2025

Tarea 2:
N-gramas

Santiago Martínez Novoa
202112020
Diego Alejandro González Vargas
202110240

ISIS-4221. PROCESAMIENTO DE LENGUAJE NATURAL
UNIVERSIDAD DE LOS ANDES

Contenido

1. Introducción

El presente informe se realiza con el fin de dar a conocer la forma de implementación propuesta por el grupo para un modelo de generación de textos basada en N-gramas. En este sentido, se presentan cada una de las etapas propuestas por el enunciado para construir el modelo. En primer lugar, se presenta la consolidación de los archivos consolidados para cada uno de los datasets entregados. Posteriormente, se enseñan las tareas de tokenización y normalización de cada una de las frases y palabras que componen los archivos consolidados para cada dataset. En tercer lugar, se enseñan las tareas realizadas para crear los nuevos datasets a partir de los tokens extraídos: el de entrenamiento y el de testeo. Luego de esto, se presenta como se construyeron los modelos para unigramas, bigramas, y trigramas y el formato en que se decidió almacenarlos. Finalmente, se expone un segundo cuaderno de Python con los métodos de testeo de estos modelos: Cálculo de perplejidad y resultados de ejercicios de generación de textos basados en las probabilidades almacenadas en cada uno de los modelos.

2. Creación de archivos consolidados

2.1. BAC

Para el caso de la consolidación de los archivos del dataset de BAC, se hace necesario la consideración del formato XML en el que vienen escritos, de acuerdo con este formato, se identificó que cada uno de los archivos que componían el dataset se encontraba compuesto de 3 etiquetas: Etiqueta blog, que sale al inicio y final del documento. Etiqueta date, con unas estampas de tiempo. Finalmente, la etiqueta “post” con el contenido en lenguaje natural de interés. Además, se identificó que cada uno de los archivos contaba con caracteres que generaban conflictos de lectura como entidades de HTML, símbolos ampersan “&” sueltos, y otros caracteres inválidos en XML.

Así las cosas, se construyeron 3 funciones que permitían realizar el pipeline completo necesitado: En primer lugar, limpiar los caracteres problemáticos. Segundo, para sacar el contenido de las etiquetas date y post. Finalmente, la función principal que usa las 2 anteriores para construir el archivo compilado en formato JSONL con el contenido de cada elemento POST y DATE

2.2. 20N

Para el caso de la consolidación de los archivos del dataset de 20N, se hace necesario la consideración del formato en el que vienen escritos, de acuerdo con este formato,

se identificó que, aunque los archivos no tienen una terminación para cargue en alguna herramienta, cada uno de los archivos que componían el dataset contenía un potencial mensaje de correo electrónico. En consecuencia, se identificaron líneas en todos los archivos con etiquetas de origen y destino, además del contenido de texto en lenguaje natural de interés. Al final del proceso, se construye también un archivo en formato JSONL con el contenido de cada archivo, y de cada una de las líneas de metadatos identificadas anteriormente.

Debido a la organización de los datos en subcarpetas para esta fuente de datos se utilizaron recorridos recursivos para poder acceder a todos los archivos en un recorrido y un generador para optimizar la iteración sobre los nombres y no cargarlos todos directamente en memoria.

3. Tokenización y Normalización

Para el caso de este segundo taller, se hace necesario realizar una tokenización a nivel de frases. Adicionalmente, el enunciado describe la necesidad de creación de normalización de cada una de estas oraciones sin eliminar las palabras de parada como se realizó en la tarea 1. Por otra parte, se menciona que, para cada una de las frases, se van a agregar cadenas de caracteres que nos ayuden a representar el inicio y la finalización de estas. Finalmente, se describe como se intercambiaron las palabras que solo se mencionaran una vez en el dataset por la cadena de caracteres <UNK>.

3.1. Normalización

```
def normalize_text(t: str) -> str:
    """
     Normaliza el texto eliminando caracteres no deseados y convirtiendo a
minúsculas.

    Args:
        text (str): El texto a normalizar.

    Returns:
        str: El texto normalizado.
    """
    t = t.strip()
    return re.sub(r"\s+", " ", t)
```

3.2. Conversión de Dígitos a NUM

En este punto, se ingresan los textos resultados de la función de normalización, y se verifica si alguna de las palabras contiene dígitos numéricos, en cuyo caso se intercambia dicho elemento por la cadena de caracteres “NUM” .

```
def map_numbers(tok: str) -> str:
    """
    Recibe una cadena de caracteres y verifica si tiene números para
    poder cambiar ese elemento por la cadena de caracteres NUM.

    Args:
        tok (str): El texto a verificar si tiene números.

    Returns:
        str: NUM si tiene números o la palabra.
    """
    return "NUM" if _HAS_DIGIT.match(tok) else tok
```

3.3. Conversión de palabras de única aparición por <UNK> y cadenas de inicio/finalización

Para reducir significativamente el vocabulario del modelo, se intercambiaron las palabras que salen una sola vez en el dataset por <UNK>. Además, se agregaron las cadenas de caracteres <s> y </s> al inicio y final de cada oración respectivamente. Para poder alcanzar estos objetivos, se realizó de manera previa el conteo de las apariciones de cada una de las palabras. Luego, en un recorrido por cada una de las frases identificadas con la librería NTLK y PUNKT, se colocan los caracteres de inicio y terminación. Finalmente, recorriendo palabra por palabra, y con el contador anteriormente mencionado como insumo, se realizan los cambios de las palabras que salgan una sola vez por UNK.

```
def write_sentences_with_unk(in_path: str, out_path: str, counts: Counter,
text_key: str = "text"):
    """
    Separa cada uno de los registros de los datasets en oraciones(sentences).
     Agrega los caracteres de inicio y de terminacion para cada una de las
frases.
    Recibe un elemento Counter con el # de apariciones de cada palabra.
    Para cada una de las frases, recorre las palabras y cambia por <UNK>
    las que salgan 1 sola vez en el Counter

    Args:
        in_path (str): El path del JSONL a procesar.
        out_path (str): El path del JSONL a crear.
        counts (Counter): El contador de las palabras.
        text_key(str): la llave dentro de cada registro del JSONL que contiene
                        el texto en lenguaje natural de interes

    Returns:
        No retorna nada, pero escribe el archivo JSONL en el path especificado
```

```
        con las frases ya procesadas.
    """
    texts = load_texts_from_jsonl(in_path, text_key)
    with open(out_path, "w", encoding="utf-8") as out:
        for text in texts:
            txt = normalize_text(text.lower())
            for sent in split_sentences(txt):
                toks = [map_numbers(t) for t in tokenize(sent)]
                toks = [t if counts[t] > 1 else "<UNK>" for t in toks]
                if not toks:
                    continue
                sent_tokens = ["<s>", *toks, "</s>"]
                            out.write(json.dumps({"sentence":   sent_tokens},
ensure_ascii=False) + "\n")
```

4. División del Dataset en Entrenamiento y Testeo

4.1. Implementación Randomizada

Para poder realizar la división del dataset, se realiza la una función que, utilizando la librería random de Python, y una semilla (un número entero), se haga la mezcla randomizada de las frases contenidas por cada uno de los archivos JSONL de cada dataset. Finalmente, esta versión ya randomizada de los contenidos se parte en las proporciones indicadas por el enunciado (80/20), con el formato de nombres sugerido. La función se muestra a continuación:

```
def    split_train_test(input_jsonl,    prefix,    group_code,    out_dir=".",
train_ratio=0.8, seed=42):
    """
    Mezcla las frases del archivo JSONL usando la libreria random
    Utiliza la semilla para garantizar qeu se pueda reproducir
    Se crean los archivos JSONL con los datasets de entrenamiento y
    testeo de acuerdo con las proporciones recomendadas y el formato
    de nombramiento

    Args:
        input_jsonl (str): El path del JSONL a procesar.
        prefix (str): Para poder reusar la función para los 2 datasets,
                        se especifica el nombre para el formateo.
        out_dir (str): El path de los JSONL a crear
        train_ratio (float): proporcion de datos para entrenamiento
        seed (int): semilla para el randomizador

    Returns:
        No retorna nada, pero escribe los JSONL en el path especificado
        con las frases ya procesadas y randomizadas.
    """
```

```
random.seed(seed)

# cargar todas las oraciones
sentences = []
with open(input_jsonl, "r", encoding="utf-8") as f:
    for line in f:
        if not line.strip():
            continue
        obj = json.loads(line)
        sentences.append(obj)

# barajar
random.shuffle(sentences)

# dividir 80/20
split_idx = int(len(sentences) * train_ratio)
train_sents = sentences[:split_idx]
test_sents = sentences[split_idx:]

# nombres de salida
out_train = Path(out_dir) / f"{prefix}_{group_code}_training.jsonl"
out_test = Path(out_dir) / f"{prefix}_{group_code}_testing.jsonl"

# guardar
with open(out_train, "w", encoding="utf-8") as f:
    for s in train_sents:
        f.write(json.dumps(s, ensure_ascii=False) + "\n")

with open(out_test, "w", encoding="utf-8") as f:
    for s in test_sents:
        f.write(json.dumps(s, ensure_ascii=False) + "\n")

print(f"Training: {len(train_sents)} → {out_train}")
print(f"Testing : {len(test_sents)} → {out_test}")
```

5. Construcción de los modelos de N-gramas

5.1. Lectura del JSONL

En primer lugar, en el proceso de consecución de los modelos de N-gramas, se construyó una función que recibiese como input la ubicación del archivo JSONL producido por el pipeline hasta ahora descrito, es decir, en este caso, alguno de los archivos de entrenamiento de cualquier conjunto de datos. A partir de esta ubicación, se abre el archivo y se extrae cada una de las frases que contiene, armando una lista de listas, donde cada una de las listas que componen la lista principal corresponde a oraciones procesadas extraídas del conjunto de entrenamiento establecido.

```
def process_jsonl_for_ngrams(
    jsonl_file_path: str,
    n: int = 2,
    laplace: float = 1.0,
    batch_size: int = 50000
) -> Dict[Tuple[str, ...], float]:
    """
    Lee un archivo JSONL con sentencias y calcula probabilidades de n-gramas.

    Args:
        jsonl_file_path: Ruta al archivo JSONL
        n: Tamaño del n-grama (1=unigrama, 2=bigrama, etc.)
        laplace: Factor de suavizado de Laplace
        batch_size: Tamaño del lote para procesamiento en memoria
        max_sentences: Máximo número de sentencias a procesar (None = todas)
         remove_special_tokens: Si remover tokens especiales como <s>, </s>,
<UNK>

    Returns:
        Diccionario con n-gramas como claves y probabilidades como valores
         Diccionario con contextos (n-1)-grama como claves y sus conteos como
valores
    """

    sentences = []

    print(f"Leyendo archivo JSONL: {jsonl_file_path}")

    try:
        with open(jsonl_file_path, 'r', encoding='utf-8') as file:
            for line_num, line in enumerate(file, 1):

                line = line.strip()
                if not line:  # Saltar líneas vacías
                    continue
                try:
                    # Parsear JSON
                    data = json.loads(line)

                    # Extraer sentencia
                    if "sentence" not in data:
                          print(f"Advertencia: línea {line_num} no tiene clave
'sentence'")
                        continue

                    sentence = data["sentence"]

                    # Validar que sea una lista
```

```
                    if not isinstance(sentence, list):
                        print(f"Advertencia: línea {line_num} - 'sentence' no
es una lista")
                        continue

                    # Saltar sentencias vacías
                    if not sentence:
                        continue

                    sentences.append(sentence)


                except json.JSONDecodeError as e:
                    print(f"Error parseando JSON en línea {line_num}: {e}")
                    continue
                except Exception as e:
                    print(f"Error procesando línea {line_num}: {e}")
                    continue

    except FileNotFoundError:
        raise FileNotFoundError(f"No se encontró el archivo: {jsonl_file_path}")
    except Exception as e:
        raise Exception(f"Error leyendo archivo: {e}")

    if not sentences:
        raise ValueError("No se encontraron sentencias válidas en el archivo")

    print(f"Total de sentencias cargadas: {len(sentences)}")
    print(f"Calculando {n}-gramas con suavizado de Laplace (α={laplace})...")
```

Posteriormente, se hace un llamado a la función calculate_ngram_probabilities. Esta función, de acuerdo con el n escogido, utiliza la lista de listas anteriormente mencionada para procesarla y crear el modelo de n-gramas con las probabilidades de cada n-grama, con suavizado de LaPlace.

```
probabilities, context_counts = calculate_ngram_probabilities(
        sentences=sentences,
        n=n,
        laplace=laplace,
        batch_size=batch_size
    )
```

Para poder alcanzar estos objetivos, se siguen los siguientes pasos, todos dentro de esta función:

5.2. Conteo de apariciones de cada n-grama

En primer lugar, teniendo en cuenta que los datasets y las estructuras de datos creadas pueden llegar a ocupar toda la memoria disponible en el dispositivo donde se corre el código, y para evitar que dicha ejecución se vea interrumpida por acceso a recursos, se utiliza una aproximación de procesamiento por batch de las frases. Así pues, de acuerdo con el número de registros definidos para cada batch, se realiza el conteo del número de apariciones de cada n-grama encontrado. Adicionalmente, en caso de que el modelo no sea de unigramas, se calcula el conteo de apariciones de cada uno de los contextos, definidos como las combinas de n-1 palabras previas encontradas para completar los n-gramas.

```
ngram_counts = Counter()
    context_counts = Counter() if n > 1 else None
    vocab_size = 0
    total_ngrams = 0
    # Procesar en lotes para memoria
    for batch_start in range(0, len(sentences), batch_size):
        batch_end = min(batch_start + batch_size, len(sentences))
        batch_sentences = sentences[batch_start:batch_end]
        # Set temporal para vocabulario del batch
        batch_vocab = set()
        for sent in batch_sentences:
            batch_vocab.update(sent)
            # Contar n-gramas
            for i in range(len(sent) - n + 1):
                ngram = tuple(sent[i:i + n])
                ngram_counts[ngram] += 1
                total_ngrams += 1
                # Solo calcular contextos si n > 1
                if context_counts is not None:
                    context = tuple(sent[i:i + n - 1])
                    context_counts[context] += 1

        # Actualizar tamaño de vocabulario
        vocab_size = len(batch_vocab | set().union(*[
            set(sent) for sent in sentences[:batch_start]
        ])) if batch_start > 0 else len(batch_vocab)
```

5.3. Evaluación de las consultas

Posteriormente, se usan los conteos tanto de los n-gramas como de los contextos (en caso de tenerlos) para calcular las probabilidades de ocurrencia de cada ngrama, siguiendo con la formula propuesta en el curso. Así pues, se tiene en cuenta la utilización del suavizado de La Place, el cual se traduce en agregar uno en el numerador y agregar el tamaño del vocabulario en el denominador. Dichos cálculos están descritos por el siguiente código:

```
probabilities = {}

    if n == 1:
        # Para unigramas - calcular total una sola vez
        for ngram, count in ngram_counts.items():
            prob = (count + laplace) / (total_ngrams + laplace * vocab_size)
            probabilities[ngram] = prob
    else:
        # Para n-gramas superiores
        for ngram, count in ngram_counts.items():
            context = ngram[:-1]
            context_count = context_counts[context]
            prob = (count + laplace) / (context_count + laplace * vocab_size)
            probabilities[ngram] = prob
```

Es importante mencionar aquí que no se guarda toda la matriz de probabilidades, pues al ser tan dispersa se estaría guardando una cantidad de probabilidades que a la larga terminarán siendo iguales. Se decidió que por eficiencia únicamente se guardarían las probabilidades de aquellos n-gramas encontrados.

5.4. Persistencia de los resultados

Finalmente, se retorna tanto las probabilidades como los conteos de contextos en caso de tenerlos. Ahora bien, para generar persistencia de estas probabilidades obtenidas, así como de los conteos retornados, se guarda toda esta información en archivos de tipo PICKLE, como sigue

```
with open('ngrams/unigrams_20n.pkl', 'wb') as f:
    pickle.dump(unigrams_20n, f)
with open('ngrams/bigrams_20n.pkl', 'wb') as f:
    pickle.dump(bigrams_20n, f)
with open('ngrams/trigrams_20n.pkl', 'wb') as f:
    pickle.dump(trigrams_20n, f)
with open('ngrams/context_counts_unigrams_20n.pkl', 'wb') as f:
    pickle.dump(context_counts_uni, f)
with open('ngrams/context_counts_bigrams_20n.pkl', 'wb') as f:
    pickle.dump(context_counts_bi, f)
with open('ngrams/context_counts_trigrams_20n.pkl', 'wb') as f:
    pickle.dump(context_counts_tri, f)
```

La decisión de guardar los conteos de los contextos obedece a ganar eficiencia en la obtención de los resultados de la evaluación, como se verá más adelante, dichos conteos se utilizan para el cálculo de perplejidades y predicción de nuevas palabras. Se ejecutan estos mismos métodos para ambos conjuntos de datos y de esta forma se da cumplimiento a este punto del taller.

Este cálculo hace referencia a aquellas combinaciones que no aparecieron durante el cálculo de probabilidades pero que pueden aparecer en un futuro (contexto conocido para una palabra no conocida o contexto desconocido).

6. Cálculo de la Perplejidad

Para el calculo de la perplejidad, se utilizó POO. En este sentido, se creó una clase NgramModel en donde se puede cargar el modelo obtenido en el formato PICKLE. A partir de ahí, se desarrollan métodos dentro de la clase que permiten hacer cada uno de los procesos para los testeos necesarios. En el caso de la perplejidad, dichos procesos se describen a continuación:

6.1. Proceso

Para poder extraer la perplejidad de un conjunto de datos de prueba, se extrae la probabilidad de cada una de las frases mencionadas. Para esto, se hace necesario dividir cada una de las frases en n-gramas dependiendo del modelo que se esté evaluando. Una vez se tienen estos N-gramas, se extrae la probabilidad de dicho n-grama, directamente del modelo precargado, y se suman dichas probabilidades una vez se les saca el logaritmo, para evitar overflow. La función anteriormente descrita se muestra a continuación:

```
def calculate_perplexity(self, test_sentences: List[List[str]]) -> float:
        """
        Calcula la perplejidad del modelo sobre un conjunto de prueba.

        Args:
            test_sentences: Lista de oraciones tokenizadas para evaluar

        Returns:
            float: Valor de perplejidad
        """
        total_log_prob = 0.0
        total_ngrams = 0
        unseen_count = 0
        for sentence in test_sentences:
            # Generar n-gramas de la oración
            for i in range(len(sentence) - self.n + 1):
                ngram = tuple(sentence[i:i + self.n])

                if ngram in self.ngram_probs:
                    # N-grama visto: usar su probabilidad
                    prob = self.ngram_probs[ngram]
                else:
                    # N-grama no visto: usar suavizado correcto
                    unseen_count += 1
```

```
                    prob = self.get_ngram_probability(ngram)

                total_log_prob += math.log(prob)
                total_ngrams += 1

        if total_ngrams == 0:
            return float('inf')

        # Calcular perplejidad
        avg_log_prob = total_log_prob / total_ngrams
        perplexity = math.exp(-avg_log_prob)

        print(f"Estadísticas para {self.n}-gramas:")
        print(f"  Total n-gramas evaluados: {total_ngrams}")
        print(f"  N-gramas no vistos: {unseen_count}")
        print(f"  Perplejidad: {perplexity:.4f}")

        return perplexity
```

Es importante notar que en vez de utilizar un valor fijo para la probabilidad de n-gramas no vistos se hace un llamado a la función get_ngram_probability(). En esta función es posible observar la razón por la cual se guardaron las cuentas de cada contexto, de esta manera se calcula de manera precisa la probabilidad para ngramas no vistos dependiendo si se conocía el contexto o no.

```
    def get_ngram_probability(self, ngram: Tuple[str, ...]) -> float:
        """
         Calcula la probabilidad de un n-grama específico usando suavizado de
Laplace.

        Args:
            ngram: N-grama a evaluar

        Returns:
            float: Probabilidad del n-grama
        """

        # Si el n-grama fue visto durante entrenamiento
        if ngram in self.ngram_probs:
            return self.ngram_probs[ngram]

        # Si no fue visto, aplicar suavizado de Laplace correcto
        if self.n == 1:
            # Para unigramas no vistos - usar aproximación o 1/vocab_size para
OOV
            return 1.0 / self.vocab_size
```

```
        else:
            # Para n-gramas superiores
            context = ngram[:-1]

            if context in self.context_counts:
                # Contexto conocido: usar count real
                context_count = self.context_counts[context]
                    return self.laplace / (context_count + self.laplace *
self.vocab_size)
            else:
                # Contexto no visto: probabilidad uniforme
                return 1.0 / self.vocab_size
```

6.2. Resultados Obtenidos

Una vez se ejecutaron los procesos anteriormente mencionados, se obtuvieron los siguientes resultados para cada modelo/dataset:

20N	BAC
========== EVALUACIÓN DE PERPLEJIDAD ========== UNIGRAMAS: Estadísticas para 1-gramas: Total n-gramas evaluados: 1503195 N-gramas no vistos: 3095 Perplejidad: 637.8866 BIGRAMAS: Estadísticas para 2-gramas: Total n-gramas evaluados: 1445603 N-gramas no vistos: 136116 Perplejidad: 981.5244 TRIGRAMAS: Estadísticas para 3-gramas: Total n-gramas evaluados: 1388011 N-gramas no vistos: 422560 Perplejidad: 7127.5982 ========== RESUMEN COMPARATIVO ========== UNIGRAMS : 637.89 BIGRAMS : 981.52 TRIGRAMS : 7127.60	========== EVALUACIÓN DE PERPLEJIDAD ========== UNIGRAMAS: Estadísticas para 1-gramas: Total n-gramas evaluados: 35828187 N-gramas no vistos: 16252 Perplejidad: 656.3425 BIGRAMAS: Estadísticas para 2-gramas: Total n-gramas evaluados: 34045601 N-gramas no vistos: 1528478 Perplejidad: 734.1268 TRIGRAMAS: Estadísticas para 3-gramas: Total n-gramas evaluados: 32263015 N-gramas no vistos: 7196407 Perplejidad: 11842.1412 ========== RESUMEN COMPARATIVO ========== UNIGRAMS : 656.34 BIGRAMS : 734.13 TRIGRAMS : 11842.14

7. Generación de Texto

La generación de texto con modelos de n-gramas permite observar hasta qué punto el modelo ha capturado dependencias locales en el corpus de entrenamiento. Para este ejercicio se utilizaron los modelos de unigramas y bigramas entrenados sobre los conjuntos 20 Newsgroups y Blog Authorship Corpus (BAC), aplicando la función generate_sentence implementada en el notebook. Esta función utiliza la instancia/objeto de la clase NgramModel del modelo que está evaluando, específicamente la función predict_next_word, para extraer la siguiente palabra a aparecer basado en las probabilidades calculadas en dicho modelo y en el contexto dado (para n gramas con n>1).

Una diferencia importante en la generación de oraciones entre el modelo de unigramas y los de mayor orden fue la forma de seleccionar la siguiente palabra. En el caso de los unigramas, no siempre se escogía la palabra con mayor probabilidad, ya que este modelo no considera ningún contexto y, por lo tanto, tendería a devolver siempre la misma palabra más frecuente (en este caso, NUM). Para evitar esa repetición, se implementó un muestreo aleatorio ponderado: en lugar de elegir siempre la más probable, se seleccionaba una palabra al azar teniendo en cuenta la distribución de probabilidades, lo que permitía mayor diversidad en los resultados. En cambio, para los n-gramas de orden superior, sí se optó por seleccionar la palabra más probable condicionada al contexto, lo que garantizó una mayor coherencia en las secuencias generadas.

El procedimiento consistió en suministrar un contexto inicial de una o dos palabras y dejar que el modelo complete la secuencia hasta alcanzar un token de cierre (</s>) o una longitud máxima predefinida.

7.1. Ejercicios:

- Ejemplo 1:
  - Modelo: Unigramas en dataset 20N
  - Entrada: the
  - Salida: the or NUM satellites 's do the lumping , j article
- Ejemplo 2:
  - Modelo: Unigramas en dataset 20N
  - Entrada: we
  - Salida: we . platforms those rick , -- the </s>
- Ejemplo 3:
  - Modelo: Bigramas en dataset BAC
  - Entrada: the
  - Salida: the same time . </s>
- Ejemplo 4:
  - Modelo: Bigramas en dataset BAC
  - Entrada: we

    - Salida: we were n't know what i 'm not to the same
- Ejemplo 5:
    - Modelo: Bigramas en dataset BAC
    - Entrada: do
    - Salida: do n't know what i 'm not to the same time . </s>
- Ejemplo 6:
    - Modelo: Trigramas en dataset BAC
    - Entrada: how do
    - Salida: how do you think you 're not going to be a good thing . </s>
- Ejemplo 7:
    - Modelo: Trigramas en dataset BAC
    - Entrada: '<s>','this'
    - Salida: <s> this is the best of all the time . </s>
- Ejemplo 8:
    - Modelo: Trigramas en dataset BAC
    - Entrada: 'the','problem'
    - Salida: the problem is that i have to go to the point of view . </s>

8. Conclusiones

A lo largo del desarrollo de esta tarea se logró implementar de manera completa el pipeline propuesto para la construcción de modelos de lenguaje basados en n-gramas. Desde la consolidación y limpieza de los datasets hasta la normalización, tokenización y división en subconjuntos de entrenamiento y prueba, cada etapa permitió estructurar adecuadamente la información para el posterior cálculo de probabilidades y métricas de evaluación.

Los resultados de perplejidad obtenidos muestran con claridad la tendencia esperada: los modelos de unigramas, aunque simples, alcanzan valores más bajos de perplejidad debido a que se apoyan únicamente en la frecuencia de aparición individual de cada palabra. En contraste, los bigramas y trigramas incrementan de forma significativa este valor, evidenciando tanto la mayor complejidad de los contextos que intentan capturar como las limitaciones de datos frente a la diversidad de combinaciones posibles. Este comportamiento confirma que los n-gramas de mayor orden pueden sobreajustarse fácilmente y presentar dificultades para generalizar. Sin embargo, vale la pena mencionar que, para el caso de la generación de texto, el uso de contextos más grandes aporta un mayor valor en el uso práctico del algoritmo. En este sentido, se identifica que, para el caso de trigramas y bigramas, a pesar de tener una mayor perplejidad, las oraciones generadas tienen más sentido y se acercan más al uso natural del lenguaje que con el caso de los modelos de unigramas

En el apartado de generación de texto se evidenció que los modelos son capaces de construir oraciones gramaticalmente coherentes cuando el contexto inicial es claro y

frecuente en el corpus. Sin embargo, también se observó la tendencia a producir repeticiones o frases sin sentido, especialmente en los casos donde se requieren dependencias más largas o cuando el modelo se enfrenta a contextos poco comunes. Esto pone de manifiesto las restricciones inherentes a los modelos n-grama, los cuales, si bien capturan patrones locales, carecen de una verdadera comprensión semántica.

En conclusión, el trabajo realizado permitió no solo cumplir con los requerimientos técnicos del enunciado, sino también comprender de manera práctica las fortalezas y debilidades de los modelos n-grama. Si bien estos modelos son útiles como aproximación inicial y permiten observar con claridad el impacto del contexto en las predicciones, sus limitaciones abren la puerta a explorar alternativas más avanzadas, como modelos neuronales recurrentes o basados en transformadores, que podrían superar los problemas de escasez de datos y mejorar la coherencia en la generación de texto.