EP4130: Data Science and Analysis

A brief study on Zipf's Law

2nd May 2023

By-

Anshul Gupta
EE20BTECH11042

Pushkal Mishra
EE20BTECH11042

An introduction to Zipf’s Law

It is an empirical law that describes the rank-frequency distribution of a given dataset. The Zipfian distribution is a member of a discrete power law probability distributions that share similar characteristics. While it is related to the Zeta distribution, there are distinct differences between the two. It was initially observed in quantitative linguistics, where it stated that the word frequency was inversely proportional to its rank in the frequency distribution table.

This law has also been observed in numerous human-created systems, such as rankings of mathematical expressions, musical notes, corporation sizes, income rankings, and viewership of TV channels, among others. This distribution has also been found in uncontrolled environments such as transcriptomes of cells.

Some example plots for illustration

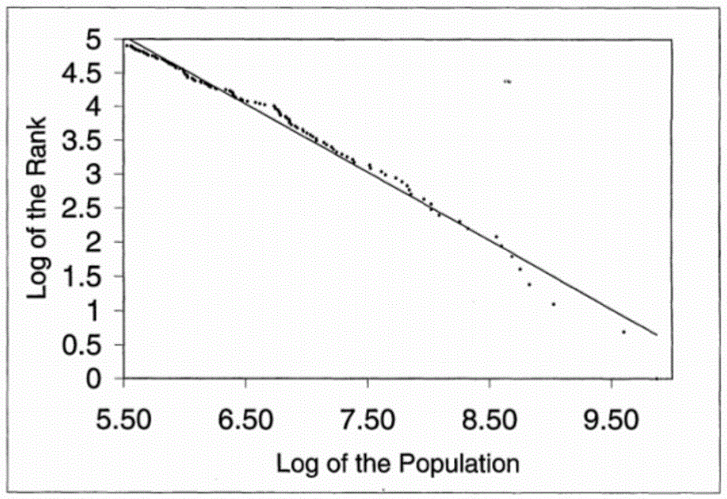
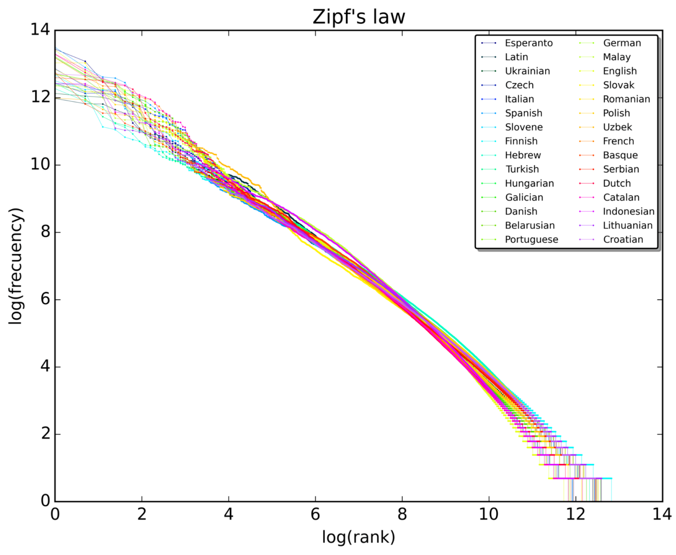
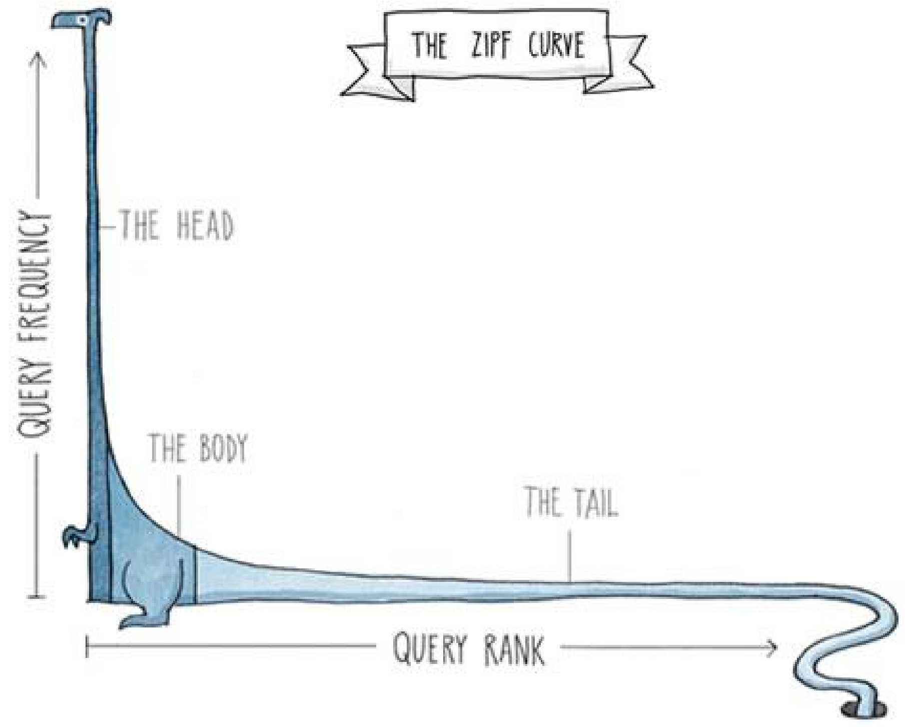


FIGURE I
Log Size versus Log Rank of the 135 largest U. S. Metropolitan Areas in 1991
Source: Statistical Abstract of the United States [1993].

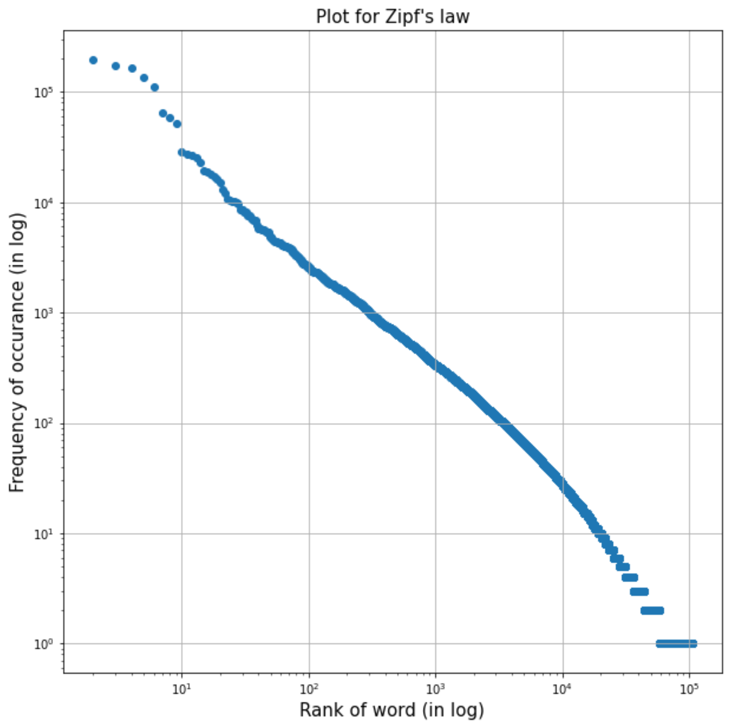




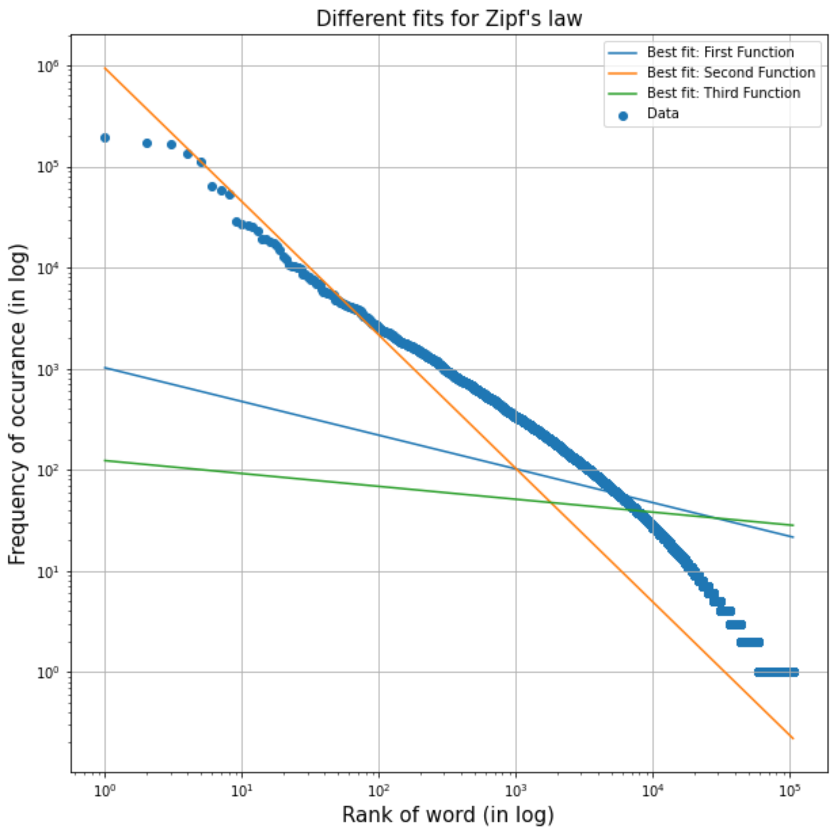
Dataset we experimented with

1) We extracted text from a dictionary here and applied various analysis techniques as mentioned below.

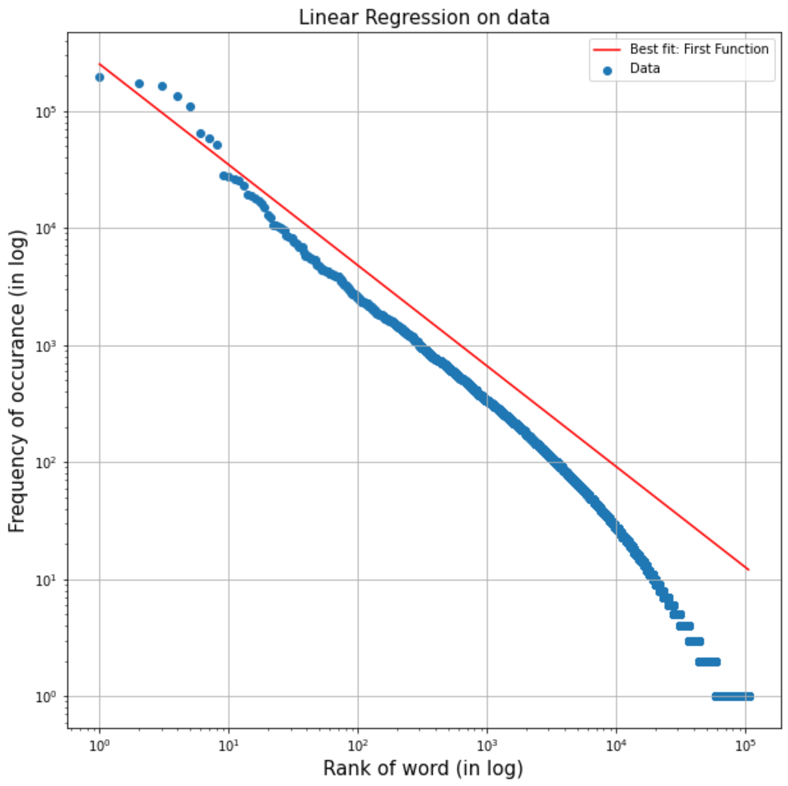
   Scatter plot for frequency vs rank in log-log scale.



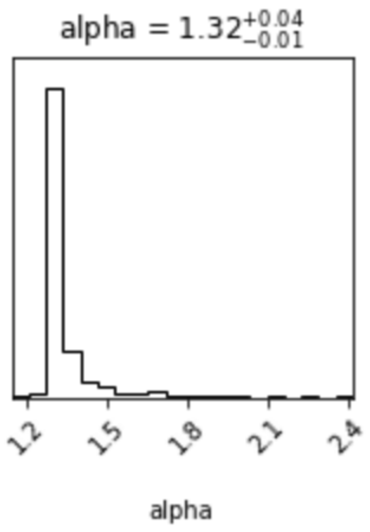
We referred to this paper for fitting different power law forms and from the plot, the 2nd function accurately mimics Zipf's law.



We also performed linear regression on the Log-Log plot to fit a line.



We also tried applying MCMC to find out the best fit parameters. The corner plot result is as follows-



Conclusions

We found that Zipf's law can be observed in various phenomenon and it models the data to a good measure, as seen by the KS test values and p-value for linear fit.

Code is attached below

[1]:
```python
import json
import numpy as np
import matplotlib.pyplot as plt
```

[2]:
```python
file = open('dictionary.json')
data = json.load(file)
meanings = []
l = 0
not_allowed = ".,-()[]0123456789#!;\+*`&:$%/<=>@^"
for word in data:
    meaning = data[word].translate({ord(c): None for c in not_allowed}).lower()
    meaning = meaning.translate({ord(c): None for c in "\"\'"})
    meanings += meaning.split()
    l += 1
print("Number of words:", l)
```

```
Number of words: 102217
```

[3]:
```python
unique, index, count = np.unique(meanings, return_index = True, return_counts =␣
 ↪True)
```

[4]:
```python
order = np.flip(np.argsort(count))
unique = unique[order]
index = index[order]
count = count[order]
rank = 1 + np.arange(len(index))
```

[5]:
```python
# new_order = np.where(count >= 50)
# unique = unique[new_order]
# index = index[new_order]
# count = count[new_order]
```

[6]:
```python
print("Total number of words:", np.max(index))
```

```
Total number of words: 3421131
```

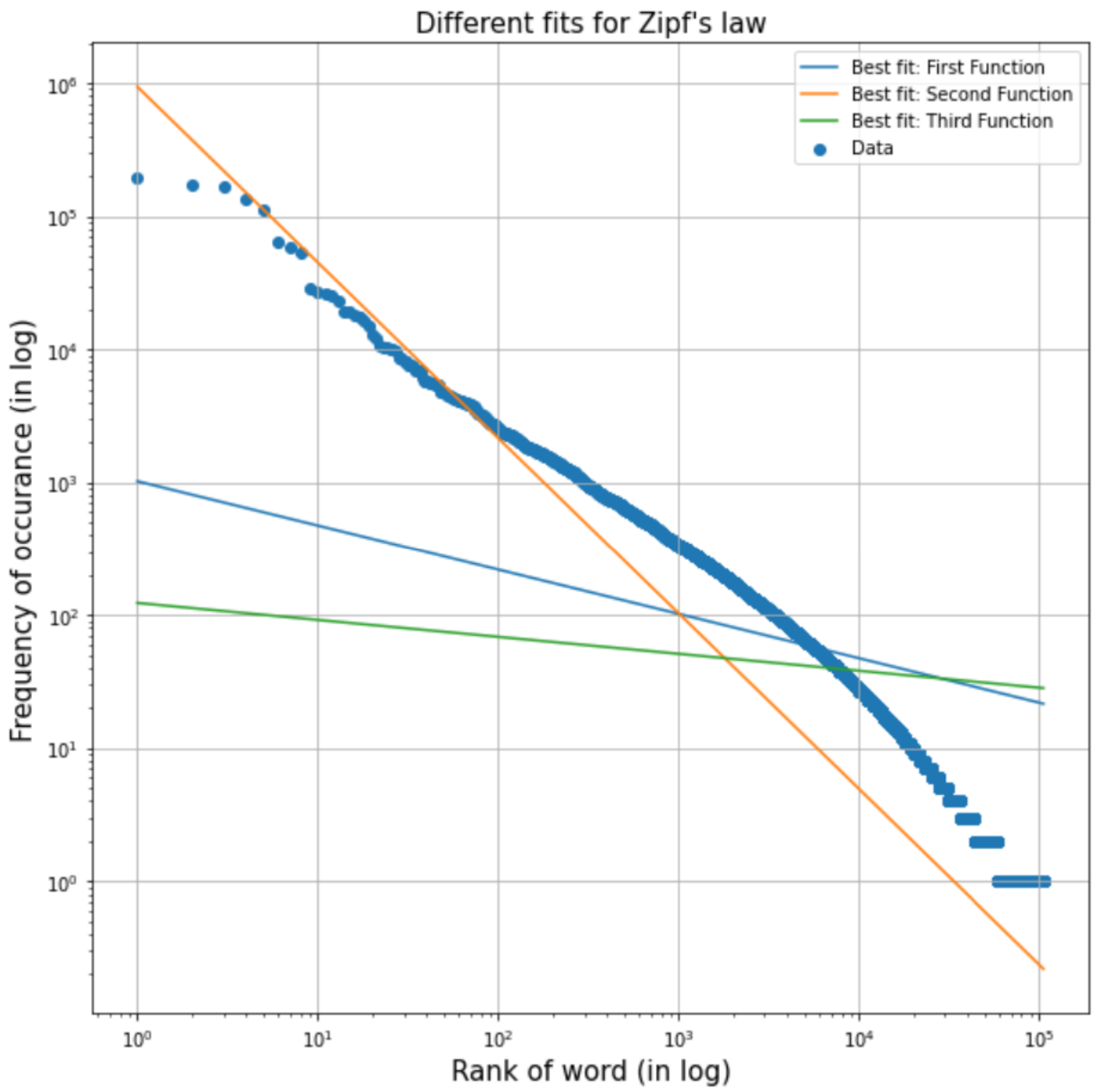
[8]:
```python
fig = plt.figure(figsize = (10, 10))
plt.scatter(rank + 1, count)
plt.title("Plot for Zipf's law", size = 15)
```

```
[10]: def mle_estimate(word_counts, function):
          params = []
          if function == "first_function":
              mle = minimize_scalar(ll_first_function, bracket=(1 + 1e-10, 4),␣
       ↪args=word_counts, method="brent")
              beta = mle.x
              alpha = 1 / (beta - 1)
              C = ((np.arange(len(word_counts)) + 1) ** (-alpha)).sum()
              params = {"alpha" : alpha, "C" : C}

          elif function == "second_function":
              mle = minimize_scalar(ll_second_function, bracket=(1 + 1e-10, 4),␣
       ↪args=word_counts, method="brent")
              beta = mle.x
              alpha = 1 / (beta - 1)
              C = ((np.arange(len(word_counts)) + 1) ** (-alpha)).sum()
              params = {"alpha" : alpha, "C" : C}

          elif function == "third_function":
              mle = minimize_scalar(ll_third_function, bracket=(1 + 1e-10, 4),␣
       ↪args=word_counts, method="brent")
              beta = mle.x
              alpha = 1 / (beta - 1)
              C = ((np.arange(len(word_counts)) + 1) ** (-alpha)).sum()
              params = {"alpha" : alpha, "C" : C}

          return params

      def  ll_first_function(beta, count):
          return -np.sum(np.log(1 / (zeta(beta, 1) * (count ** beta))))

      def ll_second_function(beta, counts):
          return -np.sum(np.log((1 / counts) ** (beta - 1) - (1 / (counts + 1)) **␣
       ↪(beta - 1)))

      def ll_third_function(beta, count):
          likelihood = -np.sum(
              np.log(gamma(1) * (beta - 1) * gamma(count + 1 - beta) / (gamma(2 -␣
       ↪beta) * gamma(count + 1)))
          )
          return likelihood

      def ll_log_normal(theta, x):
          mu, sigma = theta
          return np.sum(lognorm.logpdf(x, s=sigma, scale=np.exp(mu)))
```

[11]:
```
params = mle_estimate(count, "first_function")
alpha_1 = params['alpha']
C_1 = params['C']

params = mle_estimate(count, "second_function")
alpha_2 = params['alpha']
C_2 = params['C']

params = mle_estimate(count, "third_function")
alpha_3 = params['alpha']
C_3 = params['C']
```

```
/var/folders/9l/0v1lmq9n7vv9hrrfhyb6bd4w0000gn/T/ipykernel_84873/399045807.py:27
: RuntimeWarning: invalid value encountered in log
  return -np.sum(np.log(1 / (zeta(beta, 1) * (count ** beta))))
/var/folders/9l/0v1lmq9n7vv9hrrfhyb6bd4w0000gn/T/ipykernel_84873/399045807.py:34
: RuntimeWarning: invalid value encountered in true_divide
  np.log(gamma(1) * (beta - 1) * gamma(count + 1 - beta) / (gamma(2 - beta) *
gamma(count + 1)))
/var/folders/9l/0v1lmq9n7vv9hrrfhyb6bd4w0000gn/T/ipykernel_84873/399045807.py:34
: RuntimeWarning: divide by zero encountered in log
  np.log(gamma(1) * (beta - 1) * gamma(count + 1 - beta) / (gamma(2 - beta) *
gamma(count + 1)))
/var/folders/9l/0v1lmq9n7vv9hrrfhyb6bd4w0000gn/T/ipykernel_84873/399045807.py:34
: RuntimeWarning: invalid value encountered in log
  np.log(gamma(1) * (beta - 1) * gamma(count + 1 - beta) / (gamma(2 - beta) *
gamma(count + 1)))
/var/folders/9l/0v1lmq9n7vv9hrrfhyb6bd4w0000gn/T/ipykernel_84873/399045807.py:34
: RuntimeWarning: overflow encountered in multiply
  np.log(gamma(1) * (beta - 1) * gamma(count + 1 - beta) / (gamma(2 - beta) *
gamma(count + 1)))
```

[12]:
```
fig = plt.figure(figsize = (10, 10))
plt.plot(rank, (count.sum() / C_1) * (rank ** (-alpha_1)), label='Best fit:␣
  ↪First Function')
plt.plot(rank, (count.sum() / C_2) * (rank ** (-alpha_2)), label='Best fit:␣
  ↪Second Function')
plt.plot(rank, (count.sum() / C_3) * (rank ** (-alpha_3)), label='Best fit:␣
  ↪Third Function')
plt.scatter(rank, count, label='Data')
plt.title("Different fits for Zipf's law", size = 15)
plt.xlabel("Rank of word (in log)", size = 15)
plt.ylabel("Frequency of occurance (in log)", size = 15)
plt.xscale('log')
plt.yscale('log')
plt.legend()
plt.grid()
plt.savefig('f1.png')
plt.show()
```

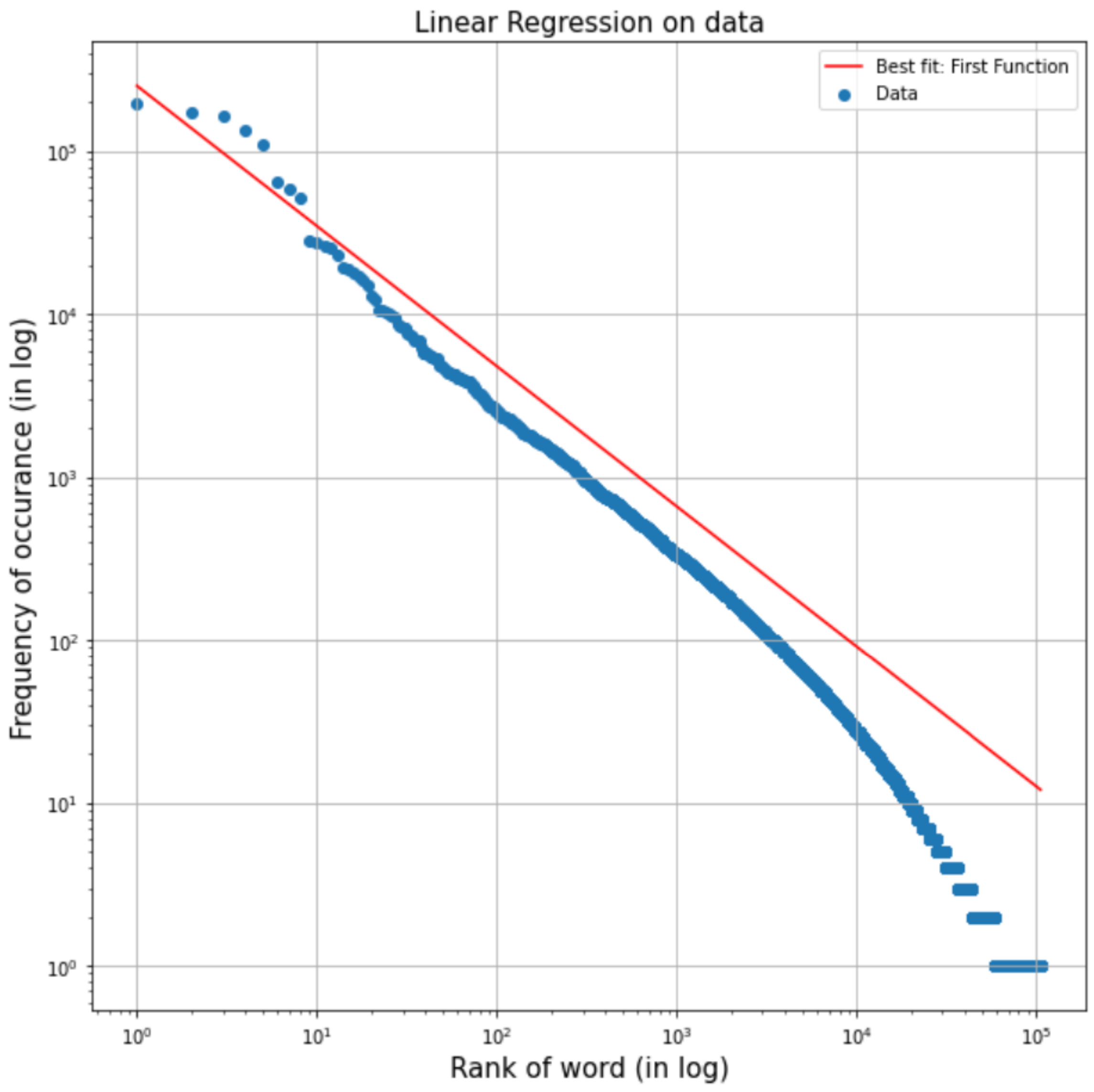


[13]:
```
def linear_fit(x, a, k):
    return k / x ** a

def least_squares_minimization(rank, count, curve, init_guess):
    params = []
    if curve == "linear":
        fit = curve_fit(linear_fit, rank, count, p0=init_guess)
        coeffs, cov = fit
        params = coeffs
    return params
```

```
a, k = least_squares_minimization(rank, count, "linear", [1, 1])
```

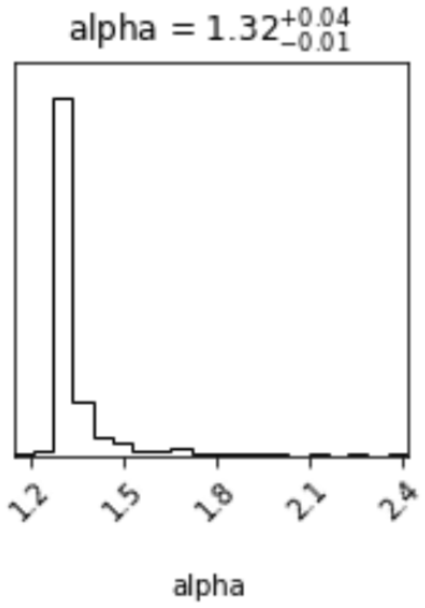
```
/var/folders/9l/0v1lmq9n7vv9hrrfhyb6bd4w0000gn/T/ipykernel_84873/3046372582.py:2
: RuntimeWarning: divide by zero encountered in true_divide
  return k / x ** a
/var/folders/9l/0v1lmq9n7vv9hrrfhyb6bd4w0000gn/T/ipykernel_84873/3046372582.py:2
: RuntimeWarning: overflow encountered in true_divide
  return k / x ** a
```

[14]:
```
fig = plt.figure(figsize = (10, 10))
plt.plot(rank, linear_fit(rank, a, k), color = 'red', label='Best fit: First␣
 ↪Function')
plt.scatter(rank, count, label='Data')
plt.title("Linear Regression on data", size = 15)
plt.xlabel("Rank of word (in log)", size = 15)
plt.ylabel("Frequency of occurance (in log)", size = 15)
plt.xscale('log')
plt.yscale('log')
plt.legend()
plt.grid()
plt.savefig('f2.png')
plt.show()
```



[15]:
```python
import emcee
import corner

# likelihood function is same the second function
def log_likelihood(alpha, count):
    beta = 1 / alpha + 1
    logl = np.sum(
        np.log((1 / count) ** (beta - 1) - (1 / (count + 1)) ** (beta - 1))
    )
    return logl

def log_prior(alpha):
    if 0.1 <= alpha <= 5:
        return 0.0
    else:
        return -np.inf

def log_posterior(alpha, count):
    lp = log_prior(alpha)
    if not np.isfinite(lp):
        return -np.inf
    return lp + log_likelihood(alpha, count)

nwalkers, ndim = 100, 1

pos = np.random.uniform(0.1, 5.0, size=(nwalkers, ndim))

sampler = emcee.EnsembleSampler(nwalkers, ndim, log_posterior, args=(count,))
nsteps = 50
sampler.run_mcmc(pos, nsteps, progress=True)

chain = sampler.chain[:, int(nsteps/2):, :].reshape(-1, ndim)

labels = ["alpha"]
fig = corner.corner(chain, labels=labels, show_titles=True)
```

```
100%|                                    | 50/50 [00:15<00:00,
3.16it/s]
```



[16]:
```python
from scipy.stats import zipf, weibull_min, lognorm, kstest
dict_counts = count
params_weibull = weibull_min.fit(dict_counts)
params_lognorm = lognorm.fit(dict_counts)
```

[17]:
```python
# p_zipf = kstest(dict_counts, 'zipf', args=params_zipf)
p_weibull = kstest(dict_counts, 'weibull_min', args=params_weibull)
p_lognorm = kstest(dict_counts, 'lognorm', args=params_lognorm)

# Print results
# print('Best fit:', best_fit)
# print('Best fit parameters:', best_params)
print('Zipf =', params, 'Weibull =', p_weibull, 'Log-normal =', p_lognorm)
```

```
Zipf = {'alpha': 0.12732200701925442, 'C': 27762.04397148232} Weibull =
KstestResult(statistic=0.4420059545849638, pvalue=0.0, statistic_location=1,
statistic_sign=1) Log-normal = KstestResult(statistic=0.4414598273971979,
pvalue=0.0, statistic_location=1, statistic_sign=1)
```