Tecnológico de Costa Rica

Escuela de Computación

Programa de Maestría en Computación

Generación de código a partir de comandos en español con un modelo neuronal sintáctico y ASTs genéricos

Tesis para optar por el grado de *Magíster Scientiae* en Computación, con énfasis en Ciencias de la Computación

Steven Solano Rojas

Profesor asesor
Antonio González Torres

San José, Costa Rica, 31 de agosto de 2024

a mis padres; por apoyarme siempre

Agradecimientos

Damos un profundo agradecimiento a todas las personas que brindaron su apoyo en el proceso de esta investigación. A amigos y compañeros que siempre estuvieron disponibles para ayudar para que este trabajo se realizara de la mejor manera.

Agradecemos de manera especial a nuestro compañero Brian Wagemans Alvarado, por sus aportes esenciales en el desarrollo de este trabajo. Gran parte de sus contribuciones están contempladas en los resultados de esta investigación, y serán de ayuda para investigaciones futuras.

De manera especial, agradecemos a nuestras familias que siempre han estado en los momentos buenos y malos. Ellos fueron el motor esencial para completar los retos más difíciles. Gracias a Dios por permitirnos tener el apoyo de todas esas buenas personas.

Steven Solano Rojas

San José, 31 de agosto de 2024

Resumen

Los estudiantes sin conocimiento en programación pueden enfrentar retos importantes al tener que aprender a pensar con una lógica diferente a la acostumbrada, y memorizar una serie de comandos nuevos que no siempre se asemejan a un lenguaje natural. Por lo tanto, programar en una lengua materna puede disminuir la carga cognitiva, haciendo más fácil la enseñanza del pensamiento computacional.

Existen modelos LLM como GPT-4, CodeGen o CodeT, que han logrado generar buenos resultados en la tarea de generación de código a partir de instrucciones en lenguaje natural. Sin embargo, la mayoría de estas propuestas se basan en el inglés, y el costo para el entrenamiento de estos modelos puede ser muy alto.

En este trabajo se presenta un modelo capaz de generar código a partir de instrucciones provenientes de audios en español que fue diseñado bajo el paradigma divide y conquista, permitiendo separarlo en tres componentes principales: el primer componente recibe como entrada un audio para ser transformado a texto plano con un modelo Speech To Text, el segundo componente transforma el texto plano a código Python usando el modelo Tranx, y el tercer componente nos permite transformar el código Python generado a otros lenguajes de programación como C# y Java usando GAST.

Para el entrenamiento y las pruebas de los modelos se creó un conjunto de datos en español llamado EsPython. Este conjunto de datos es explicado en detalle en el trabajo, junto con los resultados obtenidos del modelo para la generación de código a partir de comandos en español.

Palabras clave: Generación de código, AST, GAST, Tranx, PLN.

Abstract

Students without programming knowledge may face significant challenges when they have to learn to think with a different logic from what they are used to, and memorize a set of new commands that do not always resemble natural language. Therefore, programming in a native language can reduce cognitive load, making the learning process of computational thinking easier.

There are LLM models like GPT-4, CodeGen, or CodeT that have achieved good results in the task of generating code from natural language instructions. However, most of these proposals are based on English, and the cost for training these models can be high.

This work presents a model that can generate code from instructions coming from audio in Spanish. It was designed under the divide and conquer paradigm, allowing it to be separated into three main components: The first component takes an audio input and transforms it into plain text using a Speech To Text model. The second component converts the plain text into Python code using the Tranx model, and the third component transforms the generated Python code into other programming languages such as C# and Java using GAST.

The dataset in Spanish called EsPython was created for the training and testing of the model. This dataset is detailed in the paper, along with the results obtained from the model for generating code from commands in Spanish.

Keywords: Code generation, AST, GAST, Tranx, NLP.

Índice general

Índice de figuras

Capítulo 1

Introducción

En la actualidad, el desarrollo de software por lo general depende de lenguajes de programación con una sintaxis definida. Estas sintaxis pueden llegar a limitar el razonamiento de un programador aprendiz para resolver un problema específico, debido a que sus pensamientos se desarrollan en su lengua materna, la cual ha aprendido a lo largo de su vida [45].

Escribir programas sin errores sintácticos puede frustrar a los estudiantes en los cursos iniciales de programación. Además, algunos adultos mayores y personas con discapacidades que tengan limitaciones para usar un teclado pueden llegar a tener mayor dificultad en el proceso de aprendizaje de programación, al tener que escribir para estos lenguajes que son más rígidos en comparación con su lengua materna.

En ciertas ocasiones los programadores saben lo que requieren hacer de forma general, pero no tienen la habilidad de convertir sus ideas en una implementación concreta [62]. Para resolver este problema, normalmente se realizan búsquedas de soluciones en la web por medio de lenguaje natural que luego pueden ser modificadas a un lenguaje de programación especifico [6]. Existen modelos LLM como GPT-4 [47], CodeGen [42], CodeT [7] que han logrado generar buenos resultados en la tarea de generación de código a partir de instrucciones en lenguaje natural. Sin embargo, la mayoría de estas propuestas se basan en el inglés como lenguaje natural a tratar.

La enseñanza del desarrollo de software en estudiantes sin conocimiento en programación puede resultar complicada, porque deben aprender a pensar con una lógica diferente a la acostumbrada y memorizar una serie de comandos. Por lo tanto, programar en una lengua materna podría beneficiar el pensamiento computacional, el cuál pretende desarrollar habilidades como pensamiento algorítmico, cooperación, creatividad, pensamiento crítico y resolución de problemas [10].

En esta investigación se presenta un método para efectuar la traducción de lenguaje natural a lenguaje de programación utilizando como base el español. Este método permite traducir instrucciones de audio en español a diferentes lenguajes de programación para minimizar la interacción con un teclado al desarrollar software.

1.1 Planteamiento del problema

En las comunicaciones diarias que tenemos con otras personas, el lenguaje natural presenta complicaciones por la ambigüedad que surge de las diferentes interpretaciones que se pueden dar a una palabra, oración, o conversación. En el contexto de la programación y el diseño de algoritmos, entre las técnicas que se utilizan se encuentra el uso de pseudocódigo. Este se asemeja a un lenguaje natural simplificado que permite describir los algoritmos detalladamente. Sin embargo, hereda los problemas de los lenguajes naturales, porque no existen estándares que definan cuáles palabras utilizar [24].

La traducción de lenguajes naturales a código es compleja y efectuar la traducción de español a múltiples lenguajes de programación presenta retos importantes. Esto requiere mapear la intención del ser humano en estructuras más rígidas y que tienen un formato definido sin ambigüedades. Cabe señalar que las personas expresan las ideas con algunos errores gramaticales y pueden interpretarse de múltiples maneras.

Con el fin de que el estudiante pueda desarrollar programas con instrucciones en español y obtener el resultado en diferentes lenguajes de programación, el modelo presentado en esta investigación muestra un analizador semántico que transforma lenguaje natural a múltiples lenguajes de programación. Actualmente el Tecnológico de Costa Rica cuenta con el GAST[1], un AST genérico que ayuda a traducir código fuente de un lenguaje de programación a otro lenguaje de programación. Para resolver el problema de traducción de lenguaje natural a lenguaje de programación se utilizó la estrategia de divide y conquista, el cual genera el analizador semántico con tres pasos principales. El primer paso transforma los audios en español a texto, el segundo genera código en el lenguaje de programación Python a partir de texto, y el ultimo paso transforma el código Python a múltiples lenguajes de programación utilizando el GAST.

En concreto, la investigación realizada presenta un método que permite crear programas simples en lenguaje natural en español con una sintaxis con cierto grado de flexibilidad. Estos programas luego pueden ser traducidos a los diferentes lenguajes de programación soportados por el GAST. Esto tiene por fin apoyar el desarrollo del pensamiento computacional al mostrar al estudiante sus ideas en diferentes lenguajes de programación, y facilitar la inclusión de personas mayores o con alguna discapacidad interesadas en aprender a programar.

[1]El GAST es un árbol de sintaxis abstracto genérico, y se le denomina de esa forma por Generic Abstract Syntax Tree en inglés.

1.2 Justificación del problema

Existen diversas investigaciones que han demostrado que la exposición temprana de los niños a iniciativas y actividades STEM[2] los impacta de forma positiva. Muchos países han apostado por invertir en la educación de estas disciplinas debido a la necesidad de profesionales y la alta demanda que exige la economía global en alta tecnología [8]. En consecuencia, un gran número de países desarrollados está preparando a los estudiantes como agentes creativos y transformadores [15]. Por lo tanto, están estimulando la enseñanza de la informática, el pensamiento computacional y la programación en los currículos de educación tanto en primaria como secundaria [25], incluyendo las carreras de ciencias, tecnología, ingeniería y matemáticas. En un contexto de habla hispana, lograr realizar tareas de programación usando español como lenguaje natural le facilitaría a los estudiantes desarrollar habilidades tanto en computación como en otras disciplinas STEM.

Los lenguajes de programación tradicionales como Java y C# son formales y usan una representación más cercana a la forma en que trabajan las computadoras. En tanto, los lenguajes de programación visual usan una representación más cercana al lenguaje humano, donde los estudiantes por lo general solo deben de arrastrar y soltar bloques de comandos. Esto reduce la complejidad y el uso de elementos sintácticos como las llaves, paréntesis, puntos y comas. Estas características contribuyen con la reducción de la carga cognitiva en los estudiantes, y les permite enfocarse en la lógica y estructura necesaria sin tener que preocuparse por problemas de compilación o memorización de una gran cantidad de instrucciones [36].

En los últimos años se han desarrollado tecnologías de computación más accesibles como HTML5, el estándar W3C o los lectores de pantalla. Algunos centros educativos han llevado a cabo de forma activa capacitaciones diseñadas para niños ciegos y con baja visión con el fin de promover oportunidades científicas en las disciplinas STEM. En ese contexto, se encuentra la programación visual, la cual reduce la carga cognitiva en las personas. Sin embargo, las personas con discapacidades para ver una pantalla, utilizar un teclado o mover un ratón no pueden aprovechar sus ventajas [55].

Las habilidades de programación pueden mejorar la calidad de vida y lazos sociales de los adultos mayores al participar en actividades desafiantes, creativas y colaborativas. Esto puede permitirles mejorar habilidades para mantener una actividad laboral y podría contribuir a diversificar el área de la tecnología que está sesgada hacia los jóvenes [20]. Sin embargo, el aprendizaje del pensamiento computacional en estas personas puede verse afectado por las dificultades que pueden tener al usar un teclado o ratón. Aunado a esto, los lenguajes tradicionales tienen una sintaxis rígida. Incluso algunas personas adultas han informado frustraciones como la disminución percibida en las habilidades cognitivas, y problemas para lidiar con tecnologías de software en constante cambio.

La traducción del lenguaje natural en español a un lenguaje de programación podría facilitar el desarrollo del pensamiento computacional en estudiantes y la inclusión de personas

[2]STEM es el acrónimo en inglés de ciencias, tecnología, ingeniería y matemáticas

con discapacidades y adultos mayores con dificultades para utilizar los dispositivos de entrada para programar. El uso de un lenguaje natural para resolver un problema y ver el resultado en un lenguaje de programación pretende facilitar la comprensión del proceso de resolución de problemas, por lo tanto, puede promover la educación e inclusión en disciplinas que tienen una alta demanda en la economía actual.

1.3 Hipótesis

Hipótesis 1: El analizador semántico Tranx que está basado en un autoencoder y entrenado con texto en inglés pueden tener resultados similares con conjuntos de datos en español.

Hipótesis 2: El analizador semántico puede recibir como entrada texto generado por un sistema de reconocimiento automático de voz (ASR[3]) a partir de audios en español y generar código Python.

Hipótesis 3: Se puede generar código en múltiples lenguajes de programación a partir de texto con instrucciones en español, mapeando el texto al GAST por medio de un analizador semántico[4] [62, 11].

1.4 Objetivos

Esta sección presenta el objetivo general y los objetivos específicos de nuestra investigación.

1.4.1 Objetivo general

Implementación de un analizador semántico para traducir instrucciones simples de audio en español a diferentes lenguajes de programación que puedan ser utilizados en el proceso de aprendizaje del pensamiento computacional por personas de diferentes edades y también por aquellas que tienen problemas de accesibilidad.

1.4.2 Objetivos específicos

- Generación de un analizador semántico por medio de un autoencoder entrenado con un conjunto de datos en español.

[3]El reconocimiento automático de voz es conocido en inglés como Automatic Speech Recognition (ASR).

[4]Un analizador semántico (conocido como semantic parser en inglés) examina el significado de expresiones en lenguaje natural y las transforman a representaciones interpretables por una máquina [11].

- Representación de un programa descrito en un audio en español con el GAST a partir del resultado de un analizador semántico.

- Evaluación de la eficacia del analizador semántico para traducir audios en español a diferentes lenguajes de programación.

1.5 Alcances y limitaciones

1.5.1 Alcances

Esta investigación tenía como fin probar la eficacia de un modelo para traducir un programa descrito por un audio en español a diferentes lenguajes de programación. En este proceso se utiliza el GAST, un analizador semántico basado en un autoencoder y un ASR. La estructura del GAST permite mapear código de un lenguaje de programación especifico a múltiples lenguajes de programación, sin embargo, cabe destacar que esta investigación se enfocó en la generación de código en Java y C#.

La transformación de audios en español a texto en español se realiza con un sistema ASR y el mapeo del texto al GAST se hace por medio de un autoencoder, que fue entrenado como parte de este trabajo.

1.5.2 Limitaciones

El conjunto de datos que se utilizó fue creado de forma manual por dos personas de habla hispana nativa y que tienen experiencia con Python. La razón para efectuarlo de esta forma es que no se cuenta con conjuntos de datos públicos funcionales para la investigación realizada. El uso de técnicas de aumento de datos[5] fue utilizado para generar más ejemplos debido a la limitación en cuanto a cantidad de datos que se pueden generar al tener que realizar el conjunto de datos de forma manual. Los datos creados manualmente tienen una sección con un texto en lenguaje natural describiendo el programa y otra sección con el código fuente esperado a partir de la descripción dada.

Los audios en español se basan en el conjunto de datos creados con texto en español y fueron grabados manualmente. Debido a que los audios son grabados de forma manual, se limitó a grabar solo las instrucciones que se usaron para pruebas del modelo que genera código a partir de texto en español.

Nuestros modelos tienen la intención de poder ser entrenados y ejecutados en computadoras personales debido al gran costo asociado que tienen los modelos recientes, la gran cantidad de datos para entrenarlos y el difícil acceso al hardware utilizado para estos.

[5]Aumento de datos o Data augmentation en inglés, se refiere a estrategias para aumentar la cantidad de ejemplos de entrenamiento sin recolectar nuevos datos [59].

1.6 Estructura de la investigación

En este documento se encuentra una introducción en el capítulo 1, donde se detallaron los problemas encontrados en la transformación de instrucciones en español a código fuente, las hipótesis y objetivos del trabajo, junto con los alcances y limitaciones que se plantearon. En el capítulo 2 se explicó las diferentes soluciones existentes para la tarea de generación de código a partir de lenguaje natural, diferentes soluciones que existen para el reconocimiento automático de voz, técnicas de aumento de datos para el entrenamiento de modelos, los ASTs genéricos y herramientas como la detección de entidades en texto, necesarias para los métodos implementados en este trabajo.

En el capitulo 3 se inicia con los detalles del método desarrollado en la investigación para la transformación de audios en español código fuente. En este capítulo se mostró de una forma general los tres componentes principales que nos permiten realizar la generación de código, los cuales son: El primer componente que transforma audios en español a texto, el segundo componente que pasa texto a código Python 3, y el tercer componente que traduce Python 3 a otros lenguajes de programación.

La metodología que se siguió es descrita en el capitulo 4, aquí se detalló la estructura de la metodología junto con las validaciones implementadas para realizar las pruebas de nuestro método. Luego en el capítulo 5 se describió cómo se implementó el método de este trabajo, mostrando las diferentes tecnologías usadas y las arquitecturas de los diferentes componentes que utiliza el método.

En el capítulo 6 se documentó los resultados obtenidos para cada una de las pruebas realizadas, el código fuente generado y el resultado del entrenamiento de los diferentes modelos que se desarrollaron. Por último, en el capitulo 7 se desarrollaron las conclusiones a las que se llegó con el trabajo realizado basado en los resultados obtenidos, para luego enfocarse en los trabajos futuros que pueden desarrollarse con los hallazgos de la investigación.

Capítulo 2

Marco teórico

Natural Lenguaje Processing (NLP) es un tipo de procesamiento de inteligencia artificial (IA) que tiene como objetivo que una computadora comprenda el lenguaje natural humano. Este extiende el análisis de texto más allá del simple procesamiento sintáctico al semántico. Este último es crítico y tiene gran importancia porque es una habilidad natural del ser humano [24].

La combinación de NLP con analizadores semánticos ha sido objeto de estudio para diseñar métodos de traducción de lenguajes naturales a lenguajes de programación. En este contexto, algunas investigaciones proponen modelos de aprendizaje profundos[1] que aprenden a efectuar el análisis semántico a partir de ejemplos con expresiones en lenguaje natural y el significado de estas en un lenguaje de programación.

El resto de este capítulo describe con más detalle los conceptos y herramientas necesarias para implementar un traductor de instrucciones en español (provenientes de audios) a múltiples lenguajes de programación.

2.1 Generación de código

Para la tarea de generación de código a partir de texto se han presentado varios modelos. Actualmente entre los más populares se encuentran los LLM (Large Language Models) como CodeGen [42]. Los modelos CodeGen son transformers autoregresivos con predicción de tokens. Los modelos estan entrenados en múltiples tamaños como 350 millones de parámetros, 2.7 billones, 6.1 billones, y 16.1 billones. CodeGen tiene una familia de modelos entrenados secuencialmente con tres conjuntos de datos llamados, ThePile, BigQuery y BigPython. ThePile contiene 825.18 GB de datos con texto para modelos de lenguaje. Parte de los datos provienen de repositorios GitHub con más de 100 estrellas. El conjunto de datos BigQuery es un sub conjunto del conjunto de datos público de Google con el mismo nombre. BigQuery tiene código en múltiples lenguajes de programación y para

[1] El término aprendizaje profundo es conocido como Deep Learning o DL por sus siglas en inglés.

el entrenamiento multilenguaje los lenguajes de programación utilizados son: C, C++, GO, Java, JavaScript y Python. Y por último, el conjunto de datos BigPython tiene una cantidad de datos considerable en el lenguaje de programación Python. Este conjunto de datos se creó con repositorios GitHub públicos, que no tuvieran información personal y sus licencias permiten usar el código.

CodeT [7] es otro LLM que demostró una mejora de 20% en versiones previas de estado del arte con respecto a la métricas pass@1 en el conjunto de datos HumanEval, al utilizar modelos pre entrenados para generar código y pruebas unitarias. Este método selecciona como mejor solución cuando esta pueda pasar más pruebas, y entre más diferentes pruebas genere.

Reflexion [56] actualmente es uno de los modelos de estado-del-arte con respecto al benchmark de código HumanEval, superando por 11% a GPT-4 en la métrica pass@1. Este método utiliza tres modelos, un Actor que genera texto y acciones, un Evaluador que da un puntaje a las salidas del Actor, y un modelo llamado Autorreflexión que genera señales de refuerzo verbal para ayudar al Actor a mejorar.

Lamentablemente estos grandes modelos requieren de gran capacidad de hardware y consumo eléctrico. Debido a este alto costo, se estima que el entrenamiento de GPT-3 puede generar aproximadamente 305% más emisiones de Co2 que un vuelo de pasajeros entre San Francisco y Nueva York [50]. Estos grandes modelos por lo general tienen buenos resultados en gran variedad de tareas, sin embargo, estudios han demostrado que pequeños modelos pueden tener mejores resultados si se concentra su capacidad en resolver una tarea específica en lugar de ser un modelo general que resuelva diferentes tipos de tareas [14, 21]. Por ejemplo, al utilizar Chat-GPT se puede ver que algunas tareas específicas como traducir un comando en español a Python no brinda los resultados adecuados por no tener el contexto adecuado. Con el fin de demostrar lo anterior, se realizó una consulta a Chat-GPT usando la palabra *retornar* (ver Figura 2.1).

Después de consultar instrucciones en Python. En un dominio de generación de código a partir de instrucciones en Español, lo esperado sería obtener como resultado simplemente la instrucción “return”.

Algunos analizadores semánticos se han desarrollado por medio de un tipo de red neuronal conocida como autoencoder [31]. Este tipo de red neuronal permite generar la sintaxis esperada por los lenguajes de programación con la ayuda de Árboles Sintácticos Abstractos (AST por sus siglas en inglés) [11, 62, 63] o un modelo común [44] similar al GAST. El modelo que utilizamos en nuestra investigación fue Tranx, debido a que aunque es un modelo pequeño en comparación a los LLM, ha demostrado tener buenos resultados en Python, el cual es uno de los lenguajes soportados por el GAST, necesario para la traducción de código en múltiples lenguajes de programación.

Entendi que estás buscando información sobre cómo usar `return` en Python. En Python, la declaración `return` se utiliza dentro de una función para especificar el valor que la función debe producir como salida. Cuando se encuentra una declaración `return`, la función deja de ejecutarse y el valor especificado se envía de vuelta al llamante de la función.

Aquí tienes un ejemplo básico:

python Copy code

```python
def sumar_numeros(a, b):
    resultado = a + b
    return resultado

resultado_suma = sumar_numeros(3, 4)
print(resultado_suma)  # Esto imprimirá 7
```

En este ejemplo, la función `sumar_numeros` toma dos argumentos `a` y `b`, calcula su suma y luego utiliza `return` para enviar el resultado (7) de vuelta al llamante.

También puedes tener múltiples declaraciones `return` en una función, dependiendo de diferentes condiciones:

python Copy code

```python
def verificar_edad(edad):
    if edad < 18:
        return "Eres demasiado joven."
    else:
        return "¡Bienvenido!"

resultado = verificar_edad(20)
print(resultado)  # Esto imprimirá "¡Bienvenido!"
```

En este caso, la función `verificar_edad` retorna mensajes diferentes dependiendo de la edad proporcionada como argumento.

Recuerda que una vez que se ejecuta una declaración `return` en una función, la función sale y no se ejecuta más código dentro de la función.

Figura 2.1: Resultado de Chat-GPT al enviar como entrada solo la palabra “retornar”.

En trabajos anteriores se han explorado herramientas para programar por medio de comandos en inglés como Talon, Dragon, Serenade o VoiceCode.io [30]. HyperCode [38] es otra herramienta que se desarrolló para programar en Java por voz utilizando el IDE IntelliJ. Una solución similar a la planteada en nuestro trabajo es la de Nizzad et al. [43], donde se utiliza un modelo para transformar audios en inglés a texto, y luego el texto a código. Aunque existen pocos trabajos de investigación sobre la generación de código a partir de audios en español, algunos trabajos como el realizado por Delgado et al. [9] proponen entornos de programación guiados por voz en Español.

Un caso que merece especial atención en esta investigación es Tranx, un autoencoder que usa el framework Abstract Syntax Description Language (ASDL) para aprender acerca del proceso de generación de código. Por lo cual, en la sección 2.1.1 se detallan las características este framework.

2.1.1 Tranx

Este framework ASDL es una gramática formal para definir ASTs. Un autoencoder es una red neuronal que se entrena con la intención de generar una salida similar a su entrada. Estas tienen una capa interna h que se encuentra oculta y describe un código usado para representar la entrada [16]. La red neural que utiliza el autoencoder tiene dos componentes (ver Figura 2.2):

Encoder : Puede ser representado por la función $h = f(x)$. f mapea x a h.

Decoder : Produce una reconstrucción $r = g(h)$. g mapea h a r.

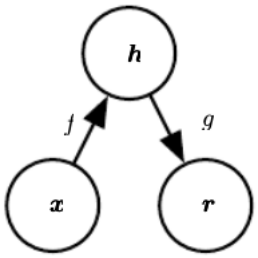

Figura 2.2: Estructura general de un autoencoder (tomado de 2.2)

Los autoencoders no aprenden a copiar perfectamente la entrada, y por lo general, están restringidos de manera tal que pueden copiar solo una aproximación. Estos normalmente aprenden propiedades útiles de los datos, debido a que sus modelos están forzados a dar prioridad a aspectos importantes de la entrada.

Tranx factoriza los pasos para generar un AST de forma secuencial por medio de acciones, las cuales pueden ser:

ApplyConstr(c): Esta acción aplica una producción al árbol sintáctico de tipo c, generando un nodo hijo por cada parámetro del constructor.

GenToken(v): El uso de esta acción permite agregar una frontera con un token v. Los campos de tipo string pueden tener múltiples tokens, estos se generan con una secuencia de acciones $GenToken$. Para finalizar la generación de tokens se agrega al final $GenToken[< /f >]$.

Reduce(): Con esta acción se marca la finalización de nodos hijos para un parámetro con cardinalidad opcional o múltiple.

Encoder

El encoder utilizado se describe en el trabajo de Yin [62], la cual consiste de una Red Neuronal Recurrente (RNN) LSTM[2] bidireccional [3], con la diferencia de que nuestro encoder lee una secuencia de palabras en español.

Una RNN lee una secuencia x como entrada, leyendo desde la primera palabra x_1 hasta la última x_t. Para la tarea de traducción se han usado redes neuronales recurrentes bidireccionales (BiRNN), la cual permite obtener información tanto de las palabras anteriores como las siguientes.

Por lo tanto, se puede definir que una BiRNN procesa hacia adelante la entrada desde x_1 a x_t, y calcula una secuencia de estados ocultos hacia adelante $\overrightarrow{\mathbf{h}}_i = f_{\text{LSTM}}\left(\mathbf{x}_i, \overrightarrow{\mathbf{h}}_{i-1}\right)$

Y además, una BiRNN procesa hacia atrás la entrada desde x_t a x_1, resultando en una secuencia de estados ocultos hacia atrás $\overleftarrow{\mathbf{h}}_i = f^{\leftarrow}_{\text{LSTM}}\left(\mathbf{x}_i, \overleftarrow{\mathbf{h}}_{i+1}\right)$

Con esto se puede obtener información de palabras posteriores y anteriores de cada palabra x_j, concatenando los estados ocultos hacia adelante $\overrightarrow{\mathbf{h}}_i$ con los estados ocultos hacia atrás $\overleftarrow{\mathbf{h}}_i$ de tal forma que $\mathbf{h}_i = [\overrightarrow{\mathbf{h}} : \overleftarrow{\mathbf{h}}_i]$

Decoder

El decoder utiliza una RNN LSTM, con la diferencia de que no es bidireccional. Para calcular el estado interno en un paso **t** se utiliza:

$\mathbf{s}_t = f_{\text{LSTM}}\left([\mathbf{a}_{t-1} : \mathbf{c}_t : \mathbf{p}_t : \mathbf{n}_{f_t}], \mathbf{s}_{t-1}\right)$

f_{LSTM} es la función de actualización del LSTM, [:] denota la concatenación de vectores. $\mathbf{a}_{t-1}$ es la acción anterior, $\mathbf{c}_t$ es el vector con el contexto obtenido a partir de la entrada procesada por el encoder y soft attention. $\mathbf{p}_t$ es un vector que codifica la información del nodo padre y $\mathbf{n}_{f_t}$ indica si el nodo es de tipo terminal o no terminal. Y por último, $\mathbf{s}_{t-1}$ es el estado oculto anterior calculado por el LSTM.

[2]Las redes neuronales recurrentes de memoria a largo y corto plazo se conocen como Long Short Term Memory (LSTM) en inglés.

2.2 Reconocimiento Automático de Voz

El reconocimiento automático de voz (ASR) es la tarea de procesar en tiempo real un lenguaje hablado y transformarlo a un texto que pueda ser interpretado en una máquina. En la actualidad se ha logrado mejorar el reconocimiento de voz incluso con ruidos de fondo, diferentes acentos y tonos [26].

La popularidad de asistentes virtuales como Siri, Alexa o Cortana ha incrementado la importancia de los sistemas ASR en los últimos años. En la actualidad, los dispositivos móviles, vehículos y otros sistemas usados en el hogar se utilizan para realizar tareas como, enviar un mensaje de texto, reproducir música con comandos de voz y leer texto por medio de un asistente. Debido a esto, el uso del teclado o ratón ya no es indispensable porque se pueden realizar las actividades en estos dispositivos por comandos de voz.

Las aplicaciones de reconocimiento de voz se pueden clasificar en dos categorías: las aplicaciones que mejoran la comunicación humano a humano y las que mejoran la comunicación humano a máquina.

Las aplicaciones de comunicación humano a máquina más populares son las siguientes [64]:

Búsqueda por voz: Permiten buscar información de diferente índole por medio de voz. Estas aplicaciones han sido muy populares en dispositivos móviles.

Asistentes digitales personales (PDA): Permiten realizar tareas como marcar un número telefónico, agendar una cita, responder una pregunta o reproducir música por medio de comandos de voz.

Juegos de video: Se pueden utilizar ASRs para hablar con los personajes, obtener información y realizar tareas en los juegos de video.

Sistemas de entretenimiento: Los sistemas en salas de estar o en vehículos tiene funcionalidades similares. En estos sistemas el usuario también puede usar el habla para reproducir música, pedir información o controlar el dispositivo.

Algunas de las soluciones que se pueden encontrar para la transformación de audios a texto son:

OpenAI Whisper : Whisper es un modelo desarrollado por OpenAI que usa una arquitectura de transformers encoder-decoder. Es un ASR entrenado con 680,000 horas de ejemplos multilenguaje y multitareas de manera supervisada obtenidas de la web. Soporta tareas como traducción de múltiples lenguajes a inglés y transcripciones en múltiples lenguajes [52].

Este trabajo muestra que el uso de conjunto de datos grandes y diversos puede mejorar la detección de acentos, ruido de fondo y lenguaje técnico, con lo cual, se pueden obtener buenos resultados sin la necesidad de técnicas de autosupervisión y autoentrenamiento.

Google mSLAM :Este modelo multilenguaje fue pre entrenado con gran cantidad de datos de audios y texto con diferentes lenguajes sin etiquetas. Este utiliza wv2-BERT para audios, en conjunto con SpanBERT para texto, junto con CTC (Connectionist Temporal Classification) para aprender en un único modelo y poder representar señales de audio y texto en una única representación espacial compartida. Los conjuntos de datos que utilizaron para reportar resultados fueron VoxPopuli con 14 lenguajes, MLS-10Hr y con Babel reportaron 5 lenguajes para la tarea de Babel-ASR [4].

Kaldi : Es una herramienta gratuita de código abierto para investigaciones en reconocimiento de voz, desarrollada en C++ y con soporte para sistemas operativos Unix y Windows [51].

La herramienta brinda diferentes tipos de arquitecturas DNN-HMM como nnet1, nnet2, nnet3 y chain (una implementación especial de nnet3) [1].

IBMWatson : IBM cuenta con un servicio llamado IBM Watson™ Speech to Text el cual provee APIs para la transcripción de audios. Adicionalmente puede producir información sobre diferentes detalles del audio. El servicio puede ejecutarse en la nube o puede instalarse en servidores propios del usuario. El servicio tiene dos modelos llamados previous-generation y next-generation. Según la documentación dada por IBM, el modelo next-generation ofrece mejor rendimiento y precisión en las transcripciones [23].

El servicio permite personalizar los modelos por medio de interfaces. El modelo de lenguaje se puede personalizar para ampliar el vocabulario de un modelo base con un dominio específico. El modelo acústico se puede personalizar para adaptar el modelo base a las características acústicas de audios específicos. Adicionalmente el modelo de lenguaje soporta gramáticas para restringir las frases que el servicio puede reconocer [22].

Microsoft Azure Speech to Text : Este modelo es parte de los servicio de Azure Cognitive de Microsoft. Esta funcionalidad usa la misma tecnología para Cortana y los productos de Office. La funcionalidad se puede usar por medio del SDK Speech, una API REST y con el CLI Speech [41].

El servicio utiliza un modelo de lenguaje universal como base, el cual es entrenado con datos propiedad de Microsoft. Este modelo base es pre entrenado con dialectos y fonéticas que representan varios dominios de uso común. El servicio también permite construir un modelo personalizado entrenándolo con datos adicionales para un dominio especifico requerido [40].

Google Speech to Text : Google Cloud cuenta con el servicio Speech To Text para el reconocimiento automático de voz por medio de algoritmos de redes neuronales usando deep learning. El servicio se puede utilizar por medio de las APIs en la nube o por medio de Speech-to-Text On-Prem el cual permite usar la tecnología en los servidores del usuario [17].

El servicio también permite seleccionar entre diferentes modelos de machine learning entrenados para tipos específicos de audios y fuentes. Esto puede permitir mejores resultados al procesar el audio con un modelo entrenado para reconocer un audio para un tipo de dominio específico [18].

2.3 Analizador semántico

El análisis semántico en este contexto es la tarea de mapear un enunciado de lenguaje natural en una representación formal [29]. Esta tarea identifica las partes significativas en los enunciados para transformarlos en alguna estructura de datos que pueda ser manipulada por una computadora para realizar tareas a un alto nivel [5]. Algunas de las arquitecturas más comunes para desarrollar analizadores semánticos son las siguientes:

Base de conocimientos : Estas bases usan un conjunto predefinido de reglas o una base de conocimientos para obtener una solución.

Sin supervisión : Tienden a requerir una mínima intervención humana para ser funcionales mediante el uso de recursos existentes que se pueden usar como inicio para una aplicación o un dominio en particular.

Supervisado : Estos sistemas necesitan una cantidad suficiente de datos con anotaciones manuales de elementos para que se puedan aplicar algoritmos de aprendizaje automático.

Semi-supervisado : La anotación manual suele ser muy costosa y no produce suficientes datos para capturar todos los detalles del dominio. En estos casos, se puede expandir automáticamente el conjunto de datos en el que se entrenan los modelos, usando salidas generadas por la misma máquina o utilizando un modelo existente que corrige su salida por medio de interacción humana. En muchos casos, se utiliza un modelo de un dominio particular y se adapta a un nuevo dominio.

En esta investigación se utilizó la arquitectura supervisada para desarrollar el analizador semántico.

2.4 Aumento de datos

El aumento de datos es un concepto que se refiere a la creación de datos sintéticos de entrenamiento adicionales, similares a los reales, mediante la aplicación de transformaciones a ejemplos, sin cambiar la etiqueta original del ejemplo. La transformación de datos se expresa como $q(\hat{x} \mid x)$, tal que $\hat{x}$ son los datos sintéticos creados a partir de un ejemplo original x. Se requiere que la transformación que genera los datos sintéticos mantenga la etiqueta del ejemplo para que sea válida.

Un conjunto de datos etiquetados adicionales que son generados a partir de un conjunto de datos de entrenamiento etiquetado original se conoce como data augmentation supervisado [60]. El aumento automático de datos se usa comúnmente en la solución de problemas asociados con la visión por computadora y el reconocimiento de voz. Sin embargo, también puede ayudar a entrenar modelos más robustos, particularmente cuando se utilizan conjuntos de datos pequeños [59].

2.5 AST Génerico (GAST)

El GAST es un método propuesto por Leitón et al. [32, 33] que permite mapear el código de diferentes lenguajes de programación, como Scratch y Python, a una estructura común que luego permite transformarlos a otros lenguajes como Java, C# y LIE++[3]. Esta estructura común es un AST genérico que contiene un conjunto de elementos que permiten modelar más de un lenguaje de programación.

Un AST es una estructura de datos que representa la gramática de un programa [57]. La representación de este programa se hace con una jerarquía de nodos no terminales y terminales (también conocidos como hojas), la cual está basada en la sintaxis de un lenguaje de programación [31, 19]. El GAST está basado en los ASTs, con la diferencia de que usa el estándar MOF 2.0 [46], el cual define un árbol sintáctico genérico que soporta la mayoría de variantes de los lenguajes de programación más comunes.

El funcionamiento del GAST está basado en la obtención de datos provenientes de los archivos de código, la generación de los AST específicos del lenguaje que corresponde a los archivos, el mapeo de los AST específicos a la estructura genérica, y por último, la presentación en formato JSON del lenguaje genérico. En la Figura 2.3 se muestra la secuencia de las etapas del flujo de traducción.

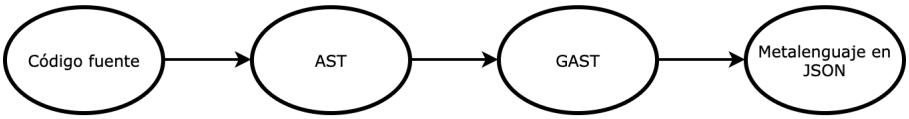

Figura 2.3: Etapas en el flujo de traducción usando el GAST, junto con sus respectivos resultados.

Este árbol sintáctico genérico se basa en 3 paquetes fundamentales los cuales son:

Sintáctico : Modelado que hace referencia a los elementos y aspectos de sintaxis, por ejemplo, sentencias anidadas, como los "if", la definición de clases, métodos, atributos, herencia, interfaces y llamadas a funciones.

[3]LIE++ es un lenguaje en español creado en conjunto entre varios investigadores del Tecnológico de Costa Rica y funcionarios de la Fundación Omar Dengo.

Semántico : Estos son los aspectos semánticos del código fuente, por ejemplo, el alcance que tiene una sentencia dentro de un método, una variable dentro de un "if" o el alcance de un bloque especifico.

Recursos : Realiza el modelado de la estructura de los archivos de un proyecto. Por ejemplo los archivos de código fuente y la localización de bibliotecas.

2.6 Named Entity Recognition

Name Entity Recognition o NER por sus siglas en ingles, es la tarea de identificar objetos importantes en un texto dado [53]. NER juega una parte importante en la extracción de información y la identificación de entidades como nombres de personas, lugares y organizaciones. También se han logrado identificar otras entidades de dominios específicos como en el área médica o legal [34].

Algunos de los softwares más utilizados para Name Entity Recognition son StanfordNLP, NLTK, SpaCy y OpenNLP. Estos son usados en diferentes tareas y cada uno tiene sus propias carecteristicas y opciones. StanfordNLP, OpenNLP y Gate son herramientas para Java, mientras que Spacy y NLTK son bibliotecas para Python.

StanfordNLP ha demostrado tener mejores resultados en identificar organizaciones, mientras que para otras entidades no hay gran diferencia con respecto a otras herramientas [54]. En nuestro caso, las entidades a detectar son nombres de variables, funciones y clases. Debido a que no se encontraron estudios al realizar la investigación que compararan Software NER para esta tarea en específico, se utilizó Spacy para la detección de estas entidades por tener una amplia y clara documentación con ejemplos en su sitio oficial.

Spacy es una biblioteca de código abierto para tareas de NLP. Usa modelos de redes neuronales y Deep learning para los diferentes componentes que posee para obtener resultados del estado del arte [49].

Capítulo 3

Diseño del Método

En este capítulo se presenta el diseño del analizador semántico para generar código a partir de instrucciones por voz en español y los diferentes componentes utilizados. El modelo consta de 3 componentes principales: Audio en español a texto, texto en español a Python 3, y Python 3 a múltiples lenguajes de programación (Java y C#).

3.1 Estructura del diseño

En esta sección se presenta de una forma general cómo está estructurado el modelo realizado para la transformación de audios en español a diferentes lenguajes de programación. En la Figura 3.1 se presenta un diagrama con la estructura separada por los tres componentes principales. A continuación se presentan los pasos implementados en cada uno de estos componentes.

3.1.1 Audio en español a texto

En este componente se procesan audios en español, ya sea que provengan de archivos .wav o algún micrófono, para luego ser transformados con en un texto adecuado que pueda ser procesado por el siguiente componente que transforma texto en español a Python 3.

1. Como primer paso, se carga en memoria el audio en español (ver flecha 1 en Figura 3.1). El método soporta la carga de múltiples archivos, utilizado para las pruebas en esta investigación, y también soporta la carga de un solo archivo para fines prácticos y pruebas individuales.

2. Los audios cargados en memoria son enviados luego a un ASR (ver flecha 2 en Figura 3.1). En este trabajo se utilizó la API de Azure Speech to Text, un ASR que soporta el idioma español y permite transformar los audios en español a texto plano.

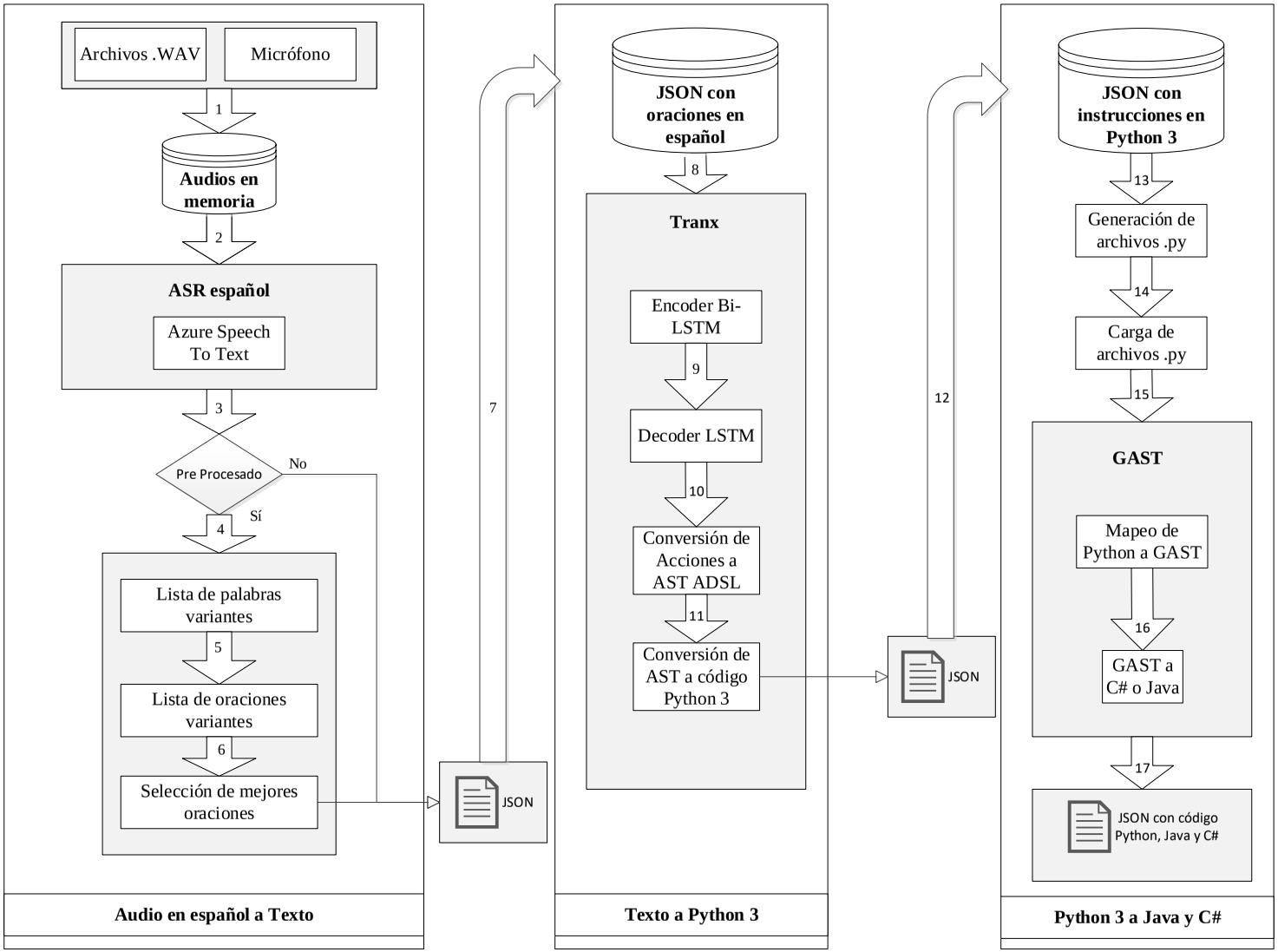

Figura 3.1: Componentes utilizados para generar el analizador semántico que transforma instrucciones en español a múltiples lenguajes de programación.

3. Para la transformación de audio a texto se implementaron dos versiones, sin procesado y con procesado. En este paso se decide cuál versión usar (ver flecha 3 en Figura 3.1). La versión sin procesado envía directamente el resultado del ASR al siguiente componente. La versión con procesado ejecuta una serie de algoritmos extra antes de ser enviado al siguiente componente.

4. El primer algoritmo para el pre procesado consiste en obtener una lista de posibles variantes de palabras de la oración resultante del ASR (ver flecha 4 en Figura 3.1).

5. El segundo algoritmo genera una lista de oraciones variantes con base en la lista de palabras generadas por el primer algoritmo (ver flecha 5 en Figura 3.1).

6. El tercer algoritmo obtiene la mejor oración de la lista de oraciones variantes generadas en el segundo algoritmo (ver flecha 6 en Figura 3.1). El resultado es guardado en un archivo Json para luego ser usado en el siguiente componente.

Para más información sobre la implementación y algoritmos diseñados ver la sección 3.3.

3.1.2 Texto en español a Python 3

En este componente se utiliza el modelo Tranx entrenado con un conjunto de datos en español (ver 3.2 para más detalle). El texto generado por el componente anterior se carga en Tranx para generar código por medio de un AST ADSL. A continuación se describen los pasos ejecutados:

1. Se carga en memoria un archivo JSON con todas las oraciones en español resultantes del componente audio a texto (ver flecha 7 en Figura 3.1).

2. Las oraciones en español se envían al Encoder de Tranx para generar un vector con la información importante del texto en español (ver flecha 8 en Figura 3.1).

3. Luego el vector generado por el Encoder se pasa por el Decoder de Tranx para generar una serie de acciones que representan un AST ADSL de forma secuencial (ver flecha 9 en Figura 3.1).

4. Las acciones generadas por el Decoder se convierten a un AST ADSL (ver flecha 10 en Figura 3.1).

5. El AST ADSL resultante se recorre para ser transformado en código Python 3 con un formato en texto plano (ver flecha 11 en Figura 3.1). El código Python 3 generado se guarda en un JSON para luego ser usado en el siguiente componente, Python 3 a Java y C#.

3.1.3 Python 3 a Java y C#

El último componente del analizador semántico consiste en pasar el código Python 3 a un Árbol Sintáctico Abstracto Genérico. Con ayuda del GAST se logra transformar primero el código generado por Tranx a un AST genérico, para luego ser transformado a otros lenguajes de programación. En este trabajo se transformó a Java y C#. A continuación se describe los pasos necesarios:

1. Se cargan en memoria todas la instrucciones en Python 3 generadas por Tranx (ver flecha 12 en Figura 3.1).
2. Por cada instrucción se genera un archivo con extensión .py en el sistema de archivos del sistema operativo con una estructura valida de Python 3 (ver flecha 13 en Figura 3.1).
3. Se utiliza la ruta donde están todos los archivos .py generados para ser cargados por el GAST (ver flecha 14 en Figura 3.1).
4. Cada archivo .py es mapeado a la estructura del GAST, creando un AST genérico (ver flecha 15 en Figura 3.1).
5. Cada AST genérico luego es transformado al lenguaje de programación requerido (ver flecha 16 en Figura 3.1).
6. Una vez transformados los ASTs a código fuente, se une el resultado en Java, C# y Python 3 en un Json que contiene la instrucción en español original y el resultado en código fuente de los 3 lenguajes (ver flecha 17 en Figura 3.1).

3.2 Texto en español a Python 3

Para generar un AST, el modelo Tranx utiliza un conjunto de producciones dadas por la gramática ASDL de Python, comenzando por un nodo inicial que puede tener múltiples nodos hijos.

Una gramática ASDL puede tener dos producciones básicas: Tipos y constructores. Además puede tener tipos compuestos definidos por un conjunto de constructores. Por ejemplo, en la Figura 3.2, el constructor *BinOp* en Python 3 es de tipo *Expr* (expresión) y representa una operación binaria. Los constructores pueden tener múltiples parámetros, los cuales también son tipados. Un parámetro de tipo compuesto puede ser instanciado por constructores del mismo tipo. También hay parámetros de tipo primitivo, los cuales pueden almacenar valores.

El Tranx que se muestra en la Figura 3.2 utiliza la gramática definida en un archivo de texto (en este caso el ASDL de Python 3) para pasar el lenguaje natural a una representación formal del lenguaje de programación. Con base en esta gramática, Tranx selecciona

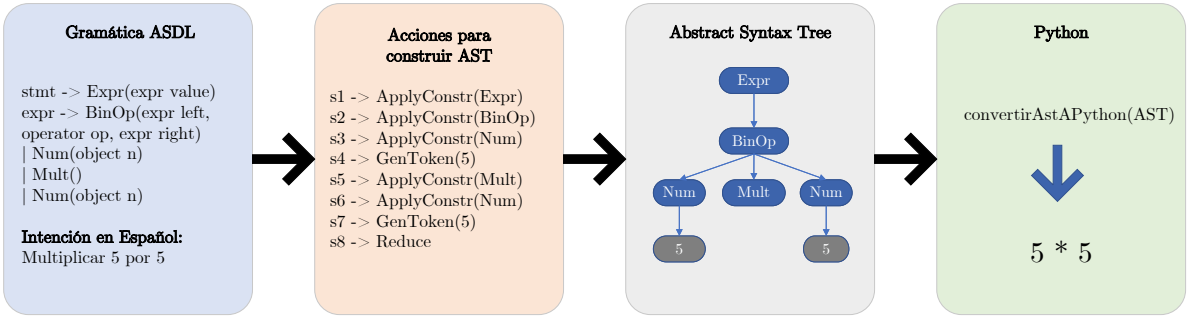

Figura 3.2: ASDL para transformar la intención "Multiplicar 5 por 5" en un AST, que luego puede ser convertido a Python.

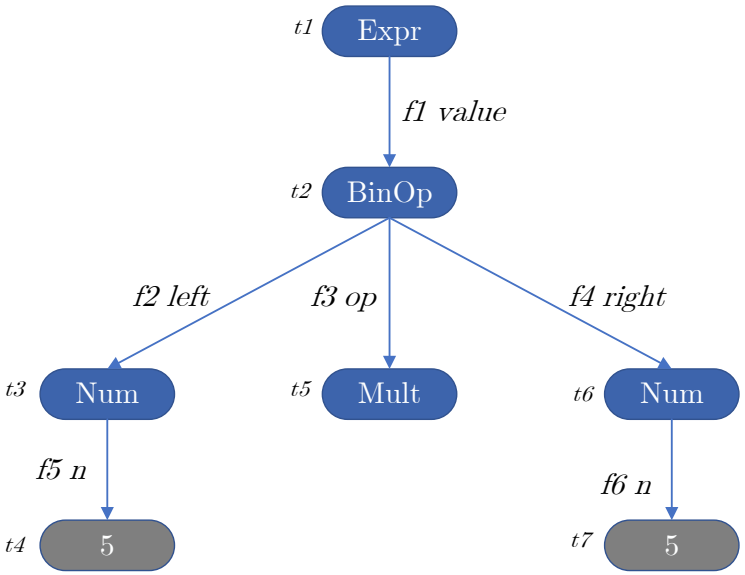

Figura 3.3: AST generado con base en la gramática ASDL de Python 3 para multiplicar 5 por 5.

acciones en cada estado de la RNN para ir construyendo de forma secuencial un AST. Por último, el AST puede convertirse a código Python recorriendo el AST (Tranx utiliza la librería Astor para generar el código fuente).

Al igual que en los trabajos de Yin [62, 63], se utiliza una RNN como Decoder, el cual retorna en forma secuencial por cada paso en el tiempo t una acción del modelo gramático. Por ejemplo, en la Figura 3.3 se puede ver un AST generado con base en la gramática ASDL de Python 3, y que representa la instrucción en Español de "Multiplicar 5 por 5". Cada paso en el tiempo **t** contiene una acción del modelo gramático. Cada acción es etiquetada por su campo **f** y representa su frontera.. El estado interno de la RNN permite computar la probabilidad del siguiente acción a generar.

3.3 Audio en español a Python 3

Otra de las hipótesis que se plantearon en este trabajo es la de poder generar código a partir de audios en Español, utilizando modelos ya entrenados de Speech To Text, para que la salida de estos modelos se utilice como entrada del modelo que genera código Python a partir de texto en español.

En este capitulo se describe el componente que realiza esta tarea, necesario para el modelo de traducción de español a diferentes lenguajes de programación. Este componente es el primero de los 3 componentes del modelo, por lo que la salida generada por este, afecta el resultado de los siguientes dos componentes.

En este trabajo desarrollamos dos versiones de este primer componente, la primera es una versión sin procesado que simplemente implementa un modelo Speech To Text que soporta el lenguaje Español. Mientras que la segunda versión que tiene un preprocesado, usa un algoritmo que transforma el texto generado antes de ser enviado al modelo que genera código a partir de texto. Estas dos versiones se explican con detalle en este capitulo.

3.3.1 Versión sin preprocesado

Actualmente existen diferentes soluciones para la transformación de audios a texto. Algunas soluciones son de código abierto y pueden ser utilizados en un dispositivo sin internet, mientras que otras soluciones se ejecutan en servidores en la nube por medio de APIs de alto nivel.

Diferentes estudios han comparado estos modelos con la métrica WER (Word error rate) con diferentes conjuntos de datos en diferentes dominios. Sin embargo, en este trabajo el objetivo no es saber cuál modelo Speach To Text tiene mejores resultados en la métrica WER, si no que se utiliza como un primer paso, el cual transforma un audio en español a un texto que el modelo que entrenamos con el conjunto de datos EsPython pueda entender para generar código en Python.

3.3.2 Versión con preprocesado

Al realizar las pruebas con la primera versión, la cual no incluía ningún preprocesado del texto antes de ser enviado a el modelo sintáctico, se encontró que existían algunos problemas que podían degradar el resultado. Uno de los problemas es que el modelo Speech To Text puede generar un texto que tiene sentido, sin embargo, entre el texto generado y el audio de entrada hay palabras homófonas, las cuales son palabras que tienen el mismo sonido, pero que no se escriben igual, ni significan lo mismo. Parte de este problema se debe a que los ejemplos generados usaban en ocasiones abreviaturas, por ejemplo, algunas oraciones contenían la palabra “var”, la cual es una abreviación de la palabra “variable”, y que fue usada para definir el nombre de algunas variables, sin

embargo, el modelo de Speech To Text lo interpretaba como la palabra "bar", la cual suena igual pero tiene diferente significado y se escribe diferente. Otro de los problemas es que el modelo Tranx requiere que las oraciones recibidas como entrada tengan entre acentos graves (‘) los nombres de variables, funciones o clases, y esto no era generado en el modelo Speech To Text. Estos problemas se minimizaron aplicando una función de preprocesado para cada ejemplo del conjunto de datos, la cual se puede definir como $y = f(x)$, donde y es la mejor oración resultante que nuestra función f pudo generar para el texto x, generado por el modelo Speech To Text.

En el algoritmo 1 se pueden ver los pasos implementados para crear la función de preprocesado. El algoritmo lo podemos dividir en tres pasos principales: Crear una lista de variantes para cada palabra de la oración original (linea 4), crear todas las oraciones posibles con la lista de variantes (linea 5) y obtener la mejor oración de la lista de oraciones generadas (linea 6). Esto se realiza para cada oración que esté en la lista que recibe como parámetro (linea 2) y el resultado se agrega a una lista final de oraciones preprocesadas. Además, cada oración se pasa por una función que remueve acentos con ayuda de la biblioteca Unidecode de Python (linea 3), la cual permite simplificar la generación de oraciones al pasar el texto a formato ASCII. A continuación se explica en detalle los algoritmos implementados para los tres pasos que componen el algoritmo principal 1 (en el Apéndice D se encuentra la implementación completa en Python 3).

Algoritmo 1: Algoritmo de preprocesado

Datos: $listaOraciones \geq 0$
Resultado: $listaOracionesPreprocesadas$

```
1 listaOracionesPreprocesadas ← [];
2 para oracion en listaOraciones hacer
3     oracion ← removerAcentos(oracion);
4     variantesDePalabras ← crearListaDeVariantesDePalabras(oracion);
5     oraciones ← todasPosiblesOraciones(variantesDePalabras);
6     mejorOracion ← obtenerMejorOracion(oracion, oraciones);
7     agregar mejorOracion a listaOracionesPreprocesadas;
8 devolver listaOracionesPreprocesadas
```

Crear una lista de variantes para cada palabra de la oración: En este algoritmo (ver algoritmo (2)) se recibe como parámetro la oración resultante del modelo Speech To Text. Para crear una lista de variantes para cada palabra se generó un diccionario de fonemas y una lista de letras o combinaciones de letras que pueden tener el mismo sonido que representa el fonema (linea 2). Un fonema es la unidad mínima del lenguaje oral, ya que se son los sonidos del habla que permiten diferenciar entre las palabras de una lengua, por ejemplo: */t/* y */l/* en pata y pala, */a/* y */o/* en sal y sol [61]. En ocasiones hay palabras que suenan igual que otras, pero que tiene distinto significado y puede tener distintas letras. A estas palabras se le llaman

homófonos [13], por ejemplo: *tubo* y *tuvo*, las cuales pueden ser representadas por el mismo fonema */b/* para *b* y *v*. Este algoritmo busca generar estas posibles variantes.

Las palabras originales de la oración se agregan como parte de la lista de variantes (linea 3 a 6). Para cada palabra de la oración revisamos si hay letras que puedan tener el mismo sonido (linea 7 a 10) con ayuda de la lista de reglas definida al inicio del algoritmo (linea 2). Si hay letras que pueden reemplazarse en la palabra y mantener su sonido, se crea una nueva variante de la palabra y se agrega a una lista (linea 11 a 13). Al realizar esto para cada palabra obtenemos una lista de variantes para cada palabra de la oración original. La implementación en Python 3 de este algoritmo se encuentra en el Apéndice A.

Algoritmo 2: Algoritmo para crear lista de variantes de palabras

```
   Datos: oracion
   Resultado: listaDeVariantesDePalabras
1  listaDeVariantesDePalabras ← [];
2  listaDeReglas ← [letras, [letrasConMismoSonido]];
3  palabrasDeOracion ← obtenerListaDePalabras(oracion);
4  para palabra en palabrasDeOracion hacer
5      listaDeVariantesDePalabras agregar [palabra];
6  para i hasta tamaño de palabrasDeOracion hacer
7      palabra ← palabrasDeOracion[i];
8      para regla en listaDeReglas hacer
9          si letras en regla está en palabra entonces
10             para variante en [letrasConMismoSonido] en regla hacer
11                 varianteDePalabra ← palabra.remplazar(letras, variante);
12                 listaDeVariantesDePalabras[i] agregar [varianteDePalabra];
13 devolver listaDeVariantesDePalabras
```

Crear todas las oraciones posibles con la lista de variantes: Utilizando las variantes generadas para cada palabra de la oración original, se genera una lista de oraciones. Para esto se definió el algoritmo 3, que de forma recursiva genera el texto con base en la lista de de palabras la cual podemos imaginar como grafos. Este algoritmo recibe como entrada la lista de variantes de palabras generadas en el paso anterior, el índice actual de la lista de palabras (se inicia con cero), una cadena de caracteres que representa una posible oración y una lista donde se guardarán las oraciones generadas. Como primer paso se comprueba si el índice ya sobrepasó la cantidad de palabras que puede tener la oración, en caso de ser así, se agrega la oración generada en la lista de resultados (linea 1 a 3). Sí aún quedan palabras de la oración por recorrer, se recorren todas las variantes de una palabra en especifico, llamando de forma recursiva el algoritmo 3, con la diferencia de que se aumenta el índice en uno para proseguir con la siguiente palabra y concatena la palabra variante a la cadena

de caracteres que va guardando el resultado de la oración generada (linea 5 a 7). Por último, cuando se han generado todas las oraciones se retorna una lista con las oraciones generadas (linea 8 a 10). En la Figura 3.4 se puede ver un ejemplo de como se generan las oraciones, las palabras son vértices de un grafo dirigido, y las aristas siempre apuntan a la palabra siguiente. En el Apéndice B se encuentra la implementación completa en Python 3.

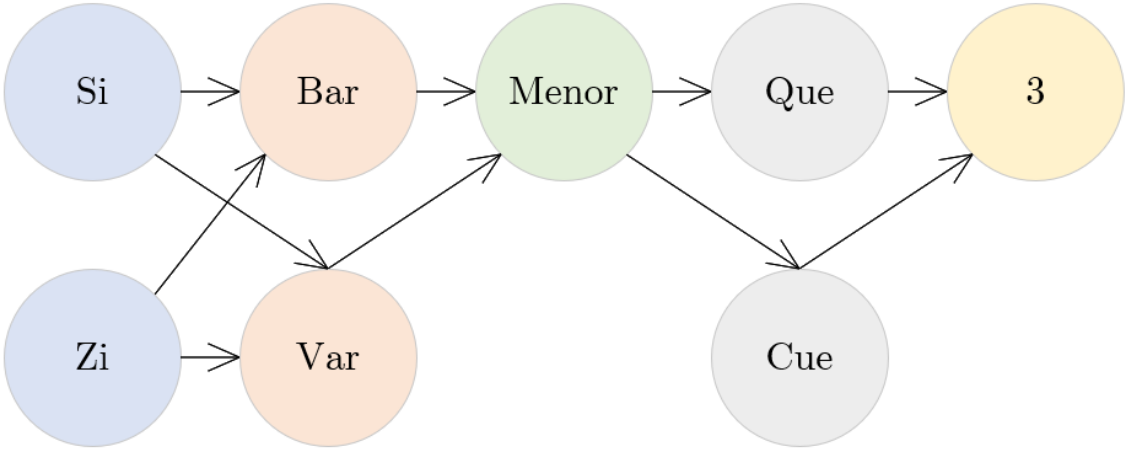

Figura 3.4: Ejemplo de un posible grafo usado para la generación de oraciones.

Algoritmo 3: Algoritmo para crear todas las oraciones posibles

Datos: *listaDeVariantesDePalabras*,*indice*,*texto*,*listaDeOraciones*
Resultado: *listaDeOraciones*

```
1 si indice ≥ tamañodelistaDeVariantesDePalabras entonces
2     listaDeOraciones agregar texto;
3     devolver
4 para i hasta tamaño de listaDeVariantesDePalabras[i] hacer
5     funcionCrearTodasOraciones(listaDeVariantesDePalabras, indice +
        1, texto + "" + listaDeVariantesDePalabras[indice][i], listaDeOraciones);
6 si indice = 0 entonces
7     devolver listaDeOraciones
8
```

Obtener la mejor oración de la lista de oraciones generadas: Con la lista de oraciones generadas en el paso anterior se selecciona la mejor oración generada. El algoritmo 4 primero genera una estructura que contiene la mejor oración vista, esta estructura contiene el texto de la oración, el puntaje BLEU de la oración y la cantidad de variables que hay en la oración. Se asume inicialmente que la mejor oración es la oración original dada por el Modelo Speach To Text, que su puntaje

BLEU es 0 y la cantidad de variables es 0 (linea 1). BLEU nos permite obtener un puntaje al comparar una posible traducción de un texto con una o más referencias de la traducción de ese texto. Esta métrica nos da un resultado desde 0 a 1, donde 1 es una traducción idéntica a la traducción de referencia [48].

Luego se recorren todas las oraciones generadas y para cada una se remplazan símbolos como "+" por "*más*", "*" por "*multiplicar*" y se quitan espacios en blanco (linea 3). Se comienza con la búsqueda de variables que tenga la oración. El proceso de búsqueda de variables se realiza con un modelo NER con la librería gratuita de código abierta spaCy (linea 6). Un modelo NER (Name Entity Recognition), nos permite etiquetar objetos del "mundo real" como personas, empresas o lugares. En nuestro caso utilizamos esta funcionalidad para etiquetar nombres de variables, clases o funciones, lo cual nos permite enviar el texto de entrada en un formato adecuado para el modelo Tranx que utilizamos para pasar español a Python. En cada oración detectamos si removiendo comas (linea 10 a 12) o uniendo palabras con números (linea 13 a 17) aumenta el numero de variables encontradas. Si luego de realizar este proceso no se encuentran más variables, funciones o clases, se termina la búsqueda (linea 19). El siguiente paso consiste en remover las comas que tenga la oración y obtener las entidades de la oración modificada (linea 22 y 23). Todas las entidades encontradas por el modelo NER se encierran entre acentos graves (‘) para el modelo Tranx (linea 23 a 26). Por último, si la cantidad de variables encontradas por el modelo NER es superior a la cantidad de variables encontradas en la última mejor oración vista, se asigna como mejor oración la actualmente evaluada (linea 27 y 28). Si la cantidad de variables es igual a la última mejor oración vista, se compara adicionalmente si el puntaje BLEU es mejor, utilizando como referencia la oración original. En caso de que el puntaje BLEU sea mejor a la última mejor oración vista, se asigna la oración actual como mejor (linea 30 a 32).

Luego de revisar todas las oraciones, se retorna la última mejor oración vista como resultado (linea 35). En el Apéndice C se encuentra la implementación completa en Python 3.

3.4 Transformación de Python 3 a Java y C#

Para la transformación de Python 3 a Java y C# se utilizó un AST genérico para representar el resultado del modelo Tranx, y luego ser transformado al lenguaje de programación requerido. En este componente primero se realiza una generación de archivos .py a partir de los resultados del modelo Tranx que son guardados en un archivo .json. Una vez generados estos archivos se procede a transformar los archivos a la estructura del GAST por medio del AST de Python. Esta acción es realizada por el componente Mapper que forma parte del GAST. Luego de tener nuestro código representado en el formato del GAST, el componente Printer recorre el AST y genera el código deseado en Java o C#.

Algoritmo 4: Algoritmo para obtener la mejor oración

Datos: *oracionOriginal*,*listaDeOraciones*
Resultado: *mejorOracion*

```
1  mejorOracion ← MejorOracion(oracionOriginal, 0, 0);
2  para oracion en listaDeOraciones hacer
3      nuevaOracion ← remplazarSimbolos(oracion);
4      detenerBusqueda ← Falso;
5      mientras detenerBusqueda sea Falso hacer
6          entidades ← obtenerEntidades(nuevaOracion);
7          para entidad en entidades hacer
8              final ← entidad.indiceFinal;
9              si nuevaOracion[final : final + 2] = "," entonces
10                 nuevaOracion ← removerComa(nuevaOracion);
11                 detener;
12             en otro caso
13                 si nuevaOracion[final + 1 : final + 2] es un numero entonces
14                     nuevaOracion ← removerEspacio(nuevaOracion);
15                     detener;
16             detenerBusqueda ← Verdero;
17     nuevaOracion ← removerComas(nuevaOracion);
18     entidades ← obtenerEntidades(nuevaOracion);
19     para entidad en entidades hacer
20         nuevaOracion ← agregarComillas(nuevaOracion, entidad);
21     si cantidad de entidades > cantidad de mejorOracion.variables entonces
22         mejorOracion ← MejorOracion(nuevaOracion, cantidad de
           entidades, bleuDeNuevaOracion);
23     en otro caso
24         si cantidad de entidades = cantidad de mejorOracion.variables Y
           bleuDeNuevaOracion > mejorOracion.bleuScore entonces
25             mejorOracion ← MejorOracion(nuevaOracion, cantidad de
               entidades, bleuDeNuevaOracion);
26 devolver mejorOracion
```

Capítulo 4

Metodología

En este capítulo se presenta la metodología que se siguió en la investigación para validar los tres componentes principales del modelo para generar código en múltiples lenguajes de programación a partir de audios en español. En primer lugar se muestra la estructura general de la metodología, con una explicación general de las validaciones que se realizaron y la relación que estas tienen con los objetivos, hipótesis y otras validaciones. Luego se explica con detalle cada una de estas validaciones, pasos necesarios, junto con los parámetros, herramientas o hardware utilizado.

4.1 Estructura de la metodología

En está sección se muestra el flujo general que se siguió para las validaciones realizadas en la investigación. Debido a que el modelo de esta investigación se divide en tres componentes principales, se siguió un orden de pruebas que permitiera probar primero los componentes que no tuvieran dependencias de otros componentes. Por lo tanto, como se puede ver en la Figura 4.1, se validó inicialmente el componente que transforma texto en español a Python 3, seguido por el que transforma audios en español Python 3, y por último la combinación de todos los componentes para validar la transformación de audios en español a múltiples lenguajes de programación. En la Figura 4.1 se detallan los pasos de las validaciones.

Validar Tranx con el dataset EsPython: En este proceso se realizan pruebas para validar que el modelo Tranx pueda generar código Python 3 a partir de texto en español si es entrenado con un conjunto de datos en español. En la investigación se generó de forma manual un conjunto de datos en español llamado EsPython, el cual se usa en varios procesos de validación. Este proceso es necesario para confirmar la primera hipótesis en la sección 1.3 y cubrir el primer objetivo en la sección 1.4.2.

Validar Tranx aumentando datos en EsPython: Similar a la validación anterior, este proceso ayuda a confirmar la primera hipótesis en la sección 1.3 y cubrir el

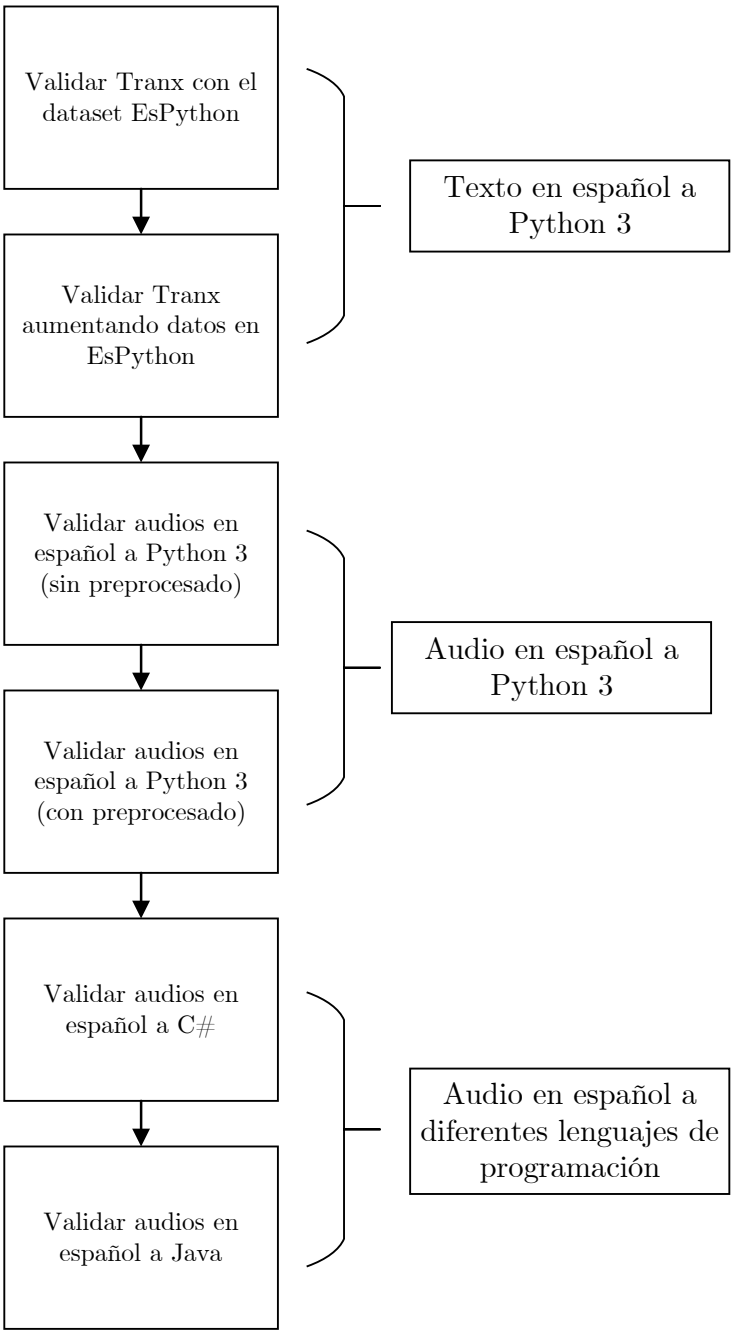

Figura 4.1: Resumen de las validaciones para el modelo propuesto en la investigación.

objetivo uno en la sección 1.4.2, con la diferencia de que se busca mejorar los resultados utilizando técnicas de aumento de datos para entrenar el modelo Tranx.

Validar audios en español a Python 3 (sin preprocesado): En esta validación se pretende realizar pruebas con múltiples audios en español que serán enviados al ASR para obtener un texto, y luego ser enviado directamente al modelo Tranx que haya tenido mejor puntaje BLEU en las validaciones anteriores. Esta validación buscaba validar la hipótesis dos de la sección 1.3 y cubrir parte del segundo objetivo en la sección 1.4.2.

Validar audios en español a Python 3 (con preprocesado): Esta validación es similar a la anterior, con la diferencia de que el resultado del ASR pasa por un algoritmo de preprocesado para mejorar el formato de entrada que utiliza Tranx y buscar mejores resultados en el puntaje BLEU. Esta validación se agregó luego de analizar la versión sin procesado, la cual arrojó múltiples problemas en el código Python generado. Los resultados se describen con más detalle en el capítulo 6.

Validar audios en español a C#: Una de las últimas validaciones es la de verificar la generación de código C# utilizando el modelo completo propuesto en esta investigación, utilizando los tres componentes principales: Audios en español a texto, texto en español a Python, y por último, Python 3 a C#. Este proceso es necesario para validar la tercera hipótesis planteada en la sección 1.3 y cubrir el tercer objetivo en la sección 1.4.2.

Validar audios en español a Java: Al igual que la validación anterior, este proceso utiliza el modelo completo propuesto en esta investigación, utilizando los tres componentes principales, con la diferencia de que que el lenguaje de programación final seleccionado es Java. Este sería el último proceso para terminar de validar la tercera hipótesis planteada en la sección 1.3 y cubrir el tercer objetivo en la sección 1.4.2.

4.2 Validaciones

A continuación se detallan los pasos necesarios utilizados en las diferentes validaciones planteadas para comprobar las hipótesis y los objetivos de la investigación. Las primeras dos validaciones son necesarias para probar el componente de transformación de texto a Python 3, luego las siguientes dos validaciones permiten probar y generar resultados uniendo los primeros dos componentes descritos en la sección 3 para la transformación de audios en español a Python 3, y por último, las siguientes dos validaciones permiten probar y generar resultados uniendo los tres componentes para la transformación de audios en español a Java y C#.

4.2.1 Validar Tranx con el dataset EsPython

Esta es una de las primera validaciones realizadas para comprobar que utilizando un autoencoder como Tranx, se puede generar código Python 3 a partir de instrucciones en español en texto plano, al igual que se ha logrado comprobar con instrucciones en inglés. En la Figura 4.2 se pueden observar los pasos implementados para esta verificación.

Objetivo: Obtener el puntaje BLEU del modelo entrenado con el conjunto de datos EsPython para la transformación de texto en español a Python 3.

Descripción: El modelo Tranx a sido utilizado para generar código en Python 3 a partir de texto con instrucciones en inglés, sin embargo, no se han realizado investigaciones

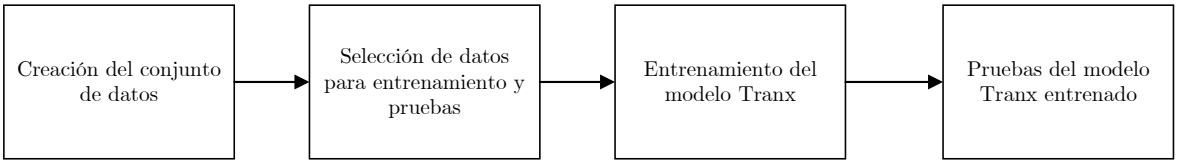

Figura 4.2: Flujo de la validación de Tranx con el dataset EsPython.

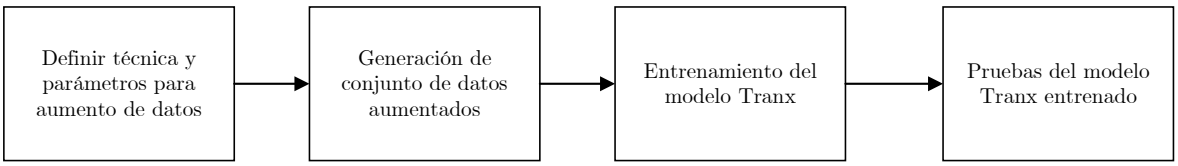

Figura 4.3: Flujo de la validación de Tranx aumentando datos en EsPython.

para validar que este modelo también pueda generar código a partir de instrucciones en español. Es por eso que está validación tiene gran importancia, ya que permite comprobar la primera hipótesis descrita en la sección 1.3. A continuación se describen los pasos realizados para esta validación:

1. Creación del conjunto de datos: Crear de forma manual un conjunto de datos con ejemplos que contengan instrucciones en español y su equivalente en código Python 3.
2. Selección de datos para entrenamiento y pruebas: Seleccionar de forma aleatoria los ejemplos para entrenar el modelo y probarlo. En el capítulo 5 se detalla cómo se realizó este proceso.
3. Entrenamiento del modelo Tranx: Entrenar el modelo Tranx con los hiper parámetros utilizados en trabajo de Yin [63] y ajustarlos para verificar si el resultado mejora.
4. Pruebas del modelo Tranx entrenado: Para el análisis de los resultados se utilizó la métrica BLEU para saber que tan similar fue el resultado al código de referencia.

4.2.2 Validar Tranx aumentando datos en EsPython

Esta validación también utiliza el conjunto de datos EsPython al igual que la validación anterior, con la diferencia de que se aplican varias técnicas de aumento de datos para entrenar el modelo Tranx (ver Figura 4.3). El aumento de datos se aplicó al archivo espython.train.json, el cual es el usado para el entrenamiento (ver capítulo 5). A continuación se describen los pasos usados para completar la validación:

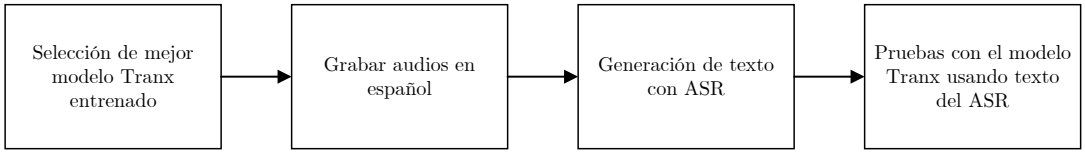

Figura 4.4: Flujo de la validación de audios en español a Python 3 (sin procesado).

Objetivo: Obtener el puntaje BLEU de múltiples modelos Tranx, entrenados cada uno con diferentes conjuntos de datos con técnicas de aumento de datos aplicadas al conjunto de entrenamiento de EsPython.

Descripción: Con el propósito de crear más ejemplos para entrenar con datos en español, se pueden aplicar diferentes técnicas para aumentar los datos de entrenamiento y verificar los resultados de cada uno. A continuación se describen los pasos realizados para esta validación:

1. Definir técnica y parámetros para aumento de datos: Seleccionar las técnicas de aumento de datos que se quieren probar y los parámetros a usar para el entrenamiento del modelo. Es importe mencionar que solo se aumentó la porción de datos de entrenamiento y no la de pruebas para evitar overfitting [27].
2. Generación de conjunto de datos aumentado: Se generó un conjunto de datos de entrenamiento nuevo por cada técnica de aumento de datos aplicada. Estos nuevos conjuntos de datos mantienen las mismas propiedades del conjunto original, con la diferencia de que el texto en español puede tener variantes al aplicar el algoritmo de aumento de datos.
3. Entrenamiento del modelo Tranx: Se generan múltiples versiones del modelo Tranx al utilizar los diferentes conjuntos de datos creados en el paso anterior. Todas estas versiones utilizaron los mismos hiper parámetros de la validación anterior para entrenarse.
4. Pruebas del modelo Tranx entrenado: Para el análisis de los resultados también se utiliza la métrica BLEU para comparar todas las versiones (incluyendo la versión original sin aumento de datos).

4.2.3 Validar audios en español a Python 3 (sin preprocesado)

Una vez realizadas las validaciones anteriores, según los resultados obtenidos se seleccionó la mejor versión del modelo Tranx. Con esto logramos poder unir el ASR y Tranx para procesar audios en español y generar código Python 3. En la Figura 4.4 se muestran los pasos realizados para realizar esta validación que permite probar los resultados obtenidos con base en la unión de estos modelos.

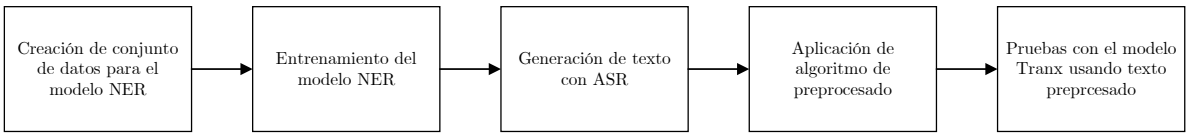

Figura 4.5: Flujo de la validación de audios en español a Python 3 (con preprocesado).

Objetivo: Obtener el puntaje BLEU del modelo Tranx usando audios de pruebas para probar los primeros componentes del método, audio en español a texto, y texto en español a Python 3.

Descripción: Esta validación además de brindar un puntaje BLEU, también permite analizar el código fuente de Python 3 generado a partir de audios con instrucciones en español para detectar los fallos y aciertos del método. Los pasos se describen con más detalle a continuación:

1. Selección del mejor modelo Tranx entrenado: Con base en el BLEU obtenido en las validaciones anteriores al probar las diferentes versiones del modelo Tranx, se selecciona la que tiene mejor puntaje.
2. Grabar audios en español: Se graban los audios en español con base en el texto de los ejemplos que contiene el conjunto de pruebas generado a partir de EsPython.
3. Generación de texto con ASR: Se guardaron todos los audios grabados en una ruta donde se cargaban en memoria todos los archivos para luego ser enviados al ASR, el cual retorna en texto plano la intención del audio.
4. Pruebas con el modelo Tranx usando texto del ASR: Se utiliza como conjunto de datos de pruebas el archivo generado en el paso anterior y se obtiene el puntaje BLEU.

4.2.4 Validar audios en español a Python 3 (con preprocesado)

Similar a la validación anterior, usando la mejor versión del modelo Tranx, se validan los audios en español, con la diferencia de que este método aplica los algoritmos detallados en el capítulo 3. En la Figura 4.5 se muestra el flujo seguido para realizar esta validación.

Objetivo: Obtener el puntaje BLEU del modelo Tranx usando audios de pruebas para probar los primeros componentes del método, audio en español a texto aplicando los algoritmos de preprocesado, y texto en español a Python 3.

Descripción: Al igual que en la validación anterior, además de brindar un puntaje BLEU, también se analiza el código fuente de Python 3 generado a partir de audios con instrucciones en español para detectar los fallos y aciertos del método, lo que permite

Figura 4.6: Flujo de la validación de audios en español a C#.

comparar los resultados de las dos versiones implementadas, con preprocesado y sin preprocesado. Los pasos se detallan a continuación:

1. Creación del conjunto de datos para el modelo NER: Con base en el conjunto de datos EsPython se crea un conjunto de datos para entrenar un modelo NER necesario para el preprocesado.
2. Entrenamiento del modelo NER: Se configuran los parámetros para el entrenamiento del modelo NER y se usa el conjunto de datos creado en el paso anterior.
3. Generación de texto con ASR: Al igual que en la validación anterior, todos los audios grabados se cargan en memoria para luego ser enviados al ASR, para obtener el texto plano de cada audio.
4. Aplicación del algoritmo de preprocesado: Se ejecuta el algoritmo de preproceado descrito en la sección 3 a cada instrucción en español generado por el ASR en el paso anterior. Este archivo se actualiza con el resultado que se obtenga del algoritmo para luego ser enviado al siguiente paso.
5. Pruebas con el modelo Tranx usando texto preprocesado: Se utiliza como conjunto de datos de pruebas el archivo generado en el paso anterior y se obtiene el puntaje BLEU.

4.2.5 Validar audios en español a C#

La validación de audios en español a C# se inició una vez completadas las validaciones anteriores debido a que con el análisis de estas, se determinó la combinación de los componentes de audio a texto y texto a Ptyhon 3 que mejores resultados tuvieron. En esta validación se agregó el último componente que permite convertir Python 3 a otros lenguajes de programación como C#. El flujo que se siguió (ver Figura 4.6) para realizar las pruebas fue el siguiente:

Objetivo: Analizar y verificar el código C# generado por el método completo, utilizando los tres componentes principales: Audio en español a texto, texto en español a Python 3, y Python 3 a C#.

Descripción: Esta es una de las últimas validaciones del trabajo, necesaria para entender que tan bien genera código C# el método planteado en esta investigación a partir de audios en español. Los pasos que se siguieron en esta validación fueron los siguientes:

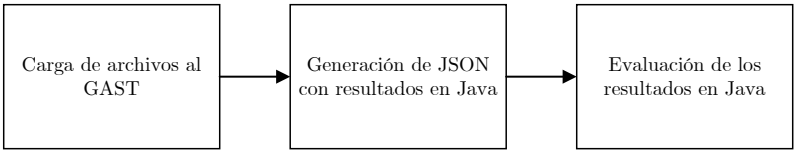

Figura 4.7: Flujo de la validación de audios en español a Java.

1. Selección de los métodos para audio a Python 3: Como primer paso elegimos los métodos que tuvieron mejor resultado en las validaciones anteriores. Se selecciona tanto la transformación de audios en español, como la transformación de texto en español a Python 3.
2. Generación de archivos de Python 3: Se envian los audios grabados a los primeros componentes para obtener a partir de audios en español, código Python 3. Los resultados generados se pasan a un archivo de Python 3 por cada instrucción.
3. Carga de archivos al GAST: Todos los archivos Python 3 se cargaron en memoria para luego ser transformados a un AST genérico (GAST).
4. Generación de JSON con resultados en C#: Todas las instrucciones representadas por el GAST se convirtieron a instrucciones en C# usando el componente Printer que recorre el AST genérico e imprime en texto plano el equivalente a un lenguaje de programación. Este resultado se guardó en un archivo JSON con la instrucción generada en C#, el código Python 3 recibido y el id del audio.
5. Evaluación de los resultados en C#: Se evaluó el código generado comparando la intención original en español y el código Python original de referencia, comparando sintaxis y semántica.

4.2.6 Validar audios en español a Java

Esta validación es similar a la anterior, con la diferencia de que el resultado esperado es en Java. Aquí se usan los mismos audios y métodos seleccionados en la validación anterior. El flujo seguido se muestra en la Figura 4.7. A continuación se detallan los pasos:

Objetivo: Analizar y verificar el código Java generado por el método completo, utilizando los tres componentes principales: Audio en español a texto, texto en español a Python 3, y Python 3 a Java.

Descripción: Al igual que la validación anterior, esta es una de las últimas validaciones del trabajo, en esta ocasión para validar el código Java generado con el método planteado en la investigación usando audios en español. A continuación se mencionan los pasos seguidos:

1. Carga de archivos al GAST: Al igual que en la validación anterior se cargaron en memoria los archivos .py para luego ser transformados a un AST genérico (GAST).
2. Generación de JSON con resultados en Java: En este paso se pasó la estructura GAST que representaba las instrucciones en Python a instrucciones en Java. Al igual que la validación anterior se guardó en un archivo JSON con la instrucción generada en Java, el código Python 3 recibido y el id del audio.
3. Evaluación de los resultados en Java: Por último se evaluó el código Java comparando la intención original en español y el código Python original de referencia, comparando sintaxis y semántica.

Capítulo 5

Implementación del Método

En este capitulo se detalla las tecnologías, herramientas, algoritmos y pasos implementados para obtener los resultados de esta investigación. En sección 5.1, se muestra cómo se creó el conjunto de datos EsPython, la creación de los sub conjuntos utilizados para el entrenamiento, el entrenamiento del modelo Tranx con datos aumentados y sin aumentar, mostrando las diferentes tecnicas de aumento de datos utilizadas y ejemplos de los diferentes modelos Tranx usados para las pruebas.

En la sección 5.2 se documentó cómo se grabaron los audios para las pruebas de generación de código Python 3 a partir de audios en español, luego se muestra las dos versiones implementadas para este componente: La versión sin preprocesado contiene los parámetros usados en la API Azure Speech To Text y el algoritmo generado para transformar todos los audios grabados a texto plano. En la versión con preprocesado se detallan los parámetros para entrenar el modelo NER y el uso de los algoritmos de preprocesado.

Por último, la sección 5.3 muestra los diferentes componentes del GAST utilizados para generar código en Java y C# a partir de código Python 3. Primero se explica el algoritmo para cargar las instrucciones en Python 3, luego se explica como se generó el AST abstracto a partir de las instrucciones Python 3, y para finalizar, cómo se pasó el AST abstracto a Java y C#.

5.1 Texto en español a Python 3

Una de las preguntas que se pretendía resolver en la investigación era si un modelo neuronal sintáctico puede generar código por medio de ASTs, usando español como lenguaje natural. Para contestar esta pregunta, se utilizó el modelo Tranx entrenado con un conjunto de datos creado de forma manual llamado EsPython. Los ejemplos de EsPython son un conjunto de pares con la intención en español y el resultado en Python. Adicionalmente se crearon otros conjuntos de datos aumentados con diferentes técnicas para crear datos sintéticos a partir del conjunto de datos original EsPython, y así comparar los resultados.

Tranx contiene funciones predefinidas para cargar archivos de texto con la definición de las gramáticas, en este trabajo se utilizó la gramática de Python 3, debido a que es uno de los lenguajes soportados por el GAST. El código fuente de este componente está documentado en el repositorio https://github.com/antgonto/seq2ast, se utilizó Python 3 y Anaconda 3 para el desarrollo, el ambiente utilizado fue Ubuntu 20.04 LTS utilizando WSL2, y el IDE donde se configuró las etapas de entrenamiento y pruebas del modelo fue Visual Studio Code.

A continuación se detallan los pasos seguidos (ver Figura 5.1) para la implementación del componente encargado de generar código Python 3 a partir de texto en español:

1. Creación del conjunto de datos: Se creó un conjunto de datos en formato JSON de forma manual. Cada instrucción en español contiene su equivalente en código Python 3.

2. Crear sub conjuntos: Se creó un script para obtener de forma estratificada y aleatoria los ejemplos de EsPython para el entrenamiento, validación y pruebas.

3. Entrenamiento de Tranx con EsPython: Con los sub conjuntos de entrenamiento y validación se entrenó el modelo Tranx con ejemplos de EsPython utilizando un batch size de 128 y 8.

4. Aumentar ejemplos: Se crearon scripts para aplicar diferentes técnicas de aumento de datos en los ejemplos de entrenamiento de EsPython.

5. Entrenar Tranx con ejemplos aumentados: Con los nuevos conjuntos de datos de entrenamiento se entrenaron nuevos modelos Tranx utilizando batches de 128 y 8.

5.1.1 Creación del conjunto de datos

EsPython es un conjunto de datos con 301 ejemplos utilizados para entrenar el modelo Tranx. Los ejemplos se realizaron de forma manual utilizando como base la sintaxis de LIE++ con variaciones para hacer más flexible las instrucciones. Cada instrucción en español tiene su equivalente en el lenguaje de programación Python 3.

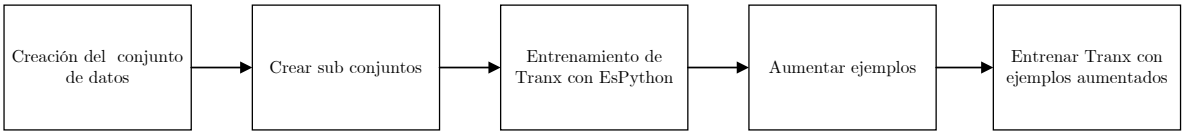

Figura 5.1: Flujo para la implementación del componente que transforma texto en español a Python 3.

El conjunto de datos es un archivo de tipo JSON (ver Figura 5.2), y dentro de este archivo se encuentra una lista de objetos con las siguientes propiedades:

Id: Campo de tipo numérico, es un identificador único del ejemplo, usado para mapear los resultados en los diferentes componentes del modelo para generar código a partir de audios en Español.

Lie_syntax: Campo de tipo booleano. Si el valor es $True$, el campo *rewritten_intent* cumple con la sintaxis del lenguaje LIE++, si es $False$, el campo *rewritten_intent* no necesariamente cumple con la sintaxis de LIE++.

rewritten_intent: Campo de tipo string que contiene la instrucción en el lenguaje natural español describiendo la intención.

snippet: Campo de tipo de string que contiene el equivalente de la instrucción en español en el lenguaje de programación Python 3. Algunos ejemplos contienen una función llamada *indtoken* junto con un salto de línea y una tabulación. Esto se utiliza para poder realizar la conversión de texto a un AST de algunos ejemplos que por sintaxis del lenguaje necesitan tener una instrucción dentro de un bloque para ser válido. Por ejemplo: $if\ b\ != 0\ :\backslash n\ indtoken()$.

```
{
    "Id": 3,
    "lie_syntax": false,
    "rewritten_intent": "Si la variable `a` es menor que 0 entonces",
    "snippet": "if a < 0:\n indtoken()"
},
{
    "Id": 4,
    "lie_syntax": true,
    "rewritten_intent": "Si `a` es menor que 0 entonces",
    "snippet": "if a < 0:\n indtoken()"
},
{
    "Id": 5,
    "lie_syntax": true,
    "rewritten_intent": "devolver `a`",
    "snippet": "return a"
}
```

Figura 5.2: Ejemplos del conjunto de datos EsPython utilizado para entrenar el modelo neuronal sintáctico.

5.1.2 Creación del sub conjunto de datos para el entrenamiento, validación y pruebas

Para entrenar y probar el modelo Tranx se crearon los conjuntos de datos train, test y dev, obteniendo los ejemplos de EsPython de forma aleatoria y estratificado por la propiedad *lie_syntax*. A continuación se describen los archivos JSON creados:

espython.train.json : Contiene 212 ejemplos para realizar el entrenamiento del modelo, el cual también es usado como base para el aumento de datos.

espython.test.json : Contiene 58 ejemplos para realizar las pruebas del modelo. Este archivo también se utiliza como base para las pruebas para generar código a partir de audios.

espython.dev.json : Contiene 30 ejemplos para ajustar los hiper parámetros en el proceso de entrenamiento.

La creación de estos subconjuntos se realizó con el algoritmo 5. Este algoritmo primero inicia una lista vacía donde se guardan los ejemplos que cumplen con la sintaxis LIE++ (línea 1), y otra lista vacía donde se guardaran los ejemplos que no cumplen con la sintaxis LIE++ (línea 2). Luego recorre todos los ejemplos del conjunto de datos EsPython (línea 3), y si es un ejemplo LIE++ se agrega a la lista de ejemplos LIE (línea 4 y 5), de lo contrario se agrega a la lista de ejemplos que no son LIE (línea 7 y 8). Luego define el número de ejemplos con sintaxis LIE++ y sin sintaxis LIE++ que va a tener el sub conjunto train y dev (línea 11 a 14), el sub conjunto train se le asigna setenta por ciento de los ejemplos, mientras que al sub conjunto dev se le asigna diez por ciento.

De manera aleatoria se van seleccionando los ejemplos con sintaxis LIE++ para el sub conjunto train, la función *obtenerEjemplos* retorna el número de ejemplos especificado en el primer parámetro, y los elimina de la lista original que se define como segundo parámetro (línea 15), lo mismo se realiza para el sub conjunto dev (línea 16), y la lista restante se asigna al sub conjunto test (línea 17).

Para asignar los ejemplos sin sintaxis LIE++ se utiliza nuevamente la función *obtenerEjemplos*, y los ejemplos restantes se asignan al subconjunto test (línea 18 a 20). Se unen las listas de ejemplos definidas para cada sub conjunto de datos (línea 21 a 23), y por último se ordenan de manera aleatoria los ejemplos en cada sub conjunto para ser retornados (línea 24 a 27).

Algoritmo 5: Algoritmo para crear los sub conjuntos de datos

Datos: $esPython$
Resultado: $train$, val, $test$

$ejemplosLIE = []$
$ejemplosNoLIE = []$
para $ejemplo$ *en* $esPython$ **hacer**
    si $ejemplo['lie\_syntax'] = True$ **entonces**
        $ejemplosLIE.agregar(ejemplo)$
    en otro caso
        $ejemplosNoLIE.agregar(ejemplo)$
$numeroEjemplosLIETrain \leftarrow ceil(0.7 * len(ejemplosLIE))$
$numeroEjemplosNoLIETrain \leftarrow ceil(0.7 * len(ejemplosNoLIE))$
$numeroEjemplosLIEDev \leftarrow ceil(0.1 * len(ejemplosLIE))$
$numeroEjemplosNoLIEDev \leftarrow ceil(0.1 * len(ejemplosNoLIE))$
$trainLIE \leftarrow obtenerEjemplos(numeroEjemplosLIETrain, ejemplosLIE)$
$devLIE \leftarrow obtenerEjemplos(numeroEjemplosLIEDev, ejemplosNoLIE)$
$testLIE \leftarrow ejemplosLIE$
$trainNoLIE \leftarrow$
  $obtenerEjemplos(numeroEjemplosNoLIETrain, ejemplosLIE)$
$devNoLIE \leftarrow$
  $obtenerEjemplos(numeroEjemplosNoLIEDev, ejemplosNoLIE)$
$testNoLIE \leftarrow ejemplosNoLIE$
$train \leftarrow unirListas(trainLIE, trainNoLIE)$
$dev \leftarrow unirListas(devLIE, devNoLIE)$
$test \leftarrow unirListas(testLIE, testNoLIE)$
$ordenarAleatoriamente(train)$
$ordenarAleatoriamente(dev)$
$ordenarAleatoriamente(test)$
devolver $train, dev, test$

5.1.3 Entrenamiento de Tranx usando EsPython

Para entrenar el modelo Tranx usando el conjunto de datos EsPython se utilizó un vector latente de tamaño 256, el tamaño de los embeddings fue de 128, un dropout de 0.3, learning rate de 0.001, learning rate decay de 0.5, máximo de epochs 80. La arquitectura del modelo se muestra en la Figura 5.3, el cual recibe como entrada un texto en español y genera una serie de acciones ASDL que pueden ser transformadas a código Python por medio de un AST.

Se entrenaron los modelos con dos tamaños de batch size, 128 y 8. Estos parámetros fueron seleccionados para todos los conjuntos de datos de entrenamiento utilizados (EsPython sin aumentar y con aumento de datos). El modelo Tranx entrenado con el conjunto de datos

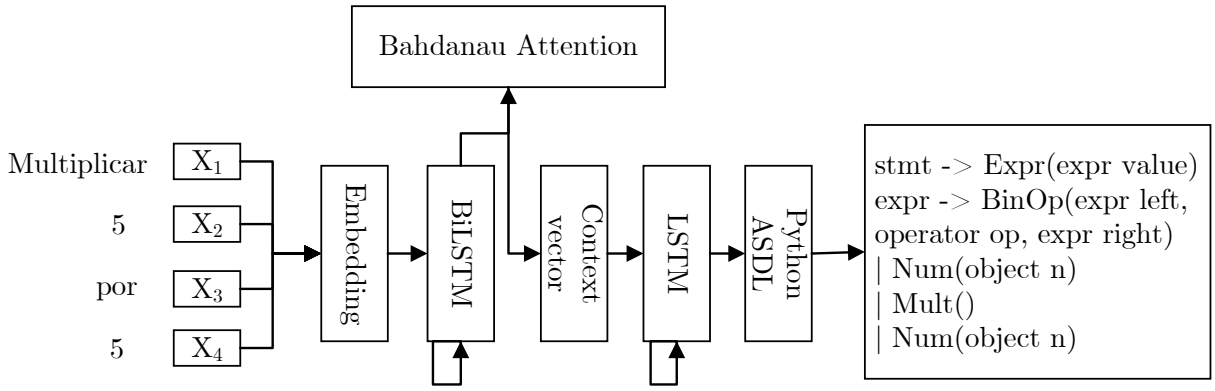

Figura 5.3: Arquitectura del modelo Tranx.

original tuvo un resultado de 0.65 usando batch size de 128, y un resultado de 0.85 usando un batch size de 8 en el proceso de entrenamiento como se puede ver en la Figura 5.4.

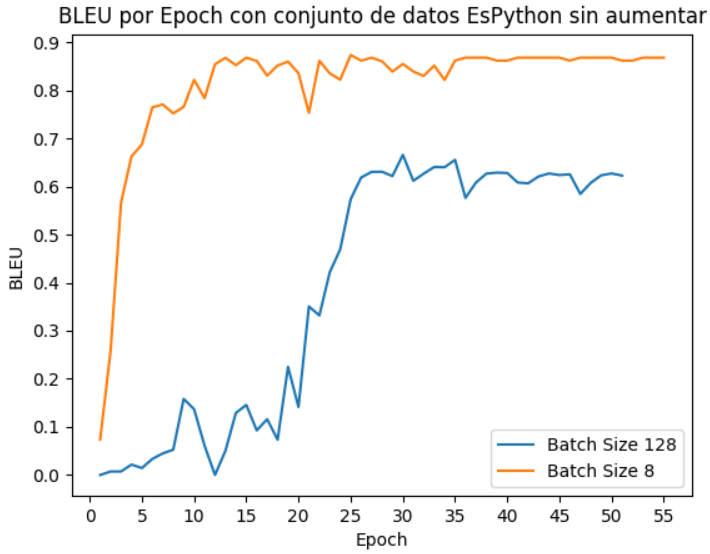

Figura 5.4: Comparación del puntaje BLEU usando un batch size de 128 y de 8 en el modelo Tranx, entrenado con el conjunto de datos EsPython .

5.1.4 Aumento de datos

Para realizar el aumento de datos de entrenamiento de EsPython se utilizó la librería de Python *NlpAug* [37], la cual permite aumentar datos de ejemplos textuales y acústicos. También se utilizó un Notebook en Google Colab para ejecutar las diferentes técnicas utilizas en este trabajo.

El algoritmo 6 describe los pasos realizados para aumentar los ejemplos de entrenamiento

de EsPython. Primero se define la técnica de aumento de datos y sus parámetros, en la línea 2 se recorren todos los ejemplos de entrenamiento de EsPython y en la línea 3 se crean dos variantes por cada instrucción en español. En la línea 4 se recorren las variantes creadas con la técnica de aumento de datos, y en la línea 5 se crea una copia del ejemplo EsPython, cambiando únicamente el campo *lie_syntax* a $False$ y el valor de *rewritten_intent* es actualizado a la variante generada. En la línea 6 se agrega el nuevo ejemplo al conjunto de datos de entrenamiento, luego en la línea 9 se ordena el nuevo conjunto de datos aumentado de forma aleatoria, y por último, en la línea 10 se retorna el conjunto de datos aumentado.

Algoritmo 6: Algoritmo para aumentar datos

Datos: *conjuntoDeDatosDeEntrenamiento*
Resultado: *conjuntoDeDatosAumentado*

```
1 tecnica = TecnicaDeAumentoDeDatos(parametros);
2 para fila en conjuntoDeDatosDeEntrenamiento hacer
3     nuevosEjemplos = tecnica.aumentar(fila['rewritten_intent'], n = 2);
4     para ejemplo en nuevosEjemplos hacer
5         nuevaFila = {'Id' : fila['Id'],' lie_syntax' : False,' rewritten_intent' :
           ejemplo,' snippet' : fila['snippet']};
6         conjuntoDeDatosDeEntrenamiento.agregar(nuevaFila)
7 ordenarAleatoriamente(conjuntoDeDatosDeEntrenamiento);
8 devolver conjuntoDeDatosDeEntrenamiento
```

Este algoritmo se usó para las diferentes técnicas de aumento de datos utilizadas en este trabajo con sus respectivos parámetros. A continuación se describe la implementación de cada técnica de aumento de datos, como lucen los ejemplos sintéticos y como varió el entrenamiento del modelo Tranx en cada técnica.

Synonim Augmentation: Remplaza palabras similares con base a un modelo preentrenado. En esta investigación se utilizó el modelo Wordnet en español con los siguientes parámetros:

aug_src: wordnet

aug_max: 2

lang: spa

En la Figura 5.5 se puede observar algunos de los ejemplos aumentados que generó el uso de esta técnica.

En la Figura 5.6 se muestra el BLEU obtenido en cada epoch al entrenar Tranx con el conjunto de datos aumentado con Synonim Augmentation, comparando el batch size de 128 y 8. Al finalizar el entrenamiento, el BLEU obtenido con el batch size de 128 fue de 0.8312 y con el de 8 fue de 0.8785.

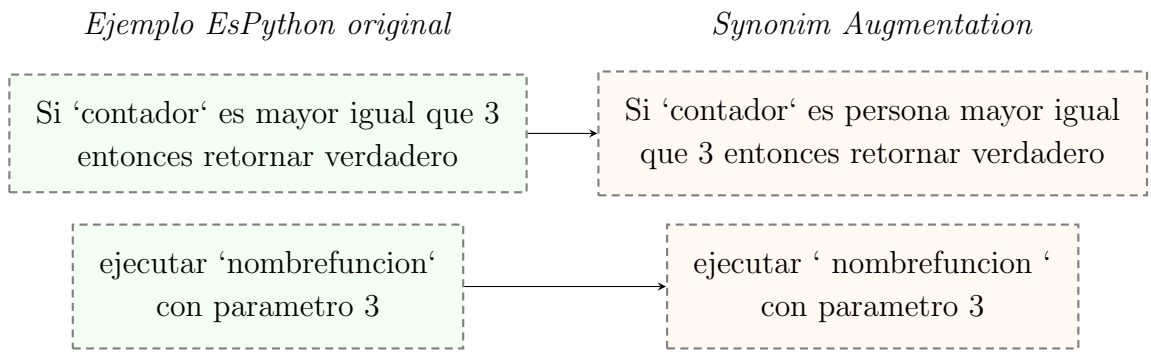

Figura 5.5: Instrucciones en español del conjunto de datos EsPython aumentadas con la técnica de Synonim Augmentation.

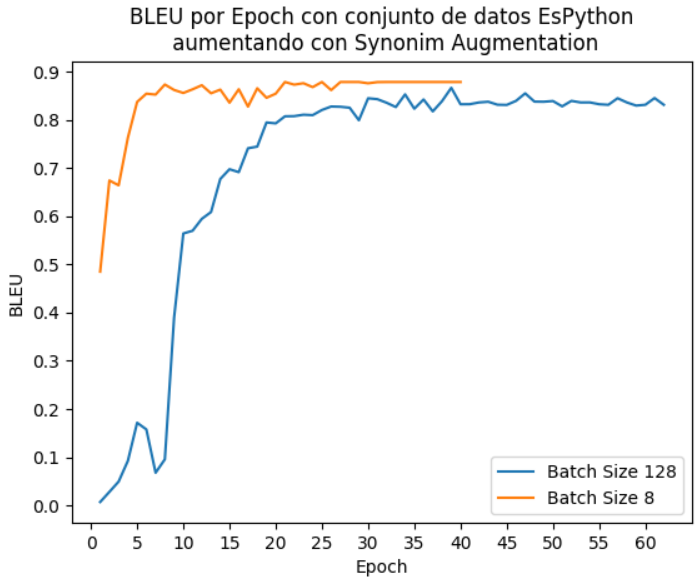

Figura 5.6: Comparación del puntaje BLEU usando un batch size de 128 y de 8 en el modelo Tranx, umentando EsPython con Synonim Augmentation .

Contextual Word Embedding: Agrega palabras en las oraciones según el contexto por medio de modelos de lenguaje preentrenados. Los parámetros utilizados fueron:

model_path: bert-base-multilingual-uncased

aug_p: 0.1

El resultado generado por este algoritmo se puede observar en la Figura 5.7, el cual muestra cambios y agregación de palabras.

En la Figura 5.8 se muestra el BLEU obtenido en cada epoch al entrenar Tranx con el conjunto de datos aumentado con Contextual Word Augmentation, comparando el batch size de 128 y 8. Al finalizar el entrenamiento, el BLEU obtenido con el batch size de 128 fue de 0.83842 y con el de 8 fue de 0.8893.

Keyboard augmentation: Substituye caracteres en las palabras simulando errores de

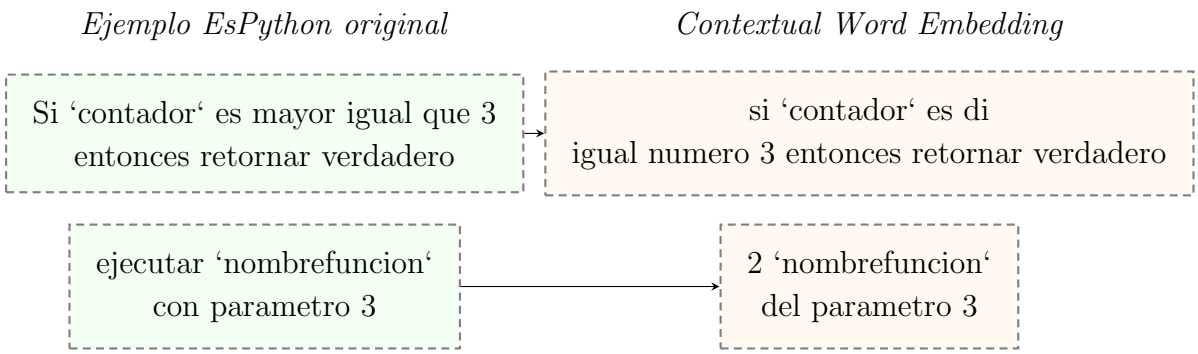

Figura 5.7: Instrucciones en español del conjunto de datos EsPython aumentadas con la técnica de Contextual Word Embedding.

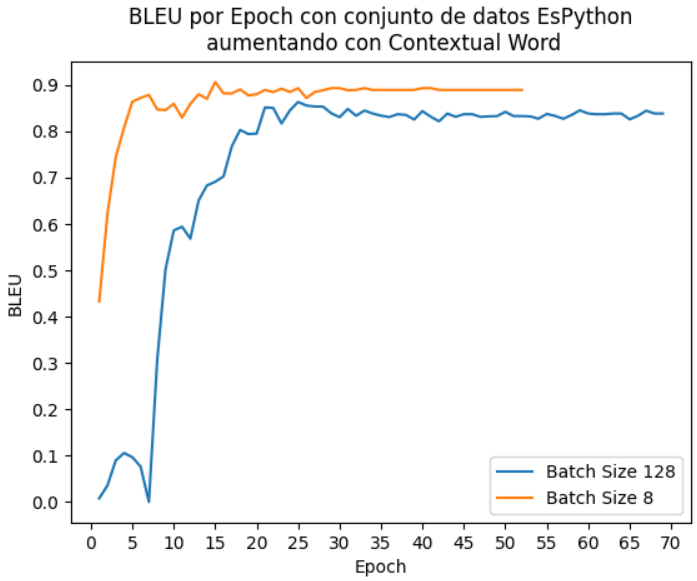

Figura 5.8: Comparación del puntaje BLEU usando un batch size de 128 y de 8 en el modelo Tranx, umentando EsPython con Contextual Word Augmentation.

escritura según la distancia de las teclas del teclado y el caracter a remplazar. Los parámetros utilizados en esta técnica fueron los siguientes:

Lang: es

include_special_char: False

aug_word_max: 2

aug_char_max: 1

Esta técnica cambió algunas letras por números y letras que están cerca al carácter original en el teclado como se puede ver en la Figura 5.9.

En la Figura 5.10 se muestra el BLEU obtenido en cada epoch al entrenar Tranx con el conjunto de datos aumentado con Keyboard Augmentation, comparando el

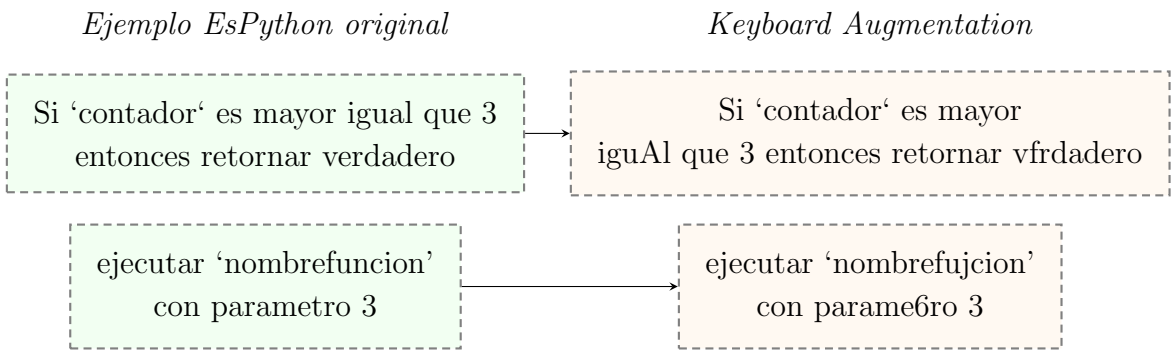

Figura 5.9: Instrucciones en español del conjunto de datos EsPython aumentadas con la técnica de Keyboard Augmentation.

batch size de 128 y 8. Al finalizar el entrenamiento, el BLEU obtenido con el batch size de 128 fue de 0.8618 y con el de 8 fue de 0.8561.

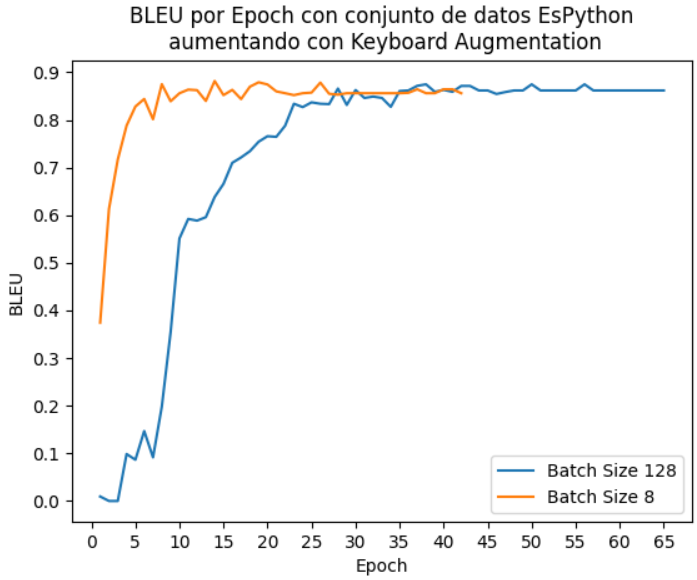

Figura 5.10: Comparación del puntaje BLEU usando un batch size de 128 y de 8 en el modelo Tranx, umentando EsPython con Keyboard Augmentation.

Random char augmentation: Genera errores con caracteres usando valores aleatorios. Esta técnica permite generar los errores ya sea insertando, remplazando, eliminando o intercambiando caracteres. En esta investigación se utilizaron todas las opciones con los siguientes parámetros:

aug_word_max: 2

aug_char_max: 1

stopwords: Lista de nombres de variables a las que no queríamos afectar sus valores.

En la Figura 5.11 se muestra el resultado con la variante de delete (eliminación de

caracteres). En los ejemplos aumentados se eliminaron letras en las palabras de forma aleatoria.

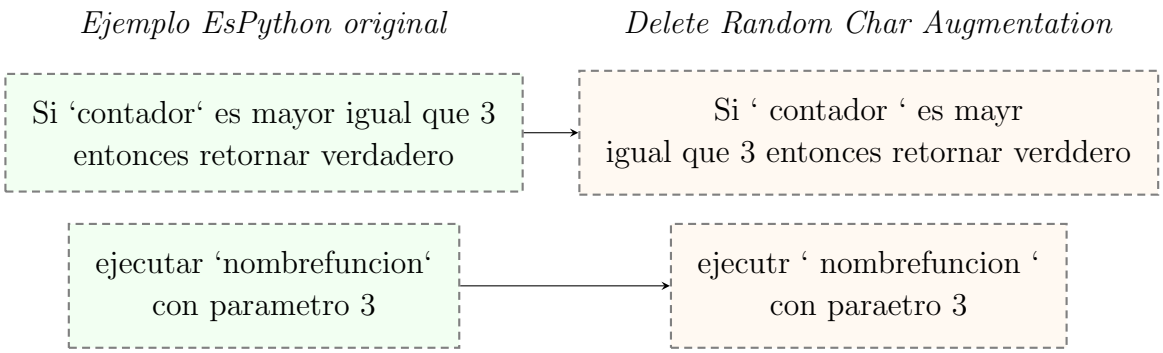

Figura 5.11: Instrucciones en español del conjunto de datos EsPython aumentadas con la técnica de delete Random char Augmentation.

En la Figura 5.12 se muestra el BLEU obtenido en cada epoch al entrenar Tranx con el conjunto de datos aumentado con Rando Char Delete, comparando el batch size de 128 y 8. El BLEU obtenido con el batch size de 128 fue de 0.8374, mientras que con el de 8, fue de 0.8491.

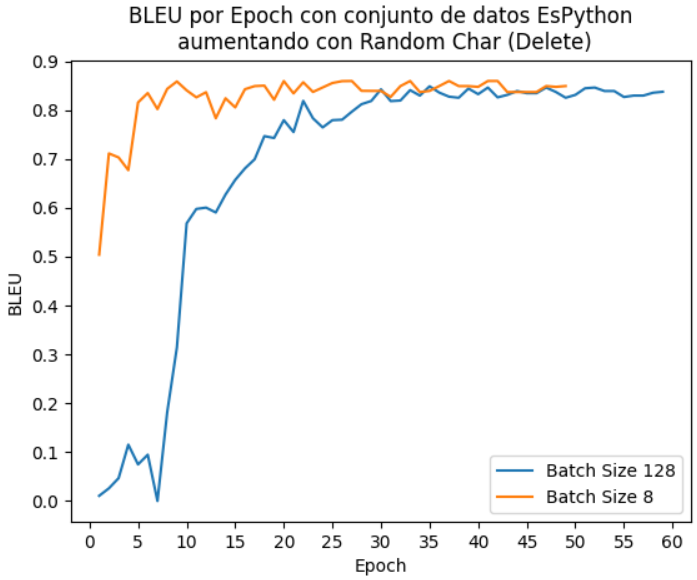

Figura 5.12: Comparación del puntaje BLEU usando un batch size de 128 y de 8 en el modelo Tranx, aumentando EsPython con Random Char (Delete) Augmentation.

La variante substitute (substitución de caracteres) cambió letras en ciertas palabras como podemos ver en la Figura 5.13.

En la Figura 5.14 se muestra el BLEU obtenido en cada epoch al entrenar Tranx con el conjunto de datos aumentado con Random Char Substitute, comparando el batch size de 128 que obtuvo un puntaje BLEU de 0.5982, y 8 que obtuvo un puntaje de 0.8710.

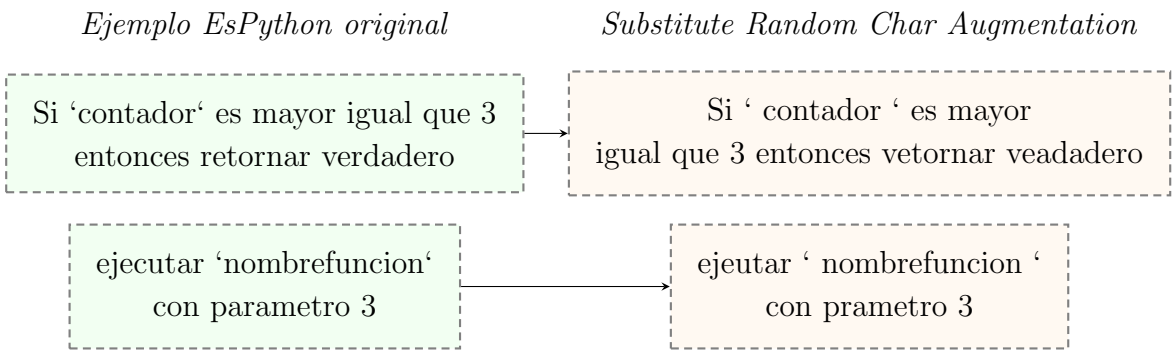

Figura 5.13: Instrucciones en español del conjunto de datos EsPython aumentadas con la técnica de substitute Random char Augmentation.

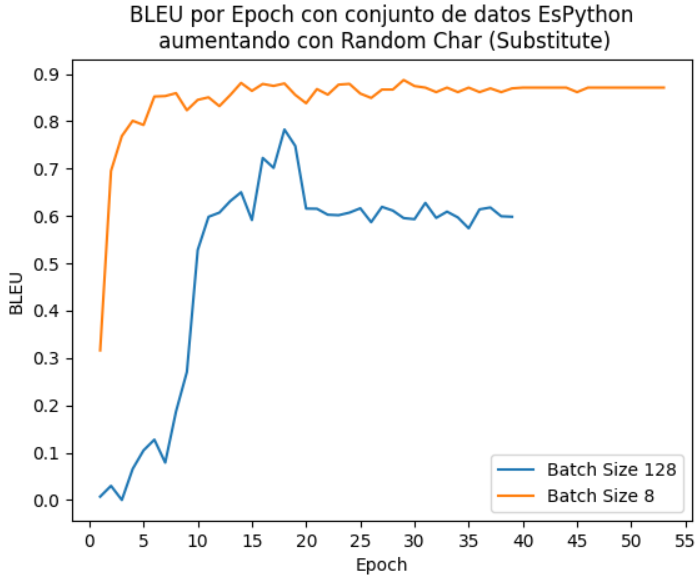

Figura 5.14: Comparación del puntaje BLEU usando un batch size de 128 y de 8 en el modelo Tranx, aumentando EsPython con Random Char (Substitute) Augmentation.

Al utilizar la variante swap (cambio de caracteres) se cambiaron de forma aleatoria letras en algunas palabras como se muestra en la Figura 5.15.

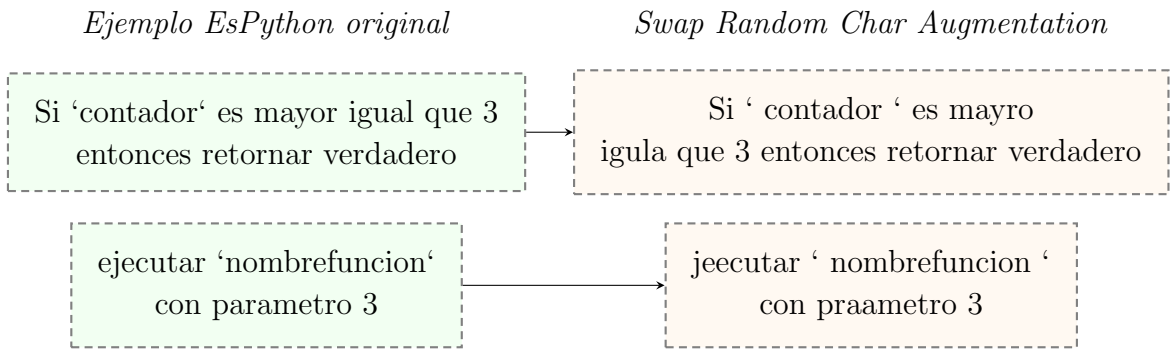

Figura 5.15: Instrucciones en español del conjunto de datos EsPython aumentadas con la técnica de swap Random char Augmentation.

En la Figura 5.16 se muestra el BLEU obtenido en cada epoch al entrenar Tranx con el conjunto de datos aumentado con Random Char Swap, comparando el batch size de 128 y 8. El BLEU obtenido con el batch de 128 fue de 0.8768, y el de tamaño de 8 fue de 0.8403.

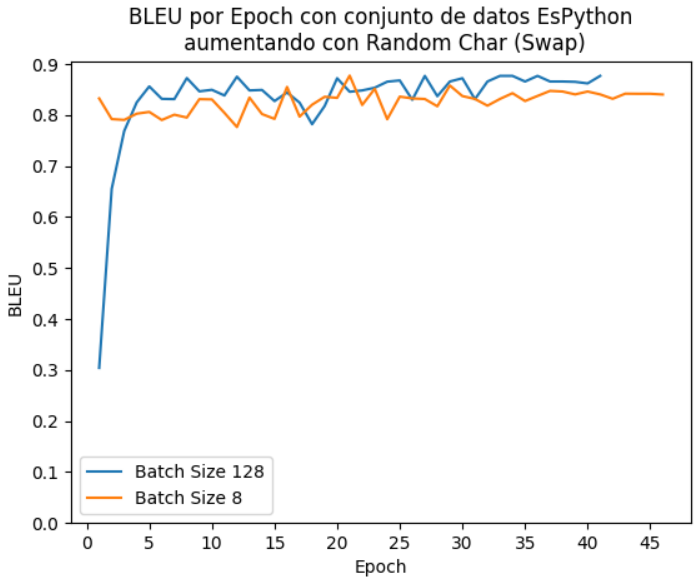

Figura 5.16: Comparación del puntaje BLEU usando un batch size de 128 y de 8 en el modelo Tranx, aumentando EsPython con Random Char (Swap) Augmentation.

En la Figura 5.17 se puede observar el resultado de la variante de insert (agregación de caracteres). De forma aleatoria se agregaron caracteres en las palabras.

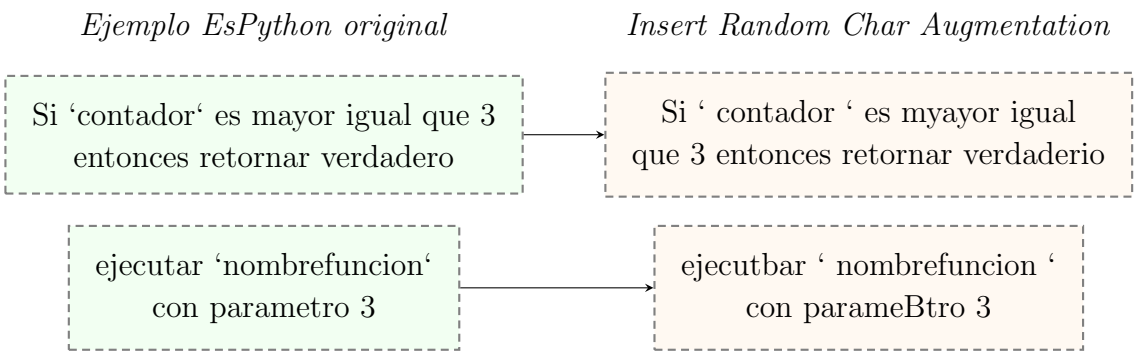

Figura 5.17: Instrucciones en español del conjunto de datos EsPython aumentadas con la técnica de insert Random char Augmentation.

En la Figura 5.18 se muestra el BLEU obtenido en cada epoch al entrenar Tranx con el conjunto de datos aumentado con Random Char Insert, comparando el batch size de 128 y 8. El puntaje BLEU obtenido con el batch de 128 fue 0.6055, mientras que con el de 8 fue de 0.8715.

Split word augmentation: Divide de 1 a N palabras de forma aleatoria según la cantidad dada por parámetro. Parámetros usados:

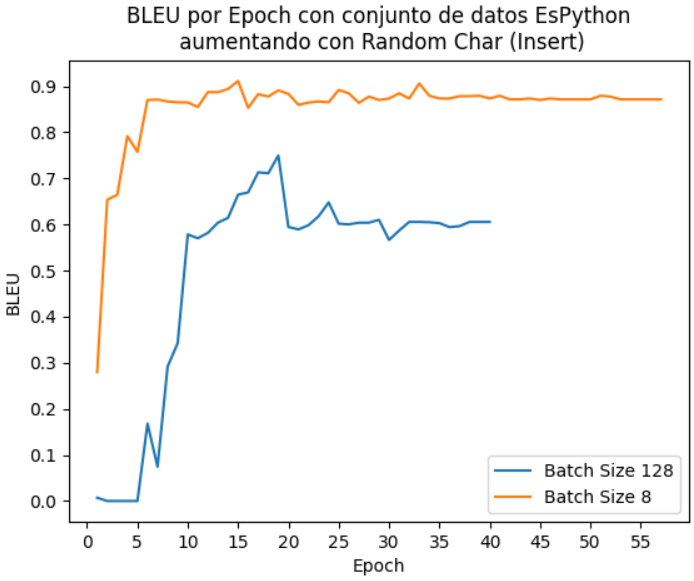

Figura 5.18: Comparación del puntaje BLEU usando un batch size de 128 y de 8 en el modelo Tranx, aumentando EsPython con Random Char (Insert) Augmentation.

aug_max: 2

stopwords: Lista de nombres de variables a las que no queríamos afectar sus valores.

Algunos de las los ejemplos aumentados con la técnica Split word augmentation se puede observar en la Figura 5.19. En estos ejemplos se puede observar que algunas de las palabras originales fueron cortadas con un espacio.

En la Figura 5.20 se muestra el BLEU obtenido en cada epoch al entrenar Tranx con el conjunto de datos aumentado con Split Word Augmentation, comparando el batch size de 128 y 8. El BLEU obtenido con el batch de 128 fue de 0.8283, y el de 8 fue de 0.7977.

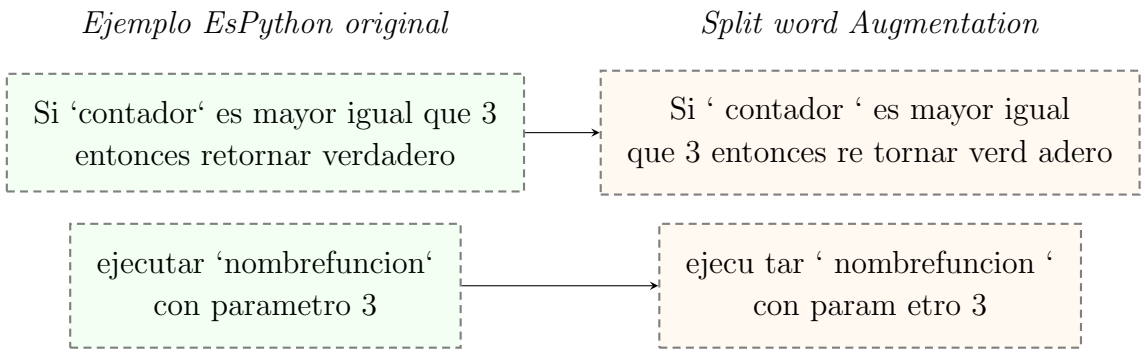

Figura 5.19: Instrucciones en español del conjunto de datos EsPython aumentadas con la técnica de Split Word Augmentation.

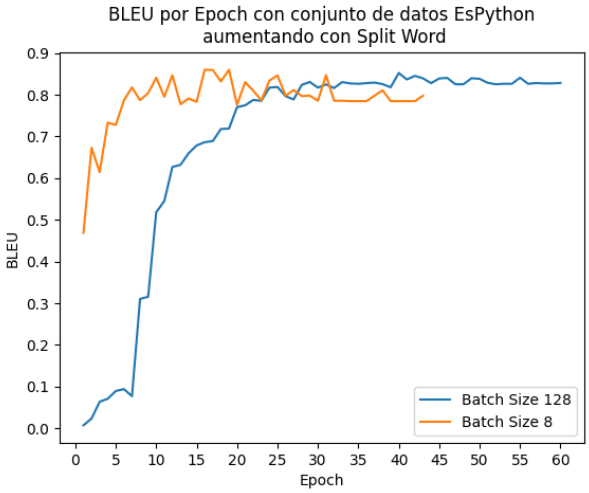

Figura 5.20: Comparación del puntaje BLEU usando un batch size de 128 y de 8 en el modelo Tranx, umentando EsPython con Split Word Augmentation.

5.2 Audio en español a Python 3

Los audios utilizados para las pruebas del modelo fueron grabados con Audacity(R) [2]. Esta aplicación de código abierto permite grabar audios a través de un micrófono y exportarlos a diferentes formatos de audios como wav, el cual es el formato utilizado en el modelo Speech To Text. Se utilizó Java para procesar los audios y llamar la API del modelo Speech To Text sin preprocesado, mientras que para el entrenamiento del modelo NER y la aplicación del algoritmo de preprocesado se hizo en Python. El código fuente y los audios grabados se encuentran en `https://github.com/antgonto/audio2source`.

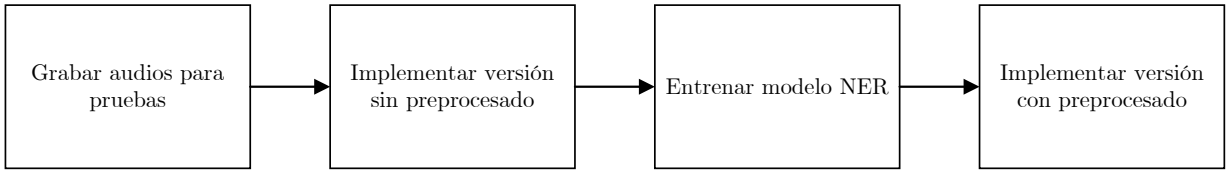

Figura 5.21: Flujo para la implementación del componente que transforma audios en español a Python 3.

Los pasos que se siguieron para la implementación de este componente se pueden ver en la Figura 5.21. A continuación se describe un resumen de estos pasos:

1. Grabar audios para pruebas: Utilizando los ejemplos de pruebas de EsPython, se grabaron manualmente audios en español en formato .wav.

2. Implementar versión sin procesado: Usa el servicio de Azure Speech To Text para transformar los audios en español a texto plano.

3. Entrenar modelo NER: Se usa el conjunto de datos EsPython para entrenamiento con Random Char Insert y los datos de validación se entrena el modelo NER.

4. Implementar versión con procesado: Utiliza el modelo NER entrenado, se aplica el algoritmo de procesado para transformar audios en español a Python 3 con un mejor formato para la entrada de Tranx.

5.2.1 Grabación de audios para pruebas

Las pruebas se realizaron utilizando el mismo conjunto de datos de que se usó en Tranx, llamado espython.test.json. Se grabó la instrucción en español que cada ejemplo tiene en la propiedad rewritten_intent. En total se grabaron 58 audios. Estos audios se usaron como entrada para el modelo sin pre procesado y con pre procesado. Se utilizó la métrica BLEU para medir los resultados de cada modelo.

La mitad de los audios se grabaron con las siguientes características:

- Sistema operativo: Windows 10
- Version Audacity: 3.3.3
- Micrófono: Logitech G331

La otra mitad fueron grabados con estas características:

- Sistema operativo: Windows 11 22H2
- Version Audacity: 3.2.1
- Micrófono: Corsair HS70 Bluetooth

Los archivos wav fueron exportados con la codificación Signed 16-bit PCM.

5.2.2 Versión sin preprocesado

Actualmente existen diferentes soluciones para la transformación de audios a texto. Algunas soluciones son de código abierto y pueden ser usados en un dispositivo sin internet, mientras que otras soluciones se ejecutan en servidores en la nube por medio de APIs de alto nivel.

Diferentes estudios han comparado estos modelos con la métrica WER (Word error rate) con diferentes conjuntos de datos en diferentes dominios. Sin embargo, en este trabajo el objetivo es poder utilizar un modelo Speach To Text como un primer paso, el cual transforma un audio en español a un texto, para luego convertirlo a Python con el modelo entrenado con el conjunto de datos EsPython, para luego usar la métrica BLEU y comparar los resultados con el código de referencia.

En este trabajo se utilizó Microsoft Azure Speech to Text para transformar los audios a texto en español ya que contábamos con acceso a este servicio, sin embargo, se diseñó una arquitectura que permitiera utilizar diferentes modelos existentes sin realizar cambios en los otros componentes del modelo (ver Figura 5.22). Esta arquitectura utiliza el patrón Dependency Injection, la cual nos permite implementar los métodos necesarios por cualquier modelo Speach To Text existente, permitiendo una mayor facilidad para experimentos futuros en caso de realizar pruebas con un modelo que no sea el de Azure Speach To Text. Independientemente de la solución usada para realizar la tarea, el output debe mantener el mismo formato respetando las propiedades necesarias que utilizan los otros componentes. Para asegurar esto, se creó una interfaz común llamada ISpeechRecognition. Esta interfaz contiene 2 métodos para implementar:

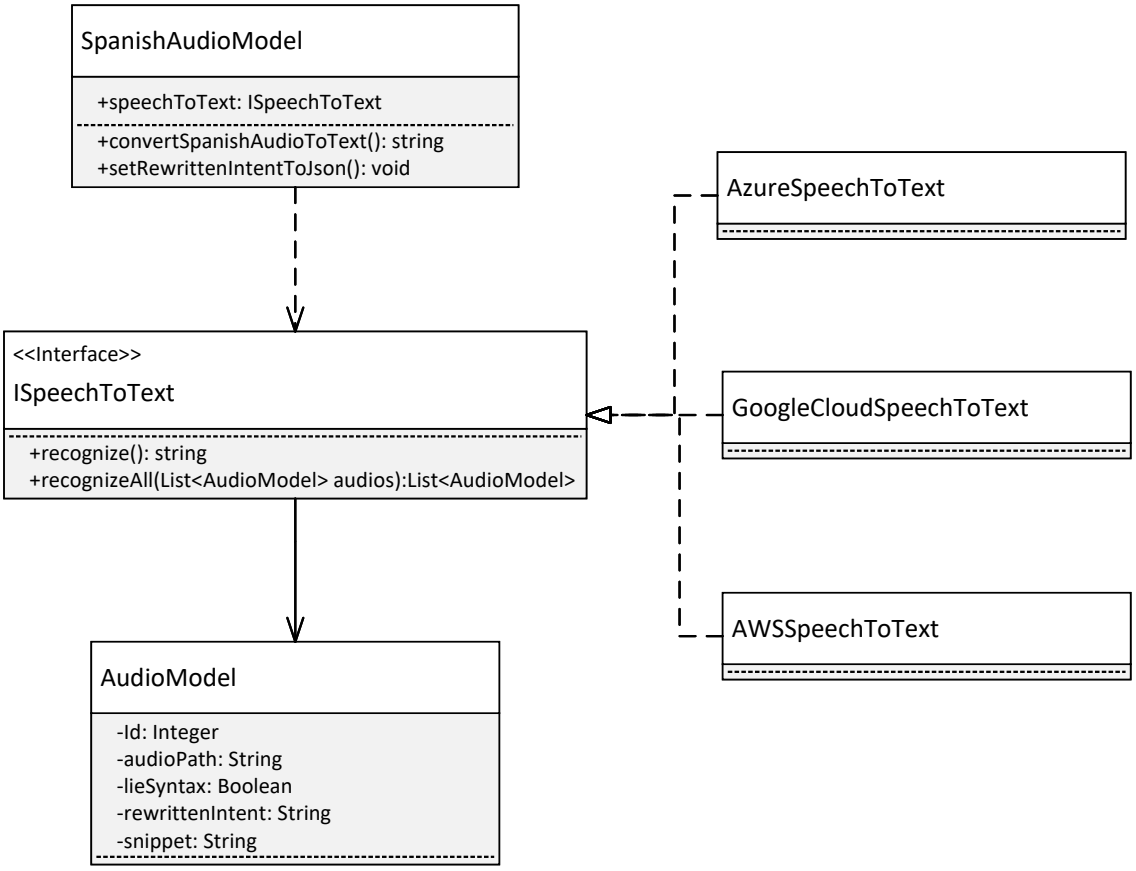

Figura 5.22: Arquitectura usada para el modelo que transforma audios en Español a texto, utilizando el patrón Dependency Injection.

- recognize: Este método permite obtener un audio en tiempo de ejecución a partir del micrófono del dispositivo donde se esté ejecutando. El audio es enviado a un modelo de SpeechToText y retorna un string con el texto en español que representa lo dicho por el usuario en el micrófono. El propósito de este método es realizar demos del modelo, para las pruebas se utilizó *recognizeAll*.

- recognizeAll: Este método recibe como parámetro una lista de estructuras *AudioModel*, la cual contiene los campos necesarios para los demás componentes del modelo.

Cada *AudioModel* tiene la ruta donde se encuentra el audio previamente grabado (audioPath), el cual es utilizada para cargar el audio en la aplicación y enviarla al modelo de SpeechToText, el cual se encargará de obtener el texto en español y asignarlo a la propiedad *rewrittenIntent*, para luego ser utilizada en el componente que transforma texto a código Python. Luego de procesar toda la lista, este retornará la lista de estructuras *AudioModel* con su valor *rewrittenIntent* actualizado. Este método es el utilizado para realizar las pruebas del modelo.

Se implementó la interfaz *ISpeechRecognition* con sus métodos correspondientes usando las siguientes configuraciones de Azure:

- Service Region: eastus
- Language: es-CR

El lenguaje utilizado fue español de Costa Rica ya que los audios grabados fueron grabados por personas con acento de Costa Rica.

Para el método *recognize* se utilizó la función *FromDefaultMicrophoneInput*, la cual crea un objeto *AudioConfig* para escuchar voz del micrófono del dispositivo corriendo la aplicación. Con la función *RecognizeOnceAsync* se inicia la detección de voz y retorna hasta que la oración finalice. En final de la oración se determina por algún silencio al final.

En el método *recognizeAll* se usa la función *FromWavFileInput*, esta recibe como parámetro la ruta del audio en formato Wav y retorna un objeto *AudioConfig* para instanciar *SpeechRecognizer* y obtener el texto a partir de audios wav con reconocimiento continuo, la cual permite tener el control de cuando se desea detener el reconocimiento de voz. Esto permite tener flexibilidad en los silencios que hay en los audios y así poder tener mayor tiempo de espera en las pausas de silencio cuando los audios son largos [39]. El texto obtenido se asigna a la propiedad *rewrittenIntent* de la estructura *AudioModel*. Una vez que se reconocen todos los audios se retorna la lista de *AudioModel* actualizada. Como se puede ver en la Figura 5.23, el resultado del modelo Speech To Text al transformar un audio a texto puede variar con respecto al texto original del ejemplo EsPython.

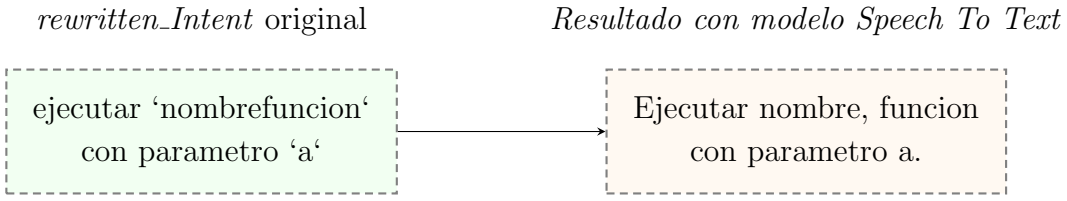

Figura 5.23: Resultado del modelo Speech To Text después de transformar un audio en español a texto.

5.2.3 Versión con preprocesado

Para la configuración base en spaCy para el entrenamiento se utilizó como lenguaje Español, CPU como hardware, un batch size de 1000 y el pipeline NER.

Para el modelo NER de spaCy se generó un conjunto de datos de entrenamiento a partir del conjunto de datos aumentado que mejor resultados obtuvo en el modelo Tranx. Para las pruebas se generó un conjunto de datos a partir de el archivo espython.dev.json. La estructura de los conjuntos de datos de entrenamiento y pruebas para el modelo NER se puede observar en el JSON de ejemplo 5.24. En la línea 3 se define la instrucción en español y luego en la línea 5 se define un objeto con la propiedad entities, la cual tiene un arreglo que define donde están los nombres de variables, funciones o clases. En el primero ejemplo, solo se tiene un nombre de variable llamada "var", por lo que se asigna el índice 10 como el inicio donde se encuentra la variable, y 13 representa el final del índice. El parámetro 'TOKEN' le indica a el modelo NER de spaCy que es una entidad que se quiere aprender a detectar en los textos. En la línea 12, 13 y 14 se define al ejemplo de la línea 9 el nombre de la función, y el nombre de sus parámetros respectivos.

```
1   [
2       [
3           "tamaño de var",
4           {
5               "entities": [[10, 13, 'TOKEN']]
6           }
7       ],
8       [
9           "correr nombrefuncion con parametros a, b",
10          {
11              "entities": [
12                  [ 7, 20, 'TOKEN'],
13                  [ 36, 37, 'TOKEN'],
14                  [ 39, 40, 'TOKEN']
15              ]
16          }
17      ],
18  ]
```

Figura 5.24: Ejemplo de la estructura usada para el entrenamiento y pruebas del Modelo NER con spaCy.

Una vez entrenado el modelo NER con spaCy se puede aplicar el algoritmo 1 de preprocesado descrito en el capítulo 3. En la Figura 5.25 se muestra como el algoritmo transforma el texto generado por el modelo Speech To Text a un texto que pueda entender con más facilidad el modelo Tranx al insertar los acentos a los nombres de variables, funciones y clases.

5.3 Python 3 a Java y C#

Las bibliotecas usadas para la transformación de código Python 3 a Java y C# fueron **avib-printers** para transformar el AST genérico a código Java y C#, **avib.coderetriever.mapper** encargado de transformar el código Python a el AST genérico, y **avib.coderetriever.gast** el cual tiene la estructura que define el AST genérico.

Figura 5.25: Resultado del algoritmo de preprocesado al recibir como entrada el texto generado por el modelo Speech To Text a partir de un audio en español.

La implementación de este componente la podemos dividir en 3 pasos como podemos ver en la Figura 5.26:

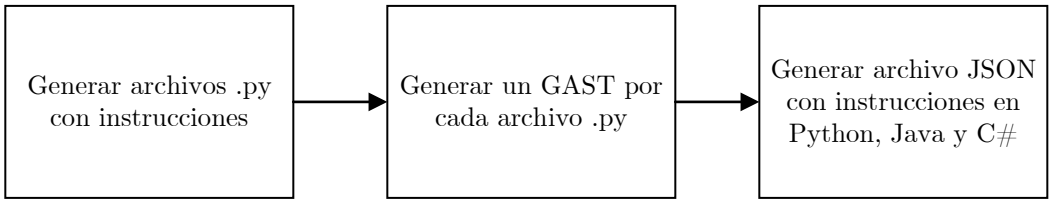

Figura 5.26: Flujo para la implementación del componente que transforma audios en español a Python 3.

1. Generar archivos .py con instrucciones: Pasar todas las instrucciones Python generadas por Tranx a archivos .py para que estos puedan ser cargados en el GAST. Código fuente del GAST `https://github.com/Softlab-TEC/GAST`.

2. Generar un archivo GAST por cada archivo .py: Cargar los archivos .py para que el Mapper los pase a la estructura del GAST. El código fuente del Mapper se encuentra en `https://github.com/Softlab-TEC/Mapper`.

3. Generar archivo JSON con instrucciones en Python, Java y C#: El Printer genera el código Java y de C# recorriendo el AST, guardando el resultado en archivos .java para el código Java y .cs para el código de C#. Guardar un archivo JSON en el sistema de archivos con los diferentes lenguajes de programación generados a partir de la instrucción en español. El código fuente del Printer se encuentra en `https://github.com/Softlab-TEC/Printers`.

```
1    class DummyClass:
2        ${code}
```

Figura 5.27: El texto *${code}* es remplazado por la instrucción en Python que generó Tranx. Por ejemplo: *def sumar(a, b).*

5.3.1 Generar archivos .py con instrucciones

Para el primer paso se utiliza el algoritmo 7 que consiste en cargar todas las instrucciones en Python a transformar, y guardarlas en un archivo para que luego el GAST pueda cargarlos. En la línea 1 se carga en memoria las instrucciones que provienen del JSON generado por Tranx. Luego se recorre cada instrucción y se obtiene el id, junto con el código Python generado (línea 3 y 4). Debido a que Tranx genera el código con espacios en blanco en operadores de comparación y aumento, en la línea 5 se remueven estos espacios para que el GAST los pueda interpretar. Cuando la instrucción es la definición de una función, se usa una plantilla que define la función en una clase Dummy (ver Figura 5.27), caso contrario se define la instrucción dentro de una plantilla que tiene una clase Dummy y una función Dummy (ver Figura 5.28). Estas plantillas se usan para que el GAST pueda recorrer el arból sintáctico abstracto, ya que el componente Mapper no soporta instrucciones Python que no estén dentro de una clase o función.

Algoritmo 7: Algoritmo para guardar instrucciones generadas por Tranx en archivos .py

Datos: *rutaJSON*

1 $ejemplos \leftarrow cargarArchivo(rutaJSON)$;
2 **para** *ejemplo en ejemplos* **hacer**
3 $\quad id \leftarrow ejemplo[\text{"}Id\text{"}]$;
4 $\quad codigoPython \leftarrow ejemplo[\text{"}Output\text{"}]$;
5 $\quad codigoPython \leftarrow removerEspaciosEnOperadores()$;
6 $\quad$ **si** $codigoPython.contiene(\text{"}def\text{"})$ **entonces**
7 $\quad\quad codigoPython \leftarrow asignarPlantillaParaFuncion(codigoPython)$
8 $\quad$ **en otro caso**
9 $\quad\quad codigoPython \leftarrow asignarPlantillaParaInstruccion(codigoPython)$
10 $\quad guardarCodigoEnArchivo('code -' + id + '.py', codigoPython)$;

5.3.2 Generar un archivo GAST por cada archivo .py

Como segundo paso, una vez generados todos los archivos .py, se cargan todos los archivos en memoria y uno por uno se empieza a transformar en un Árbol Sintactico Abstracto Genérico con el Mapper implementado para Python. Luego estos árboles genéricos se transforman a C# y Java utilizando el componente Printer, que permite recorrer el árbol

```
1    class DummyClass:
2      def dummyFunction():
3        ${code}
```

Figura 5.28: El texto *${code}* es remplazado por la instrucción en Python que generó Tranx. Por ejemplo: *a = 2 * a*

.

y transformar cada instrucción al respectivo lenguaje. El Printer genera archivos .java para las instrucciones en Java y .cs para las instrucciones en C#.

5.3.3 Generar archivo JSON con instrucciones en Python, Java y C#

El tercer y último paso consiste en generar un JSON que contenga la lista de instrucciones en los diferentes lenguajes de programación, asociados al id del ejemplo en EsPython. Para este paso se usó el algoritmo 8, el cual carga todos los archivos en memoria generados para cada lenguaje de programación (línea 1 a 3), luego se crea un hash map para los archivos de C# y Java, donde la llave es el id del ejemplo de EsPython y el valor es el código fuente que contiene el archivo (línea 4 a 5). Se recorre todos los archivos Python y se obtiene el id de EsPython con base en el nombre del archivo (paso 6 a 7), para luego cargar el código que tiene cada archivo y remover cualquier token dummy usado en pasos anteriores (línea 8 a 11). Como paso final, se crea el objeto para guardarlo en el archivo JSON final utilizando los diferentes lenguajes de programación y el id de EsPython (ver Figura 5.29)

Algoritmo 8: Algoritmo para generar JSON con los diferentes lenguajes de programación

```
1  archivosPython ← cargarArchivo(carpetaPython);
2  archivosCSharp ← cargarArchivo(carpetaCSharp);
3  archivosJava ← cargarArchivo(carpetaJava);
4  hashMapCSharp ← generarHashMap(archivosCSharp);
5  hashMapJava ← generarHashMap(archivosJava);
6  para archivoPy en archivosPython hacer
7      id ← obtenerIdDelArchivo(archivoPy);
8      codigoPython ← leerArchivo(archivoPy);
9      codigoPython ← removerTokensDummies(codigoPython);
10     codigoCSharp ← removerTokensDummies(hashMapCSharp[id]);
11     codigoJava ← removerTokensDummies(hashMapJava[id]);
12     guardarResultado(id, codigoPython, codigoCSharp, codigoJava);
```

```
[
    ...,
    {
        "python": "if suma == 99:\n    ",
        "csharp": "if (suma == 99)\r\n      {\r\n          \r\n      }",
        "java": "if (suma == 99) {\r\n      \r\n    }",
        "id": 10
    },
    {
        "python": "continue",
        "csharp": "continue;",
        "java": "continue;",
        "id": 100
    },
    ...
]
```

Figura 5.29: Ejemplo del JSON generado en el paso final, después de guardar el resultado de los diferentes lenguajes de programación.

Capítulo 6

Resultados

En esta sección presentamos los resultados obtenidos de los diferentes componentes utilizados con modelo desarrollado en esta investigación. En la sección 6.1 se muestran los resultados BLEU y ejemplos de código Python generado por el modelo Tranx entrenado con el conjunto de datos EsPython sin utilizar ninguna técnica de aumento de datos, y además se muestran los resultados del modelo Tranx entrenado con las diferentes técnicas de aumento de datos aplicadas al conjunto de datos de entrenamiento.

En la sección 6.2 se detalló los resultados obtenidos al transformar audios en español a código Python utilizando las dos versiones documentadas en el trabajo, sin preprocesado y con preprocesado. Por último, en la sección 6.3 se muestra el código generado en Java, C# y Python al utilizar los tres componentes de manera conjunta para transformar audios con instrucciones en español a código.

6.1 Texto en español a Python 3

Los resultados que se muestran en esta sección fueron obtenidos a partir de las pruebas realizadas con el conjunto de datos "espython.test.json". Se realizaron pruebas con los diferentes modelos Tranx entrenados en el capitulo 5. Para cada modelo detallamos su puntaje BLEU y si fue entrenado con un batch size de 8 o 128, además se muestra algunos ejemplos generados al realizar las pruebas. Para ver los resultados completos de todos los modelos que transformaron texto en español a Python se puede consultar la siguiente dirección `https://github.com/antgonto/seq2ast/tree/main/results`.

6.1.1 Resultados Tranx usando EsPython

En la Tabla 6.1 podemos ver los resultados de las pruebas realizadas para el modelo entrenado con el conjunto de datos EsPython original sin utilizar ninguna técnica de aumento de datos.

El conjunto de datos sin aumentar obtuvo un puntaje BLEU de 0.8373 con un batch size de 8 y un puntaje BLEU de 0.5869 con un batch size de 128. Esta diferencia notable en el puntaje con los diferentes batch sizes es probablemente debido a que el conjunto de datos es relativamente pequeño, por lo que utilizar un batch size de 128 disminuye la cantidad de iteraciones para minimizar el error de la función de optimización.

Conjunto de datos	Batch size	BLEU
EsPython	8	0.8373
EsPython	128	0.5869

Tabla 6.1: Modelo Tranx entrenado con el conjunto de datos EsPython sin aumentar.

En la Figura 6.1 se muestran algunas de las instrucciones en español transformadas a código Python. Para las instrucciones *"mientras 'var1' sea diferente a -1 entonces"*, y *"repetir 4 elevado a la 6 veces"*, se puede observar como la versión entrenada con un batch size de 8 generó el código con la sintaxis y semántica esperada, mientras que la versión entrenada con un batch size de 128 no cumplió con la semántica esperada. En el tercer ejemplo las dos versiones generaron el código esperado. Hay que recordar que el token "*indtoken()*" es utilizado para instrucciones de bloques ya que Python espera al menos una instrucción dentro del bloque, sin embargo estos tokens son eliminados posteriormente y no son considerados como parte de la semántica en estos resultados.

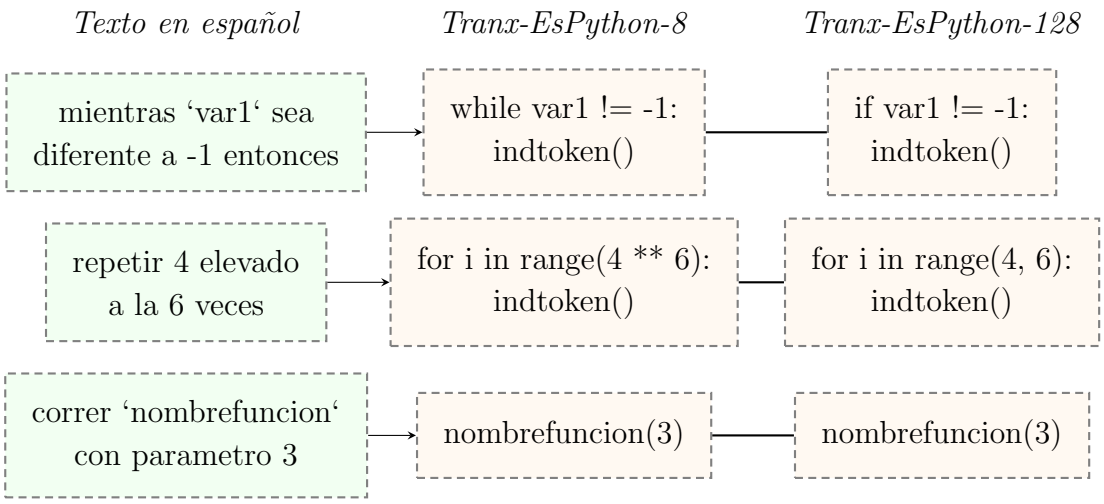

Figura 6.1: Código Python generado a partir de instrucciones en español con el modelo Tranx entrenado con el conjunto de datos EsPython sin aumentar utilizando un batch size de 8 y de 128.

6.1.2 Resultados Tranx al aumentar datos de EsPython

En la Tabla 6.2 y 6.3 se muestran los resultados de las pruebas realizadas utilizando los modelos Tranx entrenados con diferentes técnicas de aumento de datos. En la mayoría de los casos, el utilizar una técnica de aumento de datos incrementó el puntaje BLEU.

Utilizando un batch size de 8, la única técnica que tuvo un puntaje menor que la versión sin aumentar fue Split Word, que obtuvo un puntaje de 0.8266. En el caso del batch size de 128, la versión de Random Char Insert y Random Char Substitute, fueron las que tuvieron menor puntaje que la versión sin aumentar. Las otras técnicas si tuvieron mejor puntaje.

Técnica	**BLEU**
Synonim Augmentation	0.8895
Contextual Word Embedding	**0.8976**
Keyboard	0.8895
Random char-delete	0.8441
Random char-insert	0.8782
Random char-substitute	0.8796
Random char-swap	0.8875
Split word	0.8266

Tabla 6.2: Modelo entrenado con el conjunto de datos EsPython original y aumentado con diferentes técnicas, usando un batch size de 8.

Técnica	**BLEU**
Synonim Augmentation	0.8637
Contextual Word Embedding	0.8652
Keyboard	0.8637
Random char-delete	0.8254
Random char-insert	0.5013
Random char-substitute	0.4377
Random char-swap	0.8416
Split word	**0.8936**

Tabla 6.3: Modelo entrenado con el conjunto de datos EsPython original y aumentado con diferentes técnicas, usando un batch size de 128.

La siguiente lista resume los resultados obtenidos con cada una de las técnicas de aumento de datos utilizadas:

Synonim Augmentation

Esta técnica tuvo un un puntaje BLEU de 0.8895 utilizando un batch size de 8 y un 0.8637 al utilizar un batch size de 128. En la Figura 6.2 se pueden observar algunos resultados del código Python generado a partir del texto con instrucciones en español. Instrucciones como *"mientras 'a' sea mayor o igual que 0 entonces"* generaron el código Python esperado, mientras que instrucciones como *realizar 'nombrefuncion'* no lograron ser transformadas al código esperado el cual era *"nombrefuncion()"*. Otra de las instrucciones que no generó el

código esperado fue *"Si la variable 'a' es menor que 0 entonces llame la funcion 'guardar'"*, ya que en la versión con batch size de 8 asigna un valor a una variable en lugar de llamar la función, mientras que la versión de 128 no realiza nada después de la instrucción "*if*".

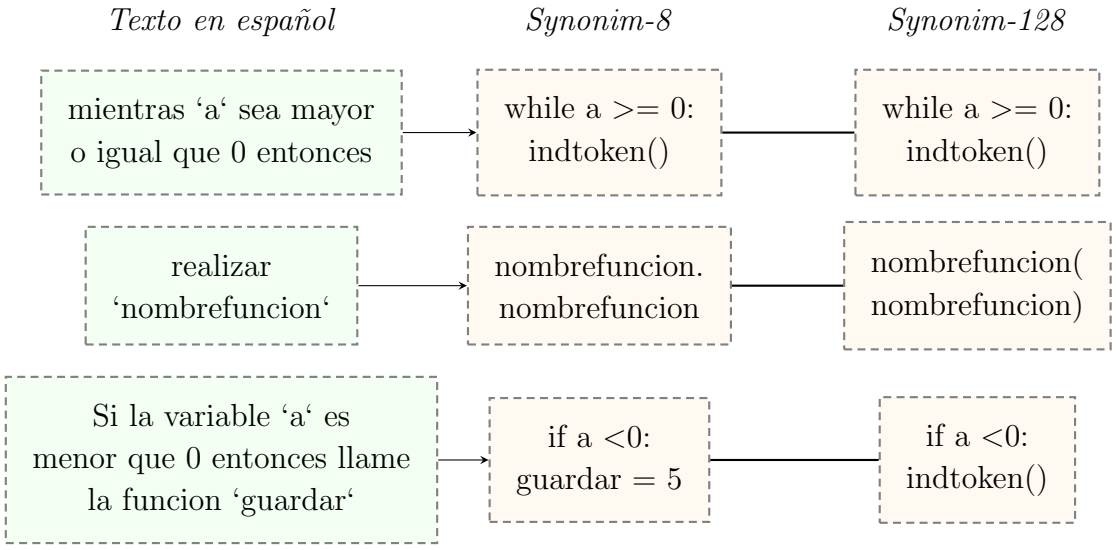

Figura 6.2: Código Python generado a partir de instrucciones en español con el modelo Tranx entrenado con el conjunto de datos EsPython aumentado con la técnica de Synonim Augmentation utilizando un batch size de 8 y de 128.

Contextual Word Embedding

Con el modelo Tranx entrenado con ejemplos aumentados con Contextual Word Embedding tuvo un un puntaje BLEU de 0.8976 utilizando un batch size de 8 y un 0.8652 al utilizar un batch size de 128. En la Figura 6.3 se pueden observar algunos resultados del código Python generado a partir del texto con instrucciones en español.

Instrucciones como "*Instanciar clase 'Carro'*" generó el resultado esperado en la versión de batch size de 8, pero no fue correcto en la versión de 128. Instrucciones como "*realizar 'nombrefuncion'*" también generó el código esperado en la versión de 8, pero no en la de 128. La instrucción "*Si la variable 'b' es menor o igual que 999*", generó el resultado esperado en ambas versiones.

Keyboard augmentation

Con la técnica de Keyboard Augmentation el modelo Tranx tuvo un un puntaje BLEU de 0.8895 al usar un batch size de 8 y un 0.8637 al usar un batch size de 128. En este caso, la instrucción "*Instanciar clase 'Carro'*" no fue transformada correctamente por ninguna versión como se logra observar en la Figura 6.4. La instrucción "*realizar 'nombrefuncion'*" tampoco generó de forma correcta el código Python esperado. En el caso de la instrucción "*Si la variable 'b' es menor o igual que 99*", ambos modelos lograron transformar el texto en español al código esperado.

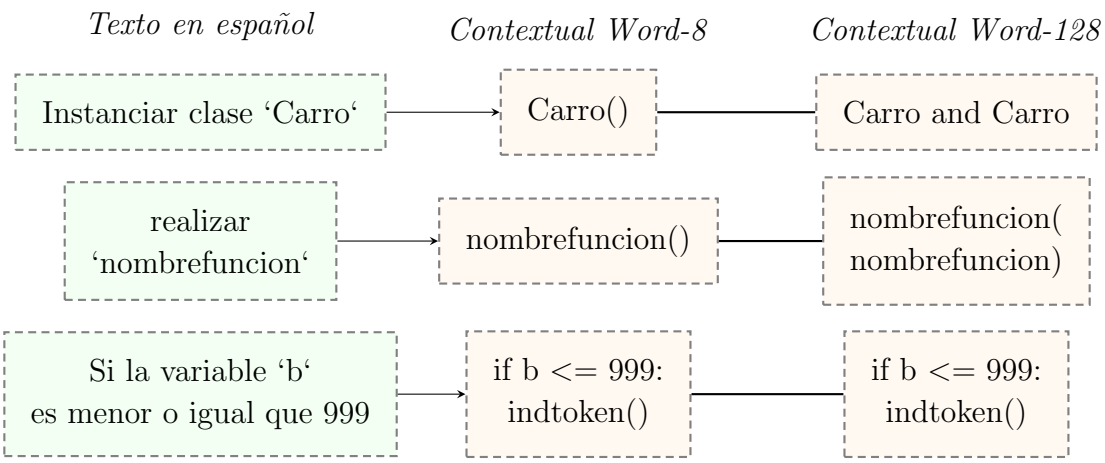

Figura 6.3: Código Python generado a partir de instrucciones en español con el modelo Tranx entrenado con el conjunto de datos EsPython aumentado con la técnica de Contextual Word Embedding utilizando un batch size de 8 y de 128.

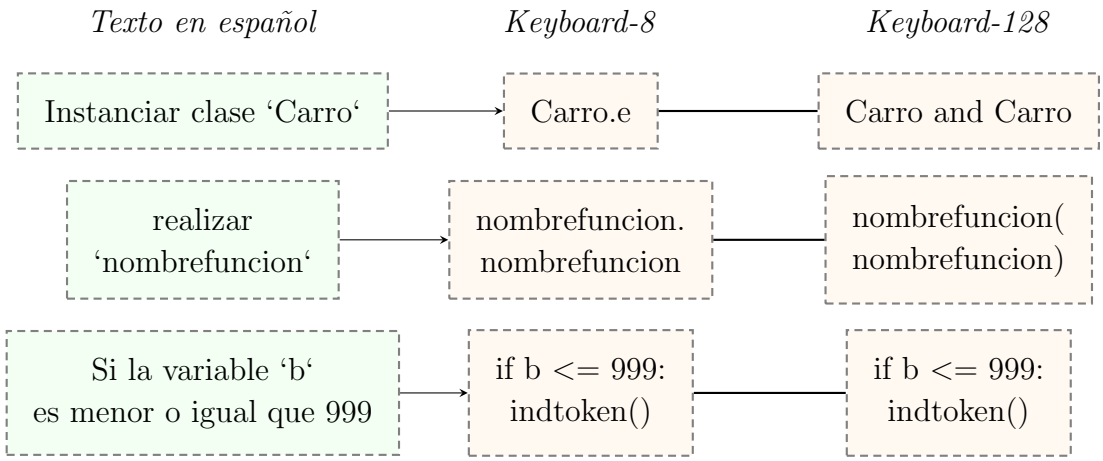

Figura 6.4: Código Python generado a partir de instrucciones en español con el modelo Tranx entrenado con el conjunto de datos EsPython aumentado con la técnica de Keyboard Augmentation utilizando un batch size de 8 y de 128.

Random char Delete

La variante Delete de la técnica Random Char tuvo un un puntaje BLEU de 0.8441 utilizando un batch size de 8 y un 0.8254 al utilizar un batch size de 128. En este caso tampoco se logró transformar de manera correcta la instrucción "*Instanciar clase ‘Carro‘*" como se logra ver en la Figura 6.5.

La versión con un batch size de 128 si instancia la clase carro, sin embargo, lo hace con un parámetro no indicado. Instrucciones como "*realizar ‘nombrefuncion‘*" si generó el código esperado en ambas versiones, al igual que la instrucción "*Si la variable ‘b‘ es menor o igual que 999*".

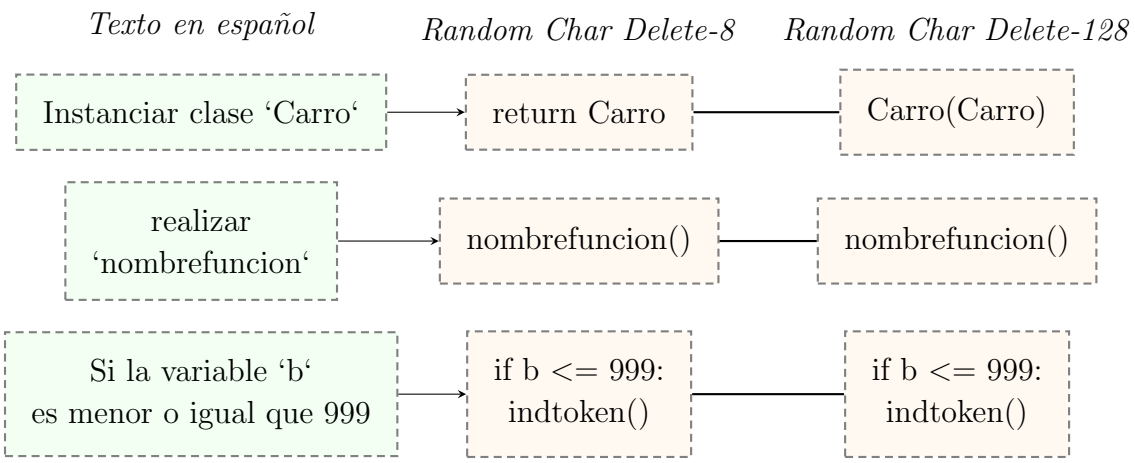

Figura 6.5: Código Python generado a partir de instrucciones en español con el modelo Tranx entrenado con el conjunto de datos EsPython aumentado con la técnica de Random Char Delete utilizando un batch size de 8 y de 128.

Random Char Insert

Con la variante insert, el puntaje BLEU fue de 0.8782 utilizando un batch size de 8 y un 0.5013 al utilizar un batch size de 128. A diferencia de la versión de delete, los ejemplos seleccionados fueron transformados a código Python de manera correcta tanto en la versión con un batch size de 8 y la versión de 128. En la Figura 6.6 se pueden observar los resultados obtenidos.

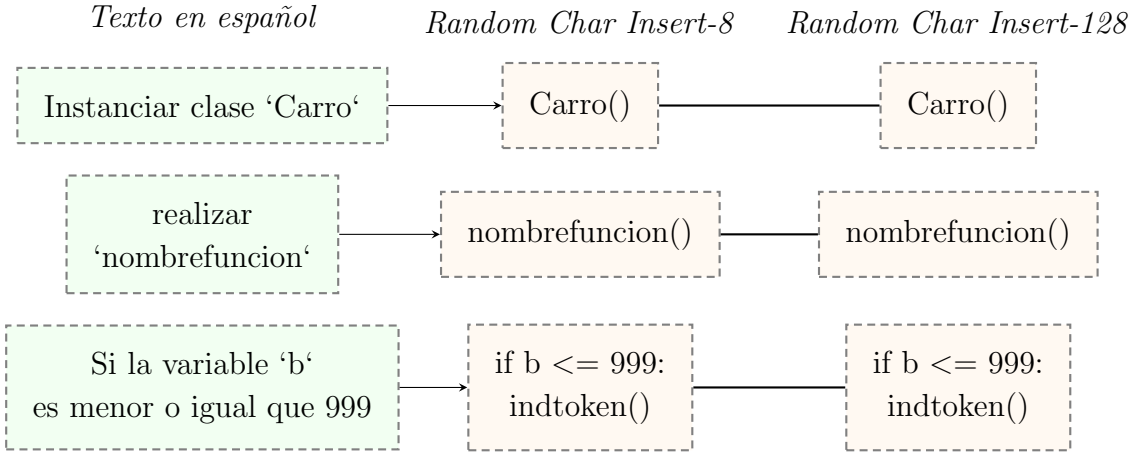

Figura 6.6: Código Python generado a partir de instrucciones en español con el modelo Tranx entrenado con el conjunto de datos EsPython aumentado con la técnica de Random Char Insert utilizando un batch size de 8 y de 128.

Random Char Substitute

En el caso de la variante Substitute se obtuvo un un puntaje BLEU de 0.8796 utilizando un batch size de 8 y un 0.4377 al utilizar un batch size de 128. La instrucción "*Instanciar clase*

'Carro'" generó el resultado esperado en la versión de batch size de 128, pero en la versión de 8 en lugar de instanciar la clase asignó un valor a una variable. Las otras instrucciones fueron transformadas a código Python de manera correcta en ambas versiones.

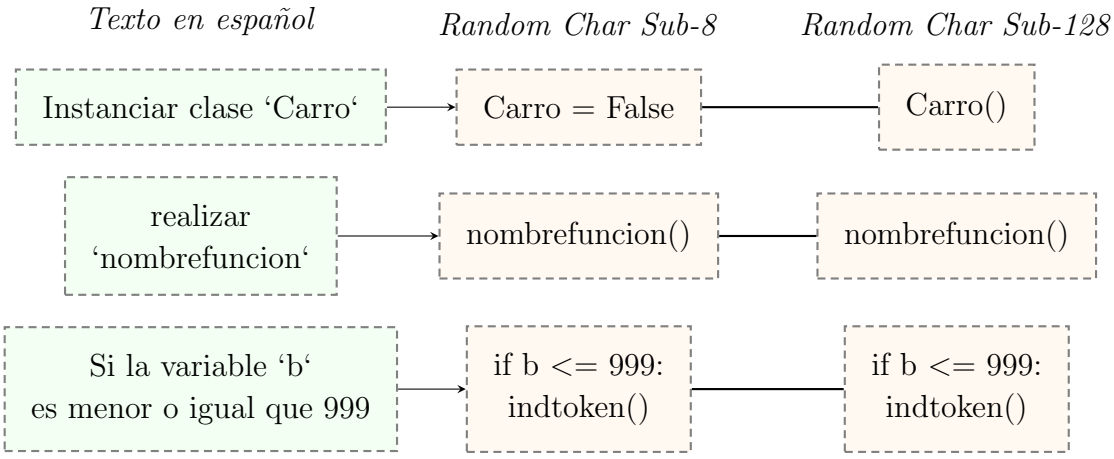

Figura 6.7: Código Python generado a partir de instrucciones en español con el modelo Tranx entrenado con el conjunto de datos EsPython aumentado con la técnica de Random Char Substitute utilizando un batch size de 8 y de 128.

Random Char Swap

El modelo Tranx entrenado con ejemplos aumentados con la variante Random Char Swap obtuvo un BLEU de 0.8875 al usar un batch size de 8 y un 0.8416 al usar un batch size de 128. En la Figura 6.8 se observa como la instrucción "*Instanciar clase 'Carro'*" generó el resultado esperado en la versión de batch size de 8, pero no fue correcto en la versión de 128 ya que asignó el valor "False" a una variable. Instrucciones como *realizar 'nombrefuncion'* no fueron transformados de manera correcta por ninguna versión. La instrucción *Si la variable 'b' es menor o igual que 999*, generó el resultado esperado en ambas versiones.

Split word

La técnica Split Word obtuvo puntaje BLEU de 0.8266 utilizando un batch size de 8 y un 0.8936 al utilizar un batch size de 128. En la Figura 6.9 se puede observar que el código Python generado para la instrucción "*Instanciar clase 'Carro'*" no generó el resultado esperado en ninguna versión. La instrucción "*realizar 'nombrefuncion'*" generó el código esperado en ambas versiones.

En el caso de la instrucción "*Si la variable 'a' es menor que 0 entonces llame la funcion 'guardar'*, no generó el código esperado ya que en ambas versiones crearon una variable "guardar" que no fue especificada en la instrucción.

En la Tabla 6.4 se puede ver la evaluación de diferentes conjuntos de datos utilizando el modelo Tranx para la tarea de generación de código Python. En esta tabla incluimos el

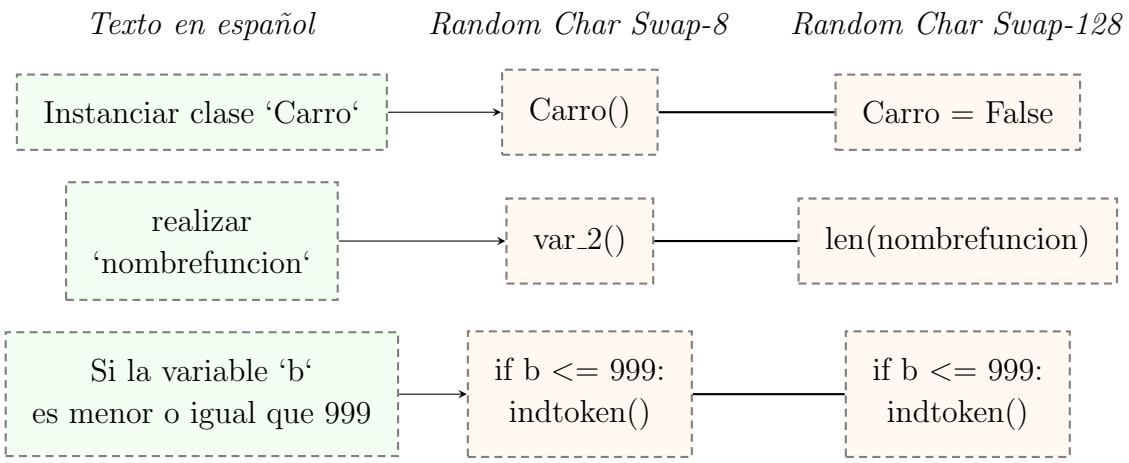

Figura 6.8: Código Python generado a partir de instrucciones en español con el modelo Tranx entrenado con el conjunto de datos EsPython aumentado con la técnica de Random Char Swap utilizando un batch size de 8 y de 128.

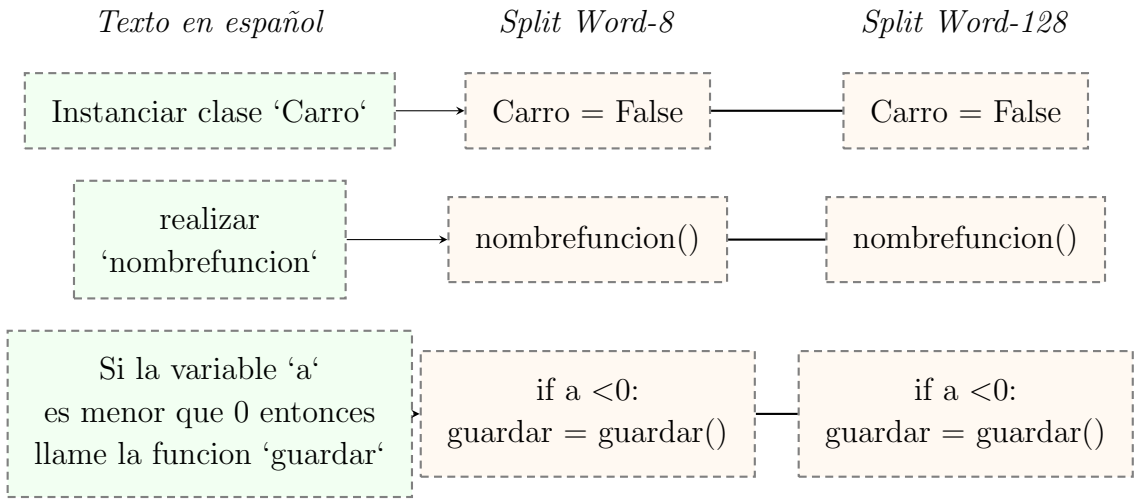

Figura 6.9: Código Python generado a partir de instrucciones en español con el modelo Tranx entrenado con el conjunto de datos EsPython aumentado con la técnica de Split Word utilizando un batch size de 8 y de 128.

resultado del conjunto de datos EsPython original y con la técnica de aumento de datos Contextual Word Embedding . Los conjuntos de datos de DJANGO [35], CoNaLa [63] son conjuntos de datos con intenciones en inglés y código fuente en Python. El conjunto de datos MCoNaLa es un conjuto de datos multilenguaje que incluye inglés, español, japonés y ruso [58].

6.2 Audio en español a Python 3

La Tabla 6.5 contiene los resultados obtenidos en esta investigación para transformar audios a código Python. La versión sin preprocesado obtuvo un puntaje BLEU de 0.5690

y la versión con preprocesado obtuvo un puntaje BLEU de 0.7543. El modelo Tranx que se utilizó fue el entrenado con el conjunto de datos EsPython aumentado con Contextual Word Embedding con un batch size de 8.

Conjunto de Datos	Métrica	Puntaje
EsPython-Original	BLEU	83.7
EsPython-Contextual Word Embedding	BLEU	89.76
DJANGO	Accuracy	84.5
CoNaLa	BLEU	24.30
MCoNaLa-ES-TEST	BLEU	2.46

Tabla 6.4: Resultados del modelo Tranx con diferentes conjuntos de datos.

Se utilizó la versión sin preprocesado como modelo base para realizar pruebas con paired bootstrap y lograr demostrar si hay una diferencia estadísticamente significativa [28, 12]. Se comparó los resultados de la versión sin preprocesado y con preprocesado, arrojando un p-value de 0.00049, lo cual indica que sí hay una diferencia estadísticamente significativa entre ambos modelos, por lo que podemos decir que nuestra versión con preprocesado tuvo mejores resultados que nuestra versión sin preprocesado.

A continuación se dan a conocer algunos resultados obtenidos a partir de la transformación de audios en español a Python con la versión sin preprocesado y con preprocesado.

6.2.1 Versión sin preprocesado

En la versión sin preprocesado algunos resultados tuvieron los problemas observados al realizar la implementación como nombres de variables, el formato necesario para el modelo Tranx y textos que no reflejaban la intención del audio. Por ejemplo, como se puede observar en la Figura 6.10, un audio con la siguiente intención: "*ejecutar nombre función con parámetro a*", el resultado en código Python fue el siguiente: "*while true: var_0(a)*". El resultado esperado era "*nombrefuncion(a)*". En este caso el modelo no logró generar un código que reflejara la intención del audio debido a que el modelo Tranx espera que el nombre de la variable y la función esté entre comillas. En la versión sin preprocesado solo se envía el texto generado por el modelo Speech To Text de Azure por lo que parece afectar el resultado en Python.

Speech To Text	Texto a Python	BLEU
Azure	Tranx	56.90
Azure+Preprocesado	Tranx	75.43

Tabla 6.5: Resultado BLEU al transformar instrucciones en español a partir de audios a código Python, usando preprocesado y sin usar preprocesado.

En el siguiente ejemplo, el audio "*Pasar la cadena var a entero*", el código Python generado fue "*int(cadena)*". Aunque la intención fue correctamente interpretada, el modelo Tranx

no pudo identificar el nombre de la variable correctamente y el resultado final convirtió a un entero "cadena" en lugar de "a".

En el caso del audio "*Defina la clase pajaro*", el resultado fue "*class Carro: dummy attribute = 0*". En este caso la intención también era la correcta, pero el nombre de la clase no.

6.2.2 Versión con preprocesado

En la versión con preprocesado se lograron corregir varios de los problemas presentados en la versión sin preprocesado. Por ejemplo, como se puede observar en la Figura 6.11, el audio con la intención: "*ejecutar nombre función con parámetro a*", el código generado fue "*def nombrefuncion(a): indtoken()*". Aunque la intención no es la correcta debido a que está declarando una función en lugar de ejecutarla, el código generado sí mejora con respecto la versión sin preprocesado, la cual pierde totalmente la idea que transmitía el audio en español.

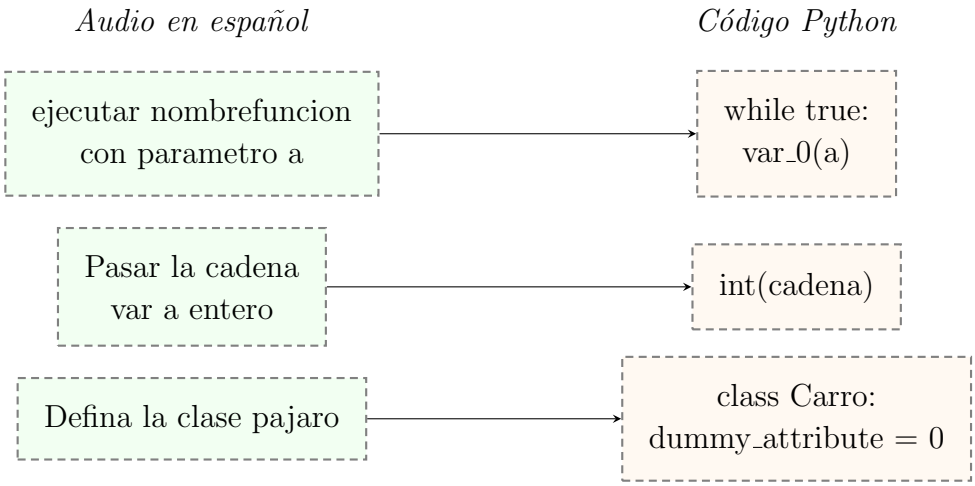

Figura 6.10: Resultados al transformar audios en español a código Python utilizando Azure sin preprocesado y Tranx.

Para el audio "*Pasar la cadena var a entero*", el código generado fue "*int(var)*". En este caso si se respetó el nombre de la variable y el resultado es exactamente el esperado.

Otro ejemplo de los resultados es la instrucción en español "*Defina la clase pájaro*" que se transformó al código Python "*class Carro: dummy attribute = 0*". Al igual que la versión sin preprocesado, el nombre de la clase no fue el correcto.

6.3 Evaluación de código Python, Java y C#

En esta sección se muestran los resultados de la evaluación humana de la transformación de audios en español a código en Python, C# y Java, generado por los tres componentes

en conjunto que forman el modelo desarrollado en esta investigación. Las pruebas se realizaron con los audios grabados manualmente, los cuales son la entrada del modelo, estos audios se transforman a texto, el texto luego es transformado a código Python, y por último a partir del código Python se genera código Java y C#.

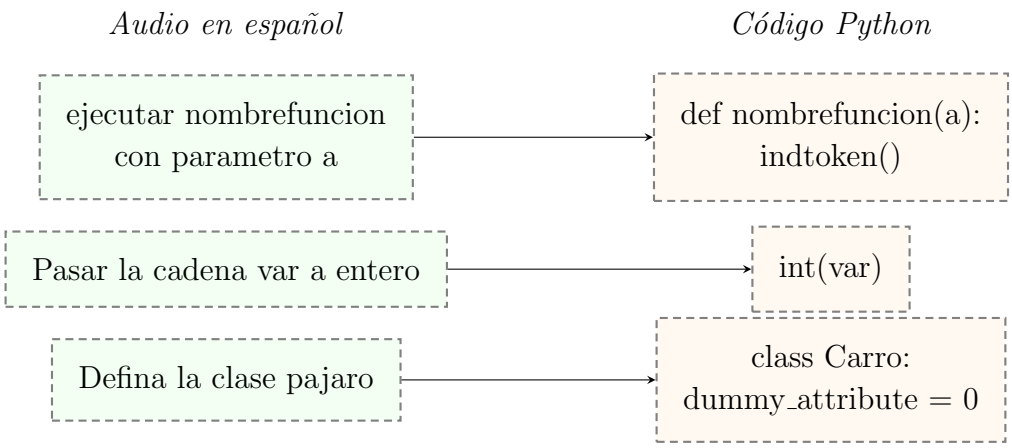

Figura 6.11: Resultados al transformar audios en español a código Python utilizando Azure con preprocesado y Tranx.

La Figura 6.12 muestra la cantidad de ejemplos que tenían una sintaxis valida en el lenguaje especificado. En el caso de Python, 54 de 55 audios fueron transformados con la sintaxis correcta. En el caso de C#, 32 audios generaron código sin errores sintácticos y 23 con algún error de sintaxis. Con el lenguaje de Java se obtuvo el mismo resultado que C#.

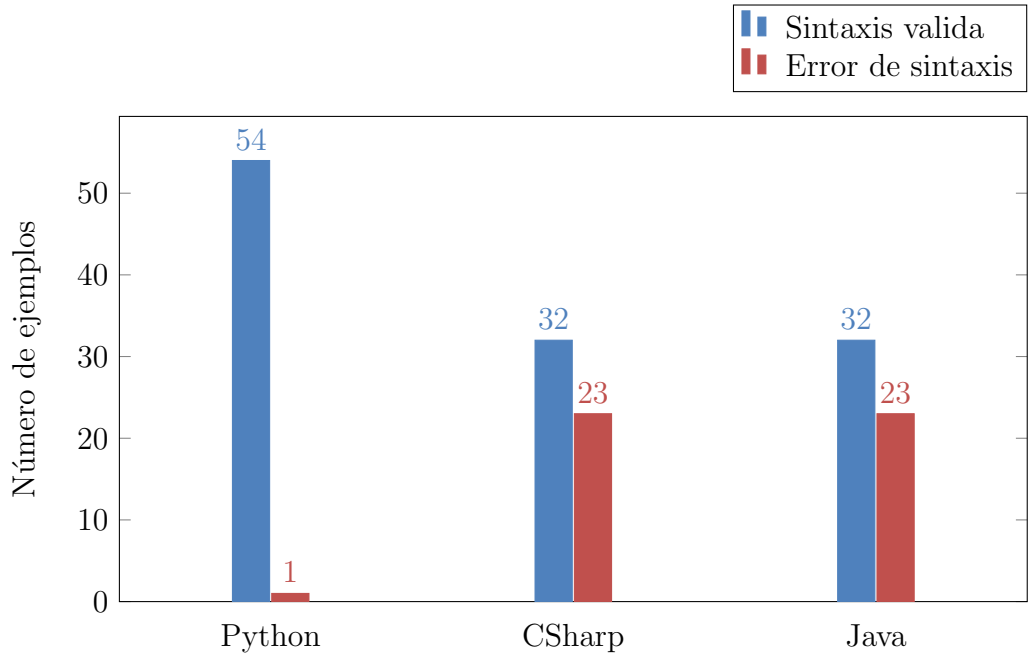

Figura 6.12: Número de ejemplos que cumplen o no cumplen con la sintaxis del lenguaje de programación de salida.

En la Figura 6.13 se puede ver el audio donde el código Python tuvo un error de sintaxis.

En este caso la instrucción en español especificaba como parámetro una variable "pos", sin embargo esta variable fue cambiada por "i", causando que pos no exista al usarse en la condición del if. El mismo problema se repite al generar el código para C# y Java, ya que el modelo siempre usa como base el código Python generado por Tranx. Además, la función "len" no existe en C# y Java, por lo que genera un error de sintaxis adicional en estos lenguajes.

```python
# Python
def func(lista, i):
  for i in range(len(i)):
    if pos == i[i]:
      return true
```

```csharp
// C#
func( lista,  i)
{
  for ( i; i < len(i);
  i+=1)
  {
    if (pos == i[i])
    {
      return true;
    }
  }
}
```

```java
// Java
func( lista,  i) {
  for ( i; i < len(i);
  i+=1) {
    if (pos == i[i]) {
      return true;
    }
  }
}
```

Figura 6.13: Código Python, C# y Java generado a partir del audio *"para func con parámetros lista, pos. Repetir tamaño de lista veces. Si pos igual que i entonces devolver lista en posición i"* con el modelo implementado.

Otro problema encontrado en múltiples ejemplos fue la inicialización de las instrucciones "for" como se muestra en la Figura 6.14. A diferencia de Python, C# y Java necesitan inicializar la variable "i" para realizar las iteraciones, en caso contrario, el compilador arrojará un error de sintaxis.

```python
# Python
for i in range(4 ** 6):
```

```csharp
// C#
for ( i; i < 4 ** 6; i+=1)
{

}
```

```java
// Java
for ( i; i < 4 ** 6; i+=1) {

}
```

Figura 6.14: Código Python, C# y Java generado a partir del audio *"repetir 4 elevado a la 6 veces"* con el modelo implementado.

En la Figura 6.15 se muestra cono el audio *"si suma es igual que 99"* fue transformado a los diferentes lenguajes de programación de forma correcta, sin ningún error de sintaxis.

Los resultados de cuántos audios fueron transformados a código en cada lenguaje de

```python
# Python
if suma == 99:
```

```csharp
// C#
if (suma == 99)
{

}
```

```java
// Java
if (suma == 99) {

}
```

Figura 6.15: Código Python, C# y Java generado a partir del audio *"si suma es igual que 99"* con el modelo implementado.

programación respetando la intención original en español se puede ver en la Figura 6.16. En este caso se evaluó si el código generado contemplaba la idea especificada en el audio en español, sin tomar en cuenta nombres de variables o librerías de un framework específico. Para el caso de Python, 35 ejemplos si cumplían con la intención del audio y 20 no la cumplían. En el caso de C# y Java, 34 audios generaron código que cumplía con la intención mencionada en el audio y 21 no la cumplían.

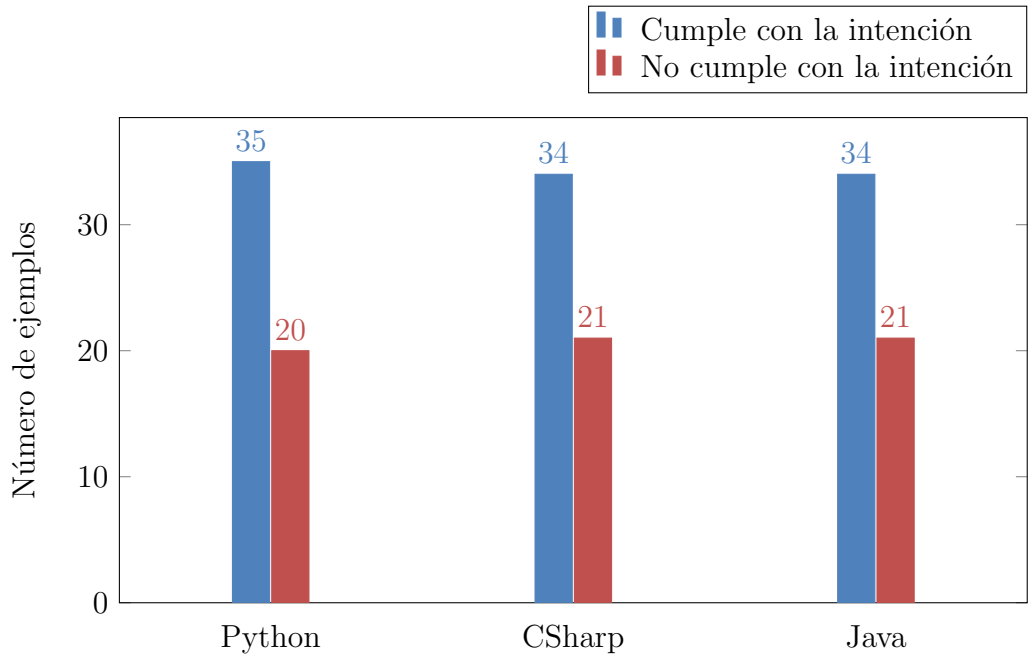

Figura 6.16: Número de ejemplos que cumplen o no cumplen con la semántica del lenguaje de programación de salida.

Uno de los ejemplos que generaron un código congruente con lo dicho en el audio y con una sintaxis correcta fue *"mientras 'a' sea menor que 100"* (ver Figura 6.17). El código generado era el esperado tanto en Python como en C# y Java.

En la Figura 6.18 se muestra un ejemplo donde el audio generó un código que refleja la intención de las instrucciones en español, sin embargo posee un error de sintaxis en C# y Java debido a que la función "int" no existe en las librerías de estos lenguajes. La intención

```python
# Python
while a < 100:
```

```csharp
// C#
while (a < 100)
{

}
```

```java
// Java
while (a < 100) {

}
```

Figura 6.17: Código Python, C# y Java generado a partir del audio *"mientras 'a' sea menor que 100"* con el modelo implementado.

"pasar la cadena 'var' a entero" espera transformar la variable "var" a un número entero, por lo que en el caso de C# se hubiera esperado usar una función como "Int32.Parse", y en Java se podría utilizar "Integer.parseInt".

```python
# Python
int(var)
```

```csharp
// C#
int(var);
```

```java
// Java
int(var);
```

Figura 6.18: Código Python, C# y Java generado a partir del audio *"pasar la cadena 'var' a entero"* con el modelo implementado.

En el caso del audio *"mientras 'var1' sea diferente a -1 entonces"* que se muestra en la Figura 6.19, el código generado en todos los lenguajes no cumple con la intención original en español, debido a que el número que se compara en el código es positivo, sin embargo en el audio original se especifica un menos uno (un uno negativo). Este error puede cambiar completamente el resultado de un programa por lo que se trata como un caso donde no se cumple con la intención original del audio en español.

```python
# Python
while var1 != 1:
```

```csharp
// C#
while (var1 != 1)
{

}
```

```java
// Java
while (var1 != 1) {

}
```

Figura 6.19: Código Python, C# y Java generado a partir del audio *"mientras 'var1' sea diferente a -1 entonces"* con el modelo implementado.

Capítulo 7

Conclusiones

7.1 Conclusiones

En este trabajo presentamos un modelo capaz de generar código a partir de instrucciones provenientes de audios en español. Nuestro modelo fue diseñado bajo el paradigma divide y conquista, permitiendo separarlo en tres componentes principales: el primer componente recibe como entrada un audio para ser transformado a texto plano, el segundo componente transforma el texto plano a código Python, y el tercer componente nos permite transformar el código Python generado a otros lenguajes de programación como C# y Java.

Se entrenaron múltiples modelos usando la arquitectura del autoencoder Tranx y utilizando técnicas de aumento de datos para incrementar el número de ejemplos del conjunto de datos EsPython. Este conjunto de datos fue creado de forma manual para tener ejemplos de instrucciones en español con su equivalente en código Python. Al realizar las pruebas se observó que es importante tener en cuenta el tamaño del conjunto de datos para ajustar los parámetros del modelo como el batch size, ya que puede variar de forma importante el resultado. También se logró identificar una mejora en el puntaje BLEU de los modelos al aumentar los datos de entrenamiento con ejemplos sintéticos, sin embargo, no todas las técnicas de aumento de datos mejoraron el resultado, por lo que es también importante experimentar con múltiples técnicas de aumento de datos cuando se requiera incrementar los ejemplos para el entrenamiento de un modelo.

Para transformar los audios en español a texto plano se crearon dos versiones. La primera versión utiliza el modelo Speech To Text de Azure, y la segunda versión preprocesa la respuesta de Azure con un modelo NER y una serie de algoritmos que ayudan a dar el formato necesario al texto para que este sea recibido como entrada para el modelo Tranx. Al comparar las dos versiones en los experimentos se comprobó que la versión con preprocesado del texto en español mejora el puntaje BLEU al transformar audios en español a Python. Contrario a lo que inicialmente se pensó, unir un modelo Speach To Text con un modelo de generación de código como Tranx no funciona de una manera "Plug and Play", si no que requiere ajustes al texto generado que es requerido como entrada

para el modelo de generación de código.

Para transformar los audios en español a diferentes lenguajes de programación se utilizaron los tres componentes principales de nuestro modelo en conjunto. Para transformar audio a texto se utilizó la API Speech To Text de Azure con el algoritmo de preprocesado debido a que tuvo mejores resultados. Para transformar el texto en español a Python se utilizó el modelo Tranx entrenado con la técnica de Contextual Word Embedding y con un batch size de 8 ya que fue el modelo que obtuvo el puntaje BLEU más alto. Por último, el código Python fue transformado a C# y Java con el GAST, el cual permite modelar el código Python en un árbol sintáctico abstracto genérico para luego recorrerlo y pasarlo a código en otros lenguajes de programación.

Al analizar los resultados encontramos que algunas de las palabras utilizadas en el conjunto de datos en realidad estaban en inglés, pero al ser tan utilizadas en programación, estas pueden pasar desapercibidas al crear un conjunto de datos por personas con experiencia en lenguajes de programación. Por lo que palabras como string, pueden quedar en un conjunto de datos que idealmente es para el idioma español. Al utilizar un modelo Speach To Text entrenado en español, el texto resultante podía en ocasiones tener errores ya que no lograba identificar estas palabras en inglés, enviando al modelo Tranx palabras que no eran las indicadas en el audio, y por lo tanto, generar código que no era el esperado. Para solucionar este problema se podría pensar en utilizar modelos Speach To Text que sean multilenguajes o verificar que el conjunto de datos no contiene palabras ajenas al idioma utilizado en los ejemplos.

Al realizar una evaluación humana de nuestro modelo se logró identificar retos en la generación de código en diferentes lenguajes de programación como las librerías que utilizan los diferentes lenguajes, y como estas no pueden ser transformadas actualmente por el GAST. Tambien las diferencias que existen entre los lenguajes de programación de tipado dinámico y los que son fuertemente tipados representan un reto en este tipo de tareas debido a que es difícil asumir cuál es el tipo necesario a escoger si se requiere transformar una instrucción proveniente de Python a otro lenguaje de programación.

Muchas de las investigaciones para la generación de código a partir de lenguaje natural utilizan conjuntos de datos en inglés. A pesar de que hay trabajos donde se utilizan conjuntos de datos multilenguaje, son pocas las opciones para el entrenamiento de estos modelos usando el lenguaje español. Nuestro conjunto de datos se basa en una serie de instrucciones en español similar a un pseudocódigo, con una mayor generalidad para evitar la rigidez de una sintaxis de un lenguaje de programación.

7.2 Trabajos futuros

Debido a que nuestro modelo utiliza un paradigma de divide y conquista, es posible realizar diferentes pruebas de manera fácil al estar separado en tres componentes. Para la transformación de audios en español se podría utilizar otros modelos de Speech To Text

para verificar si los resultados varían al generar código. En este trabajo se utilizó el idioma español, pero también se podría utilizar modelos multilenguajes.

Los algoritmos implementados en la investigación para la generación de variantes de oraciones son pruebas de concepto, por lo que es muy posible que estos puedan ser optimizados en investigaciones que lo requieran donde el rendimiento es importante.

En el caso de generación de código a partir de texto en español se pueden intentar otros modelos como CodeGen o CodeT, realizando un fine tunning con el conjunto de datos EsPython. Con los nuevos modelos LLM se requiere mayor capacidad de cómputo pero podrían mejorar los resultados si se logran entrenan con instrucciones en español. Aunque el conjunto de datos es relativamente pequeño en comparación a los conjuntos de datos para entrenar los modelos LLM, se podrían aplicar las técnicas de aumento de datos y también recolectar ejemplos de sitios web en español como Stack Overflow, GitHub o foros.

El utilizar modelos LLM permitiría utilizar métricas que se han creado específicamente para la tarea de generación de código como Pass@k [65]. Esta métrica mide la funcionalidad de fragmentos de código a través de ejecución y pruebas, en lugar de comparar la similitud del código generado con el código de referencia. Pass@k es actualmente utilizada en investigaciones relacionadas a tareas de generación de código y permitiría comparar resultados con nuevos modelos.

Aunque en este trabajo se entrenó el modelo Tranx para generar código Python, puede ser posible entrenar el modelo para generar directamente la estructura del GAST. Con esto se podría corregir algunos problemas encontrados con Python al pasarlo a otros lenguajes fuertemente tipados. Sin embargo, generar ejemplos con instrucciones en español y su equivalente GAST podría ser una tarea difícil debido a que no hay suficientes ejemplos que puedan ser recolectados de manera simple. También se podría mejorar los resultados de este trabajo agregando mejoras al GAST, como el soporte de instrucciones sin tener que estar en una clase o función, o iniciar contadores en un *For*, que fue uno de los errores más comunes encontrados al pasar Python a Java y C#. Tranx también podría ser entrenado para generar C# o Java, lo cual podría también corregir los problemas de Python al transformarlos con el GAST, ya que los tipos simplemente serían ignorados cuando el GAST sea transformado a un lenguaje de tipado dinámico.

El desarrollo de investigaciones futuras puede utilizar nuestro modelo para efectuar estudios que conduzcan a determinar su eficacia en la promoción del desarrollo del pensamiento computacional en estudiantes sin conocimiento en programación. La propuesta del método contempla efectuar la traducción de comandos de voz en español a cualquier lenguaje de programación soportado por el GAST. Por lo que la adición de otros lenguajes al GAST proporcionarán mayor robustez al modelo.

Bibliografía

[1] Besim Alibegović, Naser Prljača, Melanie Kimmel, and Matthias Schultalbers. Speech recognition system for a service robot - a performance evaluation. In *2020 16th International Conference on Control, Automation, Robotics and Vision (ICARCV)*, pages 1171–1176, 2020. URL https://doi.org/10.1109/ICARCV50220.2020.9305342.

[2] Audacity®. Free audio editor and recorder. Audacity® software is copyright © 1999-2021 Audacity Team. https://audacityteam.org/, 1999-2021. Accesado el 29 de enero, 2024.

[3] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate, 2016. URL https://doi.org/10.48550/arXiv.1409.0473.

[4] Ankur Bapna, Colin Cherry, Yu Zhang, Ye Jia, Melvin Johnson, Yong Cheng, Simran Khanuja, Jason Riesa, and Alexis Conneau. mslam: Massively multilingual joint pre-training for speech and text, 2022. URL https://doi.org/10.48550/arXiv.2202.01374.

[5] Daniel Bikel and Imed Zitouni. *Multilingual Natural Language Processing Applications: From Theory to Practice*. IBM Press, 1st edition, 2012.

[6] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. Two studies of opportunistic programming: Interleaving web foraging, learning, and writing code. New York, NY, USA, 2009. Association for Computing Machinery. URL https://doi.org/10.1145/1518701.1518944.

[7] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests, 2022. URL https://doi.org/10.48550/arXiv.2207.10397.

[8] Nancy Dejarnette. America's children: Providing early exposure to stem (science, technology, engineering and math) initiatives. *Education*, 133:77–84, 09 2012. URL http://bit.ly/499m6HV.

[9] César Iván Delgado Silva, Juan Pablo Sandoval Alcocer, and Wendoline Arteaga Sabja. Blockly Voice: un entorno de programación guiado por voz. *Acta Nova*, 9:115 – 129, 03 2019. URL https://bit.ly/42E0fFZ.

[10] Tenzin Doleck, Paul Bazelais, David Lemay, Anoop Saxena, and Ram Basnet. Algorithmic thinking, cooperativity, creativity, critical thinking, and problem solving: exploring the relationship between computational thinking skills and academic performance. *Journal of Computers in Education*, 4, 08 2017. URL https://doi.org/10.1007/s40692-017-0090-9.

[11] Li Dong and Mirella Lapata. Coarse-to-fine decoding for neural semantic parsing. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 731–742, Melbourne, Australia, July 2018. Association for Computational Linguistics. URL https://doi.org/10.18653/v1/P18-1068.

[12] Rotem Dror, Gili Baumer, Segev Shlomov, and Roi Reichart. The hitchhiker's guide to testing statistical significance in natural language processing. In Iryna Gurevych and Yusuke Miyao, editors, *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1383–1392, Melbourne, Australia, July 2018. Association for Computational Linguistics. URL https://doi.org/10.18653/v1/P18-1068.

[13] Real Academia Española. Homófono. URL https://bit.ly/3HVk8P4. Accesado el 29 de enero, 2024.

[14] Yao Fu, Hao Peng, Litu Ou, Ashish Sabharwal, and Tushar Khot. Specializing smaller language models towards multi-step reasoning, 2023. URL https://doi.org/10.48550/arXiv.2301.12726.

[15] Marcos Román González. Test de Pensamiento Computacional: principios de diseño, validación de contenido y análisis de ítems. In *Perspectivas y avances de la investigación: I Jornada de Doctorandos, Madrid, 8 de mayo de 2015, 2015, ISBN 978-84-362-6981-9, págs. 291-314*, pages 291–314. Universidad Nacional de Educación a Distancia – UNED, 2015.

[16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[17] Google. Cloud speech-to-text on-prem. https://bit.ly/489InUF, 2023. Accesado el 29 de enero, 2024.

[18] Google. Speech-to-text request construction. https://bit.ly/3wd8TPv, 2023. Accesado el 29 de enero, 2024.

[19] Dick Grune, Kees van Reeuwijk, Henri E. Bal, Ceriel J.H. Jacobs, and Koen Langendoen. *Modern Compiler Design*. Springer Publishing Company, Incorporated, 2nd edition, 2012. URL https://doi.org/10.1007/978-1-4614-4699-6.

[20] Philip J. Guo. *Older Adults Learning Computer Programming: Motivations, Frustrations, and Design Opportunities*, page 7070–7083. Association for Computing

Machinery, New York, NY, USA, 2017. URL https://doi.org/10.1145/3025453.3025945.

[21] Cheng-Yu Hsieh, Chun-Liang Li, Chih-Kuan Yeh, Hootan Nakhost, Yasuhisa Fujii, Alexander Ratner, Ranjay Krishna, Chen-Yu Lee, and Tomas Pfister. Distilling step-by-step! outperforming larger language models with less training data and smaller model sizes, 2023. URL https://doi.org/10.48550/arXiv.2305.02301.

[22] IBM. About Speech to Text — IBM Cloud API. https://ibm.co/3SVIvCH, 2023. Accesado el 29 de enero, 2024.

[23] IBM. Speech to Text — IBM Cloud API. https://ibm.co/48l1KtH, 2023. Accesado el 29 de enero, 2024.

[24] Ayad Imam and Ayman Alnsour. The use of natural language processing approach for converting pseudo code to c code. *Journal of Intelligent Systems*, 29:1388–1407, 04 2019. URL https://doi.org/10.1515/jisys-2018-0291.

[25] Ignacio Jara and Pedro Hepp. Enseñar Ciencias de la Computación: Creando oportunidades para los jóvenes de América Latina. Technical report, Microsoft America Latinoamerica, August 2016. URL https://bit.ly/3nEAG5h.

[26] Uday Kamath, John Liu, and James Whitaker. *Deep Learning for NLP and Speech Recognition*. Springer Publishing Company, Incorporated, 1st edition, 2019. URL https://doi.org/10.1007/978-3-030-14596-5.

[27] Sosuke Kobayashi. Contextual augmentation: Data augmentation by words with paradigmatic relations, 2018. URL https://doi.org/10.48550/arXiv.1805.06201.

[28] Philipp Koehn. Statistical significance tests for machine translation evaluation. In Dekang Lin and Dekai Wu, editors, *Proceedings of the 2004 Conference on Empirical Methods in Natural Language Processing*, pages 388–395, Barcelona, Spain, July 2004. Association for Computational Linguistics. URL https://doi.org/10.18653/v1/D18-1195.

[29] Igor Labutov, Bishan Yang, and Tom Mitchell. Learning to learn semantic parsers from natural language supervision. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1676–1690, Brussels, Belgium, October-November 2018. Association for Computational Linguistics. URL https://doi.org/10.18653/v1/D18-1195.

[30] M. Lagergren and M. Soneryd. Programming by voice efficiency in the reactive and imperative paradigm. 2021. URL https://bit.ly/3SScdZf. Accesado el 29 de enero, 2024.

[31] Triet H. M. Le, Hao Chen, and Muhammad Ali Babar. Deep learning for source code modeling and generation. *ACM Computing Surveys*, 53(3):1–38, Jul 2020. URL http://dx.doi.org/10.1145/3383458.

[32] Jason Leiton-Jimenez and Luis Barboza-Artavia. Método para convertir código fuente escrito en diversos lenguajes de programación a un lenguaje universal. Master's thesis, Escuela de Computación, oct 2021.

[33] Jason Leiton-Jimenez, Luis Barboza-Artavia, Antonio Gonzalez-Torres, Pablo Brenes-Jimenez, Steven Pacheco-Portuguez, Jose Navas-Su, Marco Hernández-Vasquez, Jennier Solano-Cordero, Franklin Hernandez-Castro, Ignacio Trejos-Zelaya, and Armando Arce-Orozco. Gast: A generic ast representation for language-independent source code analysis. *Enfoque UTE*, 14(4):9–18, Oct. 2023. URL https://doi.org/10.29019/enfoqueute.957.

[34] Jing Li, Aixin Sun, Jianglei Han, and Chenliang Li. A survey on deep learning for named entity recognition. *IEEE Transactions on Knowledge and Data Engineering*, 34(1):50–70, jan 2022. URL https://doi.org/10.1109%2Ftkde.2020.2981314.

[35] Wang Ling, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiský, Andrew Senior, Fumin Wang, and Phil Blunsom. Latent predictor networks for code generation, 2016. URL https://doi.org/10.48550/arXiv.1603.06744.

[36] S. Y. Lye and J. Koh. Review on teaching and learning of computational thinking through programming: What is next for k-12? *Comput. Hum. Behav.*, 41:51–61, 2014. URL https://doi.org/10.1016/j.chb.2014.09.012.

[37] Edward Ma. Nlp augmentation. https://github.com/makcedward/nlpaug, 2019. Accesado el 29 de enero, 2024.

[38] Rinor Maloku and Besart Pllana. Hypercode: Voice aided programming. *IFAC-PapersOnLine*, 49:263–268, 12 2016. URL https://doi.org/10.1016/j.ifacol.2016.11.073.

[39] Microsoft. How to recognize speech. https://bit.ly/3uxDOFO, 2023. Accesado el 29 de enero, 2024.

[40] Microsoft. Speech to text FAQ - Azure Cognitive Services. https://bit.ly/3SyXOQt, 2023. Accesado el 29 de enero, 2024.

[41] Microsoft. Speech to text overview - Speech service - Azure Cognitive. https://bit.ly/49dTpJC, 2023. Accesado el 29 de enero, 2024.

[42] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis, 2023. URL https://doi.org/10.48550/arXiv.2203.13474.

[43] A.R. M. Nizzad and Samantha Thelijjagoda. Designing of a voice-based programming ide for source code generation: A machine learning approach. In *2022 International Research Conference on Smart Computing and Systems Engineering (SCSE)*, volume 5, pages 14–21, 2022. URL https://doi.org/10.1109/SCSE56529.2022.9905095.

[44] Sheila Nurul Huda, Zainudin Zukhri, Teduh Dirgahayu, and Chanifah Ratnasari. Extending the translation from pseudocode to source code with reusability. *Journal of Theoretical and Applied Information Technology*, 97:1457–1466, 03 2019. URL https://bit.ly/49vvvsX.

[45] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. Learning to generate pseudo-code from source code using statistical machine translation. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 574–584, 2015. URL https://doi.org/10.1109/ASE.2015.36.

[46] OMG. About the Meta Object Facility Specification version 2.5.1 [online]. 2016 [visitado el 31 de agosto de 2024]. URL https://www.omg.org/spec/MOF. Accesado el 29 de enero, 2024.

[47] OpenAI. Gpt-4 technical report, 2023. URL https://doi.org/10.48550/arXiv.2303.08774.

[48] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA, July 2002. Association for Computational Linguistics. URL https://doi.org/10.3115/1073083.1073135.

[49] Eleni Partalidou, Eleftherios Spyromitros-Xioufis, Stavros Doropoulos, Stavros Vologiannidis, and Konstantinos I. Diamantaras. Design and implementation of an open source greek pos tagger and entity recognizer using spacy, 2019. URL https://doi.org/10.48550/arXiv.1912.10162.

[50] David Patterson, Joseph Gonzalez, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David So, Maud Texier, and Jeff Dean. Carbon emissions and large neural network training, 2021. URL https://doi.org/10.48550/arXiv.2104.10350.

[51] Daniel Povey, Arnab Ghoshal, Gilles Boulianne, Lukas Burget, Ondrej Glembek, Nagendra Goel, Mirko Hannemann, Petr Motlicek, Yanmin Qian, Petr Schwarz, Jan Silovsky, Georg Stemmer, and Karel Vesely. The kaldi speech recognition toolkit. In *IEEE 2011 Workshop on Automatic Speech Recognition and Understanding*. IEEE Signal Processing Society, December 2011. URL https://doi.org/10.1109/ASRU17718.2011. IEEE Catalog No.: CFP11SRW-USB.

[52] Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. Robust speech recognition via large-scale weak supervision, 2022. URL https://doi.org/10.48550/arXiv.2212.04356.

[53] Arya Roy. Recent trends in named entity recognition (ner), 2021. URL https://doi.org/10.48550/arXiv.2101.11420.

[54] Xavier Schmitt, Sylvain Kubler, Jérémy Robert, Mike Papadakis, and Yves LeTraon. A replicable comparison study of ner software: Stanfordnlp, nltk, opennlp, spacy, gate. In *2019 Sixth International Conference on Social Networks Analysis, Management and Security (SNAMS)*, pages 338–343, 2019. URL https://doi.org/10.1109/SNAMS.2019.8931850.

[55] JooYoung Seo and Sean McCurry. Latex is not easy: Creating accessible scientific documents with r markdown. *Journal on Technology and Persons with Disabilities*, 7:157–171, 2019. URL http://doi.org/10211.3/210398.

[56] Noah Shinn, Federico Cassano, Beck Labash, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning, 2023. URL https://doi.org/10.48550/arXiv.2303.11366.

[57] Douglas Thain. *Introduction to Compilers and Language Design*. Independently published (June 18, 2020), 2nd edition, 2020. URL http://compilerbook.org.

[58] Zhiruo Wang, Grace Cuenca, Shuyan Zhou, Frank F. Xu, and Graham Neubig. Mconala: A benchmark for code generation from multiple natural languages, 2023. URL https://doi.org/10.48550/arXiv.2203.08388.

[59] Jason Wei and Kai Zou. EDA: Easy data augmentation techniques for boosting performance on text classification tasks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 6383–6389, Hong Kong, China, November 2019. Association for Computational Linguistics. URL https://doi.org/10.18653/v1/D19-1670.

[60] Qizhe Xie, Zihang Dai, Eduard Hovy, Minh-Thang Luong, and Quoc V. Le. Unsupervised data augmentation for consistency training, 2020. URL https://doi.org/10.48550/arXiv.1904.12848.

[61] Julián Pérez Porto y Ana Gardey. Fonema - qué es, definición y concepto. URL https://definicion.de/fonema/. Actualizado el 29 de septiembre de 2021, accesado el 29 de enero, 2024.

[62] Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation, 2017. URL https://doi.org/10.48550/arXiv.1704.01696.

[63] Pengcheng Yin and Graham Neubig. TRANX: A transition-based neural abstract syntax parser for semantic parsing and code generation. *CoRR*, abs/1810.02720, 2018. URL https://doi.org/10.48550/arXiv.1810.02720.

[64] Dong Yu and Li Deng. *Automatic Speech Recognition: A Deep Learning Approach*. Springer Publishing Company, Incorporated, 2014. URL https://doi.org/10.1007/978-1-4471-5779-3.

[65] Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. Codereval: A benchmark of pragmatic code generation with generative pre-trained models, 2023. URL https://doi.org/10.1145/3597503.3623322.

Apéndice A

Algoritmo de preprocesado en Python

```
data = pd.read_json('source.json', encoding= 'unicode_escape')

result = []
for index, row in data.iterrows():
  new_intent = strip_accents(row['rewritten_intent'])
  list_of_variants = create_list_of_variants(new_intent)
  result_list = create_all_possible_senteces(list_of_variants, 0, "", [])
  new_intent = get_best_intent_from_variant_sentences(new_intent,
  ↪  result_list).row
  split_new_intent = new_intent.split()
  intent_changed = False
  for i in range(len(split_new_intent)):
    if split_new_intent[i] == "menos" and i > 0:
      if not ("`" in split_new_intent[i-1] or
      ↪  split_new_intent[i-1].isnumeric() or "menos" ==
      ↪  split_new_intent[i-1]):
        split_new_intent[i] = "-"
        intent_changed = True
  if intent_changed:
    new_intent = " ".join(split_new_intent)
    new_intent = new_intent.replace(" - ", " -")
  sample = {'Id': row['Id'], 'lie_syntax': row['lie_syntax'],
  ↪  'rewritten_intent': new_intent, 'snippet': row['snippet']}
  result.append(sample)
  json_string = json.dumps(result, indent=4)
  with open("audio_to_text_result" + ".json", "w", encoding='utf-8') as
  ↪  write_file:
      write_file.write(json_string)
```

Apéndice B

Algoritmo para crear lista de variantes en Python

```
splited_text = text.split()
list_of_variants = []
# Add original sentence as the first possible variant
for word in splited_text:
  list_of_variants.append([word])
# Search for letters that are fonemas on every word an create the
↪  variants
for i in range(len(splited_text)):
  word = splited_text[i]
  for fonema in rules:
    if fonema.letters in word:
      for variant in fonema.variants:
        new_word = word.replace(fonema.letters, variant)
        list_of_variants[i].append(new_word)
return list_of_variants
```

Apéndice C

Algoritmo para crear todas las oraciones posibles en Python

```
1  if index >= len(list_variants):
2    result_list.append(string)
3    return
4  for i in range(len(list_variants[index])):
5    create_all_possible_senteces(list_variants, index+1, string + " " +
     ↪  list_variants[index][i],result_list)
6  if index == 0:
7    return result_list
```

Apéndice D

Algoritmo para obtener la mejor oración en Python

```
intent_with_more_score = BestIntent(original_sentence, 0, 0)
for row in variant_sentences:
  new_intent = row

  new_intent = new_intent.replace(".", " ").replace("+",
  ↪  "mas").replace("*", "multiplicar")
  new_intent = new_intent.strip()
  new_intent = new_intent.lower()
  ents_finish = False
  while not ents_finish:
    doc = trained_nlp(new_intent)
    if len(doc.ents) == 0:
      break
    for ent in doc.ents:
      second_quote_index = ent.end_char
      # Remove comma if there is a variable before
      if new_intent[second_quote_index:second_quote_index+2] == ", ":
        new_intent = new_intent[:second_quote_index] +
        ↪  new_intent[second_quote_index+2:]
        ents_finish = False
        break
      # Remove blank space if there is a number after a variable
      elif new_intent[second_quote_index+1:second_quote_index+2]
      ↪  .isnumeric():
        ents_finish = False
        new_intent = new_intent[:second_quote_index] +
        ↪  new_intent[second_quote_index+1:]
        break
```

```
        ents_finish = True
    new_intent = new_intent.replace(",", "")
    doc = trained_nlp(new_intent)
    char_offset = 0
    result_without_quotes = new_intent
    for ent in doc.ents:
      first_quote_index = ent.start_char + char_offset
      second_quote_index = ent.end_char + char_offset
      # Add single quotes to variables
      new_intent = new_intent[:first_quote_index]  + "`" +
      ↪  new_intent[first_quote_index:second_quote_index] + "`" +
      ↪  new_intent[second_quote_index:]
      char_offset += 2
    if len(doc.ents) > intent_with_more_score.vars:
      # Add sentece as best intent if it has more variables that the
      ↪  previous ones
      intent_with_more_score.row = new_intent
      intent_with_more_score.vars = len(doc.ents)
      intent_with_more_score.score =
      ↪  sentence_bleu(original_sentence,result_without_quotes)
    elif len(doc.ents) == intent_with_more_score.vars and
    ↪  sentence_bleu(original_sentence,result_without_quotes) >
    ↪  intent_with_more_score.score:
      # Add sentece as best intent if it has more variables that the
      ↪  previous ones and has better bleu score
      intent_with_more_score.row = new_intent
      intent_with_more_score.vars = len(doc.ents)
      intent_with_more_score.score =
      ↪  sentence_bleu(original_sentence,result_without_quotes)
  return intent_with_more_score
```