Петрозаводский государственный университет
Институт математики и информационных технологий
Кафедра Информатики и математического обеспечения

Отчет о прохождении производственной практики

Генерация текста с использованием марковской модели и статистики префиксов

Выполнил студент 2 курса группы 22207:
Гордеев Никита Владиславович

Направление подготовки бакалавриата:
09.03.04 Программная инженерия

Место прохождения практики:
Институт математики и информационных технологий

Период прохождения практики:
29.05.22-09.06.23

Руководитель:
Богоявленская Ольга Юрьевна, к.т.н, доцент

Оценка:

_____
оценка

Дата:

_____
дата

Петрозаводск — 2023

Содержание

Введение

Актуальность исследования.

Цепи Маркова актуальны в математике, экономике, биологии, физике, компьютерных науках и медицине. Они используются для анализа временных процессов и прогнозирования состояний системы. В экономике определяются оптимальные стратегии и планирование бизнеса. В медицине моделируют заболевания, оценивают эффективность лечения. В генетике изучают эволюцию генов. В компьютерных науках разрабатывают алгоритмы машинного обучения и распознавания образов. Цепи Маркова широко применяются в анализе сложных процессов и предсказании будущих событий.

Цель данной работы: Генерация текста с использованием марковской модели и статистики префиксов.

Задачи данной работы:

1. Изучение теории, посвященной цепям Маркова;
2. Разработка программного кода для реализации алгоритма генерации текстов.

Методы, способы достижения поставленных целей и задач:

1. Анализ научной литературы и интернет-источников;
2. Проведения экспериментов, анализ их результатов.

1 Изучение литературы

1.1 Цепи Маркова

Цепи Маркова предоставляют уникальный метод генерации случайных текстов на основе статистической модели, оценивающей частоты букв, слов и словосочетаний в референсном тексте. Основная идея состоит в том, что частота различных словосочетаний может быть использована для создания текста, обладающего схожими статистическими свойствами.

Метод основан на использовании перекрывающихся словосочетаний исходного текста для определения вероятности следующего слова-суффикса для данного префикса из двух слов. Вначале алгоритм выбирает два первых слова текста и выводит их. Затем в цикле для текущего префикса случайно выбирается следующий суффикс на основе статистики исходного текста.

Следующий пример демонстрирует результаты генерации случайного текста с использованием предложенного подхода.

> *Show your flowcharts and tables and be mystified. Show your flowcharts will be obvious.*

Такой текст более структурирован и осмыслен по сравнению с методами, где слова или буквы генерируются независимо друг от друга или на основе их абсолютной частоты в алфавите.

2 Разработка программы

Роли в команде

Наш сегмент проекта основывается на результатах труда коллег. Для успешной реализации проекта, задачи были распределены между членами команды.

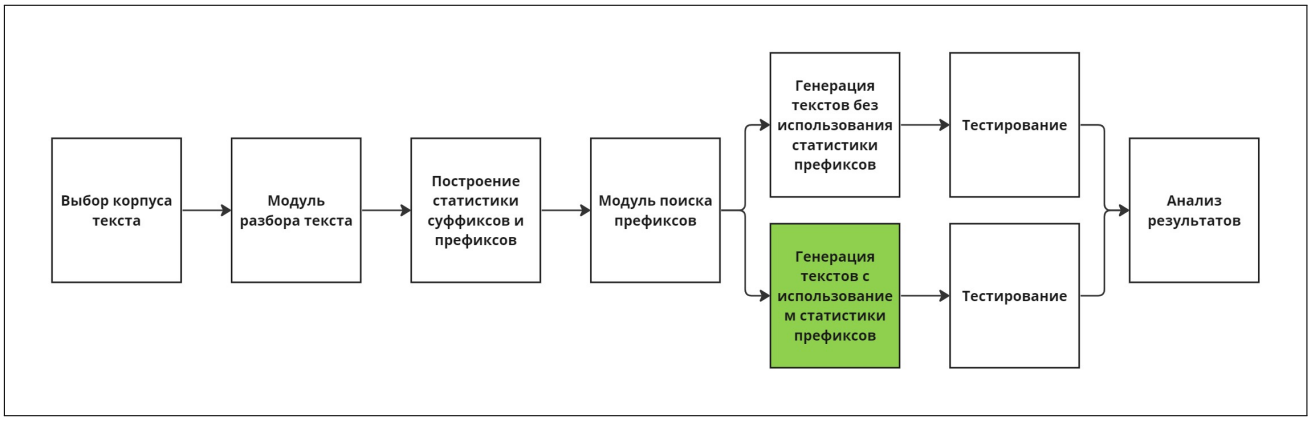


Рис. 1 – Сегменты проекта

Укпере Вильямс	Гордеев Никита	Данило Пётр
1) Реализация базового решения программы генерации случайного текста с использованием алгоритма цепи Маркова	1) Добавление функций для проверки грамматической правильности и улучшения связности сгенерированного текста 2) Разработка пользовательского интерфейса	1) Реализация решения с использованием рекуррентной нейронной сети для генерации текста

Таблица 1 – Распределение задач

2.1 Функция проверки грамматики и улучшения структуры текста

Для проверки грамматики и улучшения структуры текста можно использовать API LanguageTool[4]. Это библиотека с открытым исходным кодом, предназначенная для проверки грамматической правильности текста.

Основные возможности API LanguageTool:

- Поддержка множества языков: LanguageTool поддерживает более 20 языков, включая английский, русский, французский, немецкий и испанский.
- Несколько типов проверки: LanguageTool может проверять грамматику, орфографию, стиль и другие аспекты текста.
- Расширяемость: пользователи могут создавать собственные правила проверки и добавлять новые функции.

Чтобы начать использовать LanguageTool API, необходимо сначала установить его:

pip install language_tool_python

Теперь добавим функцию для обработки и коррекции сгенерированного текста. Она принимает текст на вход и пытается исправить его, используя "en-US" (американский английский) словарь LanguageTool. После этого она возвращает исправленный текст:

```
import language_tool_python

def grammar_and_style_check(text):
    tool = language_tool_python.LanguageTool('en-US')
    matches = tool.check(text)
    corrected_text = language_tool_python.utils.correct(text, matches)
    return corrected_text
```

Чтобы применить эту функцию к сгенерированному тексту, вызовем её, прежде чем сохранить текст в файл:

```
...
generated_text = generate_text(prefix_stats, initial_prefix, text_length)
corrected_text = grammar_and_style_check(generated_text)

with open(answer_filepath, 'w') as f:
    f.write(corrected_text)
...
```

LanguageTool не гарантирует идеальную грамматику или стиль в сгенерированном тексте, но помогает улучшить его качество.

Полный текст программы находится в Приложении .1.

2.2 Пользовательский интерфейс

Будем создавать графический интерфейс, в котором по нажатию на кнопку "генерировать" новый сгенерированный текст сохраняется в файл answer.txt и одновременно этот же новый тексты выводится в графическом интерфейсе в многострочном окне.

Для создания графического интерфейса, будем использовать библиотеку Tkinter. В ней есть виджеты для создания кнопок и многострочных окон. Выполним следующие шаги:

Импоруем библиотеку Tkinter:

```
import tkinter as tk
```

Создадим главное окно:

```
root = tk.Tk()
```

Создадим многострочное текстовое поле:

```
answer_text = tk.Text(root, wrap="word", font=("Consolas", 10))
```

Создадим полосу прокрутки:

```
scrollbar = tk.Scrollbar(root, command=answer_text.yview)
```

Назначим полосу прокрутки текстовому полю:

```
answer_text.config(yscrollcommand=scrollbar.set)
```

Создадим кнопку для генерации и сохранения текста:

```
generate_button = tk.Button(root, text="Generate", command=generate_and_save_text)
```

Создадим функцию для генерации и сохранения текста в файл и отображения сгенерированного текста на экране:

```
def generate_and_save_text():
        corpus = load_text_from_file(source_filepath)

        cleaned_text = preprocess_text(corpus)
        prefix_stats = build_prefix_stats(cleaned_text, prefix_length)
        initial_prefix_lc = initial_prefix.lower()
        generated_text = generate_text(prefix_stats, initial_prefix_lc, text_length)
        corrected_text = grammar_and_style_check(generated_text)

        with open(answer_filepath, 'w') as f:
            f.write(corrected_text)

        answer_text.delete("1.0", tk.END)
        answer_text.insert("1.0", generated_text)
```

Сделаем поле ввода параметров генерации текста и передачи значений в соответствующие переменные:

```
def update_parameters():
        global prefix_length, initial_prefix, text_length, source_filepath

        prefix_length = int(prefix_length_entry.get())
        initial_prefix = initial_prefix_entry.get()
```

```
        text_length = int(text_length_entry.get())
        source_filepath = source_filepath_entry.get()

        prefix_length_label.config(text=f"Prefix Length: {prefix_length}")
        initial_prefix_label.config(text=f"Initial Prefix: {initial_prefix}")
        text_length_label.config(text=f"Text Length: {text_length}")
        source_filepath_label.config(text=f"Source File: {source_filepath}")

prefix_length_entry = tk.Entry(parameters_frame, width=5)
prefix_length_entry.insert(tk.END, str(prefix_length))
prefix_length_entry.pack(anchor=tk.W, padx=5, pady=5)
prefix_length_entry.bind("<FocusOut>", lambda event: update_parameters())

initial_prefix_entry = tk.Entry(parameters_frame, width=20)
initial_prefix_entry.insert(tk.END, initial_prefix)
initial_prefix_entry.pack(anchor=tk.W, padx=5, pady=5)
initial_prefix_entry.bind("<FocusOut>", lambda event: update_parameters())

text_length_entry = tk.Entry(parameters_frame, width=5)
text_length_entry.insert(tk.END, str(text_length))
text_length_entry.pack(anchor=tk.W, padx=5, pady=5)
text_length_entry.bind("<FocusOut>", lambda event: update_parameters())

source_filepath_entry = tk.Entry(parameters_frame, width=20)
source_filepath_entry.insert(tk.END, source_filepath)
source_filepath_entry.pack(anchor=tk.W, padx=5, pady=5)
source_filepath_entry.bind("<FocusOut>", lambda event: update_parameters())
```

Разместим все элементы в нужном порядке:

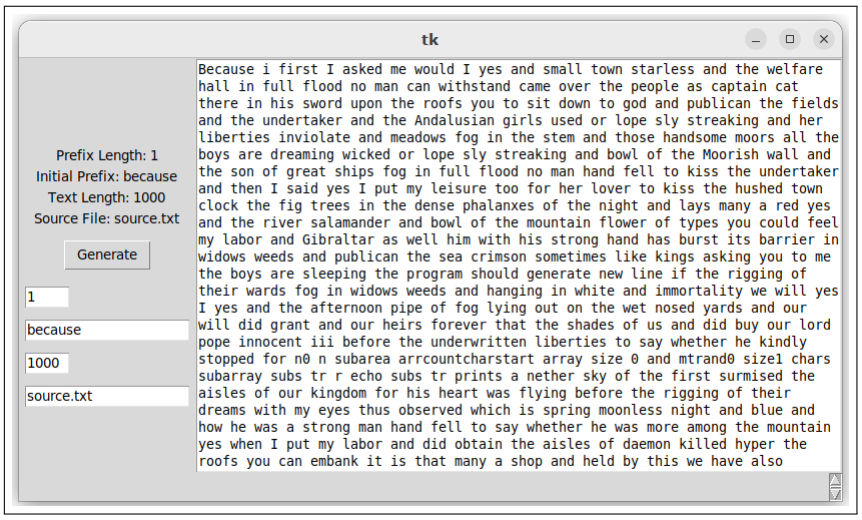
```
answer_text.pack(expand=True, fill="both")
scrollbar.pack(side=tk.RIGHT, fill=tk.Y)
generate_button.pack(pady=10)
```

Запустим бесконечное повторение программы:

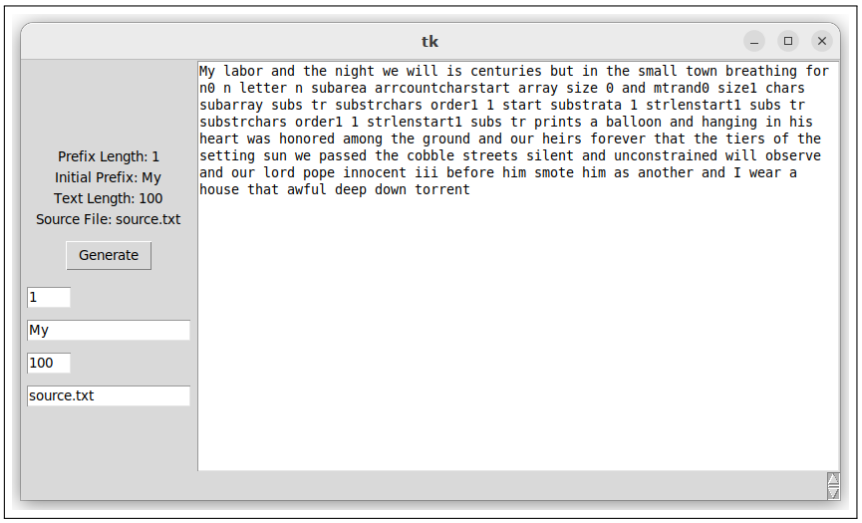
```
root.mainloop()
```

В итоге, получается программа, которая генерирует текст на основе заданных параметров, отображает его в многострочном текстовом поле и сохраняет его в файл при нажатии на кнопку "Generate". Полный текст программы находится в Приложении .2.

(a) Приветственный экран



(b) Генерация с параметрами: "1", "because", "1000", "source.txt"



(c) Генерация с параметрами: "1", "my", "100", "source.txt"

Рис. 2 – Экраны приложения

3 Тестирование программы

3.1 Время выполнения

Для того, чтобы программа печатала время выполнения, воспользуемся модулем time. Добавим в код строки:

```
import time
start_time = time.time()

# тут идут все основные функции и вызовы

end_time = time.time()
print(end_time - start_time)
```

Таким образом, при завершении программы в консоль будет выведено время, затраченное на ее выполнение.

Запуск	Программа без функции проверки грамматики	Программа с функцией проверки грамматики
1	0.0009577274322509766	7.03693962097168
2	0.003083467483520508	7.452900648117065
3	0.0038442611694335938	8.426994562149048
4	0.004538059234619141	10.353859424591064
5	0.0029518604278564453	10.223316669464111
6	0.0027184486389160156	10.636064767837524
7	0.0022034645080566406	12.667102098464966
8	0.0026845932006835938	13.87061095237732
9	0.0019922256469726562	12.87858390808105
10	0.0023589134216308594	14.952302932739258

Таблица 2 – Время затраченное на ее выполнение программ

Несмотря на то, что функция проверки грамматики замедляет работу программы, использование такой функции позволяет генерировать более грамотный текст, без ошибок и несогласования в частях речи. Это важно, если требуется использовать результаты программы в каких-то серьезных целях, например, в автоматическом создании текстов для сайтов, каталогов или других документов. Поэтому, если важна грамотность генерируемого текста, необходимо принимать в расчет возможное замедление работы программы и использовать функцию проверки грамматики. Кроме того, можно искать способы для

оптимизации работы программы, например, уменьшения количества проверяемых вариантов, чтобы достичь более быстрой работы, не ухудшая качество генерируемого текста.

Заключение

В процессе производственной практики была создана программу для генерации текста на естественном языке с использованием цепей Маркова. Я изучил основные принципы работы с такой цепью, а также написал код с помощью языка Python для реализации генерации текста.

Список литературы

1. Керниган Б., Пайк Р. Практика программирования. - 8-е изд. - М., СПб., Киев: Издательский дом "Вильямс 2015. - 288 с.

2. Использование Марковских цепей при решении различных прикладных задач // Научный журнал Фундаментальные исследования URL: https://fundamental-research.ru/ru/article/view?id=1686

3. Краткое введение в цепи Маркова // Хабр URL: https://habr.com/ru/articles/455762/

4. language-tool-python 2.7.1 // PyPI URL: https://pypi.org/project/language-tool-python/

5. tkinter — Python interface to Tcl/Tk // Python URL: https://docs.python.org/3/library/tkinter.html

6. Разметка виджетов в Tkinter — pack, grid и place // Python 3 URL: https://python-scripts.com/tkinter-layout-example

7. Изучаем lambda-функции в Python // Python 3 URL: https://python-scripts.com/lambda

8. time — Time access and conversions // Python URL: https://docs.python.org/3/library/time.html

Приложение

.1 Код программы с добавленными функциями грамматической правильности и связности сгенерированного текста

```
# Импортируем необходимые библиотеки
import string
import random
import language_tool_python
import time

# Функция проверки текста на грамматику и стиль
def grammar_and_style_check(text):
    # Создаем объект класса LanguageTool на английском языке
    tool = language_tool_python.LanguageTool('en-US')
    # Получаем список найденных ошибок и предлагаемых исправлений
    matches = tool.check(text)
    # Исправляем найденные ошибки в тексте
    corrected_text = language_tool_python.utils.correct(text, matches)
    return corrected_text

# Загрузка текста из файла
def load_text_from_file(filepath):
    with open(filepath) as f:
        corpus = f.read()
    return corpus

# Предварительная обработка текста
def preprocess_text(text):
    # Приводим текст к нижнему регистру
    cleaned_text = text.lower()
    # Удаляем всю пунктуацию из текста
    cleaned_text = cleaned_text.translate(str.maketrans("", "", string.punctuation))
    # Заменяем символ новой строки на пробел
    cleaned_text = cleaned_text.replace("\n", " ")
    # Удаляем лишние пробелы, объединяя слова в тексте
    cleaned_text = " ".join(cleaned_text.split())
    return cleaned_text

# Построение статистики по префиксам
def build_prefix_stats(text, prefix_length):
    prefix_stats = {}
    words = text.split()

    # Проходим по всем словам в тексте
    for i in range(len(words) - prefix_length):
        prefix = tuple(words[i : i + prefix_length])
        suffix = words[i + prefix_length]
        # Заполняем словарь статистикой по префиксам и соответствующим им суффиксам
```

```
        prefix_stats.setdefault(prefix, []).append(suffix)

    return prefix_stats

# Генерация текста на основе статистики по префиксам
def generate_text(prefix_stats, initial_prefix, length):
    current_prefix = tuple(initial_prefix.split())
    generated_text = list(current_prefix)

    # Генерируем текст, пока не достигнем заданной длины или не будет найдена последовательность без
     соответствующих суффиксов
    while len(generated_text) < length:
        if current_prefix not in prefix_stats or not prefix_stats[current_prefix]:
            break

        next_word = random.choice(prefix_stats[current_prefix])
        generated_text.append(next_word)
        current_prefix = tuple(generated_text[-len(current_prefix) :])

    return " ".join(generated_text)

# Основная функция программы
def main():
    # Пути к файлам с исходным и результативным текстом
    source_filepath = "source.txt"
    answer_filepath = "answer.txt"

    # Настройки для генерации текста
    prefix_length = 1
    initial_prefix = "because"
    text_length = 1000

    # Загружаем текст из файла
    corpus = load_text_from_file(source_filepath)
    # Предварительно обрабатываем текст
    cleaned_text = preprocess_text(corpus)

    # Строим статистику по префиксам
    prefix_stats = build_prefix_stats(cleaned_text, prefix_length)
    # Приводим начальный префикс к нижнему регистру
    initial_prefix = initial_prefix.lower()
    # Генерируем текст на основе полученной статистики по префиксам
    generated_text = generate_text(prefix_stats, initial_prefix, text_length)

    # Проверяем генерированный текст на грамматику и стиль
    corrected_text = grammar_and_style_check(generated_text)

    # Записываем результативный текст в файл
    with open(answer_filepath, 'w') as f:
        f.write(corrected_text)

```

```
# Запуск программы
if __name__ == "__main__":
    main()
```

.2 Код программы с пользовательским интерфейсом

```
# Импортируем необходимые библиотеки
import string
import random
import tkinter as tk
from tkinter import ttk
import language_tool_python

# Функция проверки текста на грамматику и стиль
def grammar_and_style_check(text):
    # Создаем объект класса LanguageTool на английском языке
    tool = language_tool_python.LanguageTool('en-US')
    # Получаем список найденных ошибок и предлагаемых исправлений
    matches = tool.check(text)
    # Исправляем найденные ошибки в тексте
    corrected_text = language_tool_python.utils.correct(text, matches)
    return corrected_text

# Загрузка текста из файла
def load_text_from_file(filepath):
    with open(filepath) as f:
        corpus = f.read()
    return corpus

# Предварительная обработка текста
def preprocess_text(text):
    # Приводим текст к нижнему регистру
    cleaned_text = text.lower()
    # Удаляем всю пунктуацию из текста
    cleaned_text = cleaned_text.translate(str.maketrans("", "", string.punctuation))
    # Заменяем символ новой строки на пробел
    cleaned_text = cleaned_text.replace("\n", " ")
    # Удаляем лишние пробелы, объединяя слова в тексте
    cleaned_text = " ".join(cleaned_text.split())
    return cleaned_text

# Построение статистики по префиксам
def build_prefix_stats(text, prefix_length):
    prefix_stats = {}
    words = text.split()

    # Проходим по всем словам в тексте
    for i in range(len(words) - prefix_length):
        prefix = tuple(words[i : i + prefix_length])
```

```
        suffix = words[i + prefix_length]
        # Заполняем словарь статистикой по префиксам и соответствующим им суффиксам
        prefix_stats.setdefault(prefix, []).append(suffix)

    return prefix_stats

# Генерация текста на основе статистики по префиксам
def generate_text(prefix_stats, initial_prefix, length):
    current_prefix = tuple(initial_prefix.split())
    generated_text = list(current_prefix)

    # Генерируем текст, пока не достигнем заданной длины или не будет найдена последовательность без
     соответствующих суффиксов
    while len(generated_text) < length:
        if current_prefix not in prefix_stats or not prefix_stats[current_prefix]:
            break

        next_word = random.choice(prefix_stats[current_prefix])
        generated_text.append(next_word)
        current_prefix = tuple(generated_text[-len(current_prefix) :])

    return " ".join(generated_text)

# Функция для создания графического интерфейса пользователя.
def create_gui():
    # Создаем основное окно приложения.
    root = tk.Tk()

    # Создаем фрейм для настройки параметров генерации.
    parameters_frame = tk.Frame(root)
    parameters_frame.pack(side=tk.LEFT)

    # Создаем метки для отображения текущих значений параметров.
    prefix_length_label = tk.Label(parameters_frame, text=f"Prefix Length: {prefix_length}")
    prefix_length_label.pack()

    initial_prefix_label = tk.Label(parameters_frame, text=f"Initial Prefix: {initial_prefix}")
    initial_prefix_label.pack()

    text_length_label = tk.Label(parameters_frame, text=f"Text Length: {text_length}")
    text_length_label.pack()

    source_filepath_label = tk.Label(parameters_frame, text=f"Source File: {source_filepath}")
    source_filepath_label.pack()

    # Создаем кнопку для запуска генерации текста.
    generate_button = tk.Button(parameters_frame, text="Generate")
    generate_button.pack(pady=10)

    # Создаем текстовое поле для отображения сгенерированного текста.
    answer_text = tk.Text(root, wrap="word", font=("Consolas", 10))
```

```
    answer_text.pack(expand=True, fill="both")

    # Добавляем вертикальный скроллбар для текстового поля.
    scrollbar = tk.Scrollbar(root, command=answer_text.yview)
    scrollbar.pack(side=tk.RIGHT, fill=tk.Y)

    answer_text.config(yscrollcommand=scrollbar.set)

    # Создаем виджеты для задания параметров генерации текста и добавляем команду для кнопки генерации.
    create_widgets(parameters_frame, answer_text, prefix_length_label, initial_prefix_label,
     text_length_label, source_filepath_label, generate_button)

    # Запускаем главный цикл обработки событий.
    root.mainloop()

# Функция для создания виджетов пользовательского интерфейса, которые будут использоваться для генерации
     текста
def create_widgets(parameters_frame, answer_text, prefix_length_label, initial_prefix_label,
     text_length_label, source_filepath_label, generate_button):

    # Обновление параметров генерации текста на основе значений, введенных пользователем в поля.
    def update_parameters():
        global prefix_length, initial_prefix, text_length, source_filepath

        # Получаем текущие значения параметров из соответствующих полей.
        prefix_length = int(prefix_length_entry.get())
        initial_prefix = initial_prefix_entry.get()
        text_length = int(text_length_entry.get())
        source_filepath = source_filepath_entry.get()

        # Обновляем соответствующие метки, отображающие значения параметров.
        prefix_length_label.config(text=f"Prefix Length: {prefix_length}")
        initial_prefix_label.config(text=f"Initial Prefix: {initial_prefix}")
        text_length_label.config(text=f"Text Length: {text_length}")
        source_filepath_label.config(text=f"Source File: {source_filepath}")

    # Функция для генерации и сохранения текста на основе параметров.
    def generate_and_save_text():
        # Загружаем текст из файла
        corpus = load_text_from_file(source_filepath)

        # Предварительно обрабатываем текст
        cleaned_text = preprocess_text(corpus)

        # Строим статистику по префиксам
        prefix_stats = build_prefix_stats(cleaned_text, prefix_length)

        # Приводим начальный префикс к нижнему регистру
        initial_prefix_lc = initial_prefix.lower()

        # Генерируем текст на основе полученной статистики по префиксам
```

```
        generated_text = generate_text(prefix_stats, initial_prefix_lc, text_length)

        # Проверяем генерированный текст на грамматику и стиль
        corrected_text = grammar_and_style_check(generated_text)

        # Записываем результативный текст в файл
        with open(answer_filepath, 'w') as f:
            f.write(corrected_text)

        # Отображаем сгенерированный и откорректированный текст в соответствующем поле
        answer_text.delete("1.0", tk.END)
        answer_text.insert("1.0", corrected_text)

    # Создаем поля ввода для задания параметров генерации текста
    prefix_length_entry = tk.Entry(parameters_frame, width=5)
    prefix_length_entry.insert(tk.END, str(prefix_length))
    prefix_length_entry.pack(anchor=tk.W, padx=5, pady=5)
    prefix_length_entry.bind("<FocusOut>", lambda event: update_parameters())

    initial_prefix_entry = tk.Entry(parameters_frame, width=20)
    initial_prefix_entry.insert(tk.END, initial_prefix)
    initial_prefix_entry.pack(anchor=tk.W, padx=5, pady=5)
    initial_prefix_entry.bind("<FocusOut>", lambda event: update_parameters())

    text_length_entry = tk.Entry(parameters_frame, width=5)
    text_length_entry.insert(tk.END, str(text_length))
    text_length_entry.pack(anchor=tk.W, padx=5, pady=5)
    text_length_entry.bind("<FocusOut>", lambda event: update_parameters())

    source_filepath_entry = tk.Entry(parameters_frame, width=20)
    source_filepath_entry.insert(tk.END, source_filepath)
    source_filepath_entry.pack(anchor=tk.W, padx=5, pady=5)
    source_filepath_entry.bind("<FocusOut>", lambda event: update_parameters())

    # Добавляем команду для кнопки генерации текста
    generate_button.config(command=generate_and_save_text)

# Пути к файлам с исходным и результативным текстом
source_filepath = "source.txt"
answer_filepath = "answer.txt"

# Настройки для генерации текста
prefix_length = 1
initial_prefix = "because"
text_length = 1000

# Запуск программы
if __name__ == "__main__":
    create_gui()
```