به نام خدا

پردازش زبان‌های طبیعی

دکتر احسان عسگری

مستندات تمرین دوم: تبدیل متن‌های فینگیلیش به فارسی

گروه ۶

علی نظری - ۹۹۱۰۲۴۰۱

مهدی لطفیان - ۹۹۱۰۵۶۸۹

سیدمحمدیوسف نجفی - ۹۹۱۰۲۳۶۱

شایان صالحی - ۹۹۱۰۵۵۶۱

بهار ۱۴۰۳

فهرست مطالب

مقدمه 3
مرحله ۱: تولید تست کیس‌های متنوع فینگلیش..... 5
مرحله ۲: کراول کردن داده برای مدل bigram و داشتن context..... 6
مرحله ۳: پیش‌پردازش کلمه‌های ویکی‌پدیا..... 8
مرحله ۴: نگاشتن مستقیم حرف‌های انگلیسی به فارسی 9
مرحله ۵: انجام پیش‌پردازش روی کلمه‌های به دست آمده 10
مرحله ۶: استفاده از دیکشنری فارسی برای انتخاب کلمات معنی‌دار..... 12
مرحله ۷: استفاده از مدل آماری برای یافتن کلمه درست..... 14
مرحله ۸: استفاده از jaccard similarity 15
مرحله ۹: استفاده از Embedding برای بهبود مدل..... 19
مرحله ۱۰: استفاده از bigrams برای تعیین بهترین جمله‌ها 21
مرحله ۱۱: بررسی تست کیس‌ها..... 23
مرحله ۱۲: ارتباط با سرور matrix و ربات چت opsdroid..... 24

مقدمه

فینگیلیش که با نام پینگلیش نیز شناخته می شود، یک تلاقی منحصر به فرد از زبان فارسی و حروف لاتین است. فینگیلیش که عمدتاً از افزایش استفاده از بسترهای ارتباطی دیجیتال در میان کاربران ایرانی نشات می گیرد، به فارسی زبانان اجازه می دهد تا از صفحه کلید استاندارد QWERTY استفاده کنند، بنابراین نیاز به روش های ورودی ویژه فارسی را دور می زند. این پدیده به دلیل فراگیر شدن صفحه کلیدهای انگلیسی در دستگاه های تلفن همراه و رایانه ها ایجاد شده است و زبان فینگیلیش را به یک جایگزین مناسب و در دسترس برای بسیاری از فارسی زبانان، به ویژه نسل جوان و کسانی که در خارج از کشور زندگی می کنند، تبدیل کرده است.

در سال‌های اخیر، فینگیلیش در میان پلتفرم‌های مختلف اینترنتی، از جمله رسانه‌های اجتماعی، برنامه‌های پیام‌رسان، و انجمن‌ها، افزایش قابل توجهی در محبوبیت داشته است. این استفاده گسترده، سادگی و در دسترس بودن آن را در ارتباطات دیجیتال غیررسمی و پرسرعت که جابجایی بین زبان ها می تواند دست و پا گیر باشد برجسته می کند. محبوبیت فینگیلیش گواهی بر ماهیت تطبیقی زبان و توانایی آن در تکامل در پاسخ به تغییرات تکنولوژیکی و اجتماعی است. با این حال، این سازگاری مجموعه ای از چالش ها و مفاهیم را برای حفظ و استفاده صحیح از زبان فارسی به همراه دارد.

ظهور فینگیلیش چالش‌های مشخصی را برای سیستم‌های پردازش زبان طبیعی (NLP) ایجاد می‌کند، که معمولاً برای کار با زبان‌ها و اسکریپت‌های خاص طراحی شده‌اند. نویسه‌گردانی تنوع و ابهام ایجاد می‌کند، زیرا هیچ سیستم استانداردی برای تبدیل صداهای فارسی به حروف لاتین وجود ندارد. این تنوع می تواند منجر به ناهماهنگی در املا و نحو شود و کار الگوریتم های NLP را که بر داده های ساختار یافته و قابل پیش بینی تکیه دارند، پیچیده کند. توسعه سیستم های قوی برای تبدیل دقیق فینگیلیش به زبان فارسی برای بهبود پردازش متن، قابلیت های جستجو و تجزیه و تحلیل داده ها برای محتوای فارسی زبان ضروری است.

توسعه یک سیستم موثر برای تبدیل زبان فینگیلیش به فارسی پیامدهای فنی و آموزشی عمیقی دارد. چنین سیستمی می‌تواند ابزارهای یادگیری زبان را تقویت کند و زبان‌آموزان را قادر می‌سازد تا به طور یکپارچه بین اسکریپت‌ها جابه‌جا شوند و مهارت خود را در زبان فارسی بهبود بخشند. همچنین می‌تواند پلت‌فرم‌های دیجیتال را تقویت کند و با ارائه ترجمه‌ها و ترجمه‌های دقیق، آن‌ها را برای فارسی زبانان جامع‌تر و کاربرپسندتر کند. در زمینه‌های آموزشی، این فناوری می‌تواند از تلاش‌های حفظ زبان پشتیبانی کند و درک عمیق‌تری از زبان فارسی را در میان نسل‌های جوان‌تر که ممکن است به استفاده از زبان فینگیلیش عادت دارند، تقویت کند.

در این پروژه ما به دنبال این هستیم که با استفاده از روش های پردازشی و آماری و مبتنی بر یادگیری، یک مدل کارامد برای تبدیل نویسه‌ی فینگیلیش به زبان فارسی طراحی کنیم و سپس این مدل را در ربات چت Opsdroid به مرحله استقرار(Deploy) برسانیم.

مرحله ۱: تولید تست کیس‌های متنوع فینگیلیش

در این بخش ابتدا با کمک موارد موجود در اینترنت به تولید تست کیس هایی که به نوشتار فینگیلیش باشند پرداختیم که ۲۲ تست کیس متنوع تولید شد تا بتوانیم انواع مختلف پیچیدگی های این تبدیل را درک کنیم و در مثال های خودمان مشاهده کنیم که نمونه های آن را در زیر مشاهده می‌کنیم:

```
salam halet chetore? tazegi ha che khbar? emroz miayi cafe berim? ba'desham mirim sham mikhorim. movafeghi?

sakhre navardi varzeshe sakhtie. vali az badansazi behtare. be nazaram samte badansazi naro bayad aval ghavi tar beshi.

miaid berim shomal kenare sahel ghadam bezanim?

tahghighatet be che natije i resid? tonesti jense monasebo peida koni? az digi kala ham check kon gheymat ha ro.

daneshgahe sanati sharif sale 1344 tasis shod va ta be hal hodode 57 sal ast ke mashghol be tarbiate niroye motekhases baraye keshvar va jame'e ye elmi jahani ast.

vaziat mali mardome iran dar hale hazer aslan khob nist va hame gereftar hastan va daran baraye zende mandan talashe ziadi mikonand.

tanhayi behtar az hamneshine bad ast

forsat ha mesle abr migozarand bayad ghadreshan ra bedanim

elon musk pishnahad dad har kas ke parchame america ra be payin bekeshad va parchame felestin ra bala bebarad yek safare yek tarafe ye ejbari be an keshvar bokonad.

sokhangoye nezam pezeshki migoyad feshare kar va dastmozde payeen dalile khodkoshi resident ha ast.

nasa tajrobe ye soghot be dakhele siahchale ra shabih sazi kard.

yek afsare niroye havayi amrica be zarbe golole police dar florida koshte shod.

khahare ronaldo dar defa azash toye instagram post gozashte.

raftam ghorube khorshid ro bebinam.

pitza napolitan ye sabke jadide pitza hast ke marbut be shahre napoli italiast.

sazmane afve beinolmelal gofte ast mamno'iat hejab dar makan haye omumi naghze hoghoghe zanane mosalman ast.

ghadimi tarin nemone ye rozhe lab mote'alegh be panj hezar sale pish dar iran peida shod.

binayi mashin yek ghabeliate computer baraye moshahede ye mohite piramon ast ke omdatan az yek ya chand durbine videoyi ba ghabeliate analog be digital va hamchenin yek sisteme tahlile
signal estefade mikonad.

dashtane zahere araste ba'ese jalbe tavajohe jame'e va sabte khatereye khob dar zehne anha mishe.

be mahze in ke az daneshgah kharej shodi be maman zang bezan montazereteh.
```

مرحله ۲: کراول کردن داده برای مدل bigram و داشتن context

در این بخش نیاز داشتیم تا داده بیشتری در اختیار داشته باشیم تا بتوانیم مدل bigram را استفاده کنیم و همچنین بتوانیم یک ایده‌ای از context و تاثیر آن روی نتیجه داشته باشیم.

برای اینکار از داده‌های ویکیپدیا استفاده کردیم و از ریپازیتوری wikipedia-crawler استفاده کردیم. به این شکل هم عمل کردیم که چندین موضوع اصلی را به عنوان شروع‌های مختلف در نظر گرفتیم و با شروع از هر کدام، تعدادی صفحه مشابه را crawl کردیم.

به عنوان نمونه تاپیک‌های موجود چیزهایی شبیه فیزیک، ورزش، کامپیوتر و ... بود و کد مربوط به این بخش هم خیلی ساده به شکل زیر است:

```
import os
from tqdm import tqdm

file1 = open('../topics.txt', 'r')
main_list = list()

while True:
    line = file1.readline()
    if not line:
        break
    line = line.strip()
    main_list.append(line)

file1.close()

for url in tqdm(main_list):
    os.system(f"python3 wikipedia-crawler.py {url} --articles=125 --interval=1")
```

که صرفا تاپیک به تاپیک جلو رفته‌ایم و 125 صفحه مشابه را پیدا کرده و با intervalهای یک ثانیه‌ای درخواست می‌زنیم و آن صفحه را crawl می‌کنیم که در نتیجه خروجی شبیه به زیر می‌شود:

تاریخ گرایش‌های گوناگونی دارد و هر تاریخ‌دان ممکن است در حوزه ویژه‌ای از تاریخ پژوهش کند.

آنان که سرگذشت خود را به یاد نسپارند محکوم به تکرار آن هستند.

جرج سانتایانا–

ه ناظر به رخدادهای گذشته و گاه معطوف به پژوهش و بررسی رویدادها است؛ بنابراین، هم به علم و هم به موضوع آن، تاریخ گفته می‌شود. برای تفکیک این دو مقوله، اصطلاحاً تاریخ را تاریخ و علم تاریخ را تاریخ می‌نامند

ی» در واقع کل اموری است که در حیات آدمی مؤثر است، نظیر امور اقتصادی، مذهبی، سیاسی، هنری، حقوقی، نظامی و علمی. پژوهشگرانی که دربارهٔ تاریخ می‌نویسند، تاریخ‌نگار نامیده می‌شوند. هرچند غالباً این رشتهٔ م
د، با این حال می‌توان آن را به عنوان پلی بین این دو شاخه تلقی کرد؛ زیرا روش‌های مطالعاتی مختلف آن از هر دو شاخه وام گرفته شده‌اند. تاریخ به عنوان یک رشتهٔ مطالعاتی دارای شاخه‌ها و گرایش‌های جانبی زیادی است

ین حال پژوهش‌های تاریخی تنها به این منابع محدود نمی‌شوند. به‌طور کلی، منابع دانشورانهٔ تاریخی را می‌توان به سه رده تقسیم کرد: منابع مکتوب، منابع منقول و منابع مادی. تاریخ‌نگاران اغلب از هر سه مورد استفاده می‌کنند

نقطه عطف تاریخ یکی از اصطلاحات رایج در جامعه برای اشاره به وقایع مهم و اثرگذار در روند تاریخ و وقایع پیشرو هست.

تعریب و معرب بودن واژهٔ تاریخ باید توجه داشت که کمترین شباهت آوایی و قرابت واکه‌ای یا هم‌آهنگی واکه‌ای، تشابه صرفی یا تکواژی بین «ماه روز» و «تاریخ» وجود ندارد و به همین دلیل، آن را تعرب غریبی می‌دانند

نوان وحشی» و ابن منظور از «أَرَخَ» و «وَرخَ» دانسته‌است. اصمعی هم آورده که قیسیان و تمیمیان هر دو برای تعیین زمان شکل‌هایی از این واژه را به‌کار برده‌اند و این حاکی از آن است که تاریخ واژه‌ای عربی است

مرحله ۳: پیش‌پردازش کلمه‌های ویکی‌پدیا

ما از آن کلمه‌های ویکی‌پدیا فارسی که در اختیارمان گذاشته بودید، استفاده کردیم ولی خب در این دیتاست همه نوه به عنوان نمونه صرف فعل‌ها نیامده بوده و برای همین به شکل مستقیم نمی‌توانستیم ازشان استفاده کنیم و برای همین اول یک سری پیش‌پردازش‌ها انجام دادیم که مهم‌ترین آنها، lemmatize کردن است و برای اینکار هم از hazm استفاده کردیم.

```python
def process_text_file(file_path):
    lemmatized_freq = defaultdict(int)

    with open(file_path, 'r', encoding='utf-8') as file:
        for line in file:
            parts = line.strip().split()
            word = parts[0]
            freq = int(parts[1])

            lemma = lemmatizer.lemmatize(word).split('#')[0]

            lemmatized_freq[lemma] += freq

    return lemmatized_freq
```

بخش اصلی کد برای اینکار هم این بخش است که هر کلمه داخل دیتاست را صرفا lemmatize کردیم.

سپس می‌خواستیم مانند فرمت همان فایلی که دادید باشد و برای همین این کد را زدیم:

```python
def write_frequencies_to_file(frequencies, output_file):
    sorted_frequencies = sorted(frequencies.items(), key=lambda item: item[1], reverse=True)

    with open(output_file, 'w', encoding='utf-8') as file:
        for word, freq in sorted_frequencies:
            file.write(f"{word}\t{freq}\n")
    print(f"Data written to {output_file}")
```

مرحله ۴: نگاشتن مستقیم حرف‌های انگلیسی به فارسی

در این مرحله حروف صدادار و بی‌صدایی که در نوشتار فینگیلیش می‌آیند و ممکن است به یک یا چند حرف فارسی مپ شوند را همگی در نظر می‌گیریم به این شکل که به ازای هر حرف صدا دار یا بی صدا یا ترکیب حروفی که با هم تلفظ می‌شوند مثلا sh که صدای "ش" می‌دهد را هم در این بخش پوشش دادیم که بخشی از آن را در زیر می‌بینید که حروف به unicode آن‌ها در UTF-8 نگاشت شده اند.

```
    "x": [
        "\u0627\u06a9\u0633"
    ],
    "y": [
        "\u0648\u0627\u06cc"
    ],
    "z": [
        "\u0632\u062f",
        "\u0632\u06cc"
    ],
    "kh": [
        "\u062e"
    ],
    "gh": [
        "\u0642",
        "\u063a"
    ],
    "ch": [
        "\u0686"
    ],
    "sh": [
        "\u0634"
    ],
    "zh": [
        "\u0698"
    ],
    "ph": [
        "\u0641"
    ],
    "th": [
        "\u062b"
    ],
    "oo": [
        "\u0627\u0648",
        "\u0639\u0648"
    ],
    "ou": [
        "\u0627\u0648",
        "\u0639\u0648"
    ]
```

مرحله ۵: انجام پیش‌پردازش روی کلمه‌های به دست آمده

در گام نخست با توجه به مپینگ‌های به دست آمده، باید همه حالت‌های موجود را به دست بیاوریم و از حالت پیش‌پردازش شده دیتاست هم استفاده می‌کنیم:

```
10  def find_all_words_possible(word, first_letter_to_harf_mapping, letter_to_harf_mapping):
11      converted_words = []
12      j = 1
13      if len(word) > 1 and word[0: 2] in first_letter_to_harf_mapping:
14          j += 1
15          for letters in first_letter_to_harf_mapping[word[0: 2]]:
16              converted_words.append(letters)
17      else:
18          for letters in first_letter_to_harf_mapping[word[0]]:
19              converted_words.append(letters)
20
21      while j < len(word):
22          if word[j] == word[j - 1]:
23              j += 1
24              continue
25          new_converted_words = []
26          if j + 1 < len(word) and word[j: j + 2] in letter_to_harf_mapping:
27              for converted_word in converted_words:
28                  for letters in letter_to_harf_mapping[word[j: j + 2]]:
29                      new_converted_words.append(converted_word + letters)
30              j += 1
31          else:
32              for converted_word in converted_words:
33                  for letters in letter_to_harf_mapping[word[j]]:
34                      new_converted_words.append(converted_word + letters)
35          converted_words = new_converted_words
36          j += 1
37      return converted_words
```

این کد تابعی به نام `find_all_words_possible` را تعریف می‌کند که وظیفه دارد تمام کلمات ممکن را که می‌توان از یک کلمه مشخص با استفاده از نگاشت‌های مشخص شده حروف به حروف فارسی تولید کند. ورودی‌های تابع به شکل زیر است:

1. word: کلمه‌ای که می‌خواهیم تبدیل‌های ممکن از آن را پیدا کنیم.
2. first_letter_to_harf_mapping: یک دیکشنری که نگاشت‌های حروف اول به حروف فارسی را نگه می‌دارد.
3. letter_to_harf_mapping: یک دیکشنری که نگاشت‌های دیگر حروف به حروف فارسی را نگه می‌دارد.

converted_words به عنوان لیست خروجی حروف تبدیل شده، تعریف می‌شود.

حلقه اول بررسی می‌کند که آیا طول کلمه ورودی بیش از 1 است و آیا دو حرف اول کلمه در نگاشت first_letter_to_harf_mapping وجود دارد یا خیر. در این صورت، حروف نگاشت شده مربوط به دو

حرف اول به لیست converted_words اضافه می‌شوند و در غیر این صورت، فقط حروف نگاشت شده مربوط به حرف اول کلمه به converted_words اضافه می‌شوند.
سپس یک حلقه while برای پیمایش در بقیه کلمه ورودی هست. اگر حرف فعلی با حرف قبلی برابر باشد، از آن می‌گذرد (این مورد را نادیده می‌گیرد). همچنین یک لیست جدید به نام new_converted_words برای نگه‌داشتن ترکیب‌های جدید ایجاد می‌شود. سپس بررسی می‌کند که آیا دو حرف فعلی در نگاشت letter_to_harf_mapping وجود دارند یا خیر که در صورت وجود، برای هر ترکیب موجود در converted_words، حروف جدید اضافه می‌شوند و در غیر این صورت، فقط حرف فعلی تبدیل و به ترکیب‌های موجود اضافه می‌شود. و در نهایت لیست converted_words با ترکیب‌های جدید جایگزین می‌شود. در نهایت هم لیست converted_words به عنوان خروجی برگردانده می‌شود.

مرحله ۶: استفاده از دیکشنری فارسی برای انتخاب کلمات معنی‌دار

در ادامه نیاز است که با استفاده از دیکشنری کلماتی که داریم، کلمه‌های معنی‌دار را استخراج کنیم:

```
def sort_meaningful_words(words, words_tf):
    lemmatizer = Lemmatizer()
    new_words = []
    for word in words:
        lemmed_word = lemmatizer.lemmatize(word).split('#')[0]
        if lemmed_word in words_tf:
            new_words.append((words_tf[lemmed_word], word))
    new_words.sort(reverse=True)
    return new_words


def load_dictionary(dict_path):
    words_tf = {}

    with open(dict_path, 'r', encoding='utf-8') as txt_file:
        for line in txt_file:
            parts = line.split('\t')
            words_tf[parts[0]] = int(parts[1])

    return words_tf


def translate_puncs(punc):
    if punc == ',':
        return '،'
    elif punc == ';':
        return '؛'
    elif punc == '?':
        return '؟'
    return punc
```

تابع sort_meaningful_words:

این تابع کلماتی را که در یک لیست داده شده وجود دارند و معنی‌دار هستند را مرتب می‌کند.

یک نمونه از کلاس Lemmatizer ایجاد می‌شود و سپس لیستی به نام new_words برای ذخیره کلمات معنی‌دار تشکیل می‌شود. برای هر کلمه در words، کلمه ریشه‌یابی می‌شود و اولین قسمت آن که توسط # جدا شده است، گرفته می‌شود. اگر این کلمه در words_tf وجود داشت، آن را به همراه

مقدار فراوانی‌اش به لیست new_words اضافه می‌کند و در نهایت لیست new_words به صورت نزولی مرتب می‌شود.

تابع load_dictionary:

این تابع یک دیکشنری را از یک فایل متنی بارگذاری می‌کند.

دیکشنری‌ای به نام words_tf برای نگه‌داری کلمات و فراوانی آن‌ها ایجاد می‌شود. فایل متنی با استفاده از مسیر داده شده باز می‌شود. برای هر خط از فایل، خط به قسمت‌های جدا شده توسط t\ تقسیم می‌شود. کلمه و مقدار فراوانی آن استخراج و در دیکشنری words_tf ذخیره می‌شوند.

تابع translate_puncs:

این تابع برخی از علائم نگارشی انگلیسی را به معادل‌های فارسی آن‌ها ترجمه می‌کند.

اگر punc برابر , باشد، معادل فارسی آن ، برگردانده می‌شود.

اگر punc برابر ; باشد، معادل فارسی آن ؛ برگردانده می‌شود.

اگر punc برابر ? باشد، معادل فارسی آن ؟ برگردانده می‌شود.

در غیر این صورت، خود punc برگردانده می‌شود.

مرحله ۷: استفاده از مدل آماری برای یافتن کلمه درست

در نهایت هم کدی هست که از همه موارد استفاده می‌کند و کلمه به کلمه و با در نظر گرفتن تعداد تکرار کلمه، بهترین حالت کلمه را تشخیص می‌دهد:

```
def find_all_sentences_sorted(words, first_letter_to_harf_mapping, letter_to_harf_mapping, words_tf):
    converted_sentences = [(1, '')]
    sentence_len = len(words)
    for word in words:
        new_converted_sentences = []
        if word in string.punctuation:
            for converted_sentence in converted_sentences:
                converted_word = translate_puncs(word)
                new_converted_sentences.append((converted_sentence[0], converted_sentence[1].strip() + converted_word))
        else:
            converted_words = find_all_words_possible(word, first_letter_to_harf_mapping, letter_to_harf_mapping)
            converted_words = sort_meaningful_words(converted_words, words_tf)
            for converted_sentence in converted_sentences:
                for converted_word in converted_words:
                    new_converted_sentences.append((converted_sentence[0] * (converted_word[0] ** (1 / sentence_len)),
                                                    converted_sentence[1].strip() + ' ' + converted_word[1]))
        converted_sentences = new_converted_sentences
    return list(sorted(converted_sentences, reverse=True))
```

این تابع جملات ممکن را از یک لیست از کلمات تولید و بر اساس فراوانی کلمات مرتب می‌کند.

پارامترهای ورودی:

words: لیستی از کلمات.

first_letter_to_harf_mapping: دیکشنری‌ای که نگاشت‌های حروف اول به حروف فارسی را نگه می‌دارد.

letter_to_harf_mapping: دیکشنری‌ای که نگاشت‌های دیگر حروف به حروف فارسی را نگه می‌دارد.

words_tf: دیکشنری‌ای که فراوانی کلمات را نگه می‌دارد.

لیست converted_sentences به عنوان لیست جملات تبدیل شده با مقدار اولیه‌ای شامل یک تاپل (1, '') (یک جمله خالی با وزن 1) تعریف می‌شود. برای هر کلمه در words، لیستی به نام new_converted_sentences برای نگه‌داری جملات تبدیل شده جدید ایجاد می‌شود. اگر کلمه یک علامت نگارشی باشد، برای هر جمله تبدیل شده در converted_sentences، علامت نگارشی با استفاده از تابع translate_puncs به معادل فارسی آن تبدیل می‌شود و سپس جمله تبدیل شده جدید به لیست new_converted_sentences اضافه می‌شود. اگر هم کلمه یک علامت نگارشی نباشد، کلمات ممکن با استفاده از تابع find_all_words_possible پیدا می‌شوند. این کلمات با استفاده از تابع sort_meaningful_words بر اساس فراوانی مرتب می‌شوند.

مرحله ۸: استفاده از jaccard similarity

در این بخش با استفاده از دیتاست کلمه‌ها می‌خواهیم از jaccard similarity استفاده کنیم تا اشتباه‌های املایی و نحوی را تشخیص دهیم و آنها را اصلاح کنیم.

```
all_documents = list()

file1 = open('persian-wikipedia.txt', 'r')

while True:
    line = file1.readline()
    if not line:
        break
    line = line.strip()
    if not line.startswith("#"):
        all_documents.append(line.split())

file1.close()
```

نخست با استفاده از این کد کل کلمه‌ها و تعداد تکرارشان را استخراج می‌کنیم و در یک لیست نگه می‌داریم. سپس یک کلاس SpellCorrection داریم که به این شکل داده‌های ورودی را پردازش می‌کند:

```python
def shingling_and_counting(self, all_documents):
    all_shingled_words = dict()
    word_counter = dict()

    for doc in tqdm(all_documents):
        word = doc[0]
        count = int(doc[1])
        if not word in all_shingled_words.keys():
            new_word = f"${word}$"
            shingles = self.shingle_word(new_word)
            all_shingled_words[word] = shingles
        if not word in word_counter:
            word_counter[word] = count

    return all_shingled_words, word_counter
```

در این بخش، هر کلمه و تعداد تکرار آن را داریم و همه را به دو دیکشنری وارد می‌کنیم و همانطور که می‌دانیم برای اینکه شروع و پایان را در این shingle کردن مشخص کنیم، آن کلمه را بین $ قرار می‌دهیم و سپس آن را shingle می‌کنیم و تعداد تکرار را هم که داریم.

```python
def shingle_word(self, word, k=2):
    shingles = set()
    for i in range(len(word) - k + 1):
        shingle = word[i:i + k]
        shingles.add(shingle)
    return shingles
```

خود shingle کردن هم که به این شکل انجام می‌شود و مثلا k برابر با 2 باشد، دو حرف دو حرف با همپوشانی جلو می‌رویم و همه را به شکل یک set در پایتون نگه می‌داریم.

```python
def jaccard_score(self, first_set, second_set):
    return len(first_set.intersection(second_set)) / len(first_set.union(second_set))
```

خود محاسبه jaccard similarity هم به این شکل است و روی set دو کلمه تعداد مجموعه اشتراک را تقسیم بر تعداد مجموعه اجتماع می‌کنیم.

```python
def spell_check(self, query):
    final_result = []
    query = query.split()
    for word in query:
        if word in self.all_shingled_words.keys():
            final_result.append(word)
        else:
            correct_words = self.find_nearest_words(word)
            final_result.append(correct_words[0])
    return " ".join(final_result)
```

وقتی هم که می‌خواهیم بررسی روی یک جمله انجام دهیم، اگر آن کلمه در دیتاست ما بود که نیاز به اصلاح ندارد ولی اگر نبود، باید اصلاح شود و با jaccard similarity هم اینکار انجام می‌شود. برای پیدا کردن نزدیک‌ترین کاندید هم:

```python
def find_nearest_words(self, word):
    scores = dict()
    shingled_word = self.shingle_word(f"${word}$")
    for dict_word in self.all_shingled_words.keys():
        score = self.jaccard_score(shingled_word, self.all_shingled_words[dict_word])
        scores[dict_word] = score
    scores = dict(sorted(scores.items(), key=lambda item: item[1], reverse=True))
    top5 = list()
    for word in scores.keys():
        if len(top5) == 5 or scores[word] == 0:
            break
        top5.append(word)
    top5_counts = list()
    for word in top5:
        top5_counts.append(self.word_counter[word])
    max_value = max(top5_counts)
    min_value = min(top5_counts)
    for i, count in enumerate(top5_counts):
        top5_counts[i] = ((count - min_value + 1) / (max_value - min_value + 1))
    final_dict = dict()
    for i in range(len(top5)):
        final_dict[top5[i]] = scores[top5[i]] * top5_counts[i]
    final_dict = dict(sorted(final_dict.items(), key=lambda item: item[1], reverse=True))
    return list(final_dict.keys())
```

در واقع score آن کلمه ورودی با تک به تک کلمه‌های دیتاست را در نظر می‌گیریم و حساب می‌کنیم. سپس تعداد تکرار آن کاندیدها را هم در نظر می‌گیریم و به شکلی در انتخاب نهایی تاثیر می‌دهیم و در

نهایت آن کاندیدی که به نظر بیشترین شانس را دارد را انتخاب می‌کنیم و بر می‌گردانیم. روی یک نمونه جدای از ماژول اصلی هم به این شکل عمل می‌کند:

```
obj.spell_check("ظارک برویم که گفقتم من")

'من گفتم که برویم پارک'
```

که همانطور که می‌بینیم، گفقتم و ظارک اشتباه است و با کمک دیتاست به درستی، درست شده‌اند. ولی از این ماژول در ادامه ماژول‌های قبلی استفاده می‌شود و نه به شکل جداگانه.

مرحله ۹: استفاده از Embedding برای بهبود مدل

پس از بررسی مدل آماری تمام جملات ممکن براساس دیکشنری که داشتیم، حال ۱۰ جمله با پرتکرارترین توکن‌ها را انتخاب می‌کنیم و برای انتخاب بهترین جمله از لحاظ معنایی و غلط املایی از مدل Embedding-base از قبل train شده ParsBERT استفاده می‌کنیم.

```
tokenizer = BertTokenizer.from_pretrained('HooshvareLab/
bert-base-parsbert-uncased')
model = BertModel.from_pretrained('HooshvareLab/bert-base-parsbert-uncased')
```

پس از آن تمامی جملات را گرفته و به وسیله tokenizer توکن‌بندی کرده و به هر جمله یک معیار similarity اختصاص می‌دهیم به طوری که تمامی توکن‌های یک جمله رو به هم مرتبط کند.

```
def get_sentence_embedding(sentence):
    inputs = tokenizer(sentence, return_tensors='pt', padding=True,
    truncation=True, max_length=512)
    outputs = model(**inputs)
    return outputs.last_hidden_state.mean(dim=1).detach().numpy()

def return_best_sentence(sentences):
    top_sentences = [s[1] for s in sentences[:10]]
    embeddings = [get_sentence_embedding(sentence) for sentence in
    top_sentences]

    similarities = cosine_similarity(np.vstack(embeddings))

    max_sim_score = -1
    best_sentence = ''
    for i in range(len(similarities)):
        for j in range(i + 1, len(similarities[i])):
            if similarities[i][j] > max_sim_score:
                max_sim_score = similarities[i][j]
                best_sentence = top_sentences[i]

    return best_sentence
```

دلیل این کار به این خاطر است که هر جمله از لحاظ معنایی زمانی که پیوستگی معنایی بیشتری داشته باشد احتمالا جمله درست‌تری بوده و غلط نگارشی و یا کلمه نامربوطه ندارد.

برای این منظور در اینجا ما از معیار cosine similarity بهره برده‌ایم، به طوری که تمامی بردارهای فضای embedding توکن‌های مختلف را بایکدیگر cosine گرفته و یک مقدار خروجی به ما برمی‌گرداند. هر چه قدر این مقدار بیشتر باشد این معنا را به ما می‌دهد که توکن‌های یک جمله از لحاظ معنایی به یکدیگر نزدیک‌تر بوده و جمله صحیح‌تر و رساتری از لحاظ نگارشی است.

دلیل استفاده از این مدل به جای بالا آوردن یک مدل learning به این خاطر بود که زمان محاسبه تمامی embeddingها و cosine similarity در حد معقولی بوده، در صورتی در موردهای دیگر ممکن بود جواب را پس از طی یک زمان طولانی به ما برگرداند. همچنین این embeddingها را صرفا از مدل ParsBERT گرفته‌ایم و نیاز به train کردن یک مدل کاملا جدید نداشته‌ایم.

در خصوص آنکه چرا برای ۱۰ جمله پرتکرار این عمل را انجام داده‌ایم باید گفت که اگر می‌خواستیم برای همه آنها انجام دهیم حساب کردن تمامی این cosine similarityها زمان زیادی از ما می‌برد، و در صورتی که صرفا بر مدل آماری تکیه می‌کردیم به جملاتی می‌رسیدیم که شاید جمع تکرار تمامی توکن‌های آنها بیشترین بود اما لزوما پیوستگی معنایی در آن وجود نداشت. از این جهت این مدل در جهت بهبود و مکمل مدل آماری است.

مرحله ۱۰: استفاده از bigrams برای تعیین بهترین جمله‌ها

در این بخش از داده‌های خزش شده از ویکی‌پدیا استفاده کردیم. در ابتدا به کمک قطعه کد زیر و جملات خام خزش شده، یک فایل شامل ترکیبات دو کلمه‌ای فارسی و تعداد تکرار آن‌ها ایجاد کردیم. قابل توجه است که به کلمات اول و آخر هر جمله علامت $ اضافه شده تا مشخص شود که احتمال وقوع این کلمه در ابتدا یا انتهای جمله وجود دارد.

```
def create_biwords(biwords, tokens):
    """
    Creates biwords from a list of tokens.
    Returns a list of biwords.
    """
    for i in range(len(tokens) - 1):
        biword = tokens[i] + " " + tokens[i + 1]
        biwords.append(biword)
    return biwords

def create_biword_counts(biwords):
    """
    Creates a dictionary of biword counts.
    Returns a dictionary where keys are biwords and values are their counts.
    """
    biword_counts = {}
    for biword in biwords:
        if biword in biword_counts:
            biword_counts[biword] += 1
        else:
            biword_counts[biword] = 1
    return biword_counts


cleaned_persian = remove_numbers(remove_punctuation(str_wiki))

sentences = cleaned_persian.split('.')


biwords = []
for sentence in sentences:
    tokens = sentence.split()
    tokens.insert(0, '$')
    tokens.append('$')
    biwords = create_biwords(biwords, tokens)

biword_counts = create_biword_counts(biwords)
```

در ادامه جملات برتر انتخاب شده توسط مدل آماری را به قطعه کدی داده تا آن‌ها را به صورت bigrams با فرمت بالا (اضافه کردن علامت $ در ابتدا و انتهای جمله) درآورده و سپس امتیاز قبلی جمله را در تعداد تکرار bigramهای جمله که در دیتاست موجود است، ضرب کند. به این ترتیب امتیاز جملاتی که کلمات آن‌ها بیشتر کنار هم دیده شده‌اند زیاد شده و به نوعی مدل نسبت به معنا و دستور زبان جمله حساس تر می‌شود.

```
def consider_biwords(checking_sentences):
    checked_sentences = []
    for sentence in checking_sentences:
        cleaned_sentence = remove_numbers(remove_punctuation(sentence[1]))
        tokens = cleaned_sentence.split()
        tokens.insert(0, '$')
        tokens.append('$')
        biwords = create_biwords(tokens)
        biword_value = 1
        for biword in biwords:
            biword_value *= int(biword_counts[biword])
        checked_sentences.append((biword_value * sentence[0],  sentence[1]))
    return list(sorted(checked_sentences, reverse=True))

#example
consider_biwords([(25, 'باید بدانیم'), (25, 'باید بدنیم')])
```

در نهایت جملات بر اساس امتیازهای جدیدی که به دست‌آورده‌اند مرتب شده و جمله با بیشترین احتمال در بالاترین اولویت قرار می‌گیرد. برای مثال در example بالا امکان وجود بدانیم در آخر جمله (به همراه $) بسیار بیش‌تر از وجود بدنیم بوده و پس از اعمال امتیاز bigramها امتیاز این جمله که همان متناسب با احتمال درست بودن آن نیز می‌باشد بسیار بالاتر می‌رود.

قابل توجه است که عملکرد مناسب‌تر این مدل نیازمند استفاده از دیتاست بسیار بزرگ از biwordهای زبان فارسی در موضوعات و لحن‌ها متفاوت و متنوع است که متاسفانه در حال حاضر موجود نیست. اما همین دیتاست کوچک ایجاد شده نیز باعث بهبودهای محسوسی در نتایج به‌دست آمده توسط مدل شده‌است.

مرحله ۱۱: بررسی تست کیس‌ها

حال تعدادی از نتایج تبدیل تست کیس ها با استفاده از مدل خودمان را در زیر قرار می‌دهیم:

```
salam halet chetore? tazegi ha cheh khbar? emroz berim sham mikhorim. movafeghi?

miaid berim shomal kenare sahel ghadam bezanim?

daneshgahe sanati sharif sale 1344 tasis shod.

vaziat mali mardome iran dar hale hazer aslan khob nist.

tanhayi behtar az hamneshine bad ast

forsat ha mesle abr migozarand bayad ghadreshan ra bedanim

sokhangoye nezam pezeshki migoyad feshare kar va dastmozde payeen dalile khodkoshi parastar ha ast.

tajrobe ye soghot be dakhele siahchale ra shabih sazi kard.

yek afsare niroye havayi amrika be zarbe golole polis dar florida koshte shod.

raftam ghorube khorshid ro bebinam.

ghadimi tarin nemone ye rozhe lab mote'alegh be panj hezar sale pish dar iran peida shod.

binayi mashin yek ghabeliate computer baraye moshahede ye mohite piramon ast.

be mahze in ke az daneshgah kharej shodi be maman zang bezan.

estefade az selahe mosalsal dar jang jenayate jangi ast.

sakhtemane pastor dar tehran mahale esteghrar dolat ast.

aramgah ferdosi dar shiraz nist

kharid  ro anjam bedeh

goshi jadid kharidam choon baatry ghabli moshkel peida kardeh bud

perspolis bazi seh hich bakhteh ra dar payan bord

nooshidane ab gheir ashamidani mojeb masmoom shodan mishavad
```

نتایج:

سالم حالت چطور؟ تازگی ها چه خبر؟ امروز بریم شام میخوریم. موفقی؟

میاید بریم شمال کنار ساحل قدم بزنیم؟

دانشگاه صنعتی شریف سال ۱۳۴ تاسیس شد.

وازیت ملی مردم ایران در حال حاضر اصلان خوب نیست.

تنهایی بهتر از همنشین بعد است

فرصت ها مثل ابر میگذارند باید قدرشان را بدانیم

سخنگوی نظام پزشکی میگوید فشار کار و دستمزد پین دلیل خودکشی پارستر ها است.

تجرب ی سقوط ب داخل سیاهچال را شبیه سازی کرد.

یک افسر نیروی هوایی آمریکا ب ضرب گلول پلیس در فلوریدا کشت شد.

رفتم غروب خورشید رو ببینم.

قدیمی ترین نمون ی رژ لب متعلق ب پنج هزار سال پیش در ایران پیدا شد.

بینایی ماشین یک قابلیت کومپوتر برای مشهد ی محیط پیرامون است.

ب محض این ک از دانشگاه خارج شدی ب مامان زنگ بزن.

استفاد از سلاح مسلسل در جنگ جنایت جنگی است.

ساختمان پاستور در تهران محل استقرار دولت است.

آرامگاه فردوسی در شیراز نیست

خرید رو انجام بده

گوشی جدید خریدم چون باتری قبلی مشکل پیدا کرده بود

پرسپولیس بازی سه هیچ باخته را در پایان برد

نوشیدن آب غیر آشامیدنی موجب مسموم شدن میشود

مرحله ۱۲: ارتباط با سرور matrix و ربات چت opsdroid

برای این مرحله هم مطابق ویدیوی قرار گرفته عمل کردیم و OPSDroid و موارد لازم را نصب کردیم و یک فایل config قرار دادیم که skillها را در آن نوشتیم و سه skill در کل داریم که یکی برای accept کردن درخواست‌های اضافه شدن به یک room است و یکی به عنوان help بات است و یکی هم skill اصلی است که به کدهای توضیح داده شده در بخش‌های قبل وصل می‌شود و نوشته finglish را به فارسی تبدیل می‌کند. به عنوان نمونه کانفیگ ما به این شکل است:

```yaml
connectors:
  matrix:
    # Required
    mxid: "@alinz8102:matrix.org"
    password: [redacted]
    rooms:
      'main': '#NLP_HW2:matrix.org'
    # Optional
    homeserver: "https://matrix.org"
    nick: "TestBot"
    enable_encryption: True

skills:
  test:
    path: ./test.py
  accept_invite:
    path: ./accept_invite.py
  help:
    path: ./help.py
```

و سه skill مشخص هستند.

```python
from opsdroid.skill import Skill
from opsdroid.matchers import match_event
from opsdroid.events import UserInvite, JoinRoom

class AcceptInvites(Skill):
    @match_event(UserInvite)
    async def user_invite(self, invite):
        if isinstance(invite, UserInvite):
            await invite.respond(JoinRoom())
```

در skill مربوط به accept invite که خیلی ساده صرفا هر نوع درخواست UserInvite که بیاید، JoinRoom انجام می‌شود. در help هم صرفا وقتی help نوشته شود، یک نوشته به عنوان توضیح می‌آید.

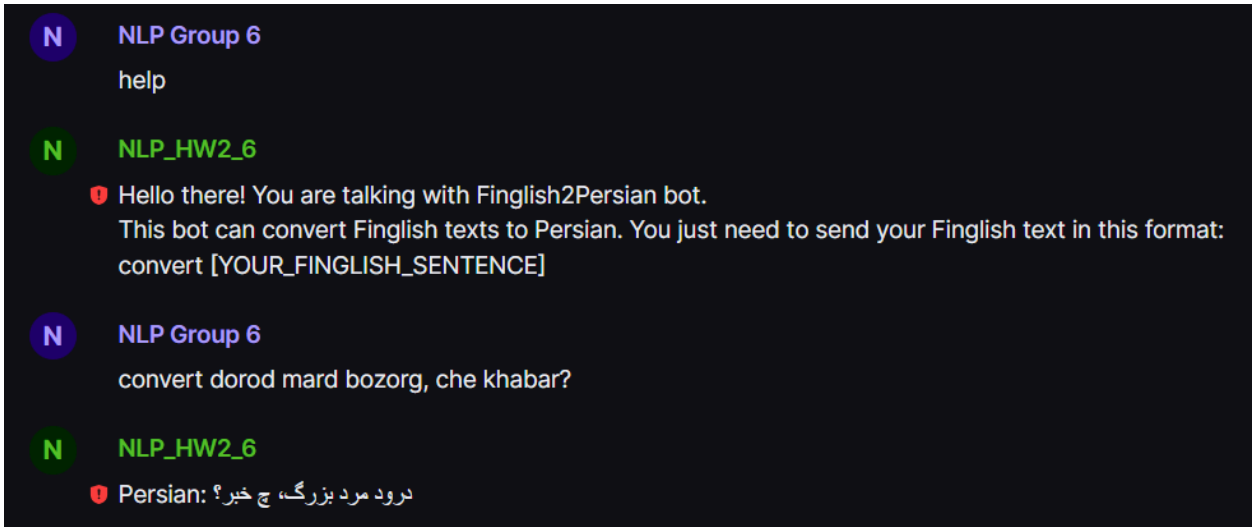
```python
import sys
sys.path.append('.')

from opsdroid.skill import Skill
from opsdroid.matchers import match_parse
from convert import convert
from chose_best_sentence import return_best_sentence


class PingSkill(Skill):
    @match_parse(r"convert {message}")
    async def ping(self, event):
        user_input = str(event.entities['message']['value'])
        sentences = convert(user_input)
        best_sentence = return_best_sentence(sentences)
        await event.respond(f"Persian: {best_sentence}")

```

در skill اصلی هم همه موارد لازم برای تبدیل اضافه شده‌اند و با دستور convert و سپس پاس دادن نوشته finglish می‌توان آن را تبدیل به نوشته فارسی کرد. یک نمونه اجرا هم مانند عکس زیر است:



که می‌بینیم به درستی کار کرده است. البته این عکس مربوط به نسخه نهایی بات نیست و در نسخه نهایی دقت هم بهتر شده است که در این داک به آن اشاره شده است.