Escuela Técnica Superior de
Ingeniería Informática

Procesamiento del Lenguaje Natural

Ejercicio teórico - práctico

Profesor

Jose Francisco Quesada Moreno

Alumno

Germán Lorenz Vieta

Índice

- **Apartado 1**

```
def addPunctuationBasic(line):
    # Caso especial
    if(line == ''):
        return ''
    # Convertimos el primer caracter de la oración a mayuscula
    first_letter = line[0].upper()
    return first_letter + line[1:] + '.'
```

- **Apartado 2**

Con la siguiente función tokenizo por palabras considerando tokens cuando estas se encuentran separadas por espacios o tokens reservados. De esta forma generar una lista de tokens para tratar las frases.

```
# Funcion para tokenizar
def tokenizador(texto, tokens):

    # Verificacion strings con datos
    if len(texto) == 0: return 0
    texto_tokenizado = []

    # Generacion de tokens
    start = 0
    for i in range(len(texto)):
        # Espacios
        if texto[i] == ' ':
            texto_tokenizado.append(texto[start:i])
            start=i+1
        # Analisis de prueba con tokens
        for j in range (len(tokens)):
            # tokens
            if texto[i] == tokens[j]:
                texto_tokenizado.append(texto[start:i])
                start=i
    return texto_tokenizado
```

Inspirándome en el algoritmo de Levenshtein utilice la función referenciada [3] para generar una matriz que me permita encontrar el recorrido óptimo con las reglas mínimas de inserción (insertion), borrado (deletion) y reemplazo (substitution). Al utilizar el algoritmo indicado obtuve una representación distinta a la recomendada en el enunciado

como minimal en la salida de verifyPunctuation. Luego, realizó la función para encontrar el recorrido mínimo con matriz.

```
def minimal_operations(matriz_levenshtein, check_len, test_len):
    col = test_len
   row = check_len
   problemas = []
   while col > 1 and row > 1:

       minimo = min(matriz_levenshtein[row-1][col],      # deletion
                    matriz_levenshtein[row][col-1],    # insertion
                    matriz_levenshtein[row-1][col-1]) # substitution

       # Insertion
       if col == 1:
           while row > 1:
               problemas.append("I,"+str(row))
               row = row - 1

       # Insertion
       elif row == 1:
           while col > 1:
               problemas.append("D,"+str(row))
               col = col - 1

       # Diagonalizo
       elif matriz_levenshtein[row-1][col-1] == minimo:
           if matriz_levenshtein[row-1][col-1] == matriz_levenshtein[row][col] -
1:
               # Substitution
               problemas.append("S,"+str(row))
           col = col - 1
           row = row - 1
       elif matriz_levenshtein[row-1][col] == minimo:
           # Deletion
           problemas.append("D,"+str(row))
           row = row - 1
       else:
           # Insertion
           problemas.append("I,"+str(row))
           col = col - 1
   return np.flip(problemas)
```

Agregue un parámetro de reporte para poder visualizar la matriz y la posibilidad de no tokenizar así procesa la información por carácter en vez de por token (palabra)

```
def verifyPunctuation(check, test, tokenizar=True, extended_report=False):
    # Caso especial
    if(check == '' or test == ''):
        return [('D', 0), ('S', 0), ('I', 0)]
    # Tokens reservados
    reserve_tokens = ['.',',',';',':','?','!']

    if tokenizar:
        # Tokenizamos input
        data_check = tokenizador(check, reserve_tokens)
        data_test = tokenizador(test, reserve_tokens)
    else:
        data_check = check
        data_test = test

    matriz_levenshtein = iterative_levenshtein(data_check, data_test)

    # Generar reporte
    if extended_report:
        # Impresion de verificacion, opcional
        print('DATOS CORRECTOS')
        print(data_check)
        print('DATOS A SER PROBADOS')
        print(data_test)

        # Impresion Matriz Levenshtein opcional
        rows = len(data_check)+1
        for r in range(rows):
            print(matriz_levenshtein[r])

    return minimal_operations(matriz_levenshtein, len(data_check),
len(data_test))
```

- ## Apartado 3

Reporte general que voy a usar en los demás enunciados

```
def reporteConMetricas(check, test, punctuationBasic = False):
```

```
    # Como los valores verdaderos son de check, asumo todos verdaderos ya que no
tengo un algoritmo de prediccion
    y_true = np.ones(len(test))

    # Inicio mi vector de prediccion al cual voy a agregarle el resultado de mi
verifyOuntuaction
    y_pred = np.zeros(len(test))

    # Variables
    tp = 0
    fp = 0
    fn = 0

    # Proceso
    for i in tqdm(range(len(test)),ncols = 100 , desc ="Verifico Metricas ...",
disable = True):
        if punctuationBasic:
            test_proceced = addPunctuationBasic(test[i])
        else:
            test_proceced = test[i]

        analysis = len(verifyPunctuation(check[i], test_proceced)) > 0
        # Verificacion
        if not analysis:
            y_pred[i] = 1

    tn, fp, fn, tp = confusion_matrix(y_true, y_pred).ravel()
    precision = tp / (tp + fp)
    recall = tp / (tp + fn)
    f1 = ((precision * recall)/(precision + recall)) * 2

    return (precision * 100, recall * 100, f1 * 100)
```

aparentemente el origen de las actuales según la innumerable cantidad de muestras recogidas y analizadas.
Este interés llevó a un análisis exhaustivo para establecer las relaciones entre diferentes especies (extintas o no extintas) d

- ## Apartado 4

Función para el entrenamiento del modelo

```
import re

token_regular_expression = r'[^.,:;?\s]+|[.,:;?]'
character_regular_expression = '([.,;:?!])'


def processTrainData(train_data):
    model_data = ' '.join(train_data).rstrip()
    tokens = re.findall(token_regular_expression, model_data)
    n_grams = []
    for i in range(0, len(tokens)-4):
        j = i+4
        n_grams.append(tokens[i:j])
    train_grams = np.array(n_grams)

    salida = {}

    for n_gram in train_grams:
        word_4ngram = n_gram[3]
        clave_palabra = ''.join(n_gram[0:3])
        next = ''
        if re.search(character_regular_expression, word_4ngram):
            next = 'C ' + word_4ngram
        elif word_4ngram[0].isupper():
            next = 'M'
        elif word_4ngram[0].isupper() != True:
            next = 'm'

        if clave_palabra in salida:
            salida[clave_palabra] = np.append(salida[clave_palabra], next)
        else:
            salida[clave_palabra] = np.array([next])


    for clave_palabra in salida:
        unique, counts = np.unique(salida[clave_palabra], return_counts=True)
        common = np.where(counts == max(counts))
        salida[clave_palabra] = unique[common]

    return salida
```

Función requerida por el enunciado la cual fue muy desafiante de desarrollar tomando ideas de los bigramas vistos en clase

```
def addPunctuation4gram(test_data, train_data):

    train_data = processTrainData(train_data)

    output = []

    for w in tqdm(range(len(test_data)),ncols = 100 , desc =" Entrenando ...",
disable = True):
        tokens = re.findall(token_regular_expression, test_data[w])
        i = 0

        # Proceso principal
        while i < len(tokens)-4:
            sentence = tokens[i: i+4]
            items = sentence[0:3]
            key =  ''.join(items)
            if key in train_data:
                value = train_data[key][0]

                if 'C' in value:
                    target_token = tokens[i+3]
                    character =  value.split()[1]
                    if target_token != character:
                        tokens.insert(i+3,character)
                elif 'M' in value:
                    target_token = tokens[i+3]
                    if target_token[0].isupper() != True:
                        tokens[i+3] = target_token.capitalize()
                elif 'm' in value:
                    target_token = tokens[i+3]
                    if target_token[0].isupper():
                        tokens[i+3] = target_token.lower()

            i+=1

        transformation = ' '.join(tokens)
        # Agrego el trabajo de Apartado 1
        transformation = addPunctuationBasic(transformation)
        output.append(transformation)

    return output
```

- ## Apartado 5

Sobre las métricas obtenidas

```
Metrica de implementacion Basica
 Precision: 100.0
 Recall: 0.2989848421638159
 F1: 0.5961871750433275

Metrica de 4-grams
 Precision: 100.0
 Recall: 0.32679738562091504
 F1: 0.6514657980456027
```

Los resultados que se pueden observar muestran que a pesar del entrenamiento con el corpus con 4-gramas no logra una mejora significativa dada la información con la que se evalúa aunque se sospecha que la implementación de tokenización puede ser la responsable del bajo rendimiento de ambos modelos.

- # Apartado 6

Sobre el análisis del modelo mencionado.

Análisis del modelo

El modelo opera con texto no segmentado a través de un tipo de red neuronal recurrente. Entonces evita el trabajo de tokenizado en componentes más pequeños.

- El modelo busca resolver específicamente problemas con comas, puntos, signos de exclamación, dos puntos, signos de interrogación los puntos y rayas.

- El modelo BRNN permite utilizar contextos de tamaño no fijo antes y después de la posición actual de procesado.

- Las capas recurrentes usan un tipo de unidad recurrente cerrada (GRU) para capturar dependencias de largo alcance en distintas unidades de tiempo. De esta forma las unidades tienen las mismas ventajas que las unidades LSTM pero siendo más sencillas en definitiva.

- Este modelo se implementa con la librería theano y los pesos se van actualizando con un learning rate de 0.02.

Comparativa con el modelo realizado

El modelo propuesto en los apartados anteriores al ser comparado con esté arroja una métrica inferior dado que la capacidad de aprendizaje con el corpus de entrenamiento

le brinda más flexibilidad a la hora de obtener el comportamiento deseado con los sets de prueba.

- # Apartado 7

Sobre el análisis de papers relacionados con modelos de punctuation

43 Language Multilingual Punctuation Prediction Neural Network Model

El modelo se basa en BPE multi-idioma la cual es una técnica de compresión de datos que permite representar las palabras en unidades pequeñas reduciendo la dispersión de las palabras compartiendo unidades (subpalabras entre distintas palabras). Los investigadores demuestran que un modelo multilingüe que considere la compatibilidad entre varios idiomas, a diferencia de los modelos monolingües tradicionales puede mejorar la precisión. En el artículo demuestran que un solo modelo multilingüe basado en BPE puede lograr un rendimiento similar o superior a los modelos monolingües por separado basados en palabras. EL modelo logra un puntaje F1 promedio del 80,2% analizando noticias mientras que en el reconocimiento de voz fuera del dominio logra un puntaje de 73,3% en F1. [2]

EL modelo presenta los siguientes resultados al combinar idiomas.

- Español intersectado con la unión de Francés e italiano en un 93.2%.
- Francés intersectado con la unión de Italiano y Español en un 95,4%.
- Italiano intersectado con la unión de Francés y ESpañol en un 95,8%.

Token-Level Supervised Contrastive Learning for Punctuation Restoration

En este trabajo se unifican varias estrategias de codificación existentes para predecir la puntuación utilizando múltiples predicciones en cada palabra en diferentes ventanas. Los investigadores demuestran que se pueden lograr mejoras significativas al optimizar estrategias luego de entrenar un modelo aumentando el tiempo de respuesta sin necesidad de re-entrenamiento.

El modelo propuesto logra una mejora sustancial cuando hay poco o ningún contexto en el right-side al entrenar. La estrategia que utilizan es usar un token [PUNCT] que informa al modelo de la posición de la puntuación que se va a predecir. Los textos de entrada se

tokenizan y la predicción se da evaluando múltiples posibles lugares donde el token de puntuación debiera ser insertado. Además se optimiza el tamaño de ventana y la longitud de zancada al momento de la clasificación. Por último cuando se debe elegir entre un modelo u otro existe una compensación entre la latencia y la precisión entre modelos. De esta forma se utiliza un modelo de clasificación cuando no hay un contexto futuro mientras que un modelo de etiquetado es usado cuando hay contexto del lado derecho a predecir.

Los valores generales que ambos modelos logran en F1 para enfoques de clasificación y etiquetado para la predicción de puntuación en tiempo real con anticipación (lookahead) de hasta 4 es de 76,3% en TAgging y 73,9 en clasificación, mientras que sin anticipación l = 0 el modelo de Tagging logra 42,1% y el de clasificación 47,3% [1]

- # Bibliografía

[1] Token-Level Supervised Contrastive Learning for Punctuation Restoration. Visitado el 17 de Junio, 2022. https://arxiv.org/pdf/2112.08098v2.pdf

[2] 43 Language Multilingual Punctuation Prediction Neural Network Model Visitado el 17 de Junio, 2022. https://indico2.conference4me.psnc.pl/event/35/contributions/2972/attachments/610/641/Mon-3-1-9.pdf

[3] Levenshtein Distance. Visitado el 14 de Junio, 2022. https://python-course.eu/applications-python/levenshtein-distance.php