UNIVERSITAT DE BARCELONA

Trabajo de Final de Grado

GRADO EN INGENIERIA INFORMÁTICA

Facultad de Matemáticas e Informática
Universidad de Barcelona

EVALUACIÓN AUTOMÁTICA DE PROGRAMAS EN PYTHON A PARTIR DE ÁRBOLES DE SINTAXIS ABSTRACTA

Estela Pérez Bassedas

Director: Daniel Ortiz Martínez
Realizado a: Departamento de
Matemáticas e Informática

Barcelona, 13 de junio de 2023

Agradecimientos

Quiero expresar mi más sincero agradecimiento a todas aquellas personas que han sido clave para la realización de mi trabajo de final de grado. En primer lugar, quisiera agradecer a mi tutor Daniel Ortiz Martínez por su inestimable colaboración y disposición constante a lo largo de todo el proyecto. Su conocimiento y experiencia han sido clave para enfocar el proyecto de manera efectiva, así como para poder disponer de una gran cantidad de datos para llevar a cabo mejores pruebas y resultados.

También quiero aprovechar esta oportunidad para extender mi gratitud a mi familia y amigos, quienes han sido mi apoyo durante esta etapa hasta, actualmente, llegar a la entrega de este trabajo final de grado.

Resumen

Este trabajo de final de grado se centra en la posible automatización en la evaluación de ejercicios en Python mediante árboles de sintaxis abstracta (AST). Nuestra hipótesis se basa en que el AST procedente del ejercicio de un estudiante se considerará correcto de forma directamente proporcional a su semejanza al AST de referencia, es decir, al AST del código del profesor. Pudiendo usar más de un código de referencia, para obtener resultados más precisos. Hemos cuantificado esta similitud mediante la distancia entre los AST y calculado la correlación respecto a la nota puesta por un corrector humano para realizar un estudio de esos resultados.

Una de las etapas previas al cálculo de la distancia y bastante relevante en este trabajo, consiste en el "preproceso" del AST con el fin de obtener una cualificación que esté más correlacionada con mejores resultados perfilando ciertos factores.

La investigación nos lleva a la posible reducción de la carga de trabajo para un corrector humano, pero con la necesidad de usar esta técnica junto a otras para refinar la cualificación y que llegue a ser adecuada para automatizar la evaluación.

Índice

1 Introducción

El proyecto

En estudios relacionados con las ciencias de la computación, los alumnos se encuentran a diario con tareas de programación para poner en práctica los conocimientos adquiridos en sus clases. Estas tareas pueden ir desde la creación de un programa simple de operaciones entre dos parámetros de entrada, en un nivel introductorio y básico, a conseguir una función más compleja, en un nivel más avanzado.

Estas tareas se evalúan mayoritariamente mediante pruebas automatizadas simples, donde se compara la salida generada por el estudiante, con una solución proporcionada por el evaluador y se valora el número de coincidencias correctas para determinar la calificación final. Sin embargo, únicamente con esta consideración, valoraríamos negativamente una solución proporcionada que no genere ninguna salida, sin calificar el contenido de esta, por lo tanto, se incluye el factor de evaluación humano y manual necesario para una correcta evaluación de la tarea.

Recordamos que la programación es el proceso de crear programas de computadora utilizando lenguajes de programación. Esto implica la escritura de código que puede ser interpretado por una computadora para realizar tareas específicas, como procesamiento de datos, cálculos y automatización de tareas. Hay varios elementos importantes que forman parte de la programación y que son relevantes:

- El **lenguaje de programación**
- Los **algoritmos** utilizados; un mismo problema puede ser abordado con diferentes soluciones.
- La **estructura de datos**; formas de organizar y almacenar datos en un programa (listas, arreglos, diccionarios y otro tipo de estructuras).
- Las **bibliotecas** y ***frameworks*** utilizados.
- La **depuración** también es importante, es el proceso de encontrar y corregir errores de un programa.

Entonces, las tareas de programación no son únicamente conseguir el resultado esperado, hay muchos factores importantes que es necesario evaluar detenidamente.

Motivación

La revisión manual del código en un grupo reducido de estudiantes resulta viable, pero a medida que aumenta la cantidad de estudiantes se vuelve cada vez más trabajoso calificar manualmente todas las tareas. Una manera de facilitar la evaluación sería involucrar a más evaluadores, no obstante, esto puede generar inconsistencias en la calificación, ya que cada evaluador podría tener un criterio distinto. Además, este tipo de evaluación es propensa a errores, puesto que los programas de los estudiantes pueden variar ampliamente y ser difíciles de analizar en profundidad.

Por este motivo, llegamos a la conclusión de que es complicado revisar manualmente todas las tareas de los estudiantes y aunque lo ideal sería la revisión humana para fomentar buenas prácticas de codificación, podrían explorarse técnicas que automaticen el proceso de corrección.

2 Objetivo

Nuestro objetivo se centra en conseguir reducir el tiempo de evaluación de ejercicios en Python mediante la automatización de la evaluación de estos, haciendo uso de árboles de sintaxis abstracta (AST). La hipótesis inicial consiste en que la similitud entre los AST de soluciones correctas tiende a ser mayor que al AST de soluciones incorrectas.

La idea es cuantificar la similitud mediante la distancia entre el AST del código del estudiante y el AST del código de referencia y calcular la correlación respecto a la evaluación de un corrector humano para realizar un estudio de esos resultados.

Es necesario también añadir unas etapas previas al cálculo de la distancia que consistirían en el "preproceso" del AST con el fin de obtener una cualificación más correlacionada con la evaluación de referencia.

3 Estado del arte

Este apartado está basado en el trabajo presentado por Si Chuang Li [1]. La entrega de programas es un método utilizado en la educación de ciencias de la computación para reforzar y evaluar conceptos prácticos enseñados en clase.

Por lo que la evaluación automática es un tema que se ha estudiado desde diferentes puntos, con el objetivo de reducir la intervención humana en la corrección, generando así un aumento en la objetividad y la consistencia. Esto se debe a que un proceso automático no es subjetivo, como inconscientemente lo es uno manual. Esta genera una evaluación igual de los ejercicios entregados, ya que sigue los mismos criterios, consiguiendo consistencia en la puntuación otorgada.

La automatización ha sido valorada desde diferentes puntos y se han utilizado diferentes sistemas, plataformas o entornos de programación. Dentro de la gran cantidad de herramientas que podemos llegar a encontrar, hablaremos de aquellas que se centran en la interoperabilidad. Esto significa que pueden adaptarse y emplearse en una variedad de contextos, en lugar de estar limitadas a un solo dominio o tarea específica, ya que nos proporcionan flexibilidad y versatilidad, lo que nos permite interactuar con diversas tecnologías y entornos.

Por otro lado, encontramos las herramientas diseñadas para dominios o tareas específicas. Estás más enfocadas a satisfacer las necesidades de un área o problema particular, pero la investigación de este trabajo trata el caso anterior.

3.1 Evaluación Entrada/Salida

La evaluación de Entrada/Salida (Input/Output) es una técnica comúnmente utilizada en la corrección automatizada de programas para evaluar la funcionalidad de un programa. Esta técnica es el ejemplo más antiguo y común que encontramos de evaluación automatizada que se remonta a 1989 por Isaacson y Scott [2].

Esta técnica permite comprobar rápidamente si un programa funciona correctamente. La idea es que se proporciona una entrada al programa, y se captura su salida. Luego se compara con la salida esperada para esa entrada en particular y se repite el proceso para un determinado número de diferentes entradas. La tasa de coincidencias entre las salidas esperadas y las producidas por el programa se utiliza para evaluar la calidad del programa y consecuentemente que funcione correctamente en todas las situaciones.

Este tipo de evaluación, cuando hablamos de cualificar el ejercicio entregado por un alumno, no se podría utilizar como resultado único y final, ya que no cumple con el objetivo deseado. Esto se debe a que se calificaría como medida esencial que el código compilase y ejecutase, cuando el objetivo como evaluador sería tener en cuenta el proceso, la idea, la estructura entre otras cosas, para poder proporcionar una calificación que valorase todo el proceso del programador y así poder mejorar su codificación y que se pudiera aprender de ello. Un código que no falla en este tipo de evaluación no implica que sea apto para la mayor nota, ya que podría estar formado por un código ineficiente.

3.2 Evaluación de rellenado de huecos

Otro método de evaluación sería el de rellenado de huecos o espacios en blanco, este consiste en proporcionar a los estudiantes un código con espacios en blanco para completarlo. Esto limita al estudiante proporcionándole una guía a seguir. Lieberman [3] ha afirmado que este estilo de asignación es la forma más efectiva para que los programadores principiantes apliquen los conceptos enseñados. A más, esto genera un código evaluable pequeño y rígido, por lo que ha generado el desarrollo de muchas técnicas de análisis de estos espacios en blanco.

La desventaja de esta evaluación y por la que se entra en detalle, es que se utilizan principalmente en clases de programación para principiantes en lugar de clases de nivel superior y en este trabajo el objetivo es abarcar también las clases de un nivel superior que no disponen de rellenado de huecos.

3.3 Evaluación basada en los AST

La evaluación basada en los AST (Abstract Syntax Trees o Árboles de Sintaxis Abstracta) es un enfoque utilizado para evaluar y comprar programas de forma automatizada. Un AST representa la estructura de un programa de manera jerárquica, capturando la sintaxis y las relaciones entre las diferentes partes del código.

En este tipo de evaluación se comparan los AST de los programas para determinar similitudes y diferencias en su estructura y lógica. Esto permite evaluar la calidad y la corrección de los programas de manera objetiva y eficiente, sin necesidad de ejecutarlos. Esto se utiliza en diferentes áreas, como la educación, detección de plagio, programación y optimización de código. Permite automatizar la evaluación de programas, proporcionar retroalimentación rápida a los estudiantes o programadores y mantener la consistencia en la corrección.

Esta es en la que se basa este proyecto porque se adecúa al objetivo propuesto, permitiéndonos realizar pruebas y disponer de un abanico de posibilidades.

4 Análisis del problema

El objetivo se basa en querer automatizar la evaluación de códigos de alumnos escritos en Python a través de AST. Para conseguirlo se necesita crear un código en Python que siga una serie de pasos hasta llegar, mediante pruebas, a diferentes resultados que nos permitan obtener resultados y llegar a una conclusión. En este apartado se explicarán conceptos necesarios hasta llegar a la implementación.

4.1 Descripción de la arquitectura propuesta

La arquitectura que se sigue en él código está formada por una serie de bloques que contienen procesos necesarios para ir tratando y trasformando los resultados.

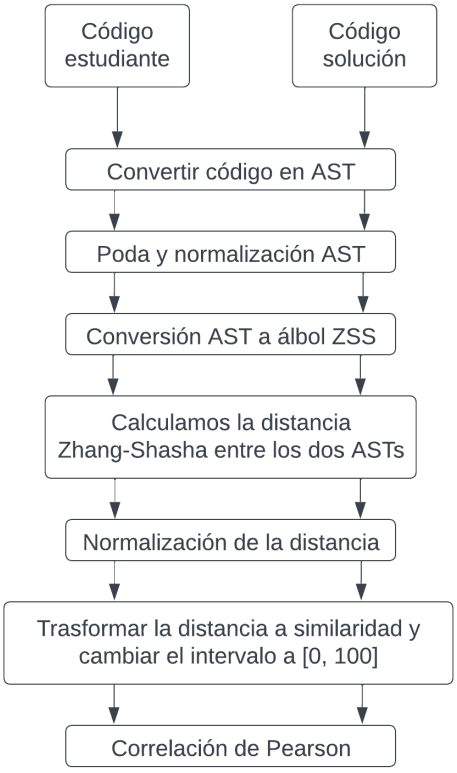

Figura 1: Arquitectura por bloques del código Python realizado

Este diagrama [Figura 1] muestra el proceso que siguen los códigos que se quieren evaluar para acabar obteniendo una cualificación automatizada. El primer proceso por el que pasan los códigos es ser trasformados en sus correspondientes AST, luego hacemos un preproceso del AST obtenido, antes de tratarlo, podando y normalizando. Seguidamente convertimos el AST en el árbol de la librería ZSS para a continuación calcular la distancia entre los dos AST, normalizar dicha distancia, y calcular la correlación de Pearson entre el conjunto de valores. Más adelante se entrará más detenidamente en detalle de en qué consiste cada bloque exactamente.

4.1.1 AST

Los árboles de sintaxis abstracta son una representación simplificada de la estructura de un código de programación en un lenguaje formal. Cada nodo del árbol representa una construcción específica del código fuente. A diferencia de la sintaxis real, el AST no incluye todos los detalles, como puntuación no esencial y delimitadores (corchetes, punto y coma, paréntesis, entre otros). También simplifica, por ejemplo, una construcción como "IF condición THEN" se presenta con un solo nodo y dos ramas.

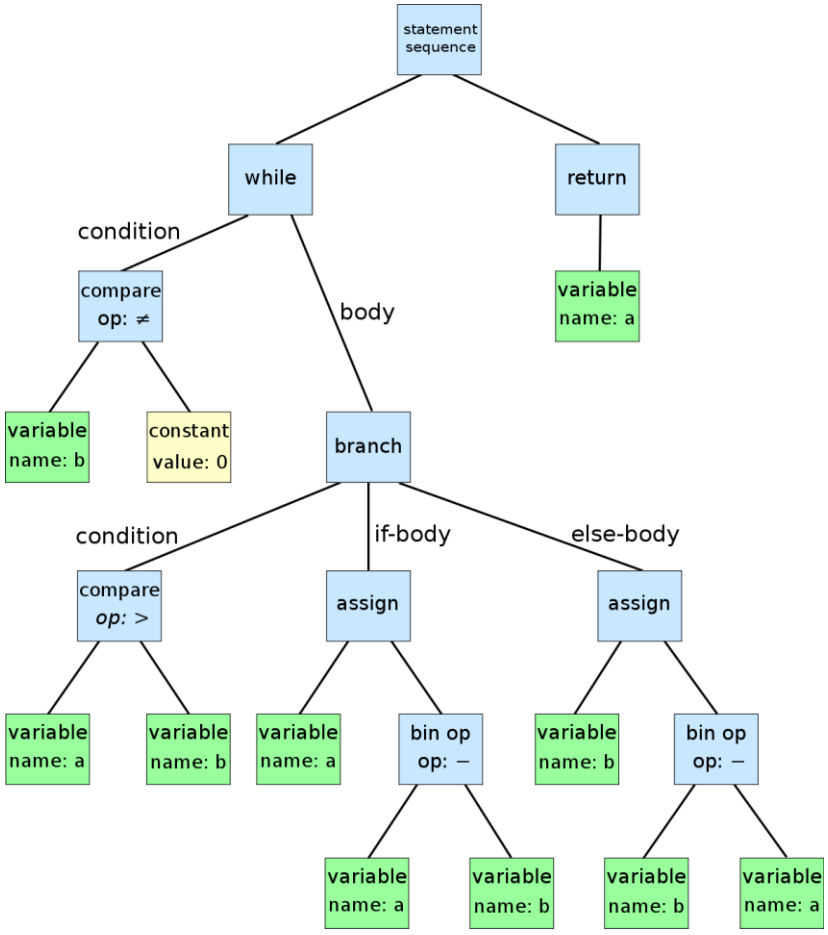

Figura 2: Árbol de sintaxis abstracta de un código extraído de Wikipedia [4].

Una vez construido el AST puede ser editado y mejorado con propiedades y anotaciones para cada elemento que contiene, por lo que podremos hacer ese "preproceso" del AST

para obtener un mejor resultado. Hecho que es imposible con el código fuente de un programa, ya que esto implicaría cambiarlo. Este sería un ejemplo esquemático de como estaría construido un AST [*Figura 2*], definido por instrucciones, variables, literales, funciones, clases, entre otros elementos que pueden formar parte.

4.1.2 Poda y normalización de los AST

La poda y normalización de un árbol de sintaxis abstracta consiste en técnicas de manipulación del árbol para eliminar elementos innecesarios y asegurarse de que el árbol se ajuste a un formato de estructura común. La poda puede consistir en eliminar nodos o ramas del árbol que no son necesarios para el análisis.

En el caso de los AST, que están construidos por elementos que forman parte de un código, sería analizar hasta qué punto la información es necesaria para nuestro objetivo, si son relevantes la cantidad de variables creadas, o el nombre de estas, o si los atributos de cada clase se deberían valorar o son innecesarios.

4.1.3 Distancia entre los AST

La distancia entre los AST será definida por la distancia de edición del árbol (Tree Edit Distance [TED]), que funciona de una manera similar a la distancia de edición de cadenas, como sería por ejemplo la distancia de *Levenshtein* o de *Hamming*, pero con árboles.

Más concretamente, esta distancia es el número mínimo de operaciones de selección necesarias para trasformar un árbol (T1) en otro (T2). Generalmente, las operaciones son; volver a etiquetar, eliminar e insertar un nodo. Las operaciones están asociadas a diferentes costes, y el resultado TED sería el costo de la secuencia de acciones menos costosa que trasforma T1 en T2.

Podemos ver un ejemplo sacado de una fuente de información [5] que es bastante simple para ver de manera rápida en que consiste esta distancia de edición.

- Definimos el coste de cada operación:
  - coste (volver a etiquetar) = 2.
  - coste (eliminar) = coste(insertar) = 3.

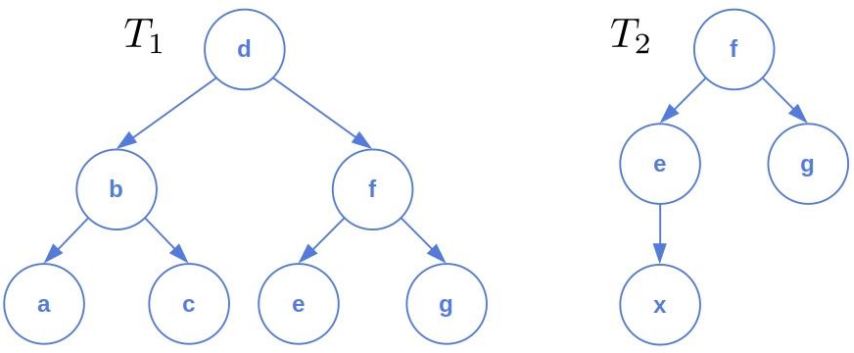

Figura 3: Árboles T1 y T2 representados gráficamente [5].

- Los dos árboles T1 y T2 de los que queremos calcular esa distancia de edición tienen la siguiente estructura [Figura 3].
- Ahora se pasa a identificar la secuencia menos costosa, que en este caso sería:
    - Eliminar nodo c, e y g.
    - Volver a etiquetar: nodo 'f' → nodo 'g' // nodo 'd' → nodo 'f' // nodo 'b' → nodo 'e' // nodo 'a' → nodo 'x'.

Por lo que finalmente el mínimo costo será de TED (T1, T2) = 3*2 + 4*3 = 17. En el caso de la operación de eliminación de un nodo que tenga "hijos", estos se enlazan al nodo al que estaba enlazado el nodo eliminado.

4.1.4 Conversión de distancia a puntuación

La distancia necesita un proceso una vez obtenida para poder valorarse como puntuación, mayoritariamente por el intervalo en el que se encuentran los resultados. Primero debemos normalizarla para que se encuentre en el intervalo de [0, 1]. Para conseguirlo cogemos la distancia y la dividimos por el total de nodos que conforman los AST transformados para calcular dicha distancia. Esto lo hacemos porque, como hemos comentado, la distancia es la cantidad de operaciones que deben realizarse para trasformar un árbol en otro, por lo que como máximo las ediciones serán la suma de los nodos de los dos árboles, que corresponderán al máximo de ediciones posibles.

Una vez obtenido ese resultado, restamos esa distancia normalizada a 1, para obtener la similitud en vez de una medida de distancia, es decir, si el valor es más cercano a 1 índica similitud alta y si el valor es cercano a 0 indica una similitud baja. Después multiplicamos por 100 para obtener la cualificación en el mismo intervalo que la cualificación manual.

Las fórmulas son las siguientes:

$$D_{normalizada} = \frac{Distancia}{n_{T1} + n_{T2}}\ [0,1] \quad \rightarrow \quad P = (1 - D_{norm}) \times 100$$

Siendo 'n_{T1}' el número total de nodos del árbol generado por el código solución, 'n_{T2}' el número de nodos del árbol generado por el código del estudiante y 'D' la distancia.

4.2 Implementación

En este apartado se hablará sobre la implementación del código escrito en Python, de su estructura, los detalles de los paquetes y librerías utilizadas y finalmente los pasos que sigue la ejecución hasta llegar a los resultados.

4.2.1. Estructuración del código

El código está estructurado en 4 archivos para separar las funciones según su utilidad:

- ***main.py***: contiene la función principal que recorre los códigos a evaluar que se encuentran en sus respectivos directorios y llama a cada función para ir recopilando la información hasta devolver los resultados.
- ***ast_processing***: contiene las funciones relacionadas con el procesado del AST.
- ***distance_processing***: contiene las funciones relacionadas con la distancia.
- ***results_processing***: por último, contiene las funciones relacionadas con el formato de devolución del resultado.

Es necesario ejecutar el código proporcionando uno serie de argumentos, entre ellos los nombres de los directorios y archivos que se desea analizar. Empezando por el directorio donde se encuentran los códigos de los alumnos (que deben diferenciarse por el nombre), luego el *path* del directorio que contiene el o los códigos o código solución, el *path* del Excel que contiene la nota manual puesta a cada alumno y el número de las columnas que contienen la información necesaria (explicación más detallada en el 'Readme.md' incluido en el código o utilizando la comanda '–help').

Por otro lado, al finalizar la ejecución, los resultados se guardan en una carpeta que se crea automáticamente, si no existe, llamada "Resultados". En ella, encontraremos 3 archivos que se han autogenerado:

- ***resultados.csv***: contiene los resultados que se han ido recopilando durante la ejecución y lo escribe en este documento, separados por comas y filas.
- ***tabla_resultados.html***: contiene los resultados del archivo generado anteriormente llamado '*resultados.csv',* pero trasformados en una tabla para ver más claramente los resultados (esta se abre automáticamente con el navegador al acabar la ejecución). Ejemplo del resultado de la tabla en el Anexo.
- ***diagrama_dispersión.png***: como el nombre indica, se trata del diagrama de dispersión de los datos obtenidos a más de incluir el resultado de la correlación en formato imagen.

Es necesario comentar que los códigos evaluados son del tipo Jupyter Notebook utilizando el lenguaje de programación Python.

4.2.2. Paquetes y librerías utilizadas

Se hace uso de una serie de paquetes y librerías en el código, estas se utilizan según lo que estemos tratando en cada archivo:

En el caso de estar tratando los AST, las librerías y paquetes utilizados son:

- **import ast [11]**: librería que ayuda a procesar árboles de gramática de sintaxis abstracta en Python. En el código se utiliza para generar el AST y tratarlo.
- **import nbformat [12]**: esta librería se utiliza para trabajar con Notebooks de Jupyter. En el código se leen los Notebooks de los alumnos y así poder acceder a sus celdas hasta encontrar aquella que proporciona su resultado al ejercicio.
- **from zss import Node [13]**: dentro de la librería *zss*, se utiliza la clase *Node* para poder convertir el AST en un árbol formado con *Nodes* de *zss*.

En el caso de estar tratando la distancia, las librerías y paquetes utilizados son:

- **from zss import simple_distance [13]**: dentro de la librería *zss*, se utiliza la función 'simple_distance' para poder calcular la distancia entre los dos AST convertidos a árboles con *Nodes* de *zss*.

En el caso de estar tratando los resultados para guardarlos y luego mostrarlos, las librerías y paquetes utilizados son:

- **import csv [14]**: esta librería implementa clases para leer y escribir tablas en formato CSV. Es utilizado para escribir los resultados en un CSV, para luego leerlos y pasarlos a una versión tabla '*html*'.
- **import os [15]**: librería que contiene funciones para interactuar con el sistema operativo que ejecuta Python. Se utiliza en este código para comprobar si existen directorios (si no crearlos), obtener '*path'* actual o ejecutar el '*html'* en el navegador automáticamente.
- **import ezodf [16]**: paquete de Python para realizar acciones en archivos OpenDocument (ODF). Se utiliza en este código para abrir el CSV que contiene las notas manuales de los alumnos y así poder comprarlas.
- **import matplotlib.pyplot as plt [17]**: biblioteca de Python que se utiliza para crear visualizaciones y gráficos en 2D/3D. Se utiliza en este código para crear el diagrama de dispersión de las notas manuales y las notas predichas.
- **import pandas as pd [18]**: librería de análisis de datos en Python que se utiliza para manipular y analizar conjuntos de datos. Proporciona estructura de datos flexibles para trabajar con tablas y matrices de datos, entre otras funciones. Se utiliza para leer los archivos del CSV y así tener un '*dataframe'* para poder extraer dos columnas y crear el gráfico de dispersión con esos datos y por otro lado para crear un '*html'* con ese '*dataframe'*. También se utiliza para calcular correlación.
- **from unidecode import unidecode [19]**: librería de Python que convierte caracteres Unicode en sus equivalentes ASCII más cercanos. Lo utilizamos para eliminar los acentos de los nombres de los alumnos y poder comparar el nombre del código con la nota manual de este, extraída del código.

4.2.3. Implementación del código

Para entender mejor los pasos seguidos en la implementación, seguiremos la arquitectura por bloques, con una explicación previa y un ejemplo visual de cada apartado. Cada apartado enlazará con el siguiente.

Obtención de los códigos

Para obtener los códigos de la función, recorremos las carpetas que contienen el código del alumno (cada carpeta contiene el código de un alumno, en concreto en un Notebook). El *path* de este directorio se habrá pasado previamente como argumento al ejecutar el código. Continúa abriendo el Notebook de Jupyter y se recorre sus celdas buscando aquella que contenga una función llamada como la que buscamos. En el caso de encontrar dos celdas que contengan la función, nos quedamos con aquella que contenga la función más larga, que se intuye que será la que realmente está implementada.

En el caso del código con la solución, se realiza de la misma manera, se recoge como argumento el *path* del directorio donde se encontrarán las o la solución o soluciones y se recorre el directorio que contiene los Notebooks que incluyen soluciones al ejercicio planteado. Por cada Notebook se sigue el mismo flujo que en el caso del código del estudiante, buscando la función y al mismo tiempo que se van generando los AST por cada solución, se van guardando hasta retornar una lista que trataremos más adelante.

Para entender el concepto gráfico de lo que realmente se está haciendo, estamos queriendo evaluar un ejercicio, que debe encontrarse encapsulado en una función dada con un nombre determinado. Mostramos el siguiente código ejemplo, bastante simple.

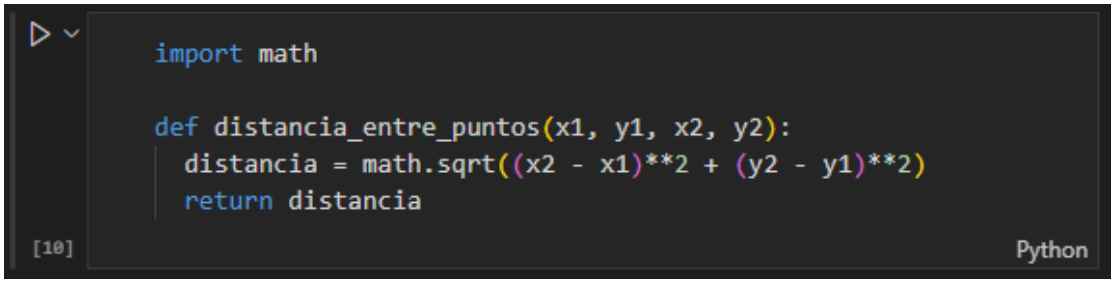

Figura 4: Código solución escrito en un Notebook Jupyter en Python que retorna la distancia entre puntos.

Por ejemplo, se quiere evaluar la función mostrada en la [Figura 4], y este código sería la versión solución que se utilizaría como referencia para calcular la distancia de edición entre esta solución y la entrega del alumno. En este caso buscamos la función con el nombre 'distancia_entre_puntos' dentro de cada código entregado y extremos el contenido de la celda del Notebook para poder seguir con el flujo del programa.

Extracción AST

Por cada código de la función recogido generamos su respectivo AST, tanto para los códigos de los alumnos, como para los códigos de la solución. De esta manera podemos comparar y tratar los AST. Para hacernos una idea de cómo funciona el etiquetado del formato AST, extraemos el AST del ejemplo [Figura 4].

less Copy code

```
Module(body=[
    FunctionDef(
        name='distancia_entre_puntos',
        args=arguments(
            args=[
                arg(arg='x1', annotation=None),
                arg(arg='y1', annotation=None),
                arg(arg='x2', annotation=None),
                arg(arg='y2', annotation=None)
            ],
            vararg=None,
            kwonlyargs=[],
            kw_defaults=[],
            kwarg=None,
            defaults=[]
        ),
        body=[
            Assign(
                targets=[Name(id='distancia', ctx=Store())],
                value=Call(
                    func=Attribute(
                        value=Name(id='math', ctx=Load()),
                        attr='sqrt',
                        ctx=Load()
                    ),
                    args=[
                        BinOp(
                            left=BinOp(
                                left=BinOp(
                                    left=Name(id='x2', ctx=Load()),
                                    op=Sub(),
                                    right=Name(id='x1', ctx=Load())
                                ),
                                op=Pow(),
                                right=Num(n=2)
                            ),
                            op=Add(),
                            right=BinOp(
```

Figura 5: Árbol de Sintaxis Abstracta (AST) del ejemplo de código [Figura 5]. Generado por IA [21] para los colores.

En la documentación de la librería ***ast*** nos explica la gramática abstracta que utiliza esta librería, pero con este ejemplo visualmente se entiende bastante bien cómo funciona.

Conversión AST a árbol ZSS

Llegados a este punto y sabiendo de antemano que vamos a utilizar la librería ZSS para calcular la distancia entre dos AST, se necesita principalmente transformarlos en un árbol del formato que permite ZSS para calcular la distancia. Para ello se crea un método recursivo, que recorre el AST, por cada nodo coge su etiqueta y crea un *Node* con esa etiqueta manteniendo la jerarquía y una manera de identificarlo, creando *Nodes*. De esta manera mantenemos la estructura general AST para tratarlo.

Como se puede intuir, esto implica aplicar una poda y normalización del AST muy drástica, ya que únicamente nos quedamos con la jerarquía y la identificación de cada nodo, que realmente es su función.

El formato que necesita ZSS se comenta en la métrica utilizada [apartado 5.1.1] por lo tanto, se mostrará únicamente un ejemplo visual de como quedaría el árbol trasformado, basándonos en el AST anterior [Figura 5].

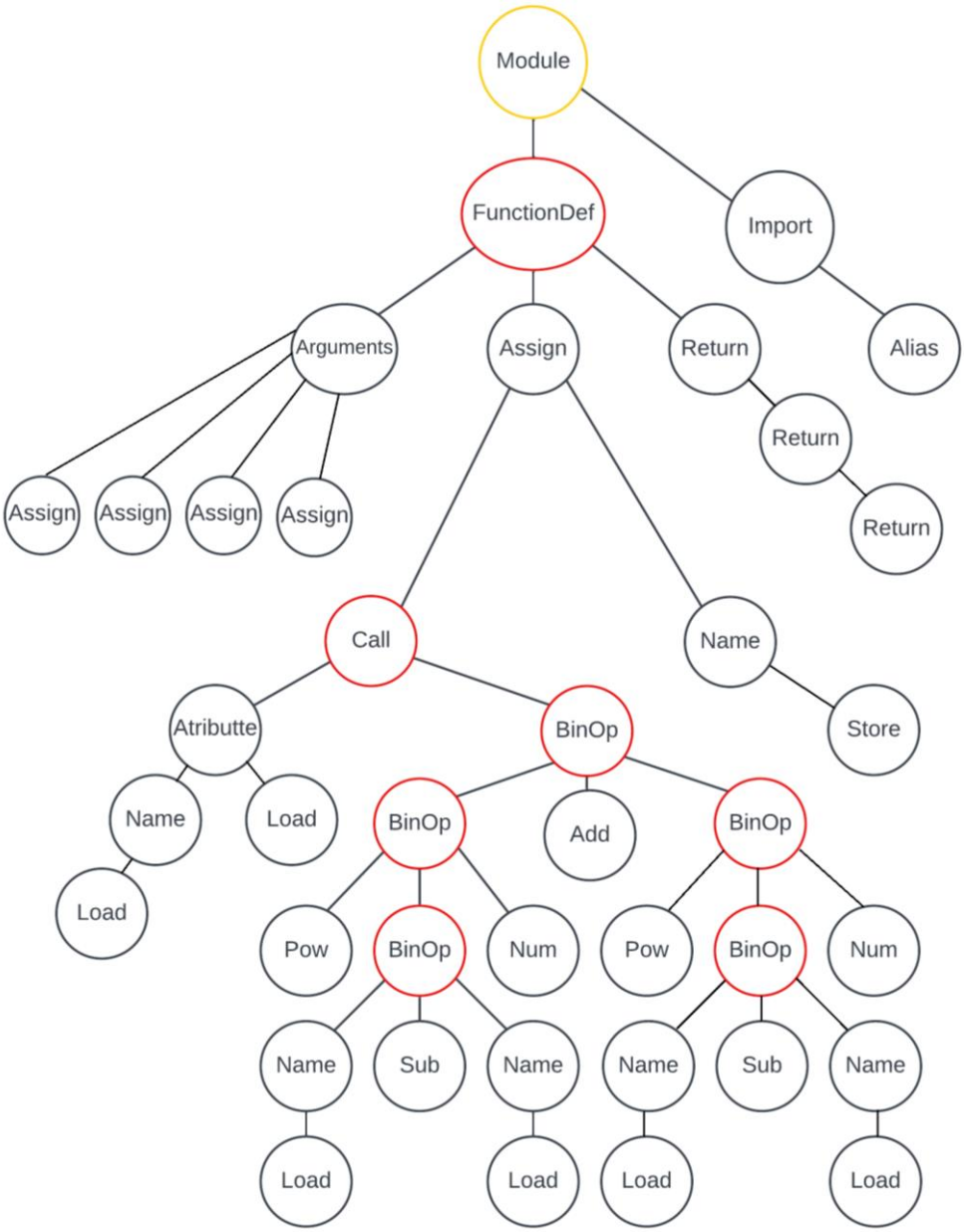

Figura 6: AST trasformado a árbol ZSS del código solución de la función distancia entre puntos [Figura 5].

Siguiendo los nodos que se van creando en el código a través de 'prints', generamos este diagrama que nos muestra como quedaría el árbol transformado en este caso particular.

Distancia entre ASTs

Llegados a este punto se utiliza la función "*simple_distance*" de la librería ***zss*** sobre dos árboles, uno correspondiente al código solución y otro al código del estudiante. Esta función permite principalmente introducir las raíces de los dos árboles.

En el ejemplo que se ha ido tratando anteriormente [Figura 6], que es el código solución, se modifica para generar un código a evaluar y así poder calcular la distancia entre ellos.

```python
import math

def distancia_entre_puntos(x1, x2):
    distancia = math.sqrt((x2 - x1)**2 - (y2)**2)
    return distancia
```

[3] Python

Figura 7: Código a evaluar escrito en un Notebook Jupyter en Python que retorna la distancia entre puntos.

Como vemos el cambio realizado respecto al código solución ha sido eliminar la variable ‘y1’ de la operación, esto hace que el AST cambié y que se pueda calcular la distancia de edición para saber el número de operaciones necesarias para que este AST se convierta en el AST formado por el código solución.

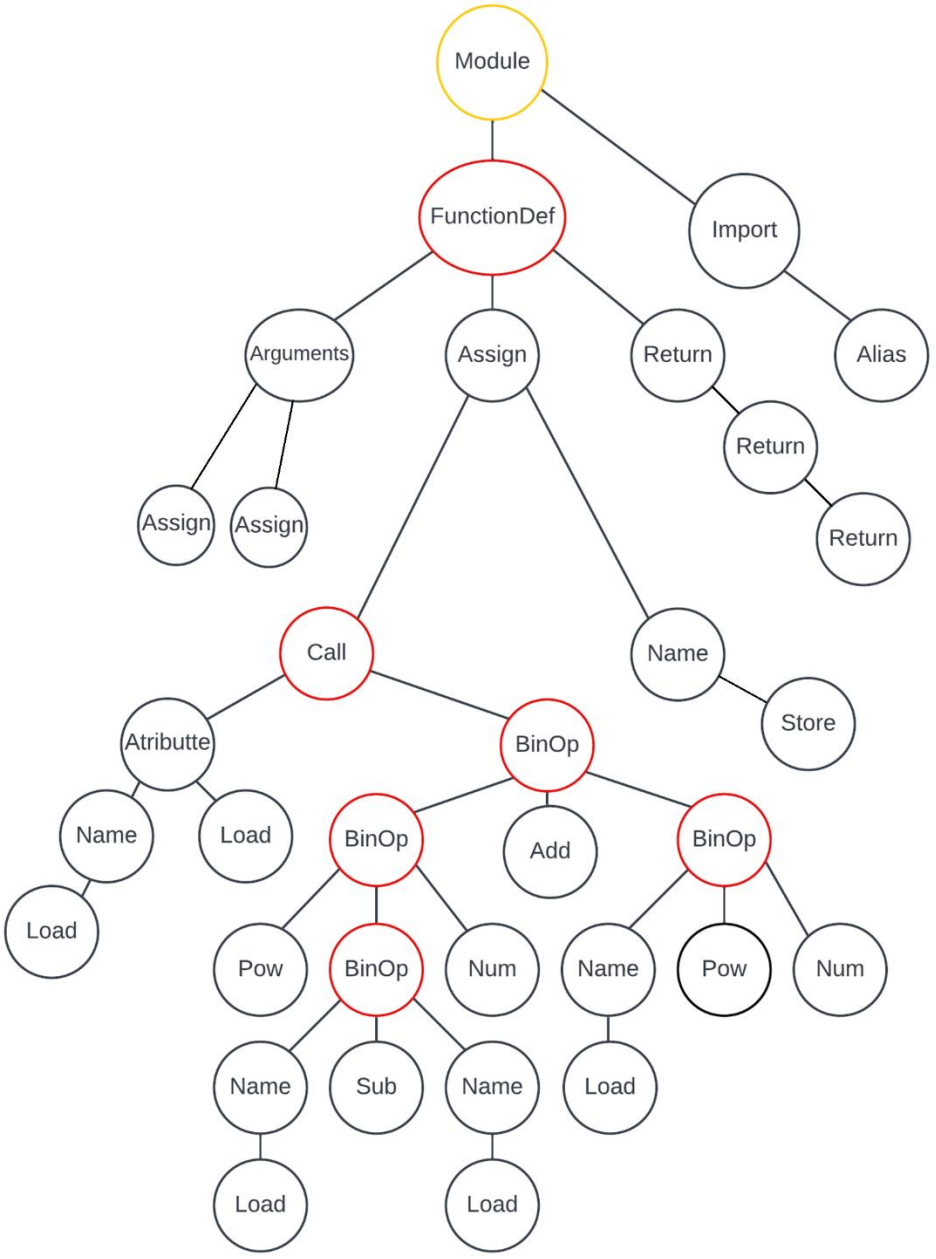

Figura 8: AST trasformado a árbol ZSS del código solución de la función distancia entre puntos [Figura 7].

Se puede observar como la diferencia entre diagramas [Figura 6] y [Figura 8] es que desaparecen visualmente 5 nodos importantes. La distancia resultante es de 5, eso implica que la mínima distancia para convertir uno de los árboles en el otro necesita de 4 operaciones, ya sean de inserción, eliminación o cambio de etiquetas. En este caso se trata de 4 eliminaciones y un cambio de etiqueta, para convertir el árbol de la función a evaluar en el árbol de la función solución.

Conversión de distancia a puntuación

El intervalo de la distancia calculada por ZSS se encuentra entre [0, infinito), esto implica que se debe procesar la distancia para convertirla en lo que necesitamos que es una puntuación. Como ya hemos comentado el proceso que seguimos en otro apartado [apartado 4.1.4]. Se llevará a la práctica siguiendo la distancia obtenida anteriormente basándonos en los árboles [Figura 6] y [Figura 8].

Se obtiene una distancia de 4 operaciones, el total de nodos del árbol ZSS correspondiente al código solución es de 40 (hay un método en el código que lo calcula) y el del código que queremos evaluar, tiene 36 nodos. Con estos datos podemos normalizar la distancia dividiendo la distancia por la suma de los nodos totales de los dos árboles. El resultado lo convertimos en similitud, restándolo a 1 y lo multiplicamos por 100 para que esté en el rango de una puntuación (entre 0 y 100). Esto nos da un resultado de 94’74 como puntuación automática al ejercicio entregado.

Diagrama de dispersión y correlación

Por último, se calcula la correlación entre los resultados obtenidos automáticamente y las calificaciones obtenidas manualmente. Esta correlación nos permite refinar el código fijándonos en los valores atípicos para mejorar la automatización.

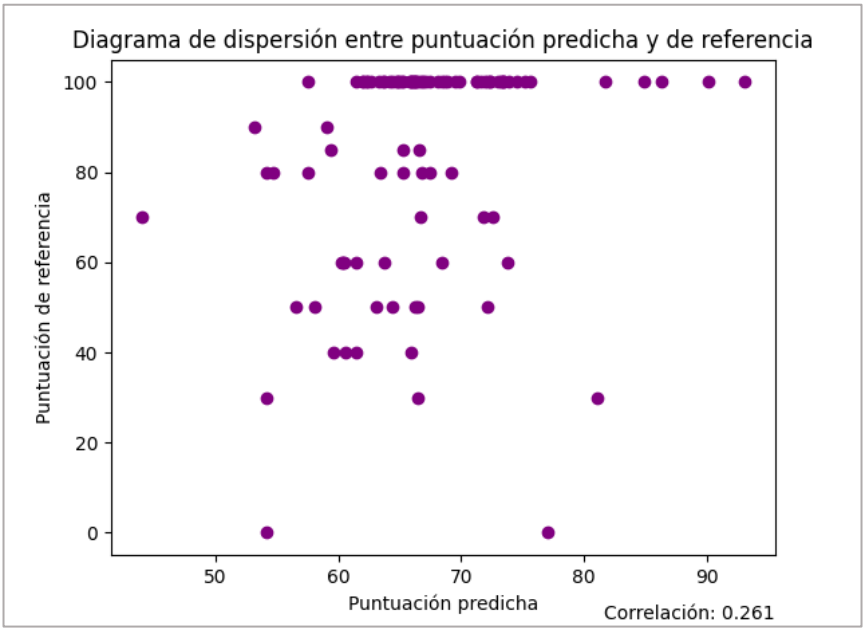

Figura 7: Diagrama de dispersión resultante entre el conjunto de datos de 108 alumnos.

Para ello es necesario una considerable cantidad de códigos a evaluar, dicho de otra manera, un conjunto de datos. Entonces se utilizan las entregas de 108 alumnos de primero de carrera, a un ejercicio propuesto en clase y la solución a dicho ejercicio. El resultado del diagrama de dispersión se puede ver en la [Figura 9], junto a la correlación resultante, que en este caso es de un 0'261.

5 Resultados y discusión

En este apartado se hablará de los datos, métricas, pruebas y resultados utilizados y obtenidos con los que se llegará a las conclusiones.

5.1 Datos utilizados

Los datos que se han utilizado para las pruebas realizadas han sido, en un principio, un código fácil y breve que calculaba la distancia entre puntos. Una versión con la solución y otra que correspondería al estudiante, que íbamos editando para ir viendo el comportamiento recibido. De esta manera podíamos ver más claramente los cambios y los resultados en un periodo más corto de tiempo.

Una vez se ha conseguido una versión más o menos definitiva del código de evaluación automática, se ha pasado a utilizar datos reales y de una gran magnitud. Estos datos han sido los ejercicios entregados por los alumnos en la asignatura de Algorítmica, en primero de carrera de Ingeniería informática. Este ejercicio consistía en devolver la subcadena más larga sin repetir ningún carácter de la cadena introducida por parámetro. Para ser exactos se han utilizado 108 códigos entregados por alumnos, teniendo un código solución de referencia para calcular la distancia, aunque posteriormente se han creado otros a través de inteligencia artificial o con cambios muy simples.

También se disponía de la nota manual puesta por un evaluador humano, factor que nos ha permitido realizar una correlación con los datos y poder refinar el código creado.

5.2 Métricas

Las métricas utilizadas consisten en la distancia entre los AST y el coeficiente de correlación entre la nota automatizada y la nota del corrector humana.

5.1.1 Distancia de edición Zhang-Shasha

Como ya se ha comentado, calculamos la distancia de edición entre árboles en Python con Zhang-Shasha v1.1, un algoritmo utilizado para comparar la similitud entre dos árboles, dando como resultado una distancia. La distancia de edición Zhang-Shasha necesita la modificación del árbol AST para seguir un formato predefinido. Este formato es el que se puede ver en la documentación oficial de la librería [13] [Figura 9].

```
from zss import simple_distance, Node

A = (
    Node("f")
        .addkid(Node("a")
            .addkid(Node("h"))
            .addkid(Node("c")
                .addkid(Node("l"))))
        .addkid(Node("e"))
    )
B = (
    Node("f")
        .addkid(Node("a")
            .addkid(Node("d"))
            .addkid(Node("c")
                .addkid(Node("b"))))
        .addkid(Node("e"))
    )
assert simple_distance(A, B) == 2
```

Figura 9: Estructura de árbol con Node y función de edición de distancia ZSS.

Una vez realizado este cambio podemos utilizar dos de las funciones de las que dispone para calcular la distancia de edición de los árboles. Estas funciones son las siguientes:

distance(zss_ast1, zss_ast2, get_children, insert_cost, remove_cost, update_cost)

simple_distance(zss_ast1, zss_ast2, get_children, get_label, label_distance)

Estas calculan la cantidad mínima de operaciones que deben realizarse para trasformar un árbol en otro. Cada operación puede consistir en la adición, eliminación del nodo o modificación de la etiqueta. La diferencia entre las dos es que la función 'distance' permite introducir como argumentos opcionales el coste de las diferentes operaciones de edición del AST y la 'simple_distance' tiene argumentos opcionales que permiten añadir funciones tanto para obtener la etiqueta del nodo (get_label) o para crear una función que retorne la distancia de edición de las etiquetas que estamos comparando (label_distance). Si es la misma retornaríamos 0.

5.1.2 Coeficiente de correlación de Pearson

El coeficiente de correlación de Pearson [20] lo utilizamos como última métrica de procesado de la calificación automática con la manual. Esta es una medida de la fuerza y dirección de la relación lineal entre dos variables continuas, y oscila entre -1 y 1. Un valor de 1 indica una correlación positiva perfecta, un valor de -1 indica una correlación negativa perfecta y un valor de 0 indica la ausencia de correlación lineal entre las variables. La ausencia implica que no existe ninguna relación entre el conjunto de datos.

La fórmula de la correlación de Pearson se puede escribir como:

```
r = cov(x, y) / (std(x) * std(y))
```

Figura 10: Fórmula de la correlación de Pearson en Python.

Donde *cov*(x, y) es la covarianza entre *x* e *y*, y *std*(x) y *std*(y) son las desviaciones estándar de x e y, respectivamente. La covarianza es una medida estadística que indica cómo las variables cambian juntas y la desviación estándar es otra medida estadística que indica cuánto se desvían los datos de su media. En este proyecto se utiliza una función de la librería ***pandas*** que calcula la correlación pasada el conjunto de datos en *dataframe*.

5.3 Pruebas y resultados

5.3.1 Pruebas con un código simple

Las primeras pruebas realizadas, con una primera versión del código ya definida, se han realizado con el código de la función que calcula la distancia entre puntos que hemos ido utilizando de ejemplo [Figura 4]. Como el ejemplo está hecho manualmente, no tenemos la cualificación puesta por un profesor del ejercicio, pero la hemos utilizado para ir modificando el código a evaluar. Se ha comparado con el código solución para ver qué resultados se obtienen según lo que se modifica, añade o elimina.

A continuación, añado una tabla con las pruebas más relevantes, las otras pruebas están incluidas en el anexo de la memoria [Anexo 1]:

Pruebas añadiendo variables		
Pruebas realizadas	Distancia	Nota

Se cambia el nombre de la variable creada anteriormente pero no se cambia donde se utiliza:
def **distancia_entre_puntos**(x1, y1, x2, y2):
  i = (x2-x1)
  distancia = math.sqrt(item**2 + (y2 - y1)**2)
 return distancia
	13.0	84.71

Se añade otra variable (ahora hay 2 variables) pero no se sustituye el valor en la operación: def **distancia_entre_puntos**(x1, y1, x2, y2): i = (x2-x1) o = (y2 - y1) distancia = math.sqrt(i**2 + (y2 - y1)**2) **return** distancia	22.0	76.6

Pruebas añadiendo comentarios		
Pruebas realizadas	Distancia	Nota
Se añade un comentario en vez de que con el "#", de la siguiente manera: ' ' ' Hola ' ' '	2.0	97.56
Se le añade un pequeño comentario: # Hola	0.0	100.0
Se le añade un comentario más grande: # Hola sfsdfs dfsdfsdfsd fsdfdfdfdfdf dfdfdfdfdfdf dfdfdfdf dfdfdfdf fdfdf	0.0	100.0
Se le añaden dos comentarios de los añadidos anteriormente para ver qué pasa: ' ' 'Hola ksdkmdskmfs df s df sd f sdfdfd fdfdfdf dfdfsdfsd sdfsdf sdfsdf sdf ' ' ' ' ' 'Hola ksdkmdskmfs df s df sd f sdfdfd fdfdfdf dfdfsdfsd sdfsdf sdfsdf sdf ' ' '	4.0	95.24

Pruebas extrayendo métodos		
Pruebas realizadas	Distancia	Nota

Se le añade un nuevo método:
def **getItems**(x2, x1):
  return (x2-x1)
def **getSum**(x1, y1, x2, y2):
  return **getItems**(x2,x1)**2 + **getItems**(y2,y1)**2
def **distancia_entre_puntos**(x1, y1, x2, y2):
  distancia = math.sqrt(**getSum**(x1, y1, x2, y2))
  return distancia
	49.0	55.86

Se crea un método y se extrae parte de la operación:
def **getItem1**(x2, x1):
  return (x2-x1)
def **distancia_entre_puntos**(x1, y1, x2, y2):
  distancia = math.sqrt(**getItem1**(x2,x1)**2 + (y2 - y1)**2)
  return distancia
	14.0	84.78

Pruebas modificando el código un poco		
Pruebas realizadas	Distancia	Nota
Se cambia el cálculo: def **distancia_entre_puntos**(x1, y1, x2, y2): distancia = (x2-x1)**2 + (y2 - y1)**2 **return** distancia	5.0	93.33
Se cambia el cálculo: def **distancia_entre_puntos**(x1, y1, x2, y2): distancia = (x2-x1) + (y2 - y1) **return** distancia	11.0	84.06
Se cambia el cálculo: def **distancia_entre_puntos**(x1, y1, x2, y2): distancia = (x2-x1) **return** distancia	19.0	68.85
Se cambia el cálculo: def **distancia_entre_puntos**(x1, y1, x2, y2): distancia = x2 **return** distancia	23.0	59.65

Pruebas modificando el código totalmente para que no sea igual que el de la solución		
Pruebas realizadas	Distancia	Nota
Código totalmente distinto y extenso (versión completa que se utiliza, incluida en las tablas del anexo).	86.0	40.28
Se coje el código anterior como código de profesor y modifico el del estudiante de la siguiente manera: def **distancia_entre_puntos**(lista): **return** [abs(num) for num in lista]	90.0	26.23

Una vez realizadas estas pruebas se extraen conclusiones interesantes:

- El código se centra únicamente en la estructura, no tiene en cuenta si se utiliza una variable que no existe, por lo que no tiene valor, o si hay declarada una que finalmente no se utiliza. Tampoco le importa si el resultado no es el correcto, únicamente valora la semejanza en las operaciones con el código solución.
- Los comentarios incluidos con la almohadilla (‘#’) no los tiene en cuenta, pero aquellos que están incluidos con las comillas sí.

- El hecho de extraer operaciones en métodos genera un incremento de la distancia de edición.
- Se ve también que modificando el código un poco, sobre todo eliminando elementos, genera un aumento de la distancia y a consecuencia una bajada de la cualificación automática.
- Conseguir un ejemplo de códigos totalmente diferentes que puedan llegar a una distancia que genere una puntuación cercana al 0 es bastante complicado.

5.3.2 Pruebas con un conjunto de datos

Ahora se realizan pruebas con datos reales. Se utilizan los códigos entregados por 108 alumnos implementando una función pedida en la clase de Algorítmica en el primer curso de Ingeniería informática de la Universidad de Barcelona. La función tiene el nombre específico de 'subcadena_mes_llarga' y disponemos también de un código que contiene una de las posibles soluciones al problema planteado. Por otro lado, también tenemos un Excel con las notas puestas a cada alumno manualmente por el profesorado.

En la primera ejecución del código con el conjunto de datos obtenemos el siguiente diagrama de dispersión [Figura 11]:

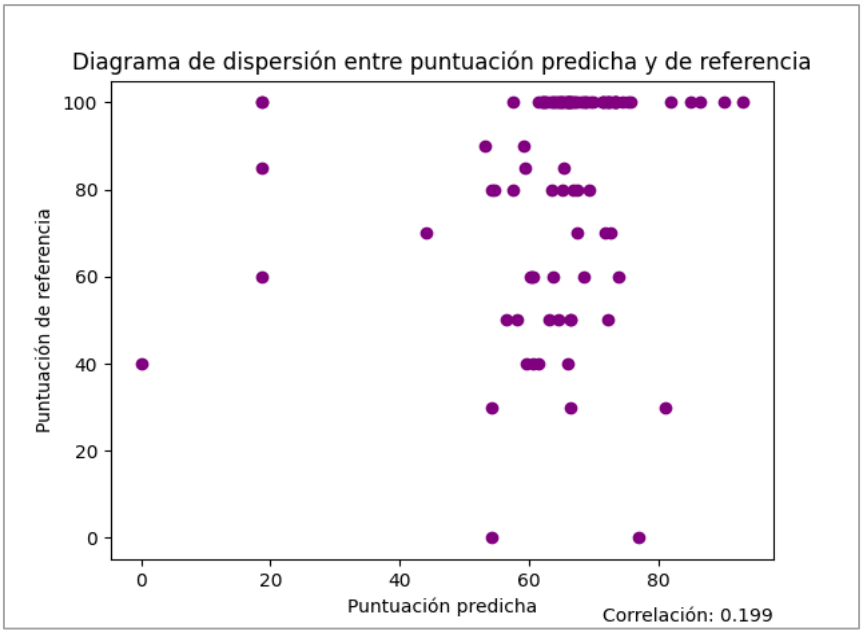

Figura 11: Diagrama de dispersión entre puntuación predicha y de referencia con todo el conjunto de datos.

Se puede observar como hay valores atípicos en el diagrama [Figura 11]. Estos han sido estudiados uno a uno para entender por qué se daban y conseguir mejorar esa correlación. Se va a ir comentando los cambios que se han realizado y los resultados obtenidos a medida que se iban mejorando estas diferencias.

Casos de error

Se ve un punto donde la puntuación predicha es 0 y la manual es 4. Este caso se ha dado por el hecho de que el código de un alumno está mal tabulado y no se puede directamente el AST esto genera que salte una excepción.

En este caso, se modifica el código para dar una puntuación predicha de “NaN” y mostrar únicamente este resultado en la tabla ‘html’ que se genera y no en el diagrama de dispersión. De esta manera se puede detectar que pasa algo y que el evaluador pueda mirarlo manualmente o mirar de tabular automáticamente.

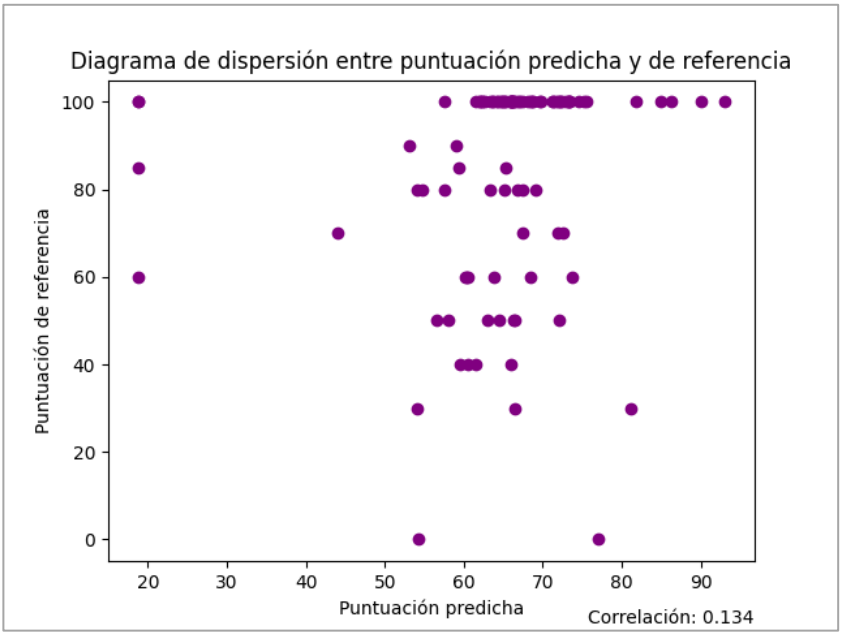

Figura 12: Diagrama de dispersión entre puntuación predicha y de referencia extrayendo los casos de error del conjunto de datos.

Los resultados [Figura 12] extrayendo ese error, genera una correlación de 0’134, peor de la que ya teníamos. Esto no es una mala señal, simplemente falta arreglar más puntos.

Casos inusuales

También se observa en el diagrama de dispersión [Figura 12] tres puntos que tienen la misma puntuación predicha, un 1’18, algo bastante extraño porque a más tienen diferentes notas de referencia. Una vez observado el código de cada alumno que generaba esos resultados. Se observa que los estudiantes han implementado la función en otra celda, dejando la del enunciado también, es decir, hay 2 celdas donde podemos encontrar la función con el nombre ‘subcadena_mes_llarga’ motivo por el cual el código actual cogía la primera celda que encontraba con esa función y era la que contenía la función vacía, únicamente con el comentario que venía por defecto.

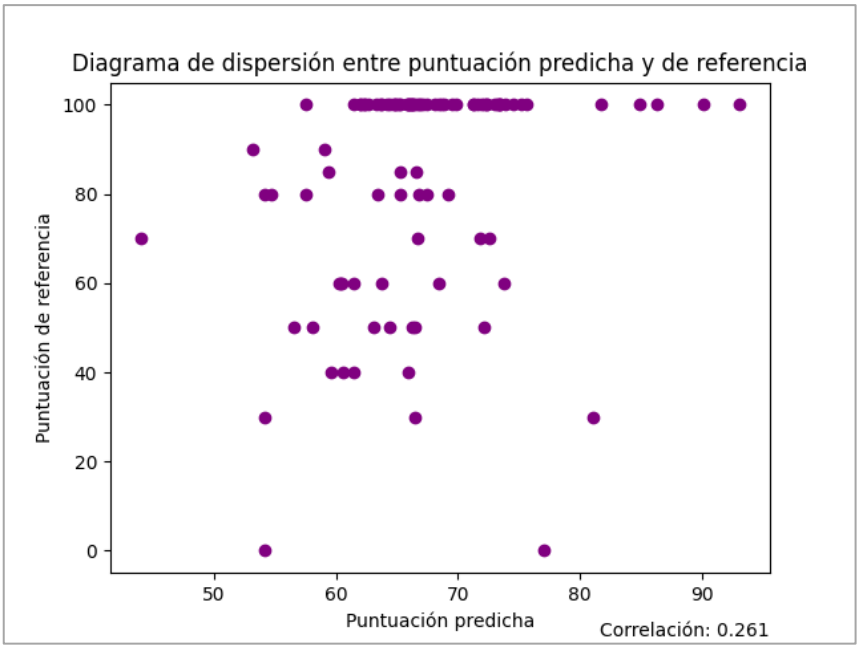

Figura 13: Diagrama de dispersión entre puntuación predicha y de referencia extrayendo los casos de error y los casos inusuales descubiertos.

Entonces se realiza el cambio en el código para recorrer todas las celdas del Notebook y si encuentra más de 2 celdas de tipo código donde se encuentra la función, se quede aquella que tiene más contenido, deduciendo que será la que está implementada. Obtenemos como resultado una correlación de un **0'261** [Figura 13], considerablemente mejor que los anteriores resultados [Figura 11] [Figura 12].

Casos donde la puntuación de referencia es menor a 5

Después de modificar estos casos particulares y consiguiendo mejorar la correlación a un 0'261, pasamos a enfocarnos en otros casos. Se pone el foco en los casos en los cuales la puntuación de referencia es más pequeña que 5:

- Se observan 2 casos en los que la puntuación de referencia es 0 y la predicha es o 5'4 en un caso o 7'8 en el otro.
- Se observa un caso que tiene bastante diferencia entre las puntuaciones, donde la puntuación de referencia es un 3 y la predicha un 8'1.
- También hay casos con una diferencia menor, pero que se deben a lo mismo.

Estos resultados son debidos a que el algoritmo únicamente tiene en cuenta la estructura del código, por lo que aquellos códigos que reciben una nota manual menor que 5, mayoritariamente es debido a que el código no da buenos resultados, detalle que el cálculo de la distancia no valora.

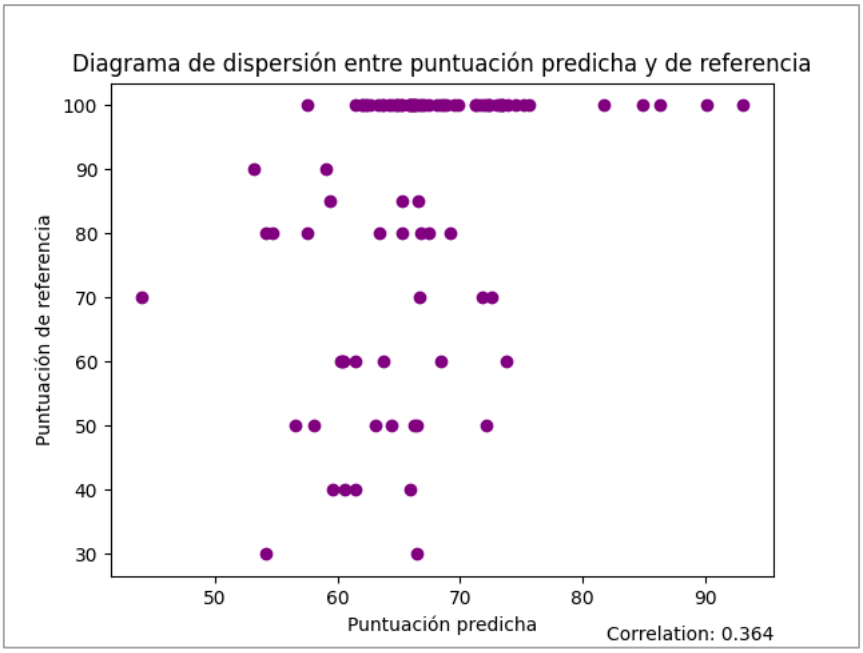

Figura 14: Diagrama de dispersión entre puntuación predicha y de referencia, extrayendo los casos de error, los inusuales y algunos casos donde la puntuación de referencia es menor a 5.

En el caso de eliminar algunos de los datos que dan resultados tan dispares para ver su efecto en la correlación, obtendríamos el siguiente diagrama [Figura 14] que como se ve aumenta la correlación a un **0'364**, mejorando así la relación lineal entre el conjunto de datos (únicamente se han eliminado los 3 datos más destacados comentados anteriormente).

Casos donde la puntuación de referencia es mayor a la puntuación predicha

Por otro lado, se ven valores que, por lo contrario, en vez de encontrarse con una puntuación de referencia baja y una puntuación de predicha alta, se encuentran una puntuación de referencia alta y una puntuación de predicha baja. Esto es debido a que un mismo problema puede ser codificado de diferente manera, es decir, siguiendo otra estructura y no por ello menos buena, por lo tanto, es necesario hacer la prueba con más de un código de solución diferente.

Para solucionar lo comentado, primero se ha modificado la solución de referencia por otra generada por una inteligencia artificial [21], y comprobado mediante pruebas que funcionaba correctamente. Se le ha pedido el nuevo código solución diciéndole exactamente: "otra manera de hacer este código, siguiendo otra estructura totalmente diferente *se introduce código de la solución actual*". Al ejecutar el programa utilizando este código como referencia, se ha obtenido el siguiente resultado:

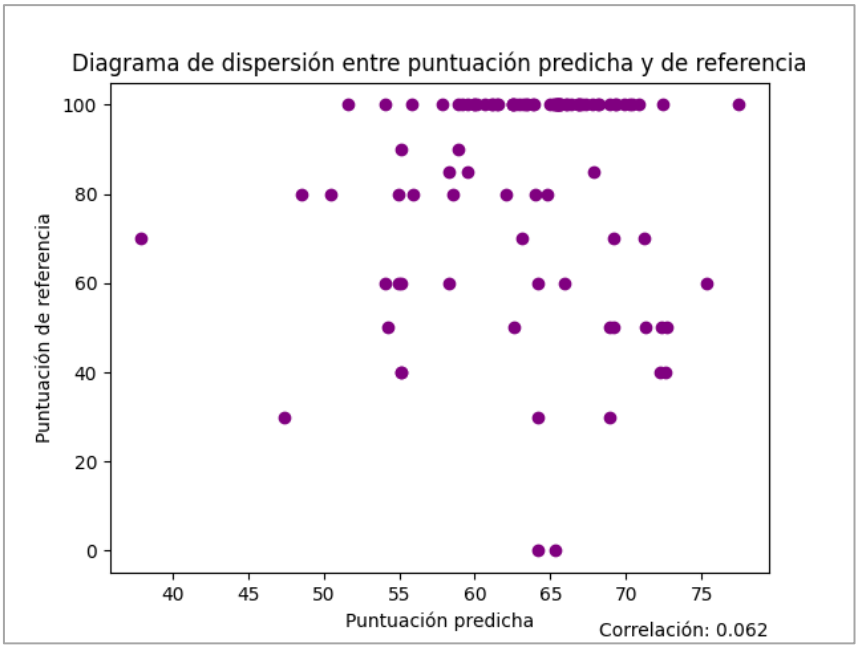

Figura 15: Diagrama de dispersión entre puntuación predicha y de referencia, extrayendo casos [Figura 14] e incluyendo otro código de referencia.

El resultado [Figura 15] nos da una correlación de un 0'062, resultado bastante negativo, pero si se mira individualmente cada resultado, hay casos en los que la puntuación predicha se asemeja más a la puntuación de referencia, que usando la otra solución.

Otra prueba que se hizo es seleccionar uno de los códigos de un estudiante que tuviera una puntuación manual de un 10, eso implicaba que funcionaba perfectamente y la estructura también era correcta. Por otro lado, que la puntuación predicha fuera la más baja en los resultados. Ese valor era de 5'54, entonces implicaba que no seguía la estructura de la solución de referencia, pero aun así era una solución correcta.

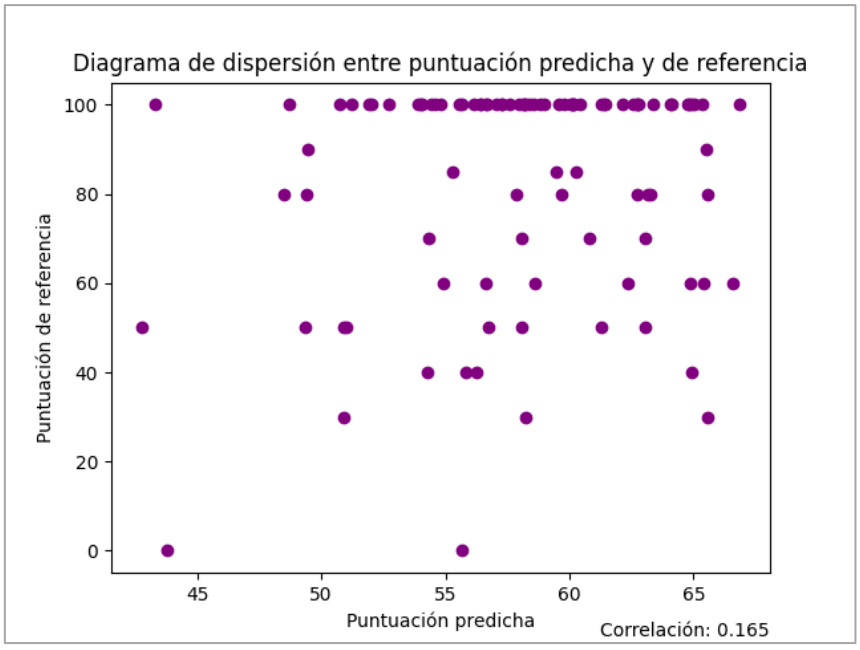

Figura 16: Diagrama de dispersión entre puntuación predicha y de referencia, extrayendo casos [Figura 14] y cambiando el código de referencia.

El resultado [Figura 16] nos mejora la correlación obtenida anteriormente [Figura 15] pero sigue siendo mucho más baja que utilizando la primera solución de referencia.

Se añade la funcionalidad de poder incluir más códigos de referencia

Viendo los resultados que se obtenían, se realizó una modificación del código para que se pudieran valorar diferentes soluciones de referencia. Exactamente lo que se ha añadido es crear los AST de todas las soluciones proporcionadas y calcular la distancia respecto a la solución del estudiante para cada una. Una vez están todas calculadas, seleccionamos aquella que tenga la puntuación más alta, por el hecho de que, si se está siguiendo la hipótesis de que el AST procedente del ejercicio de un estudiante, se considerará correcto de forma directamente proporcional a su semejanza al AST de referencia, es decir, al AST del código del profesor, indica que nos interesa la semejanza a la solución más alta.

Añadido esta nueva funcionalidad al código, probamos a hacer uso de 4 códigos solución, usando todos los datos [Figura 13]. Dos de los códigos se han generado por inteligencia artificial pidiéndole que tengan estructuras totalmente diferentes y comprobado su correcto funcionamiento. Otro que sea el código de la alumna que su puntuación manual es un 10 y la nota predicha es un 5’4 y por último la solución original del profesor. El resultado obtenido es el siguiente:

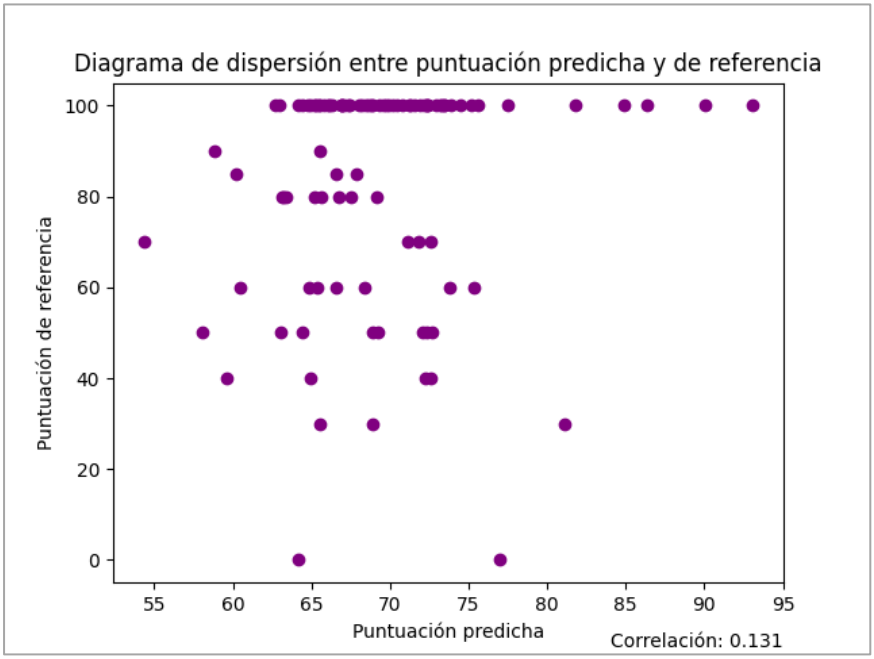

Figura 17: Diagrama de dispersión entre puntuación predicha y de referencia, extrayendo casos [Figura 13] e incluyendo otros 3 códigos de referencia.

Se observa como la correlación empeora [Figura 17] con un 0’131, entonces se deduce que los códigos referencia tienen que pensarse a conciencia y ser útiles para extraer la puntuación automática.

Mirando los códigos de los estudiantes buscando un patrón común, se descubre que los códigos que obtienen más distancia son aquellos que implementan “whiles” en vez de “fors” que es lo que utiliza el código solución. Se prueba de poner únicamente 2 referencias de códigos solución, una correspondiente a la original del profesor y la otra a la original del profesor, pero en vez de utilizar bucles “for” utilizamos bucles “while” y se genera el siguiente resultado [Figura 18].

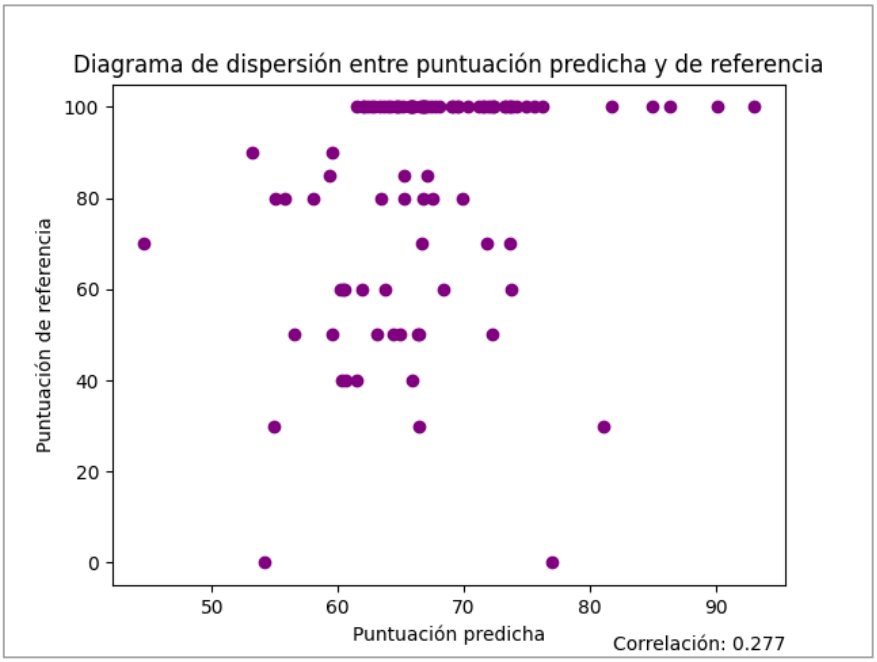

Figura 18: Diagrama de dispersión entre puntuación predicha y de referencia, extrayendo casos [Figura 13] e incluyendo un código de referencia.

En este caso sí que ha mejorado la correlación a un 0’277, augmentando un 0’016 respecto [Figura 13]. Probamos de eliminar esos valores atípicos como en el diagrama de dispersión [Figura 14], para ver si hay mejora.

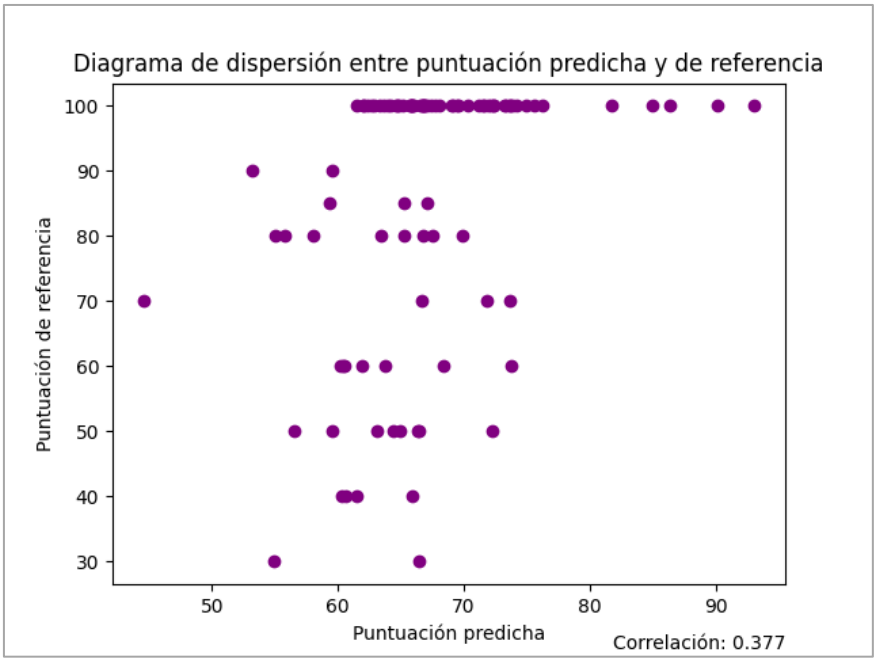

Figura 19: Diagrama de dispersión entre puntuación predicha y de referencia, extrayendo casos [Figura 14] e incluyendo un código de referencia.

Con esta última prueba se mejora considerablemente la correlación llegando a 0’377 [Figura 19]. Con estas pruebas se han podido extraer una serie de conclusiones.

6 Conclusiones y trabajo futuro

6.1 Conclusiones

La conclusión principal a la que se llega es que se ha podido comprobar como comparando los AST no sé puede afirmar que un código funciona correctamente. Que sea estructuralmente igual o muy semejante al código de referencia no implica que el código sea correcto, entendiendo correcto como que sigue la estructura y da el resultado esperado para cualquier ejemplo dado. Esto implica que la hipótesis planteada donde se comentaba que el AST procedente del ejercicio de un estudiante se considerará correcto de forma directamente proporcional a su semejanza al AST de referencia, es decir, al AST del código del profesor, no es del todo cierta.

Esto hace que sea estrictamente necesario utilizar un soporte a esta herramienta para poder utilizarla para evaluar un código automáticamente. Sabiendo cuál es la carencia, el soporte que sería necesario consistiría en añadir una evaluación previa de entrada/salida, para distinguir y tratar de diferente manera los códigos que, pasan las pruebas de ejecución y de resultados, a los que no las pasan.

Por otro lado, la observación realizada a aquellos resultados que se encontraban con una puntuación de referencia menor a 5, se ha podido ver cómo extrayendo parte de esos valores atípicos (eliminando la entrega de esos alumnos de la ejecución) ha aumentado considerablemente la correlación entre los datos. Eso sí, únicamente se han extraído los más evidentes, con gran diferencia entre la puntuación de referencia y la predicha y que se veían fuera del diagrama perfectamente, por no arriesgar a extraer datos necesarios. No se podía saber detalladamente los criterios de evaluación manual para acabar puntuando con un valor menor a 5. Entonces se puede concluir que si se hiciera esa evaluación previa que comentábamos, podríamos mejorar ese resultado.

Dentro de los casos donde la puntuación de referencia es mayor que la puntuación predicha, el caso más claro es no tener suficientes códigos de referencia para evaluar correctamente las estructuras posibles utilizadas al codificar. Aunque hay que tener en cuenta que intentar abarcar las posibles soluciones puede llevarnos a acabar aumentando la puntuación de códigos incorrectamente (sin la evaluación previa que se ha comentado para filtrar los códigos que funcionan correctamente).

Es complicado conseguir una nota excelente evaluando basándonos en los AST. Se han realizado una serie de pruebas para conseguir incrementar la nota predicha de aquellas notas de referencia que tienen el 10 y añadiendo dos pares de soluciones, pero no ha habido un buen resultado. Esto por pruebas que se han realizado se podría solucionar con el preproceso del AST más que añadiendo más soluciones de referencia. Para

conseguir el 10 en la nota predicha literalmente debería ser el mismo código, pero pudiendo modificar el nombre de las variables, ya que no las tiene en cuenta. Entonces sería necesario quizá poner un límite, que llegado a un 8 – 8,5 ya se considere un 10. Aun así, no solucionaría todos los casos.

6.2 Trabajo futuro

El trabajo futuro tendría diferentes vías de desarrollo. Primero sería necesario añadir una evaluación previa de entrada/salida, para distinguir y tratar de diferente manera los códigos que pasan las pruebas de ejecución y de resultados, a los que no las pasan. Ya que añadir esta evaluación previa permitirá hacer pruebas sin tener ese ruido dentro de los datos que no puedes acabar de tratar, pero que altera los resultados.

La distinción se debe a poder decidir que códigos queremos evaluar basándonos en los AST. Lo más lógico sería querer evaluar de esta manera aquellos códigos que pasen correctamente la evaluación previa. Pero, por otro lado, si se dan casos parcialmente correctos (fallo en ejecución /no pasa todas las pruebas / no pasa ninguna prueba), se debería valorar utilizar la evaluación basándose en los AST igualmente, pero como guía. Calculándolo sobre un porcentaje, por ejemplo. Considerando las posibles causas, es decir, consistiría en valorar como tener en cuenta el resultado obtenido de la evaluación basada en AST. El caso más claro se da si se evaluase con un 0 directamente el código si este no pasa exhaustivamente las pruebas previas.

Por otro lado, lo siguiente a añadir y considerando las pruebas realizadas, habría dos formas de afrontar como continuar el trabajo futuro de este trabajo.

Una forma sería enfocarse a crear una herramienta especializada para el procesado de AST, es decir, que pueda llegar a trasformar el AST generado por el código del alumno, de manera que modifique aquellos factores que generan más distancia entre el código solución, pero realmente son similares.

Un claro ejemplo es el caso del ‘while’ y ‘for, generan más distancia, pero hacen lo mismo. También tratar el hecho de que los métodos auxiliares generan un AST más grande y, por lo tanto, generan más operaciones de edición que no es correcto tenerlas en cuenta. Serían muchos factores que valorar, teniendo en cuenta que modificar el AST no es un trabajo simple y rápido.

Por otro lado, se podría hacer pruebas sobre el costo de las operaciones de edición, pero más explícitamente poder tratar que tipo de nodo se está operando para que sea un costo u otro. Por ejemplo, hacer que la eliminación de un ‘for’ sea más costoso que eliminar una simple variable. Se podría valorar también si vale la pena intentar arreglar los errores de sintaxis automáticamente para poder valorar la entrega o no.

Referencias

[1] Si Chuang Li. *Automating Programming Assignment Marking with AST Analysis*, 2018. University of Waterloo.

[2] Peter C Issacson and Terry A Scott. *Automating the execution of student programs. ACM SIGCSE Bulletin*, 1989.

[3] Hery Liberman. *An example based environment for beginning programmers*. *Instructional Science*, 1986.

[4] Wikipedia. *Árbol de sintaxis abstracta*. https://es.wikipedia.org/wiki/Árbol_de_sintaxis_abstracta.

[5] Baeldung. *Tree Edit Distance* (TED). Last updated: June 8, 2023. https://www.baeldung.com/cs/tree-edit-distance.

Librerías utilizadas en el código:

[11] ast – Abstract Syntax Trees. https://docs.python.org/3/library/ast.html.

[12] nbformat – Python API for working with notebook files. https://nbformat.readthedocs.io/en/latest/api.html.

[13] ZSS – Zhang-Shasha: Tree edit distance in Python. https://pythonhosted.org/zss/.

[14] csv – CSV File Reading and Writing. https://docs.python.org/3/library/csv.html.

[15] os – Miscellaneous operating system interfaces. https://docs.python.org/3/library/os.html.

[16] ezodf – EzODF.py Python package. https://github.com/T0ha/ezodf.

[17] matplotlib.pyplot – Matplotlib documentation. https://matplotlib.org/stable/.

[18] pandas – Pandas: powerful Python data analysis toolkit. https://pypi.org/project/pandas/.

[19] unicode - https://pypi.org/project/Unidecode/.

[20] iMec - ¿Qué es el coeficiente de correlación de Pearson?. https://www.cimec.es/coeficiente-correlacion-pearson/.

[21] ChatGPT - https://openai.com/blog/chatgpt.

Anexos

Ejemplo del formato de los resultados obtenidos en la tabla HTML generada:

Nombre_estudiante	Distancia	Número de operaciones	Puntuación predicha	Puntuación referencia
APELLIDO1 APELLIDO2 NOMBRE	109.0	Eliminar: 36 Insertar: 47 Etiquetar: 26 Iguales: 93	66.04	100.0
APELLIDO1 APELLIDO2 NOMBRE	111.0	Eliminar: 23 Insertar: 63 / Etiquetar: 24 / Iguales: 108	68.38	60.0
APELLIDO1 APELLIDO2 NOMBRE	131.0	Eliminar: 23 / Insertar: 85 / Etiquetar: 23 / Iguales: 109	64.78	100.0
APELLIDO1 APELLIDO2 NOMBRE	108.0	Eliminar: 34 / Insertar: 52 / Etiquetar: 22 / Iguales: 99	67.07	100.0
APELLIDO1 APELLIDO2 NOMBRE	96.0	Eliminar: 61 / Insertar: 20 / Etiquetar: 14 / Iguales: 80	64.44	50.0
APELLIDO1 APELLIDO2 NOMBRE	78.0	Eliminar: 33 / Insertar: 28 / Etiquetar: 16 / Iguales: 106	74.51	100.0
APELLIDO1 APELLIDO2 NOMBRE	82.0	Eliminar: 44 / Insertar: 19 / Etiquetar: 19 / Iguales: 92	71.23	100.0
APELLIDO1 APELLIDO2 NOMBRE	128.0	Eliminar: 35 / Insertar: 60 / Etiquetar: 29 / Iguales: 90	62.02	100.0
APELLIDO1 APELLIDO2 NOMBRE	121.0	Eliminar: 37 / Insertar: 60 / Etiquetar: 24 / Iguales: 94	63.66	100.0
APELLIDO1 APELLIDO2 NOMBRE	216.0	Eliminar: 17 / Insertar: 177 / Etiquetar: 21 / Iguales: 117	54.14	80.0

Algunas de las pruebas realizadas:

Pruebas añadiendo variables			
Pruebas realizadas	**Distancia**	**Nota**	**Observaciones**
Si comparo con el mismo código	0.0	100.0	
Se hace que la operación final esté directamente en el return: def **distancia_entre_puntos**(x1, y1, x2, y2): **return** math.sqrt((x2-x1)**2 + (y2 - y1)**2)	6.0	92.0	

Se mueve una operación a una variable
def **distancia_entre_puntos**(x1, y1, x2, y2):
  item1 = (x2-x1)
  distancia = math.sqrt(item1**2 + (y2 - y1)**2)
  return distancia
	13.0	84.71	

Se canvia el nombre de la variable creada anteriormente, pero no se cambia donde se usa:
def **distancia_entre_puntos**(x1, y1, x2, y2):
  i = (x2-x1)
  distancia = math.sqrt(item**2 + (y2 - y1)**2)
  return distancia
	13.0	84.71	OBSERVACIÓN: Si al cambiar el nombre de la variable, no lo cambio de donde se usa, no detecta el error y la distancia es la misma.

Se añade otra variable (ahora tengo 2 variables) pero pero no se sustituye el valor en la operación:
def **distancia_entre_puntos**(x1, y1, x2, y2):
  i = (x2-x1)
  o = (y2 - y1)
  distancia = math.sqrt(i**2 + (y2 - y1)**2)
  return distancia
	22.0	76.6	

Se añade otra variable (ahora hay 2 variables):
def **distancia_entre_puntos**(x1, y1, x2, y2):
  i = (x2-x1)
  o = (y2 - y1)
  distancia = math.sqrt(i**2 + o**2)
  return distancia
	25.0	72.22	

Se añade otra variable para calcular el interior de la raíz cuadrada:
`def distancia_entre_puntos(x1, y1, x2, y2):`
`  i = (x2-x1)`
`  o = (y2 - y1)`
`  s = (i**2) + (o**2)`
`  distancia = math.sqrt(i**2 + o**2)`
`  return distancia`
	37.0	64.76	OBSERVACIÓN: Crear una variable que no utilizo finalmente, genera más distancia.

Se añade otra variable para calcular el interior de la raiz cuadrada:
`def distancia_entre_puntos(x1, y1, x2, y2):`
`  i = (x2-x1)`
`  o = (y2 - y1)`
`  s = (i**2) + (o**2)`
`  distancia = math.sqrt(s)`
`  return distancia`
	31.0	67.37	

Se hace que la operación final esté directamente en el return:
`def distancia_entre_puntos(x1, y1, x2, y2):`
`  i = (x2-x1)`
`  o = (y2 - y1)`
`  s = (i**2) + (o**2)`
`  return math.sqrt(s)`
	26.0	71.11	

Pruebas añadiendo comentarios			
Pruebas realizadas	**Distancia**	**Nota**	**Observaciones**
Si se compara con el mismo código	0.0	100.0	
Si se le añade un pequeño comentario: `# Hola`	0.0	100.0	
Si se le añade un comentario más grande: `# Hola sfsdfs dfsdfsdfsd fsdfdfdfdfdf dfdfdfdfdfdf dfdfdfdf dfdfdfdf fdfdf`	0.0	100.0	
Si le se le añaden tres comentarios más grandes en tres líneas separadas	0.0	100.0	
Si se añade un comentario en vez de que con el "#", de la siguiente manera: `' ' ' Hola ' ' '`	2.0	97.56	

Se prueba que pasa si el comentario es más grande que un simple 'hola': ' ' 'Hola ksdkmdskmfs df s df sd f sdfdfd fdfdfdf dfdfsdfsd sdfsdf sdfsdf sdf ' ' '	2.0	97.56	OBSERVACIÓN: Sea como sea de grande el comentario, no varía en la distancia obtenida.
Se añaden dos comentarios de los añadidos anteriormente para ver que pasa: ' ' 'Hola ksdkmdskmfs df s df sd f sdfdfd fdfdfdf dfdfsdfsd sdfsdf sdfsdf sdf ' ' ' ' ' 'Hola ksdkmdskmfs df s df sd f sdfdfd fdfdfdf dfdfsdfsd sdfsdf sdfsdf sdf ' ' '	4.0	95.24	OBSERVACIÓN: Cada comentario añadido de más, incrementa la distancia en 2 unidades de forma proporcional.

Pruebas extrayendo métodos			
Pruebas realizadas	**Distancia**	**Nota**	**Observaciones**
Si comparo con el mismo código	0.0	100.0	

Se crea un mètodo y se extrae parte de la operación:
def **getItem1**(x2, x1):
  return (x2-x1)
def **distancia_entre_puntos**(x1, y1, x2, y2):
  distancia = math.sqrt(**getItem1**(x2,x1)**2 + (y2 - y1)**2)
  return distancia
	14.0	84.78	

Se usa el método anterior para obtener otra parte de la ecuación:
def **getItem1**(x2, x1):
  return (x2-x1)
def **distancia_entre_punto**s(x1, y1, x2, y2):
  distancia = math.sqrt(g**etItem1**(x2,x1)**2 + **getItem1**(y2,y1)**2)
  return distancia
	17.0	81.72	

Se comprueba si cambiando la longitud del nombre de las variables, o del método influye en la distancia o no.	17.0	81.72	OBSERVACIÓN: No hace variar la distancia ni el hecho de cambiar el nombre de los métodos auxiliares, ni cambiar el nombre de las variables.

Se añade un nuevo método:
def **getItems**(x2, x1):
  return (x2-x1)
def **getSum**(x1, y1, x2, y2):
  return **getItems**(x2,x1)**2 + **getItems**(y2,y1)**2
def **distancia_entre_puntos**(x1, y1, x2, y2):
  distancia = math.sqrt(**getSum**(x1, y1, x2, y2))
  return distancia
	49.0	55.86	OBSERVACIÓN: Lo que haya dentro de los métodos exteriores también influye. Como crear una variable en vez de directamente retornar la operación.

Si se añade ese nuevo método de forma diferente: def **getItems**(x2, x1):
  return (x2-x1)
def **getSum**(item1, item2):
  return item1 + item2
def **distancia_entre_puntos**(x1, y1, x2, y2):
  distancia = math.sqrt(getSum(**getItems**(x2,x1)**2, **getItems**(y2,y1)**2))
  return distancia
	32.0	69.52	

Si se añade ese nuevo método de forma diferente: def **getItems**(x2, x1):
  return (x2-x1)
def **getSum**(item1, item2):
  return (item1**2 + item2**2)
def **distancia_entre_puntos**(x1, y1, x2, y2):
  distancia = math.sqrt(getSum(getItems(x2,x1), getItems(y2,y1)))
  return distancia
	44.0	58.1	

Pruebas modificando el código totalmente para que no sea igual que el de la solución			
Pruebas realizadas	**Distancia**	**Nota**	
Si se compara con el mismo código	0.0	100.0	

Si se cambia el cálculo:
def **distancia_entre_puntos**(x1, y1, x2, y2):
  distancia = (x2-x1)**2 + (y2 - y1)**2
  return distancia
	5.0	93.33	

Si se cambia el cálculo:
def **distancia_entre_puntos**(x1, y1, x2, y2):
  distancia = (x2-x1)**2 + (y2 - y1)
  return distancia
	8.0	88.89	

Si se cambia el cálculo:
def **distancia_entre_puntos**(x1, y1, x2, y2):
  distancia = (x2-x1) + (y2 - y1)
  return distancia
	11.0	84.06	

Si se cambia el cálculo:
def **distancia_entre_puntos**(x1, y1, x2, y2):
  distancia = (x2-x1)
  return distancia
	19.0	68.85	

Si se cambia el cálculo:
def **distancia_entre_puntos**(x1, y1, x2, y2):
  distancia = x2
  return distancia
	23.0	59.65	

Si se cambia el cálculo:
def **distancia_entre_puntos**(x2):
  distancia = x2
  return distancia
	26.0	51.85	

Si se cambia el cálculo: def **distancia_entre_puntos**(a, b): **return** a**2 + b**2	21.0	65.0	

Código totalmente distinto:
def **distancia_entre_puntos**(lista):
  abs_list = []
  for num in lista:
  abs_list.append(abs(num))
  return abs_list
	25.0	64.29	

Código totalmente distinto y extenso:
def **distancia_entre_puntos**(lista):
  conteos = {}
  max_frecuencia = 0
  numero_mas_frecuente = None
  for numero **in** lista:
  if numero in conteos:
  conteos[numero] += 1
  else:
  conteos[numero] = 1
  if conteos[numero] > max_frecuencia:
  max_frecuencia = conteos[numero]
  numero_mas_frecuente = numero
  elif conteos[numero] == max_frecuencia and numero < numero_mas_frecuente:
  numero_mas_frecuente = numero
  return numero_mas_frecuente
	86.0	40.28	

Se coje el código anterior como código de profesor y modifico el del estudiante de la siguiente manera: def **distancia_entre_puntos**(lista): **return** [abs(num) for num in lista]	90.0	26.23	