KTH Matematik
Matematisk Statistik

SF1912 Sannolikhetsteori och statistik, HT 2025
Laboration 1 för CINTE3

1 Introduktion

Denna demonstration är inte poänggivande, men utgör en förberedelse för den andra laborationen på kursen, vilken kan ge bonuspoäng på tentamen. Syftet med denna laboration är dels att ge en introduktion till att arbeta med statistik i Python, dels att ge djupare förståelse för några viktiga begrepp i kursen genom att illustrera dem med hjälp av Python.

Denna laboration kan utföras antingen på egen hand eller vid det schemalagda laborationstillfället. Om du väljer det senare alternativet, så kan det vara bra att ha läst igenom specifikationen två gånger och ha börjat försöka lösa uppgifterna före laborationstillfället. Fokusera när du gör labben på att lära dig att använda Python och se till att du förstår de kommandon som du använder.

2 Jupyter Notebook

Det finns flera sätt att interagera med Python (genom kommandotolken eller en integrerad utvecklingsmiljö). För laborationerna i kursen väljer vi att använda Jupyter Notebook, ett grafiskt webbaserat användargränsnitt med möjlighet att visa plottar och resultat interaktivt med hjälp av Javascript Jupyter Notebook finns installerat på de flesta datorer på KTH och kan laddas ner gratis via internet och installeras på andra datorer. Användargränsnittet har fördelen med att det ser mer eller mindre likadant ut oberoende av var den körs.

Antingen startar du Jupyter genom att klicka på ikonen om en sådan är tillgänglig. Annars startas Jupyter från en vanlig kommandotolk genom att skriva `jupyter notebook`. Detta startar en lokal webbserver och pekar din webbläsare till servern. Säkerhetsinställningarna är sådana att bara du kan komma åt webbservern – den är inte synlig för andra datorer i samma nätverk.

För att öppna en ny notebook trycker du på ”New” i högra hörnet av fillistan och väljer ”Python 3” (Jupyter kan användas tillsammans med andra

programmeringsspråk, som till exempel Julia). Nu öppnas en notebook med en tom cell markerad `In [1]`. Här kan du börja skriva Pythonkod som sedan exekveras när du trycker Ctrl-Enter (du kan också trycka Shift-Enter, vilket exekverar koden och flyttar markören till en ny cell).

Vi börjar med ett enkelt räkneexempel

```
In [1]:     3 * 11.5 + 2.3 ** 2 / 4

Out [1]:    35.558
```

Om den sista raden är ett uttryck som ovan eller ett variabelnamn skriver Jupyter ut det. I andra fall måste vi tvinga fram en utskrift genom att använda funktionen `print`.

```
In [2]:     a = 1
            print(a)
            b = 2.12

            1
```

Detta tilldelar `a` värdet 1, skriver ut dess värde och tilldelar `b` värdet 2.12.

Python har inbyggda funktioner för grundläggande matematiska operationer, som vi ser ovan, men för mer avancerade funktioner behöver vi importera moduler. I standardbiblioteket finns modulen `math` men denna har ett ganska begränsat utbud. Istället kommer vi använda det mer kraftfulla NumPy genom modulen `numpy`, vilken oftast importeras genom aliaset `np`:

```
In [3]:     import numpy as np
```

Vi kan nu anropa NumPy-funktioner genom `np.` följt av funktionsnamnet:

```
In [4]:     np.sqrt(36)

Out [4]:    6.0
```

NumPy inkluderar bara grundläggande matematiska funktioner med fokus på linjär algebra. För statistiska beräkningar behöver vi SciPy genom modulen `scipy.stats`, vilken vi importerar genom

```
In [5]:     from scipy import stats
```

Till sist behöver vi biblioteket Matplotlib för att plotta grafer och importerar därför `matplotlib.pyplot` och ger den aliaset `plt` med

```
In [6]:     import matplotlib.pyplot as plt
```

En plottningsfunktion vi behöver saknas i Matplotlib (närmare bestämt plottning av tvådimensionella normalfördelningar). Denna finns tillgänglig på kurshemsidan i Pythonfilen `plotting.py`. Ladda ner denna och lägg den i samma katalog som din notebook. För att importera denna skriver vi

```
In [7]:     import plotting
```

Totalt har vi alltså fyra moduler som vi importerat: `np`, `stats`, `plt` och `plotting`. I koden nedan antar vi att dessa har importerats i början av notebooken.

Vektorer och matriser i NumPy skapar vi genom funktionen `np.array`:

```
In [8]:     x = np.array([1, 3, 7])
            y = np.array([2, 1, 8])
            z = np.array([[1, 3], [2, 4]])

            z

Out [8]:    array([[1, 3]
                   [2, 4]])
```

Vi har alltså två tredimensionella vektorer `x` och `y` samt en 2×2 matris `z`. Vektorer kan kombineras genom addition (`x + y`) eller elementvis multiplikation (`x * y`). Matrismultiplikation sker genom operatorn `@` eller funktionen `np.dot`.

NumPy innehåller också andra standardfunktioner, som `np.exp`, `np.log`, `np.sin`, `np.arcsin`, `np.cos`, `np.arccos`, `np.tan`, `np.arctan` och så vidare. Observera att `np.log` är den naturliga logaritmen (med avseende på e). Prova att plotta en funktion, t.ex. genom följande kommandon:

```
In [9]:     x = np.linspace(0.5, 2, 100)
            y = np.log(x)

            plt.plot(x, y)
            plt.show()
```

Den första raden skapar en vektor av 100 stycken linjärt utspridda värden mellan 0.5 och 2. I andra raden beräknas den naturliga logaritmen för varje x-värde. Fjärde raden säger till Matplotlib att plotta `x` och `y`, men denna visas inte förrän vi anropar `plt.show` i raden under (Jupyter är egentligen smart här och märker av om du har glömt `plt.show` och visar plotten ändå, men det är en god vana att skriva `plt.show`).

För att simulera stokastiska variabler använder vi modulen `stats` som vi importerat. Vi kan till exempel generera 6 utfall från binomialfördelningen med $n = 7$ och $p = 0.3$:

```
In [10]:    w = stats.binom.rvs(7, 0.3, size=6)

            w

Out [10]:   array([1, 3, 1, 3, 0, 6])
```

Här har vi anropat funktionen `stats.binom.rvs` (kortform av "random variates", dvs. slumpmässiga utfall), vilket tar som parametrar `n` och `p` samt antalet utfall genom `size`-parametern. Alternativt kan vi först definiera den stokastiska variabeln `X`, vilket fixerar parametrarna `n` och `p`, och sedan anropa `X.rvs`:

```
In [11]:    X = stats.binom(7, 0.3)
            w = X.rvs(size=6)

            w

Out [11]:   array([2, 2, 2, 3, 2, 2])
```

Fördelen är här att vi inte måste specificera parametrarna varje gång, men annars är anropen ekvivalenta (även om den senare varianten tar lite mer tid).

Om vi behöver hjälp finns det flera inbyggda hjälpfunktioner. En är standardbibliotekets `help`:

```
In [12]:    help(np.linspace)

            Help on function linspace in module numpy:

            linspace(start, stop, num=50, endpoint=True, ...)
```

Jupyter Notebook erbjuder också specialkommandot `?`:

```
In [13]:    ?np.linspace
```

Detta öppnar en liten flik med dokumentation. Mer dokumentation för NumPy, SciPy och Matplotlib finns också på deras respektive webbsidor:

- `https://www.numpy.org/doc/stable/`
- `https://docs.scipy.org/doc/scipy/reference/`
- `https://matplotlib.org/stable/users/index.html`

3 Simulering

Temat för den här datorlaborationen är simulering. Sannolikhetsteoridelen av kursen handlar om hur man genom beräkningar kan ta fram olika storheter som sannolikheter, väntevärden osv. för en given stokastisk modell. För mer komplicerade system är det ibland inte alls möjligt att göra exakta beräkningar, eller så är det så tidskrävande att man avstår.

I sådana sammanhang kan simulering vara ett alternativ. Simulering innebär att man med hjälp av en dator simulerar ett antal replikeringar av det

stokastiska systemet, och sedan använder t.ex. medelvärden eller empiriska kvantiler för att uppskatta de storheter man söker. I den här laborationen skall vi göra detta för några enklare problem men grundprinciperna går att använda på långt mer komplicerade problem som vi inte kan lösa med enkla beräkningar.

I det allra enklaste fallet kan det vara fråga om att uppskatta väntevärdet för en fördelning. Antag att vi har en fördelningsfunktion F och låt X vara en stokastisk variabel med denna fördelningsfunktion. Antag också att vi vill uppskatta tillhörande väntevärde, μ säg. Om vi nu drar observationer $x_1, x_2, \ldots, x_n$ från X kan vi uppskatta μ med hjälp av

$$\hat{\mu} = \frac{1}{n}\sum_{k=1}^{n} x_k. \tag{1}$$

Att detta är en rimlig uppskattning följer av stora talens lag. Om vi vill uppskatta fördelningsfunktionen $F(a) = P(X \leq a)$ för något tal a, så, kan vi göra detta genom att räkna ut hur stor andel av de n simulerade observationerna som är mindre än eller lika med a. Vi kan skriva detta som

$$\hat{F}(a) = \frac{\text{antal } x_k \text{ som är mindre än } a}{n} = \frac{1}{n}\sum_{k=1}^{n} \mathbb{I}(x_k \leq a). \tag{2}$$

Här är $\mathbb{I}$ en så kallad *indikatorfunktion*, som antar värdet 1 om villkoret inom parentes är uppfyllt och annars antar värdet 0. Alltså är $\mathbb{I}(x_k \leq a)$ lika med 1 precis för de k sådana att $x_k \leq a$, så att summan ovan räknar antalet index k som uppfyller villkoret.

4 Laborationsuppgifter

Gå igenom dessa uppgifter själv och se till att du förstår vad som händer. Kom ihåg att den här laborationen redovisas inte, så det är upp till dig att bestämma när du är färdig.

Problem 1 – Simulering av exponentialfördelade stokastiska variabler

SciPys funktion för exponentialfördelade stokastiska varialer är `stats.expon`. Använd gärna `help(stats.expon)`, `?stats.expon` eller webbdokumentationen för att ta reda på hur funktionen fungerar. Observera att SciPys `stats.expon` har parametern `scale`, vilket motsvarar inversen $1/\lambda$ av intensiteten λ som används i Blom, m.fl. [2]. Följande kod genererar N stycken Exp(1/10)-fördelade slumptal, ritar upp ett normaliserat histogram (`density=True` konverterar de absoluta frekvenserna i histogrammet till relativa frekvenser normaliserade med avseende på intervallstorleken), samt plottar den sanna täthetsfunktionen som jämförelse genom `stats.expon.pdf`.

```
mu = 10
N = 10000
y = stats.expon.rvs(scale=mu, size=N)
plt.hist(y, 80, density=True)

t = np.linspace(0, 100, N // 10)
pdf = stats.expon.pdf(t, scale=mu)
plt.plot(t, pdf, 'red')

plt.show()
```

Vad betyder parametern 80 till `plt.hist`? Vad händer om du ändrar denna? Hur förhåller sig histogrammet till den röda linjen och hur förklaras variationen kring denna linje? Prova med $N = 1000$ och $N = 1000$. Vad händer och varför?

Problem 2 – Stora talens lag

Stora talens lag säger att för oberoende, likafördelade stokastiska variabler $X_1, X_2, ...$, så konvergerar det aritmetiska medelvärdet mot väntevärdet när antalet termer i medelvärdet går mot oändligheten. Vi ska nu undersöka denna konvergens genom att simulera de stokastiska variablerna X_i (som vi här låter vara exponentialfördelade) och studera beteendet hos medelvärdet $S_n = (X_1 + \cdots + X_n)/n$.

```
mu = 1
M = 500
x = stats.expon.rvs(scale=mu, size=M)

plt.plot(np.ones(M) * mu, 'r.-')

for k in range(1, M + 1):
    plt.plot(k, np.mean(x[:k]), 'k.')

plt.legend(['Sant $\mu$', 'Skattning av $\mu$'])
plt.show()
```

Punkten med x-värde `k` är medelvärdet av `k` stycken exponentialfördelade stokastiska variabler. Ser det ut som förväntat? Hur påverkar valet av `mu` konvergensen av medelvärdet? Vad händer om du ersätter `x[:k]` i koden ovan med dess kvadrat `x[:k] ** 2`? Varför?

Problem 3 – Monte Carlo-skattning av väntevärde

Antag att vi nu inte vet att exponentialfördelningen som vi simulerade i Problem 1 har väntevärdet just 1 och att vi vill uppskatta detta från våra

simulerade data. Det kan vi göra genom att beräkna medelvärdet av slumptalen.

```
mu = 1
N = 10000
y = stats.expon.rvs(scale=mu, size=N)
print(np.mean(y))
```

Hur bra blir din uppskattning? Prova att göra om simuleringen och medelvärdesberäkningen några gånger. Prova också olika värden på `N`.

Att använda medelvärdet av simulerade slumptal för att beräkna väntevärden kallas för Monte Carlo-metoder. Idén bakom Monte Carlo-metoder har funnits inom matematiken åtminstone sedan 1700-talet, men kom till praktisk användning först under andra halvan av 1900-talet då det blev möjligt att utföra stora beräkningar med dator. De första Monte Carlo-simuleringarna utfördes under 1940-talet av Stanisław Ulam och John von Neumann [1] i samband med Manhattanprojektet vars syfte var att ta fram den första atombomben. Eftersom arbetet var hemligt behövdes ett kodnamn och metoden namngavs efter casinot i Monte Carlo.

Problem 4 – Monte Carlo-simulering av talet π

Fördelen med numerisk simulering av väntevärden är att de kan användas även för väntevärden som är svåra att beräkna exakt. Vi ska nu använda Monte Carlo-metoder för att bestämma ett approximativt värde på talet π. Låt U och V vara två oberoende stokastiska variabler som är likformigt fördelade på $[-1, 1]$. Paret (U, V) antar värden i $[-1, 1] \times [-1, 1]$ och kan ses som punkter i en kvadrat i planet. Sannolikheten att punkten (U, V) hamnar i enhetscirkeln är

$$P(\sqrt{U^2 + V^2} \leq 1) = \frac{\text{arean av enhetscirkeln}}{\text{arean av kvadraten } [-1, 1] \times [-1, 1]} = \frac{\pi}{4}.$$

Vi kan skatta π på följande sätt. Vi simulerar först ett stort antal punkter $(U_1, V_1), (U_2, V_2), ..., (U_N, V_N)$. För varje punkt (U_i, V_i) kontrollerar vi om $\sqrt{U_i^2 + V_i^2} \leq 1$ och beräknar andelen punkter som hamnat i enhetscirkeln. Eftersom

$$\frac{\text{antal punkter som hamnat i enhetscirkeln}}{N} \to P(\sqrt{U^2 + V^2} \leq 1) = \frac{\pi}{4},$$

då $N \to \infty$, så gäller det för stora värden på N att

$$\pi \approx \frac{4 \cdot \text{antal punkter som hamnar i enhetscirkeln}}{N}$$

enligt stora talens lag. Följande kod genererar N punkter (U_i, V_i), plottar dem i planet samt beräknar motsvarande skattning av värdet på π. Kör koden flera gånger och variera `N`. Hur bra skattning av π kan du få?

```
N = 100
U = 2 * stats.uniform.rvs(size=N) - 1
V = 2 * stats.uniform.rvs(size=N) - 1

plt.plot(U, V, 'o')

X = np.linspace(-1, 1, 200)
plt.plot(X, np.sqrt(1 - X ** 2), 'r')
plt.plot(X, -np.sqrt(1 - X ** 2), 'r')

plt.show()

Z = (np.sqrt(U ** 2 + V ** 2) <= 1)
pi = 4 * np.mean(Z)

print(pi)
```

Notera att syntaxen `(x>5)` i NumPy ger en vektor av samma storlek som `x`, men där ett element i vektorn är 1 eller 0 beroende på om motsvarande element i `x` uppfyller villkoret `> 5` eller ej. Denna syntax används för att definiera `Z` i koden ovan.

Använd metoden ovan för att skatta sannolikheten att en normalfördelad stokastisk variabel med väntevärde noll och varians ett (dvs. med fördelning $\mathrm{N}(0,1)$) befinner sig i intervallet $[-1,1]$. Jämför värdet med $\Phi(1) - \Phi(-1)$ från tabellsamlingen.

Problem 5 – Beräkning av sannolikheter

SciPy har kommandon för de vanligaste sannolikhetsfördelningarna. Läs `help` eller `?` för `stats.binom`, `stats.norm` och `stats.expon`. Var och en har funktionerna `cdf` för att beräkna fördelningsfunktionen samt `pmf` eller `pdf` för sannolikhetsfunktionen (för diskreta stokastiska variabler) eller täthetsfunktionen (för kontinuerliga stokastiska variabler). Notera att `stats.expon.cdf` och `stats.expon.pdf`, precis som `stats.expon.rvs`, har skalan `scale`, dvs. $1/\lambda$, som parameter istället för intensiteten λ som i boken. Låt X_1 vara $\mathrm{Bin}(10, 0.3)$, $X_2 \in \mathrm{N}(5,3)$, $X_3 \in \mathrm{Exp}(7)$ (dvs. att $\lambda = 7$ och `scale = 1/7`) och bestäm (med hjälp av funktionerna ovan) för $k = 1, 2, 3$,

1. $P(X_k \leq 3)$
2. $P(X_k > 7)$
3. $P(3 < X_k \leq 4)$

Problem 6 – Visualisering av sannolikhetsfördelningar

SciPys kommandon kan även användas för att visualisera de vanligaste sannolikhetsfördelningarna. Värdet i x för täthetsfunktionen för normalfördelningen $\mathrm{N}(\mu, \sigma)$ ges exempelvis av kommandot `stats.norm.pdf(x, mu, sigma)`. Följande kod genererar grafen av täthetsfunktionen för den standardiserade normalfördelningen $\mathrm{N}(0, 1)$.

```
x = np.linspace(-10, 10, 2000)
y = stats.norm.pdf(x, 0, 1)

plt.plot(x, y)
plt.show()
```

Prova även att plotta täthetsfunktionen till normalfördelningen för några andra värden på parametrarna μ och σ, exempelvis $\mu = -1$, $\sigma = 0.1$ respektive $\mu = 2$, $\sigma = 2$.

För gammafördelningen med parametrar a och b ges täthetsfunktionen av

$$f_X(x) = \frac{1}{b^a \Gamma(a)} x^{a-1} e^{-x/b},$$

(observera att [2] använder en annan definition av parametrarna i gammafördelningen). Parametern b i täthetsfunktionen motsvarar `scale` i SciPys implementation av gammafördelningen. Följande kod kan användas för att plotta denna täthetsfunktion.

```
x = np.linspace(0, 10, 1000)
y = stats.gamma.pdf(x, a=1, scale=2)
z = stats.gamma.pdf(x, a=5, scale=1)

plt.plot(x, y)
plt.plot(x, z, 'r')
plt.plot()
```

Även för fördelningfunktionerna finns kommandon för de vanligaste sannolikhetsfördelningarna. För gammafördelningen gäller exempelvis

```
x = np.linspace(0, 10, 1000)
y = stats.gamma.cdf(x, a=1, scale=2)
z = stats.gamma.cdf(x, a=5, scale=1)

plt.plot(x, y)
plt.plot(x, z, 'r')
plt.show()
```

Använd Monte Carlo–metoden ovan för att approximera sannolikheten $P(X < 3)$ för en gammafördelning med `a = 1` och `scale=2`. Jämför detta med värdet från fördelningsfunktionen `stats.gamma.cdf(3, a=1, scale=2)`.

Problem 7 – Flerdimensionell normalfördelning

Täthetsfunktionen för den flerdimensionella normalfördelningen ritas upp av funktionen `plotting.plot_mvn_pdf`. Vi undersöker hur funktionen fungerar och testar med några olika parametervärden. Parametrarna `mu_x` och `mu_y` kan anta alla reella värden, parametrarna `sigma_x` och `sigma_y` kan anta alla positiva värden och parametern `rho` kan anta alla värden på intervallet $[-1, 1]$. Observera att plottfönstret i funktionen `plotting.plot_mvn_pdf` är fixt, så för parametervärden som är av storleksordningen tio eller större så kommer merparten av täthetsfunktionen att hamna utanför fönstret.

```
mu_x = 0
mu_y = -2
sigma_x = 2
sigma_y = 4
rho = 0.7

plotting.plot_mvn_pdf(mu_x, mu_y, sigma_x, sigma_y, rho)
plt.show()
```

Hur påverkar olika parametervärden utseendet på plotten? Vad motsvarar de olika parametrarna?

Referenser

[1] Eckhardt, Roger (1987) Stan Ulam, John von Neumann and the Monte Carlo, Method *Los Alamos Sci.*, Vol **15**, p. 131-43.

[2] Blom, G., Enger, J., Englund, G., Grandell, J., och Holst, L., (2005). Sannolikhetsteori och statistikteori med tillämpningar.