3장 스트림릿(Streamlit)으로 챗봇 구현하기

스트림릿(Streamlit)은 프런트엔드에 대한 지식 없이도 머신러닝 모델을 쉽고 빠르게 시연하고 배포할 수 있도록 돕는 오픈소스 프레임워크입니다. 스트림릿을 사용하면 파이썬 코드만으로 사용자와 상호작용하는 LLM 웹 애플리케이션을 손쉽게 만들 수 있습니다. 여기에서는 스트림릿의 핵심 콘셉트와 사용 방법을 배우고, 이를 바탕으로 사용자와 웹 공간에서 상호작용하는 제미나이 챗봇을 구현합니다.

3.1. 스트림릿 기본 사용 방법

스트림릿의 설치와 기본 사용 방법에 대해 알아보겠습니다.

3.1.1. 스트림릿 설치

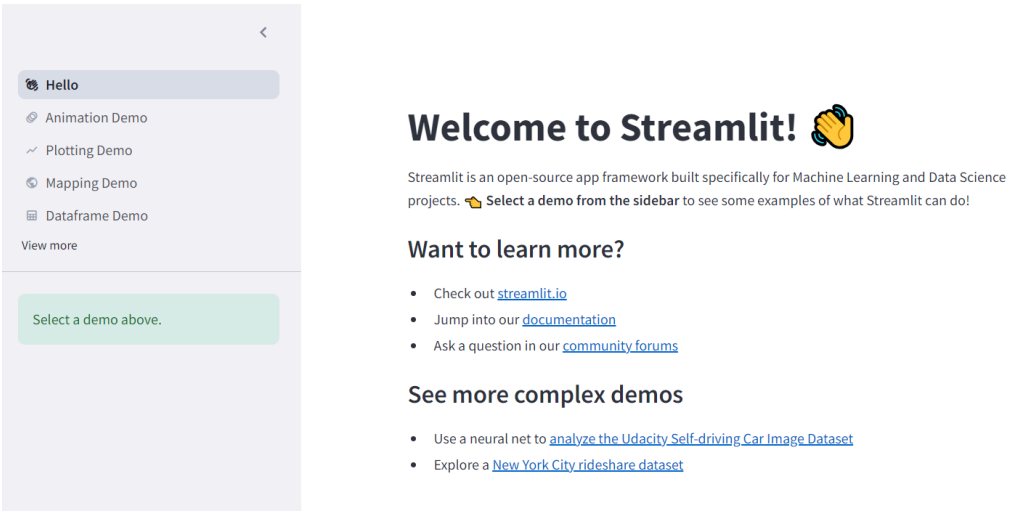
다음 명령으로 streamlit 패키지를 설치합니다.

```
>pip install streamlit
```

설치가 성공했는지 확인하기 위해 데모 프로그램인 hello를 실행합니다.

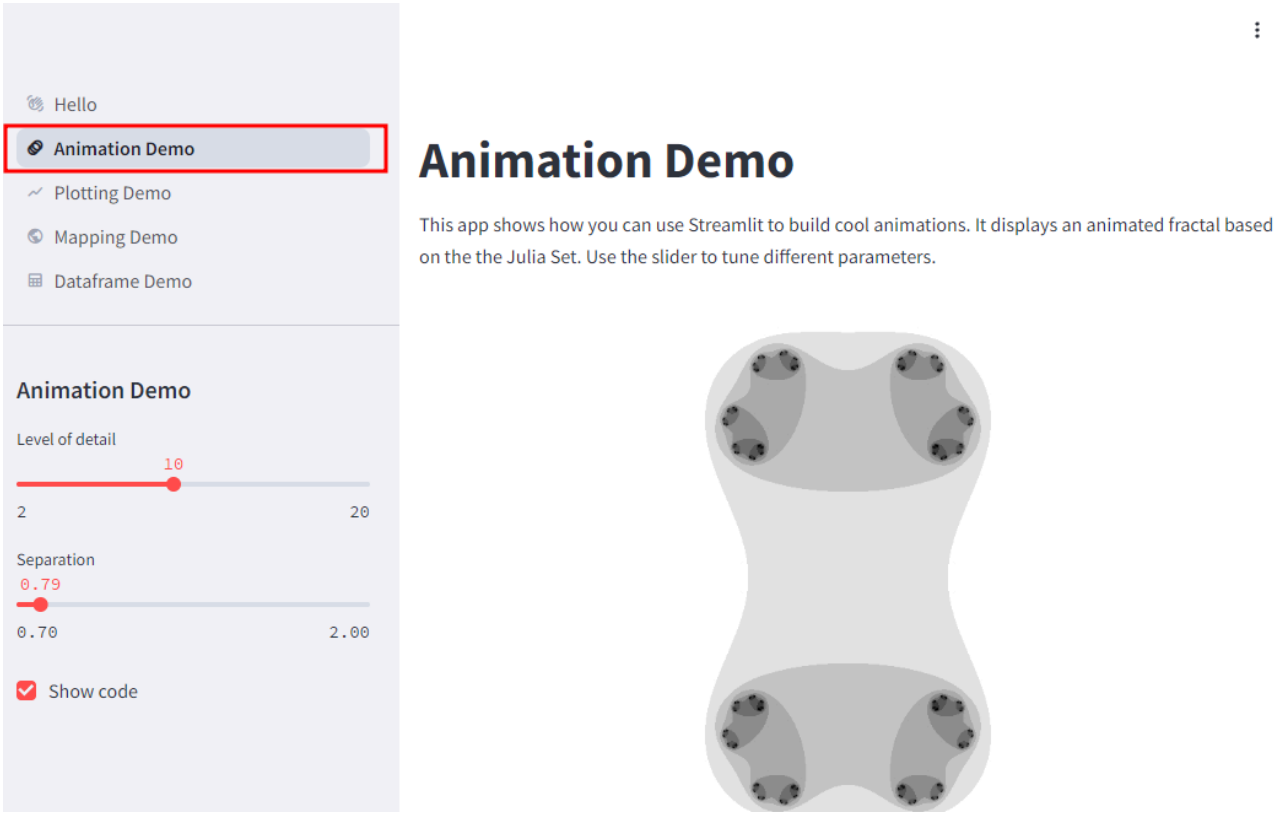
```
>streamlit hello

  Welcome to Streamlit. Check out our demo in your browser.

  Local URL: http://localhost:8502
  Network URL: http://192.168.200.67:8502

  Ready to create your own Python apps super quickly?
  Head over to https://docs.streamlit.io

  May you create awesome apps!
```

콘솔에 출력된 http://localhost:8501로 접속하여 아래와 같은 화면이 출력되면 성공적으로 설치된 겁니다.

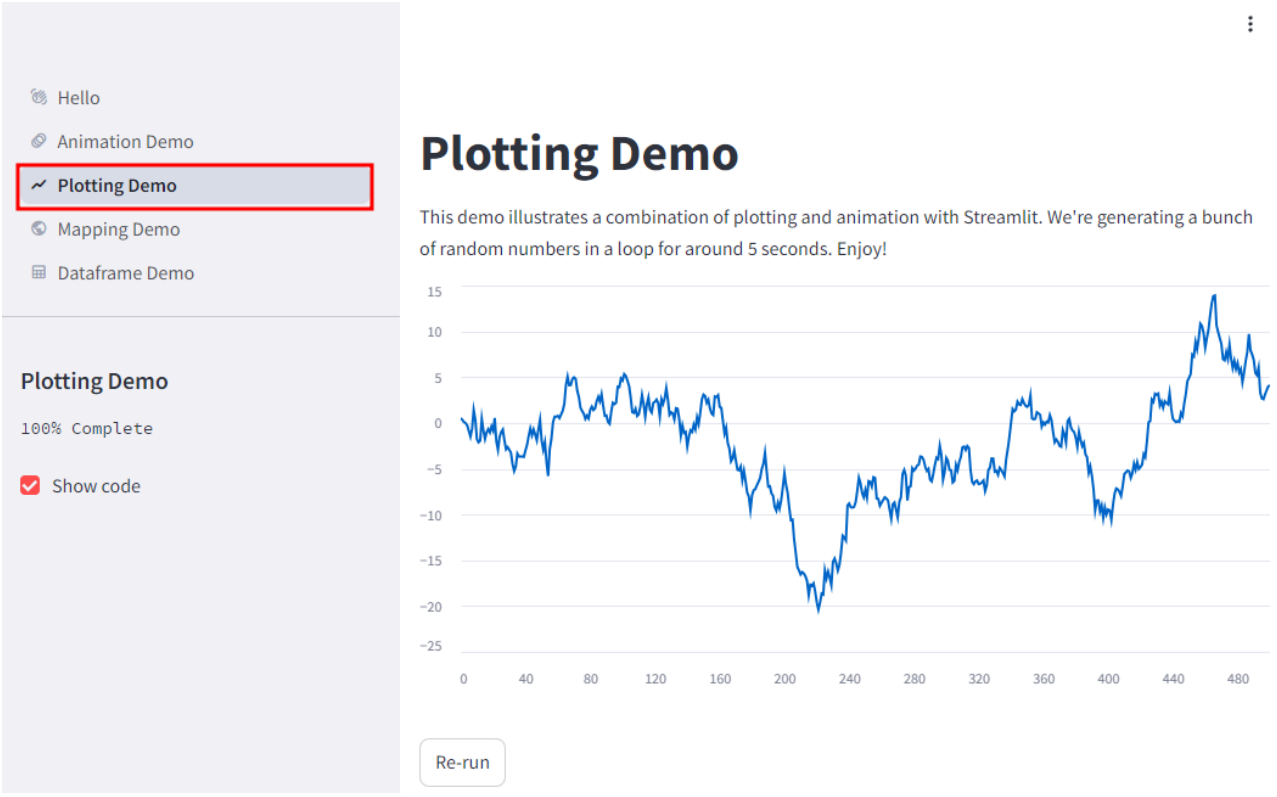


왼쪽 메뉴바에서 여러 가지 데모를 확인할 수 있습니다.

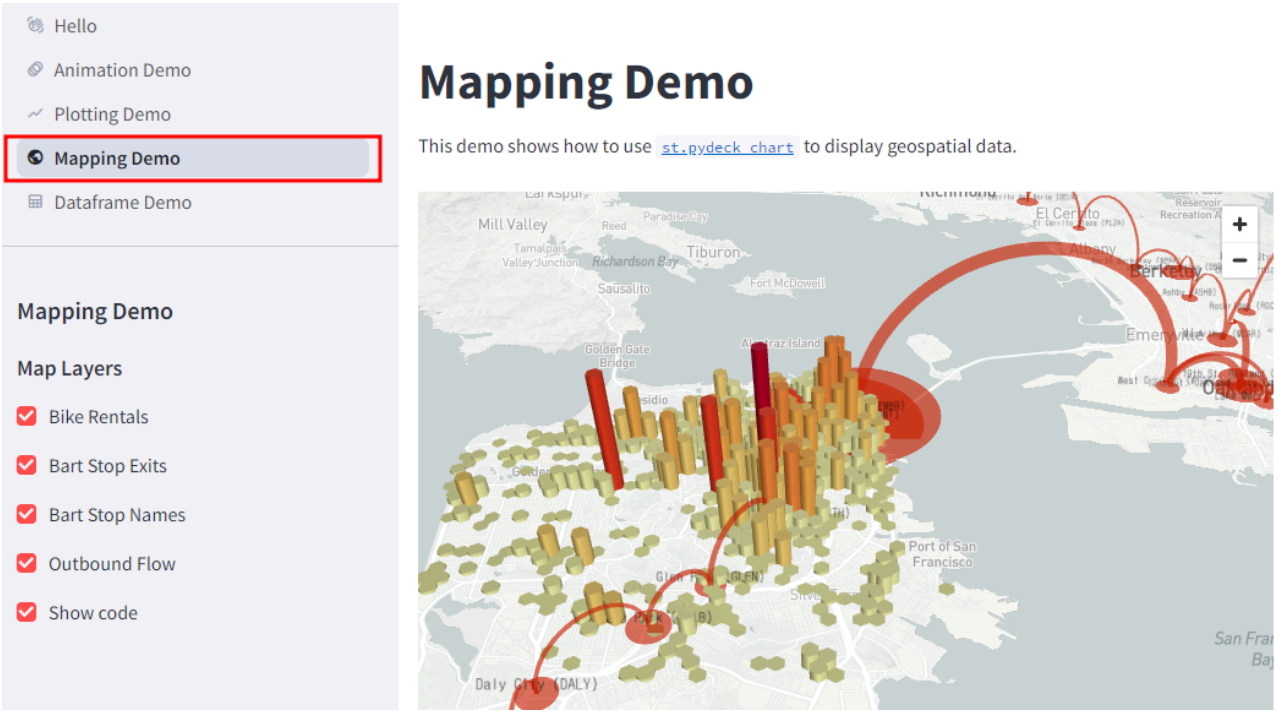
애니메이션 데모



플롯팅 데모



매핑 데모



4.1.2. 기본 사용법 알아보기

스트림릿의 사용 방법은 단순하고 직관적입니다. streamlit 패키지에서 제공하는 API를 호출하면, 수행 결과가 화면 위에서부터 아래로 순차적으로 출력됩니다. 다음은 스트림릿에서 제공하는 API를 사용하여 다양한 형식의 텍스트와 컴포넌트를 화면에 출력하는 코드입니다.

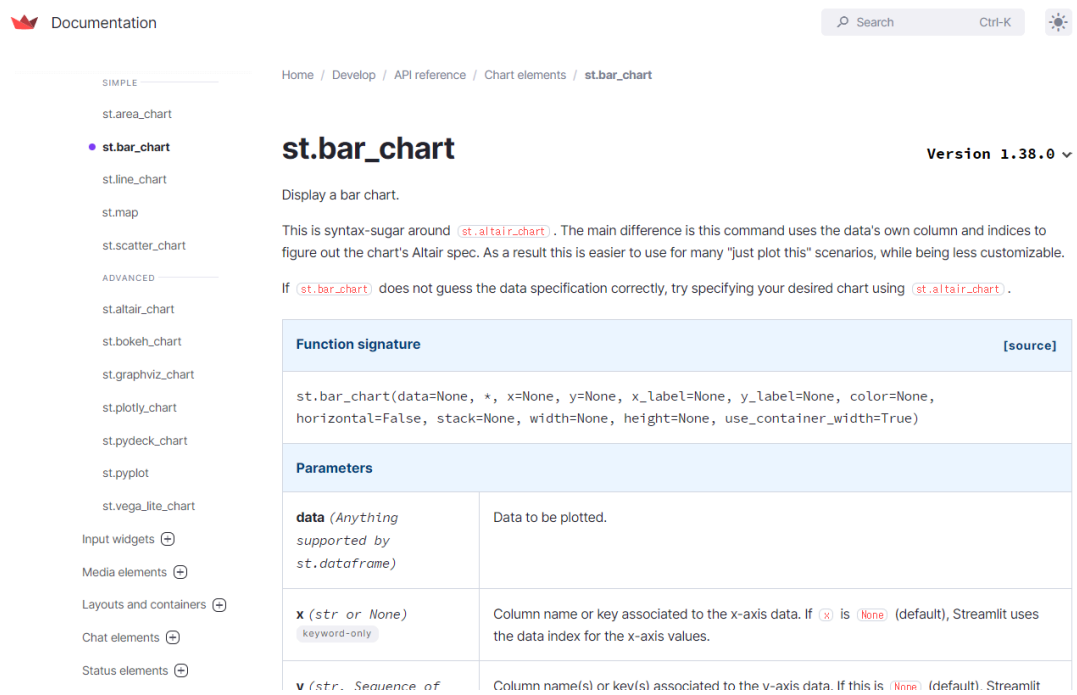
```
1  import streamlit as st
2
3  text = "마지막 레이어의 로짓값을 가정"
4  st.header(text, divider='rainbow')
5  st.subheader(text)
6  st.title(text)
7  st.write(text)
8  st.write("# Bar Chart")
9  vocab_logits = {"나는": 0.01,"내일": 0.03,"오늘": 0.25,"어제": 0.3,
10                 "산에": 0.4,"학교에": 0.5,"집에": 0.65,
11                 "오른다": 1.2,"간다": 1.05,"왔다": 0.95}
12 st.bar_chart(vocab_logits)
13 st.caption(text)
14
15
```

작성한 코드는 다음 명령어를 통해 streamlit 환경으로 실행합니다.

```
> streamlit run 09_01_st_test1.py
```

브라우저에서 터미널에 출력된 주소(localhost:8501)로 접근하면 다음의 결과를 확인할 수 있습니다.

프로그램과 실행 결과를 비교해보면 알 수 있듯이 header, title, write, bar_chart, caption 등의 메서드를 차례대로 호출했더니 화면에서는 그 순서대로 UI 요소들이 출력되었습니다. 이렇게 스트림릿을 사용하면 코드 상의 API와 화면에서 출력되는 내용을 직관적으로 매칭할 수 있어 개발하기가 용이합니다. 따라서 스트림릿에서 제공되는 API와 사용 방법만 알면 다양한 기능을 쉽게 적용할 수 있습니다. 스트림릿 공식 문서를 보면 API에 대한 세부적인 설명과 사용 예제가 자세히 나와 있어서 이 문서를 보고 애플리케이션을 구축하는 것이 가장 정확하고 효율적입니다.

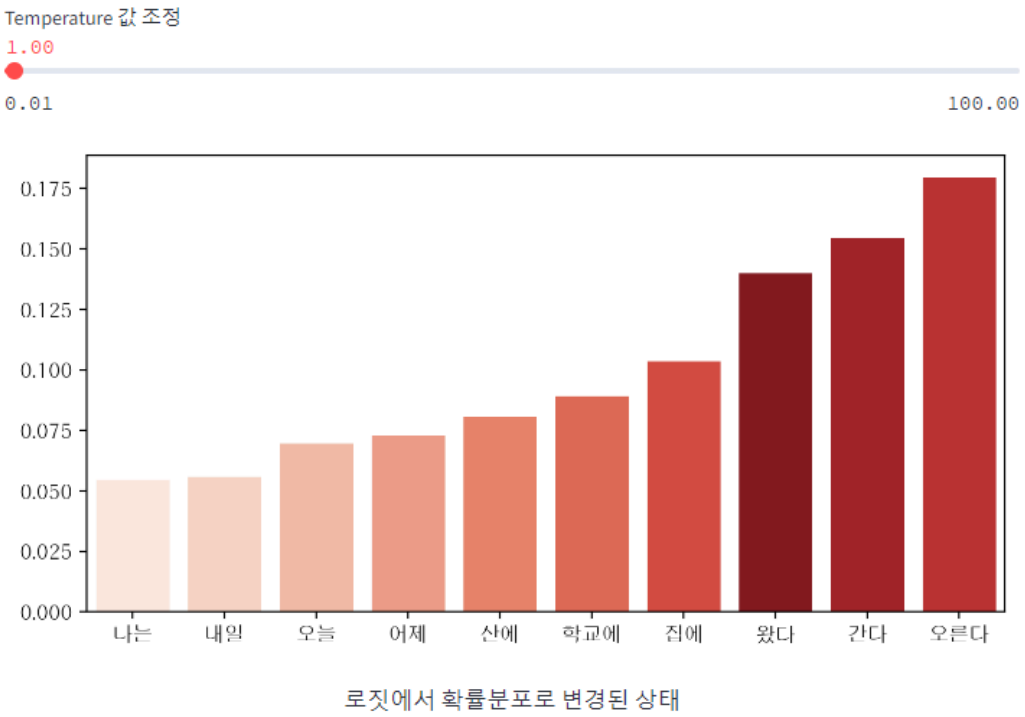


Streamlit API 공식 문서

한편, 앞서 살펴본 예제처럼 내장 컴포넌트를 사용하지 않고, Matplotlib, Seaborn 등 외부 라이브러리와 연동하여 시각화할 수 있는 방법도 있습니다. 다음은 3장에서 다루었던 시각화 코드를 스트림릿으로 적용한 예제입니다.

```python
import streamlit as st
import matplotlib.font_manager as fm
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

# for font in fm.fontManager.ttflist: print(font.name)
plt.rcParams['font.family'] = 'HYGraphic-Medium'

vocab_logits = {"나는": 0.01, "내일": 0.03, "오늘": 0.25, "어제": 0.3,
                "산에": 0.4, "학교에": 0.5, "집에": 0.65,
                "오른다": 1.2, "간다": 1.05, "왔다": 0.95}

def softmax_with_temperature(values, temperature):
    epsilon = 1e-2
    temperature = max(temperature, epsilon)

    exp_values = np.exp(np.array(values) / temperature)
    sum_exp_values = np.sum(exp_values)
    softmax_probs = exp_values / sum_exp_values

    return softmax_probs.tolist()

def draw_prob_graph(vocab, probs):
    fig = plt.figure(figsize=(8, 4))

    colors = sns.color_palette("Reds", n_colors=len(vocab))

    sorted_vocab_prob = sorted(zip(vocab, probs), key=lambda x: x[1])
    sorted_vocab, sorted_probs = zip(*sorted_vocab_prob)

    palette_as_list = [colors[vocab.index(word)] for word in sorted_vocab]

    sns.barplot(x=sorted_vocab, y=sorted_probs, hue=sorted_vocab, palette=palette_as_list, dodge=False)
```

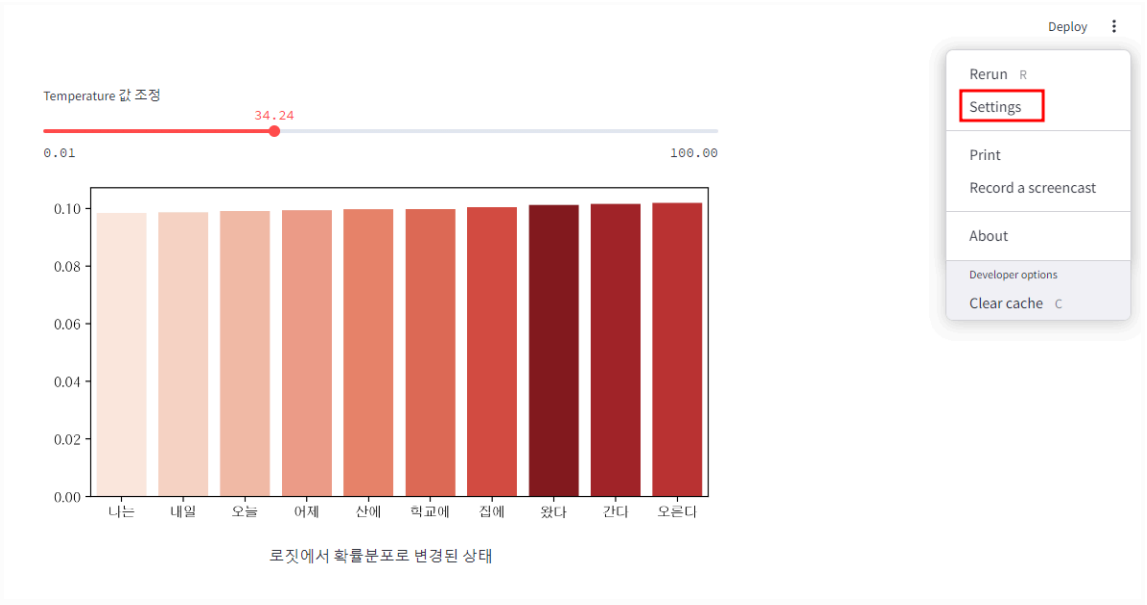
```
    plt.legend([],[], frameon=False)

    st.pyplot(fig)

temperature = st.slider("Temperature 값 조정", min_value=0.01, max_value=100.0, value=1.0, step=0.01, key='temp_slider')
vocab = list(vocab_logits.keys())
logits = list(vocab_logits.values())
probs = softmax_with_temperature(logits, temperature=temperature)

draw_prob_graph(vocab, probs)

centered_text = "<div style='text-align:center'> 로짓에서 확률분포로 변경된 상태</div>"
st.markdown(centered_text, unsafe_allow_html=True)
```

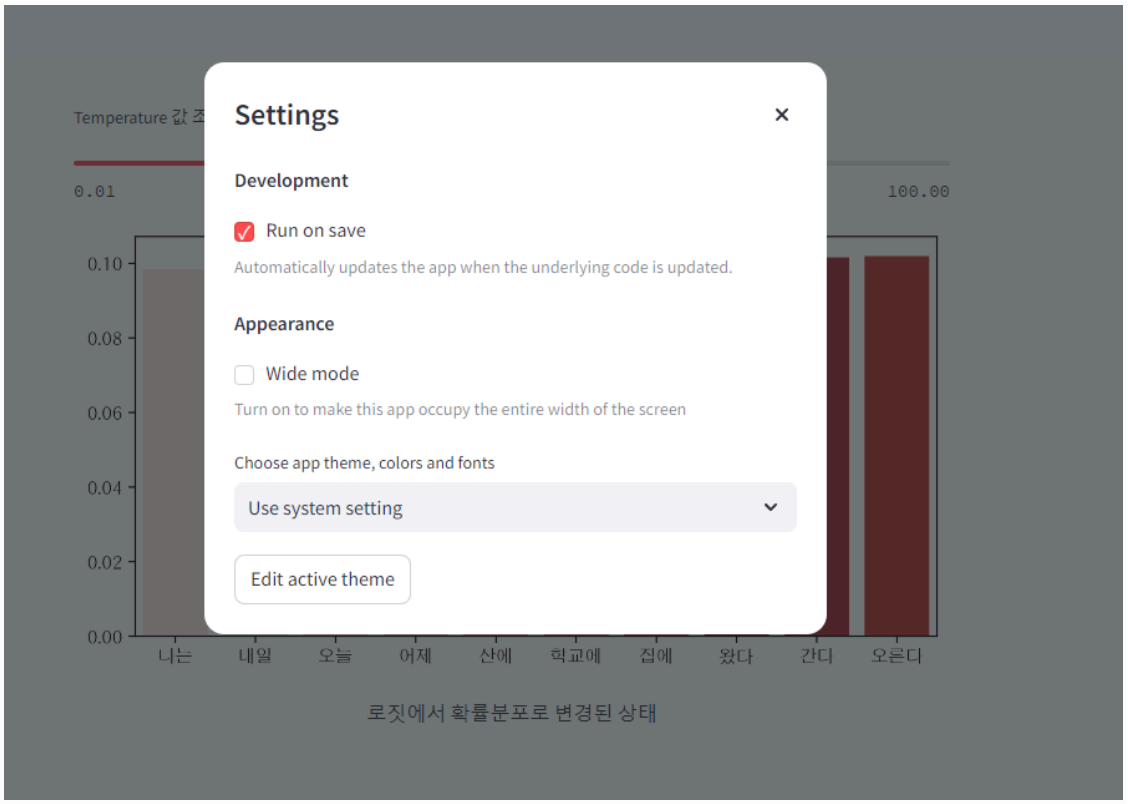


로짓에서 확률분포로 변경된 상태

스트림릿에서 제공하는 슬라이더 컴포넌트를 활용하여 temperature를 동적으로 적용되도록 시각화했습니다. 원래 코드와 달라진 부분은 matplotlib의 figure 객체를 스트림릿의 pyplot API로 전달하는 이 한 부분 밖에 없습니다. 이처럼 스트림릿을 사용하면 내장 컴포넌트뿐만 아니라 파이썬 외부 라이브러리와 손쉽게 연동함으로써 더욱 다채롭고 세밀한 시각적 경험을 사용자들에게 효과적으로 제공할 수 있습니다.

설정 창에서 “Run on save” 항목을 체크하면 편집한 코드가 즉시 화면에 반영됩니다.



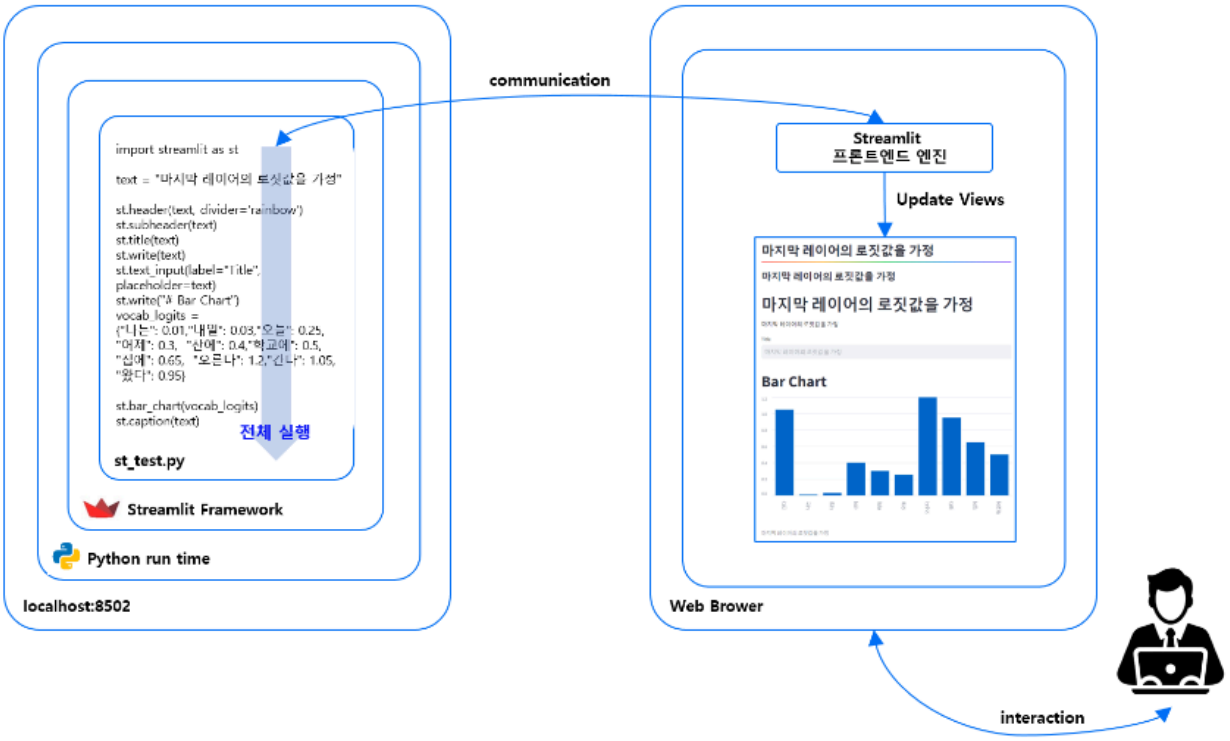


4.2. 스트림릿 핵심 콘셉트

스트림릿 (Streamlit) 프레임워크는 UI 요소와 실시간 연동되는 파이썬 기반의 컴포넌트 객체를 통해 사용자와 상호 작용합니다. 이를 위해 스트림릿은 먼저 파이썬 스크립트가 최초 실행될 때 화면의 레이아웃과 구성 요소를 배치합니다. 이후 사용자의 인터랙션이 발생하면, 파이썬 스크립트를 다시 실행하여 애플리케이션의 전체적인 상태를 갱신하고, 변경된 상태 정보를 기반으로 UI를 새롭게 렌더링합니다.

4.2.1. 반응형 프로그래밍

반응형 프로그램은 UI 요소에 일어나는 변화를 효과적으로 감지하고 처리하기 위해 애플리케이션의 데이터 상태와 UI 요소의 상태를 동기화하여 관리합니다. 스트림릿 프레임워크도 이러한 반응형 프로그래밍의 원리를 따르고 있습니다. 다음은 스트림릿 프레임워크에서 파이썬 스크립트와 UI 요소가 연동하는 과정을 나타내는 다이어그램입니다.



다이어그램에서 볼 수 있듯이 스트림릿 환경에서 수행되는 파이썬 스크립트는 스트림릿에서 관리하는 프론트엔드 엔진으로 전달되어 화면상의 컴포넌트로 출력됩니다. 스트림릿의 프론트엔드 엔진은 SPA(Single Page Application)로 동작하기 때문에 최초 출력 이후에는 웹 페이지 전체를 리로드하지 않고, 서버로부터 받은 데이터나 정보를 바탕으로 화면을 업데이트합니다. 위의 다이어그램에서 주목할 부분은 사용자가 화면상의 컴포넌트에 인터랙션을 가하면, 스트림릿 프레임워크는 애플리케이션의 상태를 최신화하기 위해 파이썬 스크립트 전체를 다시 실행한다는 점입니다.

```
66  print("start...")
67  import streamlit as st
68  text = "마지막 레이어의 로짓값을 가정"
69  st.header(text, divider='rainbow')
70  st.subheader(text)
71  st.title(text)
72  st.write(text)
73  st.text_input(label="Title", placeholder=text)
74  st.write("# Bar Chart")
75  vocab_logits = {"나는": 0.01,"내일": 0.03,"오늘": 0.25,"어제": 0.3,
76                  "산에": 0.4,"학교에": 0.5,"집에": 0.65,
77                  "오른다": 1.2,"간다": 1.05,"왔다": 0.95}
78  st.bar_chart(vocab_logits)
79  st.caption(text)
80  print("end...")
```

맨 앞줄과 맨 끝줄에 로그를 출력하는 코드를 추가했습니다. 화면의 input 상자에 텍스트를 입력한 후 탭 등을 클릭하여 포커스를 벗어나면 콘솔 창에 다음처럼 출력되는 것을 확인할 수 있습니다.

```
문제   출력   디버그 콘솔   터미널   포트

  Local URL: http://localhost:8501
  Network URL: http://192.168.200.67:8501

start...
end...
```

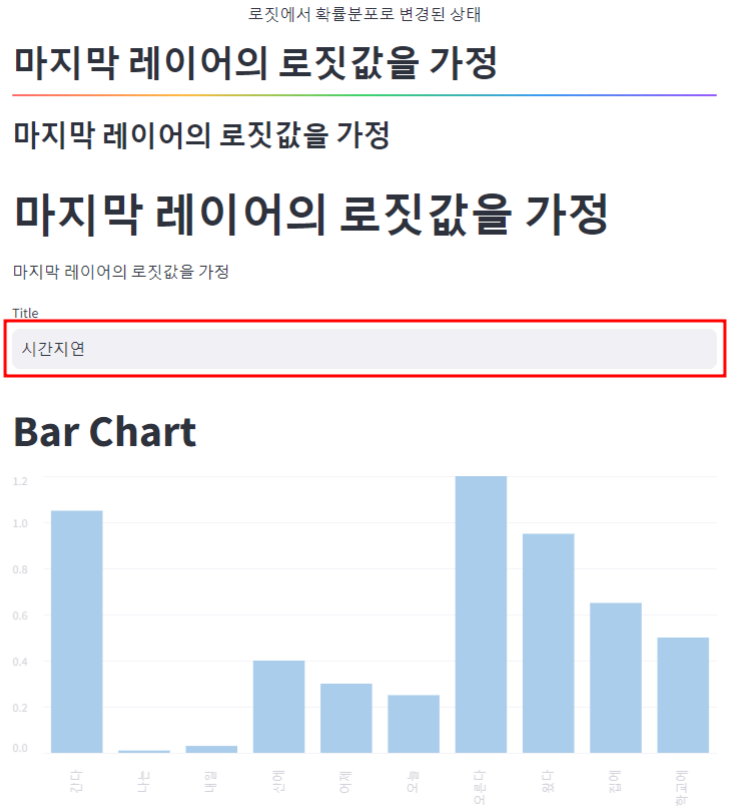
추가된 결과 화면

4.2.2. 데이터 캐싱

전체 스크립트를 다시 실행하면 개발 과정을 단순화하는 장점이 있지만, 중복된 내용을 재실행하는 문제점 역시 존재합니다. 앞의 예제에서 로짓 정보를 가져오는 작업이 LLM 모델을 호출한다고 가정하면 사용자와 상호작용할 때마다 불필요한 추론 작업을 거쳐야 할 것입니다.

다음은 로짓 산출 시 10초의 딜레이를 가정하고 실험하는 예제입니다.

```python
import streamlit as st
import time
def get_vocab_logits():
    print(f"get_vocab_logits() starting")
    time.sleep(10)
    vocab_logits = {"나는": 0.01,"내일": 0.03,"오늘": 0.25,"어제": 0.3,
                    "산에": 0.4,"학교에": 0.5,"집에": 0.65,
                    "오른다": 1.2,"간다": 1.05,"왔다": 0.95}
    print(f"get_vocab_logits() ending")
    return vocab_logits

text = "마지막 레이어의 로짓값을 가정"
st.header(text, divider='rainbow')
st.subheader(text)
st.title(text)
st.write(text)
st.text_input(label="Title", placeholder=text)
st.write("# Bar Chart")
st.bar_chart(get_vocab_logits())
st.caption(text)
```



input 상자에 글자를 입력하고 탭을 클릭하면, 약 10초 동안 위와 같이 그래프가 흐리게 나타나면서 지연 현상이 발생합니다. 이런 현상이 나타나는 이유는, 사용자의 상호작용에 의해 화면에 변화가 생기면, 이를 반영하기 위해 파이썬 스크립트 전체가 다시 실행되어야 하고, 그 과정에서 로짓 산출을 위해 10초 간의 대기 시간이 발생하기 때문입니다.

이러한 불합리 점을 개선하기 위해 스트림릿에서는 캐싱 매커니즘을 제공합니다. 스트림릿의 @st.cache_data 데코레이터를 사용하면 함수를 실행한 결과가 캐시에 보관됩니다. 이후부터는, 파라미터가 달라졌거나 구현 내용이 달라진 경우에 한해서만 함수를 다시 실행합니다. 만일 함수의 코드가 변경되지 않았고, 이전에 호출된 파라미터와 동일하다면, 캐시에 보관되어 있는 결과를 반환함으로써 불필요한 재실행을 막습니다.

다음은 스트림릿의 캐싱 메커니즘을 적용하여 사용자 경험을 향상시키는 예제입니다.

```python
# st_cache_data.py
import streamlit as st
import time

@st.cache_data
def get_vocab_logits(param=0):
    print(f"get_vocab_logits({param}) starting")
    time.sleep(10)
    vocab_logits = {"나는": 0.01,"내일": 0.03,"오늘": 0.25,"어제": 0.3,
                    "산에": 0.4,"학교에": 0.5,"집에": 0.65,
                    "오른다": 1.2,"간다": 1.05,"왔다": 0.95}
    vocab_logits = {word: logit + param for word, logit in vocab_logits.items()}
    print(f"get_vocab_logits({param}) ending")
    return vocab_logits

text = "마지막 레이어의 로짓값을 가정"
st.header(text, divider='rainbow')
st.subheader(text)
st.title(text)
st.write(text)

user_input = st.number_input(label="로짓값에 더해지는 숫자를 입력하세요.", value=0)

st.write("# Bar Chart")
st.bar_chart(get_vocab_logits(user_input))
st.caption(text)
```

get_vocab_logits 함수에 @st.cache_data 데코데이터를 적용했고, 화면에서 입력받은 값을 로짓에 일괄적으로 더하도록 프로그래밍했습니다. 다음은 input 상자의 값을 0→1→0→1.. 로 실험하는 과정 중의 일부를 캡처한 화면입니다.

예상대로 처음 실행한 파라미터에서 대해서는 지연 현상이 발생하지만, 이미 실행했던 파라미터에서 대해서는 지연 없이 화면에 그래프로 출력되는 것을 확인할 수 있습니다.

@st.cache_data 데코레이터를 사용할 때 한 가지 유의할 점이 있습니다. @st.cache_data는 함수에 전달되는 파라미터와 구현 내용이 동일하다면 함수의 호출 결과 역시 항상 동일할거라는 가정 하에 캐싱 메커니즘을 적용한다고 설명했습니다. 따라서 함수 내에서 데이터베이스에 접근하거나 네트워크 통신 등을 통해 그 결과가 변경되도록 코드가 구현되어 있다면 원하는대로 동작하지 않을 수 있다는 점을 알고 있어야 합니다.

앞의 코드를 변경하여, 함수가 호출될 때마다 수행 결과가 달라지지만 그 내용이 화면에 반영되지 않도록 만든 예제입니다.

```
import streamlit as st
import time

total=0

@st.cache_data
def get_vocab_logits(param=0):
    print(f"get_vocab_logits({param}) starting")
    global total
    time.sleep(10)
    vocab_logits = {"나는": 0.01,"내일": 0.03,"오늘": 0.25,"어제": 0.3,
                    "산에": 0.4,"학교에": 0.5,"집에": 0.65,
                    "오른다": 1.2,"간다": 1.05,"왔다": 0.95}
    vocab_logits = {word: logit + param + total for word, logit in vocab_logits.items()}
    total +=1
    print(f"get_vocab_logits({param}) ending")
    return vocab_logits

text = "마지막 레이어의 로짓값을 가정"
st.header(text, divider='rainbow')
st.subheader(text)
st.title(text)
st.write(text)

user_input = st.number_input(label="로짓값에 더해지는 숫자를 입력하세요.", value=0)

st.write("# Bar Chart")
st.bar_chart(get_vocab_logits(user_input))
st.caption(text)
```

전역변수 total을 만들어서 get_vocab_logits 함수가 호출될 때마다 1씩 누적되도록 했고, 그 값이 로짓에 더해지도록 했습니다. 하지만 입력값을 0→1→0→1..로 테스트하면, 캐싱된 다음부터는 로짓값이 변하지 않는다는 사실을 확인할 수 있습니다. 이처럼 스트림릿의 @st.cache_data 데코레이터는 함수의 파라미터 값과 코드 내용이 같다면 캐시된 값을 반환하기 때문에 데이터베이스나 전역변수 같은 외부 요소에 의존하는 경우 그 결과가 올바르게 반영되지 않을 수 있습니다.

4.2.3. 리소스 캐싱

@st.cache_data 데코레이터가 데이터에 대한 캐싱 메커니즘을 적용한다면 @st.cache_resource 데코레이터는 머신러닝 모델이나 데이터베이스 컨넥션 등의 리소스를 효율적으로 사용하도록 돕습니다. @st.cache_resource를 사용하면 프로그램에서 사용되는 이러한 리소스를 한 번만 로드하거나 연결하고, 이후의 호출에서는 캐싱된 인스턴스를 재사용함으로써 처리 시간을 단축시킵니다.

다음은 앞에서 다룬 @st.cache_data 데코레이터를 활용해서 캐싱을 시도하다가 오류가 발생하는 예시입니다.

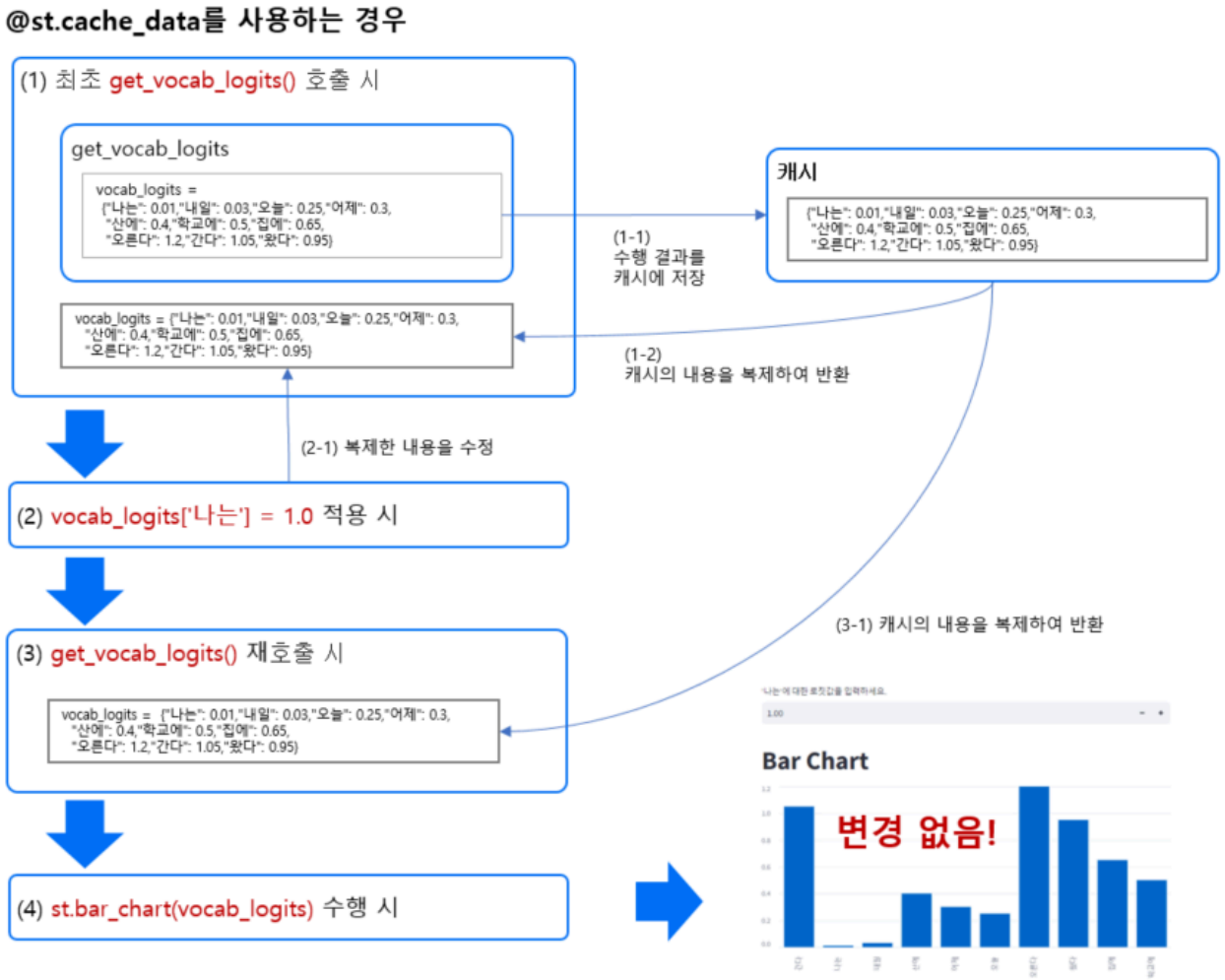
```
33  class LinearModel:
34      def __init__(self, filepath="weight.txt"):
35          self.file_handle = open(filepath, "r")
36
37      def predict(self, input_data):
38          self.file_handle.seek(0)
39          try:
40              weight = float(self.file_handle.read().strip())
41          except ValueError:
42              weight = 0
43          return input_data * weight + 2
44
45      def close_file(self):
46          if self.file_handle:
47              self.file_handle.close()
48              self.file_handle = None
49  @st.cache_data
50  def load_linear_model():
51      print("loading started...")
52      model = LinearModel()
53      return model
54
55  try:
56      model = load_linear_model()
57      st.write("모델이 정상적으로 로드되었습니다.")
58  except Exception as e:
59      st.write(f"Error loading model: {e}")
```

Error loading model: Cannot serialize the return value (of type `__main__.LinearModel`) in `load_linear_model()`. `st.cache_data` uses pickle to serialize the function's return value and safely store it in the cache without mutating the original object. Please convert the return value to a pickle-serializable type. If you want to cache unserializable objects such as database connections or Tensorflow sessions, use `st.cache_resource` instead (see our docs for differences).

이 예제를 보면 self.file_handle = open(filepath, "r") 코드를 통해 파일의 참조값을 멤버변수에 할당하고 있습니다. 하지만 파일의 참조값은 실행 환경에 종속되는 것으로서 직렬화하거나 역직렬화할 수 없고, 이에 따라 @st.cache_data 데코레이터를 사용하면 오류가 발생합니다. 출력된 메시지를 보면 'pickle을 사용하여 직렬화하기 때문에 데이터베이스 연결이나 텐서플로우 세션 같은 직렬화할 수 없는 객체를 캐싱하려면 st.cache_resource를 사용하라'고 권고합니다. 권고대로 예제에서 사용된 @st.cache_data를 @st.cache_resource로 교체하면 프로그램이 정상적으로 기동된다는 것을 확인할 수 있습니다.

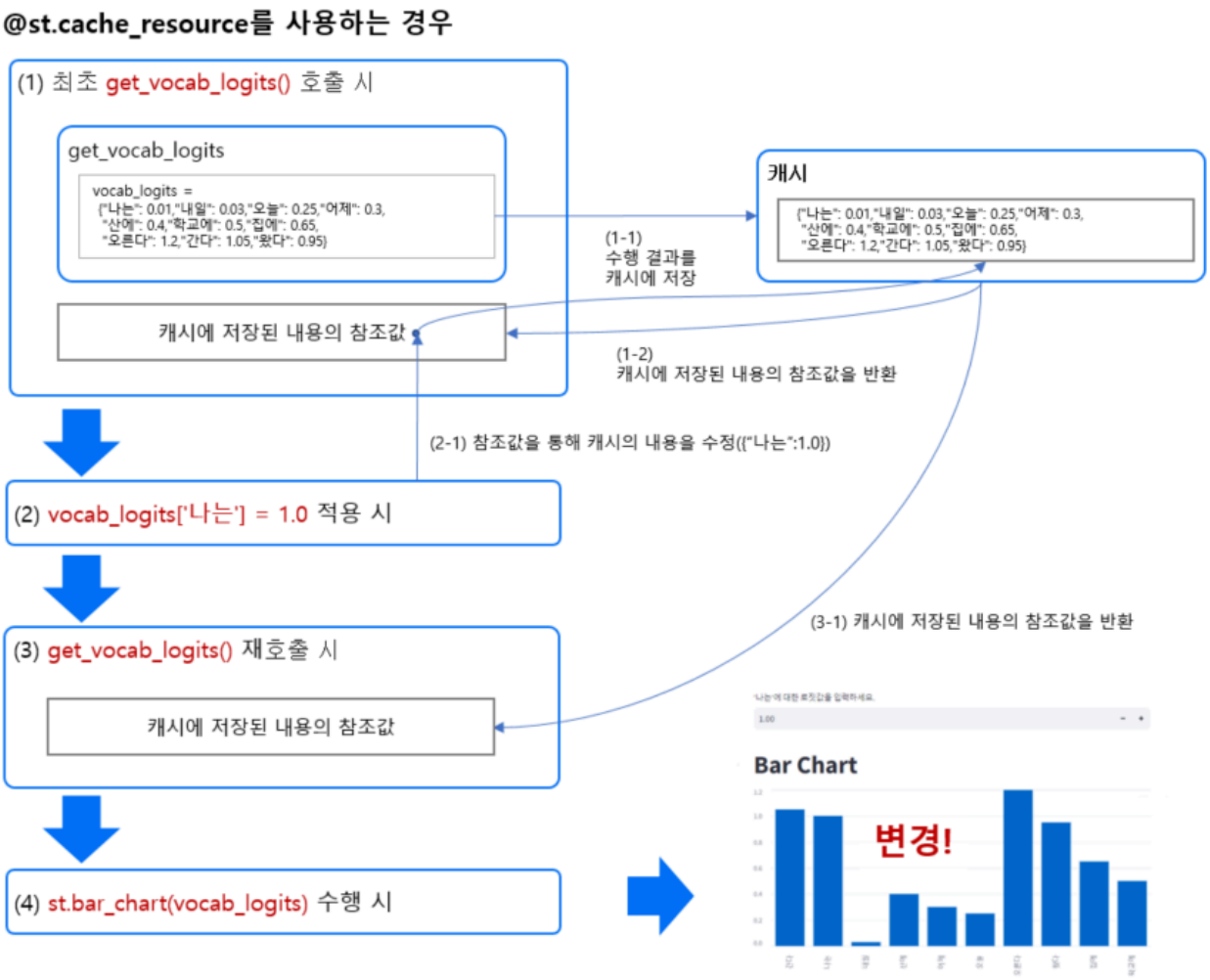
이 밖에도 @st.cache_data를 사용하면 함수에서 산출된 최종 결괏값을 복제하여 반환하는 반면, @st.cache_resource를 사용하면 함수의 결괏값을 가리키는 참조(레퍼런스)를 반환합니다. 따라서 함수로부터 반환받은 결과를 프로그램 내에서 바꾸는 경우, 어떤 캐싱 메커니즘을 사용했냐에 따라 결과가 다르게 나타납니다. 다음은 원본 데이터의 변경 여부와 관련하여 두 가지 캐싱 데코레이터의 동작 과정을 설명하는 다이어그램입니다.

먼저 @st.cache_data에 대한 시나리오입니다.



@st.cache_data를 사용하면 캐시에 저장된 값을 복제하여 반환하므로, 반환받은 값을 변경하고 다시 함수를 호출하더라도 원래 값이 그대로 화면에 출력되는 것을 나타냅니다.

다음은 @st.cache_resource를 적용한 시나리오입니다.



@st.cache_resource를 사용하면 결과에 대한 참조값을 반환하므로, 반환된 값을 변경(반환된 참조값을 통해 캐시의 내용을 변경)한 후 다시 함수를 호출하면 화면에 변경된 내용이 출력된다는 것을 보여줍니다.

다음은 이상의 사니라오를 구현하는 코드입니다. @st.cache_data와 @st.cache_resource로 데코레이터를 바꾸어가며 실험하면 위에서 예시한 시나리오대로 동작하는 것을 확인할 수 있습니다.

```
62  @st.cache_data
63  # @st.cache_resource
64  def get_vocab_logits(param=None):
65      print(f"get_vocab_logits({param}) starting")
66      vocab_logits = param
67      vocab_logits = {"나는": 0.01,"내일": 0.03,"오늘": 0.25,"어제": 0.3,
68                      "산에": 0.4,"학교에": 0.5,"집에": 0.65,
69                      "오른다": 1.2,"간다": 1.05,"왔다": 0.95}
70      print(f"get_vocab_logits({param}) ending")
71      return vocab_logits
72
73  user_input = st.number_input(label="'나는'에 대한 로짓값을 입력하세요.", value=0.01)
74
75  st.write("# Bar Chart")
76  vocab_logits = get_vocab_logits()  #(1)함수의 결괏값을 복사하여 반환받음.
77  vocab_logits['나는'] = user_input  #(2)복사한 값을 변경함
78  vocab_logits = get_vocab_logits()  #(3)함수의 결괏값을 다시 복사하여 반환받음.
79  st.bar_chart(vocab_logits)
```

지금까지 살펴본 바를 요약하면, 직렬화가 가능하고 데이터의 불변성이 필요한 경우에는 데이터 캐싱 메커니즘을 사용하는 것이 적합한 반면, 직렬화할 수 없으며 결과를 전역적으로 공유해야 하는 경우에는 리소스 캐싱 메커니즘을 사용하는 것이 효율적입니다. 따라서 복잡한 연산 결과를 재사용하려고 할 때에는 데이터 캐싱 메커니즘을 사용해야 하지만, 프로그램 내에서 자원을 효율적으로 관리하고 싶을 때에는 리소스 캐싱 메커니즘을 사용해야 합니다.

4.2.4. 세션 상태 관리

사용자와 상호작용하는 챗봇을 구현하려면 사용자별 세션 정보가 필요합니다. 2장에서 배웠던 것처럼 멀티턴 대화를 위해서는 챗봇과 사용자가 나눈 대화 이력을 모델의 입력값으로 전달해야 하므로, 동시에 여러 명의 사용자가 접속한다면 사용자별로 서로 다른 이력이 관리되어야 합니다. 스트림릿 프레임워크는 session_state 객체를 통해 세션 정보를 관리할 수 있게 합니다. 다음은 스트림릿 공식 문서에서 제공하는 session_state 예제 코드입니다.

```
1  import streamlit as st
2
3  if "counter" not in st.session_state:
4      st.session_state.counter = 0
5
6  st.session_state.counter += 1
7
8  st.header(f"This page has run {st.session_state.counter} times.")
9  st.button("Run it again")
```

session_state 객체는 세션별로 딕셔너리와 유사한 구조로 데이터를 관리합니다. 따라서 if "counter" not in st.session_state: 라는 구문은 세션 정보 내에 "counter" 라는 키가 존재하는지 확인하는 조건문으로 동작합니다. 해당 키에 대한 정보가 없으면 st.session_state.counter = 0 이라는 문장을 통해 counter의 값을 만들어서 0으로 초기화합니다. 이때 st.session_state["counter"] = 0과 같이 딕셔너리 형식을 사용해도 같은 결과를 얻을 수 있습니다. 결과적으로 위의 코드는 버튼을 클릭할 때 세션별로 카운터를 증가시키고 그 결과를 화면에 출력하는 프로그램입니다. 위의 프로그램이 세션별로 독립적으로 동작하는지 확인하려면 브라우저의 새 탭을 열어 동일한 주소로 접속한 다음 각 탭에서 버튼을 클릭해보면 됩니다.

각각의 탭(또는 브라우저)에서 버튼을 클릭하면 서로 다르게 누적되는 것을 확인할 수 있습니다.

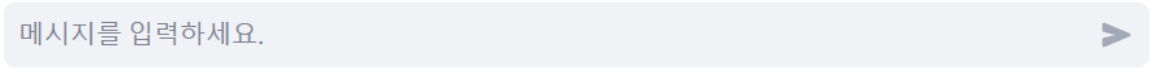
4.3. 제미나이 챗봇 만들기

스트림릿(Streamlit)은 대화형 인공지능을 시연할 수 있도록 메시지 입력, 메시지 출력 등의 챗봇 컴포넌트를 제공합니다. 이러한 컴포넌트와 지금까지 배운 스트림릿 캐싱, 세션 메커니즘을 결합하면, 인공지능과 자연스럽게 대화를 나누는 챗봇을 편리하게 구현할 수 있습니다.

4.3.1 메시지 컨테이너

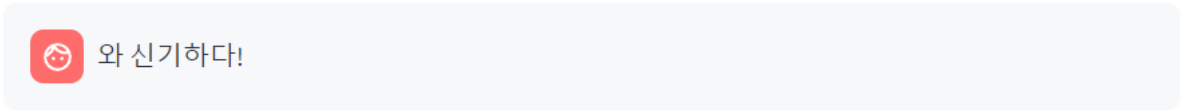
스트림릿 프레임워크의 st.chat_input 메서드를 호출하면 화면에 메시지를 넣을 수 있는 입력 메시지 컨테이너가 나타납니다.

```
prompt = st.chat_input("메시지를 입력하세요.")
```



이 컨테이너에 메시지를 입력하고 엔터키를 누르면 입력한 문자열이 st.chat_input 메서드의 반환값으로 돌아옵니다. 이때 다음과 같이 st.chat_message 메서드를 실행하면 출력 메시지 컨텍스트가 내부적으로 생성됩니다. 그리고 이 상태에서 prompt를 입력값으로 넣고 st.write 메서드를 호출하면 입력한 메시지(prompt)가 화면상에 출력 메시지 컨테이너와 함께 나타납니다.

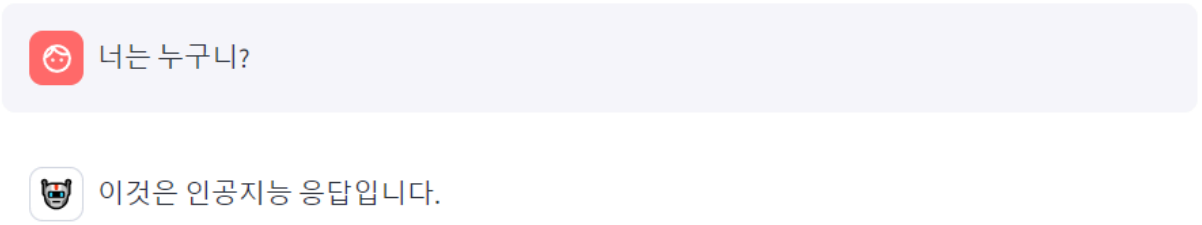
```
prompt = st.chat_input("메시지를 입력하세요.")
if prompt:
    with st.chat_message("user"):
        st.write(prompt)
```



입력 메시지와 마찬가지로 모델의 응답 메시지를 출력하고 싶으면 st.chat_message("ai") 코드를 추가하여 새로운 출력 메시지 컨텍스트를 생성하고, st.write 메서드를 호출해 응답 메시지를 화면으로 내보내면 됩니다. 참고로 매개변수에 "user", "assistant", "ai", "human" 중 하나를 전달하면 사전에 정의된 아바타가 컨테이너의 왼쪽에 출력됩니다. 만일 사용자가 직접 아바타를 설정하고 싶다면 st.chat_message(name="ai", avatar="이모지") 형식으로 작성하면 됩니다.

```
import streamlit as st

prompt = st.chat_input("메시지를 입력하세요.")
if prompt:
    with st.chat_message("user"):
        st.write(prompt)
```

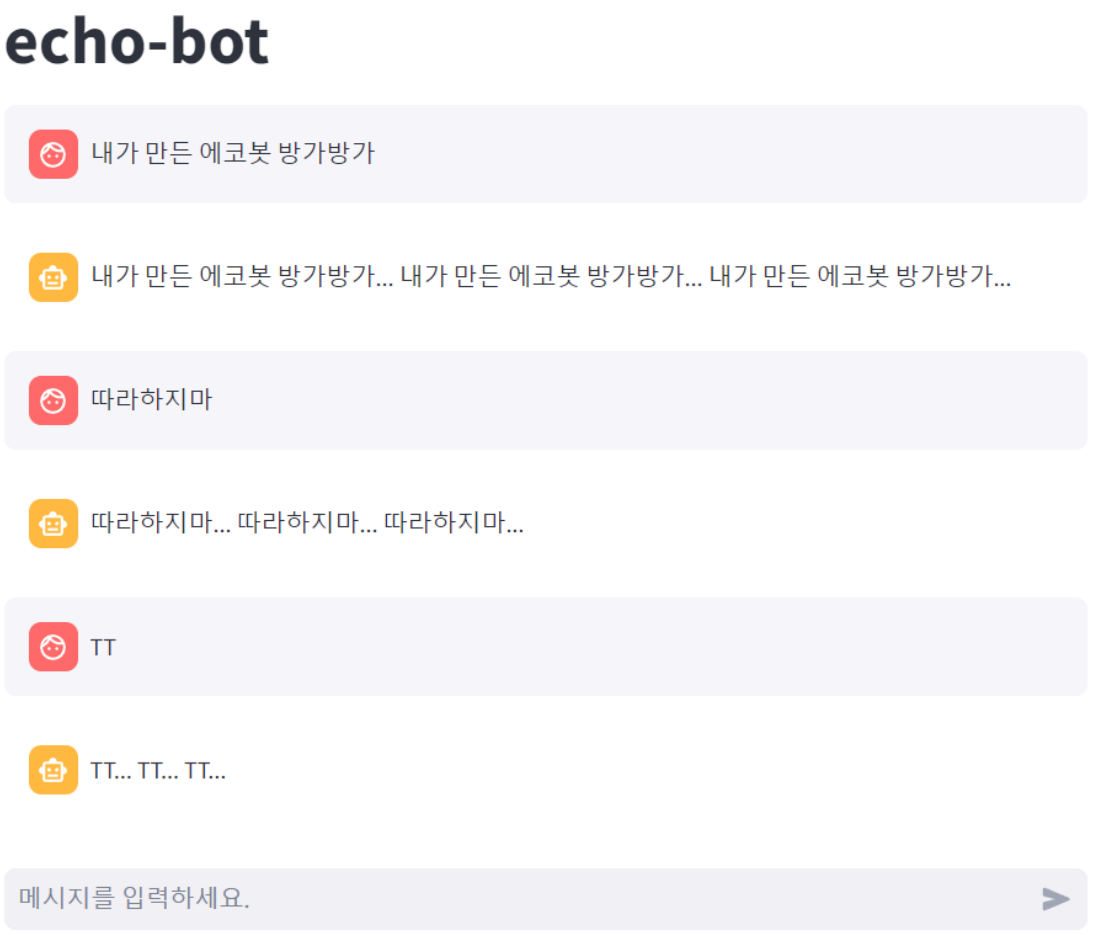
```
with st.chat_message("ai", avatar="🤖"):
    st.write("이것은 인공지능 응답입니다.")
```



그런데, 사용자와 지속적으로 상호작용하는 챗봇을 만들려면 단발적인 입력/응답 메시지뿐만 아니라 대화 이력 전체가 매번 화면에 출력되어야 합니다. 이것을 위해서는 세션 상태 메커니즘을 활용하여 대화 이력을 저장하고, 사용자의 상호작용이 발생할 때마다 이력 전체를 출력하는 로직이 추가되어야 합니다. 다음은 이러한 요구사항을 충족하는 에코 챗봇을 구현하는 코드입니다.

```python
import streamlit as st
st.title("echo-bot")

if "chat_history" not in st.session_state:
    st.session_state.chat_history = []

for content in st.session_state.chat_history:
    with st.chat_message(content["role"]):
        st.markdown(content['message'])

if prompt := st.chat_input("메시지를 입력하세요."):
    with st.chat_message("user"):
        st.markdown(prompt)
        st.session_state.chat_history.append({"role": "user", "message": prompt})

    with st.chat_message("ai"):
        response = f'{prompt}... {prompt}... {prompt}...'
        st.markdown(response)
        st.session_state.chat_history.append({"role": "ai", "message": response})
```

session_state 객체를 활용하여 대화 이력을 세션으로 관리하도록 구현했습니다. 그리고 사용자와 AI의 메시지가 생성될 때마다 대화 이력에 메시지 정보를 추가했습니다. 이렇게 대화 이력에 추가된 메시지는 사용자의 인터랙션이 있을 때마다 for-loop 구문을 통해 메시지 전체가 화면으로 출력되도록 했습니다. 이때 화면으로 출력되는 내용에는 메시지의 생성 주체가 표현되어야 하므로, 대화 이력을 저장할 때는 딕셔너리 데이터 타입을 활용하여 메시지와 생성자의 역할("user", "ai")을 함께 저장했습니다. 참고로 텍스트 표현력을 향상하기 위해 st.write는 st.markdown 메서드로 교체했습니다. 다음은 에코봇 테스트 화면입니다.



4.3.2 제미나이 챗봇 단계별 구현하기

지금까지 배운 내용을 종합하여 제미나이 챗봇을 단계별로 구현하겠습니다.

모델 가져오기

제미나이 API를 사용하는 경우, 데이터 캐싱으로 모델을 가져와도 직렬화 오류가 발생하지 않습니다. 바이너리 상태의 실제 모델을 로드하는 것이 아니라 API 연동을 위한 클래스의 인스턴스를 생성하는 것이기 때문에 그렇습니다. 하지만 사용자의 인터랙션이 있을 때마다 인스턴스를 매번 생성하는 것보다는 참조를 전달받아 접근하는 것이 더욱 효율적이므로 여기에서는 리소스 캐싱을 사용했습니다.

```
@st.cache_resource
def load_model():
    model = genai.GenerativeModel('gemini-1.5-flash')
print("model loaded...")
    return mode

model = load_model()
```

세션별 이력 관리

대화 이력은 별도의 딕셔너리 데이터 구조로 관리하지 않고, 2장에서 배웠던 구글 제미나이 API의 ChatSession을 그대로 사용하면 됩니다.

```
if "chat_session" not in st.session_state:
    st.session_state["chat_session"] = model.start_chat(history=[])
```

메시지 출력

사용자 메시지의 출력은 앞서 실습했던 코드를 그대로 적용하면 되고, 모델의 응답은 ChatSession 객체에 있는 send_message를 호출한 결과를 출력하면 됩니다.

```
if prompt := st.chat_input("메시지를 입력하세요."):
    with st.chat_message("user"):
        st.markdown(prompt)
    with st.chat_message("ai"):
        response = st.session_state.chat_session.send_message(prompt)
        st.markdown(response.text)
```

대화 이력 출력

ChatSession 객체의 history에는 사용자와 언어모델이 나눈 대화 이력이 role과 parts로 구분되어 들어 있습니다. 따라서 다음과 같이 세션 객체 들어 있는 history 정보를 활용하면 사용자와 제미나이가 나누었던 대화가 화면에 이력으로 출력됩니다. history에 있는 role을 스트림릿에 등록되어 있는 아바타 이름과 맞추기 위해 chat_message 메서드 호출 시 "model"을 "ai"로 변경하는 코드를 추가했습니다.

```
for content in st.session_state.chat_session.history:
    with st.chat_message("ai" if content.role == "model" else "user"):
        st.markdown(content.parts[0].text)
```

테스트 하기

지금까지 살펴본 제미나이 챗봇의 전체 코드와 테스트 결과입니다

```python
import google.generativeai as genai
import streamlit as st

st.title("Gemini-Bot")
genai.configure(api_key="[redacted]")

@st.cache_resource
def load_model():
    model = genai.GenerativeModel('gemini-1.5-flash')
    print("model loaded...")
    return model

model = load_model()

if "chat_session" not in st.session_state:
    # ChatSession 반환
    st.session_state["chat_session"] = model.start_chat(history=[])

for content in st.session_state.chat_session.history:
    with st.chat_message("ai" if content.role == "model" else "user"):
        st.markdown(content.parts[0].text)

if prompt := st.chat_input("메시지를 입력하세요."):
    with st.chat_message("user"):
        st.markdown(prompt)
    with st.chat_message("ai"):
        response = st.session_state.chat_session.send_message(prompt)
        st.markdown(response.text)
```

콘솔에서 단발적으로 실험했던 제미나이와의 대화가 스트림릿 프레임워크를 활용함으로써 훨씬 더 인터랙티브하게 이루어지는 것을 확인할 수 있습니다.

4.3.2. 스트리밍 응답 방식 적용하기

앞의 예제에서 개선할 사항 중 하나는, 모델의 응답이 복잡하거나 양이 많은 경우, 대기 시간이 길어져 사용자 경험에 좋지 않은 영향을 끼칠 수 있다는 점입니다.

응답 분량이 많아지는 경우 30초 이상 소요됨

이런 경우 구글 제미나이 API의 스트림 기능과 스트림릿의 플레이스 홀더 기능을 결합하면 보다 향상된 사용자 경험을 제공할 수 있습니다. 다음은 구글 제미나이 API 호출 시 stream 출력을 적용하는 코드입니다.

```
import google.generativeai as genai
import os

genai.configure(api_key="[redacted]")
model = genai.GenerativeModel('gemini-1.5-flash')
response = model.generate_content("인공지능에 대해 자세히 설명해줘.", stream=True)

for no, chunk in enumerate(response, start=1):
  print(f"{no}: {chunk.text}")
  print("="*50)
```

구글 제미나이 API를 사용할 때, stream=True 옵션을 설정하면 메시지 생성 요청에 대한 응답이 스트리밍 방식으로 제공되어, 모델이 메시지 생성을 마치기 전에도 중간 결과를 실시간으로 수신받을 수 있습니다. 위의 예제를 실행하면 다음과 같이 응답 메시지가 여러 차례 나누어서 출력됩니다.

```
1: ## 인공지
==================================================
2: 능(AI)에 대해 자세히 알아보자!

**
==================================================
3: 인공지능(AI)**은 컴퓨터가 인간의 지
==================================================
4: 능을 모방하는 기술을 말합니다. 즉, **컴퓨터가 배우고, 문제를 해결하고, 의사 결정을
==================================================
5:  내리고, 창의적으로 생각하는 능력**을 갖추도록 하는 것입니다.

### 인공지능의 역사:

*
==================================================
6:  **1950년대:** 초기 AI 연구 시작, 체커 게임 AI 개발
* **1960-70년대:** 전문가 시스템 개발, AI의 한계 인식
* **1980
==================================================
7: -90년대:** 머신 러닝, 딥 러닝 발전, 인공 신경망 연구 활발
```

제미나이 API로부터 전달받는 이와 같은 스트리밍 결과는 스트림릿의 empty 메서드를 활용함으로써 브라우저로 실시간 전달할 수 있습니다. 다음은 empty 메서드를 통해 모델의 결과를 실시간으로 화면에 출력하는 예제입니다.

```python
if prompt := st.chat_input("메시지를 입력하세요."):
    with st.chat_message("user"):
        st.markdown(prompt)
    with st.chat_message("ai"):
        # response = st.session_state.chat_session.send_message(prompt)
        # st.markdown(response.text)
        message_placeholder = st.empty()
        full_response = ""
        with st.spinner("메시지 처리 중입니다."):
            response = st.session_state.chat_session.send_message(prompt, stream=True)
            for chunk in response:
                full_response += chunk.text
                message_placeholder.markdown(full_response)
```

스트림릿의 empty 메서드를 호출하면 DeltaGenerator라는 객체가 반환됩니다. 위의 코드에서 이 객체는 st.chat_message에 의해 생성된 출력 메시지 컨텍스트에 동적으로 데이터를 업데이트하는 플레이스 홀더로서의 역할을 수행합니다. 따라서 모델로부터 스트리밍 방식으로 반환받은 문자열을 이전에 반환받은 문자열과 결합하여 이 플레이스 홀더로 전달하면, 출력 메시지 컨테이너에 문자열이 늘어나는 방식으로 모델의 응답 결과가 화면에 표현됩니다. 이에 따라 사용자는 모델이 생성 중인 응답을 실시간으로 확인하는 경험을 얻을 수 있습니다. 아울러, 위의 코드에서는 spinner 메서드를 사용하여 작업이 진행 중임을 표시함으로써 응답 과정에 대한 사용자 경험이 좀 더 향상되도록 구현했습니다.

Gemini-Bot

인공지능에 대해 자세히 설명해줘

인공지능: 인간의 지능을 모방하는 기술의 세계

인공지능(AI)은 컴퓨터 과학의 한 분야

메시지 처리 중입니다.

메시지를 입력하세요.