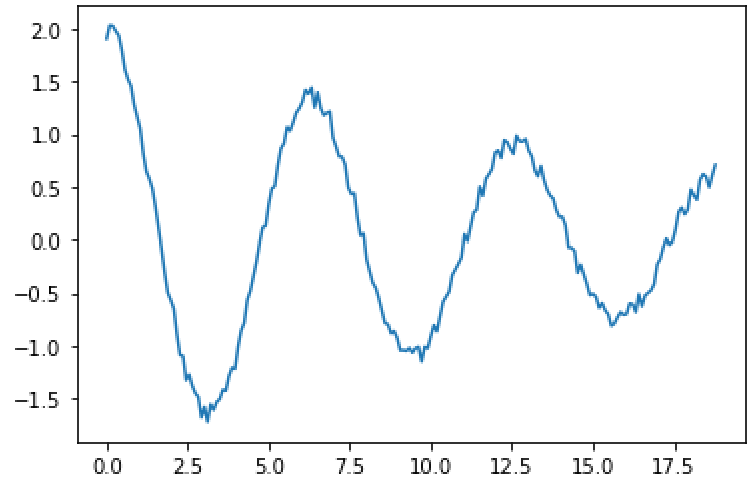
In [ ]:

```
# TODO: adding noise
noise = np.array(np.random.random(len_t) - 0.5)/NOISY_FACTOR
x_noisy = x + noise
```

> **(b)** Our task in this question is to estimate parameters a, b, and c, assuming that we only observed x_noisy

In [ ]:

```
# TODO: Plot the observed noisy data below (time vs displacement)
fig, ax = plt.subplots()

ax.plot(t_noisy, x_noisy)
plt.show()
```



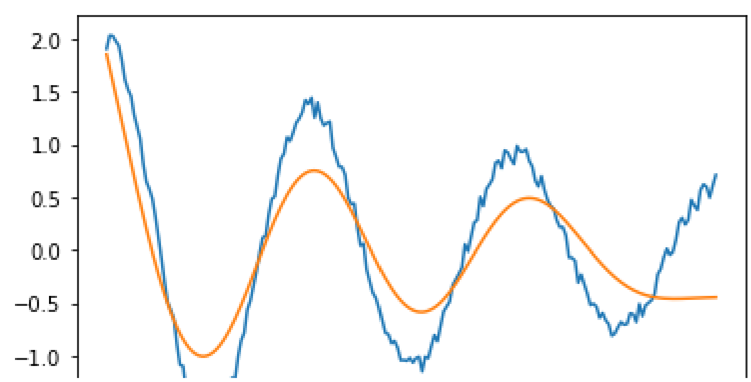
> **(c)** Real-world data is often noisy and denoising can help to reduce the noise. Denoise the above data to create x_denoised:

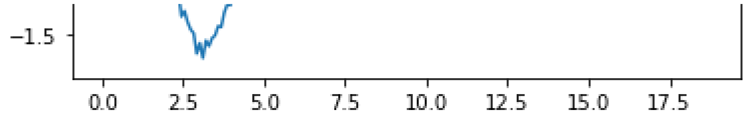
In [ ]:

```
# denoising
N, Wn = 5, 0.03   # Feel free to modify N and Wn as you see fit!
b, a = signal.butter(N, Wn, analog=False)   # module from scipy
x_denoised = signal.filtfilt(b,a,x_noisy)
```

In [ ]:

```
# TODO: Plot and insert legend to differentiate x_noisy and x_denoised vs t
fig, ax = plt.subplots()

ax.plot(t_noisy, x_noisy)
ax.plot(t_noisy, x_denoised)
plt.show()
```





Part 2: Compute derivatives x' and x'' to estimate a, b, and c given x

> **a** Using the (forward method (finite difference)).

compute x' and x'' for both x_noisy and x_denoised

In [ ]:

```
#TODO: Complete the function below
def first_derivative(X, dt):
    # approximate derivative using forward nethod
  derivative_1 = []
  for k in range(len(X)-1):
    derivative_1.append((X[k+1]-X[k])/dt)
  first_derivative = np.array(derivative_1)
  return first_derivative
```

In [ ]:

```
#TODO: Complete the functions below
def second_derivative(X_first, dt):
  # Basically differentiate the first derivative
  derivative_2 = []
  for k in range(len(X_first)-1):
    derivative_2.append((X_first[k+1]-X_first[k])/dt)
  second_derivative = np.array(derivative_2)
  return second_derivative
```

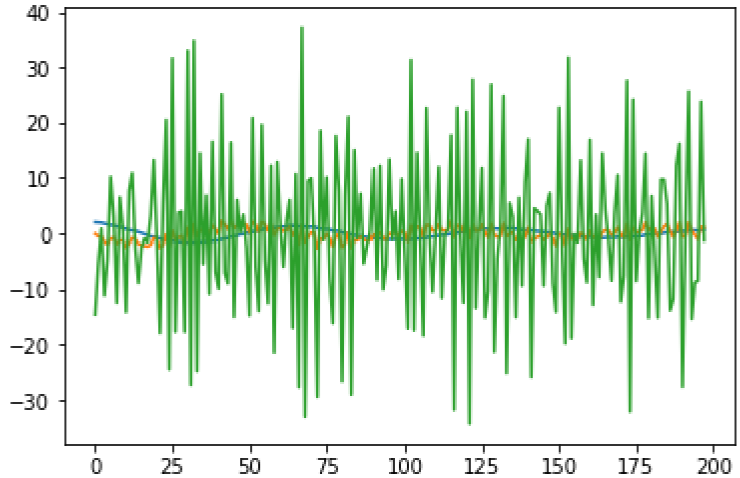
In [ ]:

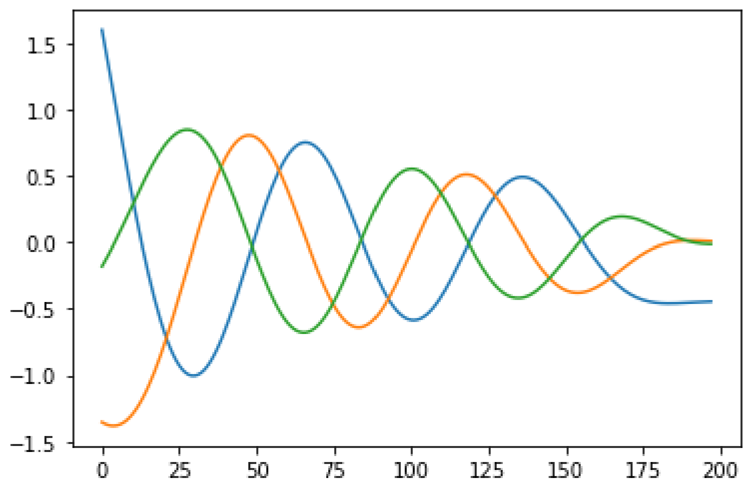
```
def get_derivatives (X):
    dt = t[1] - t[0] # time difference
    X_prime =  first_derivative(deepcopy(X), dt)
    X_prime_squared = second_derivative(deepcopy(X_prime), dt)
    # adjust to make equal lengths arrays
    return X[2:], X_prime[1:], X_prime_squared


# for noisy data
x, x_prime, x_prime_squared = get_derivatives(x_noisy)

# for denoised data
x1, x_prime1, x_prime_squared1 = get_derivatives(x_denoised)
```

In [ ]:

```
# TO DO: Fill the function belwo

def plot_figs (x, x_first, x_second):
  # TD DO: On same graph, plot x, x', x''
  fig, ax = plt.subplots()

  ax.plot(x)
  ax.plot(x_first)
  ax.plot(x_second)
  plt.show()
```

```
#   return;


plot_figs(x, x_prime, x_prime_squared)
plot_figs(x1, x_prime1, x_prime_squared1)
```





> **(b)** How do the derivative plots compare for the noisy vs the denoised samples? Whats the effect of denoising? What happens when we adjust the NOISY_FACTOR (see Part 1a)?

In [ ]:

```
# TODO: Your answer in Markdown
"""
The blue line is the same for both graphs and can be used as a basis for co
The denoising efficiency is good.
The larger the value of NOISY_FACTOR, the smaller the noise. If the value o
"""
```

Out[ ]:

```
'\nThe blue line is the same for both graphs and can be used as a basis for
comparison.\nThe denoising efficiency is good.\nThe larger the value of NOI
SY_FACTOR, the smaller the noise. If the value of NOISY_FACTOR is large (su
ch as 5000), the above two graphs will be very similar.\n'
```

> > **(c)** Now we have x, x' and x''. Since g(t) = 0; we can estimate a, b, and c via regression. If we assume c = 1,

then Equation 1 can be written as:

$$ax\prime\prime + bx\prime = -x \tag{3}$$

> > > From Equation 3, we can perform linear regression to estimate parameters a and b. Using -x as your dependent variable, and x' and x'' as your independent variables. Train a regression model below:

In [ ]:

```
#TODO: Fill the function below
from sklearn.linear_model import LinearRegression

def train_model (X, X_first, X_second):
  """ X - original x, X_first - first derivative, X_second - second derivat
  # TODO: Using Equation 3 with independent variable, (X'' and X'), depende
  #       Fit a linear regression model

  model = LinearRegression().fit(np.array((X_second, X_first)).T,-X)

  # return the regression coefficients and the model (which we will be a an
  return model.coef_

# train regression models for the noisy and denoised data
coeff_noisy   = train_model(x, x_prime, x_prime_squared)   # noisy data
coeff_denoised = train_model(x1, x_prime1, x_prime_squared1)    # denoised d
```

Part 3 From the model coefficients, we can identify parameters a and b and we know that $c = 1$. Now, our task is to predict how good our model can predict the entire dataset.

In [ ]:

```
a_noisy, b_noisy = coeff_noisy#original
a_denoised, b_denoised = coeff_denoised

print('For the noisy sample: (a = {}, b = {}, c = 1)'.format(a_noisy, b_noi
print('For the denoised sample: (a = {}, b = {}, c = 1)'.format(a_denoised,
print("The analytic solution has (a = 1, b = 0.125, and c = 1)")
```

```
For the noisy sample: (a = 0.00247642755150216, b = 0.0015409564323852036,
c = 1)
For the denoised sample: (a = 1.1825759892932903, b = 0.3661557623405786, c
= 1)
The analytic solution has (a = 1, b = 0.125, and c = 1)
```

> > **(a)** How do estimated parameters from the noisy and denoised samples compare to the analytic parameters?

In [ ]:

In [ ]:
```
# TODO: Your answer
"""
The parameters from the noisy and denoised samples are not very close to th
"""
```

Out[ ]:
```
'\nThe parameters from the noisy and denoised samples are not very close to
the analytic parameters.\n'
```

> > **(b)** From Equation 3,
> >
> > $$x = -1 * (ax'' + bx') \tag{4}$$

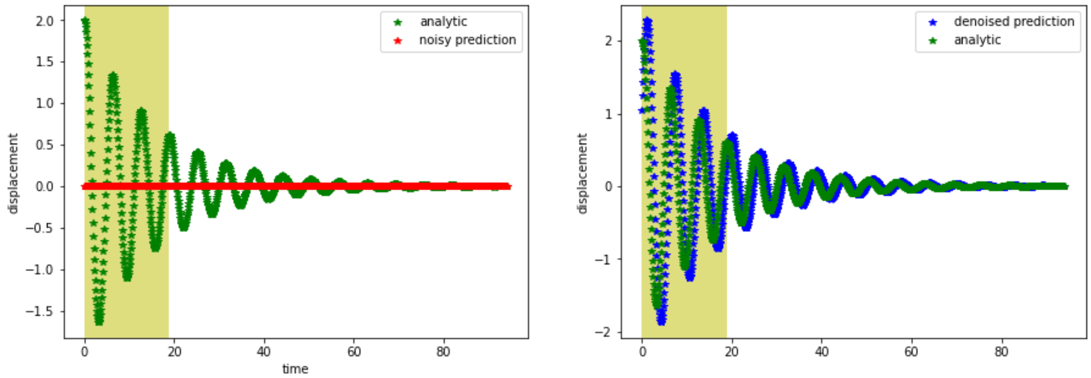
We will use this equation to test how good our parameters predict the analytic solution (given x' and x'').

In [ ]:
```
# get derivatives on entire dataset

X, X_prime, X_prime_squared = get_derivatives(x_analytic)


# TODO: Use Equation 3 to compute X
def compute_x (X_first, X_second, a, b):
  # Fill here

  X = -a*X_first - b*X_second
  return X

# TODO: Predict Y for the noisy sample, and the denoised sample
x_pred_noisy = compute_x(X_prime, X_prime_squared, a_noisy, b_noisy)
x_pred_denoised  = compute_x(X_prime, X_prime_squared, a_denoised, b_denois
```

In [ ]:
```
# adjust t to fit dimensions of predictions
LEN_T = len(t)
t_original = t[:LEN_T-2]
x_original = x_analytic[:LEN_T -2]


# Plots to show how well our parameters fit the data from the analytic solu
fig, axes = plt.subplots(1, 2, figsize = (15, 5))
axes[0].plot(t_original, x_original, '*', color = 'green', label = 'analyti
axes[0].plot(t_original, x_pred_noisy, '*', color = 'red', label = 'noisy p

axes[1].plot(t_original, x_pred_denoised, '*', color = 'blue', label = 'den
axes[1].plot(t_original, x_original, '*', color = 'green', label = 'analyti

axes[0].legend()
axes[1].legend()

# This shades the seen part (in creating the model -yellow), but the model
# extends to the unseen white part
axes[0].axvspan(0, t[len_t], color='y', alpha=0.5, lw=0)
axes[1].axvspan(0, t[len_t], color='y', alpha=0.5, lw=0)

axes[0].set_xlabel('time')
axes[0].set_ylabel('displacement')
axes[1].set_ylabel('time')
```

```
axes[1].set_ylabel('displacement')
```

Out[ ]: `Text(0, 0.5, 'displacement')`



(c) How do these two plots compare? On the same