نام گروه

Tensor Titans

اعضای گروه

غزل عسکری

یاسمین صرافی

سپیده سلیمانیان

امیرحسین رجبی

محمدرضا ویلانی

۱- فاصله‌ی لونشتاین (Levenshtein distance) یکی از تکنیک‌های پرکاربرد در پردازش زبان طبیعی (NLP) برای سنجش میزان تفاوت بین دو رشته است. این فاصله، حداقل تعداد ویرایش‌هایی را که لازم است تا یک رشته به رشته‌ی دیگر تبدیل شود محاسبه می‌کند. سه نوع ویرایش اصلی در این روش وجود دارد:

- حذف یک کاراکتر
- درج یک کاراکتر
- جایگزینی یک کاراکتر

هر کدام از این عملیات یک واحد هزینه دارد و هدف این است که کمترین تعداد عملیات لازم برای تبدیل یک رشته به رشته‌ی دیگر مشخص شود.

کاربردها:

۱) تصحیح غلط‌های املایی: برای پیدا کردن نزدیک‌ترین کلمه صحیح به کلمه‌ای که به اشتباه تایپ شده است. مانند gboard
۲) مقایسه‌ی رشته‌ها: در مسائل مقایسه متون یا رشته‌ها برای اندازه‌گیری شباهت.
۳) تشخیص هویت: در برنامه‌های تطابق و مقایسه‌ی نام‌ها و عبارات.

الگوریتم های مشابه:

فاصله Damerau-Levenshtein : این فاصله مشابه فاصله Levenshtein است، اما اجازه جابجایی را هم به عنوان یک عملیات اضافی می دهد و آن را ۴ عملیات می کند.

فاصله همینگ: فقط می تواند برای رشته هایی با طول مساوی اعمال شود، از آن استفاده می شود که تعداد موقعیت هایی را که کاراکترهای مربوطه در آنها متفاوت است اندازه گیری می کند.

Jaro-Winkler Distance : مناسب برای تطابق نام‌ها و محاسبه شباهت بر اساس جابجایی و تطابق کاراکترها.

N-grams: استفاده از گروه‌های nتایی از کاراکترها یا کلمات برای مقایسه شباهت‌ها.

توضیح الگوریتم

کران بالا و پایین: اگر و فقط اگر دو رشته یکسان باشند، فاصله لونشتاین همیشه غیر منفی و صفر است. از آنجا که نیاز به تغییر کامل یک رشته به دیگری از طریق حذف یا درج دارد، امکان پذیرترین فاصله لونشتاین بین دو رشته با طول m و n است $\max(m, n)$.

```
def levenshteinRecursive(str1, str2, m, n):
      # str1 is empty
    if m == 0:
        return n
    # str2 is empty
    if n == 0:
        return m
    if str1[m - 1] == str2[n - 1]:
        return levenshteinRecursive(str1, str2, m - 1, n - 1)
    return 1 + min(
          # Insert
        levenshteinRecursive(str1, str2, m, n - 1),
        min(
              # Remove
            levenshteinRecursive(str1, str2, m - 1, n),
          # Replace
            levenshteinRecursive(str1, str2, m - 1, n - 1))
    )

# Drivers code
str1 = "kitten"
str2 = "sitting"
distance = levenshteinRecursive(str1, str2, len(str1), len(str2))
print("Levenshtein Distance:", distance)
```

Time complexity: O(3^(m+n))

Auxiliary complexity: O(m+n)

```
# Python program for the above approach
def levenshtein_two_matrix_rows(str1, str2):
      # Get the lengths of the input strings
      m = len(str1)
      n = len(str2)
      # Initialize two rows for dynamic programming
      prev_row = [j for j in range(n + 1)]
      curr_row = [0] * (n + 1)
      # Dynamic programming to fill the matrix
      for i in range(1, m + 1):
            # Initialize the first element of the current row
            curr_row[0] = i
            for j in range(1, n + 1):
                  if str1[i - 1] == str2[j - 1]:
                        # Characters match, no operation needed
                        curr_row[j] = prev_row[j - 1]
                  else:
                        # Choose the minimum cost operation
                        curr_row[j] = 1 + min(
                              curr_row[j - 1], # Insert
                              prev_row[j],       # Remove
                              prev_row[j - 1] # Replace
                        )
            # Update the previous row with the current row
            prev_row = curr_row.copy()

      # The final element in the last row contains the Levenshtein
distance
      return curr_row[n]
```

```
# Driver code
if __name__ == "__main__":
      # Example input strings
      str1 = "kitten"
      str2 = "sitting"

      # Function call to calculate Levenshtein distance
      distance = levenshtein_two_matrix_rows(str1, str2)

      # Print the result
      print("Levenshtein Distance:", distance)
```

Time complexity: O(m*n)
Auxiliary Space: O(n)

۲- جستجوی فازی یا Fuzzy Search یکی از تکنیک‌های قدرتمند برای یافتن نتایج مشابه در متون است که به خصوص در مواقعی که ورودی‌ها ممکن است دچار اشتباهات تایپی، تفاوت‌های کوچک یا حتی تغییرات معنایی باشند، کاربردی است. این الگوریتم‌ها می‌توانند به خوبی با خطاها و تغییرات جزئی در داده‌ها سازگار شوند.

جستجوی فازی در Elastic Search

Elastic Searchیکی از ابزارهای قدرتمند در جستجوی متن است و از الگوریتم‌های جستجوی فازی برای بهبود نتایج جستجو استفاده می‌کند. در Elastic Search ، جستجوی فازی به صورت زیر پیاده‌سازی می‌شود:

تطابق فازی:(Fuzzy Matching)

در Elastic Search ، جستجوی فازی با استفاده از Fuzzy Query انجام می‌شود که به کاربران این امکان را می‌دهد که در جستجوهای خود از خطاها و اشتباهات تایپی چشم‌پوشی کنند. این الگوریتم با استفاده از مفهوم Levenshtein Distance (فاصله ویرایشی) به مقایسه کلمات می‌پردازد. این فاصله نشان‌دهنده تعداد تغییرات (اضافه، حذف، تغییر) لازم برای تبدیل یک کلمه به کلمه دیگر است.

پارامترهای اصلی:

فاصله :Levenshtein تنظیم می‌کند که چقدر تفاوت مجاز است. به عنوان مثال، مقدار پیش‌فرض این فاصله معمولاً ۲ است، به این معنی که حداکثر دو تغییر برای تطابق مجاز است.

تنظیمات نمره‌دهی: این تنظیمات به شما امکان می‌دهند تا تأثیر تغییرات در نمره جستجو را تنظیم کنید.

مزایای استفاده:

سازگاری با خطاهای تایپی: کاربران ممکن است کلمات را اشتباه تایپ کنند یا از املای متفاوتی استفاده کنند و جستجوی فازی می‌تواند این اشتباهات را شناسایی کند.

پشتیبانی از تطابق نزدیک: حتی اگر کلمه جستجو شده دقیقاً با کلمات موجود در متن تطابق نداشته باشد، این الگوریتم می‌تواند نتایج نزدیک و مفیدی ارائه دهد.

محدودیت‌ها:

هزینه محاسباتی: جستجوی فازی ممکن است از نظر محاسباتی گران‌قیمت باشد، به خصوص در جستجوهای بزرگ و پیچیده.

کیفیت نتایج: در برخی موارد، نتایج ممکن است مرتبط نباشند و نیاز به تنظیمات دقیق‌تری داشته باشند.

```
# جستجوی فازی با تمامی پارامترها
query = {
    "query": {
        "fuzzy": {
            "text": {
                "value": "cool",
                "fuzziness": "AUTO",          # فازی بودن خودکار
                "prefix_length": 1,           # تعداد کاراکترهایی که باید بدون تغییر در نظر گرفته شوند
                "max_expansions": 50,         # حداکثر تعداد کلماتی که می‌توانند گسترش یابند
                "transpositions": True,       # در نظر گرفتن جابجایی‌های دو کاراکتر
                "boost": 1.0,                 # نمره‌دهی جستجو
                "rewrite": "constant_score"   # استراتژی بازنویسی
            }
        }
    }
}

resp = client.search(index="test-index", body=query)
print("Got {} hits:".format(resp["hits"]["total"]["value"]))
for hit in resp["hits"]["hits"]:
    print("{timestamp} {author} {text}".format(**hit["_source"]))
```