

main

November 7, 2023

1 Sprawozdanie z badania Manuskryptu Wojnicza

1.0.1 Analiza porównawcza wymienionego

1.1 Autorzy:

- Kubala Piotr
- Łukosz Rafał

1.2 Importowanie bibliotek

```
[ ]: import re
from matplotlib import pyplot as plt
import numpy as np
import scipy
```

1.3 Przygotowanie danych tekstowych do analizy

```
[ ]: def extractParagraphs(file) -> list[str]:
    """
    This function take a raw text file, rids it of comments and empty lines,
    and returns a list of paragraphs.
    """
    paragraphs = []
    current_paragraph = ""

    for line in file:
        if line.strip().startswith("#") or len(line) < 2:
            continue

        last_character = line[-2]
        if last_character in ["-", "="]:
            current_paragraph += line[:-2] # '=' and new line character
        else:
            current_paragraph += line[:-1] # new line character

        if last_character == "=":
```

```

        paragraphs.append(selectFirstVersionForEachWord(current_paragraph.
strip()))
        current_paragraph = ""

    return paragraphs

def selectFirstVersionForEachWord(paragraph: str) -> str:
    """
    Source text contains ambiguous signs. The researchers weren't certain which
    sign was intended, so they left all possibilities.
    This function returns the first possibility.
    Example: ... first/second/third ... -> ... first ...
    """
    return re.sub(r"\((.*?)\|.*?\)", r"\1", paragraph)

def extractWordsFromParagraphAndLower(paragraph: str) -> list[str]:
    """
    This function takes a paragraph, splits it into words and makes them
    lowercase.
    Returns a list of words.
    """
    # split by spaces, commas, dots, brackets, colons, semicolons and quotes
    return re.split(
        r'[\s,\.\[\]\{\}:;"]', paragraph.lower().replace("-", "").replace("_", " ")
    )

```

```

[ ]: def readFormattedText(path: str) -> list[list[str]]:
    """
    Eventually you get a list of pure words. No comments, no empty lines, no
    punctuation.
    """
    with open(path, "r", encoding="utf-8") as f:
        return list(map(extractWordsFromParagraphAndLower,
extractParagraphs(f)))

def readUnformattedText(path: str) -> list[list[str]]:
    """
    Eventually you get a list of unchanged words from the source text.
    """
    with open(path, "r", encoding="utf-8") as f:
        return list(
            map(
                processWordsInLine,

```

```

        (map(extractWordsFromParagraphAndLower, f.readlines())),
    )
)

def processWordsInLine(line: list[str]) -> list[str]:
    """
    Removes descriptive characters from a line and returns a list of words.
    """
    return [
        one_word
        for one_word in map(lambda word: re.sub(r"\(\)\-\>\]", "", word), line)
        if len(one_word) > 0
    ]

```

1.4 Analiza n-gramów

```
[ ]: def countNGrams(text: list[list[str]], n: int, min_count: int) -> dict[str, int]:
    """
    Count the number of occurrences of desired ngrams.
    NGrams are sequences of n words, where n - number of words in sequence,
    min_count - minimal number of occurrences
    Returns a dict: {ngram: number of occurrences}
    """
    ngrams = {}

    for paragraph in text:
        for i in range(len(paragraph) - n + 1):
            ngram = " ".join(paragraph[i : i + n])
            if ngram not in ngrams:
                ngrams[ngram] = 0
            ngrams[ngram] += 1

    return {k: v for k, v in ngrams.items() if v >= min_count}
```

```
[ ]: def countNumbersOfDifferentNGramsForEachN(
    text: list[list[str]], min_n: int, max_n: int, min_count_for_ngram: int
) -> dict[int, int]:
    """
    Count the number of occurrences of desired ngrams.
    NGrams are sequences of n words, where n - number of words in sequence,
    min_count_for_ngram - minimal number of occurrences
    Returns a dict: {n: number of ngrams}
    """
    ngrams = {
        n: len(countNGrams(text, n, min_count_for_ngram))
    }
```

```

        for n in range(min_n, max_n + 1)
    }

    return ngrams

```

[]: def getSortedNGramsAndCalculateScores(
 ngrams: dict[str, int]
) -> list[tuple[str, int, float]]:
 """
 Return a list of tuples: (ngram, number of occurrences, score)
 """
 return [
 (ngram, count, (index + 1) / count)
 for index, (ngram, count) in enumerate(
 sorted(ngrams.items(), key=lambda x: x[1], reverse=True)
)
]

[]: def plotNGramsCountsForEachNUsingLogarithmicScale(
 ngramsCountsLists: list[list[tuple[str, int, float]]], labels: list[str]
):
 """
 Plot the number of ngrams as a function of n using logarithmic scale.
 """

 for ngramsCountsList in ngramsCountsLists:
 x_values = list(range(1, len(ngramsCountsList) + 1))
 y_values = [count for _, count, _ in ngramsCountsList]

 plt.plot(x_values, y_values)

 plt.xlabel("n")
 plt.ylabel("Number of ngrams")
 plt.legend(labels)
 plt.xscale("log")
 plt.yscale("log")
 plt.show()

[]: def fitHyperbolaToNgramsCounts(ngrams: list[tuple[str, int, float]]) -> float:
 """
 Fit a hyperbola to the ngrams counts and return the coefficient of determination.
 """

 x = list(range(1, len(ngrams) + 1))
 y = [count for _, count, _ in ngrams]

 popt, _ = scipy.optimize.curve_fit(lambda t, a: a / t, x, y)

```
    return popt[0]
```

1.5 Graf 2-gramów

```
[ ]: def plotNGramsCountsForEachN(ngramsCountsList: list[dict[int, int]], labels: list[str]):  
    """  
        Plot the number of ngrams as a function of n.  
    """  
    ngramsListSize = len(ngramsCountsList)  
    width = 0.8 / ngramsListSize  
    for i, ngramsCounts in enumerate(ngramsCountsList):  
        plt.bar(  
            np.array(list(ngramsCounts.keys())) + i * width,  
            list(ngramsCounts.values()),  
            width=width,  
        )  
  
        plt.xlabel("n")  
        plt.ylabel("Number of ngrams")  
        plt.legend(labels)  
        plt.show()
```

```
[ ]: def plotNGramsCounts(ngrams: list[tuple[str, int, float]]) -> None:  
    hyperbola_parameter = fitHyperbolaToNgramsCounts(ngrams)  
    x_values = list(range(1, len(ngrams) + 1))  
    y_hyperbola_values = [hyperbola_parameter / x for x in x_values]  
  
    plt.plot(x_values, [count for _, count, _ in ngrams], "b")  
    plt.plot(x_values, y_hyperbola_values, "r")  
    plt.show()
```

```
[ ]: def getWordsGraphs(  
    text: list[list[str]],  
) -> tuple[dict[str, dict[str, int]], dict[str, dict[str, int]]]:  
    """  
        graph_forward - key is a word, value is a dictionary {next_word: number of occurrences}  
        graph_backward - key is a word, value is a dictionary {previous_word: number of occurrences}  
    """  
  
    graph_forward = {}  
    graph_backward = {}  
  
    for paragraph in text:  
        for i in range(len(paragraph) - 1):
```

```

        word = paragraph[i]
        next_word = paragraph[i + 1]

        if word not in graph_forward:
            graph_forward[word] = {next_word: 1}
        else:
            next_words = graph_forward[word]
            if next_word not in next_words:
                next_words[next_word] = 1
            else:
                next_words[next_word] += 1

        if next_word not in graph_backward:
            graph_backward[next_word] = {word: 1}
        else:
            previous_words = graph_backward[next_word]
            if word not in previous_words:
                previous_words[word] = 1
            else:
                previous_words[word] += 1

    return (graph_forward, graph_backward)

def filterGraphByRequiringMinimalDegreeAndCount(
    graph: dict[str, dict[str, int]], minimal_degree: int, minimal_count: int
) -> dict[str, dict[str, int]]:
    """
    Were we to plot the original graph, it would be illegible. This function
    filters the graph by requiring:
    - a minimal degree (smallest number of edges connected to any vertex in the
    graph)
    - count (minimal number of occurrences of a pair of words)
    Returns
    """
    return {
        k: v
        for k, v in graph.items()
        if len(v) >= minimal_degree and max(v.values()) >= minimal_count
    }

def getFilteredWordsGraph(
    text: list[list[str]], minimal_degree: int, minimal_count: int
) -> tuple[dict[str, dict[str, int]], dict[str, dict[str, int]]]:
    """

```

```

Main function. Returns two graphs: one for words preceding other words, one for words following other words.
"""

graph_forward, graph_backward = getWordsGraphs(text)
filtered_graph_forward = filterGraphByRequiringMinimalDegreeAndCount(
    graph_forward, minimal_degree, minimal_count
)
filtered_graph_backward = filterGraphByRequiringMinimalDegreeAndCount(
    graph_backward, minimal_degree, minimal_count
)

all_words_preceding = set(filtered_graph_forward.keys())
all_words_following = set(filtered_graph_backward.keys())

return (
    {
        k: {word: count for word, count in v.items() if word in
            all_words_following}
            for k, v in filtered_graph_forward.items()
    },
    {
        k: {word: count for word, count in v.items() if word in
            all_words_preceding}
            for k, v in filtered_graph_backward.items()
    },
)

def getNMostCommonWordsWithMostConnections(text: list[list[str]]) ->
    list[tuple[str, int]]:
    """

Returns a list of tuples: (word, number of connections)
"""

graph_forward, graph_backward = getWordsGraphs(text)

all_words = set(graph_forward.keys()) | set(graph_backward.keys())

all_degrees = [
    (len(graph_forward.get(word, {})) + len(graph_backward.get(word, {})), word)
    for word in all_words
]
all_degrees.sort(reverse=True)

return [(word, degree) for degree, word in all_degrees]

```

1.6 Rysowanie

```
[ ]: def plotNWordsWithConnectionCounts(
    connection_counts_list: list[list[tuple[str, int]]], legend: list[str]
) -> None:
    """
    Plot the number of connections as a function of the word's rank.
    """
    wordsListSize = len(connection_counts_list)
    number_of_words = min(map(len, connection_counts_list))
    width = 0.8 / wordsListSize

    for i, connectionCounts in enumerate(connection_counts_list):
        truncated_connection_counts = connectionCounts[:number_of_words]

        x_values = (
            np.array(list(range(1, len(truncated_connection_counts) + 1))) + i * width
        )
        y_values = [count for _, count in truncated_connection_counts]

        plt.bar(x_values, y_values, width=width)

        x_ticks_values = [
            "\n".join(
                [connection_counts[i][0] for connection_counts in connection_counts_list]
            )
            for i in range(number_of_words)
        ]
        x_ticks_x_values = (
            np.array(list(range(1, len(x_ticks_values) + 1)))
            + (wordsListSize - 1) * width / 2
        )

        plt.xticks(x_ticks_x_values, x_ticks_values, rotation=90)

        plt.xlabel("Miejsce w rankingu ilości połączeń")
        plt.ylabel("Ilość połączeń")
        plt.legend(legend)

    plt.show()
```

```
[ ]: def drawBipartialGraph(
    graph: tuple[dict[str, dict[str, int]], dict[str, dict[str, int]]],
    title: str,
    print_labels: bool = False,
```

```

):
    """
    Draws a graph.
    """

    plt.figure(figsize=(30, 30))
    plt.title(title)
    plt.axis("off")

    graph_network_forward, graph_network_backward = graph

    max_degree_first_word = max(map(len, graph_network_forward.values()))
    max_degree_second_word = max(map(len, graph_network_backward.values()))

    max_count = max(map(max, map(lambda x: x.values(), graph_network_forward.
        ↪values())))

    all_words = set(graph_network_forward.keys())
    for words in graph_network_forward.values():
        all_words.update(words.keys())

    all_words_list = list(all_words)
    all_words_list.sort()

    word_to_index = {word: i for i, word in enumerate(all_words_list)}

    for word, words in graph_network_forward.items():
        for next_word in words.keys():
            count = graph_network_forward[word][next_word]
            count_scaled = count / max_count

            degree_scaled_first_word = len(words) / max_degree_first_word
            degree_scaled_second_word = (
                len(graph_network_backward[next_word]) / max_degree_second_word
            )

            first_word_y = word_to_index[word]
            second_word_y = word_to_index[next_word]

            plt.plot(
                [0, 1], [first_word_y, second_word_y], color=(0, 0, ↪
            ↪count_scaled, 1)
            )

            if print_labels:
                plt.text(0, first_word_y, word, horizontalalignment="right")
                plt.text(1, second_word_y, next_word, ↪
            ↪horizontalalignment="left")

```

```

plt.show()

def plotKMostCommonNGrams(
    k: int, legend: list[str], *ngrams_tuple: list[tuple[str, int, float]]
) -> None:
    plt.figure(figsize=(12, 10))
    plt.title("Most common ngrams")
    plt.xlabel("2-gram")
    plt.ylabel("Number of occurrences")
    x_values_count = min(k, min(list(map(len, ngrams_tuple))))
    number_of_ngrams_lists = len(ngrams_tuple)
    bar_width = 0.8 / number_of_ngrams_lists
    x_labels = [
        "\n".join([ngrams[i][0] for ngrams in ngrams_tuple])
        for i in range(x_values_count)
    ]

    plt.xticks(np.array(range(1, x_values_count + 1)) + 0.4, x_labels, rotation=90)
    for i, ngrams in enumerate(ngrams_tuple):
        x_values = np.array(list(range(1, x_values_count + 1))) + i * bar_width
        y_values = [
            count for _, count, _ in ngrams[:x_values_count]
        ] # tuple(ngram: str, count: int, score: float)

        plt.bar(x_values, y_values, width=bar_width)

    plt.legend(legend)
    plt.show()

```

2 Właściwa analiza tekstów

2.1 1. Badane materiały

- Manuskrypt Wojnicza, w przypadku wątpliwości transliteracji (podanie wielu możliwych znaków w danym fragmencie), wybierany był pierwszy z nich
- Synteza artykułu z Wikipedii o Układzie Słonecznym, Kodeksu Karnego Republiki Słowenii, Księgi Koheleta, fragmentu 1. Listu Pawła do Koryntian

```
[ ]: print(
    "Voynich words count: {}".format(
        sum(map(len, readFormattedText("input/voynich.txt")))
    )
)
```

```

print(
    "Slovenian words count: {}".format(
        sum(map(len, readUnformattedText("input/slovenian.txt"))))
)
)

```

Voynich words count: 30134
 Slovenian words count: 30570

Uwaga: Jeżeli badana statystyka brała pod uwagę wyrazy sąsiednie, to łączenie wyrazów w krotki odbywało się wyłącznie w obrębie akapitów, ze względu na to, że takie zależności powinny najbardziej odpowiadać zależnościom semantycznym w tekście.

2.2 2. Prawo Zipfa

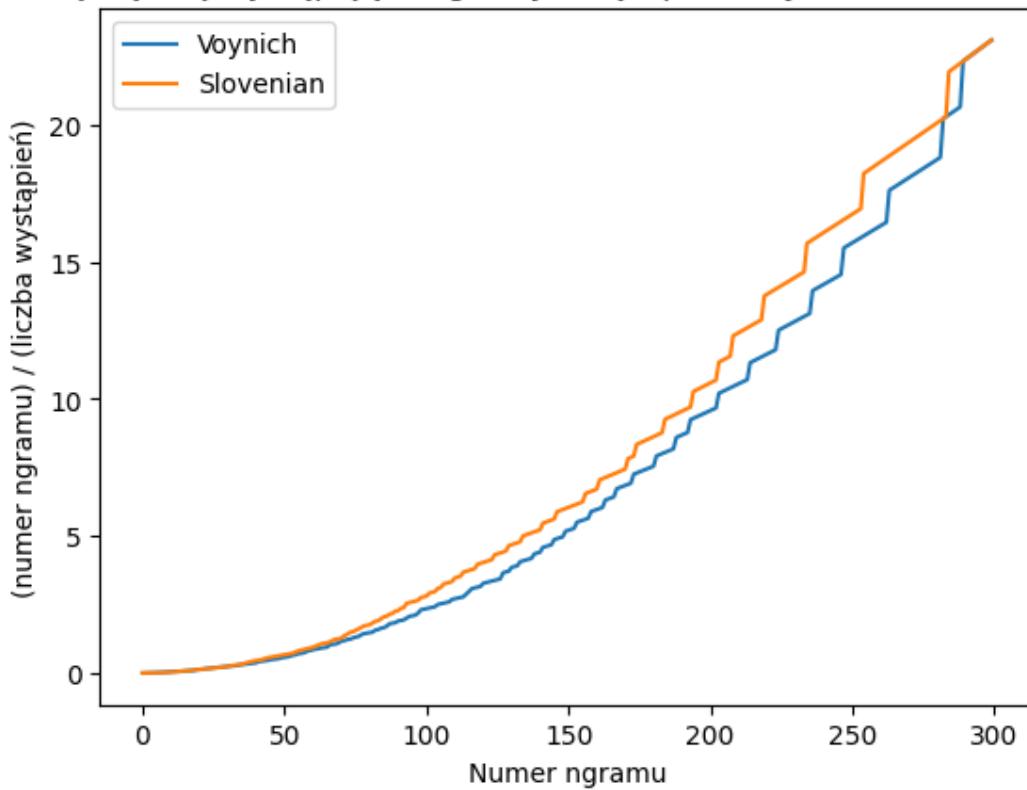
Pierwszym etapem przeprowadzonego badania, było sprawdzenie stopnia zgodności częstości występowania poszczególnych słów w obu tekstach z empirycznym prawem dotyczącym języków naturalnych, prawem Zipfa, stwierdzającym, że iloczyn częstotliwości występowania słów w danym tekście i ich rangi w rankingu częstotliwości jest stały. W tym celu, dla obu tekstów, wyznaczono częstotliwości występowania słów, a następnie posortowano je malejąco według tych wartości. Następnie, dla każdego słowa, wyznaczono jego rangę w rankingu częstotliwości. W kolejnym kroku, dla każdego słowa, wyznaczono iloczyn jego rangi i częstotliwości występowania. Gdyby prawo Zipfa było idealnie spełnione, obliczona funkcja powinna być stała. Otrzymane wyniki dla 300 najczęściej występujących słów przedstawia następujący wykres:

```

[ ]: plt.title("Najczęściej występujące ngramy w rękopisie Voynicha i słoweńskim")
plt.plot(
    [
        score
        for _, _, score in getSortedNGramsAndCalculateScores(
            countNGrams(readFormattedText("input/voynich.txt"), 1, 2)
        )[:300]
    ],
)
plt.plot(
    [
        score
        for _, _, score in getSortedNGramsAndCalculateScores(
            countNGrams(readUnformattedText("input/slovenian.txt"), 1, 2)
        )[:300]
    ]
)
plt.xlabel("Numer ngramu")
plt.ylabel("(numer ngramu) / (liczba wystąpień)")
plt.legend(["Voynich", "Slovenian"])
plt.show()

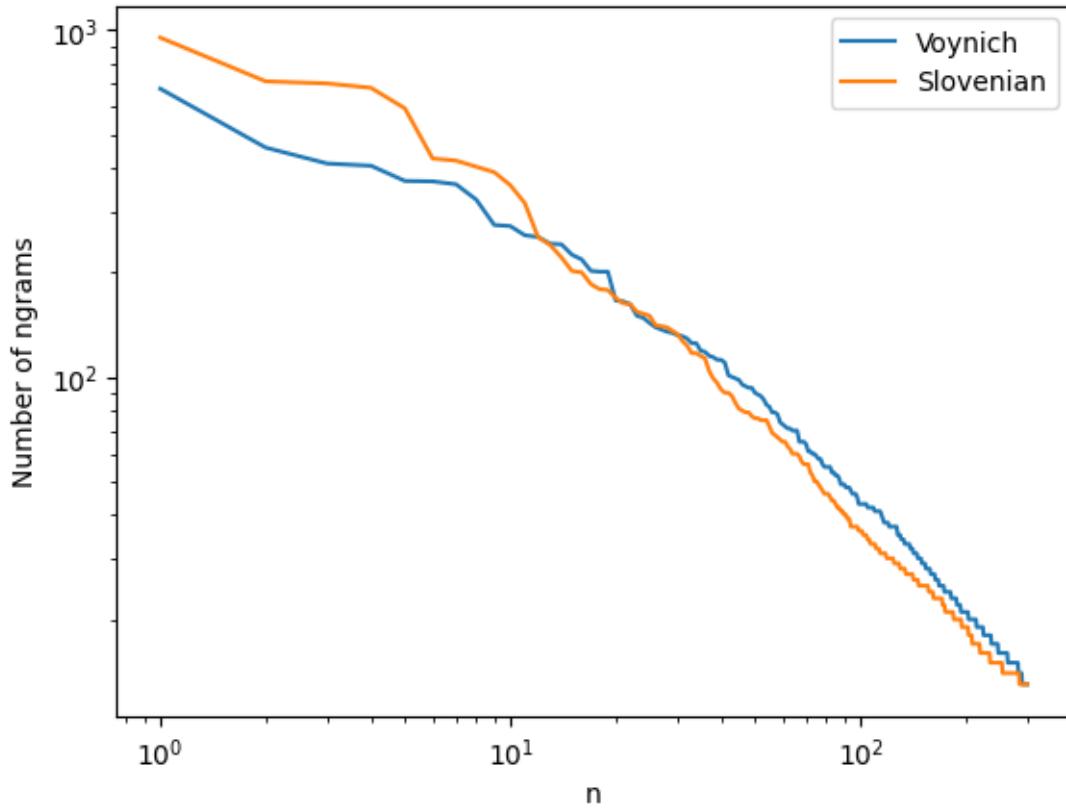
```

Najczęściej występujące ngramy w rękopisie Voynicha i słoweńskim



Stworzono również wykres ilości n-gramów w zależności od n w skali logarytmicznej dla najczęściej występujących 300 n-gramów. Zakładającąc zgodność z prawem Zipfa, otrzymane wyniki powinny być zbliżone do wykresu funkcji linowej. Otrzymane wyniki przedstawia poniższy wykres:

```
[ ]: plotNGramsCountsForEachNUsingLogarithmicScale(
    [
        getSortedNGramsAndCalculateScores(
            countNGrams(readFormattedText("input/voynich.txt"), 1, 2)
        )[:300],
        getSortedNGramsAndCalculateScores(
            countNGrams(readUnformattedText("input/slovenian.txt"), 1, 2)
        )[:300],
    ],
    ["Voynich", "Slovenian"],
)
```



Kolejnym etapem było dopasowanie hiperbole do wykresów zależności częstości słów w zależności od ich rangi. W tym celu, dla każdego tekstu, wyznaczono współczynnik a funkcji $f(x) = a/x$, które najlepiej dopasowują się do wykresów. Ograniczono się do wyrazów, które wystąpiły co najmniej dwa razy w tekście. Otrzymano następujące wartości:

```
[ ]: hyperbola_voynich = fitHyperbolaToNgramsCounts(
    getSortedNGramsAndCalculateScores(
        countNGrams(readFormattedText("input/voynich.txt"), 1, 2)
    )
)
hyperbola_slovenian = fitHyperbolaToNgramsCounts(
    getSortedNGramsAndCalculateScores(
        countNGrams(readUnformattedText("input/slovenian.txt"), 1, 2)
    )
)

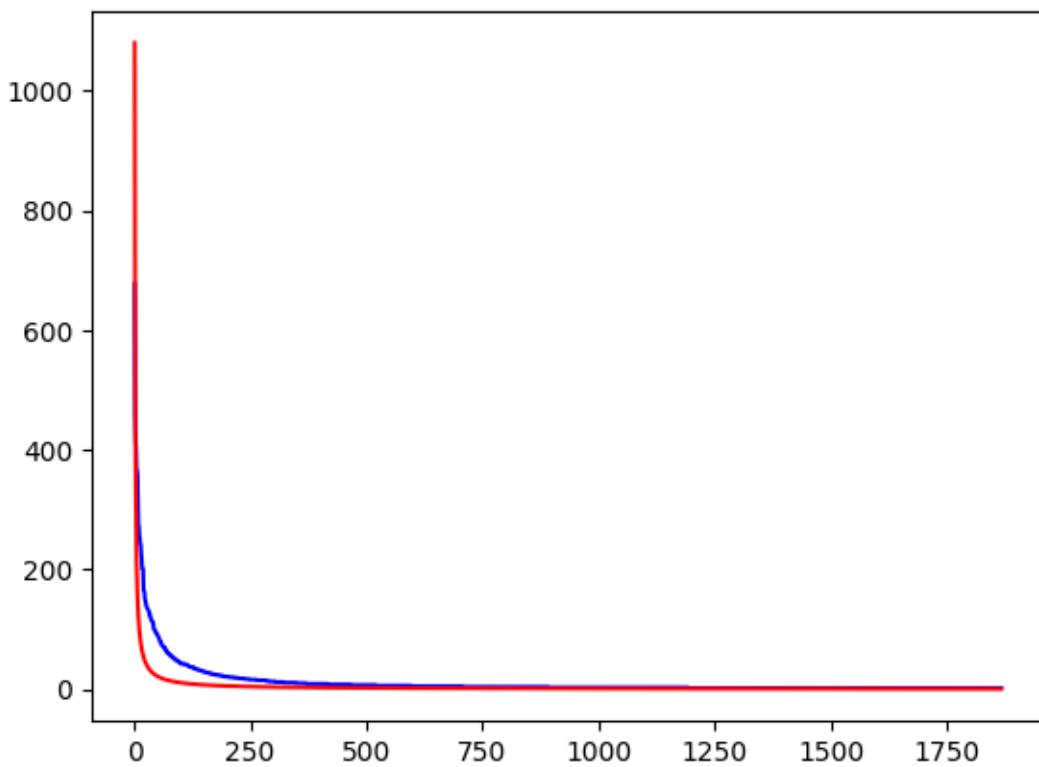
print(hyperbola_voynich, hyperbola_slovenian)
```

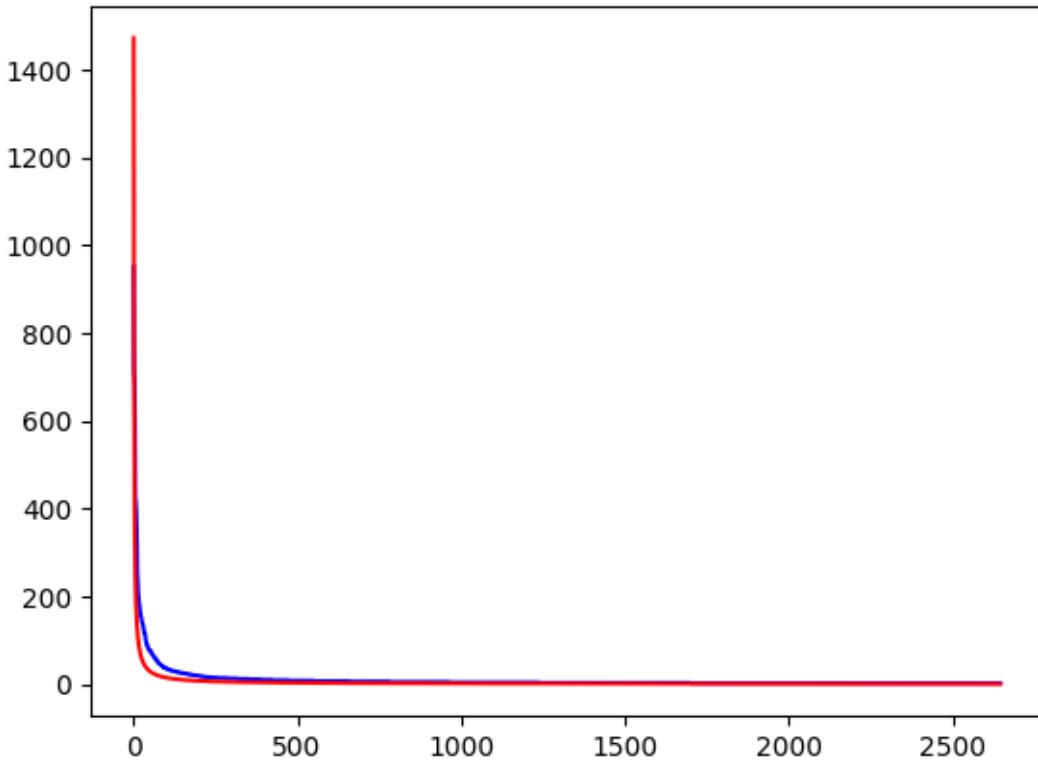
1079.4423991197284 1471.6500834013275

Wartości współczynników obu funkcji okazały się być dość oddalone od siebie. Rysując wykresy funkcji $f(x) = a/x$ dla obu tekstów, można zauważyć, że funkcja dla tekstu w języku słoweńskim

lepiej dopasowuje się do wykresu:

```
[ ]: plotNGramsCounts(
    getSortedNGramsAndCalculateScores(
        countNGrams(readFormattedText("input/voynich.txt"), 1, 2)
    )
)
plotNGramsCounts(
    getSortedNGramsAndCalculateScores(
        countNGrams(readUnformattedText("input/slovenian.txt"), 1, 2)
    )
)
```





2.2.1 Wnioski z analizy częstotliwości występowania słów:

Można zauważyć, że odpowiednie wykresy wizualizujące zależność częstości słów dla obu tekstów wykazują podobną charakterystykę. Ponadto nie można określić, że jest ona niezgodna z prawem Zipfa w wysokim stopniu.

Ostatecznie można stwierdzić, że tekst manuskryptu Wojnicza wykazuje cechy charakterystyczne dla języka naturalnego (słoweńskiego), gdy jest poddawany badaniom zgodności z prawem Zipfa. Różnice nie wydają się wystarczające, aby uznać, że tekst manuskryptu Wojnicza z całą pewnością nie jest tekstem w języku naturalnym.

2.3 3. Analiza n-gramów

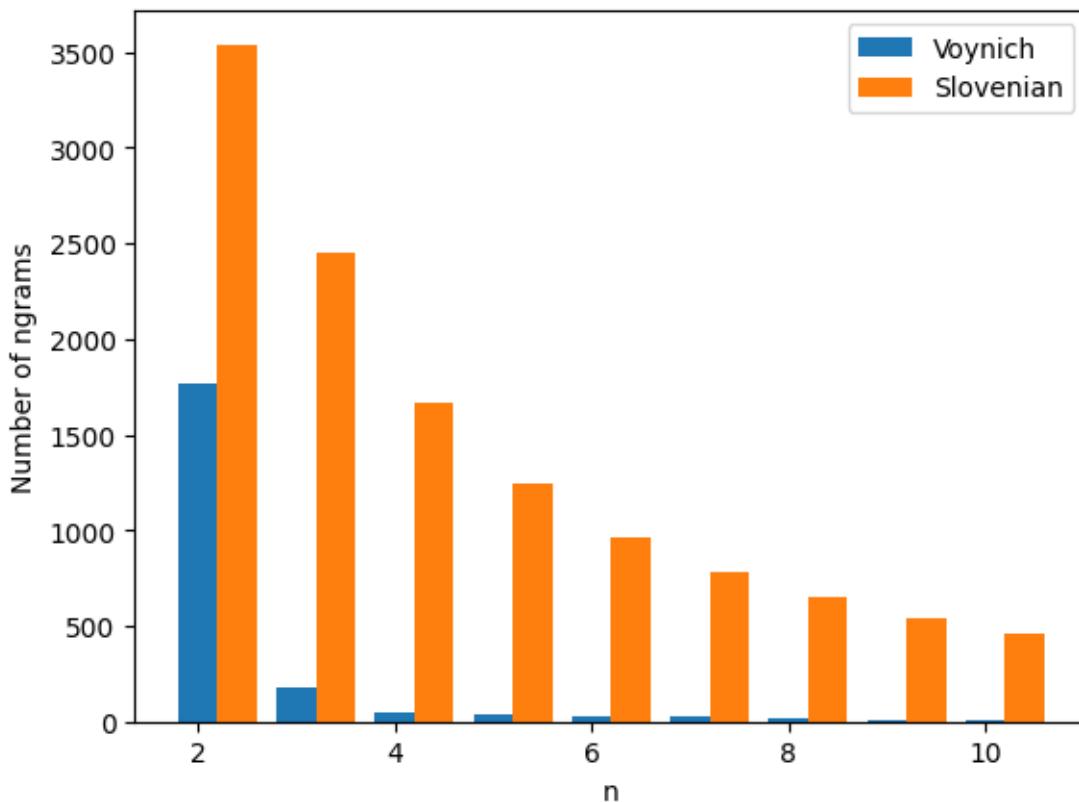
N-gramem nazywamy ciąg kolejnych n słów występujących obok siebie w danym tekście. W ramach badania, dla obu tekstów, wyznaczono ilości różnych n-gramów dla n w 1..10, takich, że dla każdego unikalnego n-gramu w tekście wystąpiły co najmniej jego dwie instancje. Wyniki przedstawia poniższy wykres:

```
[ ]: plotNGramsCountsForEachN(
  [
    countNumbersOfDifferentNGramsForEachN(
      readFormattedText("input/voynich.txt"), 2, 10, 2
    ),
  ],
)
```

```

        countNumbersOfDifferentNGramsForEachN(
            readUnformattedText("input/slovenian.txt"), 2, 10, 2
        ),
    ],
    ["Voynich", "Slovenian"],
)

```

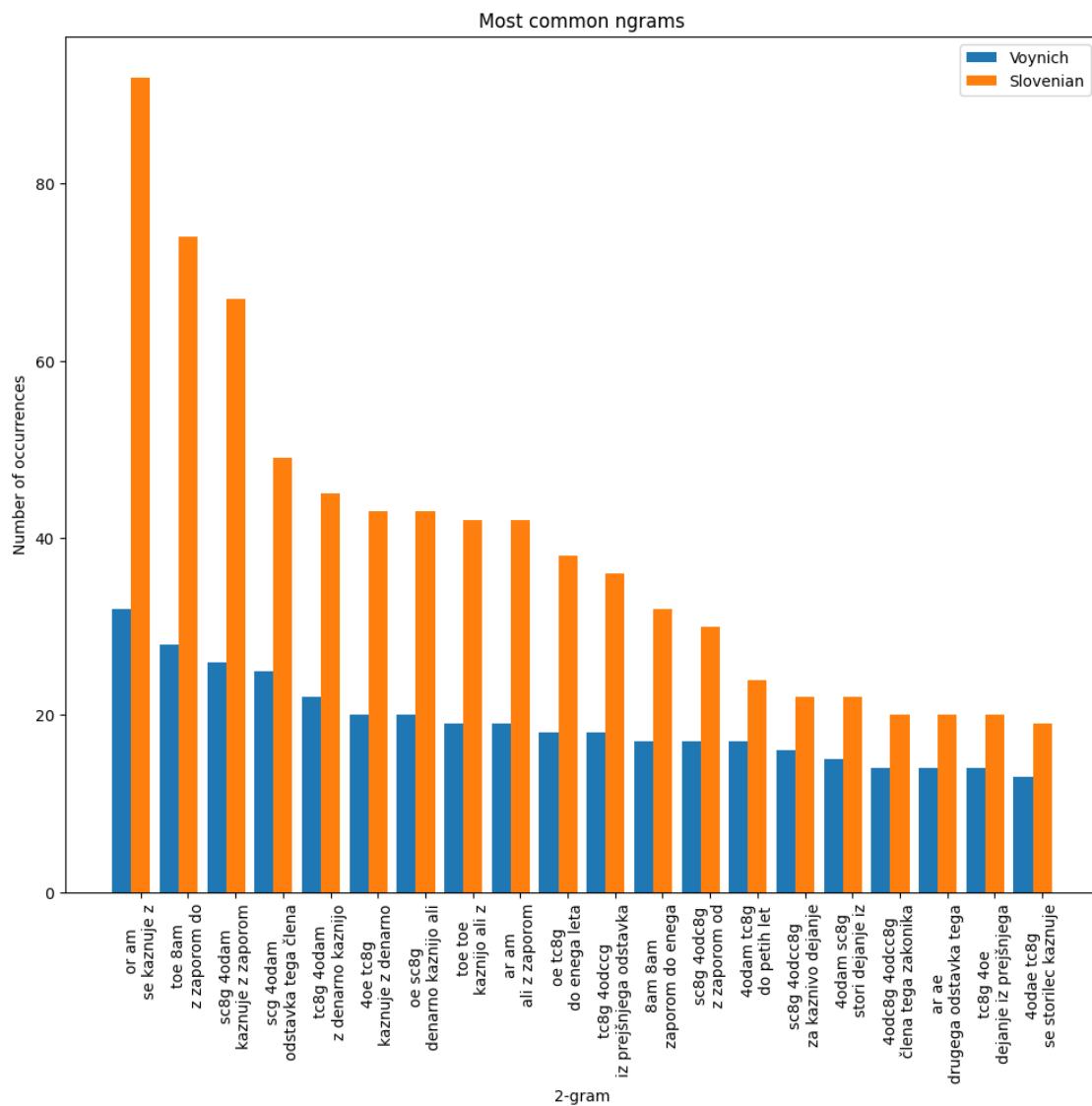


2.3.1 Wnioski z analizy ilości różnych n-gramów dla kolejnych wartości n:

Tekst manuskryptu Wojnicza posiada zdecydowanie mniejszą liczbę n-gramów w porównaniu do źródła w języku słoweńskim. Jednak znaczna ilość długich n-gramów w analizowanym tekście słoweńskim może wynikać z jego prawniczego charakteru, a nie z faktu, że jest to tekst w języku naturalnym, zawierającym znaczną liczbę często powtarzanych sformułowań, których długość może być czasami nawet większa niż 40 słów. W związku z tym, nie można na podstawie tej analizy jednoznacznie stwierdzić, że tekst manuskryptu Wojnicza nie jest tekstem w języku naturalnym, szczególnie zwróciwszy uwagę na niezerową liczbę n-gramów o długościach większych niż 3.

Przeprowadzono również analizę najczęściej występujących 2-gramów w obu tekstach. Uwzględniiono tylko 10 najczęściej występujących par wyrazów. Wyniki przedstawia poniższy wykres:

```
[ ]: plotKMostCommonNGrams(
  20,
  ["Voynich", "Slovenian"],
  getSortedNGramsAndCalculateScores(
    countNGrams(readFormattedText("input/voynich.txt"), 2, 10)
  ),
  getSortedNGramsAndCalculateScores(
    countNGrams(readUnformattedText("input/slovenian.txt"), 3, 10)
  ),
)
```



Tu również można zauważać znaczaco mniejszą liczbę wystąpień konkretnych najczęściej występujących 2-gramów w kodeksie Wojnicza w porównaniu do dokumentu napisanego w języku słoweńskim.

2.3.2 Wnioski z analizy n-gramów:

Wykonana analiza pozwala na stwierdzenie innej struktury częstości występowania najczęstszych 2-gramów dla obu tekstów. Powód wystąpienia zauważonej różnicy nie jest jednak łatwy do znalezienia. Można postawić hipotezę, że duża ilość 2-gramów w tekście w języku słoweńskim wynika ze stylu w jakim jest napisany: język prawniczy jest zazwyczaj bardzo precyzyjny i może zawierać wielokrotne powtórzenia poszczególnych sformułowań, co może prowadzić do zwiększenia liczby wystąpień poszczególnych 2-gramów.

2.4 4. Graf połączeń słów

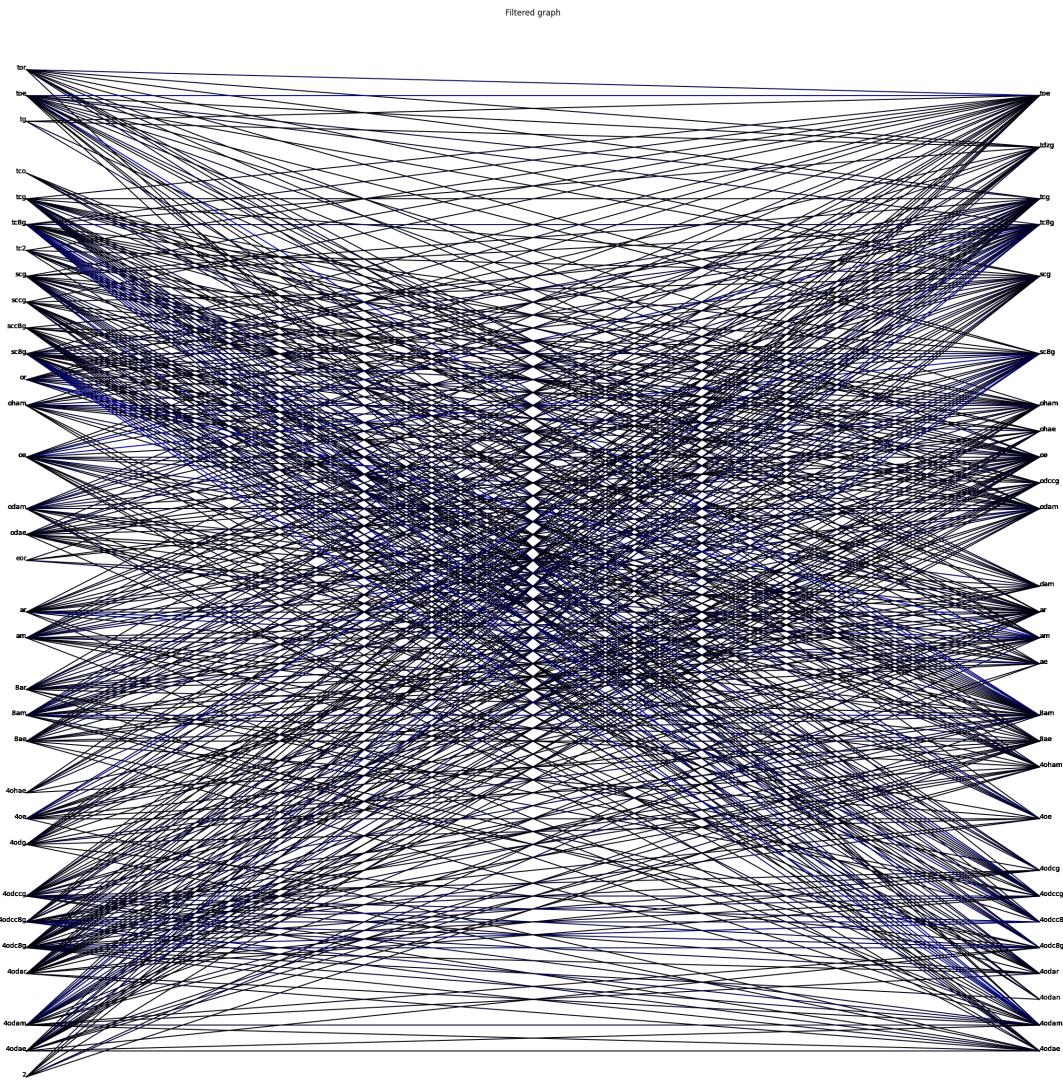
Ostatni z etapów przeprowadzonej analizy polegał na stworzeniu grafów dwudzielnych połączeń słów dla obu tekstów.

Niech $G = (V_1, V_2, E)$, gdzie V_1 i V_2 to zbiory wierzchołków, a E to zbiór krawędzi. Jeżeli w jednym z paragrafów tekstu występuje 2-gram "u v" składający się ze słów "u" oraz "v", to w zbiorze V_1 istnieje wierzchołek odpowiadający słowu "u", w zbiorze V_2 wierzchołek odpowiadający słowu "v", a w zbiorze E istnieje krawędź z wierzchołka odpowiadającego słowu "u" do wierzchołka odpowiadającego słowu "v". Wagi krawędzi odpowiadają ilości wystąpień danego 2-gramu w tekście. W przedstawionej poniżej wizualizacji grafu, wierzchołki ze strony lewej należą do zbioru V_1 , natomiast wierzchołki ze strony prawej należą do zbioru V_2 . Stopień intensywności koloru niebieskiego krawędzi odpowiada ilości wystąpień danego 2-gramu w tekście, gdzie kolor czarny oznacza najmniejszą ilość wystąpień, a kolor niebieski oznacza największą ilość wystąpień ze wszystkich 2-gramów w tekście.

W celu zwiększenia czytelności wizualizacji grafów, uwzględniono wyłącznie 2-gramy, które wystąpiły co najmniej 8 razy w tekście.

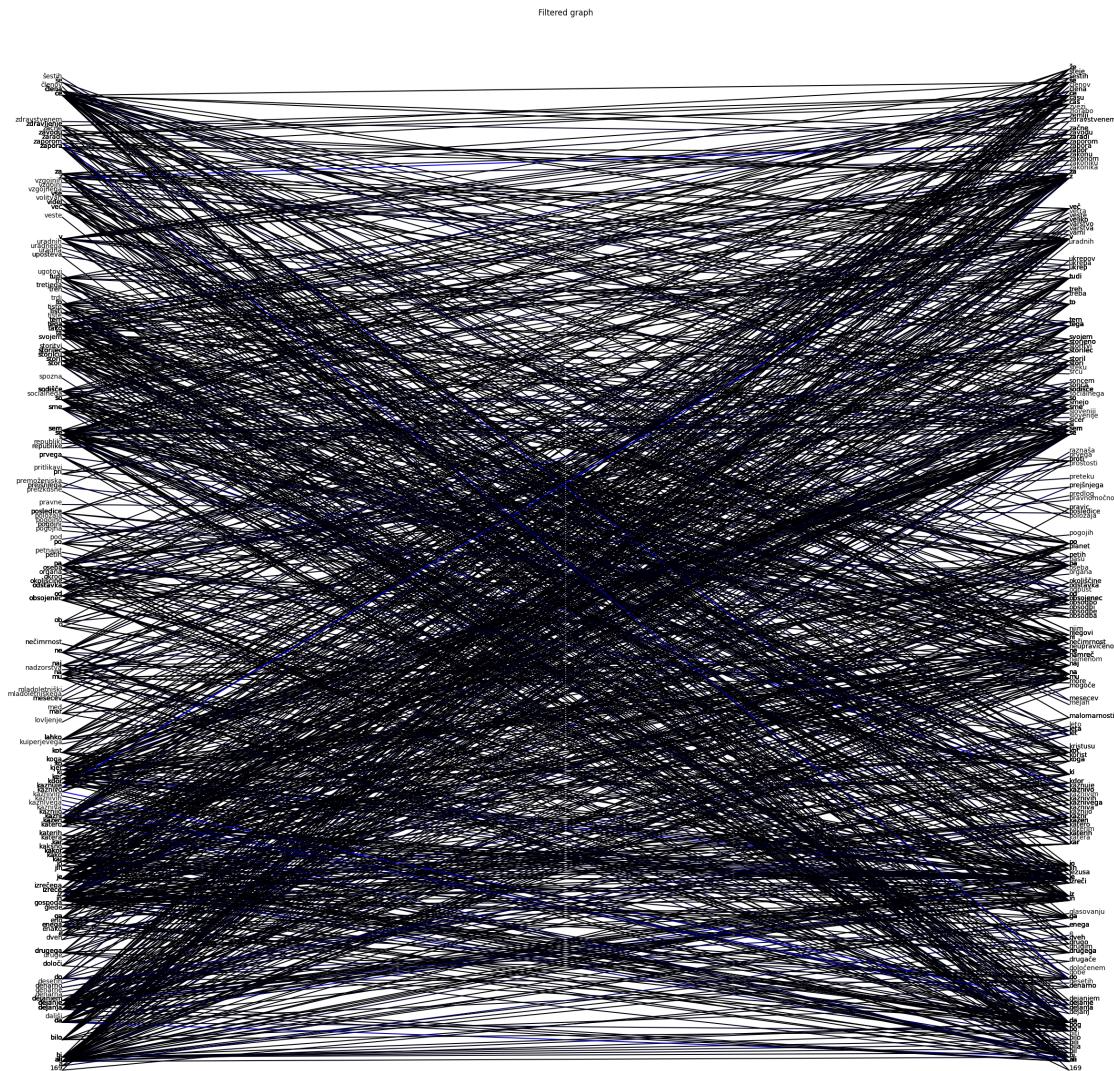
Graf stworzony dla tekstu manuskryptu Wojnicza:

```
[ ]: drawBipartiteGraph(  
    getFilteredWordsGraph(readFormattedText("input/voynich.txt"), 1, 8),  
    "Filtered graph",  
    True,  
)
```



Graf stworzony dla tekstu w języku słoweńskim:

```
[ ]: drawBipartialGraph(  
    getFilteredWordsGraph(readUnformattedText("input/slovenian.txt"), 1, 8),  
    "Filtered graph",  
    True,  
)
```



Badając oba grafy można zauważyc, iż ten, który powstał podczas badania tekstu napisanego w języku słoweńskim jest znacznie większy (mierząc liczbą wierzchołków). Ponadto jego wizualna gęstość wydaje się być znacząco większa, co najprawdopodobniej spowodowane jest dużą częstością występowania lokalnych związków wyrazowych w języku słoweńskim.

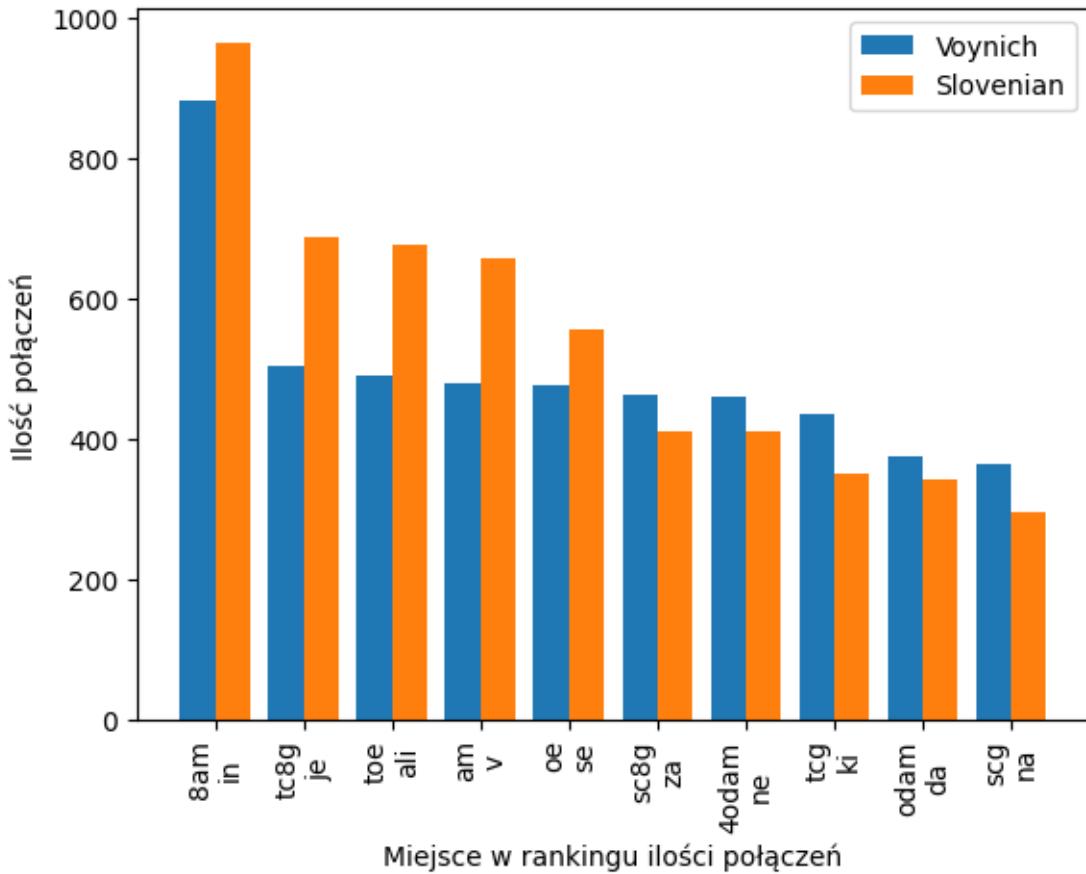
Została również przeprowadzona analiza porównawcza ilości najczęściej występujących 2-gramów w obu tekstach. Wyniki przedstawia poniższy wykres:

```
[ ]: plotNWordsWithConnectionCounts(
  [
    getNMostCommonWordsWithMostConnections(readFormattedText("input/voynich.
    ↵txt"))[
      :10
    ],
  ])
```

```

        getNMostCommonWordsWithMostConnections(
            readUnformattedText("input/slovenian.txt")
        )[:10],
    ],
    ["Voynich", "Slovenian"],
)

```



Wykres przedstawia dość podobną charakterystykę ilościową największych stopni wierzchołków grafu dla obu tekstów. Można zauważyć, że w przypadku tekstu Wojnicza, największy stopień wierzchołka, powiązany z wyrażeniem “8am”, jest znacznie większy niż stopień odpowiadający drugiemu w kolejności wyrazowi “tc8g”. Podobna charakterystyka nie występuje jednak w przypadku tekstu w języku słoweńskim, gdzie największy stopień wierzchołka, powiązany z wyrażeniem “ali”, jest tylko nieznacznie większy od stopnia odpowiadającego drugiemu w kolejności wyrazowi “in”. W przypadku kolejnych wyrazów, różnice w ilości wystąpień są już znacznie mniejsze dla manuskryptu Wojnicza, natomiast dla tekstu w języku słoweńskim, stopnie wierzchołków spadają znacznie szybciej (dla 10 największych stopni wierzchołków).

2.5 4. Wnioski

Na bazie przeprowadzonych analiz, można stwierdzić, że manuskrypt Wojnicza wykazuje niektóre cechy podobne do języka słoweńskiego. Jednakże, nie można jednoznacznie określić natury tego tekstu, bowiem pewne metryki wydają się znaczaco odbiegać względem wartości obliczonych dla słoweńskich odpowiedników. Niestety, odnalezienie przyczyny tego zjawiska nie jest możliwe wykorzystując wyłącznie przedstawione wyżej metody. Analiza nie pozwala na stwierdzenie czy kodeks Wojnicza jest zapisany tekstem w języku naturalnym. W tym celu konieczne jest przeprowadzenie kolejnych badań.