تمرین سری اول مبانی بازیابی

دکتر رهائی

هلیا شمس زاده

۴۰۰۵۲۱۴۸۶

سوال اول)

دیتاست مورد استفاده: اخبار فارسی BBC Persian (حدود ۴۰۰۰ رکورد)

کتابخانه pandas برای کار با داده‌های ساختارمند استفاده می‌شود. از hazm برای پردازش زبان طبیعی فارسی شامل نرمال‌سازی، توکن‌سازی، ریشه‌یابی، لمات‌سازی و لیست کلمات توقف استفاده شده است. TfidfVectorizer از sklearnبرای ساخت ماتریس TF-IDF جهت بردارسازی متون استفاده می‌شود. همچنین cosine_similarity برای محاسبه شباهت بین بردارها به کار می‌رود. کتابخانه‌ی faiss برای ایجاد اندیس سریع جستجو بر پایه‌ی بردار استفاده شده است. Numpy برای عملیات عددی، tqdm برای نمایش نوار پیشرفت در زمان پردازش، json برای بارگذاری فایلJSON ، و files از google.colab برای آپلود فایل از سیستم استفاده می‌شود.

```
import pandas as pd
from hazm import Normalizer, WordTokenizer, Stemmer, Lemmatizer, stopwords_list
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
import faiss
import numpy as np
from tqdm import tqdm
import json
from google.colab import files
import time
```

لیست‌هایی به نام titles و bodies ساخته می‌شوند که به ترتیب شامل عنوان و بدنه‌ی متنی هر سند هستند. پس از آن، برای بررسی اولیه، پنج نمونه‌ی ابتدایی از بدنه متون چاپ می‌شود تا محتوای متنی فایل بررسی شود و اطمینان حاصل گردد که داده‌ها به درستی بارگذاری شده‌اند.

Open dataset file and extract Titles and Bodies

```
] with open(filename, 'r', encoding='utf-8') as f:
      data = json.load(f)

  # extract Titles and Bodies
  titles = [item['title'] for item in data]
  bodies = [item['body'] for item in data]


  # example of body
  for body in bodies[:5]:
      print(body)
```

در این بخش، ابتدا یک normalizer ساخته می‌شود تا حروف غیرمتعارف یا اشکالات نوشتاری اصلاح شوند. سپس WordTokenizer برای تبدیل جملات به لیستی از واژگان تعریف می‌شود. در ادامه، Stemmer برای حذف پسوندها و رساندن کلمات به ریشه‌شان وlemmatizer برای یافتن شکل صحیح و مرجعی از هر واژه ساخته می‌شوند. در نهایت، لیست stopwords از hazm گرفته شده و به مجموعه تبدیل می‌شود تا در مرحله‌ی بعدی فیلتر شوند.

Define normalization, tokenization, stemming, lemmatization and stop words

```
]  normalizer = Normalizer()
   tokenizer = WordTokenizer()
   stemmer = Stemmer()
   lemmatizer = Lemmatizer()
   stop_words = set(stopwords_list())
```

تابع preprocess وظیفه‌ی آماده‌سازی متون برای تحلیل و بردارسازی را دارد. ابتدا متن ورودی نرمال‌سازی می‌شود تا اشکالات نگارشی و املایی احتمالی برطرف شود. سپس متن به کلمات تقسیم می‌شود. در گام بعدی، تمامی کلمات پرتکرار و کم‌اهمیت که در لیست کلمات ایست قرار دارند، حذف می‌شوند. در نهایت، هر واژه با استفاده از ریشه‌یاب به ریشه تبدیل شده و با لماتایزر به شکل اصلی‌تر خود بازگردانده می‌شود. خروجی این تابع یک رشته‌ی مناسب برای مرحله‌ی بردارسازی است.

تابع preprocess روی تمام متون بدنه اجرا می‌شود و خروجی آن‌ها در لیستی به نام documents ذخیره می‌شود. این مرحله با tqdm نمایش داده می‌شود تا پیشرفت فرایند را ببینیم. سپس، از TfidfVectorizer برای ساخت ماتریس TF-IDF استفاده می‌شود. این ماتریس نشان می‌دهد که هر واژه در هر سند چقدر مهم است، با در نظر گرفتن کل مجموعه اسناد. پس از ساخت این ماتریس، برای انجام جستجوی سریع‌تر، یک اندیس FAISS با ساختار IndexFlatIP ایجاد می‌شود. این اندیس از ضرب داخلی برای محاسبه شباهت بین بردارها استفاده می‌کند. برای افزودن بردارها به اندیس، ابتدا ماتریس sparse به آرایه‌ی معمولی (dense) تبدیل می‌شود و سپس به FAISS اضافه می‌گردد. این مرحله اساس جستجوی برداری سریع در مراحل بعدی را فراهم می‌سازد.

Preprocessing function

rmalization, tokenization, stemming, and lemmatization

```
def preprocess(text):
    text = normalizer.normalize(text)
    tokens = tokenizer.tokenize(text)
    tokens = [t for t in tokens if t not in stop_words]
    tokens = [lemmatizer.lemmatize(stemmer.stem(t)) for t in tokens]
    return " ".join(tokens)
```

Preprocess, TF-vectorization and FAISS indexing for FAST Similarity search

```
documents = [preprocess(text) for text in tqdm(bodies)]

# TF-IDF vectorization
vectorizer = TfidfVectorizer()
tfidf_matrix = vectorizer.fit_transform(documents)

# FAISS Indexing for Fast Similarity Search
faiss_index = faiss.IndexFlatIP(tfidf_matrix.shape[1])
faiss_index.add(tfidf_matrix.toarray())
```

```
100%|██████████| 3780/3780 [00:40<00:00, 93.71it/s]
```

تابع normal_search از تابع cosine_similarity از کتابخانه‌ی sklearn برای محاسبه‌ی شباهت بین این بردار پرس‌وجو و تمامی بردارهای اسناد موجود در ماتریس tfidf_matrix استفاده می‌شود. خروجی این تابع یک آرایه از نمرات شباهت است که با استفاده از مرتب‌سازی اسناد دارای بالاترین شباهت انتخاب می‌شوند. این روش به‌رغم سادگی، دقت بالایی دارد و به‌ویژه برای دیتاست‌های کوچک یا متوسط بسیار مناسب و سریع است، اما با افزایش حجم داده ممکن است زمان‌بر شود.

راه حل برای سرچ سریع‌تر

تابع fast_search: ورودی این تابع یک کوئری است که ابتدا با استفاده از همان تابع preprocess پیش‌پردازش می‌شود. سپس متن پردازش‌شده با همان TfidfVectorizer که روی اسناد اعمال شده بود، به بردار TF-IDF تبدیل می‌گردد. بردار به دست‌آمده به اندیس FAISS داده می‌شود و FAISS با استفاده از شباهت ضرب داخلی یا معادل نرمال‌شده‌ی شباهت کسینوسی، اسناد مشابه را پیدا کرده و ایندکس آن‌ها را به همراه نمرات شباهت بازمی‌گرداند. خروجی نهایی این تابع لیستی شامل عنوان، بخشی از متن اصلی سند و نمره‌ی شباهت برای هر سند بازیابی‌شده است. **این روش**

بهینه و سریع است، به‌ویژه برای دیتاست‌های بزرگ، زیرا FAISS ساختاری مبتنی بر جستجوی برداری دارد.

Normal search

```
] def normal_search(query, top_k=5):
      query_vector = vectorizer.transform([preprocess(query)])
      similarities = cosine_similarity(query_vector, tfidf_matrix).flatten()
      top_indices = similarities.argsort()[-top_k:][::-1]
      return [(titles[i], bodies[i], similarities[i]) for i in top_indices]
```

Fast search function: uses vectorization & indexing

```
def fast_search(query, top_k=5):
    # vectorize the query after preprocessing
    query_vector = vectorizer.transform([preprocess(query)]).toarray()

    # search the FAISS index
    scores, indices = faiss_index.search(query_vector, top_k)

    # return top results
    return [(titles[i], bodies[i], scores[0][j]) for j, i in enumerate(indices[0])]
```

مثال) یک نمونه از اجرای سیستم طراحی برای کوئری "دلار":

```
results = fast_search("دلار", top_k=5)

# display results
for title, body, score in results:
    print(f"Title: {title}\nScore: {score:.4f}\nBody: {body[:200]}...\n{'='*50}\n")
```

```
Title: رئیس بانک مرکزی ایران در عراق در طلب بدهی‌های ابانباشتی میلیارد دلاری
Score: 0.3813
Body: برای پخش این فایل لطفا جاوا اسکریپت را فعال یا از یک مرورگر دیگر استفاده کنید.
رئیس بانک مرکزی ایران در عراق در طلب بدهی‌های ابانباشتی میلیارد دلاری

بانک مرکزی ایران اعلام کرده که از امروز برای جلوگیری...
==================================================

Title: رکوردهای تازه قیمت دلار در ایران؛ ادگشت اتهام مجلس به سوی دولت
Score: 0.3350
Body: برای پخش این فایل لطفا جاوا اسکریپت را فعال یا از یک مرورگر دیگر استفاده کنید.
رکوردهای تازه قیمت دلار در ایران، ادگشت اتهام مجلس به سوی دولت

قیمت دلار در بازار ایران به رقم بی‌سابقه ۲۲ هزار تومان رس...
==================================================

Title: آیا دلار به '۱۵ هزار تومان' برمی‌گردد؟
Score: 0.3153
Body:
هادی چاوشیروزنامه‌نگار


سخنان رئیس‌جمهور ایران درباره احتمال رسیدن قیمت دلار به ۱۵ هزار تومان، همزمان با کاهش قیمت ارزهای معتبر در بازار تهران طی روزهای اخیر، این سوال را برای بسیاری از شهروندان پیش...
==================================================

Title: بیت‌کوین قیمت بالای '۵۰ هزار دلار' را هم تجربه کرد
Score: 0.2887
Body:


براساس گزارش رویترز قیمت بیت‌کوین در معاملات امروز سه‌شنبه (۲۸ بهمن/ ۱۶ فوریه) برای اولین بار مرز ۵۰ هزار دلاری را ر رد کرد و رکورد جدیدی بر جای گذاشت.
این رمزارز که چند روزی به نرخ کمی کمتر از ۵۰ ...
==================================================
```

مقایسهٔ زمانی سرچ عادی و سریع:

Comparing query time

```
query = "اقتصاد ایران"

start = time.time()
_ = fast_search(query, top_k=5)
print(f"fast_search duration: {time.time() - start:.4f} seconds")

start = time.time()
_ = normal_search(query, top_k=5)
print(f"normal_search duration: {time.time() - start:.4f} seconds")
```

```
fast_search duration: 0.0657 seconds
normal_search duration: 0.0223 seconds
```

در بخش پایانی، برای ارزیابی عملکرد، مدت زمان اجرای هر دو روش اندازه‌گیری شد. برخلاف انتظار که معمولاً FAISS باید سریع‌تر عمل کند، در این آزمایش خاص، normal_search با زمان حدود ۰.۰۲۲ ثانیه سریع‌تر از fast_search با زمان حدود ۰.۰۶۶ ثانیه بود. دلیل این نتیجه، کوچک بودن اندازه‌ی دیتاست مورد استفاده بود (حدود ۴۰۰۰ رکورد)، زیرا در مجموعه‌های کوچک، سربار ناشی از تبدیل ماتریس Sparse به Dense و هزینه‌ی اضافی اندیس‌سازی FAISS بیشتر از مزیت سرعت آن است. به‌عبارت دیگر، FAISS برای **داده‌های بزرگ‌تر** (مثلاً بالای ۱۰ هزار سند) عملکرد بهتری دارد.

سوال دوم)

... همانند بخش قبل دیتاست را (که همان دیتاست قبلی است) آپلود و پیش‌پردازش می‌کنیم ...

در این بخش، یک فایل به نام "feedback_memory.pkl"به عنوان بانک حافظه برای ذخیره‌سازی بردارهای بهبود یافته‌ی کوئری‌ها استفاده شده است. اگر این فایل قبلاً وجود داشته باشد، محتوای آن با استفاده از pickle.load بارگذاری می‌شود تا اطلاعات قبلی حفظ شوند. در غیر این صورت، یک دیکشنری خالی به عنوان حافظه‌ی بازخورد ایجاد می‌شود. این ساختار کمک می‌کند تا نتایج جستجوی بهبودیافته در جلسات بعدی نیز مورد استفاده قرار گیرند.

```
FEEDBACK_FILE = "feedback_memory.pkl"

if os.path.exists(FEEDBACK_FILE):
    with open(FEEDBACK_FILE, "rb") as f:
        feedback_memory = pickle.load(f)
else:
    feedback_memory = {}
```

تابع apply_rocchio پیاده‌سازی الگوریتم معروف Rocchio برای بهبود بردار کوئری است. این الگوریتم با ترکیب بردار کوئری اولیه با میانگین بردارهای اسناد مرتبط و غیرمرتبط (بر اساس بازخورد کاربر)، یک بردار جدید ایجاد می‌کند که بازتاب بهتری از نیاز اطلاعاتی کاربر است. ضرایب alpha، beta، و gamma مشخص می‌کنند که هر بخش چقدر در بردار جدید تأثیرگذار باشد. خروجی این تابع، بردار نرمال‌شده‌ی جدیدی است که می‌توان از آن برای جستجوی دقیق‌تر استفاده کرد. مقادیر alpha، beta، و gamma بر حسب تجربه به ترتیب برابر ۱، ۰.۷۵ و ۰.۲۵ قرار داده شده اند.

```python
def apply_rocchio(query_vec, relevant_vecs, nonrelevant_vecs, alpha=1.0, beta=0.75, gamma=0.25):
    if len(relevant_vecs) > 0:
        relevant_centroid = np.mean(relevant_vecs, axis=0)
    else:
        relevant_centroid = 0

    if len(nonrelevant_vecs) > 0:
        nonrelevant_centroid = np.mean(nonrelevant_vecs, axis=0)
    else:
        nonrelevant_centroid = 0

    updated_vec = alpha * query_vec + beta * relevant_centroid - gamma * nonrelevant_centroid
    return normalize(updated_vec.reshape(1, -1))[0]
```

این تابع نتایج اولیه جستجو را به کاربر نمایش می‌دهد، سپس از او می‌خواهد که شماره‌ی اسناد مرتبط (relevant) را وارد کند. با توجه به ورودی کاربر، بردارهای اسناد مرتبط و غیرمرتبط جدا شده و در قالب دو لیست بازگردانده می‌شوند. این اطلاعات ورودی‌های مورد نیاز برای اجرای الگوریتم Rocchio در مرحله‌ی بعدی هستند.

```python
def get_user_feedback(results):
    print("\nInitial results:")
    for idx, (real_index, score) in enumerate(results):
        score_percent = round(score * 100, 2)
        print(f"[{idx}] Title: {titles[real_index]} ({score_percent}%)\n→ Body:{documents[real_index][:60]}...\n")

    input_str = input("Enter relevant result number (e.g. 1 3): ")
    relevant_indices = list(map(int, input_str.strip().split()))

    relevant_vecs, nonrelevant_vecs = [], []

    for idx, (real_index, score) in enumerate(results):
        vec = doc_vectors[real_index].toarray().astype(np.float32)[0]
        if idx in relevant_indices:
            relevant_vecs.append(vec)
        else:
            nonrelevant_vecs.append(vec)

    return relevant_vecs, nonrelevant_vecs
```

این تابع، ابتدا کوئری را از کاربر دریافت می‌کند. اگر کوئری قبلاً در حافظه بازخورد ثبت شده باشد، بردار ذخیره‌شده قبلی برای جستجو استفاده می‌شود و نتایج نمایش داده می‌شوند. اما اگر کوئری جدید باشد، ابتدا بردار آن ساخته و نرمال می‌شود. سپس با استفاده از اندیس FAISS جستجو انجام می‌شود و نتایج نمایش داده می‌شوند. بعد از آن، از کاربر خواسته می‌شود تا نتایج مرتبط را مشخص کند. سپس با استفاده از تابع apply_rocchio، بردار کوئری اصلاح می‌شود و این بردار جدید در حافظه ذخیره می‌شود تا در دفعات بعد بدون نیاز به بازخورد مجدد استفاده گردد.

```python
def run_feedback_search():
    query = input("Your query: ").strip()

    if query in feedback_memory:
        query_vec = feedback_memory[query]
        print("Using previous vector of this query...")
        D, I = index.search(query_vec.reshape(1, -1), 5)
        results = list(zip(I[0], D[0]))
        for idx, (real_index, score) in enumerate(results):
          score_percent = round(score * 100, 2)
          print(f"[{idx}] Title: {titles[real_index]} ({score_percent}%)\n→ Body: {documents[real_index][:60]}...\n")
        return

    else:
        query_vec = vectorizer.transform([query]).toarray().astype(np.float32)
        query_vec = normalize(query_vec)[0]

    # search
    D, I = index.search(query_vec.reshape(1, -1), 5)
    results = list(zip(I[0], D[0]))

    # show results and collect feedback
    relevant_vecs, nonrelevant_vecs = get_user_feedback(results)

    # update with Rocchio
    updated_vec = apply_rocchio(query_vec, relevant_vecs, nonrelevant_vecs)

    # save updated vector
    feedback_memory[query] = updated_vec
    with open(FEEDBACK_FILE, "wb") as f:
        pickle.dump(feedback_memory, f)

    print("Model updated.")
```

مثال ۱) کوئری "قرارداد"

```
run_feedback_search()
```

```
Your query: قرارداد

Initial results:
[0] Title: (%44.59) 'سند همکاری ایران و چین؛ 'شر در جزئیات نهفته است
→ Body:سیاو اردلانی‌پی‌من قضاو قرارداد ۲۵ ابر چین دامیده خارج چارچو...

[1] Title: (%38.77) کرسنت؛ عرامت '۶۰۰ میلیون دلاری' برای شرکت اماراتی در پرونده شکایت علیه ایران
→ Body:شرک « دانا گاز » امار اعلا هیئ داور بین‌الملل پرونده شکا شرک...

[2] Title: (%37.22) رابطه ایران و چین، موضوع کشمکش تازه سیاسی در ایران
→ Body:، فایل لطفا جاوا اسکریپ فعال مرورگر استفاده . رابطه ابر چین:...

[3] Title: (%28.99) تابستان جذاب فوتبال اروپا؛ ستاره‌هایی که آزاد می‌شوند
→ Body:باشگاه دوس بازیکن علاقه ارزان قیم بخرد ، اینکه دلیل شیوع کر...

[4] Title: (%27.28) کارگری خود را کنار چاه نفت هویزه کشت
→ Body:فایل لطفا جاوا اسکریپ فعال مرورگر استفاده . کارگر فعال مید ن...

Enter relevant result number (e.g. 1 3): 0 2
Model updated.
```

```
run_feedback_search()
```

```
Your query: قرارداد
Using previous vector of this query...
[0] Title: (%67.67) 'سند همکاری ایران و چین؛ 'شر در جزئیات نهفته است
→ Body: سیاو اردلانی‌پی‌من قضاو قرارداد ۲۵ ابر چین دامیده خارج چارچو...

[1] Title: (%63.05) رابطه ایران و چین، موضوع کشمکش تازه سیاسی در ایران
→ Body: ، فایل لطفا جاوا اسکریپ فعال مرورگر استفاده . رابطه ابر چین ...

[2] Title: (%38.31) دولت حامی و متقدان عصبانی؛ از توافق ۲۵ ساله ایران و چین چه درز کرده؟
→ Body: جنجال توافق ۲۵ ابر چین ادامه . دول تاکید موضوع مخف نمایندگ م...

[3] Title: (%35.48) سند همکاری ایران و چین؛ 'حکمت‌آمیز' یا 'قرارداد ترکمن‌چای'
→ Body: برنامه بلندمد همکاری ابر چین دول حسن روحان « سند راهبرد » مخ...

[4] Title: (%34.14) کرسنت؛ عرامت '۶۰۰ میلیون دلاری' برای شرکت اماراتی در پرونده شکایت علیه ایران
→ Body: شرک « دانا گاز » امار اعلا هیئ داور بین‌الملل پرونده شکا شرک...
```

مثال ۲) کوئری "فوتبال انگلیس"

```
run_feedback_search()
```

```
Your query: فوتبال انگلیس

Initial results:
[0] Title: (%48.89) پیش از کرونا چه چیزهایی فوتبال انگلیس را تعطیل کرد؟
→ Body:تماشاگر فوتبال انگلیس نقاط چه مشکل همگیر جهان ویروس کرونا ...

[1] Title: (%41.67) کرونا دنیای فوتبال دوستان را تیره و تار کرد؛ فوتبال انگلستان، آلمان و فرانسه تعلیق شد
→ Body:فایل لطفا جاوا اسکریپ فعال مرورگر استفاده . کرونا دنیا فوتبا...

[2] Title: (%38.78) تیم‌های یورو ۲۰۲۰: انگلیس با پیروزی مقابل دانمارک حریف ایتالیا در فینال شد
→ Body:ت مل فوتبال انگلیس ۵۵ سال انتظار توانس فینال تورنمن ( جا جها...

[3] Title: (%38.44) فینال یورو ۲۰۲۰؛ آرزوی بزرگ انگلیس و ایتالیا
→ Body:یویا عدالتیخبردگار دیدار #هست تورنمنت شیوع کرونا برگزار سال ...

[4] Title: (%35.06) یورو ۲۰۲۰؛ مسابقه با دانمارک پس از شب رویایی انگلیس در رم
→ Body:ت مل فوتبال انگلیس بازی تاریخ اوکراین گل شهر ر شکس . ت چهارش...

Enter relevant result number (e.g. 1 3): 2 3 4
Model updated.
```

```
run_feedback_search()
```

```
Your query: فوتبال انگلیس
Using previous vector of this query...
[0] Title: (%69.24) تیم‌های یورو ۲۰۲۰: انگلیس با پیروزی مقابل دانمارک حریف ایتالیا در فینال شد
→ Body: ت مل فوتبال انگلیس ۵۵ سال انتظار توانس فینال تورنمن ( جا جها...

[1] Title: (%67.46) فینال یورو ۲۰۲۰؛ آرزوی بزرگ انگلیس و ایتالیا
→ Body: یویا عدالتیخبردگار دیدار #هست تورنمنت شیوع کرونا برگزار سال ...

[2] Title: (%65.49) یورو ۲۰۲۰؛ مسابقه با دانمارک پس از شب رویایی انگلیس در رم
→ Body: ت مل فوتبال انگلیس بازی تاریخ اوکراین گل شهر ر شکس . ت چهارش...

[3] Title: (%57.04) تیم‌های یورو ۲۰۲۰؛ انگلیس-دانمارک: رویای ۶۶ یا رویای ۹۶؟
→ Body: عاقبگیرکننده تیم یورو ۲۰۲۰ ؛ انگلیس دانمارک باز تیمن یورو ...

[4] Title: (%46.9) قهرمانی ایتالیا در یورو ۲۰۲۰؛ ماجرایی روی ابرها
→ Body: ایتالیا در یورو ۲۰۲۰ قهرمان ایتالیا ...
```

5s completed at 3:46

همانطور که مشخص است، در هر دو مثال، پس از گرفتن کوئری و وارد کردن داکیومنت‌های مرتبط، در اجرای بعدی برای همان کوئری، نتایج انتخاب شده در صدر جدول نتایج و با امتیازات بیشتری نمایش داده می‌شوند. همچنین سایر نتایج هم بر حسب شباهت به فیدبکی که داده شد، امتیاز متفاوتی دریافت می‌کنند. من جوری پیاده‌سازی کردم که برای هر کوئری فقط یکبار بتوان فیدبک داد، اما می‌توان طوری پیاده‌سازی کرد که بتوان برای هر کوئری چند بار فیدبک گرفت.

سوال سوم)

در ابتدا کتابخانه‌های لازم برای پردازش زبان طبیعی، محاسبات فاصله و شباهت، و نرمال‌سازی صدا وارد می‌شوند. این شامل numpy برای عملیات برداری، spacyبرای بردارهای معنایی، difflib و Levenshtein برای فاصله‌ی ویرایشی، nltk برای n-grams، و jellyfish برای محاسبات صوتی است. سپس مدل پیش‌فرض انگلیسی en_core_web_md ازspaCy بارگذاری می‌شود که برای استخراج بردارهای متنی کاربرد دارد.

```
import numpy as np
import spacy
from difflib import SequenceMatcher
from Levenshtein import distance as edit_distance
from nltk.util import ngrams
from nltk.corpus import stopwords
from collections import Counter
import jellyfish
```

تابع kgram_score دو عبارت را با استفاده از k-grams با پیش‌فرض k=۲ مقایسه می‌کند. ابتدا برای هر عبارت مجموعه‌ای ازk-grams استخراج می‌شود و سپس ضریب شباهت Jaccard بین دو مجموعه محاسبه می‌گردد. این معیار حساس به ترتیب و نزدیکی حروف است و برای اصلاح املایی است.

```
def kgram_score(query, candidate, k=2):
    def get_kgrams(text):
        return set([''.join(gram) for gram in ngrams(text, k)])
    q_k = get_kgrams(query.replace(" ", ""))
    c_k = get_kgrams(candidate.replace(" ", ""))
    return len(q_k & c_k) / len(q_k | c_k)
```

تابع noisy_channel_score از فاصله‌ی ویرایشی Levenshtein استفاده می‌کند تا میزان تغییر لازم برای تبدیل عبارت query به candidate را بسنجد. سپس با تقسیم فاصله به طول عبارت بلندتر، نرمال‌سازی انجام می‌گیرد و در نهایت نمره به صورت normalized_distance – ۱محاسبه می‌شود.

```
def noisy_channel_score(query, candidate):
    return 1 - (edit_distance(query, candidate) / max(len(query), len(candidate)))
```

در این تابع با کمک الگوریتم Soundex از کتابخانه‌ی jellyfish ، دو عبارت بر اساس شباهت آوایی ارزیابی می‌شوند. اگر Soundex هر دو عبارت برابر باشد، نمره کامل ۱.۰ وگرنه ۰.۰ بازگردانده می‌شود. این معیار برای خطاهای ناشی از تلفظ است.

```
def sound_score(query, candidate):
    q_sound = jellyfish.soundex(query)
    c_sound = jellyfish.soundex(candidate)
    if q_sound == c_sound:
      return 1.0
    else:
      return 0.0
```

در این تابع، شباهت معنایی بین دو عبارت با استفاده از بردارهای زبانی spaCy سنجیده می‌شود. ابتدا بردارهای هر دو عبارت گرفته می‌شود و سپس با استفاده از ضرب داخلی نرمال‌شده(cosine similarity) ، شباهت معنایی بین آن‌ها محاسبه می‌گردد. اگر یکی از بردارها صفر باشد، نمره ۰.۰ برگردانده می‌شود.

```
def semantic_score(query, candidate):
    q_vec = nlp(query).vector
    c_vec = nlp(candidate).vector
    if np.linalg.norm(q_vec) == 0 or np.linalg.norm(c_vec) == 0:
        return 0.0
    return np.dot(q_vec, c_vec) / (np.linalg.norm(q_vec) * np.linalg.norm(c_vec))
```

تابع correct_query برای هر کاندید، چهار نمره‌ی بالا را محاسبه کرده و میانگین آن‌ها را به عنوان نمره نهایی تعیین می‌کند. سپس لیستی از کاندیدها با نمره نهایی مرتب شده (بیشترین به کمترین) برمی‌گرداند. هر عنصر خروجی شامل عبارت کاندید، نمره نهایی، و نمرات جزئی چهارگانه است.

```python
def correct_query(query):
    results = []
    for c in candidates:
        k = kgram_score(query, c)
        n = noisy_channel_score(query, c)
        s = sound_score(query, c)
        sem = semantic_score(query, c)
        final = (k + n + s + sem) / 4
        results.append((c, final, k, n, sem, s))
    results.sort(key=lambda x: x[1], reverse=True)
    return results
```

مثال ۱)

```python
query = "machin lernng"

candidates = [
    "machine learning",
    "deep learning",
    "reinforcement learning"
]

res = correct_query(query)

print(f"{'query':<25} | final score  | k-gram  | noise  | context  | sound")
print("-"*80)
for c, f, k, n, ctx, s in res:
    print(f"{c:<25} | {f:.4f}       | {k:.4f}  | {n:.4f} | {ctx:.4f}   | {s:.4f}")
```

```
query                     | final score  | k-gram  | noise  | context  | sound
--------------------------------------------------------------------------------
machine learning          | 0.5572       | 0.5000  | 0.8125 | -0.0838   | 1.0000
deep learning             | 0.1331       | 0.2222  | 0.3846 | -0.0745   | 0.0000
reinforcement learning    | 0.1283       | 0.1538  | 0.4091 | -0.0496   | 0.0000
```

عبارت غلط املایی "machin lernng" به عنوان ورودی داده شده و سه عبارت پیشنهادی برای تصحیح آن در لیست candidates قرار گرفته‌اند. تابع correct_query روی آن اعمال شده و خروجی به صورت جدول مرتب شده‌ای از نمره نهایی و هر چهار معیار چاپ شده است. نتیجه نشان می‌دهد که "machine learning" با نمره ۰.۵۵۷۲ بیشترین شباهت را داشته و به عنوان گزینه‌ی برتر انتخاب می‌شود.

مثال ۲)

```
query = "forrest gomp"

candidates = [
    "forrest gump",
    "forest gum",
    "forrest dump"
]

res = correct_query(query)

print(f"{'query':<25} | final score  | k-gram  | noise  | context  | sound")
print("-"*80)
for c, f, k, n, ctx, s in res:
    print(f"{c:<25} | {f:.4f}       | {k:.4f}  | {n:.4f} | {ctx:.4f}   | {s:.4f}")
```

```
query                     | final score  | k-gram  | noise  | context  | sound
--------------------------------------------------------------------------------
forrest gump              | 0.8533       | 0.6667  | 0.9167 | 0.8298   | 1.0000
forrest dump              | 0.7954       | 0.5385  | 0.8333 | 0.8098   | 1.0000
forest gum                | 0.5926       | 0.5000  | 0.7500 | 0.1203   | 1.0000
```