AM207 Homework 2

Data: wine_quality_missing.csv

Harvard University
Fall 2018
Instructors: Rahul Dave
Due Date: Saturday, September 22nd, 2018 at 11:59pm

Instructions:

- Upload your final answers in the form of a Jupyter notebook containing all work to Canvas.
- Structure your notebook and your work to maximize readability.

Michelle (Chia Chi) Ho, Jiejun Lu, Jiawen Tong

In [1]:
```
import numpy as np
import scipy.stats
import scipy.special

import matplotlib
import matplotlib.pyplot as plt
import matplotlib.mlab as mlab
from matplotlib import cm
import pandas as pd
%matplotlib inline

from math import factorial
from IPython.display import display
```

Question 1: Give Me the Full Monte, Carlo Ancelotti

Coding required

In the quiet moments that transpire just before the sun rises that find us taking the walk of shame we can only send up quiet prayers to deities unknown that our path has not unknowingly taken us down the Boulevard of Broken Dreams (Green Day). Along this road you'll find scattered the shattered hearts of formerly aspiring humorists -- the sorts of folk who might admire "the giggle at a funeral" (Hozier) distributed according to the function $\heartsuit(\theta) \sim \frac{\sin^2 4\theta}{\theta^2}$ from $0 < \theta < \infty$ and otherwise. As a current aspiring humorist, it is your job to try to integrate $\heartsuit(\theta)$ in order to size up the jar you'll take with you as you go heart collecting (Christina Perri). Who do you think you are anyway?

1.1. Visualize $\heartsuit(\theta)$. Make sure your plot includes a title and axes labels.

1.2. The domain of $\heartsuit(\theta)$ is unbounded. The version of Monte Carlo that we've explored so far requires a bounded domain. Make an argument that we can integrate this function over the bounded domain [0, M] and get an accurate result. What value of M should you choose to get a result within 0.001 of the exact solution?

1.3. Write a function `simulate_heart_collection` to estimate $\int_0^\infty \heartsuit(\theta)$ using the standard Monte Carlo method with $N = 100000$. Use the bounds you justified in 1.2. What is your estimate?

1.4. It turns out that integrals of the form $\int_0^\infty \frac{\sin x^{2n}}{x^2} dx$ have the closed form solution $\frac{\pi}{2^{2n-1}}\binom{2n-2}{n-1}$. How accurate was your estimate?

1.5. The teaching staff of AM 207 has gone on and on in class and lab about putting error bars on estimates and finding confidence intervals. In order to do this you need to run your experiment a number of times. Repeat your estimation process 1000 times and plot a histogram of your results marking the exact answer and your estimate with a vertical line.

1.6. Based on your experiments, find the standard error of your estimate as well as a 95% confidence interval. Was the true value of $\int_0^\infty \heartsuit(\theta)$ within the 95% confidence interval?

1.7. It turns out that an appropriately chosen change of variables will allow you to estimate the integral on the part of the domain you truncated in 1.2 and 1.3. Execute this change of variables and use monte carlo integration to evaluate $\int_M^\infty \heartsuit(\theta)$

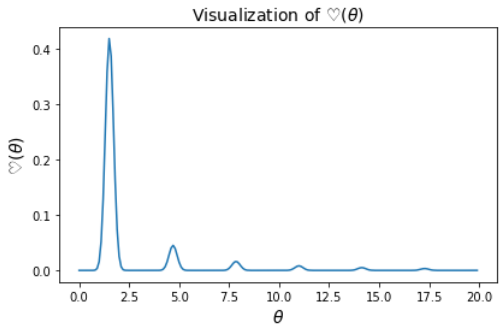
1.8. Based on your answer in 1.7 was your choice of M justified?

Answers

1.1. Visualize $\heartsuit(\theta)$. Make sure your plot includes a title and axes labels.

In [2]:
```
x_lin = np.arange(1e-9, 20, 0.1)
heart_x = np.power(np.sin(x_lin), 24) / x_lin**2

fig, ax = plt.subplots(1, 1, figsize=(6, 4))
ax.plot(x_lin, heart_x)
ax.set_xlabel(r'$\theta$', fontsize=14)
ax.set_ylabel(r'$\heartsuit(\theta$)', fontsize=14)
ax.set_title(r'Visualization of $\heartsuit(\theta$)', fontsize=14)
plt.tight_layout()
```



1.2. The domain of $\heartsuit(\theta)$ is unbounded. The version of Monte Carlo that we've explored so far requires a bounded domain. Make an argument that we can integrate this function over a bounded domain and get an accurate result. What bounds should you choose to get a result within 0.001 of the exact solution?

$$\int_0^\infty \heartsuit(\theta)d\theta = \int_0^M \heartsuit(\theta)d\theta + \int_M^\infty \heartsuit(\theta)d\theta$$

$$I = I_{lower} + I_{upper}$$

To satisfy $|I - I_{lower}| = |I_{upper}| < 0.001$, we need to ensure:

$$\int_M^\infty \heartsuit(\theta)d\theta = \int_M^\infty \frac{sin^{24}\theta}{\theta^2}d\theta < \int_M^\infty \frac{1}{\theta^2}d\theta = -\int_M^\infty d(\frac{1}{\theta}) = \frac{1}{M}$$

Therefore, $\frac{1}{M} < 0.001$ gives one possible bounded domain for M: $M > 1000$.

We chose $M = 1001$ for the rest of this exercise.

1.3. Write a function `simulate_heart_collection` to estimate $\int_0^\infty \heartsuit(\theta)$ using the standard Monte Carlo method with $N = 100000$. Use the bounds you justified in 1.2. What is your estimate?

In [3]:
```
def simulate_heart_collection(n, M, N=100000):
    x_unif = np.random.uniform(low=1e-9, high=M, size=N)
    heart_x_unif = np.power(np.sin(x_unif), 2*n) / (x_unif**2)
    return M * np.sum(heart_x_unif) / N

np.random.seed(1)
MC_integral = simulate_heart_collection(n=12, M=1001)
print('the MC estimate: ', MC_integral)
```

```
the MC estimate:  0.2631447405632876
```

1.4. It turns out that integrals of the form $\int_0^\infty \frac{\sin x^{2n}}{x^2}dx$ have the closed form solution $\frac{\pi}{2^{2n-1}}\binom{2n-2}{n-1}$. How accurate was your estimate?

In [4]:
```
def n_choose_k(n, k):
    return factorial(n) / factorial(k) / factorial(n-k)

n = 12
exact_integral = np.pi * n_choose_k(2*n-2, n-1) / np.power(2, 2*n-1)

print('Exact integral: ', exact_integral)
print('MC estimate: ', MC_integral)
print('Absolute error: ', np.abs(exact_integral-MC_integral))
print('Relative error in %: ', np.abs(exact_integral-MC_integral)/exact_integral * 100)
```

```
Exact integral:  0.26418924198235927
MC estimate:  0.2631447405632876
Absolute error:  0.0010445014190716417
Relative error in %:  0.39536107194757997
```

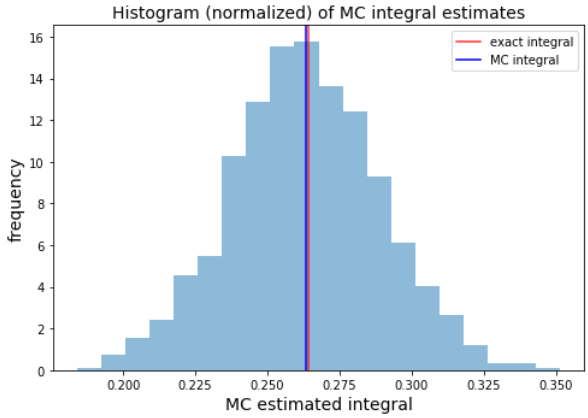
The absolute error of our estimate is 0.001;
The relative error of our estimate with regard to the exact intergral is 0.395%

1.5. The teaching staff of AM 207 has gone on and on in class and lab about putting error bars on estimates and finding confidence intervals. In order to do this you need to run your experiment a number of times. Repeat your estimation process 1000 times and plot a histogram of your results marking the exact answer and your estimate with a vertical line.

In [5]:
```
# repeat estimation for 1000 times
np.random.seed(1)
n_sim = 1000
MC_integral_estimates = np.zeros((n_sim,))
for i in range(n_sim):
    MC_integral_estimates[i] = simulate_heart_collection(n=12, M=1001)
```

In [6]:
```
# plot estimates in a histogram, overlay
fig, ax = plt.subplots(1, 1, figsize=(7, 5))
ax.hist(MC_integral_estimates, normed=True, bins=20, alpha=0.5)
ax.axvline(x=exact_integral, label='exact integral', color='r', alpha=0.8)
ax.axvline(x=MC_integral, label='MC integral', color='b')

ax.set_title('Histogram (normalized) of MC integral estimates', fontsize=14)
ax.set_xlabel('MC estimated integral', fontsize=14)
ax.set_ylabel('frequency', fontsize=14)
plt.legend()
plt.tight_layout()
```



1.6. Based on your experiments, find the standard error of your estimate as well as a 95% confidence interval. Was the true value of $\int_0^\infty \heartsuit(\theta)$ within the 95% confidence interval?

In [7]:
```
print('Standard error of the MC estimate: ', np.std(MC_integral_estimates))

MC_estimate_percentile = np.percentile(MC_integral_estimates, [2.5, 97.5])
print('The 95% confidence interval: [{}, {}] '.format(MC_estimate_percentile[0], MC_estimate_percentile[1]))

print('True exact integral value: ', exact_integral)
```

```
Standard error of the MC estimate:  0.025836322654947196
The 95% confidence interval: [0.21116177650256676, 0.31383570365739005]
True exact integral value:  0.26418924198235927
```

The standard error of the estimates is 0.026, and the true value 0.264 is within the 95% confidence interval $[0.211, 0.314]$.

1.7. It turns out that an appropriately chosen change of variables will allow you to estimate the integral on the part of the domain you truncated in 1.2 and 1.3. Execute this change of variables and use monte carlo integration to evaluate $\int_M^\infty \heartsuit(\theta)$

Let $x = \frac{1}{\theta}$, then

$$\int_M^\infty \heartsuit(\theta)d\theta = \int_{\frac{1}{M}}^0 \frac{(sin\frac{1}{x})^{2n}}{\frac{1}{x^2}} d(\frac{1}{x}) = \int_0^{\frac{1}{M}} (sin\frac{1}{x})^{2n} dx$$

Therefore, we can apply the MC method on the bounded domain $[0, \frac{1}{M}]$, on the function $(sin\frac{1}{x})^{2n}$. (n = 12)

In [8]:
```
def transformed_simulate_heart_collection(n, M, N=100000):
    x_unif_new = np.random.uniform(low=1e-9, high=1/M, size=N)
    heart_x_unif_new = np.power(np.sin(1/x_unif_new), 2*n)
    return (1/M) * np.sum(heart_x_unif_new) / N

np.random.seed(1)
MC_integral_new = transformed_simulate_heart_collection(n=12, M=1001)
print('the MC estimate: ', MC_integral_new)
```

```
the MC estimate:  0.0001625818011973657
```

In [9]:
```
# repeat estimation for 1000 times
np.random.seed(1)
n_sim = 1000
MC_integral_estimates_new = np.zeros((n_sim,))
for i in range(n_sim):
    MC_integral_estimates_new[i] = transformed_simulate_heart_collection(n=12, M=200)

np.mean(MC_integral_estimates_new)
```

Out[9]:
```
0.0008016279818675977
```

1.8. Based on your answer in 1.7 was your choice of M justified?

It is observed that the using $M = 1001$ produces $\int_M^\infty \heartsuit(\theta)d\theta \approx 0.0001$, almost 10x smaller than 0.001. And we tested to have observed that $M \approx 200$ can already achieve an absolute error within 0.001.

That being said, our choice of $M > 1000$ is still justified in that this guarantees a loose bound for the absolute error $<$ 0.001.

Question 2: Rally to Me!

Some Coding required

Suppose you observe the following data set $\mathbf{x}^{(0)} = (0.5, 2.5), \mathbf{x}^{(1)} = (3.2, 1.3), \mathbf{x}^{(2)} = (2.72, 5.84), \mathbf{x}^{(3)} = (10.047, 0.354)$. By convention, for any vector $\mathbf{x}$, we will denote the first component of $\mathbf{x}$ by x_1 and the second component by x_2. Suppose that the data is drawn from the same two-dimensional probability distribution with pdf f_X, that is, $\mathbf{x}^{(i)} \overset{iid}{\sim} f_X$, where

$$f_X(\mathbf{x}) = 4\lambda_1^2 x_1 x_2 \exp\left\{-\lambda_0(x_1^2 + x_2^2)\right\}.$$

You should assume that $\lambda_1, \lambda_0 > 0$ and that f_X is supported on the nonnegative quandrant of $\mathbb{R}^2$ (i.e. f_X is zero when either component is negative).

2.1. What are the values for λ_0 and λ_1 that maximize the likelihood of the observed data? **Support your answer with full and rigorous analytic derivations.**

2.2. Visualize the data along with the distribution you determined in 2.1 (in two dimensions or three).

Answers

2.1. What are the values for λ_0 and λ_1 that maximize the likelihood of the observed data?

$$f_X(\mathbf{x}) = 4\lambda_1^2 x_1 x_2 \exp\left\{-\lambda_0(x_1^2 + x_2^2)\right\}.$$

To make f_X a valid distribution density, we need

$$\iint f_X(\mathbf{x})dx_1 dx_2 = 1$$

Take polar coordinates transformation, $x_1 = rcos\theta$, $x_2 = rsin\theta$, the equivalent region for the nonnegative quandrant of $\mathbb{R}^2$ is

$$\begin{cases} r > 0 \\ 0 < \theta \leq \frac{\pi}{2} \end{cases}$$

$$\iint f_X(\mathbf{x}) dx_1 dx_2 = \iint 4\lambda_1^2 r^2 sin\theta cos\theta e^{-\lambda_0 r^2} r d\theta dr$$
$$= 4\lambda_1^2 \int_0^{\frac{\pi}{2}} sin\theta cos\theta d\theta \int_0^{\infty} r^2 e^{-\lambda_0 r^2} r dr$$
$$= 4\lambda_1^2 \int_0^{\frac{\pi}{2}} sin\theta d(sin\theta) \frac{1}{2} \int_0^{\infty} r^2 e^{-\lambda_0 r^2} d(r^2)$$
$$= (\lambda_1^2 sin^2\theta \Big|_0^{\frac{\pi}{2}}) * \frac{1}{-\lambda_0} (r^2 e^{-\lambda_0 r^2} \Big|_0^{\infty} - \int_0^{\infty} e^{-\lambda_0 r^2 d(r^2)})$$
$$= \frac{\lambda_1^2}{-\lambda_0} (0 - \frac{1}{-\lambda_0} e^{-\lambda_0 r^2} \Big|_0^{\infty}) = \frac{\lambda_1^2}{\lambda_0^2} = 1$$

Therefore $\lambda_0 = \lambda_1 = \lambda > 0$. The likelihood of the data is

$$L(D) = \prod_{i=1}^{N} f_X(x_1, x_2)$$

Take log,

$$\log [L(D)] = \sum_{i=1}^{N} \log [4\lambda^2 x_{i1} x_{i2} e^{-\lambda(x_{i1}^2 + x_{i2}^2)}]$$
$$= N \log (4\lambda) + \sum_{i=1}^{N} \log (x_{i1} x_{i2}) - \lambda \sum_{i=1}^{N} (x_{i1}^2 + x_{i2}^2)$$
$$\frac{\partial \log [L(D)]}{\partial \lambda} = \frac{2N}{\lambda} - \sum_{i=1}^{N} (x_{i1}^2 + x_{i2}^2) = 0$$

Therefore,

$$\lambda_{MLE} = \frac{2N}{\sum_{i=1}^{N} (x_{i1}^2 + x_{i2}^2)}$$

In [10]:
```
N = 4
X = np.array([
    [0.5, 2.5],
    [3.2, 1.3],
    [2.72, 5.84],
    [10.047, 0.354]
])

lambda_mle = 2*N / np.sum(X**2)
lambda_mle
```

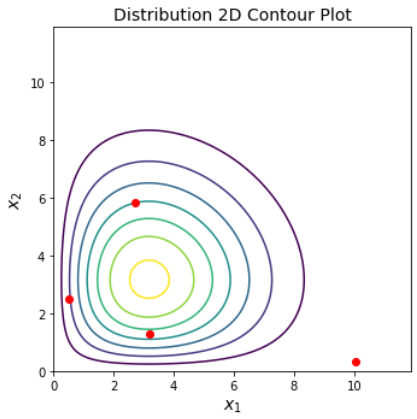
Out[10]: 0.049688970337392765

2.2. Visualize the data along with the distribution you determined in 2.1 (in two dimensions or three).

In [11]:

```
x_range = np.arange(0, 12, 0.1)
y_range = np.arange(0, 12, 0.1)
xx, yy = np.meshgrid(x_range, y_range)
zz = 4*lambda_mle**2*xx*yy*np.exp(-lambda_mle*(xx**2+yy**2))

fig, ax = plt.subplots(1, 1, figsize=(5, 5))
ax.contour(xx, yy, zz)
ax.plot(X[0,0], X[0,1], 'ro')
ax.plot(X[1,0], X[1,1], 'ro')
ax.plot(X[2,0], X[2,1], 'ro')
ax.plot(X[3,0], X[3,1], 'ro')
ax.set_xlabel(r'$x_1$', fontsize=14)
ax.set_ylabel(r'$x_2$', fontsize=14)
ax.set_title('Distribution 2D Contour Plot', fontsize=14)
plt.tight_layout()
```



Question 3: Still Missing!

Coding required

Recall from Homework 1 Question 2 that we explored working with missing data using the wine quality dataset from the UCI Machine Learning Repository. Re-Read the data in the `wine_quality_missing.csv` into a pandas dataframe and store the dataframe in the variable `wine_df` .

3.1. Drop impute `wine_df` and re-calculate estimates of the mean and standard deviation of the values of the Ash feature in the drop imputed dataset.

3.2. Use non-parametric bootstrap on the drop imputed dataset to find the standard errors for both your mean and standard deviation estimates.

3.3. Mean impute `wine_df` and re-calculate estimates of the mean and standard deviation of the values of the Ash feature in the mean imputed dataset.

3.4. Use non-parametric bootstrap on the mean imputed dataset to find the standard errors for both your mean and standard deviation estimates.

3.5. Compare the standard errors between the two different types of imputation. Do they differ? If so what might be the cause of the difference?

Answers

3.1. Drop impute `wine_df` and re-calculate estimates of the mean and standard deviation of the values of the Ash feature in the drop imputed dataset.

In [12]:
```
# read data
wine_df = pd.read_csv('wine_quality_missing.csv')

# before dropping nans
print('=== Before drop imputation ===')
print('# observations:', wine_df.shape[0])
print('mean = {}, std = {}'.format(wine_df['Ash'].mean(), wine_df['Ash'].std()))

# drop nan
wine_drop = wine_df.dropna()
print('\n=== After drop imputation ===')
print('# observations:', wine_drop.shape[0])

# mean, std
print('mean = {}, std = {}'.format(wine_drop['Ash'].mean(), wine_drop['Ash'].std()))
```

```
=== Before drop imputation ===
# observations: 178
mean = 2.366217948717948, std = 0.2832059388586288

=== After drop imputation ===
# observations: 43
mean = 2.369767441860465, std = 0.3032324926999321
```

After drop imputation,

mean of Ash: 2.369767441860465
standard deviation of Ash: 0.3032324926999321

3.2. Use non-parametric bootstrap on the drop imputed dataset to find the standard errors for both your mean and standard deviation estimates.

In [13]:
```
np.random.seed(2)
ash_drop_imputed_np_samples = np.random.choice(wine_drop['Ash'].values,
                                               size=(10000, wine_drop.shape[0]),
                                               replace=True)
print('Standard error for mean estimates: ', np.std(np.mean(ash_drop_imputed_np_samples, axis=1)))
print('Standard error for std estimates: ', np.std(np.std(ash_drop_imputed_np_samples, axis=1)))
```

```
Standard error for mean estimates:  0.04607327866876724
Standard error for std estimates:  0.037225828025113046
```

After drop imputation, the standard error from non-parametric bootstrapping are:

for mean: 0.04607327866876724
for standard deviation: 0.037225828025113046

3.3. Mean impute `wine_df` and re-calculate estimates of the mean and standard deviation of the values of the Ash feature in the mean imputed dataset.

In [14]:
```
# mean imputation
wine_mean_imputed = wine_df.fillna(np.mean(wine_df))

print('=== After mean imputation ===')
print('# observations:', wine_mean_imputed.shape[0])

# mean, std
print('mean = {}, std = {}'.format(wine_mean_imputed['Ash'].mean(), wine_mean_imputed['Ash'].std()))
```

```
=== After mean imputation ===
# observations: 178
mean = 2.36621794871795, std = 0.2650217834532965
```

After mean imputation,

mean of Ash: 2.36621794871795
standard deviation of Ash: 0.2650217834532965

3.4. Use non-parametric bootstrap on the mean imputed dataset to find the standard errors for both your mean and standard deviation estimates.

In [15]:
```
np.random.seed(2)
ash_mean_imputed_np_samples = np.random.choice(wine_mean_imputed['Ash'].values,
                                               size=(10000, wine_mean_imputed.shape[0]),
                                               replace=True)
print('Standard error for mean estimates: ', np.std(np.mean(ash_mean_imputed_np_samples, axis=1)))
print('Standard error for std estimates: ', np.std(np.std(ash_mean_imputed_np_samples, axis=1)))
```

```
Standard error for mean estimates:  0.019806498826119158
Standard error for std estimates:  0.018701591387785O9
```

After mean imputation, the standard error from non-parametric bootstrapping are:

for mean: 0.019806498826119158
for standard deviation: 0.018701591387785O9

3.5. Compare the standard errors between the two different types of imputation. Do they differ? If so what might be the cause of the difference?

Mean imputation gives smaller standard error than drop imputation for both mean and standard deviation estimates. Both mean and drop imputations keep the imputed mean unchanged, but mean imputation reduces the variance for the same amount of data by artificially replacing the missing observations with the same mean value. This causes the bootstrapped samples to have less variance in both their mean and standard deviation estimates.

In addition, there are more observations (rows) in the mean-imputed dataframe compared to the drop-imputed dataframe. Therefore, there are more bootstrapped samples per simulation when bootstrapping from the mean-imputed dataframe compared to when bootstrapping from the drop-imputed dataframe. Based on central limit theorem, the standard error of the samples in a given simulation scales with a factor of $\frac{1}{\sqrt{n}}$. Since n is bigger in the mean-imputed dataframe, the variance of estimates from the mean-imputed dataframe is smaller.