Estatística e Probabilidade - Projeto 02

Rodrigo Pita - 118187443

O código completo utilizado para responder todas as questões desse projeto pode ser encontrado no meu GitHub(https://github.com/RodrigoPita/Probest/blob/main/projeto_2.py)

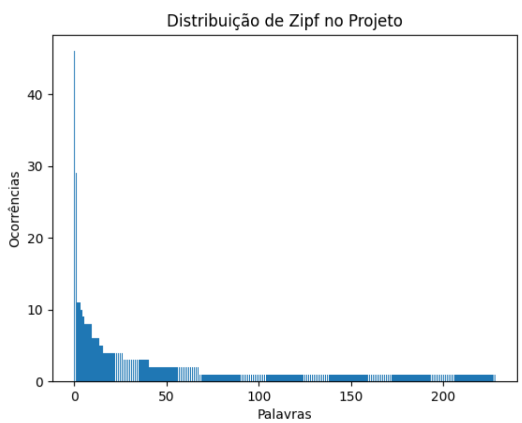
1) George Kingsley Zipf, um linguista de Harvard popularizou a distribuição de Zipf, que é a forma discreta da distribuição de Pareto (contínua). A partir da distribuição de Pareto, temos também o princípio de Pareto, que diz que 20% das causas são responsáveis por 80% das consequências (e vice-versa), mas vamos focar na Lei de Zipf por hora.
A Lei de Zipf é expressa em termos de frequência de ocorrência (contagem, quantidade) de eventos. A lei pode ser denotada como:

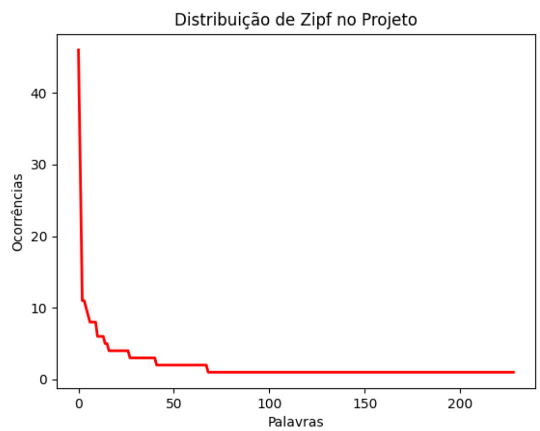
$$\mathbf{F \sim r^{-a}}$$

Onde F é a frequência de ocorrência de um evento, r é seu ranque estatístico, ou seja, sua posição numa lista ordenada dos eventos, e a é próximo de 1. Para melhor ilustrar o funcionamento da Lei, podemos visualizar o exemplo de um livro qualquer, onde cada palavra no livro seria um evento. Digamos que a palavra com o maior número de ocorrências tenha aparecido x vezes, a partir disso, podemos dizer que o número de ocorrências da segunda palavra será próximo de $x/2$, e para a terceira, $x/3$ e assim por diante, obtendo a distribuição de Zipf completa do livro.
Um estudo interessante dessa distribuição foi feito por Richard Voss e John Clarke (1975, 1978), voltado para músicas de estações de rádio de rock, blues, jazz e clássica. Eles mediram vários parâmetros, incluindo voltagem de amplificadores, flutuações de volume da música e flutuações de tom da música. Com esses estudos, eles descobriram que tanto as flutuações de tom quanto as de volume seguiam a distribuição de Zipf. Além disso, eles também fizeram um programa para gerar música usando 3 geradores de número aleatório diferentes: *uma fonte de white-noise* $(1/f^{0})$, uma fonte de *pink-noise* $(1/f)$, e uma fonte de *brown-noise* $(1/f^{2})$. Para controlar a duração das notas e o tom eles usaram geradores de números aleatórios independentes. O resultado deste experimento foi que para a maioria dos ouvintes, a música produzida a partir dos geradores de *pink-noise*, que segue a distribuição *1/f*, foi a mais agradável, enquanto a dos geradores de *white-noise* era aleatória demais e de *brown-noise* correlacionada demais.
No âmbito de compressão de arquivos, a lei de Zipf, pode ser usada para modelar a forma de compressão. De acordo com a lei, palavras com um menor número de caracteres tendem a

aparecer mais do que palavras com mais caracteres. O mesmo conceito pode ser aplicado para a frequência de ocorrência de cada letra do alfabeto, por exemplo a letra “u”, na língua portuguesa tem uma probabilidade de ocorrência menor que 25%, ou seja, podemos optar por gerar um código para ela com pelo menos 2 bits, dada a baixa probabilidade. Porém depois de uma letra “q” a probabilidade para a letra “u” cresce consideravelmente, nos incentivando a gerar um código para a letra “u” que vier depois da letra “q” mais curto.

Para acabar, um último exemplo da distribuição de Zipf funcionando pode ser observado neste próprio texto, como podemos ver abaixo:





Abaixo temos algumas funções auxiliares para manipular os dados da dissertação:

```
def get_data_from_file( file_name:str ) -> list:
    '''Abre um arquivo e coloca todo o seu conteudo em uma string'''
    file_content = ''
    for line in io.open( file_name, mode = 'r', encoding = 'utf-8' ):
        file_content += line
    return file_content

def format_string( string:str ) -> str:
    '''Formata uma string, tirando todos os simbolos sem ser carateres do alfabeto padrao'''
    formated_string = string.strip().replace( ',', '' ).replace( '.', '' ).replace( ';', '' ).lower()
    return formated_string.replace( '(', '' ).replace( ')', '' ).replace( ':', '' ).replace( '\n', '' )

def count_frequency( text:str ) -> dict:
    '''Conta a frequencia de ocorrencias de cada palavra num texto,
    retornando um dicionario com as palavras e suas respectivas ocorrencias'''
    all_words = text.split()
    frequency_by_word = {}
    for word in all_words:
        if ( frequency_by_word.get( word ) is not None ):
            frequency_by_word[word] += 1
        else: frequency_by_word[word] = 1
    return frequency_by_word
```

Aqui, a função para plotar os gráficos e o início da main:

```
def plot_frequencies( freqs:dict ) -> None:
    '''Plota um grafico de acordo com as frequencias do dicionario dado'''
    aux = list( freqs.items() )
    elements = sorted( aux, key = lambda x: x[1], reverse = True )
    words = [ element[0] for element in elements ]
    occurences = [ element[1] for element in elements ]
    x_pos = np.arange( len( words ) )

    plt.title( 'Distribuição de Zipf no Projeto' )
    plt.plot( x_pos, occurences, linewidth = 2, color = 'r' )
    plt.bar( x_pos, occurences, align = 'center' )
    plt.xlabel( 'Palavras' )
    plt.ylabel( 'Ocorrências' )
    plt.show()

def main():
    text = get_data_from_file( FILE_NAME )
    formated_text = format_string( text )
    frequencies = count_frequency( formated_text )
    plot_frequencies( frequencies )
```

Obs: Importante notar que pelos gráficos acima podemos ver que o texto não segue exatamente a distribuição de Zipf, porém está próximo da mesma. Um fator que pode ter influenciado fortemente no resultado foi o pequeno espaço amostral composto de um texto com um pouco mais do que uma página. Tivera essa dissertação sido mais longa, talvez a curva do gráfico ficasse ainda mais próxima do que esperamos ver numa distribuição de Zipf.

2-a) A função normal de média zero e variância unitária se aproxima extremamente rápido de 0, porém não chega em 0 em nenhum momento. O fato dela não chegar exatamente em 0 em momento algum, garante que P(x > 20) não seja nula. Ao tentar obter "ingenuamente" pelo python o valor será 0, pois a precisão máxima do python é de 16 casas decimais.

2-b) Abaixo podemos ver a integral que calcula a quantidade P(Z > 20):

$$\int_{20}^{+\infty} \frac{1}{\sqrt{2\pi}} {}^{-z^2/2} \, dz \left(z = 1/y, dz = -1/y^2\right)$$

$$= \int_{1/20}^{0} \frac{1}{\sqrt{2\pi}} {}^{-1/2y^2} \frac{-1}{y^2} \, dy$$

$$= \int_{0}^{1/20} \frac{1}{y^2\sqrt{2\pi}} {}^{-1/2y^2} \, dy$$

$$= \frac{20}{20} \int_{0}^{1/20} \frac{1}{y^2\sqrt{2\pi}} {}^{-1/2y^2} \, dy$$

$$= \int_{0}^{1/20} 20 \left(\frac{1}{20y^2\sqrt{2\pi}} {}^{-1/2y^2} \right) dy$$

$$= \int_{0}^{1/20} 20 g(y) \, dy = \int_{0}^{1/20} g(y) f_y(y) \, dy = E[g(y)]$$

Obs: Na quarta linha fazemos uso de um algebrismo para chegar na fórmula esperada, multiplicando a integral por 20/20 e no final, usamos a Lei do Estatístico preguiçoso para chegar à fórmula da Esperança de g(y).

2-c) Abaixo podemos ver os resultados tanto da esperança da variável aleatória Y, quanto da esperança da função g(y)

```
-> E[Y] = 0.025107146751684253
-> E[g(y)] = 2.790089177235278e-89
```

2-d) À medida que as simulações de Y aumentam a incerteza diminui. Abaixo podemos ver o desvio padrão após ter calculado 1000 simulações de E[g(y)]

```
-> Desvio Padrão = 1.2619058466574866e-90
```

Obs: O motivo de ter calculado apenas 1000 simulações da Esperança de g(y) foi de quando tentei calcular mais que isso, o programa quebrou depois de muito tempo rodando.

Seguem as funções auxiliares usadas para os cálculos deste item onde o const usado para o método list_of_Ys foi 1000000 para o item c e 100000 para o item d:

```python
def list_of_Ys( const:int ) -> list:
    '''Cria uma lista de variaveis aleatorias Y'''
    low = 0
    high = 1/20
    Y = np.random.uniform( low, high, size = const )
    return Y

def mean_y( Y:list ) -> float:
    '''Valor esperado de Y'''
    n = len( Y )
    mean = sum( Y ) / n
    return mean

def g( y:float ) -> float:
    '''Funcao g(y)'''
    ans = 1 / ( 20 * ( y ** 2 ) * math.sqrt( 2 * math.pi ) ) * ( math.e ** ( -1 / ( 2 * y ** ( 2 ) ) ) )
    return ans

def mean_g( Y:list ) -> float:
    '''Valor esperado de g'''
    G = []
    for yi in Y:
        gi = g( yi )
        G.append( gi )
    n = len( G )
    mean = sum( G ) / n
    return mean
```

Referências

[1] The Zipf Mystery

[2] Discrete Pareto Distribution vs Zipf Distribution and Power Law vs Zipf Law - Cross Validated

[3] Zipf's law - Wikipedia

[4] (PDF) Zipf's Law, Music Classification, and Aesthetics | Penousal Machado - Academia.edu

[5] Pareto principle - Wikipedia

[6] Efficient Compression Scheme for Large Natural Text Using Zipf Distribution

[7] https://www.ime.usp.br/~rbrito/docs/1996-relatorio-ic.pdf

[8] The Principle of Least Effort: Definition and Examples of Zipf's Law