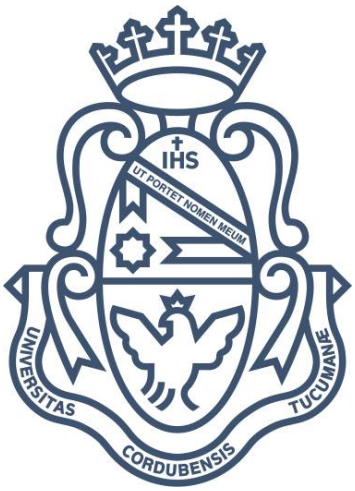
Universidad Nacional de Córdoba

Facultad de Matemática, Astronomía, Física y Computación



Aprendizaje no Supervisado para Cómputo de Similitud en Programas con Errores

Trabajo especial para la obtención del título de

Licenciado en Ciencias de la Computación

Autor: José Doffo

Directores: Carlos Areces & Franco Bulgarelli

Córdoba, Argentina 2023

Resumen

El problema que se investigará en esta tesis de licenciatura es el de predecir errores tanto sintácticos como semánticos en programas escritos por estudiantes. Más precisamente lo que esperamos es poder definir una medida de distancia entre programas con errores, de modo que programas que contengan errores similares sean considerados próximos. Para abordar tal problema se propone utilizar técnicas de inteligencia artificial del área de procesamiento del lenguaje natural que se utilizan, usualmente, para determinar cuando dos textos tienen un significado similar.

El objetivo principal de nuestro proyecto es producir una herramienta valiosa que asista al aprendizaje de un primer lenguaje de programación. Para ello utilizaremos la medida de distancia entre programas que definiremos para emparejar programas similares y así reutilizar toda la información disponible sobre ellos. De este modo, podemos mejorar la experiencia de los estudiantes con Mumuki.

Uno de los desafíos del enfoque propuesto, es que estas técnicas han sido desarrolladas para usarse sobre texto natural, es decir, texto producto de un lenguaje humano y no sobre un lenguaje formal como lo es un lenguaje de programación. Claramente, existen diferencias importantes entre la estructura de un lenguaje de programación, y el lenguaje humano (e.g., un lenguaje formal tiene una gramática mucho mas restringida que la de un lenguaje natural). Uno de los objetivos secundarios del proyecto es investigar si los métodos de procesamiento del lenguaje natural que utilizaremos pueden adaptarse a este tipo de lenguaje.

Otro desafío que merece mención es que el código producido por un conjunto de estudiantes, para un mismo problema puntual, puede variar de muchas maneras. Por ejemplo, programas equivalentes pueden variar en la selección de los nombres de las variables utilizadas.

En este trabajo, nos enfocaremos en las respuestas dadas por estudiantes a ejercicios de programación en un entorno virtual. En estos casos, es usual observar que las respuestas correctas a un ejercicio dado tienen menor variabilidad que los intentos fallidos, que constituyen la mayoría. Es decir, mientras que se puede observar que las respuestas correctas "convergen" hacia un conjunto pequeño de soluciones posibles (equivalentes), las posibilidades de error son muy amplias, y van desde pequeños errores sintácticos a errores conceptuales serios.

La variabilidad presente en los programas escritos por estudiantes reducen la efectividad de las técnicas del procesamiento del lenguaje natural. Por ello es necesario explorar estrategias para obtener representaciones simplificadas de los programas, de modo tal que estas nuevas representaciones reduzcan la variabilidad, preservando la semántica original y los errores existentes en los programas.

En este trabajo se analizaron distintas técnicas de normalización de programas para reducir la variabilidad presente en estos. Luego verificamos experimentalmente que reducir la variabilidad de los programas, efectivamente favorece el entrenamiento de modelos de inteligencia artificial.

Finalmente entrenamos un modelo que asocia programas a vectores y comprobamos experimentalmente que la distancias entre estos vectores esta directamente relacionada con la similaridad de los programas que representan.

Summary

The problem that will be investigated in this degree thesis is that of predicting both syntactic and semantic errors in programs written by students. More precisely, what we expect is to be able to define a distance measure between programs with errors, so that programs containing similar errors are considered close. To address this problem, it is proposed to use artificial intelligence techniques from the area of natural language processing that are usually used to determine when two texts have a similar meaning.

The main objective of our project is to produce a valuable tool that assists in learning a first programming language. To do this, we will use the measure of distance between programs that we will define to match similar programs and thus reuse all the information available about them. In this way, we can improve the student's experience with Mumuki.

One of the challenges of the proposed approach is that these techniques have been developed to be used on natural text, that is, text that is the product of a human language and not on a formal language such as a programming language. Clearly, there are important differences between the structure of a programming language, and human language (e.g., a formal language has a much more restricted grammar than a natural language). One of the secondary objectives of the project is to investigate if the natural language processing methods that we will use can be adapted to this type of language.

Another challenge that deserves mention is that the code produced by a group of students, for the same specific problem, can vary in many ways. For example, equivalent programs may vary in the choice of variable names used.

In this work, we will focus on the answers given by students to programming exercises in a virtual environment. In these cases, it is usual to observe that the correct answers to a given exercise have less variability than the failed attempts, which constitute the majority. That is, while it can be observed that the correct answers "converge" towards a small set of possible (equivalent) solutions, the possibilities of error are very wide, ranging from small syntactic errors to serious conceptual errors.

The variability present in student-written programs reduces the effectiveness of natural language processing techniques. For this reason, it is necessary to explore strategies to obtain simplified representations of the programs, in such a way that these new representations reduce the variability, preserving the original semantics and the existing errors in the programs.

In this work, different program normalization techniques were analyzed to reduce the variability of the programs. Then we verified experimentally that reducing the variability of the programs effectively favors the training of artificial intelligence models.

Finally, we train a model that associates programs to vectors and we verify experimentally that the distances between these vectors is directly related to the similarity of the programs they represent.

Índice general

Capítulo 1

Introducción

En este trabajo exploramos técnicas de aprendizaje no supervisado, con el fin de entrenar un modelo de inteligencia artificial que sea capaz de detectar errores en programas escritos por estudiantes. Para esta tarea IKUMI SRL nos provee ejemplos de programas escritos por los estudiantes de su plataforma Mumuki.

1.1. Contexto de la tesis

Esta tesis de licenciatura se realiza en el marco de una colaboración entre investigadores de la Universidad Nacional de Córdoba y la empresa IKUMI SRL. Esta empresa se dedica a la formación online de recursos humanos de Software y Servicios Informáticos (SSI) y al desarrollo de recursos didácticos en espacios virtuales para este objetivo.

Particularmente, para los intereses de este trabajo, IKUMI SRL desarrolló una plataforma online llamada Mumuki la cual es utilizada en academias digitales y en programas de gobierno para formación profesional.

Mumuki propone una forma amigable de aprender distintos conceptos de las ciencias de la computación, tales como: fundamentos de programación imperativa, paradigmas de programación, testing, redes e internet, análisis de datos, entre otros. Para esto Mumuki ofrece diversos cursos en distintos lenguajes de programación como Python, Haskell, Prolog, etc. Estos cursos consisten de material didáctico para abordar los contenidos del curso y múltiples ejercicios interactivos, ordenados progresivamente y agrupados por concepto (figura 1.1).

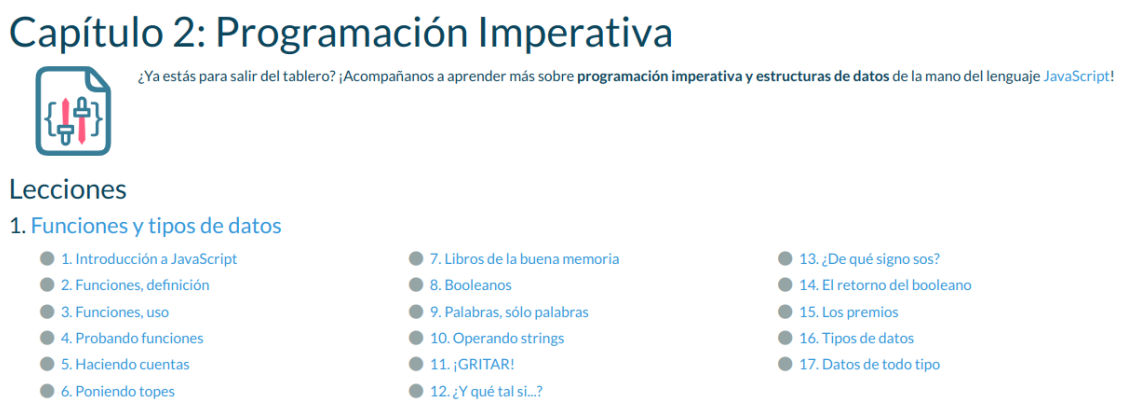


Figura 1.1: Screenshot de Mumuki: Una sección del curso "programación imperativa en JavaScript".

Cada ejercicio cuenta con una interfaz interactiva la cual contiene el respectivo enunciado del ejercicio y un recuadro de texto donde el estudiante deberá escribir el programa que satisfaga tal enunciado (figura 1.2).

Figura 1.2: Screenshot de Mumuki: Interfaz correspondiente al ejercicio 3 de la sección "Práctica Funciones y Tipos de Datos". En la sección izquierda de la interfaz se puede leer el enunciado correspondiente al ejercicio y la signatura de las funciones que el estudiante debe definir. En la sección derecha de la interfaz se encuentra un recuadro donde el estudiante deberá proporcionar la solución al problema, y debajo de este recuadro está el botón "Enviar" por medio del cual, el estudiante podrá solicitar una retroalimentación sobre la solución escrita en el recuadro de código

Cuando el estudiante termine de escribir la solución, puede solicitar una devolución o feedback del sistema. Para producirla Mumuki primero comprueba que la sintaxis de la solución es correcta, luego de eso verifica la correctitud de la misma evaluando los casos de prueba del respectivo ejercicio. Si la solución fracasa en algún caso de prueba o existen errores sintácticos en ella, la devolución exhibe estos errores (figura 1.3).

Figura 1.3: Screenshot de Mumuki: Ejemplo de solución para el enunciado del ejercicio de la figura 2. A la izquierda se encuentra la devolución proporcionada por Mumuki para el código que se encuentra a la derecha. Como se puede observar en la evaluación, la definición de la función "perimetroTriangulo" satisface los casos de prueba de Mumuki, pero la definición de la función "areaTriangulo" no satisface algunos casos de prueba. Al clickear en "Ver detalles" nos despliega una descripción: el valor a izquierda es el resultado de la definición propuesta y a la derecha el resultado esperado.

Mumuki además de comprobar la correctitud de la solución también evalúa los estándares de programación del lenguaje correspondiente y las buenas prácticas de desarrollo. Así en caso de que la solución supere los casos de prueba pero esta no satisfaga los estándares de programación o las buenas prácticas, la devolución indicará qué detalles pueden mejorarse en la solución (figura 1.4).

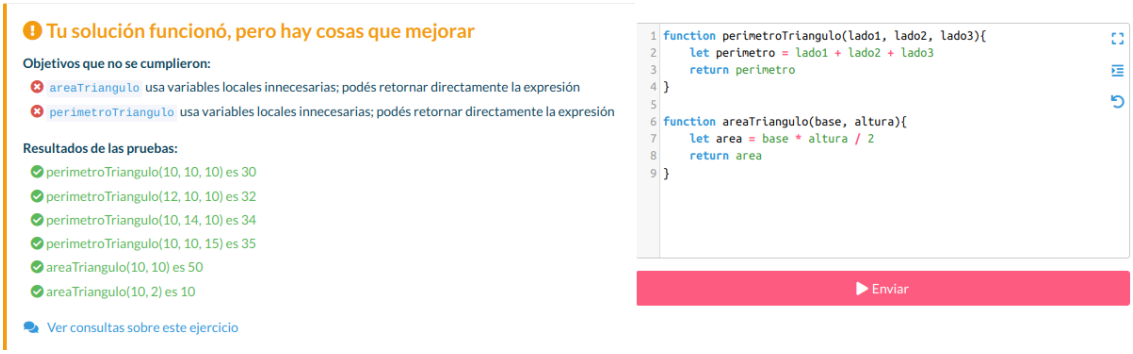


Figura 1.4: Screenshot de Mumuki: Ejemplo de solución con detalles a mejorar para el ejercicio de la figura 1.2. A la izquierda vemos la devolución que nos indica que se definieron variables locales innecesarias y a la derecha el correspondiente código asociado a la solución

Si la solución es correcta, Mumuki simplemente notifica al estudiante (figura. 1.5) y marcará al ejercicio como correcto en el curso.

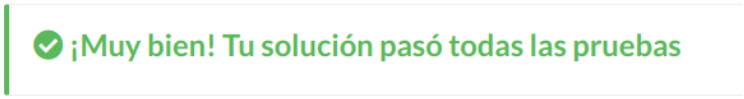


Figura 1.5: Screenshot de Mumuki: Retroalimentación que nos provee la plataforma en caso de que la solución propuesta sea correcta.

En el caso de que Mumuki produzca una retroalimentación negativa entonces el estudiante, si lo considera necesario, puede iniciar una conversación asincrónica con un profesor de Mumuki con el fin de evacuar dudas (figura 1.6). Por otro lado, cuando un profesor atiende esta conversación, recibe el último intento de solución realizado por el estudiante junto a la duda.

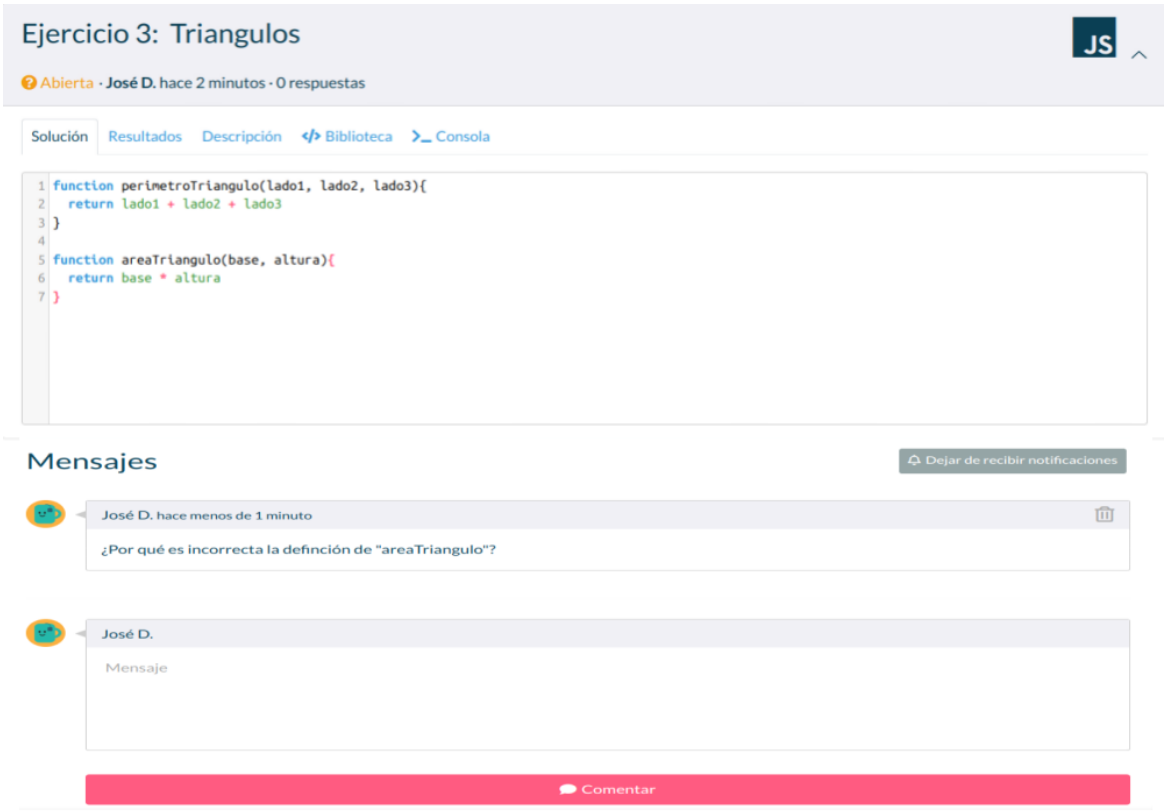


Figura 1.6: Screenshot de Mumuki: Interfaz para realizar una consulta sobre el ejercicio "Triángulos". En la sección superior se encuentra la solución incorrecta del usuario (figura 1.3) y en la sección inferior un chat asincrónico con un profesor de Mumuki.

1.2. Objetivo de la tesis

El problema que se pretende investigar en esta tesis de licenciatura es el de producir una medida de distancias que representen la similaridad entre programas, la cual sea mínima para un par de programas equivalentes.

Bajo el contexto de Mumuki, el éxito en este objetivo resulta en las herramientas necesarias para poder atender automáticamente las dudas de los estudiantes, ya que al identificar los programas más similares a uno dado, podemos buscar las respuestas a dudas que se realizaron sobre estos y así proveer automáticamente una respuesta que sea efectiva para corregir los errores del programa dado. De esta manera se favorece la interacción entre los estudiantes y Mumuki y, por otro lado, también se reducen los tiempos de espera en las comunicaciones asíncronas

1.3. Estrategia preliminar de abordaje

Para predecir las distancias entre un par de instancias de código se propone utilizar métodos de inteligencia artificial, específicamente, del procesamiento del lenguaje natural (PLN) para entrenar un modelo que sea capaz de realizar estas predicciones.

El procesamiento de lenguaje natural es un área de la ciencia de la computación y la inteligencia artificial, la cual estudia las interacciones entre las máquinas y el lenguaje humano. El objetivo del PLN es que las computadoras sean capaces de "entender" y procesar grandes cantidades de documentos escritos en lenguaje humano. Actualmente existen muchos resultados impresionantes que se lograron gracias al PLN, como por ejemplo la generación de texto humano, la traducción automática de un lenguaje a otro, asistentes virtuales, resúmenes de grandes bloques de texto entre otros.

La técnica de esta disciplina que nos interesa para este trabajo es el "word embedding", la cual es fundamental para esta área ya que permite representar matemáticamente (por medio de vectores) a las palabras del lenguaje. En PLN un word embedding es una representación de una palabra, típicamente esta representación es un vector. Los vectores de un word embedding constituyen un espacio vectorial que codifican propiedades semánticas del texto, de modo que las palabras con significado similar se representen con vectores cercanos. En el 2013 se desarrollo word2vec [17] que es un método eficiente para obtener word embeddings y desde ese entonces esta técnica se utiliza ampliamente en la investigación y en la industria para tareas relacionadas al PLN.

Para obtener un word embedding se entrena una red neuronal para que aprenda una función que mapea una pieza de texto con un vector de un espacio vectorial de muchas dimensiones, de forma tal que este espacio contenga propiedades geométricas que capturen las propiedades semánticas del lenguaje. Para que las redes neuronales sean capaces de codificar la semántica de los textos, estas se apoyan en la hipótesis distribucional [22], la cual sugiere que cuanto mas se parezca el significado de dos palabras, estas tienden a ocurrir mas en contextos lingüísticos similares.

Por las propiedades geométricas de los word embeddings, se espera que en el espacio vectorial que resulte de las instancias de código, los vectores de instancias similares sean próximas. De modo que las instancias que comparten un tipo particular de error sean cercanas y así poder efectivamente predecir la similaridad entre programas.

Si bien las instancias de código, en la práctica, para problemas en general pueden ser de distinto tamaño, los errores pueden producirse a cualquier escala, incluso en una simple línea de código. Bajo este hecho es razonable abordar el problema sobre instancias de código breve, por lo que es apropiado utilizar las instancias de código para ejercicios de Mumuki, ya que las soluciones de estos ejercicios son inherentemente breves (usualmente estos ejercicios son puntuales y piden definir un par de funciones simples). Sin embargo como las técnicas de procesamiento del lenguaje natural se pensaron originalmente para procesar lenguaje humano y un lenguaje de programación es un lenguaje formal, el cual posee reglas gramaticales estrictas y un discurso mas restringido, debemos explorar herramientas adecuadas para obtener word embeddings de instancias de código. En el capítulo 3 analizaremos en detalle dos herramientas que nos pueden ser útiles para producir los word embeddings: Code2Vec [3] y FastText [6].

1.4. Estructura de la tesis

Capitulo 2 "**Datos disponibles**"**:** En este capitulo haremos un análisis y una descripción detallada de los datos que nos ofrece IKUMI SRL para lograr nuestros objetivos

Capitulo 3 "**Desafíos y herramientas consideradas**"**:** Como utilizaremos técnicas del procesamiento del lenguaje natural y nuestros datos están compuestos principalmente por programas, surgen diversos desafíos a la hora de aprovechar tales técnicas. En este capitulo describiremos los desafíos que se presentan en nuestro enfoque de trabajo y también analizaremos herramientas que nos serán útiles para superar tales desafíos.

Capitulo 4 "**Procesamiento y normalización de los datos**"**:** Los resultados que puede obtener un modelo de machine learning dependen de la calidad de los datos de entrenamiento, en este capitulo nos enfocaremos en procesar los datos y normalizar los programas disponibles con el fin de mejorar la calidad de nuestro modelo.

Capitulo 5 "**Efectos del procesamiento y normalización de datos**"**:** En este capitulo comprobamos mediante experimentos que el procesamiento del los datos realizado es adecuado y contribuye a la obtención de mejores modelos respecto a utilizar los datos crudos.

Capitulo 6 "**Combinación de reportes y programas**"**:** Entre los datos, además de programas, tenemos diversos reportes que agregan información relevante sobre la semántica de los programas. En este capitulo compararemos dos enfoques distintos para agregar estos reportes a nuestro modelo.

Capitulo 7 "**Conclusiones**"**:** En este capítulo resumimos el trabajo realizado y analizamos posibles trabajos futuros.

Capítulo 2

Datos disponibles

La empresa IKUMI SRL nos provee un conjunto de datos, extraídos de los espacios de consulta de su plataforma Mumuki del periodo 2021-2022 del curso de "programación imperativa en JavaScript". Estos datos serán la base de nuestro enfoque basado en aprendizaje automático. En las siguientes secciones presentaremos que estructura tienen, y haremos un análisis preliminar de las posibilidades de uso en nuestra estrategia de solución al problema que queremos resolver. Estos datos están distribuidos en cuatro tablas llamadas:

- **item**
- **discussions**
- **participantes**
- **messages**

2.1. Descripción de los datos

La tabla "ítem" contiene información relacionada a los ejercicios de la plataforma. Esta tabla cuenta con 122 filas. Cada fila corresponde a un ejercicio del curso de "programación imperativa en JavaScript". A continuación describimos el contenido de las columnas:

- **id**: Identificador del ejercicio
- **name**: Nombre del ejercicio, Mumuki usa este nombre en su plataforma cuando lista los ejercicios de un curso.
- **descripción**: Enunciado asociado al ejercicio
- **test**: Test suite para verificar las soluciones del ejercicio. La test suite está definida por medio del framework Jasmine [16] (un framework de testing para JavaScript).
- **created_at**: Fecha y hora de creación del ejercicio.
- **updated_at**: Fecha de la última actualización.
- **language_id**: Identificador del lenguaje de programación asociado al enunciado.

La tabla "discussions" contiene información relacionada a las consultas realizadas por los estudiantes. Es importante mencionar que la tabla, en la columna que contiene la solución propuesta por el estudiante (i.e., código fuente), solo hace referencia a la última solución enviada y no al historial completo de intentos previos. La tabla contiene 31383 filas y el contenido de sus columnas es el siguiente:

- **id**: Identificador de la consulta.
- **description**: La pregunta del estudiante.
- **item_id**: Identificador del ejercicio asociado a la consulta (id correspondiente a la tabla "items").
- **created_at**: Fecha y hora de realización de la consulta.
- **solution**: Solución propuesta por el estudiante (código fuente).
- **submition_status**: Este campo describe el resultado de la retroalimentación generada por Mumuki para la solución propuesta por el estudiante, este campo tiene cuatro valores posibles "errored", "failed", "passed" y "passed_with_warnings". Cada uno de estos valores describen las siguientes posibilidades:
  - errored: Indica que el código tiene un error en el momento de compilación, indica un error sintáctico en el código enviado.
  - failed: Indica que el código no tiene errores de compilación, pero no superó todos los casos de prueba de la test suite correspondiente.
  - passed_with_warnings: Indica que el código que superó todos los casos de prueba, pero no satisface los estándares del lenguaje o las buenas prácticas, como por ejemplo: definir variables y no usarlas, comparar un booleano en una guarda (e.g. if boolean_var == True {...}) o no respetar las convenciones de estilo del lenguaje de programación.
  - passed: Indica que el código supero todos los casos de prueba y satisface los estándares del lenguaje.
- **result**: Reporte de los errores en tiempo de compilación (este campo tiene información sólo cuando el código no compila).
- **expectation_results**: Resultado del motor de evaluación de expectativas (este campo tiene información sólo cuando el código compila). Este tipo de resultado exhibe propiedades estructurales del programa asociado, como por ejemplo si en el hay estructuras condicionales o llamados a funciones particulares en determinados ejercicios y también exhibe si el programa satisface las convenciones del lenguaje y las buenas practicas de programación.
- **test_results**: Resultados de las pruebas de la test suite (este campo tiene información solo cuando el código compila).
- **initiator_uid**: Id del estudiante (id correspondiente a la tabla participantes) que realizó la consulta.

La tabla "participantes" contiene información básica de los estudiantes de Mumuki, esta tabla contiene 11806 filas. A continuación describimos el contenido de sus columnas:

- **created_at**: Fecha y hora de registro a la plataforma.
- **last_submission_date**: Fecha y hora de la última solución enviada por el estudiante.
- **uid**: Identificador del estudiante.
- **gender**: Género del estudiante
- **birth_year**: Fecha de nacimiento del estudiante.

La tabla "messages" contiene comunicaciones asíncronas, asociadas a las consultas realizadas por los estudiantes. Con el fin de preservar la identidad de los participantes y sus datos personales, el equipo de Mumuki realizó tareas de anonimización de los datos.

Los nombres de pila y apellidos se reemplazaron por otros nombres y apellidos comunes, reales o ficticios. Los reemplazos fueron realizados consistentemente manteniendo la trazabilidad del individuo, (i.e., Juan Perez fue reemplazado consistentemente por Hector Gonzales), su género (i.e., Ana fue reemplazada por María) y evitando asociaciones accidentales basadas en su nombre o parte del mismo (i.e., si el nombre de un participante llamado Juan fue reemplazado por Hector, no necesariamente otro participante también llamado Juan fue reemplazado por Hector). Además, se eliminaron referencias a las localidades, enlaces sensibles, números de teléfono, correos electrónicos y nombres de usuarios de redes sociales. Finalmente, el equipo de Mumuki también realizaron una revisión manual cruzada de los datos anonimizados.

Esta tabla contiene 41248 filas, a continuación describimos el contenido de sus columnas:

- **id**: Identificador del mensaje.
- **content**: Contenido del mensaje.
- **created_at**: Fecha y hora de envío del mensaje.
- **discussion_id**: Id de la consulta asociada (id correspondiente a la tabla "discussions")
- **approved**: Este campo es un booleano que indica si el mensaje proviene del equipo de mentorías de Mumuki.
- **not_actually_a_question**: Indica si el mensaje fue marcado manualmente como "no-pregunta".
- **sender_uid**: Id del emisor del mensaje.

2.2. Análisis de los datos

Si bien IKUMI SRL nos provee también datos vinculados a la interacción de los estudiantes con la plataforma Mumuki, por los intereses de este trabajo nos enfocaremos principalmente en los datos relacionados al código producido por los estudiantes.

Recordemos que el objetivo principal es entrenar un clasificador que sea capaz de predecir una medida de similaridad entre programas, por lo que es razonable considerar la información relacionada a las soluciones propuestas por los estudiantes, el

tipo de feedback que Mumuki asocia a cada solución (e.g., "failed") y los respectivos enunciados de los ejercicios a los que corresponden. Estos datos nos permitirán realizar experimentos más específicos, considerando instancias de códigos que pertenezcan a una misma categoría (i.e., que provengan del mismo ejercicio o que tengan el mismo tipo de feedback).

Existen también otros datos disponibles que pueden considerarse relevantes para este trabajo, como por ejemplo los resultados del compilador, del motor de evaluación de expectativas y los resultados de cada prueba unitaria del test suite asociado a cada ejercicio, ya que proveen información sobre la estructura y semántica de los programas.

No todas las instancias de código contienen información relevante en cada uno de estos campos. Por ejemplo, si quisiéramos considerar los resultados del compilador, esta información sólo existe en caso de que el código no se pueda compilar (solo el 20 % de las instancias de código entran en esta categoría). En caso de querer considerar los resultados del motor de evaluación de expectativas o los resultados de la test suite, tenemos la misma situación: solo las instancias de código que pueden compilarse contienen esta información (el 80 % de las instancias). Por lo ya mencionado, debemos procesar estos datos apropiadamente para obtener un conjunto de datos útil.

Para definir nuestro modelo clasificador, vamos a considerar los datos asociados al código y todo lo relacionado a los reportes de código (i.e., los resultados del compilador, de la test suite y del motor de evaluación de expectativas). Al considerar el código y sus reportes asociados, nuestros modelos pueden aprender una métrica de similaridad que esté influenciada por dos características de los programas, sintaxis y semántica. La similaridad sintáctica puede obtenerse principalmente del código y de los reportes del motor de evaluación de expectativas, mientras que la similaridad semántica se obtiene principalmente de los reportes de la test suite y del compilador.

Las instancias de código disponibles, 31383 en total, todas pertenecen a ejercicios del curso de "Programación imperativa en JavaScript" y estas están distribuidas en 122 ejercicios distintos, como se ve en la figura 2.1.

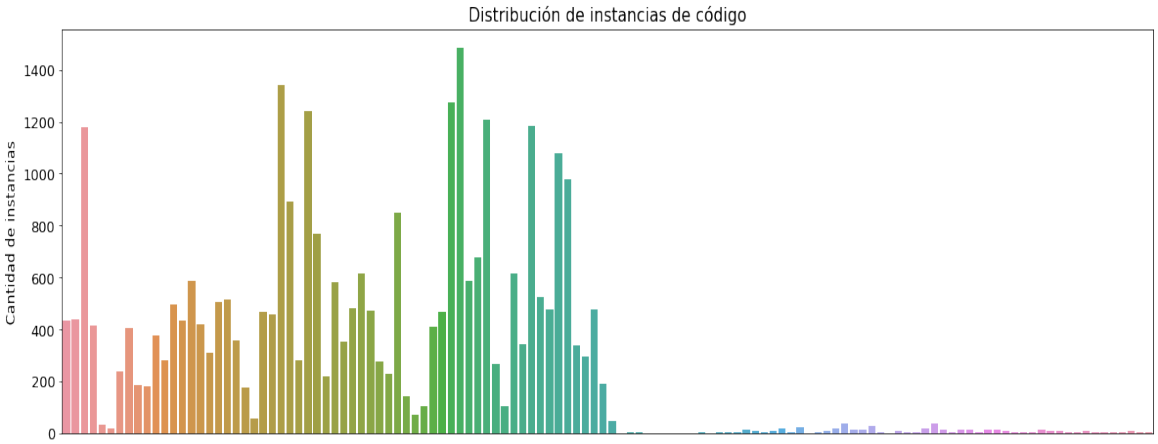


Figura 2.1: Distribución de las instancias de código por ejercicio

Como podemos ver en la figura anterior hay muchos ejercicios con pocas instancias de código. La mayoría de estos ejercicios tienen enunciados sencillos y solicitan funciones bastante intuitivas y directas, como por ejemplo definir una función que calcule la mitad de un número o ver si una determinada variable satisface una condición simple. Por otro lado los ejercicios que tienen más instancias de código suelen requerir de más intentos. Podemos considerarlos como más "complejos" porque por lo general requieren de considerar varias condiciones, por ejemplo el ejercicio "¡Quiero retruco!" tiene 1209 instancias de código y este ejercicio solicita definir una función que calcule el envido de una mano del juego Truco.

Todas las instancias de código a su vez están agrupadas en cuatro categorías (determinadas por la columna "submission_status"): "errored", "failed", "passed" y "passed_with_warnings", las cuales ya fueron descritas anteriormente. La figura 2.2 muestra la distribución de las instancias de código, según la categoría a la que pertenecen.

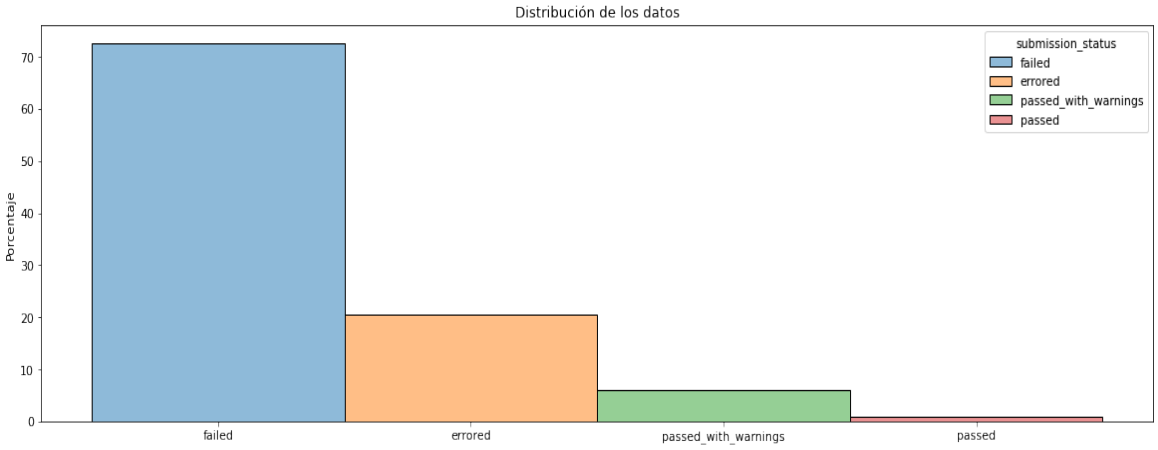


Figura 2.2: Distribución de las instancias de código por "submission_status"

Como podemos observar la mayoría de los programas están catalogados como "failed" y "errored", esto se debe principalmente a que los estudiantes normalmente resuelven progresivamente los ejercicios (interactuando con Mumuki y sus docentes en cada iteración), hasta que eventualmente obtienen una solución correcta. Los programas disponibles resultan interesantes para los propósitos de este trabajo, ya que el 70 % de estos corresponden a la clase "failed", donde este tipo de programas no satisfacen la test suite del ejercicio asociado.

Otro aspecto importante para analizar es: qué tan extensas son las instancias de código en general. Inicialmente se supuso que por las características de los enunciados de los ejercicios de Mumuki, las instancias de código asociadas a una solución posible son inherentemente breves. Para respaldar esta suposición, veremos la distribución de las instancias de código por cantidad de tokens. En este trabajo consideraremos como token a todas las cadenas de caracteres separadas por al menos un espacio en blanco en un texto. Por ejemplo la frase:

```
function HelloWorld () \n { console.log("Hello world") \n }
```

Consta de 9 tokens: *function*, *HelloWorld*, () , \n, {, *console.log("Hello*, *world")*, \n y }.

En la figura 2.3 se puede observar la distribución de las instancias de código en función de la cantidad de tokens que poseen.

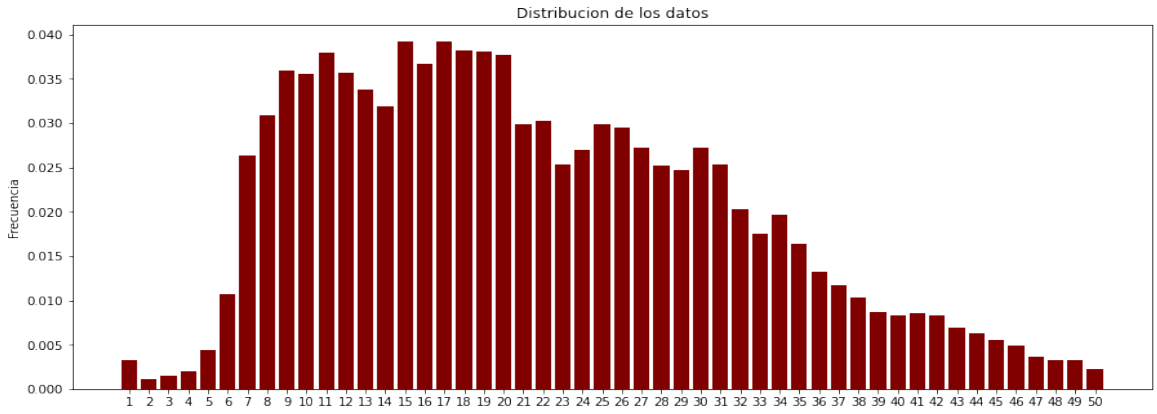


Figura 2.3: Distribución de la cantidad de tokens de instancias de código de a lo sumo 50 tokens (se acotaron los resultados para mejorar la visibilidad del gráfico de barras). Este gráfico se construyó con 29945 muestras de 31383

En promedio, las instancias de código tienen 24 tokens y por nuestra definición de token, algunos tokens resultan ser breves (e.g., "", "", "+", "*"). Por lo que, para los ejercicios de Mumuki, las instancias de código disponibles son de hecho breves. La instancia más corta de código tiene 1 token mientras que la más larga tiene 332 tokens.

La cantidad total de tokens distintos presentes en las 31383 instancias de código es de 45000. Pero si consideramos además del código, los reportes generados por el compilador, el motor de evaluación de expectativas y los resultados de la testsuite, la cantidad total de tokens distintos aumenta a 48886 y la cantidad promedio de tokens por instancia aumenta a 44.

Conclusiones: En este capítulo confirmamos nuestra hipótesis de que las soluciones para ejercicios de Mumuki son inherentemente breves, esto en principio favorece a nuestro enfoque de usar aprendizaje automático ya que los modelos deben hallar errores en pocas líneas de código. Además como los programas son breves y JavaScript es un lenguaje con una gramática estricta y limitada, en comparación con un lenguaje natural, consideramos que la cantidad total de tokens hallada en los programas es excesiva, esto se debe principalmente a la variabilidad posible de comentarios y de la elección de los nombres de variables y argumentos. Reduciremos la variabilidad de los datos por medio de la normalización de los mismos, lo cual abordaremos en el capítulo 4. Luego de realizar la normalización de los datos, la cantidad total de tokens distintos en los programas se reduce a 4535.

Capítulo 3

Desafíos y herramientas consideradas

Recordemos que la tarea motivadora de este trabajo es el de entrenar un modelo de inteligencia artificial que sea capaz de predecir una medida de similaridad entre programas, con el fin de posibilitar la construcción de herramientas que asistan durante el aprendizaje de la programación.

La información que nos provee IKUMI SRL es útil para esta tarea, ya que entre los datos disponibles contamos con programas (e información adicional relevante) etiquetados según el feedback asociado a ellos. Lamentablemente estas etiquetas son muy generales y no permiten determinar específicamente los tipos de errores posibles en un determinado programa.

Si tuviéramos un etiquetado más fino de los programas podríamos entrenar un modelo clasificador multiclase utilizando aprendizaje supervisado, pero desafortunadamente este tipo de etiquetado es muy costoso por dos motivos: primero porque requiere identificar y caracterizar apropiadamente los tipos de errores que se desean predecir (la cual no es una tarea trivial y esta puede estar fuertemente influenciada por el contexto del ejercicio asociado a los programas) y segundo porque este tipo de etiquetado, en principio, no puede realizarse automáticamente.

Por lo anterior, vamos a refinar nuestra estrategia de abordaje. Para poder predecir errores en un programa dado, vamos a buscar en los datos disponibles otros programas que sean similares al dado de modo que todos estos compartan errores similares. Como en los datos disponibles cada programa tiene una consulta asociada, esta consulta nos será de utilidad para producir una respuesta que permita corregir los errores del programa dado.

Para llevar a cabo este enfoque debemos primero determinar una buena métrica de similaridad entre programas.

La técnica de word embedding, nos permite obtener representaciones vectoriales de palabras.

Esta técnica trabaja sobre la hipótesis distribucional [22] la cual sugiere que: cuanto más similares semánticamente sean dos palabras, mas similares serán a su vez sus distribuciones en el lenguaje. Por lo cual tenderán a ocurrir con mayor frecuencia en contextos lingüísticos similares.

Con esa idea en mente, los word embeddings producen espacios vectoriales cuyas propiedades geométricas capturan propiedades semánticas del lenguaje de modo que dos palabras de semántica similar, sean representadas con vectores cercanos.

De esta forma, nuestra métrica de similaridad de programas será definida como la distancia euclídea de las representaciones vectoriales (provenientes de un word embedding) de programas. Sin embargo no debemos olvidar que la técnica del word embedding se desarrolló originalmente para textos escritos en lenguajes naturales (también conocidos como lenguajes humanos), los cuales son más flexibles y libres que los lenguajes de programación.

Por este motivo necesitaremos de una herramienta apropiada para obtener un word embedding sobre programas. En este capítulo analizaremos dos herramientas que nos permitirán obtener representaciones vectoriales de programas: Code2Vec [3] y FastText [6].

Por otro lado, queremos que efectivamente los vectores de programas equivalentes sean los más cercanos posibles, para que la métrica de similaridad sea tan representativa como se pueda. Para esto debemos trabajar con los datos disponibles con el fin de eliminar aquellas diferencias que no afectan a la semántica de los programas, como también para extraer la información más relevante en ellos. En el capítulo 4 abordaremos este problema en detalle.

3.1. Code2Vec

Code2Vec es una red neuronal desarrollada por investigadores de Technion y Facebook AI Research lab. A diferencia de otras redes neuronales utilizadas para procesamiento del lenguaje natural, las cuales trabajan sobre texto secuencial, Code2Vec aprovecha la estructura interna de los lenguajes de programación, por lo que está diseñada para aprender representaciones vectoriales de programas de un lenguaje de programación.

Oficialmente Code2Vec trabaja sobre código Java y recientemente agregaron soporte para C#. Pero existen aportes de la comunidad que la extienden de modo que pueda usarse en otros lenguajes de programación como por ejemplo Python [19] (extractor de AST-Paths para Python).

La implementación oficial, junto a los datos de entrenamiento que se usaron en el trabajo original, pueden encontrarse en `https://github.com/tech-srl/code2vec`. Además existe una demo online de Code2Vec, disponible en `https://code2vec.org/`, en la cual los autores nos demuestran las capacidades del modelo por medio de la tarea de predecir un nombre apropiado para una función a partir de su definición. Esta tarea es interesante para evaluar la calidad de las representaciones que se pueden obtener con el modelo, ya que normalmente estos nombres describen propiedades semánticas de las funciones. En la figura 3.1 observamos algunos ejemplos del desempeño de Code2Vec en esta tarea.

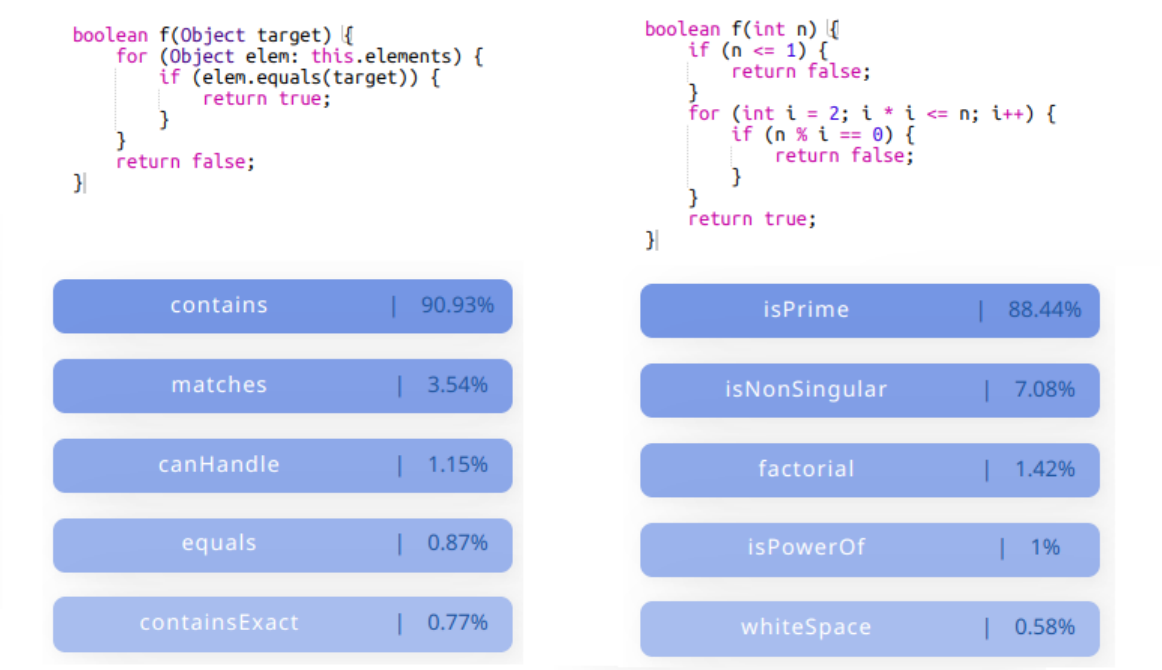


Figura 3.1: Screenshot de la demo online de Code2Vec: Distintas predicciones de nombres apropiados para las funciones dadas

Una propiedad interesante del espacio vectorial que produce el word embedding de Code2Vec es que codifica propiedades semánticas del lenguaje que se pueden obtener a partir de operaciones algebraicas entre vectores. Por ejemplo, en el word embedding, si un vector A es cercano a otros dos B y C, entonces la palabra que A representa tiene una semántica similar a la combinación de las palabras representadas por B y C (en la figura 3.2 tenemos un ejemplo). Similarmente es posible obtener analogías a partir de las representaciones vectoriales, en la demo online podemos hacerlo interactivamente.

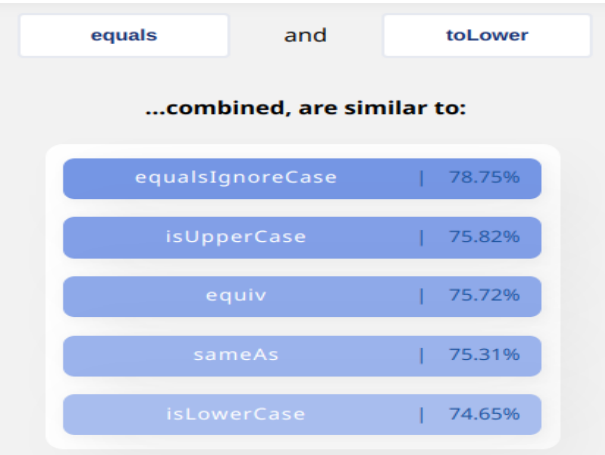


Figura 3.2: Screenshot de la demo online de Code2Vec: Ejemplo de la semántica codificada en el word embedding

Para explicar porque Code2Vec está diseñado específicamente para obtener representaciones de programas debemos introducir los siguientes conceptos.

Definición: Un Abstract Syntax Tree (AST) para una pieza de código C es una tupla $< N, T, X, s, \delta, \phi >$ donde N es un conjunto de nodos no terminales, T es un conjunto de nodos terminales, X es un conjunto de valores, $s \in N$ es la raíz, $\delta : N \to (N \cup T)^*$ es una función que mapea un nodo con sus nodos sucesores y $\phi : T \to X$ una función que mapea un nodo terminal a un valor.

Definición: Un AST-path de longitud k es una secuencia de la forma: $n_1 d_1, ..., n_k d_k n_{k+1}$ donde $n_1, n_{k+1} \in T$ son nodos terminales, para $i \in [2..k] : n_i \in N$ son nodos no terminales y para $i \in [1..k] : d_i \in \{\uparrow, \downarrow\}$ las cuales representan movimientos dentro del árbol. Si $d_i = \uparrow$, entonces $n_i \in \delta(n_{i+1})$; si $d_i = \downarrow$, entonces $n_{i+1} \in \delta(n_i)$. Dado un AST-path p, $start(p)$ denotara n_1 de p, y $end(p)$ denotara n_{k+1}

Definición: Dado un AST-path p, definimos su path-context como una tupla $< x_s, p, x_t >$, donde $x_s = (start(p))$ y $x_t = (end(p))$

3.1.1. ¿Cómo obtiene representaciones vectoriales?

Como mencionamos anteriormente Code2Vec aprovecha la estructura interna de los lenguajes de programación.

Para ello en lugar de tratar al programa como una secuencia lineal de tokens, dado un programa se calcula su AST y a partir de él se obtiene un multiconjunto de path-context.

La cantidad de path-contexts que se extraen de un programa es un parámetro de Code2Vec. Con cada path-context se obtienen los respectivos embeddings (representaciones vectoriales) de los valores de las hojas y del AST-path y luego estos se concatenan para dar lugar al embedding del path-context.

Para dar lugar a la representación vectorial final de los programas, Code2Vec propone combinar los embeddings de los path-context usando un mecanismo de atención, el cual es una red neuronal [25]. Los mecanismos de atención se utilizan para varias tareas dentro del machine learning como por ejemplo: traducción o resumen de texto, visión por computadora, reconocimiento del discurso, etc. El mecanismo de atención de Code2Vec aprende qué tanta importancia hay que destinar a cada elemento del multiconjunto; de este modo se determinan que path-context son los más relevantes en un programa, para luego agregar correctamente la información contenida en cada path-context en un único vector.

En la figura 3.3 se muestra la arquitectura de la red neuronal Code2Vec, la cual describiremos de izquierda a derecha.

Como mencionamos anteriormente, a partir de un programa obtenemos un multiconjunto de path-context, luego cada path-context se codifica como la concatenación de los embeddings (cuya dimensionalidad es un parámetro) de los valores hoja y del AST-path dando lugar a los "context vectors" de la figura 3.3. Cada "context vector" atraviesa, por separado, una capa densa (cuyo ancho es un parámetro) con función

de activación tanh dando lugar a una representación refinada de los path-context, llamada "combined context vector" en la figura 3.3.

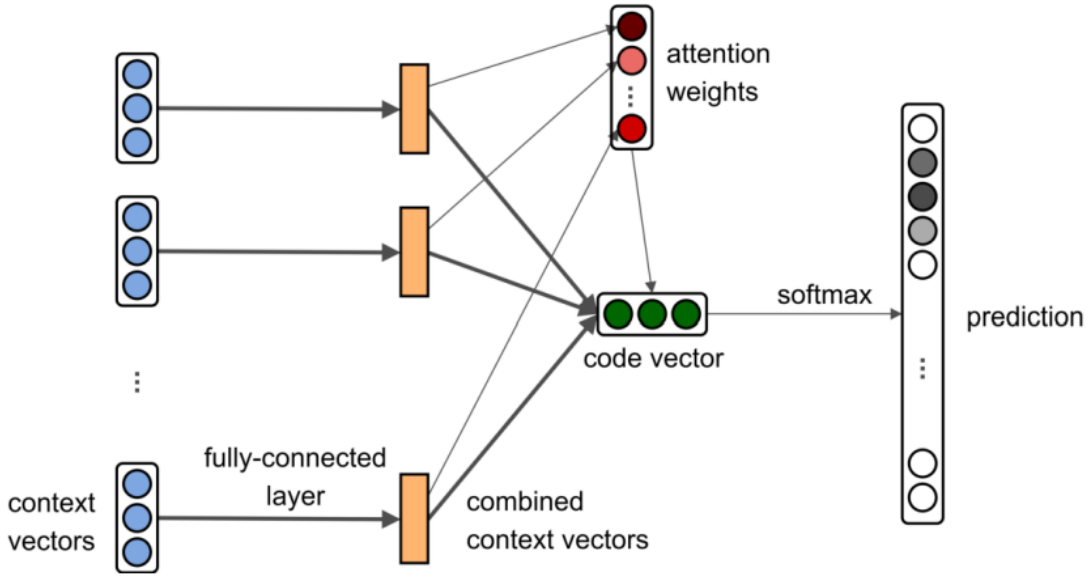


Figura 3.3: Arquitectura de la red neuronal Code2Vec para la predicción de nombres de funciones

Luego cada "combined context vector" es enviado al mecanismo de atención, el cual básicamente es una capa de neuronas que definen un "vector de atención global" (con la misma dimensionalidad que los "combined context vector"), que inicialmente tiene pesos aleatorios que luego se refinan con el entrenamiento. La cantidad de "atención" para cada path-context se calcula como el producto interno normalizado entre los "combined context vector" y el "vector de atención global", es decir:

$$\text{attention } \alpha_i = \frac{exp(\tilde{c_i}^T.a)}{\sum_i \tilde{c_i}^T.a}$$

donde $\tilde{c_i}$ es un "combined context vector" y a es el "vector de atención global". Finalmente la representación vectorial definitiva de los programas ("code vector" en la figura 3.3) se calcula como la combinación lineal de los "combined context vector" con sus respectivas cantidades de "atención", es decir:

$$\text{code vector } v_i = \sum_i \tilde{c_i}.\alpha_i$$

Además los autores de Code2Vec agregaron una layer densa con función de activación softmax para predecir un nombre apropiado para el programada dado.

3.2. FastText

FastText es una biblioteca gratuita de código abierto desarrollada por el laboratorio de investigación de inteligencia artificial de Facebook (FAIR lab). Esta biblioteca permite crear modelos de aprendizaje no supervisado que sean capaces de aprender representaciones vectoriales de texto.

Éstas representaciones permiten representar información que vas más allá de la sintaxis del lenguaje (e.g., asumiendo que palabras con significado similar ocurren en configuraciones similares con otras palabras, serán representadas en forma similar por el modelo). Además, FastText provee métodos que permiten utilizar estas representación para definir modelos aprendizaje supervisado. De esta forma, es posible definir modelos de clasificación de texto (e.g., para tareas como identificación de "hatespeach").

Podemos descargar FastText desde su página web `https://fasttext.cc/` y su código fuente esta disponible en `https://github.com/facebookresearch/fastText`.

Esta librería puede utilizarse sobre textos de cualquier idioma y de cualquier contexto. Una prueba de ello es que en la propia página de FastText, se encuentran disponibles muchos modelos pre entrenados usando la librería sobre diversos corpus. Hay 157 modelos que fueron entrenados sobre textos extraídos de Wikipedia y Common Crawl, un modelo para cada idioma distinto.

También hay modelos que se entrenaron para realizar tareas de clasificación, entre ellos hay modelos que fueron entrenados sobre DBpedia, Amazon review, Yahoo Answers, Agnews, etc.

Una característica atractiva de FastText es su eficiencia, ya que esta librería permite aprender representaciones vectoriales en poco tiempo sin necesidad de una GPU. Además los modelos que se obtienen tienen un desempeño comparable con modelos más costosos y complejos como las redes neuronales profundas.

A su vez los modelos pueden compactarse de modo que puedan implementarse en dispositivos de memoria limitada.

3.2.1. ¿Cómo obtiene representaciones vectoriales?

Para obtener representaciones vectoriales FastText utiliza extensiones de los modelos Skip-gram [18] y CBOW, los cuales forman parte de Word2Vec [17].

Los modelos Skip-gram y CBOW tienen objetivos distintos. Skip-gram predice las palabras vecinas a una dada, mientras que CBOW predice una palabra a partir de las palabras vecinas.

Sin embargo ambos modelos tienen arquitecturas muy parecidas, por lo que la explicación de como FastText extiende el modelo Skip-gram también explica como extender el modelo CBOW. A continuación explicaremos como se extiende el modelo Skip-gram en FastText.

El objetivo de entrenamiento de un modelo Skip-gram es encontrar representaciones vectoriales que sean útiles para predecir las palabras que rodean a otra dentro de una oración. Formalmente, dado una secuencia de palabras $w_1, w_2, ..., w_T$, el objetivo de un modelo Skip-gram es maximizar el siguiente promedio:

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{c\in C_t}\log\, p(w_c|w_t)$$

donde el contexto C_t es un conjunto de índices de palabras que rodean a w_t. La probabilidad de observar la palabra w_c en el contexto de la palabra w_t esta parametrizada por las representaciones vectoriales de w_c y w_t (v_{w_c} y v_{w_t}).

En el modelo Skip-gram la funcion p puede definirse de distintas maneras, la forma más básica es por medio de la funcion softmax, pero también se puede implementar usando negative sampling o heriarchical softmax. Por simplicidad veamos la definición usando softmax:

$$p(w_c|w_t) = \frac{e^{s(w_t,w_c)}}{\sum_w e^{s(w_t,w)}}$$

donde s es una funcion de puntuación.

En el paper original de Skip-gram, la funcion de puntuación s esta definida como el producto interno de las representaciones vectoriales de las palabras dadas, es decir:

$$s(w_t, w_c) = v_{w_t}^T.v_{w_c}$$

De este modo el modelo Skip-gram permite obtener representaciones vectoriales de palabras. Pero estas representaciones no consideran la estructura interna de las palabras.

Para solucionar esto, en FastText cada palabra w es representada como un multiconjunto de n-gramas de caracteres. Supongamos que consideramos la palabra w_t con multiconjunto de n-gramas $\{g_1, ..., g_k\}$, cuya representación vectorial es $\{v_{g_1}, ..., v_{g_k}\}$, con estos vectores podemos redefinir la funcion de puntuación s de la siguiente manera:

$$s(w_t, w_c) = (\sum_{i=1}^{k} v_{g_i}^T.v_{w_c}) + v_{w_t}^T.v_{w_c}$$

Con esta nueva definición de s, FastText usando el modelo Skip-gram puede aprender representaciones vectoriales de palabras considerando la morfología del lenguaje, como también permite aprender representaciones confiables para palabras raras en el vocabulario.

A partir de los vectores de palabras se puede obtener también un vector que represente a una oración completa, este se calcula como el promedio normalizado por L2 de los vectores de las palabras que constituyen la oración. El pseudocódigo de este algoritmo es el siguiente:

Listing 3.1: Pseudocodigo del algoritmo para la obtención de vectores a partir de oraciones

```
IN model
IN sentence
LET sentence_vector := new_vector(model.dimension)
LET count := 0
FOR word IN sentence DO:
    LET word_vector := getWordVector(word, model)
    LET word_norm   := L2_norm(word_vector)
    IF (word_norm > 0) THEN:
        word_vector := word_vector / word_norm
        sentence_vector := sentence_vector + word_vector
        count := count + 1
IF (count > 0) THEN:
    sentence_vector := sentence_vector / count
OUTPUT sentence_vector
```

Veamos un ejemplo de como se calcula un vector para una oración completa. Consideremos un modelo, de dimensión 3, entrenado sobre varias recetas de cocina y las palabras "banana" y "cake". Es importante tener en cuenta que para calcular el vector que represente a la frase "banana cake", implicitamente necesitamos tres vectores, uno para cada palabra y otro para un carácter especial que denota el final de la oración (EOS, End of Sentece). Para este modelo, el carácter EOS produce un vector nulo y por la definición del algoritmo 3.1, el vector de EOS se puede omitir.

Los vectores para las palabras "banana" y "cake" son:

$$\text{word\_vector(banana)} = [\text{-0.3863016 , 1.0250324 , -0.48413306}]$$
$$\text{word\_vector(cake)} = [\text{-1.0424687 , 0.31840122, -0.35989586}]$$

Luego la norma de estos vectores es 1.1976248 y 1.1478873 respectivamente. Al promediar los vectores divididos las respectivas normas obtenemos el vector de la frase "banana cake": [-0.61535966, 0.56663394, -0.3588866].

Notemos que, por la definición del algoritmo 3.1, el orden de las palabras en una oración no afectan en su representación vectorial. Por lo cual las oraciones "banana cake" y "cake banana" son representadas con el mismo vector.

3.2.2. Interfaz de Python

Actualmente FastText puede utilizarse por medio de comandos de bash o por medio de su interfaz de Python. Las funciones principales que ofrece FastText en Python son *train_supervised* y *train_unsupervised*.

Como el nombre lo indica ambas permiten entrenar modelos usando aprendizaje supervisado (para entrenar clasificadores) y no supervisado (para entrenar word embeddings) respectivamente. Los principales parámetros de estas funciones son:

- **input**: Archivo de entrenamiento.
- **model**: Tipo del modelo, FastText ofrece dos posibilidades: cbow y skipgram.
- **lr**: Learning rate.
- **dim**: Dimensión del word embedding resultante.
- **ws**: Tamaño de la ventana de contexto.

- **epoch**: Cantidad de épocas de entrenamiento.
- **minn**: Longitud mínima de los n-gramas de caracteres.
- **maxn**: Longitud máxima de los n-gramas de caracteres.
- **wordNgrams**: Longitud máxima de los n-gramas de palabras.
- **loss**: Función de costo utilizada durante el entrenamiento, FastText ofrece varias posibilidades: Negative Sampling, Hierarchical Softmax (una aproximación numérica de Softmax), Softmax y One-vs-All.

El archivo de input para ambas funciones delimita las distintas muestras del conjunto de datos por medio de ocurrencias de "\n". En caso de que utilicemos *train_supervised*, en el archivo de input debemos además agregar los respectivos labels a cada muestra del conjunto de datos siguiendo el siguiente formato.

```
__label__<label_name_1>__..._<label_name_n>__ <muestra> + \n
```

Otra funcionalidad interesante que ofrece FastText, es que se puede realizar un fine tuning automático de los hiper parámetros del modelo durante el aprendizaje supervisado. Para hacer uso de esta funcionalidad simplemente se introducen los siguientes argumentos en la función *train_supervised*.

- **autoTuneValidationFile**: Archivo de validación (correspondiente al conjunto de datos de validación)
- **autotuneDuration**: Tiempo en segundos para la búsqueda de hiperparametros óptimos.
- **autotuneMetric**: Este parámetro define el objetivo del fine tuning, por defecto busca maximizar el score f-1 de todas las labels presentes en los datos, pero también se puede enfocar en una label específica por ejemplo: "f1__label__⟨label_name⟩"

Una desventaja de esta funcionalidad es que perdemos el control sobre los hiper parámetros del modelo (como por ejemplo la dimensionalidad del word embedding), y el modelo resultante suele requerir más espacio de almacenamiento de lo normal. Otra desventaja es que no es posible recuperar los hiper parámetros de entrenamiento del modelo obtenido (al parecer es un error de la librería). De hecho sólo se puede deducir la dimensión del word embedding resultante.

3.3. Elección de herramienta

Luego de considerar las herramientas disponibles, para los objetivos de este trabajo parece razonable preferir Code2Vec sobre FastText, ya que Code2Vec fue desarrollado específicamente para obtener representaciones vectoriales a partir de programas.

Sin embargo, existen múltiples problemas que reducen la utilidad de esta herramienta en este trabajo.

Primero, si bien los autores de Code2Vec ofrecen una implementación oficial de la herramienta, esta implementación solo puede aplicarse sobre programas del lenguaje

Java, pero nuestros datos disponibles son programas de JavaScript. Es posible extender Code2Vec para que pueda utilizarse sobre otros lenguajes de programación, pero la implementación oficial no ofrece interfaces específicas para esto, por lo que se requiere de un esfuerzo considerable para implementar un algoritmo que pueda calcular los context-path del lenguaje objetivo y además integrarlo a Code2Vec.

Segundo, para poder utilizar Code2Vec debemos poder construir un AST para cada programa, pero este solo se puede construir si el programa satisface las restricciones sintácticas del lenguaje. En nuestros datos disponibles, el 20 % de los programas no compila, por lo cual no podemos usar Code2Vec en estos casos.

Tercero, entre los datos disponibles además de código tenemos reportes de la test suite, motor de evaluación de expectativas y del compilador. Estos reportes contienen información sobre la semántica y la estructura de los programas por lo que es razonable considerarlos para obtener una representación vectorial que modele la semántica de los programas. Sin embargo como Code2Vec solo trabaja sobre programas, la información de estos reportes no se puede aprovechar directamente usando únicamente Code2Vec.

Por último Code2Vec es muy sensible a variaciones que no afectan a la semántica de los programas, dando lugar a representaciones muy distintas de programas equivalentes. Veamos los siguientes ejemplos:

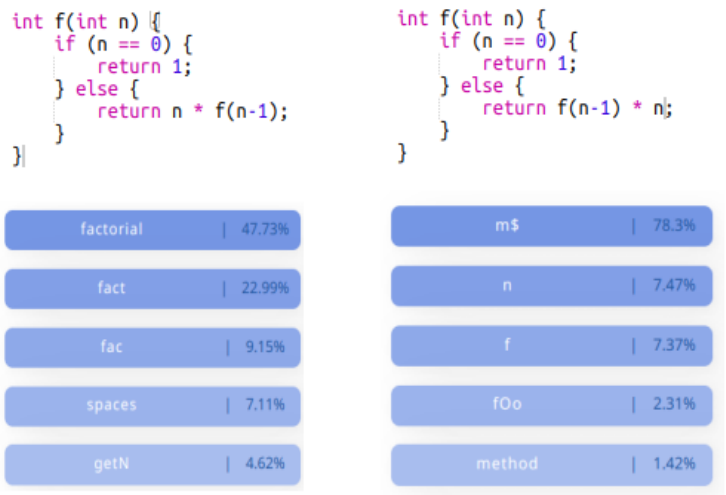


Figura 3.4: Captura de pantalla de la demo online de Code2Vec para la función factorial. En la parte superior vemos los programas de entrada y en la inferior las respectivas predicciones de Code2Vec

En la figura 3.4 la única diferencia entre ambos programas es el orden de los factores en la llamada recursiva pero esta diferencia produce vectores tan distintos que el modelo no puede detectar que ambas funcionen definen a la funcion factorial.

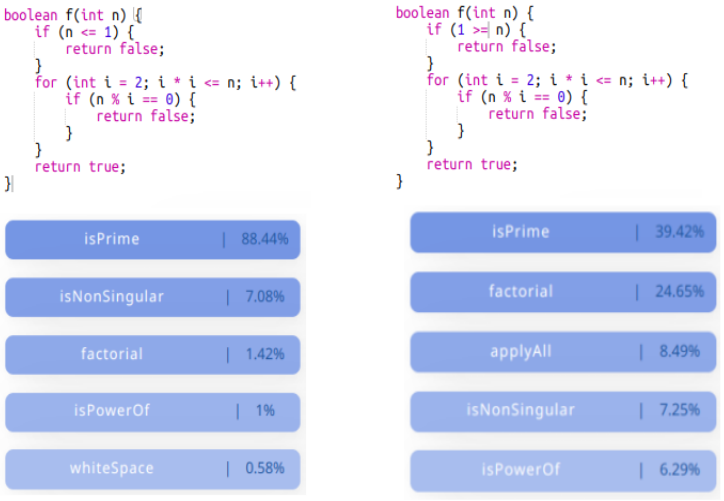


Figura 3.5: Captura de pantalla de la demo online de Code2Vec para una función que determina si el número de entrada es primo o no. En la parte superior vemos los programas de entrada y en la inferior las respectivas predicciones de Code2Vec

En la figura 3.5 la diferencia esta en como se realiza la comparación de la primera guarda, si bien en ambos casos predice correctamente el nombre "isPrime", en el programa modificado la confianza que tiene el modelo sobre esta predicción es mucho menor.

Como podemos observar en los ejemplos anteriores, las representaciones vectoriales que produce Code2Vec para dos programas equivalentes con mínimas diferencias sintácticas pueden ser muy distintas.

Por otro lado la librería FastText resulta ser más apropiada para este trabajo, ya que puede procesar cualquier tipo de texto. Además los modelos que produce FastText tienen un desempeño comparable a modelos más complejos como las redes neuronales profundas, y por la forma en que calcula vectores de oraciones, usando FastText sobre los ejemplos anteriores obtenemos los mismos vectores (lo cual es razonable para nuestro objetivo).

Otro punto a favor para FastText es su eficiencia, ya que puede aprender representaciones vectoriales en dispositivos que no cuentan con GPU, mientras que usar Code2Vec es prohibitivo en un dispositivo sin GPU.

3.4. Similitud de programas

Como mencionamos anteriormente, para nuestro modelo, la similaridad de dos programas se define en términos de la distancia euclidiana que hay entre los vectores que representan estos programas, cuanto más cercanos sean estos vectores más similares se consideran los programas correspondientes.

Por lo cual, para hallar los programas similares a uno dado, simplemente calculamos el respectivo vector del programa y buscamos los vectores más cercanos en nuestros datos. Como los programas en los datos disponibles contienen consultas y respuestas asociadas, podemos utilizar estas respuestas para ofrecer una ayuda que

permita corregir los errores presentes en el programa dado, ya que se espera que los programas similares contengan errores en común y, por lo tanto, claramente las respuestas a consultas que se den en uno de los programas es informativa para los otros.

Como hemos elegido usar FastText para obtener las representaciones vectoriales de los programas, el algoritmo general que utilizaremos para predecir los programas mas cercanos a uno dado es el siguiente:

1. De los datos disponibles extraemos los programas y los normalizamos, dando lugar al conjunto datos de entrenamiento.

2. Con estos datos entrenamos un modelo no supervisado de FastText y con el obtenemos las representaciones vectoriales de los datos de entrenamiento y guardamos estos vectores.

3. Dado un programa nuevo de entrada, lo normalizamos y luego producimos su representación vectorial con el modelo previamente entrenado.

4. Finalmente buscamos en los vectores guardados aquellos que sean mas cercanos al vector del programa de entrada.

El algoritmo anterior sólo considera los programas para producir los embeddings, pero también podemos a su vez considerar los reportes asociados a los programas, los cuales ya vimos que aportan información sobre la semántica y estructura de los programas. En el capítulo 6 analizaremos en detalle cómo utilizar la información que proveen los reportes de programas para producir vectores de programas.

Capítulo 4

Procesamiento y normalización de los datos

Usualmente, en tareas habituales del procesamiento del lenguaje natural (NLP) se trabaja sobre un lenguaje natural como lo es el Español o el Ingles. En estos lenguajes la cantidad de palabras distintas que podemos hallar en un corpus es muy alta.

Esto ocurre porque los corpus en lenguaje natural suelen ser extensos y también porque estos lenguajes son flexibles y evolucionan constantemente.

En el capitulo 2 mencionamos que en nuestros datos hay **48886** tokens distintos lo cual, para una tarea usual de NLP, se puede lo considerar como un corpus pequeño.

Sin embargo no debemos olvidar la naturaleza de nuestros datos. Estos son ejemplos de programas de JavaScript, para ejercicios con soluciones breves. Por lo cual podemos considerar que la cantidad de tokens distintos hallados es excesiva.

Podemos suponer que una gran parte de la variabilidad de los datos proviene de decisiones superfluas que no afectan a la semántica de los programas, como por ejemplo las distintas elecciones de nombres de argumentos y variables.

En este capitulo veremos como procesar y normalizar nuestros datos para reducir la variabilidad presente en estos.

El procesamiento de datos es una técnica común del machine learning, ya que en la práctica no todos los datos que se recopilan para una tarea en específico aportan información significativa para entrenar un modelo de machine learning. El objetivo principal del procesamiento de los datos es remover toda la información no significativa (para la tarea de interés) de los datos disponibles y exponer la información relevante de modo que se reduzcan los esfuerzos necesarios para que el modelo de machine learning aprenda patrones/propiedades deseables en ellos.

En el contexto de este trabajo el procesamiento de datos que realizaremos a continuación tiene como objetivo exponer la similaridad de los códigos equivalentes y extraer información relevante de los reportes asociados. Todo esto con el fin de mejorar tanto como sea posible las representaciones vectoriales de los programas, de modo de mejorar la métrica de similaridad de programas.

4.1. Procesamiento de código

El procesamiento de código que proponemos consiste de dos etapas: en la primera etapa suprimimos aquellas muestras de los datos que no son útiles para el entrenamiento de nuestro modelo y en la segunda haremos que las instancias de código equivalentes sean más similares, eliminando diferencias que no afectan a la semántica de los programas usando técnicas de normalización.

4.1.1. Eliminación de información no significativa

Entre las **31383** consultas disponibles, existen casos en que las instancias de código asociado no corresponden a un programa. Sino más bien a un comentario del estudiante. Por ejemplo:

hola que tal no entiendo cómo plantear el ejercicio

No entiendo nada

no se jugar al truco

// No estoy entendiendo nada.

Este tipo de "programas" no aporta información a nuestro modelo de machine learning, ya que estos son textos en lenguaje natural, es más, este tipo de textos aumenta el tamaño del vocabulario con palabras que no ocurren normalmente en un programa valido posible o peor aún, puede contener palabras que sí tienen una semántica definida dentro JavaScript que se usan bajo reglas gramaticales precisas y en contextos específicos (e.g., la palabra function).

También existen casos de consultas cuyo código asociado es un programa trivial, como por ejemplo los siguientes:

`function longitudNombreCompleto("Cosme", "Fulanito")`
`function gritar {return}`
`function meses(){` `}`
`function xor(a,b)`

Cuadro 4.1: Ejemplos de programas triviales hallados en los datos

Si bien estos programas respetan la estructura del lenguaje JavaScript, estos contienen definiciones vacías que no aportan información significativa, ya que no hay suficiente contenido para determinar un error puntual por parte del estudiante.

Para filtrar las instancias de consultas que no son programas sino mas bien comentarios del estudiante, podemos simplemente filtrar aquellas instancias de código en las que no ocurran ninguna de las palabras: "function", "let", "var" o "const". De este modo podemos filtrar fácilmente aquellas instancias de código que no definen nada. Este filtro elimina **252** muestras no significativas, ya que todos los ejercicios de Mumuki para el curso "programación imperativa con Javascript" solicitan definir funciones o variables.

Para filtrar las instancias de código como los mostrados en 4.1, no podemos simplemente utilizar una expresión regular para intentar capturar definiciones de funciones con cuerpo trivial, porque el 20 % de los programas disponibles no son correctos sintácticamente por lo que esta idea no funciona en estos casos. En lugar de eso se optó por implementar un filtro condicional, si el código no compila (determinado por el "submition_status" errored) entonces filtramos si la cantidad de tokens en el código es menor a 10. En caso contrario podemos usar una expresión regular para capturar el cuerpo de las definiciones de funciones en el código y filtramos en caso de que el código no tenga definiciones con al menos 5 tokens

Si bien la signatura de una función con 3 parámetros requiere de a lo sumo 11 tokens, debemos ser cuidadosos con la cota del filtro ya que podemos capturar accidentalmente código que inherentemente es muy breve, como por ejemplo las respuestas correctas del ejercicio "mitad" en donde se pide definir una función que dado un número devuelva su mitad. Del mismo modo se eligió cuidadosamente la cota en caso de que el código compilara.

Este filtro elimina **570** muestras no significativas y para verificar que efectivamente las cotas del filtro son apropiadas, se inspeccionaron manualmente los programas filtrados y se comprobó que las definiciones eran triviales. Algunos ejemplos de las instancias eliminadas por este filtro:

```
function endulzarMenu (menu){;}
```

```
function mitad(10/2){
    return
}
```

```
function lalala {
```

```
function xor(){}
```

Cuadro 4.2: Ejemplos de programas filtrados

Los ejemplos mostrados en el cuadro 4.2 claramente pueden aparecer cuando un estudiante realice una consulta a nuestro modelo, sin embargo, es usualmente aconsejable eliminar casos extremos como estos del material de entrenamiento ya que pueden producir comportamientos anómalos en el modelo.

4.1.2. Normalización de código

Para aumentar la similaridad sintáctica entre programas semejantes, aplicaremos diferentes técnicas de normalización sobre estos. Cuando normalizamos programas debemos tener el cuidado de preservar la semántica (y los errores) presentes en el texto original, de modo que al normalizar los programas obtenemos programas con menor variabilidad sintáctica, equivalentes a los originales.

Las distintas instancias de código, para un determinado ejercicio, pueden variar de muchas formas posibles: pueden variar en los nombres elegidos para las variables y/o argumentos de las funciones, también algunas instancias introducen comentarios JavaScript. En el caso de instancias de código que no compilan, la variabilidad es aun mayor debido a los diversos errores sintácticos posibles. Los siguientes son ejemplos de programas del ejercicio "Cartelitos Óptimos" hallados en los datos disponibles.

```
function escribirCartelitoOptimo (titulo, nombre, apellido) {
    return escribirCartelito(titulo, nombre, apellido,
                             longitud(nombre+apellido)>15)
}
```

```
function escribirCartelitoOptimo(t, n, a){
  return escribirCartelito(t, n, a, longitud(n+a)>15)
}
```

```
//modifica esta funcion
function escribirCartelitoOptimo(titulo, nombre, apellido) {
  return (nombre+apellido.longitud()>15) ?
         (titulo + " " + apellido) : escribirCartelito;
}
// (boolean) ? (lo devuelve por true) : (lo que devuelve por false)
```

```
function escribirCartelitoOptimo (titulo,nombre,apellido) {
  longitud(nombre+apellido) > 15;
  return escribirCartelito(titulo, nombre, apellido, quiereCartelCorto)
}
```

```
function escribirCartelitoOptimo(titulo,nombre,apellido,longitud){
    return escribirCartelito(titulo,nombre,apellido,longitud)longitud>15
}
```

Cuadro 4.3: Múltiples ejemplos de instancias de código para el ejercicio "Cartelitos Óptimos"

De los ejemplos mostrados en 4.3, los primeros dos programas son equivalentes y a su vez son soluciones correctas, pero ambas utilizan distintos nombres de argumentos. Si consideramos estas diferencias puede suceder que nuestro modelo de FastText produzca vectores que no sean cercanos para estos programas. Además muchos nombres de variables y/o argumentos ocurren infrecuentemente en nuestros datos, lo cual incrementa el tamaño del vocabulario de nuestro modelo con palabras que no son relevantes.

Si bien los ejercicios de Mumuki proponen nombres canónicos para las variables y argumentos de los programas (por ejemplo en la figura 1.2, el enunciado indica que debemos definir una función áreaTriangulo cuyos argumentos son 'base' y 'altura'), muchos estudiantes no los consideran e implementan soluciones con distintos nombres.

Para corregir esta situación se realizaron renombres de argumentos y variables por nombres genéricos de la forma: var⟨i⟩ y arg⟨i⟩, donde i es un número natural que indica el orden de definición (los órdenes de definición de variables y de argumentos son independientes) en el programa. Por ejemplo consideremos el siguiente programa

```
function perimetroTriangulo(lado1, lado2, lado3){
    let perimetro = lado1 + lado2+ lado3
    return perimetro
}
```

Luego de aplicar los renombres de variables y argumentos obtenemos el siguiente resultado:

```
function perimetroTriangulo(arg0, arg1, arg2){
    var var0 = arg0 + arg1 + arg2
    return var0
}
```

Como los renombres de variables y argumentos no cambian la semántica del programa y tampoco afectan a los errores presentes en el programa original, esta normalización resulta ser apropiada para nuestro objetivo.

Al aplicar este renombre sobre los primeros dos ejemplos (equivalentes) de la tabla 4.3, se puede ver que ahora ambos códigos resultan ser idénticos:

```
function escribirCartelitoOptimo(arg0, arg1, arg2) {
    return escribirCartelito(arg0, arg1, arg2, longitud(arg1+arg2 )>15)
}
```

En algunos casos, los estudiantes escriben programas que contienen comentarios JavaScript (como el tercer ejemplo de la tabla 4.3).

Cómo estos no afectan a la semántica del programa, agregan oraciones que no son propias de la gramática de JavaScript y aumentan innecesariamente el tamaño del vocabulario, decidimos eliminar estos comentarios.

Para ello simplemente utilizamos expresiones regulares que capturen los comentarios JavaScript.

Otro detalle a tener en cuenta es que, como consideramos token cualquier palabra separada por un espacio entonces las frases:

escribirCartelitoOptimo(titulo, nombre, apellido)

escribirCartelitoOptimo (titulo, nombre, apellido)

tienen la misma semántica (por lo menos en JavaScript) pero producen cantidades distintas de tokens, la primera tiene 3 tokens “escribirCartelitoOptimo(titulo,”, “nombre,” y “apellido)” mientras que la segunda tiene 4 “escribirCartelitoOptimo”, “(titulo,”, “nombre,” y “apellido)”.

Esta situación ocurre muchas veces en los datos con varios caracteres particulares. En los lenguajes de programación algunos caracteres especiales no imponen restricciones estrictas en la sintaxis del lenguaje. Por ejemplo: 2+3 y 2 + 3 para cualquier lenguaje de programación tienen exactamente la misma semántica.

Para regularizar esta situación, agregamos un espacio alrededor de todos los caracteres que merecen ser considerados como un token en sí mismo, como por ejemplo los siguientes:

( ) , { } + − ∗ = > < / & | ! ; [ ] .

También existen combinaciones de caracteres especiales que tienen una semántica definida dentro de JavaScript por lo que también merecen ser considerados como tokens, como los siguientes:

!=== === !== == >= <= != && || ++ −− ≪ ≫

Hay además combinaciones de caracteres que suelen ocurrir en los datos disponibles, que si bien no son combinaciones válidas dentro de JavaScript la intención de

estas es clara y denotan un error sintáctico específico, por lo que también merecen ser consideradas como tokens individuales.

>=== <=== ===> >== <== ==> ==< >= =<

Luego de considerar como tokens individuales los caracteres de la lista, ahora tanto "escribirCartelitoOptimo(titulo, nombre, apellido)" cómo "escribirCartelitoOptimo (titulo, nombre, apellido)" definirán exactamente los mismos tokens:

escribirCartelitoOptimo (titulo , nombre , apellido)

Finalmente se realizaron además otras normalizaciones mínimas como quitar las tildes, normalizar la lista vacía (para que siempre ocurra como "[]") y quitar los caracteres de salto de línea o de tabulación.

No se consideró la opción de llevar todo el texto a minúscula porque Mumuki, además de verificar la funcionalidad del código por medio de pruebas, tambien verifica que este satisfaga las convenciones de estilo de JavaScript (en las figuras 1.4 y 4.1 hay ejemplos de esto) y al llevar al texto a minúscula existen casos en que esta información se pierde, por ejemplo cuando se espera un texto en "camelCase"

4.2. Procesamiento de reportes

Entre los datos que nos provee Mumuki hay tres tipos de reportes que un programa puede tener: reportes del motor de evaluación de expectativas, reportes de testing y reportes de compilador.

4.2.1. Reportes del motor de evaluación de expectativas

Mumuki utiliza los reportes del motor de evaluación de expectativas para poder generar las devoluciones del tipo "passed_with_warnings".

De cierto modo estos reportes reflejan una evaluación de la estructura de las instancias de código. El siguiente es un reporte asociado a una instancia de código del ejercicio "Cartelitos óptimos".

Listing 4.1: Muestra del ejercicio "Cartelitos Optimos"

```
function escribirCartelitoOptimo(titulo,nombre,apellido){
    return escribirCartelito(
        titulo,
        nombre,apellido,
        longitud(nombre+apellido)
    )>15
}
```

Listing 4.2: Reporte del motor de evaluacion de expectativas asociado

```
- !ruby/hash:ActiveSupport::HashWithIndifferentAccess
  binding: "*"
  inspection: Declares:=escribirCartelitoOptimo
  result: passed
```

```
- !ruby/hash:ActiveSupport::HashWithIndifferentAccess
  binding: escribirCartelitoOptimo
  inspection: Uses:escribirCartelito
  result: passed
- !ruby/hash:ActiveSupport::HashWithIndifferentAccess
  binding: Intransitive:escribirCartelitoOptimo
  inspection: Not:UsesIf
  result: passed
- !ruby/hash:ActiveSupport::HashWithIndifferentAccess
  binding: "*"
  inspection: Not:Declares:escribirCartelito
  result: passed
- !ruby/hash:ActiveSupport::HashWithIndifferentAccess
  binding: "*"
  inspection: DeclaresComputationWithArity3:escribirCartelitoOptimo
  result: passed
```

En el reporte anterior se puede identificar el criterio de evaluación del motor de evaluación de expectativas. Las instancias correctas del ejercicio "Cartelitos óptimos" deben satisfacer (además de la testsuite) las siguientes condiciones:

- Deben declarar la función "escribirCartelitoOptimo".
- En la definición de "escribirCartelitoOptimo" no debe usarse la estructura condicional "if".
- No deben definir la función "escribirCartelito" (en el enunciado del ejercicio nos indican que esta función está disponible para usarse directamente).
- La función "escribirCartelitoOptimo" debe admitir solo 3 argumentos.

Además de estas condiciones, Mumuki siempre verifica que se utilicen las convenciones de nombres de JavaScript y las buenas prácticas de programación como por ejemplo: no definir variables innecesariamente. En la siguiente figura (figura 4.1) se muestra una instancia de código categorizada como "passed_with_warnings" para el ejercicio "Cartelitos Óptimos".

Como se puede observar estos reportes aportan información significativa para la tarea de aproximar códigos similares. Pero para considerar esta información debemos tratar dos problemas: Primero, el reporte completo contiene frases que no son significativas (e.g., !ruby/hash:ActiveSupport:: HashWithIndifferentAccess), del reporte solo nos interesan los nombres de las evaluaciones y sus respectivos resultados. Segundo las instancias de código que no compila (o catalogadas como "errored"), no tienen este tipo de reportes.

Para solucionar el primer problema, utilizaremos expresiones regulares para extraer los casos de prueba junto a sus respectivos resultados y agregamos al comienzo del reporte la palabra EXPECTATIONS. Si procesamos el reporte anterior obtenemos:

```
EXPECTATIONS
Declares:=escribirCartelitoOptimo passed
Uses:escribirCartelito passed
Not:UsesIf passed
Not:Declares:escribirCartelito passed
DeclaresComputationWithArity3:escribirCartelitoOptimo passed
```

Figura 4.1: Screenshot de Mumuki: Enunciado del ejercicio “Cartelitos Óptimos” junto al código propuesto y la respectiva retroalimentación.

Para el caso en que la instancia de código no cuente con este reporte, simplemente dejaremos un token especial, ENA, que indique esta situación. ENA significa “Evaluation Not Available”.

4.2.2. Reportes de la test suite

Mumuki utiliza los reportes de testing para poder generar las retroalimentaciones del tipo “failed”. Este tipo de reportes tiene una estructura similar a los reportes del motor de evaluación de expectativas, ya que ambos listan casos de prueba con sus respectivos resultados. Por ejemplo el siguiente es un reporte de testing asociado a una instancia de código del ejercicio “Cartelitos óptimos”:

Listing 4.3: Muestra del ejercicio ’Cartelitos optimos’

```
function escribirCartelitoOptimo(titulo,nombre,apellido){
    return escribirCartelito(
        titulo,
        nombre,
        apellido,
        longitud(nombre+apellido)
    )>15
}
```

Listing 4.4: Reporte de testing asociado

```
ruby/hash:ActiveSupport::HashWithIndifferentAccess
title: escribe un cartelito largo cuando el nombre completo es corto
status: failed
result:
```

```
```
    false == 'Ing. Carla Toledo'

```
!ruby/hash:ActiveSupport::HashWithIndifferentAccess
title: escribe un cartelito corto cuando el nombre completo es largo
status: failed
result:
```
    false == 'Dr. Schwarzschild'

```
```

Si bien la estructura de los reportes es similar, hay una diferencia a considerar. Los reportes de testing además de dar el nombre del caso de prueba y el resultado, suelen ofrecer una comparación del valor obtenido vs. valor esperado. También en algunos casos se describen errores del código, por lo que esta información del reporte es útil.

Al igual que antes no todas las instancias de código tienen un reporte de testing asociado. En caso de contar con un reporte de testing usaremos expresiones regulares para extraer los nombres de los casos de prueba junto a sus respectivos resultados y descripción, luego agregamos al comienzo la palabra TESTING. Si procesamos el reporte de testing anterior obtenemos:

```
TESTING
escribe un cartelito largo cuando el nombre completo es corto
failed  false == 'Ing. Carla Toledo'
escribe un cartelito corto cuando el nombre completo es largo
failed  false == 'Dr. Schwarzschild'
```

Nuevamente en caso de no contar con este reporte simplemente dejamos un token especial que denote esta situación TNA, que significa “Testing Not Available”.

4.2.3. Reportes del compilador

Finalmente nos queda procesar los reportes del compilador. Estos solo son producidos en Mumuki frente a programas que producen errores en tiempo de compilación. Al revisar estos reportes nos dimos cuenta de que cada reporte sólo informa un error y este solo puede ser de dos tipos posibles: ReferenceError y SyntaxError.

El siguiente es un ejemplo de reporte con el error ReferenceError

Listing 4.5: Muestra del ejercicio ’Cartelitos optimos’ con error de referencia

```
function escribirCartelitoOptimo(titulo, nombre, apellido) {
  longitud(nombre + apellido) > 15
}
return escribirCartelito(titulo, nombre, apellido, true)
```

Listing 4.6: Reporte del compilador asociado

```
return escribirCartelito(titulo, nombre, apellido, true)
                                 ^
ReferenceError: titulo is not defined
```

```
    at Object.<anonymous> (solucion.js:64:26)
    at Module._compile (module.js:409:26)
    at Object.Module._extensions..js (module.js:416:10)
    at Module.load (module.js:343:32)
    at Function.Module._load (module.js:300:12)
    at Module.require (module.js:353:17)
    at require (internal/module.js:12:17)
    at /usr/local/lib/node_modules/mocha/lib/mocha.js:219:27
    at Array.forEach (native)
    at Mocha.loadFiles (/usr/local/lib/node_modules/mocha.js:216:14)
    at Mocha.run (/usr/local/lib/node_modules/mocha/mocha.js:468:10)
    at Object._anon (/usr/local/lib/node_modules/mocha/mocha:403:18)
    at Module._compile (module.js:409:26)
    at Object.Module._extensions..js (module.js:416:10)
    at Module.load (module.js:343:32)
    at Function.Module._load (module.js:300:12)
    at Function.Module.runMain (module.js:441:10)
    at startup (node.js:139:18)
    at node.js:968:3
```

De todo este reporte solo nos interesa la frase "ReferenceError: titulo is not defined", ya que la traza del error es muy extensa e irrelevante. De modo que el reporte anterior lo procesaremos como:

```
REFERENCE_ERROR titulo is not defined
```

El siguiente es un ejemplo de reporte con el error ValueError:

Listing 4.7: Muestra del ejercicio 'Cartelitos optimos' ValueError

```
function escribirCartelitoOptimo(titulo, nombre, apellido){
  return escribirCartelito(
        titulo,
        nombre,
        apellido,
        (longitud(apellido + nombre) > 15
    )
}
return escribirCartelito(titulo, nombre, apellido, true)
```

Listing 4.8: Reporte del compilador asociado

```
solucion.js:3
escribirCartelito(titulo,nombre,apellido,(longitud(apellido + nombre) > 15)}
                                                                         ^

SyntaxError: missing ) after argument list
```

De todo este reporte solo nos interesa la descripción del error sintáctico. De modo que el reporte anterior lo procesaremos como:

```
SYNTAX_ERROR :: missing ) after argument list
```

Al igual que en los casos anteriores, no todas las instancias de código tienen un reporte de compilador asociado. En tal caso dejaremos un token especial que denote esta situación, CNA que significa Compilation Not Available.

4.3. Resultados Obtenidos

A continuación resumiremos como impactan en los datos el procesamiento y la normalización descrita en este capitulo. Los resultados que veremos en las siguientes tablas se obtuvieron sobre todos los datos crudos considerando programas y reportes.

Filtro	Programas filtrados	Tokens eliminados
Filtro A	252	223
Filtro B	570	453

Cuadro 4.4: Comparación de los filtros de programas no informativos

En la tabla 4.4 la columna "Tokens eliminados" contabiliza aquellos tokens que solo ocurren en los programas filtrados.

"Filtro A" hace referencia al filtro que captura aquellos programas que no tienen definiciones, es decir, en ellos no ocurren ninguna keyword de definición ("function", "var", "let", "const").

Mientras que "Filtro B" hace referencia al filtro que captura aquellos programas con definiciones triviales (i.e., aquellos programas que tienen muy pocos tokens en sus definiciones 4.2).

Normalización	Tokens eliminados
Quitar comentarios	620
Nombres de variables	3272
Nombres de argumentos	11529
Tokenizacion	31885

Cuadro 4.5: Comparación de la normalización de programas

En la tabla 4.5 "Tokenizacion" hace referencia al proceso de considerar como token aquellas combinaciones de caracteres con semántica definida, por ejemplo "==".

Como podemos ver en los resultados quitar los comentarios solo reduce 620 tokens del total. Esto ocurre porque en algunos ejercicios Mumuki da a modo de ayuda una definición parcial con comentarios que indican que completar. Muchos estudiantes eliminan estos comentarios y otros los dejan, pero muy pocos agregan comentarios nuevos.

Recordemos que la cantidad de tokens distintos en los programas es de **45000**, de los cuales casi 15000 corresponden a nombres de argumentos y de variables. Además en la tabla podemos observar que la mayor variabilidad en los programas se da en las diversas combinaciones posibles entre variables y operadores (e.g., .ª+ bz .ª +b"son combinaciones equivalentes pero distintas).

La aplicación de la combinación de todas las técnicas de normalización descritas en la tabla 4.5 reduce la cantidad de tokens (de programas) de 45000 a 4535. Estos resultados obtenidos confirman nuestra hipótesis de que la cantidad de tokens distintos hallada originalmente era en realidad excesiva y que la mayoría de ellos provienen de variaciones superfluas que no afectan a la semántica de los programas.

Finalmente veamos los resultados de normalizar los reportes

Reporte	**Tokens eliminados**
Test suite	1723
Expectativas	49
Compilador	726

Cuadro 4.6: Comparación de la normalización de reportes

Como podemos ver en la tabla 4.6 la normalización de los no reduce significativamente la cantidad de tokens totales. Esto ocurre porque los reportes tienen un formato estandarizado y tienen muy poca variabilidad.

Finalmente al aplicar todas las técnicas de normalización juntas sobre los programas y los reportes reducimos la cantidad total de tokens de 48886 a 6031.

Capítulo 5

Efectos del procesamiento y normalización de datos

Para demostrar que el procesamiento y normalización de los datos mejoran las representaciones de programas que se pueden obtener con FastText, proponemos realizar algunos experimentos y comparar resultados.

5.1. Experimento: Clasificación de programas por enunciado

Para este experimento vamos a comparar el desempeño de modelos clasificadores de FastText, en la tarea de clasificar programas por enunciado. Para ello usaremos como labels el id del enunciado asociado a cada ejercicio, recordemos que hay **122** ID's distintas y la distribución de ellas la podemos ver en la figura 2.1.

Todos los modelos que compararemos tendrán los mismos hiper parámetros, pero estarán entrenados sobre variaciones de los datos disponibles. Los hiper parámetros que usaremos son:

- **model**: Skip Gram
- **lr**: 0.45
- **dim**: 30
- **ws**: 5
- **epoch**: 20
- **minn**: 3
- **maxn**: 6
- **wordNgrams**: 3
- **loss**: Hierarchical Softmax

Consideraremos cuatro variaciones de los datos: en la primera solo consideraremos programas sin normalizar, en la segunda consideraremos programas normalizados, en

la tercera consideraremos toda la información (código y reportes) sin normalizar y en la cuarta consideraremos toda la información normalizada.

Para las variaciones que tienen reportes y código, para crear la muestra simplemente concatenamos todo de modo que las muestras tendrán el siguiente formato:

__label__⟨id_ejercicio⟩__ ⟨codigo⟩ ⟨reporte testing⟩ ⟨reporte expc⟩ ⟨reporte compilador⟩

Sin embargo en las cuatro variaciones filtraremos las muestras que contengan un programa trivial asociado (es decir, programas que no demuestran una intención de solución).

Para entrenar los modelos, en cada caso se fraccionó el dataset, previamente aleatorizado (usando siempre la misma seed para comparar los resultados), en dos partes. 80 % de los datos se destinaron al conjunto de entrenamiento y los 20 % restantes al conjunto de testeo. En la siguiente tabla mostramos los resultados del experimento.

Conjunto de datos	**Exactitud**	**Tamaño del vocabulario**
Programas sin normalizar	84.54 %	38236
Programas normalizados	97 %	3975
Programas + reportes	84.57 %	41589
Programas + reportes (normalizados)	97.98 %	5370

Cuadro 5.1: Resultados de la clasificación por ejercicio

Como podemos ver en los resultados 5.1, la normalización de programas da lugar a mejores representaciones de programas, de modo que es más fácil para el mismo clasificador obtener mejores resultados. Además el tamaño del vocabulario se reduce drásticamente (el tamaño del vocabulario que se muestra en la tabla corresponde al vocabulario hallado en el conjunto de datos de entrenamiento).

Notemos que el modelo de FastText al considerar únicamente programas posee suficiente información para lograr excelentes resultados en este experimento. Si bien los reportes asociados a los programas también aportan información que pueden ayudar al modelo, al combinar programas y reportes corremos el riesgo de que el modelo no "priorice" correctamente la información y este degenere su desempeño. Sin embargo se observa en los resultados que considerar la combinación de reportes y programas no afecta negativamente al desempeño del modelo FastText.

Esta observación es interesante porque nos sugiere que podemos agregar los reportes directamente sobre las muestras de entrenamiento y aun así FastText puede aprender representaciones apropiadas para los programas. Esta no es una observación trivial ya que intuitivamente uno esperaría que al agregar más información al modelo, en este caso los reportes, este podría degradarse porque ya no solo busca patrones en el código sino también en los reportes, cuyo texto tiene una estructura y reglas gramaticales distintas al código.

Seguiremos evaluando esta observación en más experimentos, ya que abre la posibilidad de que se pueda trabajar con todos los datos usando únicamente FastText.

5.2. Experimento: Clasificación de programas por feedback

Al igual que en el experimento anterior, vamos a comparar el desempeño de modelos clasificadores de FastText. En este caso la tarea es clasificar programas por feedback, para ello usaremos como label el "submission_status" asociado a cada programa, recordemos que distribución de estas labels la tenemos en la figura 2.2

Al igual que antes, consideraremos las mismas cuatro variaciones de los datos y además todos los modelos compartirán hiper parámetros, pero esta vez los hiper parámetros serán:

- **model**: Skip Gram
- **lr**: 0.9
- **dim**: 30
- **ws**: 5
- **epoch**: 30
- **minn**: 3
- **maxn**: 6
- **wordNgrams**: 3
- **loss**: Hierarchical Softmax

También particionamos el dataset de la misma forma, usaremos la misma seed para aleatorizar los datos y filtraremos las muestras que contengan un programa trivial.

Conjunto de datos	Exactitud	Tamaño del vocabulario
Programas sin normalizar	71.56 %	38236
Programas normalizados	86.49 %	3975
Programas + reportes	71.85 %	41589
Programas + reportes (normalizados)	99.92 %	5370

Cuadro 5.2: Resultados de la clasificación por status

Nuevamente como podemos ver en los resultados, que normalizar y pre procesar la información claramente mejora las representaciones que pueden aprender los modelos de FastText.

Si bien entrenar un modelo utilizando únicamente programas da buenos resultados, no hay suficiente información para distinguir apropiadamente las instancias de código que no pertenezcan a la clase "failed", en la siguiente imagen (figura 5.1) se muestra la matriz de confusión para el clasificador entrenado únicamente con código normalizado.

En esta matriz se puede apreciar que múltiples instancias de código son catalogadas erróneamente como "failed", esto se debe principalmente a la disparidad de la distribución de los datos (80 % de ellos corresponden a código "failed").

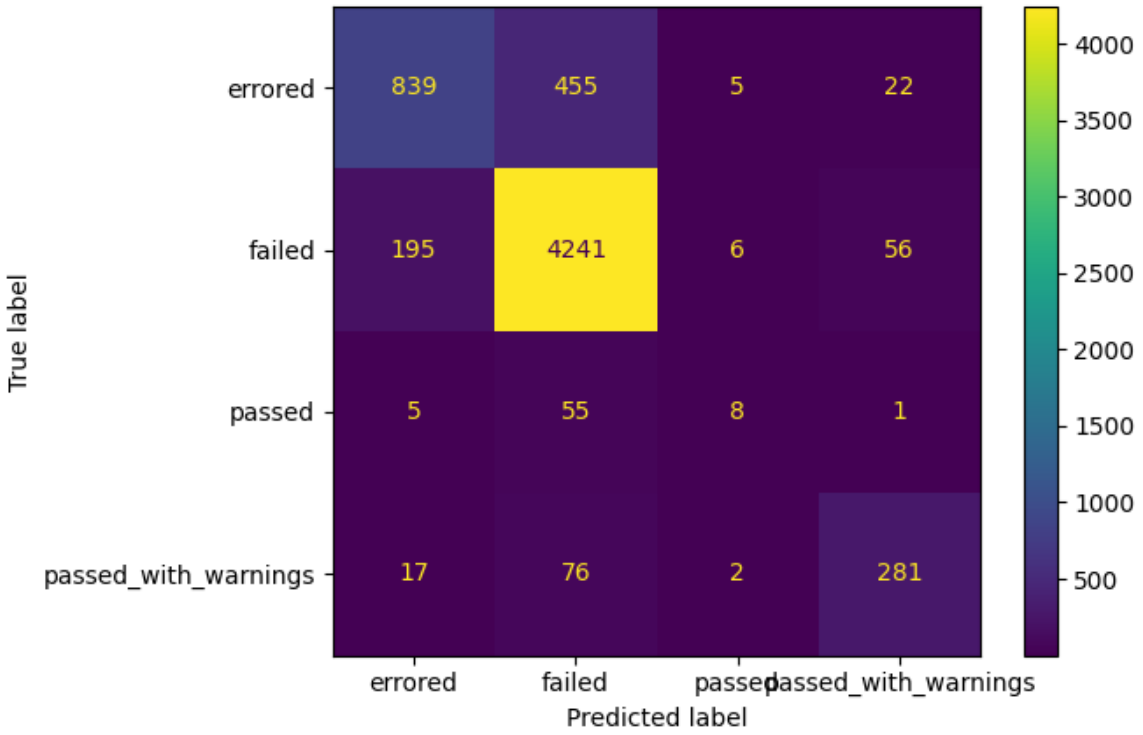


Figura 5.1: Matriz de confusión para el modelo entrenado sobre código normalizado.

Pero al considerar, además del código, los reportes asociados vemos que FastText es capaz de entrenar un modelo que puede clasificar perfectamente los programas por feedback, lo cual nuevamente nos indica que FastText puede trabajar directamente con toda esta información sin problemas.

5.3. Experimento: Visualización de Word Embeddings

En este experimento veremos los efectos que produce el preprocesamiento y la normalización de los datos a las estructuras geométricas de los word embeddings. Recordemos que los word embeddings poseen propiedades geométricas que codifican características del lenguaje como por ejemplo la semántica, de modo que dos palabras con significado similar están representadas con vectores cercanos.

Como tenemos muchos programas y los word embeddings adquieren valores en un rango acotado, para visualizar mejor los resultados, nos limitamos a trabajar solamente con el ejercicio que posee más muestras en nuestros datos, ó sea programas del ejercicio "Cartelitos óptimos". Este ejercicio cuenta con 1465 muestras (luego de filtrar las muestras triviales) de las cuales: 374 son "errored", 973 son "failed", 12 son "passed" y 106 son "passed_with_warnings".

Es razonable suponer que los resultados que expondremos en este experimento sean generales para los demás ejercicios, ya que todos los ejercicios de Mumuki para

el curso "Programación imperativa con JavaScript" tienen características similares.

Para obtener los word embeddings usaremos la función que provee FastText: *train_unsupervised*. Luego de realizar un grid search de hiper parámetros, estos son los que utilizaremos.

- **model**: Skip Gram
- **lr**: 0.05
- **dim**: 30
- **ws**: 5
- **epoch**: 15
- **minn**: 3
- **maxn**: 6
- **wordNgrams**: 3
- **loss**: Negative Sampling

Luego usando estos word embeddings construimos las representaciones vectoriales para cada muestra del ejercicio, usando la función que ofrecen los modelos de FastText *get_sentence_vector*, la cual fue explicada en la sección destinada a FastText.

Como en nuestro caso el word embedding resultante produce vectores de 30 dimensiones, las representaciones de los programas no pueden graficarse directamente.

Por eso debemos realizar primero una reducción dimensional de estos vectores, de modo que las representaciones sean graficables.

Usamos para ello dos algoritmos populares: PCA (Principal Component Analysis) que es un método popular para reducir la dimensionalidad de datos de muchas dimensiones y T-SNE (t-Distributed Stochastic Neighbor Embedding) el cual es un algoritmo de reducción de dimensionalidad popular en el procesamiento del lenguaje natural utilizado justamente, para la visualización de word embeddings.

Un problema del algoritmo T-SNE, es que depende fuertemente de sus parámetros, lo cual puede dar lugar a a la visualización estructuras que no existen en los datos originales. Por eso utilizaremos las heurísticas y parámetros recomendados que se proponen en la librería scikit-learn y además también compararemos con los resultados que produce PCA.

Similarmente a los experimentos realizados anteriormente, vamos a considerar cuatro variaciones de los datos del ejercicio "Cartelitos óptimos" (Programas sin normalizar, programas normalizados, programas + reportes sin normalizar y programas + reportes normalizados).

5.3.1. Resultados: Programas sin normalizar

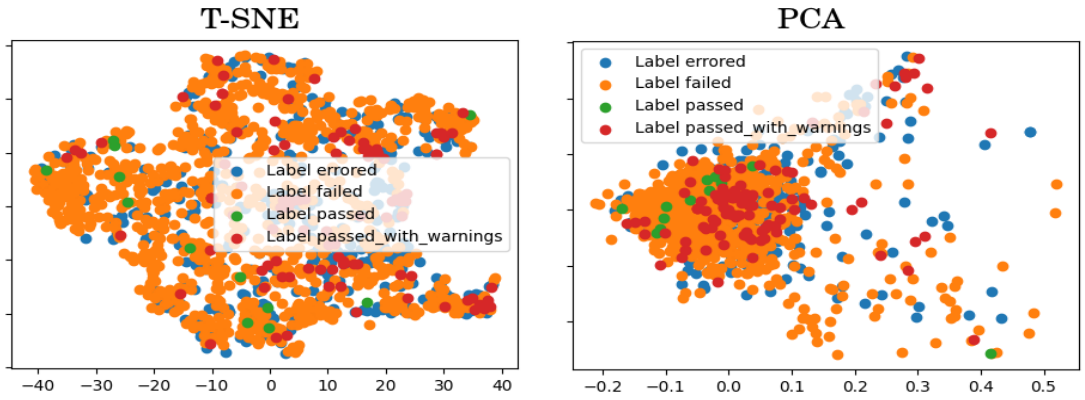


Figura 5.2: Representación gráfica de las representaciones vectoriales de los programas del ejercicio "Cartelitos óptimos", construida a partir de un word embedding entrenado sobre programas sin normalizar

Como podemos ver en la figura 5.2, las representaciones tanto de PCA como de T-SNE no presentan ningún tipo de estructura, todas las muestras están mezcladas y no se evidencian clusters razonables. Esto es consistente con los resultados del experimento anterior, ya que en este notamos que un modelo de FastText que sólo cuente con programas no tiene suficiente información para distinguir apropiadamente los programas que no pertenecen a la clase "failed".

5.3.2. Resultados: Programas y reportes sin normalizar

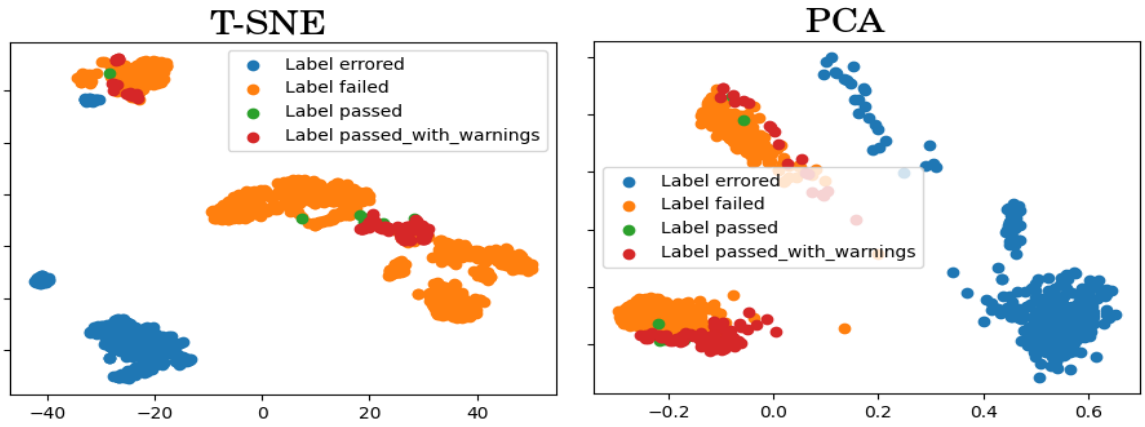


Figura 5.3: Representación gráfica de las representaciones vectoriales de los programas del ejercicio "Cartelitos óptimos", construida a partir de un word embedding entrenado sobre programas y reportes sin normalizar

La incorporación de reportes en el modelo de FastText produce una representación estructurada. Además hay que notar que las muestras de programas "errored" (las cuales son las que más pueden variar) no se mezclan con las muestras de otras categorías, la razón de esto en principio es que los reportes del compilador son muy distintos a los otros reportes, en donde se definen más de un caso de prueba.

Además los ejercicios catalogados como "errored" son los únicos que poseen reportes del compilador y no poseen reportes de testing ni de motor de evaluación de expectativas.

5.3.3. Resultados: Programas normalizados

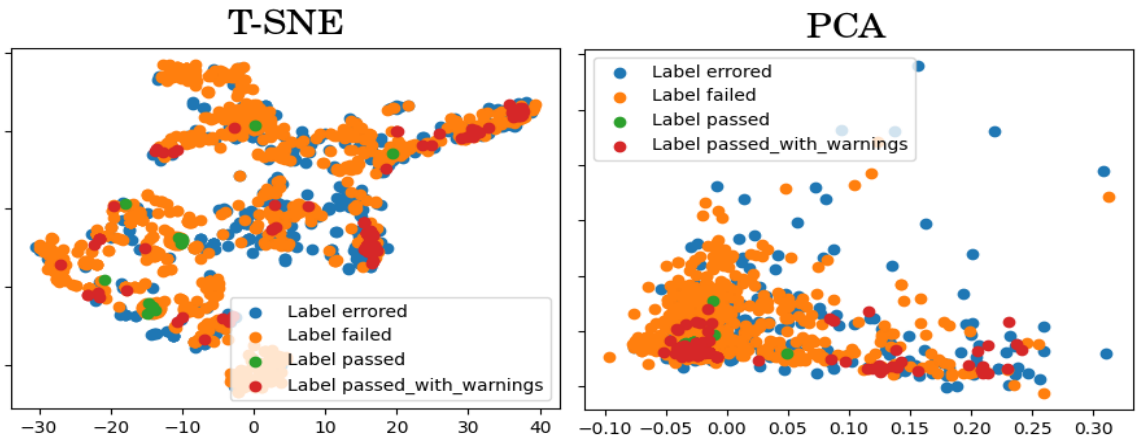


Figura 5.4: Representación gráfica de las representaciones vectoriales de los programas del ejercicio "Cartelitos óptimos", construida a partir de un word embedding entrenado sobre programas normalizados

Los resultados que obtiene el modelo usando programas normalizados son similares a los resultados obtenidos usando programas sin normalizar, la única diferencia es que esta vez hay menos variabilidad de los datos. Esto se debe a que luego de normalizar los programas, eliminamos gran parte de las diferencias (la cual era mucha, por lo que se puede ver en la reducción de los vocabularios en los experimentos) que no afecta a la semántica de estos.

Nuevamente, tal como se puede observar en la imagen, el modelo no es capaz de diferenciar apropiadamente los programas que no pertenecen a la clase "failed".

5.3.4. Resultados: Programas y reportes normalizados

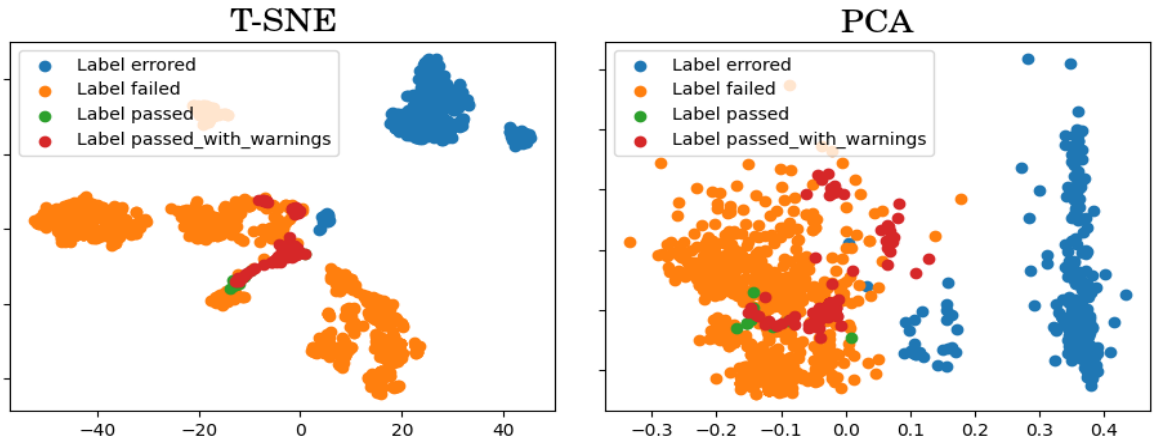


Figura 5.5: Representación gráfica de las representaciones vectoriales de los programas del ejercicio "Cartelitos óptimos", construida a partir de un word embedding entrenado sobre programas y reportes normalizados

Al considerar reportes, al igual que antes, el modelo puede diferenciar las distintas clases de programas que estamos considerando. Si observamos el gráfico T-SNE podemos observar que dentro de cada categoría se ve una estructura definida, a diferencia de la versión sin normalizar que si bien esta puede diferenciar las categorías, estas se mezclan más.

5.3.5. Metricas de clustering

Las imágenes mostradas en este experimento proveen una idea intuitiva de cómo afectan los datos a las estructuras geométricas de los word embedding resultantes. Un aspecto interesante de comparación es la calidad de los word embeddings considerando los programas de una misma clase. Ya que es deseable que nuestro modelo sea capaz de agrupar los programas en términos de los errores que estos contengan, nos interesa saber que tan estructurado es el word embedding dentro de una categoría en particular.

Para ello vamos a considerar los programas catalogados como "failed" y "errored", porque estos programas poseen la mayor cantidad de errores conceptuales y por ende son los más interesantes para los propósitos de este trabajo. Para determinar la calidad de la estructura de un word embedding vamos a recurrir a métricas de clustering: el índice de calinski-harabasz y el valor silhouette.

El silhouette es un método de interpretación y validación de consistencia dentro de datos agrupados. El valor silhouette es una medida de que tan similar es una muestra con su propio cluster respecto de los otros clusters. Los valores posibles de silhouette están acotados entre -1 y 1, donde los valores más altos indican más similaridad entre el objeto en cuestión y su propio cluster.

Definición: Sea x una muestra y sea $C(x)$ el cluster al que pertenece x, el silhouette score para x se define como:

$$s(x) = \frac{b(x) - a(x)}{\max(a(x), b(x))}$$

donde:

- $a(x)$: Es la distancia promedio de x respecto a las demás muestras de $C(x)$
- $b(x)$: Es la distancia promedio de x respecto a las muestras del cluster (distinto de $C(x)$) con el centroide más cercano a x

Definición: Para un conjunto de datos E de tamaño n_E, el cual fue agrupado en k clusters, el índice de calinski-harabasz ch está definido como la relación de la dispersión media entre clusters y la dispersión dentro del propio cluster. Es decir:

$$ch = \frac{tr(B_k)}{tr(W_k)} \times \frac{n_E - k}{k - 1}$$

donde tr denota la traza de una matriz y donde las matrices B_k y W_k están dadas por:

$$W_k = \sum_{q=1}^{k} \sum_{x \in C_q} (x - c_q)(x - c_q)^T$$

$$B_k = \sum_{q=1}^{k} n_q (c_q - c_E)(c_q - c_E)^T$$

Con C_q el conjunto de muestras en el cluster q, c_q el centroide del cluster q, c_E el centro del conjunto de datos E y n_q la cantidad de muestras en el cluster q.

Los valores posibles del índice de calinski-harabasz son los reales positivos, dando valores altos cuando los grupos son densos y están separados, lo cual coincide con el concepto usual de cluster. Un problema con el índice de calinski-harabasz es que no hay un rango de valores aceptables, por lo que no podemos saber a priori que tan buenos son los clusters para un conjunto arbitrario de datos.

Con estas metricas en mente, como mencionamos anteriormente, usaremos los programas catalogados como “failed” y ”errored”.

Uno de los problemas usuales de las métricas de clustering es que son sensibles al número de clusters definidos. Por ejemplo, es de esperar que sea más fácil realizar un clustering en 2 clases, que en un número mayor, por lo que las métricas suelen degradarse para un número elevado de clusters. Al realizar este tipo de análisis es conveniente tener presente una hipótesis inicial del número de clases que se espera encontrar en los datos, si es que este parámetro puede estimarse a partir de características del problema dado. En nuestro caso particular, por ejemplo, en el que esperamos poder clasificar problemas en términos de los errores que ellos contienen, podemos hacer las siguientes hipótesis. Claramente, habrá muchas formas distintas en las que un ejercicio puede ser realizado en forma errónea. Por otro lado, los ejercicios

son relativamente breves, y además no esperamos poder capturar todas las formas diferentes posibles. Con esto en mente, podríamos suponer que un número pequeño de clusters (2-3) puede tener buenos valores, pero no captura correctamente la estructura de los datos. Mientras que esperaríamos poder capturar un número mayor de clusters adecuadamente (4-5-6), mientras que valores mayores (7 o más) están mas allá de las posibilidades de representación del modelo.

Para realizar el clustering sobre ellos utilizaremos los vectores del word embedding y el algoritmo k-means el cual es sencillo y de uso general.

Caso	**k**	**Silhouette (promedio)**	**Calinski-Harabasz**
Programas sin normalizar	2	0.892	566
	3	0.487	749
	4	0.484	707
	5	0.378	723
	6	0.377	682
	7	0.377	689
Programas y reportes sin normalizar	2	0.748	3911
	3	0.407	2570
	4	0.412	2037
	5	0.325	1767
	6	0.301	1592
	7	0.294	1455
Programas normalizados	2	0.758	1984
	3	0.648	1751
	4	0.406	1569
	5	0.375	1449
	6	0.366	1375
	7	0.371	1362
Programas y reportes normalizados	2	0.605	523
	3	0.613	540
	4	0.614	606
	5	0.616	642
	6	0.437	585
	7	0.437	537

Cuadro 5.3: Métricas de clustering sobre programas catalogados como “failed”

Algo interesante que observamos en estos resultados 5.3 es que en los primeros tres casos la cantidad óptima de clusters es 2 y a medida que se agregan más clusters las métricas empeoran, mientras que en el último caso la cantidad óptima de clusters es 4 o 5.

No es razonable suponer que en los programas catalogados como “failed” sólo hay 2 o 3 clusters definidos ya que hay varias combinaciones posibles de errores en los programas.

Además si inspeccionamos estos clusters vemos que resultan ser muy generales y que contienen varios tipos de programas.

Lamentablemente no sabemos realmente cuántos clusters distintos hay en los datos, ya que si aumentamos la cantidad de clusters y los inspeccionamos, encontramos clusters muy pequeños que contienen programas raros y en los clusters más grandes aún se mezclan más de un tipo de programas. Sin embargo, definitivamente hay más de 3 clusters en los datos.

Como el modelo del caso cuatro obtiene mejores métricas cuando agrupamos en muchos clusters (más de 3), consideramos que este representa mejor la estructura real de los datos y por ende es el mejor modelo.

Caso	**k**	**Silhouette (promedio)**	**Calinski-Harabasz**
Programas sin normalizar	2	0.960	343
	3	0.889	577
	4	0.868	528
	5	0.870	491
	6	0.874	482
	7	0.871	488
Programas y reportes sin normalizar	2	0.801	820
	3	0.812	1840
	4	0.396	1642
	5	0.397	1427
	6	0.386	1343
	7	0.400	1319
Programas normalizados	2	0.731	698
	3	0.736	514
	4	0.574	521
	5	0.573	491
	6	0.476	489
	7	0.420	504
Programas y reportes normalizados	2	0.803	795
	3	0.661	966
	4	0.660	881
	5	0.525	858
	6	0.500	808
	7	0.412	807

Cuadro 5.4: Metricas de clustering sobre ejercicios catalogados como “errored”

Nuevamente en los programas catalogados como “errored”, vemos que el cuarto modelo tiene las mejoras métricas cuando consideramos muchos clusters (más de 3). Para este tipo de programas tampoco es razonable suponer que los datos están agrupados en 2 o 3 clusters, de hecho es más probable que en esta categoría existan más clusters que en los programas catalogados como “failed”, debido al gran número de posibilidades distintas que pueden ocasionar que un programa no compile.

Algo curioso en los resultados de la tabla 5.4 es que el modelo que trabaja solo con programas no normalizados tiene métricas muy buenas, pero por todos los resultados vistos hasta ahora seguimos concluyendo que el cuarto modelo es el mejor.

Capítulo 6

Combinación de reportes y programas

En los experimentos que presentamos en el capítulo anterior el modelo de FastText que se obtiene a partir de programas normalizados obtiene buenos resultados en general. Los reportes de testing asociados a los programas también aportan información significativa para el modelo de FastText, ya que al considerar los reportes vimos que el modelo resultante es capaz de diferenciar mejor los programas según su feedback. Por lo que resulta interesante considerar esta información para obtener las respectivas representaciones vectoriales de los programas.

En este capítulo exploraremos dos alternativas distintas para agregar los reportes asociados a los programas en el material de entrenamiento:

La primera consiste en considerar directamente el texto de los reportes, por lo que los agregamos directamente a los archivos de entrenamiento y de testeo de FastText, tal como lo hicimos en los experimentos.

En la segunda realizamos una featurizacion de los reportes, obteniendo vectores que codifican los resultados expuestos en ellos. Luego combinamos estos vectores con los vectores de programas que obtenemos a partir del word embedding y así obtenemos una nueva representación vectorial de programas que considera además del código sus correspondientes reportes.

De las dos alternativas mencionadas la primera es la más simple de llevar a cabo, ya que únicamente implica modificar el archivo de entrenamiento para que este contenga programas y sus respectivos reportes. Luego, como FastText puede trabajar sobre cualquier archivo de texto, podemos entrenar modelos sin problemas. Por otro lado, la segunda alternativa requiere pasos extras que no consideramos durante la primera alternativa, por lo que en la siguiente sección discutiremos en detalle la segunda alternativa.

6.1. Featurizacion de reportes

Para featurizar los reportes necesitamos codificar los resultados expuestos en ellos como vectores. Esta codificación tiene que ser independiente de la codificación que realiza FastText a la hora de entrenar un word embedding sobre los programas.

Al momento de pensar cómo codificar los reportes, nos dimos cuenta de que no todos los programas de un mismo ejercicio cuentan con las mismas test suite. Dentro de un mismo ejercicio la cantidad de casos de prueba puede variar. Esto se debe a que en Mumuki con el paso del tiempo se detectó que algunos ejercicios tenían un criterio erróneo o muy acotado para considerar programas como correctos y la solución que adoptaron en Mumuki para este problema fue siempre modificar o agregar nuevos casos de prueba que refuercen el criterio de aceptación de los ejercicios. Por lo que si bien dentro de un ejercicio podemos encontrar programas con distinto número de casos de prueba, estos resultan ser subconjuntos de la versión más actual de la test suite del ejercicio.

Este detalle no afecta significativamente a los resultados mostrados en los experimentos, porque en ellos cuando procesamos un caso de prueba agregamos también el nombre respectivo del caso de prueba, el cual lo diferencia de los otros casos. Otra diferencia posible entre los reportes de un mismo ejercicio es el orden en que se definen los casos de prueba en la test suite, pero al realizar nuevamente los experimentos nos dimos cuenta que esto tampoco impacta significativamente en los resultados mostrados.

Para codificar los reportes en vectores, de un ejercicio particular, debemos entonces saber cuántos casos de pruebas distintos existen en la test suite más reciente del ejercicio. También debemos establecer un orden canónico para los casos de prueba de modo que en la codificación, cada dimensión del vector resultante haga referencia al mismo caso de prueba.

Para saber cuántos casos de prueba distintos hay en los datos, simplemente utilizamos expresiones regulares para extraer los casos de prueba de los reportes y construimos un conjunto con ellos. Luego, para establecer un orden canónico en la codificación de los casos de prueba, simplemente usaremos el orden alfabético de los nombres de los casos de prueba.

Sean n_{test}, n_{expc} y n_{comp} ($n_{\text{comp}} = 2$) la cantidad de casos de casos de prueba hallados en los reportes de testing, de evaluación de expectativas y del compilador respectivamente.

Si bien el compilador no define casos de prueba, este sólo exhibe dos tipos de errores posibles: “ReferenceError” y “SyntaxError”. Por ello definimos $n_{\text{comp}} = 2$

Definimos, para cada programa, un vector de reportes de $(n_{\text{test}} + n_{\text{expc}} + n_{\text{comp}})$ dimensiones, donde en cada dimensión codificaremos el resultado de los correspondientes casos de prueba, respetando el orden canónico, con alguno de los siguientes valores:

- -1: Utilizamos este valor para indicar que el caso de prueba no esta definido. Recordemos que ningún programa cuenta con los tres reportes al mismo tiempo (test-suite, motor evaluación de expectativas y compilador) y también que las test suite de un programa pueden ser un subconjunto de una test suite más actual.
- 0: Utilizamos este valor para indicar un resultado negativo en el caso de prueba.
- 1: Utilizamos este valor para indicar un resultado en el caso de prueba.

De este modo podemos obtener una codificación vectorial de reportes para los programas de un ejercicio particular.

Por otro lado para obtener las representaciones vectoriales de los programas usaremos un modelo no supervisado de FastText, para obtener un word embedding, con los siguientes hiper parámetros:

- **model**: Skip Gram
- **lr**: 0.05
- **dim**: 30
- **ws**: 5
- **epoch**: 15
- **minn**: 3
- **maxn**: 6
- **wordNgrams**: 3
- **loss**: Negative Sampling

Ahora para cada programa tenemos dos vectores independientes, uno para el código y otro para los reportes, para combinarlos en un único vector utilizaremos un autoencoder.

Un autoencoder es una arquitectura particular de redes neuronales, la cual consiste de dos partes principales: un encoder y un decoder. Los autoencoders principalmente se utilizan para aprender representaciones eficientes de los datos. Originalmente surgieron como una generalización del algoritmo de reducción de dimensionalidad PCA.

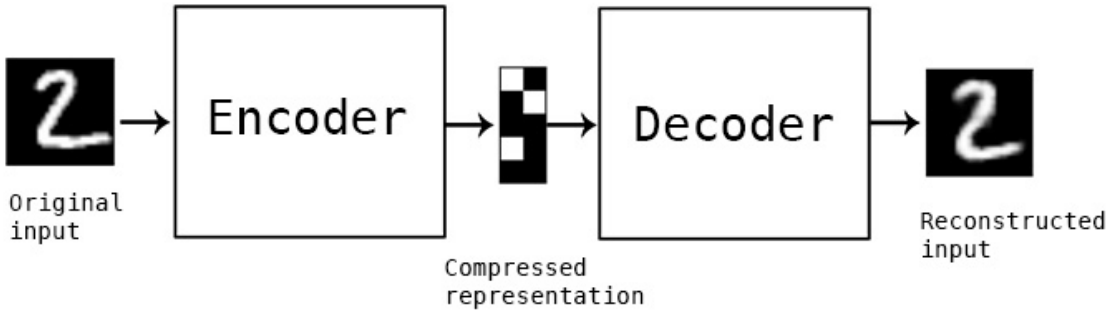


Figura 6.1: Esquema usual de un autoencoder

Una ventaja de los autoencoders es que no requieren datos etiquetados para su entrenamiento ya que la propia entrada sirve como etiqueta.

Gracias a que los autoencoders pueden aprender representaciones eficientes de los datos, podemos utilizar uno para que aprenda una representación compacta de la concatenación de los vectores de programas y reportes de modo que la representación resultante combine apropiadamente estos vectores.

Implementaremos nuestro autoencoder usando la librería Keras, con la siguiente arquitectura:

Encoder:

- Layer simbólica de entrada de $(30 + n_{\text{test}} + n_{\text{expc}} + 2)$ neuronas
- Layer de BatchNormalization
- Layer densa de 256 neuronas, con función de activación RELU
- Layer de BatchNormalization
- Layer densa de 128 neuronas, con función de activación RELU
- Layer de BatchNormalization
- Layer densa de 35 neuronas, con función de activación RELU

Decoder:

- Layer densa de 128 neuronas, con función de activación RELU
- Layer de BatchNormalization
- Layer densa de 256 neuronas, con función de activación RELU
- Layer de BatchNormalization
- Layer densa de $(30 + n_{\text{test}} + n_{\text{expc}} + n_{\text{comp}})$ neuronas, con función de activación RELU.

Para el caso del ejercicio "Cartelitos óptimos", los vectores de reportes tienen 16 dimensiones y como nuestros vectores de código tienen 30 dimensiones, los vectores que usaremos para entrenar el autoencoder son de 46 dimensiones. Los vectores de entrada son simplemente la concatenación de a pares de los vectores de código y de reportes.

Los parámetros de entrenamiento que utilizaremos en el autoencoder son los siguientes:

- **Optimizador**: Adam
- **Learning rate**: 0.0015
- **Función de costo**: Error absoluto.
- **Métrica de validación**: Error cuadrado medio.
- **Épocas de entrenamiento**: 100
- **Tamaño del batch**: 32

A continuación repetiremos el experimento de visualización de word embedding pero esta vez utilizando los vectores que obtenemos a partir del encoder.

Al igual que antes usaremos las muestras del ejercicio "Cartelitos óptimos" (el ejercicio con más muestras entre los datos disponibles). Luego construimos los vectores de códigos y reportes, para cada muestra, tal como se describió anteriormente y con cada par formamos nuevos vectores los cuales constituyen nuestro conjunto de datos en este experimento.

Para entrenar el autoencoder debemos aleatorizar nuestro conjunto de datos y reservar 20 % de ellos como conjunto de validación. Luego entrenamos usando los

parámetros mencionados anteriormente. Los resultados del entrenamiento están en la figura 6.2:

Luego recuperamos los pesos del autoencoder en la época que obtuvo las mejores métricas de validación y extraemos su encoder con el cual crearemos las representaciones vectoriales finales de las muestras del ejercicio.

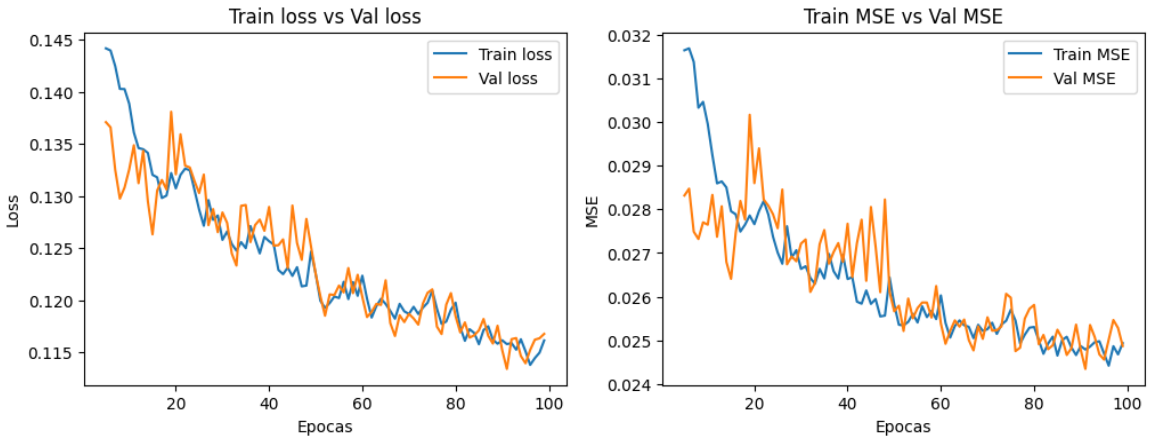


Figura 6.2: Resultados del entrenamiento del autoencoder. A la izquierda tenemos los resultados del error absoluto de entrenamiento y validación, y a derecha los resultados del error cuadrado medio de entrenamiento y validación.

Finalmente reducimos dimensionalmente estos vectores resultantes usando PCA y T-SNE (tal como se hizo en el experimento original) y los graficamos los vectores resultantes obteniendo las siguientes imágenes (figura 6.3):

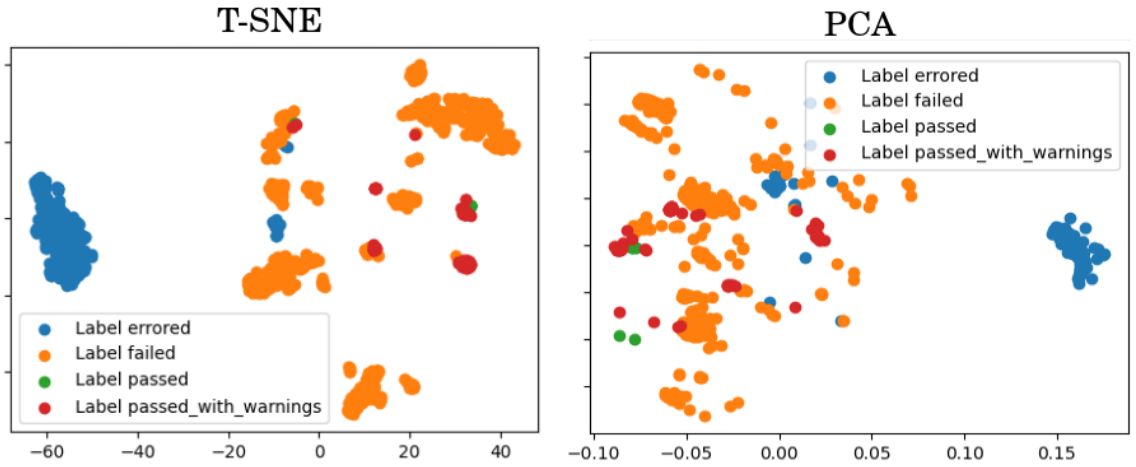


Figura 6.3: Representación gráfica de las representaciones vectoriales de los programas del ejercicio "Cartelitos Óptimos", obtenidas por medio de un autoencoder

Al igual que antes calcularemos métricas de clustering sobre estos vectores. Puntualmente en aquellos vectores que provienen de programas catalogados como "failed" y "errored". La tabla 6.1 muestra las metricas obtenidas.

Clase	k	Silhouette (promedio)	Calinski-Harabasz
Programas “failed”	2	0.463	702
	3	0.485	595
	4	0.540	592
	5	0.562	637
	6	0.574	626
	7	0.581	619
Programas “errored”	2	0.852	1084
	3	0.685	1188
	4	0.673	992
	5	0.669	880
	6	0.678	848
	7	0.528	836

Cuadro 6.1: Metricas de clustering del modelo FastText + Autoencoder

Los resultados que observamos en la tabla anterior presentan una mejora en las metricas de clustering cuando consideramos más de cuatro clusters, respecto de entrenar un modelo de FastText sobre programas y reportes. Recordemos que las metricas obtenidas por este ultimo son:

Clase	k	Silhouette (promedio)	Calinski-Harabasz
Programas “failed”	2	0.605	523
	3	0.613	540
	4	0.614	606
	5	0.616	642
	6	0.437	585
	7	0.437	537
Programas “errored”	2	0.803	795
	3	0.661	966
	4	0.660	881
	5	0.525	858
	6	0.500	808
	7	0.412	807

Cuadro 6.2: Métricas de clustering del modelo FastText entrenado sobre programas y reportes normalizados

6.2. Comparación de modelos

Las métricas de clustering no son suficientes para determinar si un modelo ofrece una mejor definición de similaridad de programas que otro modelo. Cuando realizamos una inspección manual de los programas dentro de los clusters obtenidos por ambos

modelos, los dos ofrecen un agrupamiento de programas razonable.

También se intentó determinar si los clusters de ambos modelos caracterizan un tipo de programa en particular, pero esto es sumamente difícil. Principalmente porque no sabemos realmente cuántos tipos de programas hay en los datos, también porque en la mayoría de los casos los clusters terminan abarcando más de un tipo de programa.

Tampoco sabemos qué tanto influye en la definición de similaridad de programas las métricas de clustering obtenidas por un modelo. Si bien es razonable suponer que están relacionadas no debemos ignorar que cuando buscamos programas similares solamente nos limitamos a las muestras más cercanas, mientras que las métricas de clustering están determinadas por todas las muestras como conjunto.

En las siguientes secciones intentaremos realizar una comparación de pros y contras de los mejores modelos obtenidos.

6.2.1. Modelo: FastText sobre programas y reportes

La principal ventaja de este modelo es su simplicidad, ya que utilizamos toda la información directamente en un solo modelo sin la necesidad de procesamientos adicionales (además de la normalización de los datos).

Pero corremos el riesgo de afectar la similaridad de los programas al concatenar código y sus respectivos reportes, ya que estamos combinando en una misma muestra dos textos de naturaleza distinta y con reglas gramaticales distintas. También hay que recordar que los casos de prueba de los ejercicios evolucionan con el tiempo y esto puede afectar la métrica de similaridad para programas de distintos periodos.

Para atestiguar esto consideremos los siguientes tres programas con sus correspondientes reportes normalizados, los cuales denotaremos por (p_1, r_1), (p_2, r_2) y (p_3, r_3) respectivamente

Listing 6.1: Programa p_1 y reportes r_1

Codigo:

```
function escribirCartelitoOptimo ( arg0 , arg1 , arg2 ) {
    if ( longitud ( arg1 + arg2 ) > 15 ) {
        return escribirCartelito ( arg0 , arg1 , arg2 , true )
    }
    else {
        return escribirCartelito ( arg0 , arg1 , arg2 , false )
    }
}
```

Reportes:

```
TESTING
escribe un cartelito largo passed
escribe un cartelito corto passed
EXPECTATIONS
Declares:=escribirCartelitoOptimo passed
Uses:escribirCartelito passed
Not:UsesIf failed
Not:Declares:escribirCartelito passed
CNA
```

Listing 6.2: Programa p_2 y reportes r_2

```
Codigo:
    function escribirCartelitoOptimo ( arg0 , arg1 , arg2 ) {
        if ( longitud ( "arg1" ) ) + ( longitud ( "arg2" ) ) > 15 {
            return escribirCartelito ( arg0 , arg1 , arg2 , true )
        }
        else {
            return escribirCartelito ( arg0 , arg1 , arg2 , false )
        }
    }

Reportes:
    TNA
    ENA
    SYNTAX_ERROR  Unexpected token {
```

Listing 6.3: Programa p_3 y reportes r_3

```
Codigo:
    function valorEnvido ( arg0 ) {
        if ( arg0 >= 1 && =< 7 ) {
            return arg0
        }
        else ( arg0 >= 10 && =< 12 ) {
            return 0
        }
    }

Reportes:
    TNA
    ENA
    SYNTAX_ERROR  Unexpected token =
```

Los programas p_1 y p_2 son muestras del ejercicio "Cartelitos óptimos" mientras que el programa p_3 es una muestra del ejercicio "Envido". En la siguiente tabla mostraremos las distancias que hay entre estos programas

$d(p_1, p_2)$	$d(p_1, p_3)$	$d(p_2, p_3)$
0.08672689	0.15630527	0.17255251

Cuadro 6.3: Distancias considerando únicamente programas

Al considerar solamente código, como podemos apreciar en 6.3, la distancia entre p1 y p2 es baja, principalmente porque ambos programas corresponden al mismo ejercicio y tienen muchas similitudes sintácticas. Pero debemos notar que ambos programas tienen clases de feedback distintas, uno de ellos pertenece a la clase "passed" y el otro a la clase "errored". También notemos que la distancia es más alta entre programas que no son del mismo ejercicio.

Sin embargo al considerar los reportes, la distancia entre p_1 y p_2 es más alta, lo cual es esperable por la diferencia en sus reportes. Pero la distancia entre p_2 y p_3, en comparación con la tabla 6.3, es mucho menor. Esto es un efecto negativo de combinar los reportes y códigos en la misma muestra, ya que puede producir vectores cercanos para programas de distinta semántica sólo porque coinciden en los reportes.

$d(p_1 + r_1, p_2 + r_2)$	$d(p_1 + r_1, p_3 + r_3)$	$d(p_2 + r_2, p_3 + r_3)$
1.4015073	1.5270994	0.1670435

Cuadro 6.4: Distancias considerando programas y sus respectivos reportes

A pesar de este detalle, en general el modelo agrupa razonablemente códigos similares. Por ejemplo consideremos los siguientes códigos, los cuales no fueron usados durante el entrenamiento del modelo.

Normalizamos los códigos que mostraremos a continuación para poder comparar los resultados más fácilmente.

Listing 6.4: Ejemplo de programas similares a un programa "failed"

```
Programa de entrada:
    function esFinDeSemana ( arg0 ) {
        return ( arg0 === "sabado" || arg0 === "domingo" ) ;
    }
    function estaCerrado ( arg0 , arg1 , arg2 ) {
        return ( arg0 ||
                 esFinDeSemana ( arg1 ) ||
                 arg2 !== dentroDeHorarioBancario ( arg2 ) ) ;
    }

Programas mas cercanos:
    function estaCerrado ( arg0 , arg1 , arg2 ) {
        return ( arg0 ||
                 esFinDeSemana ( "arg1" ) ||
                 arg2 !== dentroDeHorarioBancario ( arg2 ) ) ;
    }
    function esFinDeSemana ( arg0 ) {
        return ( arg0 === "sabado" || arg0 === "domingo" ) ;
    }
```

```
    function esFinDeSemana ( arg0 ) {
        return ( arg0 === 'sabado' ) || ( arg0 === 'domingo' )
    }
    function estaCerrado ( arg0 , arg1 , arg2 ) {
        return arg0 ||
               esFinDeSemana ( arg1 ) ||
               dentroDeHorarioBancario !== ( arg2 ) ;
    }
```

Listing 6.5: Ejemplo de programas similares a un programa "errored"

```
Programa de entrada:
    function longuitudNombreCompleto ( arg0 , arg1 ) {
        return longitud ( "arg0" + "arg1" ) + 1 ;
    }

Programas mas cercanos:
    function LongitudNombreCompleto ( arg0 , arg1 ) {
        return "arg0" + "arg1" + 1 ;
    }
```

```
    function LongitudNombreCompleto ( arg0 , arg1 , arg2 ) {
```

```
        return arg2 ( "arg0" + 1 + "arg1" ) ;
    }
```

Como podemos ver en los ejemplos anteriores, el modelo agrupa razonablemente los programas de entrada incluso en programas catalogados como "errored"(los cuales tienen la mayor probabilidad de variabilidad).

6.2.2. Modelo: FastText + Autoencoder

La principal ventaja de este enfoque es que, según las métricas de clustering, obtenemos las mejores representaciones vectoriales de los datos. Además de que la forma en que agregamos los reportes al modelo no beneficia particularmente las muestras de distintos programas con reportes similares como lo hace FastText.

Pero lamentablemente este modelo presenta múltiples desventajas a la hora de implementarlo en Mumuki. Principalmente porque la vectorización de reportes depende de un ejercicio puntual y esta codificación sólo tiene sentido para instancias de ese ejercicio, por lo que esto implica entrenar un autoencoder por ejercicio.

Otro problema es que en nuestra solución, dado un vector de un ejercicio x, buscamos los vectores más cercanos en los datos disponibles por lo que necesitamos guardar las representaciones vectoriales de los datos de entrenamiento. Pero si eventualmente necesitamos actualizar la codificación de los reportes porque en Mumuki se actualizó la test suite del ejercicio x, entonces los vectores guardados para este ejercicio ya no son útiles.

Mientras que utilizando el modelo que solo depende de FastText, podemos entrenar un único modelo para todos ejercicios y en caso de que se deba actualizar la test suite, los vectores calculados anteriormente aún son significativos por lo que se pueden seguir usando por más tiempo.

A continuación mostraremos algunos ejemplos de programas cercanos usando este enfoque:

Listing 6.6: Ejemplo de programas similares a un programa "failed"

```
Programa de entrada:
    function escribirCartelitoOptimo ( arg0 , arg1 , arg2 ) {
        return escribirCartelito (
            arg0 ,
            arg1 ,
            arg2 ,
            longitud ( "arg1" + " " + "arg2" ) > 15
        )
    }

Programas mas cercanos:
    function escribirCartelitoOptimo ( arg0 , arg1 , arg2 ) {
        return escribirCartelito (
            arg0 ,
            arg1 ,
            arg2 ,
            "arg1" + "arg2" > 15
        )
    }
```

```
    function escribirCartelitoOptimo ( arg0 , arg1 , arg2 ) {
        return escribirCartelito ( arg0 , arg1 , arg2 )
            || longitud ( arg1 ) + longitud ( arg2 ) >= 15
    }
```

Listing 6.7: Ejemplo de programas similares a un programa "errored"

```
Programa de entrada:
    function escribirCartelitoOptimo ( arg0 , arg1 , arg2 ) {
        {
            longitud ( arg1 + arg2 > 15 )
            return escribirCartelito (
                arg0 ,
                arg1 ,
                arg2 ,
                quiereCartelCorto
            ) ;
        }
        else {
            return escribirCartelito ( arg0 , arg1 , arg2 , false ) ;
        }
    }

Programas mas cercanos:
    function escribirCartelitoOptimo ( arg0 , arg1 , arg2 ) {
        if ( longitud ( arg1 , arg2 ) > 15 )
            return escribirCartelito ( arg0 , arg1 , arg2 ) ; }
        else {
            return escribirCartelito ( arg0 , arg2 ) ;
        }
```

```
    function escribirCartelitoOptimo ( arg0 , arg1 , arg2 ) {
        if longitud ( arg1 + arg2 ) > 15 {
            return escribirCartelito (
                arg0 ,
                arg1 ,
                arg2 ,
                quiereCartelCorto
            )
        else {
            return escribirCartelito (
                arg0 ,
                arg1 ,
                arg2 ,
                quiereCartelLargo
            )
        }
    }
```

6.3. Detalles de implementación en Mumuki

Como resultado de la comparación realizada en este capitulo se decidió implementar en Mumuki el modelo que utiliza únicamente FastText. De este modo podemos utilizar el mismo modelo para los diversos ejercicios del curso "programación imperativa en JavaScript".

Si bien este modelo puede determinar una mayor similaridad entre programas con reportes similares, vimos que en general los programas mas cercanos son coherentes. Además como este modelo integra los datos en una cadena de texto, es posible simplemente omitir los reportes si se desea y el modelo aún puede funcionar sin problemas.

Por otro lado, por las características arquitectónicas de Mumuki, la implementacion de nuestro modelo en el sistema simplemente consiste de introducir una interfaz para interactuar con este en un contenedor de Docker, para luego ser utilizado por un servicio de la pagina. Durante la etapa de experimentación de este trabajo se desarrollaron múltiples rutinas para entrenar e interactuar con modelos de FastText, las cuales posteriormente se utilizaron para la implementacion de este trabajo en Mumuki.

De este modo, Mumuki puede calcular las metricas de similaridad para una solución particular de un ejercicio particular respecto de las soluciones propuestas anteriormente en ese ejercicio y así ofrecer las consultas realizadas en los programas más similares.

Capítulo 7

Conclusiones

En este trabajo hemos definido una métrica de similaridad de programas en términos de distancias entre vectores de programas. Para la obtención de estos vectores entrenamos un modelo de inteligencia artificial, puntualmente un modelo de la librería FastText.

Para favorecer el entrenamiento de nuestro modelo y, a su vez, aumentar la representatividad de nuestra métrica de similaridad hemos aplicado distintas técnicas de normalización sobre los programas de entrenamiento.

Observamos que la normalización de programas reduce drásticamente la variabilidad presente en estos. Reduciendo el tamaño del vocabulario de 45000 tokens distintos a tan solo 4535.

También comprobamos mediante experimentos que la normalización de programas efectivamente favorece el entrenamiento de un modelo de inteligencia artificial; Dando como resultado representaciones vectoriales con una estructura geométrica más definida, que producen mejores métricas de clustering.

Lamentablemente no contamos con una forma efectiva de medir la correlación entre nuestra métrica de similaridad de programas y las métricas de clustering que obtiene nuestro modelo. Esto se debe principalmente a que no podemos evaluar automáticamente si los programas más cercanos a uno dado son realmente similares.

Sin embargo, observamos empíricamente (mediante anotación manual) que la métrica de similaridad que produce nuestro modelo parece representar razonablemente la similaridad de programas en muchos casos.

7.1. Trabajo futuro

Nuestra métrica de similaridad de programas es representativa para la mayoría de los casos, pero aun hay aspectos a mejorar que debemos considerar.

Por ejemplo, la efectividad de nuestro modelo para asociar programas similares depende fuertemente de la diversidad de los programas disponibles. Esto en sí no es un problema grave ya que las soluciones para un mismo problema tienden a "converger" a un programa común. Por lo cual la probabilidad de que nuestra métrica sea útil es alta.

Sin embargo, no debemos ignorar que existen situaciones en que el modelo no resulta efectivo. Por ejemplo, si un estudiante utiliza nuestro modelo con un programa

altamente original que no compila, es muy probable que no contemos con un programa que sea similar para ofrecer. Esta situación también puede suceder cuando en un ejercicio particular contamos con muy pocas muestras disponibles, lo cual es muy probable si el ejercicio es nuevo en Mumuki.

Otro aspecto que debemos considerar es cómo aprovechar toda la información relacionada a los reportes de los programas. Si bien en este trabajo exploramos dos alternativas para aprovechar los reportes, ambas alternativas tienen inconvenientes que dificultan su implementación. Recordemos que en el enfoque de utilizar reportes y código en la misma entrada del modelo de FastText puede favorecer incorrectamente a programas distintos que comparten reportes similares; Mientras que el enfoque de utilizar un autoencoder para featurizar los reportes tiene el problema de que esta featurizacion solo tiene sentido para un ejercicio particular, lo cual prohíbe usar el mismo modelo en distintos ejercicios.

Consideramos que merece la pena explorar distintos modelos de inteligencia artificial para la obtención de las representaciones vectoriales de los programas y analizar sus propiedades geométricas. Como hemos observado en este trabajo a medida que refinábamos nuestro conjunto de entrenamiento, nuestros vectores constituían una estructura más definida.

Quizás si continuamos refinando nuestras representaciones podríamos construir clusters que caractericen errores puntuales en los programas, de modo tal que el modelo sea útil incluso si el ejercicio es nuevo en la plataforma.

Pero para ello necesitamos que nuestro modelo considere mas información que puede encontrarse en los programas, como por ejemplo el sistema de tipos.

Javascript, al ser un lenguaje de scripting, tiene un sistema de tipos flexible. Esta propiedad es deseable en un lenguaje de programación porque permite escribir programas en menos tiempo, pero a su vez esta propiedad resulta negativa para alguien poco experimentado en la programación. Ya que el compilador de JavaScript no advertirá los posibles errores relacionados a los tipos. Por lo que es muy probable que un estudiante no detecte rápidamente que esta cometiendo un error de tipos.

Para exponer esta información en nuestros datos de entrenamiento podemos recurrir a la construcción de AST (Abstract Syntax Tree) o de arboles de parsing de los programas.

En trabajos recientes se ha demostrado la efectividad de las redes neuronales basadas en arboles para diversas tareas que involucran programas, como por ejemplo generación de código [12, 23], en la detección de vulnerabilidades y software malicioso [1, 21] y en detección de "code clones" [26].

Una posible continuación de este trabajo es considerar los arboles de parsing o AST para entrenar una nueva red neuronal basada en arboles a partir de arboles obtenidos de los programas que compilan. De este modo se puede obtener un modelo con representaciones mas enriquecidas de programas para el 80 % de los casos (recordemos que esta es la proporción de programas que compilan), mientras que para el resto de los casos se puede seguir utilizando el modelo propuesto en este trabajo.

Referencias

[1] Tamás Aladics, Péter Hegedűs, and Rudolf Ferenc. An ast-based code change representation and its performance in just-in-time vulnerability prediction. *International Conference on Software and Data Technologies*, 2023.

[2] Uri Alon, Omer Levy, and Eran Yahav. Code2seq: Generating sequences from structured representations of code. *International Conference on Learning Representations*, 2018.

[3] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning distributed representations of code. *Proceedings ACM Programming Languages*, 3(POPL):40:1–40:29, January 2019.

[4] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. Optics: Ordering points to identify the clustering structure. *SIGMOD Record*, 28(2):49–60, June 1999.

[5] Dor Bank, Noam Koenigstein, and Raja Giryes. Autoencoders. *arXiv*, abs/2003.05991, 2020.

[6] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomás Mikolov. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146, 2016.

[7] Tony Cai and Rong Ma. Theoretical foundations of t-sne for visualizing high-dimensional clustered data. *Journal of Machine Learning Research*, 23:301:1–301:54, 2022.

[8] Tadeusz Caliński and Joachim Harabasz. A dendrite method for cluster analysis. *Communications in Statistics*, 3(1):1–27, 1974.

[9] David Charte, Francisco Charte, María José del Jesús, and Francisco Herrera. A showcase of the use of autoencoders in feature learning applications. In *From Bioinspired Systems and Biomedical Applications to Machine Learning*, pages 412–421. Springer International Publishing, 2019.

[10] Dorin Comaniciu and Peter Meer. Mean shift: a robust approach toward feature space analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(5):603–619, 2002.

[11] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, page 226–231. AAAI Press, 1996.

[12] Xue Jiang, Zhuoran Zheng, Chen Lyu, Liang Li, and Lei Lyu. Treebert: A tree-based pre-trained model for programming language. *Uncertainty in Artificial Intelligence*, pages 54–63, 2021.

[13] Ian Jolliffe and Jorge Cadima. Principal component analysis: A review and recent developments. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 374, April 2016.

[14] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomás Mikolov. Bag of tricks for efficient text classification. *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics*, 2:427–431, 2017.

[15] Diederik P. Kingma and Max Welling. Auto-encoding variational bayes. In Yoshua Bengio and Yann LeCun, editors, *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014.

[16] Pivotal Labs. Jasmine: Behaviour-driven javascript. `https://jasmine.github.io/`, 2010.

[17] Tomas Mikolov, Kai Chen, Gregory Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *Proceedings of Workshop at ICLR*, 2013, January 2013.

[18] Tomás Mikolov, Ilya Sutskever, Kai Chen, Gregory Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26, 2013.

[19] JetBrains Research. AST miner. `https://github.com/JetBrains-Research/astminer`, 2018.

[20] Peter J. Rousseeuw. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20:53–65, 1987.

[21] Gili Rusak, Abdullah Al-Dujaili, and Una-May O'Reilly. AST-based deep learning for detecting malicious PowerShell. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, oct 2018.

[22] Magnus Sahlgren. The distributional hypothesis. *The Italian Journal of Linguistics*, 20:33–54, 2008.

[23] Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. Treegen: A tree-based transformer architecture for code generation. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34:8984–8991, 2019.

[24] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-SNE. *Journal of Machine Learning Research*, 9:2579–2605, November 2008.

[25] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[26] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. Detecting code clones with graph neural networkand flow-augmented abstract syntax tree. *IEEE 27th*

International Conference on Software Analysis, Evolution and Reengineering, 2020.

Los abajo firmantes, miembros del Tribunal de evaluación de tesis, damos fe que el presente ejemplar impreso se corresponde con el aprobado por este Tribunal.