Random Signal Analysis

October 24, 2023

Variables and Conditional statements

Operations over sets

A set is a list with immutable elements, which means that, once declared, an element cannot be changed or modified. However, elements from the set can be added/removed.

```
# using set() function
print(set("man"), 'size = ',len(set("man")))
# using curly brace {}
print({"Apples", ("Bananas", "Oranges")}, 'size = ',len({"Apples", ("Bananas", "Oranges")}))
print(set(("man", 33, ('a', 2, 3))), 'size = ',len(set(("man", 33, ('a', 2, 3)))))
------------------------------------------------------------------------------------
{'n', 'a', 'm'} size = 3
{('Bananas', 'Oranges'), 'Apples'} size = 2
{'man', 33, ('a', 2, 3)} size = 3
```

```
fruit_set = ["Apples", "Bananas", "Oranges"]
print('type=',type(fruit_set),'\n size= ',len(fruit_set))
for i in range(len(fruit_set)):
print(i+1,' -> ',fruit_set[i])
------------------------------------------------------------------------------------
type= <class 'list'>
size= 3
1 -> Apples
2 -> Bananas
3 -> Oranges
```

Adding and removing elements

```
fruit_set.append(("Tomatoes"))
print('adding Tomatoes: ',fruit_set)
fruit_set.remove(("Tomatoes"))
print('removing Tomatoes: ',fruit_set)
------------------------------------------------------------------------------------
adding Tomatoes: ['Apples', 'Bananas', 'Oranges', 'Tomatoes']
removing Tomatoes: ['Apples', 'Bananas', 'Oranges']
```

`00aSetTheory` completar Ejercicios

data visualization

```
import matplotlib.pyplot as plt  # data plotting
# matplotlib.pyplot is a state-based interface to an implicit
# figure GUI manager, MATLAB-like, way of plotting
from matplotlib import pyplot
from matplotlib import rcParams

rcParams['figure.figsize'] = [15,9] # A4 format - [inches]
rcParams['lines.markersize'] = 6      # marker size in points
plt.rcParams['axes.labelsize'] = 12
plt.rcParams['axes.titlesize'] = 12
```

Probabilistic measures

OUTPUT FORMAT OF RANDOM SIGNAL ANALYSIS

01aBayesRule• completar Ejercicios

```
# import the required libraries
import numpy as np
# Python library used for working with arrays
import random
# This module implements pseudo-random number generators for various distributions.

Sample = 36 # 1024, 8192 testing values

# binary data generation
Px0 = random.uniform(0.01, .99)  # probability generation of P(xi=0)
Py0 = random.uniform(0.01, .99)  # probability generation of P(yi=0)
# X set
X = np.random.choice([0, 1], size=(Sample,), p=[Px0, 1 - Px0])
print("set X: ", X)
# Y set
Y = np.random.choice([0, 1], size=(Sample,), p=[Py0, 1 - Py0])
print("set Y: ", Y)

# Setting initial values
p_00 = p_01 = p_10 = p_11 = 0
p_0 = p0_ = p_1 = p1_ = 0

# Estimation of frequencies
for i in range(Sample):
  if X[i] == 0 and Y[i] == 0:
    p_00 = p_00 +1
  if X[i] == 0 and Y[i] == 1:
    p_01 = p_01 +1
  if X[i] == 1 and Y[i] == 0:
    p_10 = p_10 +1
  if X[i] == 1 and Y[i] == 1:
    p_11 = p_11 +1
  if X[i] == 0:
    p_0 = p_0 +1
  if Y[i] == 0:
    p0_ = p0_ +1
  if X[i] == 1:
    p_1 = p_1 +1
  if Y[i] == 1:
    p1_ = p1_ +1

# marginal probabilities
p_0 = p_0/Sample ; p0_ = p0_/Sample ; p_1 = p_1/Sample ; p1_ = p1_ /Sample
print('Marginal probabilities')
print('P(X=0) = %4.3f'%(Px0),': ...
#syntax for a format placeholder is %[flags][width][.precision]type

print('P(Y=0) = ', Py0,...
    ....

-----------------------------------------------------------------------------------
Marginal probabilities
P(X=0) = 0.090  ...
P(Y=0) = 0.8169013423663365 ...
 ....
```

Symbolic Solution for statistical computing

SciPy is an open source library for scientific computing in Python built on top of NumPy

Symbolic Solution by `scipy.stats`: Calculation of *n-th* moments about the mean for a sample

```
from scipy.stats import moment

arr = [.1, -.2, .3, -.4, .5]
m1= moment(arr, moment=1); m2= moment(arr, moment=2)
print('m1= ',m1,'m2= ',m2)
print("Mean > Median: ",round(np.mean(arr),3) > round(np.median(arr),3))

m1= 0.0 m2= 0.10640000000000001
Mean > Median: False
```

```
from scipy import stats

# Binomial Random variable
X = stats.binom(10, 0.2) # Declare X to be a binomial random variable
print(X.pmf(3))      # P(X = 3)
print(X.cdf(4))      # P(X <= 4)
print(X.mean())      # E[X]
print(X.var())      # Var(X)
print(X.std())      # Std(X)
print(X.rvs())      # Get a random sample from X
print(X.rvs(10))      # Get 10 random samples form X
```

Characteristic Function of Gaussian pdf

```
# Symbolic Solution
from sympy.stats import *
from sympy import simplify, exp ,I
import sympy as sp

mu = sp.symbols('mu', bounded=True)
sigma = sp.symbols('sigma', positive=True, bounded=True)
t = sp.symbols('t', positive=True)
X = Normal('X', mu, sigma) # Normal random variable
simplify(E(exp(I*t*X))) # Expectation of exp(I*t*X)

exp(t*(I*mu - sigma**2*t/2))
```

Calculate the nth moment about the mean for a sample.

```
def nmoment(x, counts, c, n):
return np.sum(counts*(x-c)**n) / np.sum(counts)
```

```
from scipy.stats import moment
moment([1, 2, 3, 4, 5], moment=1)
moment([1, 2, 3, 4, 5], moment=2)
```

Kernel Density Estimation: Given a sample of independent, identically distributed (i.i.d) observations $(x_1, x_2, \ldots, x_n)$ of a random variable from an unknown source distribution, the kernel density estimate, is given by:

$$(\hat{x}) = \frac{1}{nh}\sum_{j=1}^{n} K(\frac{x - x_j}{h})$$

where $K(a)$ is the kernel function and h is the smoothing parameter, or bandwidth.

Kernel Computation: Let $[-2, -1, 0, 1, 2]$ be the sample points with a linear kernel given by:

$$\begin{aligned}
K(a) = 1 - |a|/h, & \qquad h{=}10 \\
x_j & = \begin{bmatrix} -2 & -1 & 0 & 1 & 2 \end{bmatrix} \\
|0 - x_j| & = \begin{bmatrix} 2 & 1 & 0 & 1 & 2 \end{bmatrix} \\
|\frac{0 - x_j}{h}| & = \begin{bmatrix} .2 & .1 & 0 & .1 & .2 \end{bmatrix} \\
K\left(|\frac{0 - x_j}{h}|\right) & = \begin{bmatrix} .2 & .1 & 0 & .1 & .2 \end{bmatrix}
\end{aligned}$$

therefore, $p(0) = \frac{1}{(5)(10)}(0.8 + 0.9 + 1 + 0.9 + 0.8) = 0.088$

```python
# Comparing kernel functions
from statsmodels.nonparametric.kde import kernel_switch

list(kernel_switch.keys())

# Create a figure
fig = plt.figure(figsize=(18, 9))

# Enumerate every option for the kernel
for i, (ker_name, ker_class) in enumerate(kernel_switch.items()):

# Initialize the kernel object
kernel = ker_class()

# Sample from the domain
domain = kernel.domain or [-3, 3]
x_vals = np.linspace(*domain, num=2 ** 10)
y_vals = kernel(x_vals)

# Create a subplot, set the title
ax = fig.add_subplot(3, 3, i + 1)
ax.set_title('Kernel function "{}"'.format(ker_name))
ax.plot(x_vals, y_vals, lw=3, label="{}".format(ker_name))
ax.scatter([0], [0], marker="x", color="red")
plt.grid(True, zorder=-5)
ax.set_xlim(domain)

plt.tight_layout()
```

12g2KDEComparison

Point Estimation

Method of Moments

Estimating rule: $\tilde{m}_{nx}$

```
expected_value = lambda values: sum(values) / len(values) # mean estimation
standard_deviation = lambda values, expected_value: np.sqrt(sum([(v - expected_value)**2 for v in
                                                     values]) / len(values)) # std estimation
```

population with a specific distribution:

```
np.random.seed(1) #
Nmodel = 2**16 # a large size simulating the theorical pdf model
# $\theta_i= mu, sigma $ - parameters of pdf
mu, sigma = 39, 1 # actual values of $\theta_i$
population = np.random.normal(mu, sigma, Nmodel) # model simulation of pdf
### population = np.random.poisson(mu, Nmodel) # model simulation of pdf, fix mu=1

# $\tilde{m}_{nx}$ estimates of $\theta_i$
mean = expected_value(population)
std = standard_deviation(population, mean)
# Histogram of population
Nbins = int(np.ceil(1 + 3.3*np.log(Nmodel))) # Sturge's Rule K = 1 + 3. 322 logN
```

Generic calculation for d parameters of the population distribution function:

```
# 1- Compute all n-moments of the sample, n=1,...,d.
#    Fix the sample size
Nsample = 2**6 # 2**4,5,6,7,8 - tested values of Nsample
Nbins = int(np.ceil(1 + 3.3*np.log(Nsample)))
#    selected sample data (random choise)
randomly_selected_items = [choice(population) for _ in range(Nsample)]
# 3- Calculate the population distribution parameters by solving equations
#    using the previously computed n-moments.
#    see the estimating rule for d= 1 (expected_value), 2(standard_deviation)

mean = expected_value(randomly_selected_items)          # n=1
s_d = standard_deviation(randomly_selected_items, mean) # n= 2
```

fitting of sampled values

```
xs = np.arange(mu-1.5*sigma, mu +1.5*sigma, 0.001)
# for comparison, the true Gaussian pdf is also plotted
actual_ys = norm.pdf(xs, mu, sigma)
ys = norm.pdf(xs, mean, s_d)
```

12dMomentsMethodEstimator

Information Metrics

Entropy of binary symbols

```
import numpy as np
from scipy.stats import entropy

base = 2 # work in units of bits
pk = np.array([1/2, 1/2]) # fair coin
H = entropy(pk, base=base)
print('H=', H)

H= 1.0 one bit!!!

H == -np.sum(pk * np.log(pk)) / np.log(base) ####

True

print(entropy(np.array([99/100, 1/100]), base=base)) # biased coin

0.08079313589591118 less than a bit

print(entropy(np.array([1/100, 99/100]), base=base)) # biased coin

0.08079313589591118 less than a bit
```

Kernel Density-based Entropy 21aEntropy

```
def KDEConditionalE(X,Y):
# Calculate the conditional entropy
X = norMaxMin(X); X = X.reshape(-1,1)
Y = norMaxMin(Y); Y = Y.reshape(-1,1)
# Define the range for X and Y values
x_range = np.linspace(X.min(), X.max(), 64)
y_range = np.linspace(Y.min(), Y.max(), 64)
params = {'bandwidth': x_range}
gs = GridSearchCV(KernelDensity(), params)
#Exhaustive search over specified params for Kernel estimator.
gs.fit(X)
kde_X=gs.best_estimator_

# Create a meshgrid for X and Y values
xx, yy = np.meshgrid(x_range, y_range)
xy = np.column_stack([xx.ravel(), yy.ravel()])
# Evaluate the KDE for X at the given meshgrid points
log_density_x = kde_X.score_samples(xy[:, 0].reshape(-1, 1))
# Fit a kernel density estimator to the data for (X, Y)
params = {'bandwidth': np.logspace(0, 1, 64)}
gs = GridSearchCV(KernelDensity(), params)
#Exhaustive search over specified params for Kernel estimator.
gs.fit(np.column_stack([X, Y]))
kde_joint=gs.best_estimator_
# Evaluate the KDE for (X, Y) at the given meshgrid points
log_density_joint = kde_joint.score_samples(xy)

# Calculate the conditional probability distribution p(Y|X) using KDE
log_conditional_density = log_density_joint - log_density_x
# Normalize the conditional density to get a probability distribution
conditional_density = np.exp(log_conditional_density)
conditional_density /= conditional_density.sum()

# Calculate the conditional entropy H(Y|X)
conditional_entropy = entropy(conditional_density, base=2)
return  conditional_entropy
```

Stationarity Analysis

Linear Responses to Stationary Signals

Scipy Circuit design using scipy package 22dRCLowPass

```
from scipy.fftpack import fft, ifft, fftfreq, fftshift
from scipy import signal

# Define the Filter function that corresponds to the low pass RC filter.
def Filter(f,R,C):
omega = 2*np.pi*f
vout=( 1./(1j*R*omega*C+1.))
return(vout)
```

Parameter set-up

```
# Desired order of the filter
N = 3 #[3,9,33,90]

# Use the 'buttap' function to generate the zeros, poles, and gain of the filter
z, p, k = signal.buttap(N)

# get the filter zeros, poles, and gain as 'f1'
f1 = signal.buttap(N)

# get the magnitude and phase response using the 'bode' function
w, mag, phase = signal.bode(f1)
```

Cut-off frequency

```
def find_nearest(array, value):
"""arguments: the input array and the search value
outputs :  the closet array value and its index in the array """
array = np.asarray(array)
idx = (np.abs(array - value)).argmin()
return array[idx], idx
```

Lcapy Symbolic linear circuit analysis using SymPy package. 22dLCapy

```
from lcapy import Circuit, j, omega, s, t

cct = Circuit("""
Vi 1 0_1 step; down
R 1 2; right, size=1.5
C 2 0; down
W 0_1 0; right
W 0 0_2; right, size=0.5
P1 2_2 0_2; down
W 2 2_2;right, size=0.5""")

H(j * omega)

from numpy import logspace
w = logspace(1, 6, 500)
ax = H1(j * omega).dB.plot(w, log_frequency=True)
```

Lcapy: symbolic linear circuit analysis with Python. PeerJ Comput Sci. 2022; 8: e875.

https://www.ncbi.nlm.nih.gov/pmc/articles/PMC9044395/

$\mathcal{L}_2$-based Linear Filtering

Wiener Scipy filter `23aWienerFil`

```
from scipy import signal

plt.plot(x, label='Original signal')
plt.plot(signal.medfilt(x), label='medfilt: median filter')
plt.plot(signal.wiener(x), label='wiener: wiener filter')
plt.xlim(0,Nsample-1)
plt.legend(loc='best')
```

Kalman KalmanFilter package `23aKalmanFilter`

```
from pykalman import KalmanFilter
import numpy as np
import pandas as pd
from scipy import poly1d

# Construct a Kalman filter
kf = KalmanFilter(transition_matrices = [1],
observation_matrices = [1],
initial_state_mean = 0,
initial_state_covariance = 1,
observation_covariance=1,
transition_covariance=.0001)

mean, cov = kf.filter(x.values)
mean = pd.Series(mean.flatten(), index=x.index)

# Compute the rolling mean with various lookback windows
mean30 = x.rolling(window = 30).mean()
mean60 = x.rolling(window = 60).mean()
```

Matched Filter scipy.signal `23aMatchedFilter`

```
from scipy.signal import correlate

# Perform cross-correlation using the matched filter
correlation1 = correlate(xi_hat, h1, mode='same')
correlation0 = correlate(xi_hat, h0, mode='same')

# Find the index with the maximum correlation
max_correlation_index1 = np.argmax(correlation1)
```

Signal Detection

Quantization

Uniform Quantization 31aQuantizer

```
def quantize_uniform(x, quant_min=-1.0, quant_max=1.0, quant_level=5):
"""Uniform quantization approach
Args:
x (np.ndarray): Original signal
quant_min (float): Minimum quantization level (Default value = -1.0)
quant_max (float): Maximum quantization level (Default value = 1.0)
quant_level (int): Number of quantization levels (Default value = 5)

Returns:
x_quant (np.ndarray): Quantized signal
"""
x_normalize = (x-quant_min) * (quant_level-1) / (quant_max-quant_min)
x_normalize[x_normalize > quant_level - 1] = quant_level - 1
x_normalize[x_normalize < 0] = 0
x_normalize_quant = np.around(x_normalize)
x_quant = (x_normalize_quant) * (quant_max-quant_min) / (quant_level-1) + quant_min
return x_quant
```

Nonuniform Quantization

```
def encoding_mu_law(v, mu=255.0):
"""mu-law encoding

Notebook: C2/C2S2_DigitalSignalQuantization.ipynb

Args:
v (float): Value between -1 and 1
mu (float): Encoding parameter (Default value = 255.0)

Returns:
v_encode (float): Encoded value
"""
v_encode = np.sign(v) * (np.log(1.0 + mu * np.abs(v)) / np.log(1.0 + mu))
return v_encode

def decoding_mu_law(v, mu=255.0):
"""mu-law decoding

Notebook: C2/C2S2_DigitalSignalQuantization.ipynb

Args:
v (float): Value between -1 and 1
mu (float): Dencoding parameter (Default value = 255.0)

Returns:
v_decode (float): Decoded value
"""
v_decode = np.sign(v) * (1.0 / mu) * ((1.0 + mu)**np.abs(v) - 1.0)
return v_decode
```

Threshold of Error

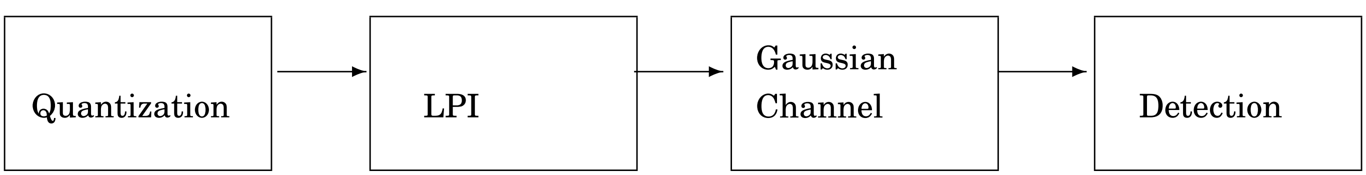
31bThresholdError

```
@interact(u=(umbral[0], umbral[-1], 0.1))
def umbral_interact(u=0):

plt.figure()
ax1 = plt.subplot2grid((3, 1), (0, 0), rowspan=2)
plt.grid(True)

plt.fill_between(x[x>u], y1[x>u], color='C0', zorder=90)
plt.fill_between(x[x>u], y0[x>u], color='C0', zorder=90)

plt.fill_between(x[x<u], y1[x<u], color='C1', zorder=90)
plt.fill_between(x[x<u], y0[x<u], color='C1', zorder=90)

plt.plot(x, y1, linestyle='--', color='w', zorder=99)
plt.plot(x, y0, linestyle='--', color='w', zorder=99)

plt.vlines(u, 0, 0.12, color='k', linestyle='--', zorder=100)

plt.ylabel('')

ax2 = plt.subplot2grid((3, 1), (2, 0))
plt.grid(True)

xerror = np.linspace(x[0], x[-1], len(error))
error_ = np.array(error)
error_value = error[(abs(x-u)).argmin()]

plt.plot(xerror, error_, color='C3', label=f'Error: {error_value:0.2f}')
plt.vlines(u, error_.min(), error_.max(), color='k', linestyle='--')
plt.legend()

plt.ylabel('Error')
plt.xlabel('threshold')
```

d' and ROC curve   31cDiscriminator

```
# Computation  of d-prime for some signal and noise
d_prime = (m_signal - m_noise) / np.sqrt(0.5 * (s_signal ** 2 + s_noise ** 2))
print(f"d' (d-prime) = {d_prime:.2f}")

# ROC computation
thresholds = np.sort(np.concatenate((signal_data, noise_data)))
tpr = [np.sum(signal_data >= threshold) / len(signal_data) for threshold in thresholds]
fpr = [np.sum(noise_data >= threshold) / len(noise_data) for threshold in thresholds]
```

Binary Channel   31cBinaryChannel



Channel simulation

NN frameworks

Keras is an API in Python, running on top of end-to-end, open-source machine learning platform – TensorFlow. Creating a Sequential model:

```
from tensorflow.keras.models import Sequential
model = Sequential(name="longaniza")
```

declare the sequence of by stacking/removing layers [add/pop]

```
from tensorflow.keras.models import Sequential
model = Sequential()
model.add(...) # input layer
model.add(...) # hidden layer
.....
model.pop()
model.add(...) # output layer
```

Model inputs: input arrangement (64 - inputs) and attributes

```
Dense(units=64, input_shape=(8,),..., activation='relu', name="layer1")
Weight Initialization: [ random_uniform, random_normal,zeros ]
```

activation function: softmax, rectified linear (relu), tanh, sigmoid, ...

Layer Types: Dense, Dropout, Concatenate

```
from tensorflow.keras.layers import Dense

model.add(Dense(units=64, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(units=10, activation='softmax'))
model.add(Dropout(0.1))
x = np.arange(20).reshape(2, 2, 5)
y = np.arange(20, 30).reshape(2, 1, 5)
concat = tf.keras.layers.Concatenate(axis=1)([x, y])
model.add(concat)
```

Model Compilation

```
model.compile(optimizer='sgd',..., loss='mse'..., metrics=...)
```

Model Optimizers: SGD, RMSprop, Adam

Loss Functions: 'mse',

```
from keras import losses
from keras import optimizers
from keras import metrics
model.compile(loss = 'mean_squared_error',
optimizer = 'sgd', metrics = [metrics.categorical_accuracy])
```

```
model.compile(loss=tf.keras.losses.categorical_crossentropy,
optimizer=tf.keras.optimizers.SGD(
learning_rate=0.01, momentum=0.9, nesterov=True))
```

Model Training

```
# x_train and y_train are Numpy arrays
model.fit(x_train, y_train, epochs=5, batch_size=32)
```

Epochs refer to the number of times the model is exposed to the training dataset.

Batch Size is the number of training instances shown to the model before a weight update is performed.

An observation sequence $x{=}[x_t{:}t \in T]$ must be partitioned into multiple segments (samples), lasting $L{<}T$, from which the model can learn.

$$
\begin{aligned}
x &\to \tilde{y} \\
x_{L+1-n}, \dots, x_{2L-n} &\to x_{2L+1-n}: && \text{one-step prediction} \\
x_{L+1-n}, \dots, x_{2L-n} &\to x_{2L+1-n}, \dots, x_{2L+1-n+m}: && m\text{-step prediction}
\end{aligned}
$$

Model output performance:

- model.evaluate(): To calculate the loss values for the input data

```
loss_and_metrics = model.evaluate(x_test, y_test, batch_size=128)
```

- model.predict(): To generate network output for the input data

```
classes = model.predict(x_test, batch_size=128)
```

Summarize the Model

- displaying a model summary

```
model.summary()
```

- retrieving a summary of the model configuration

```
model.get_config()
```

- create an image of your model structure

```
from tensorflow.keras.utils import plot_model
plot(model, to_file='model.png')
```

Simple deep MLP with Keras!!!!

https://www.kaggle.com/code/fchollet/simple-deep-mlp-with-keras/script