5.1 - Statistical Thinking in Python

## **5.1.1 - Pandas Foundation**

**Chapter-1.1: Graphical exploratory data analysis**

|  |  |  |  |
| --- | --- | --- | --- |
| Plotting a histogram | | | |
| plot([x], y, [fmt], data=**None**, \*\*kwargs) | | |
| Generating Histogram |  | In [1]: import matplotlib.pyplot as plt  In [2]: \_ = plt.hist(df\_swing['dem\_share'])  In [3]: \_ = plt.xlabel('percent of vote for Obama')  In [4]: \_ = plt.ylabel('number of counties')  In [5]: plt.show() |
| Setting bins |  | In [1]: \_ = plt.hist(df\_swing['dem\_share'], bins=20) |
| Setting Seaborn |  | In [1]: import seaborn as sns  In [2]: sns.set() |
| Generating a bee swarm plot |  | In [1]: \_ = sns.swarmplot(x='state', y='dem\_share', data=df\_swing) |
| Plot all of your data: ECDFs (Empirical cumulative distribution function) | | |
| Making an ECDF |  | In [1]: import numpy as np  In [2]: x = np.sort(df\_swing['dem\_share'])  In [3]: y = np.arange(1, len(x)+1) / len(x)  In [4]: \_ = plt.plot(x, y, marker='.', linestyle='none')  In [5]: \_ = plt.xlabel('percent of vote for Obama')  In [6]: \_ = plt.ylabel('ECDF')  In [7]: plt.margins(0.02) # Keeps data off plot edges  In [8]: plt.show() |
| ECDF function |  | def ecdf(data):  """Compute ECDF for a one-dimensional array of measurements."""  # Number of data points: n  n=len(data)  # x-data for the ECDF: x  x=np.sort(data)  # y-data for the ECDF: y  y = np.arange(1, len(x)+1) / n  return x, y |

Chapter-1.2: Quantitative exploratory data analysis

|  |  |  |
| --- | --- | --- |
| Summary Statistics |  |  |
| Mean | np.mean() | np.mean(dem\_share\_PA) |
| Median | np.median() |  |
| Percentile | np.percentile() | In [1]: np.percentile(df\_swing['dem\_share'], [25, 50, 75])  Out[1]: array([ 37.3025, 43.185 , 49.925 ])  ## 25%-50% dilimi = IQR  ## Bu bloktan 1.5 IQR sonraki veriler outlier |
| Box plot | sns.boxplot() | In [2]: import seaborn as sns  In [3]: \_ = sns.boxplot(x='east\_west', y='dem\_share',  ...: data=df\_all\_states) |
| Variance | np.var() | np.var(dem\_share\_FL) |
| Standard Deviation | np.std() | np.std(dem\_share\_FL) |
|  | np.sqrt() | np.sqrt(np.var(dem\_share\_FL)) |
| Covariance and the Pearson correlation coefficient | | |
| Scatter Plot |  | In [1]: \_ = plt.plot(total\_votes/1000, dem\_share,  ...: marker='.', linestyle='none') |
| Covariance | np.cov | # Compute the covariance matrix: covariance\_matrix  covariance\_matrix=np.cov(versicolor\_petal\_length, versicolor\_petal\_width)  print(covariance\_matrix)  # Extract covariance of length and width of petals:  petal\_cov=covariance\_matrix[0, 1]  # Print the length/width covariance  print(petal\_cov) |
|  |  |  |
|  |  |  |
| Pearson correlation coefficient | np.corrcoef | np.corrcoef(x, y) |

Chapter-1.3: Thinking probabilistically-- Discrete variables

|  |  |  |
| --- | --- | --- |
| Random number generators and hacker statistics | | |
| The np.random module | np.random.random() | * Suite of functions based on random number generation * draw a number between 0 and 1 |
| Random number seed | np.random.seed() | ● Integer fed into random number generating  algorithm  ● Manually seed random number generator if  you need reproducibility |
|  | Simulating 4 coins flip | In [1]: import numpy as np  In [2]: np.random.seed(42)  In [3]: random\_numbers = np.random.random(size=4)  In [4]: random\_numbers  Out[4]: array([ 0.37454012, 0.95071431, 0.73199394,  0.59865848])  In [5]: heads = random\_numbers < 0.5  In [6]: heads  Out[6]: array([ True, False, False, False], dtype=bool)  In [7]: np.sum(heads)  Out[7]: 1 |
|  | np.empty() |  |
| The Binomial distribution | | |
| Sampling | np.random.binomial() | In [1]: np.random.binomial(4, 0.5)  Out[1]: 2  In [2]: np.random.binomial(4, 0.5, size=10)  Out[2]: array([4, 3, 2, 1, 1, 0, 3, 2, 3, 0]) |
| Binomial CDF |  | In [3]: sns.set()  In [4]: x, y = ecdf(samples)  In [5]: \_ = plt.plot(x, y, marker='.', linestyle='none')  In [6]: plt.margins(0.02)  In [7]: \_ = plt.xlabel('number of successes')  In [8]: \_ = plt.ylabel('CDF') |
| Poisson distribution |  |  |
|  | np.random.poisson() | samples = np.random.poisson(6, size=10000) |
|  |  |  |
|  |  |  |

Chapter-1.4:

|  |  |  |
| --- | --- | --- |
| Introduction to the Normal distribution | | |
| Checking Normality of Michelson data | np.random.normal() | In [1]: import numpy as np  In [2]: mean = np.mean(michelson\_speed\_of\_light)  In [3]: std = np.std(michelson\_speed\_of\_light)  In [4]: samples = np.random.normal(mean, std, size=10000)  In [5]: x, y = ecdf(michelson\_speed\_of\_light)  In [6]: x\_theor, y\_theor = ecdf(samples)  # burada ne yapilan, biri gercek veri, digeri ise ayni “mean” ve “std” degerleri ile uretilmis verilerden iki tuple dizisi olusturuluyor. Y degeri, yuzde olarak orana tekabul ediyor.  ## dikkat “ecdf” fonksiyonu burada uretildi, standart bir fonksiyon olmayabilir. Sonuc olarak burada olay x ve y degerlerinin uretilmesine bakiyor. Bunlarin dogru uretilmesi gerekiyor |
|  |  | In [1]: import matplotlib.pyplot as plt  In [2]: import seaborn as sns  In [3]: sns.set()  In [4]: \_ = plt.plot(x\_theor, y\_theor)  In [5]: \_ = plt.plot(x, y, marker='.', linestyle='none')  In [6]: \_ = plt.xlabel('speed of light (km/s)')  In [7]: \_ = plt.ylabel('CDF')  In [8]: plt.show() |
| The Normal distribution: Properties and warnings | | |
|  |  | samples=np.random.normal(mu, sigma, size=1000000)  prob=np.sum(samples <= 144)/len(samples)  #bu komut satiri ileride de gorulecegi gibi onemli  print('Probability of besting Secretariat:', prob) |
| The Exponential distribution | | |
|  | np.random.exponential() | [1] mean = np.mean(inter\_times)  [2] samples = np.random.exponential(mean, size=10000) |

Chapter-2.1: Parameter estimation by optimization

|  |  |  |
| --- | --- | --- |
| Optimal parameters |  |  |
|  |  | np.random.seed(42)  tau = np.mean(nohitter\_times)  inter\_nohitter\_time = np.random.exponential(tau, 100000)  # poisson surecinin isledigi bir exponential dagilim  # yukarda ayni “mean” ile 100000 orneklem verileri olusturduk.  # simdi bunlardan “ecdf” fonksiyonu ile x,y degerlerini cekip, grafik cikarticaz ve uyumluluklarini gorucez.  x, y = ecdf(nohitter\_times)  x\_theor, y\_theor = ecdf(inter\_nohitter\_time)  plt.plot(x\_theor, y\_theor)  plt.plot(x, y, marker='.', linestyle='none')  plt.margins(0.02)  plt.xlabel('Games between no-hitters')  plt.ylabel('CDF')  plt.show() |
| Linear regression by least squares | | |
|  |  | Slope:  Intercept: cizginin y eksenini kestigi yer  Residual: very noktasi ile cizgi arasindaki dikey mesafe.  #linear regression cizgisinin olustugu yer/rota, bu residual’ler’n karelerinin toplaminin en dusuk olmacak sekilde hesaplanir. (least squares)  Bu hesaplamayi yapan bircok fonksiyondan biri de, |
| Least squares  # slope ve intercept bulmak icin | **np.polyfit()** | [1]: slope, intercept = np.polyfit(total\_votes, dem\_share, 1)  [2]: slope  Out[2]: 4.0370717009465555e-05  In [3]: intercept  Out[3]: 40.113911968641744 |
|  | **Dagilim komutlari ile elde ettigimiz veriler gibi buradan aldigimiz slope ve intercept degerleri ile “y” degerini olusturup grafige aktaracagiz** | # Plot the illiteracy rate versus fertility  \_ = plt.plot(illiteracy, fertility, marker='.', linestyle='none')  plt.margins(0.02)  \_ = plt.xlabel('percent illiterate')  \_ = plt.ylabel('fertility')  # Perform a linear regression using np.polyfit(): a, b  a, b = np.polyfit(illiteracy, fertility, 1)  # Print the results to the screen  print('slope =', a, 'children per woman / percent illiterate')  print('intercept =', b, 'children per woman')  # Make theoretical line to plot  x = np.array([0, 100])  y = a \* x + b  # Add regression line to your plot  \_ = plt.plot(x, y)  # Draw the plot  plt.show() |
|  | **np.**linspace() | np.linspace(0, 0.5, 100)  # (start, stop, # of points)  # 0’dan 0.5’e kadar 100 nokta olusturur |
|  | **np.empty\_like**() | np.empty\_like(a\_vals)  # bu metot, sekli ve data tipi, verilen ornege uygun olan yeni bir “array” olusturur. Adin da anlasilacagi gibi bostur ama her bir veri noktasinda rastgele bir deger gorunur, daha sonradan istedigimiz verileri girmek uzere. Burada kafa karistiran, sonradan bizim dolduracagimiz ve bos oldugu soylenen bir “array”de rastgele sayilarin gorunmesidir. Sanirim numpy de bombos bir dizi oluturmak diye birsey yok, yer kaplamasi icin rastgele degerlerle olusturuluyor. Asil espri, makinanin bunu bos kabul etmesi ve istedigim degerleri sonradan girebilmem ki bu fonksiyonu da yerine getiriyor. |
| The importance of EDA: Anscombe's quartet | | |
|  |  |  |

Chapter-2.2: Bootstrap confidence intervals

|  |  |  |
| --- | --- | --- |
| Generating bootstrap replicates | | |
| Bootstrapping |  | ## Bootstrap sample: gercek verilerden ayni sayida rasgele secilen verilerle elde edilen veri seti  # Bootstrap replicate: bootstrap sample’dan elde edilen bir istatistik |
|  | np.random.choice() | In [2]: np.random.choice([1,2,3,4,5], size=5)  Out[2]: array([5, 3, 5, 5, 2]) |
|  | **Bootstrap replicate** | In [1]: bs\_sample = np.random.choice(michelson\_speed\_of\_light, size=100)  In [2]: np.mean(bs\_sample)  Out[2]: 299847.79999999999 |
| Bootstrap confidence intervals | | |
| Bootstrap replicate function |  | In [1]: def bootstrap\_replicate\_1d(data, func):  ...: """Generate bootstrap replicate of 1D data."""  ...: bs\_sample = np.random.choice(data, len(data))  ...: return func(bs\_sample) |
| Many bootstrap replicates |  | In [1]: bs\_replicates = np.empty(10000)  In [2]: for i in range(10000):  ...: bs\_replicates[i] = bootstrap\_replicate\_1d(  ...: michelson\_speed\_of\_light, np.mean) |
| Plotting a histogram of bootstrap replicates |  | In [1]: \_ = plt.hist(bs\_replicates, bins=30, normed=True)  In [2]: \_ = plt.xlabel('mean speed of light (km/s)')  In [3]: \_ = plt.ylabel('PDF')  In [4]: plt.show() |
| SEM  (standard error of the mean) | **SEM** | standard error of the mean or SEM,  sem = np.std(data) / np.sqrt(len(data)) |
| Bootstrap confidence interval | **np.percentile()** | In [1]: conf\_int = np.percentile(bs\_replicates, [2.5, 97.5])  Out[1]: array([ 299837., 299868.]) |
| Pairs bootstrap | | |
| Generating a pairs bootstrap sample |  | In [1]: np.arange(7)  Out[1]: array([0, 1, 2, 3, 4, 5, 6])  In [1]: inds = np.arange(len(total\_votes))  In [2]: bs\_inds = np.random.choice(inds, len(inds))  In [3]: bs\_total\_votes = total\_votes[bs\_inds]  In [4]: bs\_dem\_share = dem\_share[bs\_inds] |
| Computing a pairs bootstrap replicate |  | In [1]: bs\_slope, bs\_intercept = np.polyfit(bs\_total\_votes,  ...: bs\_dem\_share, 1)  In [2]: bs\_slope, bs\_intercept  Out[2]: (3.9053605692223672e-05, 40.387910131803025)  In [3]: np.polyfit(total\_votes, dem\_share, 1) # fit of original  Out[3]: array([ 4.03707170e-05, 4.01139120e+01]) |
| Plotting bootstrap regressions |  | # Generate array of x-values for bootstrap lines: x  x = np.array([0,100])  # Plot the bootstrap lines  for i in range(100):  \_ = plt.plot(x, bs\_slope\_reps[i]\*x + bs\_intercept\_reps[i],  linewidth=0.5, alpha=0.2, color='red')  # Plot the data  \_ = plt.plot(illiteracy, fertility, marker='.', linestyle='none') |

Chapter-2.3: Introduction to hypothesis testing

|  |  |  |
| --- | --- | --- |
| Generating a permutation sample | | |
| Generating permutation replicates |  | In [1]: import numpy as np  In [2]: dem\_share\_both = np.concatenate(  ...: (dem\_share\_PA, dem\_share\_OH))  In [3]: dem\_share\_perm = np.random.permutation(dem\_share\_both)  In [4]: perm\_sample\_PA = dem\_share\_perm[:len(dem\_share\_PA)]  In [5]: perm\_sample\_OH = dem\_share\_perm[len(dem\_share\_PA):]  # 1- burada iki veri grubunu birlestirdik,  # 2- kardik (permute ettik),  # 3- data\_1’I ayni satir sayisi kadar bastan cekiyoruz, geri kalani da data\_2 olarak atiyoruz |
|  |  |  |
| EDA before hypothesis testing | sns.swarmplot() | # Make bee swarm plot  \_ = sns.swarmplot(x='ID', y='impact\_force', data=df)  # Label axes  \_ = plt.xlabel('frog')  \_ = plt.ylabel('impact force (N)')  # Show the plot  plt.show() |

Chapter-2.4: Hypothesis test examples

|  |  |  |
| --- | --- | --- |
| A/B testing |  |  |
|  | A ve B gruplarinin gozlemlenen degerlerinin ortalamalarinin farkini alan bir fonksiyon yazilmis ve bu fonksiyon kullanilarak yapilan hesaplama bir degiskene atanmis | In [1]: import numpy as np  In [2]: # clickthrough\_A, clickthrough\_B: arr. of 1s and 0s  In [3]: def diff\_frac(data\_A, data\_B):  ...: frac\_A = np.sum(data\_A) / len(data\_A)  ...: frac\_B = np.sum(data\_B) / len(data\_B)  ...: return frac\_B - frac\_A  ...:  In [4]: diff\_frac\_obs = diff\_frac(clickthrough\_A,  ...: clickthrough\_B) |
| ornek | Burada, okuryazarlik ile dogum orani arasinda bir ilski olup olmadigi, aralarindaki corelasyon katsayisi uzerinden gidirlerek hesaplama yapilmis. Bunun icinde okuryazarlik icin permutasyon yapilarak 10000 tane korelasyon katsayisi uretilmis ve p degeri burden bulunmus. | # Compute observed correlation: r\_obs  r\_obs = pearson\_r(illiteracy, fertility)  # Initialize permutation replicates: perm\_replicates  perm\_replicates = np.empty(10000)  # Draw replicates  for i in range(10000):  # Permute illiteracy measurments: illiteracy\_permuted  illiteracy\_permuted = np.random.permutation(illiteracy)  # Compute Pearson correlation  perm\_replicates[i] = pearson\_r(illiteracy\_permuted, fertility)  # Compute p-value: p  p = np.sum(perm\_replicates>=r\_obs) / len(perm\_replicates)  print('p-val =', p) |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

def fxy(x, y, z):

return x if y == 'SideA' else z

intra['newcolumn'] = intra.apply(lambda x, y, z: fxy(x['SideADeaths'], y['side'], z['SideBDeaths']), axis=1)

|  |
| --- |
| """ |
|  | Utilities for DataCamp's statistical thinking courses \*exactly\* as |
|  | originally written in Statistical Thinking I and II and Case Studies In |
|  | Statistical Thinking. |
|  | """ |
|  |  |
|  | import numpy as np |
|  |  |
|  |  |
|  | def ecdf(data): |
|  | """Compute ECDF for a one-dimensional array of measurements.""" |
|  | # Number of data points |
|  | n = len(data) |
|  |  |
|  | # x-data for the ECDF |
|  | x = np.sort(data) |
|  |  |
|  | # y-data for the ECDF |
|  | y = np.arange(1, n+1) / n |
|  |  |
|  | return x, y |
|  |  |
|  |  |
|  | def pearson\_r(x, y): |
|  | """Compute Pearson correlation coefficient between two arrays.""" |
|  | # Compute correlation matrix |
|  | corr\_mat = np.corrcoef(x, y) |
|  |  |
|  | # Return entry [0,1] |
|  | return corr\_mat[0,1] |
|  |  |
|  |  |
|  | def perform\_bernoulli\_trials(n, p): |
|  | """Perform n Bernoulli trials with success probability p |
|  | and return number of successes.""" |
|  | # Initialize number of successes: n\_success |
|  | n\_success = 0 |
|  |  |
|  | # Perform trials |
|  | for i in range(n): |
|  | # Choose random number between zero and one: random\_number |
|  | random\_number = np.random.random() |
|  |  |
|  | # If less than p, it's a success so add one to n\_success |
|  | if random\_number < p: |
|  | n\_success += 1 |
|  |  |
|  | return n\_success |
|  |  |
|  |  |
|  | def successive\_poisson(tau1, tau2, size=1): |
|  | """Compute time for arrival of 2 successive Poisson processes.""" |
|  | # Draw samples out of first exponential distribution: t1 |
|  | t1 = np.random.exponential(tau1, size=size) |
|  |  |
|  | # Draw samples out of second exponential distribution: t2 |
|  | t2 = np.random.exponential(tau2, size=size) |
|  |  |
|  | return t1 + t2 |
|  |  |
|  |  |
|  | def bootstrap\_replicate\_1d(data, func): |
|  | """Generate bootstrap replicate of 1D data.""" |
|  | bs\_sample = np.random.choice(data, len(data)) |
|  | return func(bs\_sample) |
|  |  |
|  |  |
|  | def draw\_bs\_reps(data, func, size=1): |
|  | """Draw bootstrap replicates.""" |
|  | # Initialize array of replicates |
|  | bs\_replicates = np.empty(size) |
|  |  |
|  | # Generate replicates |
|  | for i in range(size): |
|  | bs\_replicates[i] = bootstrap\_replicate\_1d(data, func) |
|  |  |
|  | return bs\_replicates |
|  |  |
|  |  |
|  | def draw\_bs\_pairs\_linreg(x, y, size=1): |
|  | """Perform pairs bootstrap for linear regression.""" |
|  | # Set up array of indices to sample from: inds |
|  | inds = np.arange(len(x)) |
|  |  |
|  | # Initialize replicates: bs\_slope\_reps, bs\_intercept\_reps |
|  | bs\_slope\_reps = np.empty(size) |
|  | bs\_intercept\_reps = np.empty(size) |
|  |  |
|  | # Generate replicates |
|  | for i in range(size): |
|  | bs\_inds = np.random.choice(inds, size=len(inds)) |
|  | bs\_x, bs\_y = x[bs\_inds], y[bs\_inds] |
|  | bs\_slope\_reps[i], bs\_intercept\_reps[i] = np.polyfit(bs\_x, bs\_y, 1) |
|  |  |
|  | return bs\_slope\_reps, bs\_intercept\_reps |
|  |  |
|  |  |
|  | def permutation\_sample(data1, data2): |
|  | """Generate a permutation sample from two data sets.""" |
|  | # Concatenate the data sets |
|  | data = np.concatenate((data1, data2)) |
|  |  |
|  | # Permute the concatenated array |
|  | permuted\_data = np.random.permutation(data) |
|  |  |
|  | # Split the permuted array into two |
|  | perm\_sample\_1 = permuted\_data[:len(data1)] |
|  | perm\_sample\_2 = permuted\_data[len(data1):] |
|  |  |
|  | return perm\_sample\_1, perm\_sample\_2 |
|  |  |
|  |  |
|  | def draw\_perm\_reps(data\_1, data\_2, func, size=1): |
|  | """Generate multiple permutation replicates.""" |
|  | # Initialize array of replicates: perm\_replicates |
|  | perm\_replicates = np.empty(size) |
|  |  |
|  | for i in range(size): |
|  | # Generate permutation sample |
|  | perm\_sample\_1, perm\_sample\_2 = permutation\_sample(data\_1, data\_2) |
|  |  |
|  | # Compute the test statistic |
|  | perm\_replicates[i] = func(perm\_sample\_1, perm\_sample\_2) |
|  |  |
|  | return perm\_replicates |
|  |  |
|  |  |
|  | def diff\_of\_means(data\_1, data\_2): |
|  | """Difference in means of two arrays.""" |
|  | return np.mean(data\_1) - np.mean(data\_2) |
|  |  |
|  |  |
|  | def draw\_bs\_pairs(x, y, func, size=1): |
|  | """Perform pairs bootstrap for single statistic.""" |
|  | # Set up array of indices to sample from: inds |
|  | inds = np.arange(len(x)) |
|  |  |
|  | # Initialize replicates: bs\_replicates |
|  | bs\_replicates = np.empty(size) |
|  |  |
|  | # Generate replicates |
|  | for i in range(size): |
|  | bs\_inds = np.random.choice(inds, len(inds)) |
|  | bs\_x, bs\_y = x[bs\_inds], y[bs\_inds] |
|  | bs\_replicates[i] = func(bs\_x, bs\_y) |
|  |  |
|  | return bs\_replicates |
|  |  |
|  |  |
|  | def swap\_random(a, b): |
|  | """Randomly swap entries in two arrays.""" |
|  | # Indices to swap |
|  | swap\_inds = np.random.random(size=len(a)) < 0.5 |
|  |  |
|  | # Make copies of arrays a and b for output |
|  | a\_out = np.copy(a) |
|  | b\_out = np.copy(b) |
|  |  |
|  | # Swap values |
|  | a\_out[swap\_inds] = b[swap\_inds] |
|  | b\_out[swap\_inds] = a[swap\_inds] |
|  |  |
|  | return a\_out, b\_out |
|  |  |
|  |  |
|  | def b\_value(mags, mt, perc=[2.5, 97.5], n\_reps=None): |
|  | """Compute the b-value and optionally its confidence interval.""" |
|  | # Extract magnitudes above completeness threshold |
|  | m = mags[mags >= mt] |
|  |  |
|  | # Compute b-value |
|  | b = (np.mean(m) - mt) \* np.log(10) |
|  |  |
|  | # Draw bootstrap replicates |
|  | if n\_reps is None: |
|  | return b |
|  | else: |
|  | m\_bs\_reps = dcst.draw\_bs\_reps(m, np.mean, size=n\_reps) |
|  |  |
|  | # Compute b-value from replicates |
|  | b\_bs\_reps = (m\_bs\_reps - mt) \* np.log(10) |
|  |  |
|  | # Compute confidence interval |
|  | conf\_int = np.percentile(b\_bs\_reps, perc) |
|  |  |
|  | return b, conf\_int |
|  |  |
|  |  |
|  | def frac\_yay\_dems(dems, reps): |
|  | """Compute fraction of Democrat yay votes.""" |
|  | frac = np.sum(dems) / len(dems) |
|  | return frac |
|  |  |
|  |  |
|  | def heritability(parents, offspring): |
|  | """Compute the heritability from parent and offspring samples.""" |
|  | covariance\_matrix = np.cov(parents, offspring) |
|  | return covariance\_matrix[0,1] / covariance\_matrix[0,0] |