Covenants Data Cleaning

Loading pre-cleaned analysis

In [1]:
```
import pandas as pd
import numpy as np

Secondary_Data = pd.read_excel("Covenants_Results_Analysis.xlsx", sheet_name="Secondary_Data")
```

In [ ]:
```
Secondary_Data
```

In [3]:
```
Verified_Table = pd.read_excel("Covenants_Results_Analysis.xlsx", sheet_name="Verified_Table")
```

In [4]:
```
Verified_Table
```

Out[4]:

	query	index	macro
0	301	17558	218102
1	1039	19551	225310
2	1409	19357	224620
3	74	19018	223307
4	244	19504	225141
...	...	...	...
1707	1336	20358	227993
1708	2910	17886	219607
1709	265	21729	232829
1710	266	16726	215318
1711	2310	21734	232859

1712 rows × 3 columns

In [5]:
```
Primary_Reference_Table = pd.read_excel("Covenants_Results_Analysis.xlsx", sheet_name="Primary_Reference_Table")
```

In [ ]:
```
Primary_Reference_Table
```

In [7]:
```
verified_mask = Secondary_Data['query'].isin(Verified_Table['query'])
```

In [8]:
```
Secondary_Cleaned = Secondary_Data[verified_mask].copy()
Secondary_Messy = Secondary_Data[-verified_mask].copy()
```

In [ ]:
```
display(Secondary_Cleaned)
```

In [10]:
```
verified_mask_primary_table = Primary_Reference_Table["Customer_No"].isin(Secondary_Cleaned['Predicted_Macro'])
```

In [11]:
```
Primary_Reference_Table_no_clean = Primary_Reference_Table[-verified_mask_primary_table].copy()
```

In [ ]:
```
display(Primary_Reference_Table_no_clean)
```

In [ ]:
```
display(Secondary_Messy)
```

In [ ]:
```
print("All columns of Secondary_Messsy:")

print(Secondary_Messy.columns)
```

In [ ]:
```
display(Secondary_Messy['Predicted_Name'].value_counts().sort_values(ascending=False))
```

In [16]:
```
value_counts = Secondary_Messy['Predicted_Name'].value_counts()
count_list = value_counts.to_list()
```

In [17]:
```
count_list = np.array(count_list)
count_list
```

Out[17]:
```
array([74, 46, 40, 34, 33, 28, 28, 26, 17, 16, 15, 15, 12, 11, 10, 10, 10,
        8,  8,  8,  8,  7,  7,  7,  7,  7,  7,  6,  6,  6,  6,  6,  6,  6,
        6,  6,  5,  5,  5,  5,  5,  5,  5,  5,  4,  4,  4,  4,  4,  4,  4,
        4,  4,  4,  4,  4,  4,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,
        3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  2,
        2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,
        2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,
        2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,
        2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,
        2,  2,  2,  2,  2,  2,  2,  2,  1,  1,  1,  1,  1,  1,  1,  1,  1,
        1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
        1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
        1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
        1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
        1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
        1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
        1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
        1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
        1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
        1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
        1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
        1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
        1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
        1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
        1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
        1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
        1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
        1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
        1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1])
```

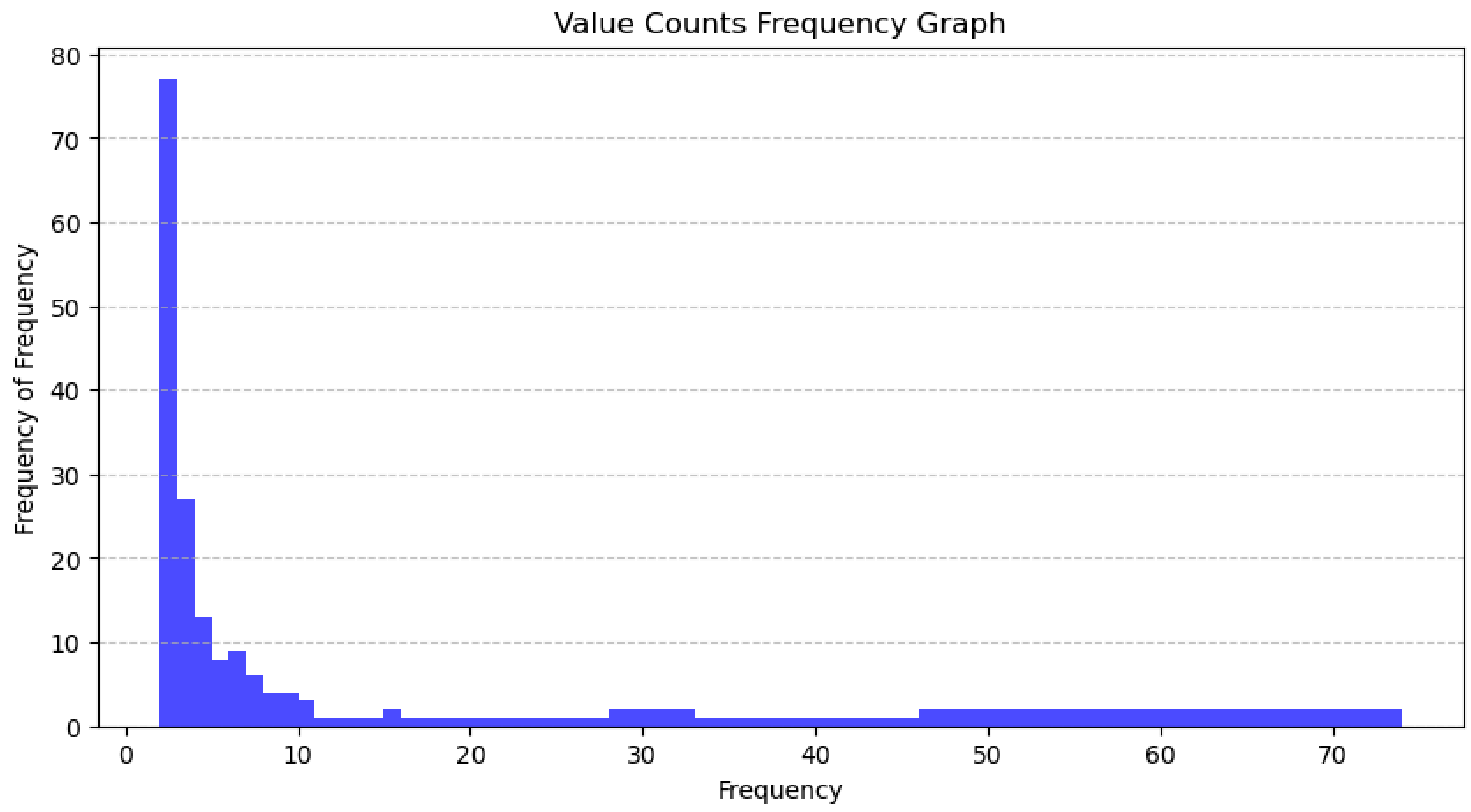
In [18]:
```
count_list = count_list[count_list!=1]
```

In [19]:
```
print(count_list)
```

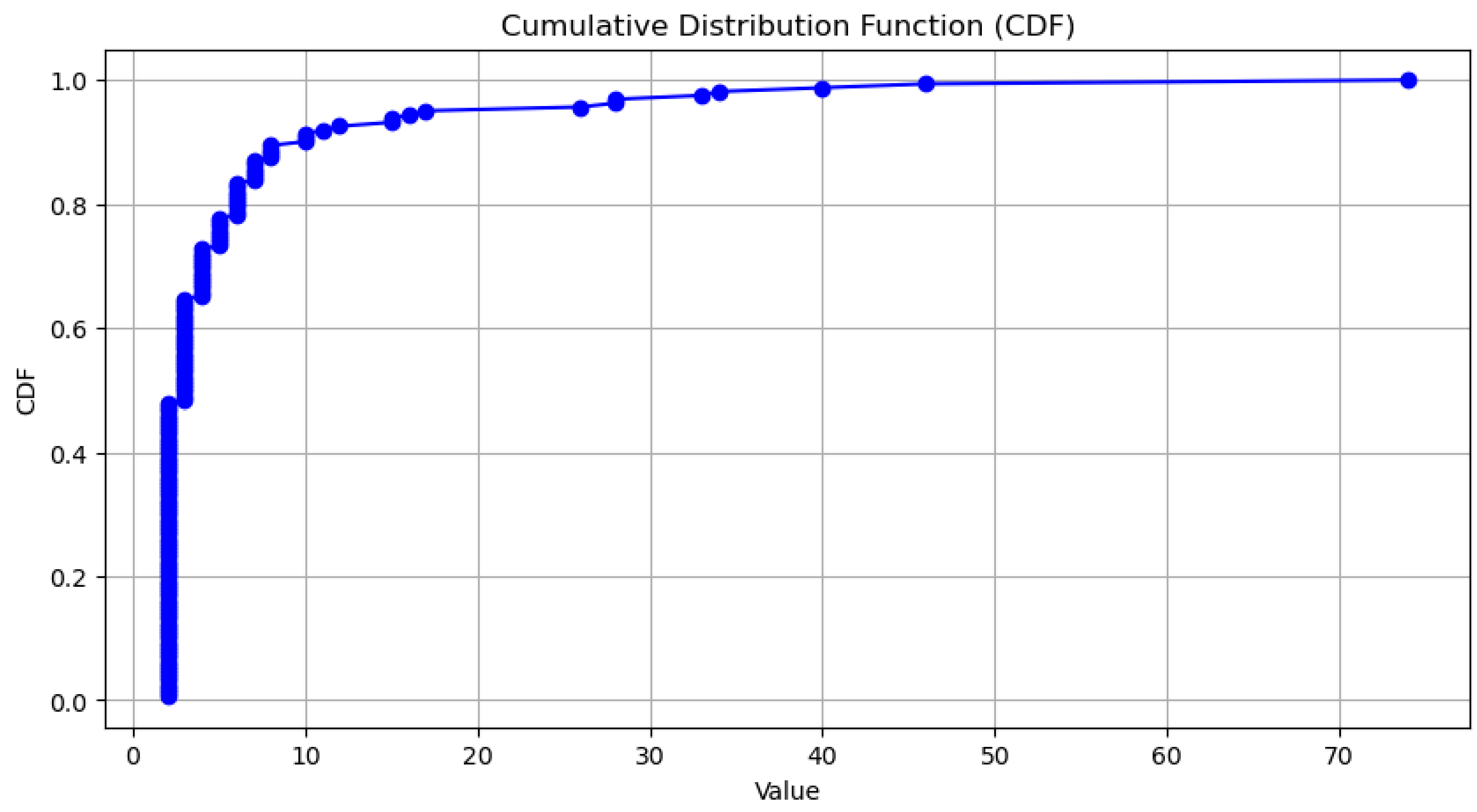
```
[74 46 40 34 33 28 28 26 17 16 15 15 12 11 10 10 10  8  8  8  8  7  7  7
  7  7  7  6  6  6  6  6  6  6  6  6  5  5  5  5  5  5  5  5  4  4  4  4
  4  4  4  4  4  4  4  4  4  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3
  3  3  3  3  3  3  3  3  3  3  3  3  2  2  2  2  2  2  2  2  2  2  2  2
  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2
  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2
  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2]
```

In [20]:
```
import matplotlib.pyplot as plt
import numpy as np

repetitions = np.array(count_list)
```

```
repetitions_increasing = np.sort(repetitions)

plt.figure(figsize=(10,5))
plt.hist(repetitions, bins=np.unique(repetitions), color='blue', alpha=0.7)
plt.xlabel("Frequency")
plt.ylabel("Frequency of Frequency")
plt.title("Value Counts Frequency Graph")
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()
```

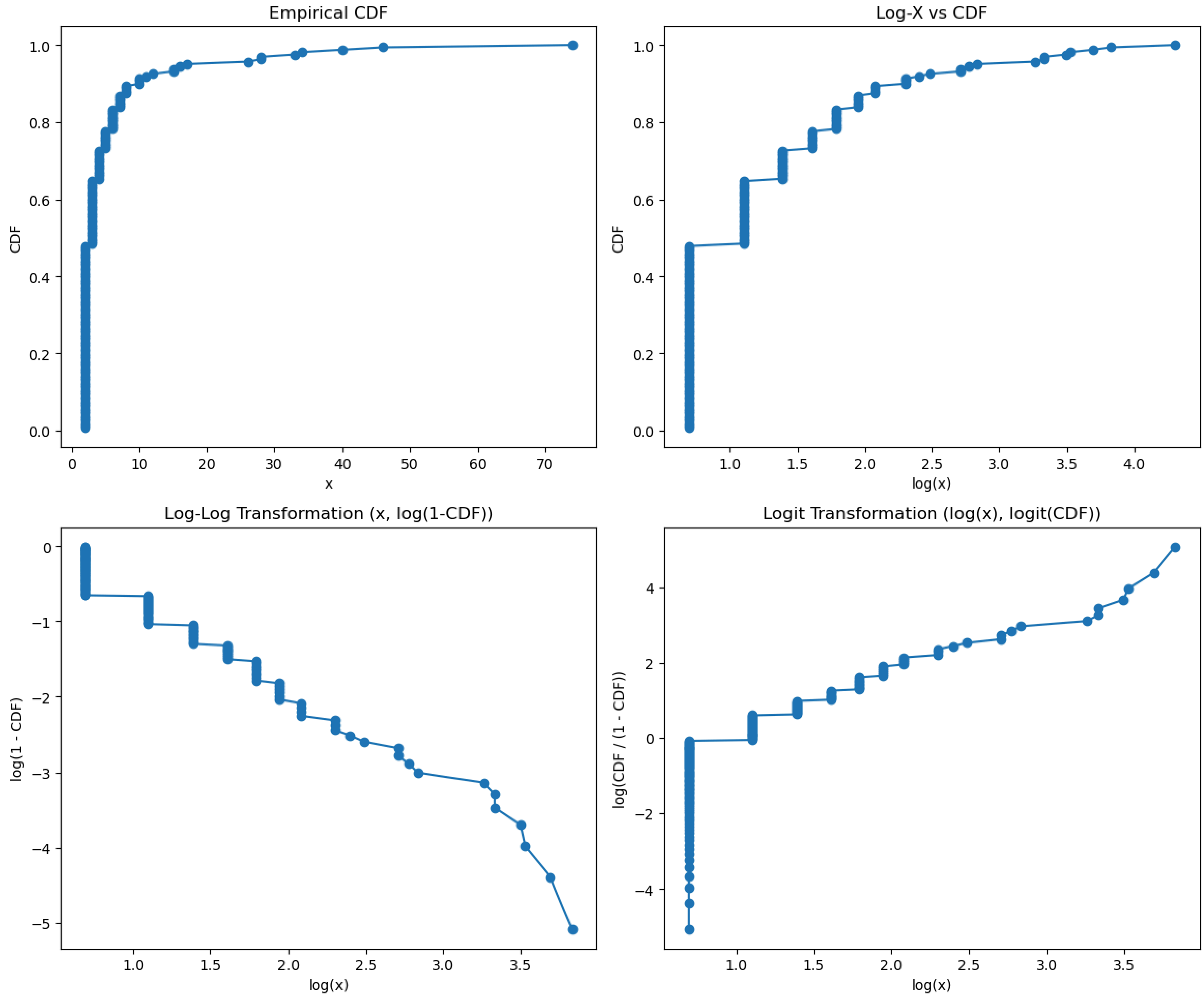


In [21]:
```
cdf = np.arange(1, len(repetitions_increasing) + 1) / len(repetitions_increasing)

plt.figure(figsize=(10,5))
plt.plot(repetitions_increasing, cdf, marker='o', linestyle='-', color='b')
plt.xlabel("Value")
plt.ylabel("CDF")
plt.title("Cumulative Distribution Function (CDF)")
plt.grid()
plt.show()
```



In [22]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import linregress

data_sorted = repetitions_increasing

# Compute the empirical CDF
n = len(data_sorted)
cdf = np.arange(1, n + 1) / n

# Apply Log Transformations
log_x = np.log(data_sorted)  # Log of data values
log_cdf = np.log(cdf)        # Log of CDF values
log1m_cdf = np.log(1 - cdf)  # Log(1 - CDF) for tail behavior
logit_cdf = np.log(cdf / (1 - cdf))  # Logit transformation

# Plot different transformations
fig, axes = plt.subplots(2, 2, figsize=(12, 10))

# Original CDF
axes[0, 0].plot(data_sorted, cdf, marker='o', linestyle='-')
axes[0, 0].set_title("Empirical CDF")
axes[0, 0].set_xlabel("x")
axes[0, 0].set_ylabel("CDF")

# Log-X Transformation
axes[0, 1].plot(log_x, cdf, marker='o', linestyle='-')
axes[0, 1].set_title("Log-X vs CDF")
axes[0, 1].set_xlabel("log(x)")
axes[0, 1].set_ylabel("CDF")

# Log-Log Transformation
axes[1, 0].plot(log_x, log1m_cdf, marker='o', linestyle='-')
axes[1, 0].set_title("Log-Log Transformation (x, log(1-CDF))")
axes[1, 0].set_xlabel("log(x)")
axes[1, 0].set_ylabel("log(1 - CDF)")

# Logit Transformation
axes[1, 1].plot(log_x, logit_cdf, marker='o', linestyle='-')
axes[1, 1].set_title("Logit Transformation (log(x), logit(CDF))")
axes[1, 1].set_xlabel("log(x)")
axes[1, 1].set_ylabel("log(CDF / (1 - CDF))")

plt.tight_layout()
plt.show()
```

```
/tmp/ipykernel_37387/3594380786.py:14: RuntimeWarning: divide by zero encountered in log
  log1m_cdf = np.log(1 - cdf)  # Log(1 - CDF) for tail behavior
/tmp/ipykernel_37387/3594380786.py:15: RuntimeWarning: divide by zero encountered in divide
  logit_cdf = np.log(cdf / (1 - cdf))  # Logit transformation
```



In [ ]:
```
top_repeated_names = value_counts[value_counts > 6].index.to_numpy()
print(top_repeated_names)
```

In [24]:
```
mask_too_repeated_PRF = Primary_Reference_Table_no_clean['Customer_Title'].isin(top_repeated_names)
print(mask_too_repeated_PRF)
```

```
0        False
1        False
2        False
3        False
4        False
         ...  
22288    False
22289    False
22290    False
22292    False
22293    False
Name: Customer_Title, Length: 20656, dtype: bool
```

In [25]:
```
Primary_Reference_Table_No_Common_Names =  Primary_Reference_Table_no_clean[-mask_too_repeated_PRF]
```

In [ ]:
```
Primary_Reference_Table_No_Common_Names
```

Finding the most common Customer_Title words

In [27]:
```
from collections import Counter
import re

all_words = " ".join(Primary_Reference_Table_No_Common_Names["Customer_Title"]).lower()


all_words = re.findall(r'\b\w+\b', all_words)

word_counts = Counter(all_words)


word_freq_df = pd.DataFrame(word_counts.items(), columns=["Words", "Count"]).sort_values(by="Count", ascending=False)
```

In [28]:
```
word_freq_df
```

Out[28]:

	Words	Count
6	or	2737
87	a	2288
41	de	2199
30	s	2198
378	inc	1635
...	...	...
8610	piuma	1
8611	lambruschini	1
8612	quimpro	1
8613	mallen	1
18077	syndicated	1

18078 rows × 2 columns

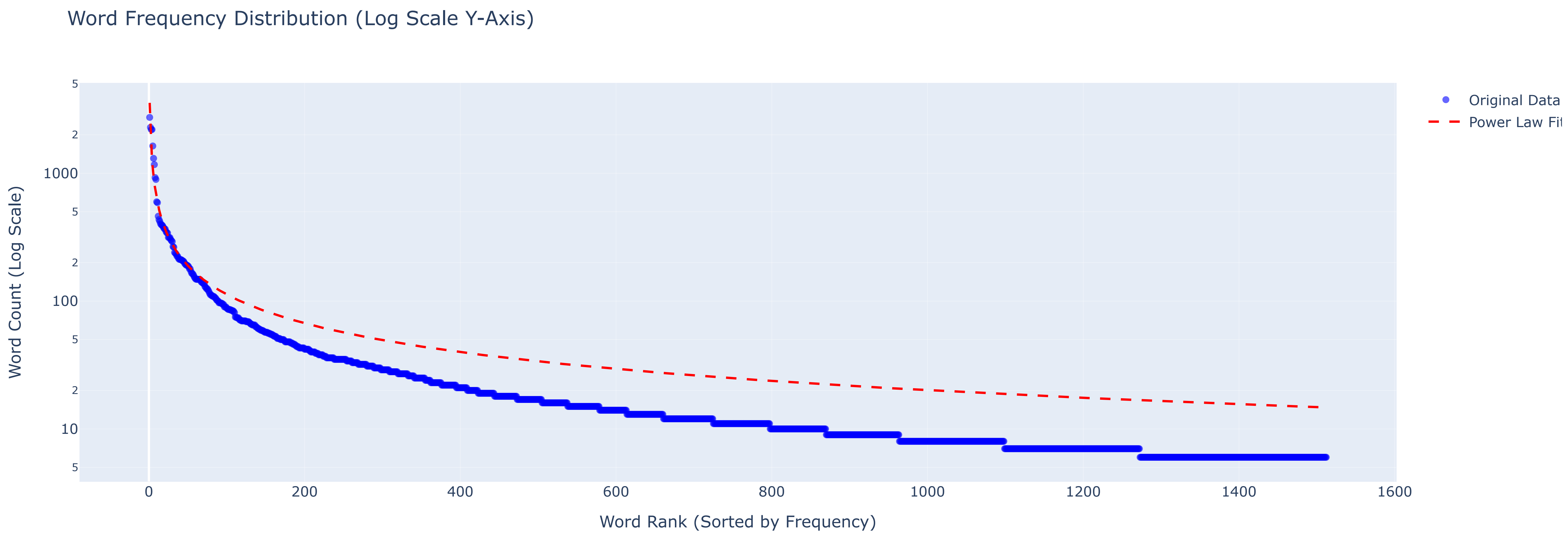
In [ ]:

```
from collections import Counter
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression


low_bound = 5

word_freq_df = word_freq_df[word_freq_df["Count"] > low_bound ]

# Plot the distribution of word counts along sorted words
plt.figure(figsize=(10, 5))
plt.plot(word_freq_df["Count"].values, marker="o", linestyle="-", color="blue")

# Labels and title
plt.xlabel("Words (Sorted by Frequency)", fontsize=12)
plt.ylabel("Word Count", fontsize=12)
plt.title("Distribution of Word Frequency in Customer_Title", fontsize=14)
plt.grid(True, linestyle="--", alpha=0.7)

# Show the plot
plt.show()

# Generate Rank-based X values (1, 2, 3, ...)
word_freq_df["Rank"] = range(1, len(word_freq_df) + 1)

# Extract X (word rank) and Y (word frequency)
x = word_freq_df["Rank"].values
y = word_freq_df["Count"].values

### Regression Models ###
# 1. Power Law Regression (y = a * x^b)
def power_law(x, a, b):
    return a * np.power(x, b)

params_power, _ = curve_fit(power_law, x, y, maxfev=10000)
a_power, b_power = params_power

# 2. Exponential Decay Regression (y = a * e^(-bx))
def exp_decay(x, a, b):
    return a * np.exp(-b * x)

params_exp, _ = curve_fit(exp_decay, x, y, maxfev=10000)
a_exp, b_exp = params_exp

# 3. Polynomial Regression (Degree 2)
poly = PolynomialFeatures(degree=2)
x_poly = poly.fit_transform(x.reshape(-1, 1))
lin_reg = LinearRegression()
lin_reg.fit(x_poly, y)

# Predictions
x_range = np.linspace(min(x), max(x), 500)
y_power_pred = power_law(x_range, a_power, b_power)
y_exp_pred = exp_decay(x_range, a_exp, b_exp)
y_poly_pred = lin_reg.predict(poly.transform(x_range.reshape(-1, 1)))

### Plot Results ###
plt.figure(figsize=(10, 6))
plt.scatter(x, y, color='blue', label="Original Data", alpha=0.5)
plt.plot(x_range, y_power_pred, label="Power Law Fit", color="red", linestyle="--")
plt.plot(x_range, y_exp_pred, label="Exponential Decay Fit", color="green", linestyle="-.")
plt.plot(x_range, y_poly_pred, label="Polynomial Fit (Degree 2)", color="orange", linestyle=":")

plt.xlabel("Word Rank (Sorted by Frequency)")
plt.ylabel("Word Count")
plt.title("Regression Fits to Flatten Word Frequency Curve")
plt.legend()
plt.grid(True)
plt.show()
```

In [30]:

```
import numpy as np
import pandas as pd
import plotly.graph_objects as go
from scipy.optimize import curve_fit
from collections import Counter


# Extract values
x = word_freq_df["Rank"].values
y = word_freq_df["Count"].values
words = word_freq_df["Words"].values  # Store words for hovering

### Power Law Regression (y = a * x^b)
def power_law(x, a, b):
    return a * np.power(x, b)

params_power, _ = curve_fit(power_law, x, y, maxfev=10000)
a_power, b_power = params_power

# Generate predictions for a smooth curve
x_range = np.linspace(min(x), max(x), 500)
y_power_pred = power_law(x_range, a_power, b_power)

# Create Interactive Plot with Plotly
fig = go.Figure()

# Scatter plot of original data points
```

```
fig.add_trace(go.Scatter(
    x=x, y=y,
    mode='markers',
    marker=dict(size=6, color='blue', opacity=0.6),
    text=[f"Word: {word}, Count: {count}" for word, count in zip(words, y)],
    hoverinfo='text',
    name="Original Data"
))

# Power-law fitted curve
fig.add_trace(go.Scatter(
    x=x_range, y=y_power_pred,
    mode='lines',
    line=dict(color='red', dash='dash'),
    name="Power Law Fit"
))

# Configure Layout
fig.update_layout(
    title="Word Frequency Distribution (Log Scale Y-Axis)",
    xaxis=dict(title="Word Rank (Sorted by Frequency)"),
    yaxis=dict(title="Word Count (Log Scale)", type="log"),  # Log scale applied to Y
    hovermode="closest"
)

# Show Interactive Plot
fig.show()
```

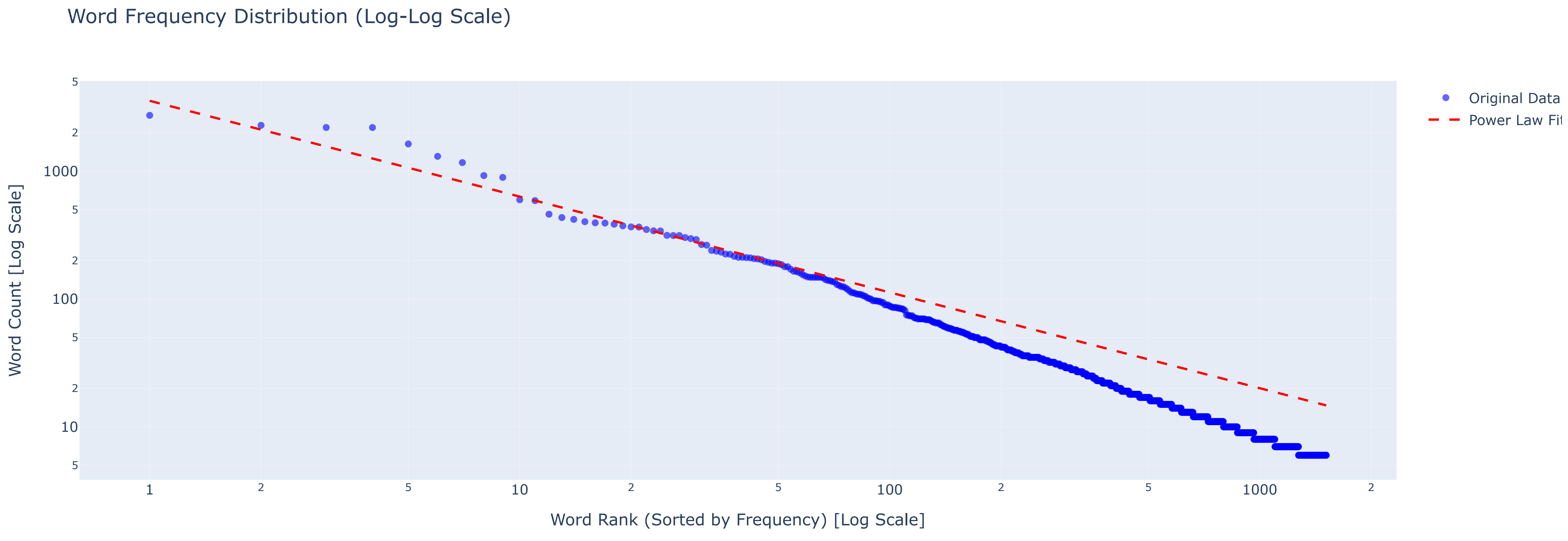


In [ ]:

In [31]:
```
private_mask = Primary_Reference_Table_No_Common_Names["Sector"].str.contains("Private", case=False, na=False)

Primary_Reference_Table_No_Common_Names.loc[private_mask, "Sector"] = "Personal"
```

In [ ]:
```
Primary_Reference_Table_No_Common_Names
```

In [33]:
```
### Power Law Regression (y = a * x^b)
def power_law(x, a, b):
    return a * np.power(x, b)

params_power, _ = curve_fit(power_law, x, y, maxfev=10000)
a_power, b_power = params_power

# Generate predictions for a smooth curve
x_range = np.linspace(min(x), max(x), 500)
y_power_pred = power_law(x_range, a_power, b_power)

# Create Interactive Log-Log Plot with Plotly
fig = go.Figure()

# Scatter plot of original data points
fig.add_trace(go.Scatter(
    x=x, y=y,
    mode='markers',
    marker=dict(size=6, color='blue', opacity=0.6),
    text=[f"Word: {word}, Count: {count}" for word, count in zip(words, y)],
    hoverinfo='text',
    name="Original Data"
))

# Power-law fitted curve
fig.add_trace(go.Scatter(
    x=x_range, y=y_power_pred,
    mode='lines',
    line=dict(color='red', dash='dash'),
    name="Power Law Fit"
))

# Configure Layout (Log-Log Scale)
fig.update_layout(
    title="Word Frequency Distribution (Log-Log Scale)",
    xaxis=dict(title="Word Rank (Sorted by Frequency) [Log Scale]", type="log"),
    yaxis=dict(title="Word Count [Log Scale]", type="log"),  # Log scale applied to both axes
    hovermode="closest"
)

# Show Interactive Plot
fig.show()
```



Displaying common words above threshold

In [34]:
```
threshold = 150
```

In [37]:
```
words_abv_thrsh = word_freq_df[word_freq_df["Count"] > 150]
```

In [38]:
```
#words_abv_thrsh = words_abv_thrsh.drop(columns=["Rank"])

words_abv_thrsh
```

Out[38]:

	Words	Count	Rank
6	or	2737	1
87	a	2288	2
41	de	2199	3
30	s	2198	4
378	inc	1635	5
11	maria	1308	6
1729	llc	1169	7
362	c	925	8
136	jose	895	9
386	corp	598	10
395	ltd	589	11
151	l	461	12
363	v	434	13
28	luis	420	14
311	limited	403	15
101	juan	395	16
412	investments	393	17
196	antonio	385	18
88	garcia	374	19
284	gonzalez	367	20
132	del	366	21
242	rodriguez	350	22
54	carlos	342	23
394	international	341	24
430	sa	315	25
425	corporation	314	26
387	the	314	27
21	francisco	303	28
260	manuel	297	29
126	fernandez	292	30
4812	seafood	267	31
439	usa	264	32
50	lopez	240	33
85	perez	237	34
1776	group	233	35
502	trust	225	36
179	y	224	37
114	r	216	38
310	holdings	212	39
561	banco	212	40
1151	trading	211	41
312	la	210	42
1101	investment	207	43
185	martinez	206	44
228	jorge	203	45
9	eduardo	197	46
616	sanchez	194	47
453	and	191	48
86	miguel	191	49
93	alberto	189	50
1460	company	186	51
375	inversiones	179	52
69	jesus	179	53
479	gomez	171	54
112	fernando	165	55
17	enrique	164	56
36	m	161	57
79	carmen	156	58
145	javier	152	59

In [42]:

```
stop_words = words_abv_thrsh["Words"].astype(str).tolist()
stop_words = ", ".join(f'"{value}"' for value in stop_words)

print(stop_words)
```

```
"or", "a", "de", "s", "inc", "maria", "llc", "c", "jose", "corp", "ltd", "l", "v", "luis", "limited", "juan", "investments", "antonio", "garcia", "gonzalez", "del", "rodriguez", "carlos", "int
ernational", "sa", "corporation", "the", "francisco", "manuel", "fernandez", "seafood", "usa", "lopez", "perez", "group", "trust", "y", "r", "holdings", "banco", "trading", "la", "investment",
"martinez", "jorge", "eduardo", "sanchez", "and", "miguel", "alberto", "company", "inversiones", "jesus", "gomez", "fernando", "enrique", "m", "carmen", "javier"
```

In [45]:

```
Primary_Reference_Table_No_Common_Names.to_excel("./Primary_Reference_Table_No_Common_Names.xlsx", sheet_name="Primary_Reference_Table", index=False)
```

In [46]:

```
Secondary_Messy.to_excel("./Messy_Secondary_Data.xlsx", sheet_name="Secondary_Data", index=False)
```

Fine Tunning and Machine Learning

After trying several configurations for the tokenization process as well as the weight assignation and normalization; the results are still off from the needed accuracy threshold. Considering manual configuration is arduous and repetitive it results a good decision to automatize the training of this hyper parameters over the data. During the first developement phase we were able to combine manual and computer labor to clean ≈1713 records which in turn will help us train our "model" over a loss cross-entropy loss function for the first development phase.

Loading Java Results

In [36]:
```
import pandas as pd
results = pd.read_csv("results.csv")
```

In [37]:
```
results
```

Out[37]:

	query	match	secondMatch	coefficientDamerau	coefficientJaccard	idMatch
0	0	-1	-1	-1.000	-1.000	0
1	1	5014	-1	0.238	0.052	1
2	2	4623	-1	0.205	0.039	1
3	3	18633	-1	0.242	0.026	1
4	4	6510	-1	0.195	0.027	1
...	...	...	...	...	...	...
2909	2909	16928	13715	0.054	0.015	0
2910	2910	16928	21513	0.054	0.030	0
2911	2911	15959	-1	0.097	0.014	1
2912	2912	10153	-1	0.118	0.026	1
2913	2913	-1	-1	-1.000	-1.000	0

2914 rows × 6 columns

Loading Verified Mappings (Manual Results)

In [ ]:
```
verified = pd.read_excel("Covenants_Results_Analysis.xlsx")
verified
```

Evaluation metrics

Let's compute the accuracy for first (top) match

In [39]:
```
import pandas as pd
import numpy as np
from sklearn.metrics import (
    confusion_matrix, accuracy_score, precision_score,
    recall_score, f1_score, classification_report
)

def compute_evaluation_metrics(verified, results, v_header: str, r_header: str):

    # Convert to numeric, forcing non-numeric values to NaN
    results[r_header] = pd.to_numeric(results[r_header], errors="coerce")
    verified[v_header] = pd.to_numeric(verified[v_header], errors="coerce")

    # Filter valid numeric entries
    valid_indices = verified[v_header].notna() & results[r_header].notna()

    # Extract valid data
    y_true = verified.loc[valid_indices, v_header]
    y_pred = results.loc[valid_indices, r_header]

    # Compute confusion matrix
    conf_matrix = confusion_matrix(y_true, y_pred, labels=np.unique(y_true))

    # Output confusion matrix
    print("Confusion Matrix:\n", conf_matrix)

    # Compute evaluation metrics
    accuracy = accuracy_score(y_true, y_pred)
    precision = precision_score(y_true, y_pred, average='macro', zero_division=0)
    recall = recall_score(y_true, y_pred, average='macro', zero_division=0)
    f1 = f1_score(y_true, y_pred, average='macro', zero_division=0)

    evaluation_metrics = {"accuracy":accuracy, "precision": precision, "recall": recall, "f1": f1}
    return evaluation_metrics

v_header = "match"
r_header = "match"

metrics = compute_evaluation_metrics(verified, results, v_header, r_header)
snd_metrics = compute_evaluation_metrics(verified, results, v_header, "secondMatch")

# Full classification report
#print("\nClassification Report:")
#print(classification_report(y_true, y_pred, zero_division=0))

def print_eval_metrics(metrics):
    print("\nEvaluation Metrics:")
    print(f"Accuracy: {metrics['accuracy']:.4f}")
    print(f"Precision: {metrics['precision']:.4f}")
    print(f"Recall: {metrics['recall']:.4f}")
    print(f"F1 Score: {metrics['f1']:.4f}")
```

```
Confusion Matrix:
 [[1 0 0 ... 0 0 0]
 [0 1 0 ... 0 0 0]
 [0 0 1 ... 0 0 0]
 ...
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]]
Confusion Matrix:
 [[0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 ...
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]]
```

Metrics for first top match

In [40]:
```
print_eval_metrics(metrics)
```

```
Evaluation Metrics:
Accuracy: 0.8989
Precision: 0.8662
Recall: 0.8629
F1 Score: 0.8615
```

Metrics for second top match

In [41]:
```
print_eval_metrics(snd_metrics)
```

```
Evaluation Metrics:
Accuracy: 0.0000
Precision: 0.0000
Recall: 0.0000
F1 Score: 0.0000
```

Metrics for combined results

This computes the metrics regardeless on whether the correct result was in second or first place.

In [42]:
```
import pandas as pd

# Assuming "verified" and "results" have the same index
# Merge the dataframes on a common index or key
merged_df = results.merge(verified[['match']], left_index=True, right_index=True, suffixes=('_res', '_ver'))

# Define function to check matches
def find_combined_match(row):
    if row['match_res'] == row['match_ver']:
        return row['match_res']
    elif row['secondMatch'] == row['match_ver']:
        return row['secondMatch']
    return row['match_res']  # Default to match_res if no match is found

# Apply the function to create "combinedMatch" column
merged_df['combinedMatch'] = merged_df.apply(find_combined_match, axis=1)

# If you want to update the original "results" dataframe
results['combinedMatch'] = merged_df['combinedMatch']



cmbnd_metrics = compute_evaluation_metrics(verified, results, v_header, "combinedMatch")

print_eval_metrics(cmbnd_metrics)
```

```
Confusion Matrix:
 [[1 0 0 ... 0 0 0]
 [0 1 0 ... 0 0 0]
 [0 0 1 ... 0 0 0]
 ...
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]]

Evaluation Metrics:
Accuracy: 0.8989
Precision: 0.8662
Recall: 0.8629
F1 Score: 0.8615
```