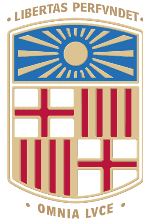


UNIVERSITAT DE BARCELONA

Treball de Final de Grau

GRAU EN ENGINYERIA INFORMÀTICA

Facultat de Matemàtiques i Informàtica
Universitat de Barcelona

AVALUACIÓ AUTOMÀTICA DE CODI FENT SERVIR TÈCNIQUES DE DEEP LEARNING

Martí Altimira Cebrian

Director: Daniel Ortiz Martínez
Realitzat a: Departament de Matemàtiques i informàtica
Barcelona, 20 de desembre de 2023

Agraïments

Vull expressar el més sincer agraïment a totes aquelles persones que han sigut clau a l'hora de realitzar aquest treball de final de grau. Principalment, voldria agrair al meu tutor Daniel Ortiz Martínez per la seva col·laboració i disposició al llarg del projecte. La seva experiència i coneixement previ han sigut de gran importància per dur a terme el projecte de manera efectiva, i disposar de les dades fetes servir per la part experimental i d'obtenció de resultats.

Resum

Català

Aquest treball de final de grau es centra en la possible automatització en l'avaluació de exercicis d'algorísmica en Python mitjançant l'ús de “Code Embeddings” i Deep Learning amb Xarxes neuronals. La nostra hipòtesi es basa en el fet que que l’embedding generat a partir de l’exercici d’un estudiant tindrà una distància respecte a l’embedding de la solució més eficient possible, i en funció d’aquesta es podrà generar una qualificació per a l’exercici. A base d’entrenar aquesta xarxa neuronal amb diferents exercicis i qualificacions esperem que es pugui arribar al punt on les notes proposades per aquesta siguin similars a les que posaria el professor corregint els exercicis i d’aquesta manera reduir la feina que porta corregir tants exercicis per un humà.

Una de les etapes més importants per al càlcul d’aquesta distància entre els Embeddings del codi es la generació d’aquests embeddings, els quals han sigut generats a partir d’un model transformatiu de codi anomenat CodeT5.

La investigació i proves realitzades ens porta a una possible reducció en la feina del corrector però amb necessitat d’entrenar a la xarxa neuronal amb una gran quantitat de dades per a millorar prediccions i resultats, alhora de fer servir aquesta tècnica en conjunt d’altres per a refinar la qualificació i que arribi a ser adequada per a automatitzar la avaluació.

Castellano

Este trabajo de fin de grado se centra en la posible automatización en la evaluación de ejercicios de algorítmica en Python mediante el uso de "Code Embeddings" y Deep Learning con Redes neuronales. Nuestra hipótesis se basa en el hecho de que el embedding generado a partir del ejercicio de un estudiante tendrá una distancia respecto al embedding de la solución más eficiente posible, y en función de esta distancia se podrá generar una calificación para el ejercicio. A través del entrenamiento de esta red neuronal con diferentes ejercicios y calificaciones, esperamos alcanzar un punto en el cual las notas propuestas por esta sean similares a las que pondría el profesor corrigiendo los ejercicios, reduciendo así el trabajo que conlleva corregir tantos ejercicios para un humano.

Una de las etapas más importantes para el cálculo de esta distancia entre los Embeddings del código es la generación de estos embeddings, los cuales han sido generados a partir de un modelo transformador de código llamado CodeT5.

La investigación y pruebas realizadas nos llevan a una posible reducción en el trabajo del corrector, pero con la necesidad de entrenar a la red neuronal con una gran cantidad de datos para mejorar predicciones y resultados, al utilizar esta técnica junto con otras para refinar la calificación y hacer que sea adecuada para automatizar la evaluación.

English

This degree thesis focuses on the potential automation in assessing algorithmic exercises in Python using "Code Embeddings" and Deep Learning with Neural Networks. Our hypothesis is based on the idea that the embedding generated from a student's exercise will have a distance from the embedding of the most efficient possible solution, and based on this distance, a grade can be generated for the exercise. By training this neural network with various exercises and expected grades, we hope to reach a point where the grades proposed by it are similar to those a teacher would assign when correcting exercises, thereby reducing the workload of grading numerous exercises for a human.

One of the crucial stages in calculating this distance between the code embeddings
is the generation of these embeddings, which have been created using a code transformer model called CodeT5.

The research and tests conducted suggest a potential reduction in the grader's workload, albeit with the need to train the neural network with a substantial amount of data to enhance predictions and outcomes when employing this technique alongside others to refine the grading system for automation.

1 Introducció...5
1.1 El projecte...5
1.2 Motivació...6
2 Objectiu...6
3 Estat de l'art...7
3.1 Anàlisi de codi estàtic...8
3.2 Avaluació Entrada/Sortida i proves unitàries...8
3.3 Completat de codi...9
3.4 Avaluació basada en AST...10
3.5 Avaluació de Code Embeddings...11
4 Anàlisi del problema...12
4.1 Arquitectura proposada...12
4.1.1 Generació de Code Embeddings...13
4.1.2 Càlcul de distància entre Embeddings...14
4.1.3 Normalització de distància...15
4.1.4 Generació de qualificació estimada a partir de distància...16
4.1.5 Càlcul de coeficient de correlació de Pearson...16
4.1.6 Entrenament de Xarxa Neuronal...17
4.1.7 Avaluació d'eficàcia de Xarxa Neuronal...17
4.2 Implementació...18
4.2.1 Estructura del codi...18
4.2.2 Paquets i llibreries utilitzades...19
4.2.3 Implementació del codi...20
Import de llibreries i definició de paràmetres inicials...21
Generació d'embeddings...22
Lectura d'embeddings previament generats i obtenció d'asserts...26
Càlcul de distància entre embeddings i solució...28
Estructuració de dades en DataFrames...29
Filtratge de dades i entrenament de MLP...31
Obtenció de mètriques i dades dels MLP...32
5 Resultats i discussió...32
5.1 Dades utilitzades...32
5.2 Proves i resultats...38
5.2.1 Proves amb el Dataset de pràctiques amb CodeT5...38
5.2.2 Proves amb el Dataset de pràctiques amb CodeBERT...42
5.2.3 Proves amb el subconjunt d'exercici Capicua Rotacions...45
5.2.4 Proves amb el subconjunt d'exercici RLE...48
5.2.5 Proves amb els Datasets combinats...51
6 Conclusions i feina futura...54
6.1 Conclusions...54
6.2 Feina futura...55
7 Bibliografia i Referències...56

1 Introducció

1.1 El projecte

En els estudis relacionats amb les ciències de la computació, els alumnes reben constantment tasques de programació per demostrar els coneixements adquirits en les assignatures de manera pràctica. Aquestes tasques poden variar en complexitat però les més comunes solen requerir la creació de funcions o programes simples que a partir de certs paràmetres o variables, aconsegueixin l'objectiu proposat per l'exercici.

Majoritàriament, aquests exercicis s'avaluen mitjançant proves automatitzades simples, on es verifica que la sortida generada per el programa de l'estudiant concorda amb la solució proporcionada per l'avaluador, generalment considerada la solució més eficient. Aquestes proves es solen dur a terme mitjançant "asserts", que retornen "True" si la sortida del programa avaluat coincideix amb la solució esperada, o "False" si es dóna el cas contrari. El problema d'aquest tipus d'avaluació és que és binaria, amb el que només pots saber si el programa funciona correctament o no, sense tenir en compte l'estructura d'aquest, o en cas que no funcioni, si l'estudiant tenia la idea correcta però hi ha hagut errors a l'hora d'escriure la funció per exemple. A causa d'això, s'inclou el factor humà i manual necessari per a una avaluació correcta de la tasca més enllà del resultat binari obtingut per les proves automatitzades.

Recordem que la programació és el procés de crear programes de computador fent servir llenguatges de programació. Això implica l'escriptura de codi que pot ser interpretat per la màquina per realitzar tasques específiques. Hi ha una varietat d'elements importants que formen part de la programació i que són rellevants:

- El **Llenguatge de programació**
- L'**algoritme** utilitzat; un mateix problema pot ser resolt de diferents maneres.
- La **estructura de dades**; maneres d'organitzar i guardar dades en un programa, com poden ser llistes, diccionaris, arrays, o altres tipus d'estructures.
- Les **biblioteques y frameworks** utilitzats.

A partir d'aquí es pot veure que les tasques de programació no tracten únicament d'aconseguir el resultat esperat, si no que hi ha molts altres factors importants a avaluar. Exemples d'aquests factors poden ser la eficiència del programa i el seu temps d'execució o l'ús de memòria del programa.

1.2 Motivació

La revisió manual de codi en un grup reduït d'estudiants pot ser viable, però a mesura que augmenta la quantitat d'estudiants, el procés s'allarga cada cop més, augmentant així el temps d'espera d'aquests alumnes per obtenir les seves qualificacions. Una de les solucions que es fan servir normalment per facilitar aquest procés d'avaluació és la de incrementar el nombre de correctors, però això també comporta desavantatges, com per exemple la inconsistència de les qualificacions, que pot tenir varies causes:

- La **disparitat de criteris de puntuació** entre correctors; quan un corrector té una mitjana de puntuació més alta que un altre en les seves correccions.
- La **interpretació subjectiva dels criteris**; cada corrector pot interpretar la complexitat d'un exercici o la solució proposta per un estudiant de manera diferent.
- El **nivell de rigor o lenitat;** La tendència personal del corrector a ser més rigorós o més flexible en les correccions.
- La **Experiència i preferències personals.**

Per aquests motius, arribem a la conclusió de que és complicat corregir manualment totes les tasques dels estudiants i encara que la revisió humana seria ideal per fomentar bones pràctiques de codificació, es podrien explorar tècniques que automatitzin aquest procés de qualificació.

2 Objectiu

El nostre objectiu és investigar si pot ser viable reduir el temps d'avaluació de tasques d'algorísmica mitjançant l'automatització de les correccions, fent ús de Code Embeddings i xarxes neuronals. La hipòtesi inicial és que a partir de la distància entre els vectors produïts com a Code Embedding de la funció presentada respecte a la solució més eficient de l'exercici, es pugui generar una qualificació.

Un cop obtingudes aquestes qualificacions, calcular la correlació respecte a l'avaluació del corrector humà i realitzar un estudi dels resultats.

Com a altra opció també s'ha considerat entrenar una xarxa neuronal, fent servir els embeddings com a "Feature Vector" i fer que aquesta doni a l'exercici una de 5 categories possibles en funció de com de correcte sigui. Aquestes categories han estat extretes de l'article de Wang[3], apartat 4.2:

- 5 - Correcte i elegant
- 4 - Correcte amb algunes imperfeccions
- 3 - Gairebé correcte i net
- 2 - Incorrecte i confús
- 1 - Incorrecte i horrible

3 Estat de l'art

L'entrega de programes és un mètode utilitzat en l'educació de les ciències de la computació per reforçar i avaluar conceptes pràctics ensenyats a classe.

L'avaluació automàtica de codi és un tema que s'ha estudiat des de diferents punts de vista, amb l'objectiu principal de reduir la intervenció humana a les correccions, aconseguint així una avaluació més objectiva i consistent. Això és degut al fet que un procés automatitzat genera una avaluació idèntica de tots els exercicis entregats, ja que sempre segueix els mateixos criteris, i no entren en joc factors humans com poden ser:

- **Errors humans**; Els correctors poden cometre errors a l'hora de corregir, sigui per descuit, fatiga, o falta d'atenció, el que pot portar a avaluacions inconsistents
- **Volum de feina**; la quantitat d'exercicis a corregir pot afectar a la consistència, quan hi ha molta feina, és possible que els correctors no tinguin temps suficient de revisar cada exercici amb la mateixa atenció.
- **Biaix personal;** Els correctors poden tenir biaixos conscients o inconscients que influeixin en la manera com avaluen els exercicis.
- **Falta d'estàndards clars;** Si no hi ha estàndards establerts per la correcció, com rúbriques clares o criteris d'avaluació específics, els correctors poden avaluar de forma inconsistent.

L'automatització és utilitzada en diferents sistemes, plataformes o entorns de programació. La majoria estan centrades en una tasca o procés en concret per millorar resultats i precisió. Per aquest Treball de final de Grau ens centrarem en aquelles que poden adaptar-se i fer-se servir en una varietat de contextos, ja que ens ofereixen flexibilitat i versatilitat.

3.1 Anàlisi de codi estàtic

L'avaluació estàtica de codi és un procés que analitza el codi font sense executar-lo. Es basa en examinar l'estructura, la sintaxi i les regles del codi per identificar possibles problemes, vulnerabilitats o àrees de millora.

El procés funciona mitjançant eines d'anàlisi estàtica que inspeccionen el codi font en cerca de patrons específics, errors comuns, pràctiques subòptimes o violacions de regles predefinides. Segueix els següents passos:

1. **Anàlisi de codi font:** Les eines d'anàlisi estàtica examinen el codi font línia per línia, tokenitzant i analitzant l'estructura del codi sense executar-lo
2. **Identificació de problemes:** Durant aquest procés, l'eina busca problemes com errors de sintaxi, ús incorrecte de variables, incompliment d'estàndards de codificació, possibles vulnerabilitats de seguretat o patrons que poden portar errors lògics.
3. **Aplicació de regles i estàndards:** Aquestes eines es basen en conjunts de regles predefinides, conegudes com a regles d'anàlisi estàtica, que estableixen les pautes per a una bona pràctica de codificació. Les eines comparen el codi amb aquestes regles per identificar àrees que no compleixen els estàndards definits.
4. **Generació d'informes:** Després de l'anàlisi, l'eina d'avaluació estàtica de codi genera informes detallats que llisten els problemes trobats, proporcionant descripcions, ubicacions en el codi i, en alguns casos, suggeriments per corregir-los.

Aquests anàlisis poden ajudar als desenvolupadors a detectar i corregir errors de manera prematura en el cicle de desenvolupament, però no són capaços de predir si el codi és correcte per a resoldre un problema en concret.

3.2 Avaluació Entrada/Sortida i proves unitàries

L'avaluació entrada/sortida és un concepte fonamental en el desenvolupament de programari que es remunta als primers dies de la informàtica moderna. No hi ha un moment exacte d'inici, ja que va anar evolucionant a mesura que la programació es feia més complexa, però sí que hi ha constància de l'ús d'aquesta tècnica per avaluar programes d'estudiants en l'any 1989 per Peter C. Isaacson i Terry A. Scott[4].

Aquesta tècnica implica provar el comportament d'una part del programa utilitzant diferents entrades i verificant que les sortides generades per aquest siguin correctes. Aquestes entrades poden ser diverses, incloent-hi valors límit, casos normals i situacions de vora.

La tassa de coincidències entre les sortides esperades i les produïdes pel programa s'usa per avaluar la qualitat del programa i determinar el seu funcionament.

A partir d'aquest procés existeixen les proves unitàries, que són petites avaluacions d'entrada/sortida que es centren a comprovar la funcionalitat d'unitats individuals de codi, com funcions, mètodes o classes.

Aquest tipus d'avaluació és molt útil a l'hora d'automatitzar la correcció d'exercicis com a una dada més però no com a resultat únic i final, ja que només aporta informació sobre el resultat del codi i sobre el seu funcionament, però no té en compte el procés, la idea, l'estructura d'aquest i altres factors. Un codi que passi tots els tests no implica treure bona nota, perquè podria ser ineficient o estructurat malament.

3.3 Completat de codi

Un altre mètode d'avaluació és el del completat de codi, on s'ofereix als alumnes el codi d'un programa o funció amb espais en blanc per que els completi. Es poden fer servir com a eines d'aprenentatge iteratiu, ja que permeten als estudiants practicar la seva habilitat de resoldre problemes de codi fonamentant un aprenentatge pràctic i l'assimilació dels conceptes de programació. Això genera un codi petit i rígid, que facilita la avaluació automatitzada i precisa, amb gran flexibilitat de nivells de dificultat.

Els principals inconvenients d'aquest mètode d'avaluació són:

- **Limitació en l'avaluació:** En ocasions, no capturen totes les habilitats dels estudiants, ja que es centren en la resolució d'un problema específic i poden passar per alt altres capacitats o habilitats.
- **Manca de context:** En alguns casos, l'exercici pot no reflectir totalment les situacions del món real.
- **Potencial d'enganxar-se a un patró:** Si els exercicis segueixen un patró massa previsible, els estudiants poden aprendre a resoldre només aquells tipus específics de problemes, limitant la seva capacitat de resolució de problemes més amplis.
- **Limitacions en la creativitat:** En certs casos, aquests exercicis poden limitar la creativitat en la solució de problemes, ja que els estudiants han de completar codi amb un patró preestablert.

3.4 Avaluació basada en AST

L'avaluació de codi basada en AST (Abstract Syntax Tree, o arbre sintàctic abstracte) és una tècnica que es basa en l'estructura jeràrquica del codi font per analitzar, avaluar i prendre decisions sobre el codi. L'AST és una representació estructurada del codi font, que captura la seva sintaxi i estructura lògica sense detalls de format o espai en blanc.

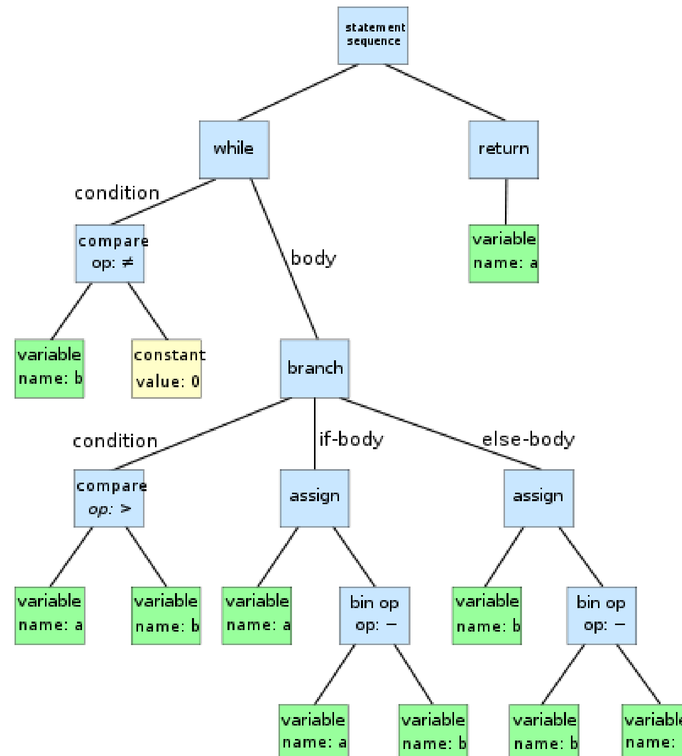


Figura 1. Arbre de sintaxis abstracta d'un codi extret de Wikipedia[5]

En aquest tipus d'avaluació es comparen els AST de dos programes per determinar similituds i diferències en la seva estructura lògica. Això permet avaluar la qualitat i la correcció dels programes de forma objectiva i eficient, sense necessitat d'executar-los.

Es fa servir en diferents àrees com ara l'educació, detecció de plagi, programació i optimització de codi. Permet automatitzar l'avaluació de programes i proporcionar retroalimentació ràpida als estudiants o programadors per mantenir consistència en la correcció.

El principal problema d'aquest tipus d'avaluació és que no es pot afirmar que un codi funciona correctament. Que un codi sigui estructuralment igual o molt semblant al codi de referència no implica que el codi sigui correcte.

3.5 Avaluació de Code Embeddings

L'avaluació de codi a partir de Code Embeddings és una tècnica basada en representar numèricament el codi font i capturar les relacions semàntiques i sintàctiques entre diferents segments de codi.

El procés d'avaluació de codi amb aquesta tècnica té 2 etapes principals:

La primera és el preprocès del codi i la generació dels embeddings, on s'agrupa el codi desitjat en un sol fitxer i es selecciona el model d'embedding que millor s'ajusti al cas d'ús.
Hi ha gran varietat de models preentrenats per a escollir, els més populars són:

- CodeT5
- Code2Vec
- CodeBERT
- CodeSearchNet

La segona etapa és la de l'entrenament de la xarxa neuronal a partir d'aquests embeddings i notes posades per a correctors humans per a la correcció de les tasques, sigui per 1 problema en concret o general per a tota mena de problemes de programació.

Aquest Treball de final de Grau es basa en aquesta tècnica, ja que hi ha molt poca informació respecte d'aquesta aplicació en el camp de la correcció automatitzada i l'ús d'embeddings ofereix molta versatilitat a l'hora d'avaluar diferents tipus de codi i llenguatges de programació.

4 Anàlisi del problema

L'objectiu és automatitzar l'avaluació de codi escrit per alumnes en Python a través de Code embeddings. Per aconseguir-ho hem creat una sèrie de funcions en Python que segueixen el procés necessari per a generar aquests embeddings, processar la informació, i entrenar i avaluar els resultats d'una xarxa neuronal per mesurar la seva eficàcia a l'hora d'assignar qualificacions i prediccions sobre l'estat del codi. En aquest apartat s'explicaran els conceptes necessaris per arribar a la implementació.

4.1 Arquitectura proposada

L'arquitectura que es segueix en el codi está formada per una serie de blocs que contenen els processos necessaris per tractar i transformar les dades:

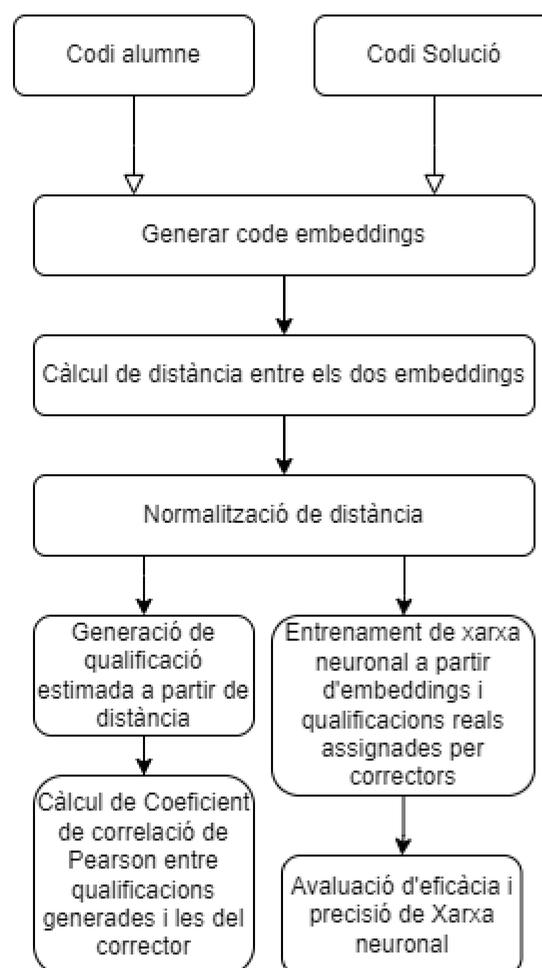


Figura 2: Arquitectura del codi realitzat

Aquest diagrama [Figura 2] mostra el procés que segueixen els programes que es volen avaluar per obtenir una qualificació automatitzada. Primer es generen els embeddings corresponents a partir del model preentrenat escollit. Un cop generats, es calcula la distància entre els 2 vectors i es normalitza per poder generar una qualificació en funció d'aquesta. Un cop generada la qualificació, es calcula la seva correlació de Pearson respecte a la qualificació posada pel corrector humà i es fan servir els embeddings per entrenar una xarxa neuronal amb l'objectiu de poder predir la qualificació d'exercicis.
Després d'entrenar-la, s'avaluen els resultats d'aquesta a l'hora de generar qualificacions automàticament. Més endavant s'entra en profunditat a detallar en què consisteix cada bloc del diagrama exactament.

4.1.1 Generació de Code Embeddings

Els Embeddings de Codi són representacions vectorials d'alta dimensionalitat generades a partir del codi font. Aquestes representacions capturen la semàntica i l'estructura del codi, permetent als models de processament del llenguatge natural entendre i treballar amb codi de manera més eficient.

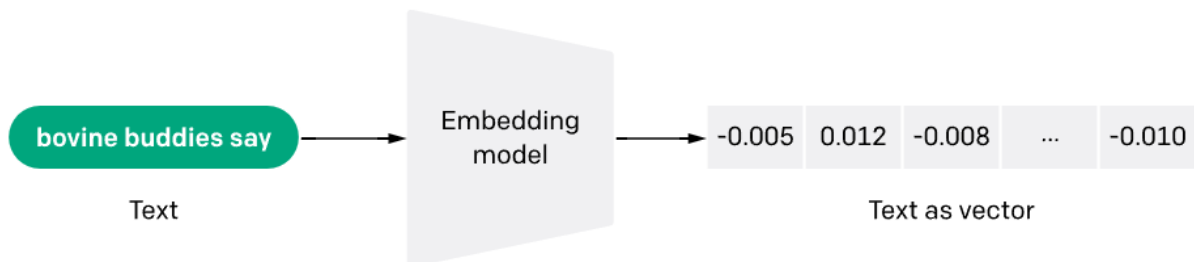


Figura 3: Representació visual de generació de code embedding extreta de OpenAI[8]

El procés de generació d'embeddings de codi és el següent:

1. **Preprocessament del codi:** Es tokenitza el codi en unitats significatives, com ara tokens de paraules, subtokens o fins i tot estructures de codi més grans (com ara funcions o blocs).
2. **Entrenament del model d'embeddings:** S'utilitza un model d'aprenentatge automàtic (com ara transformers) preentrenat en una tasca específica (com pot ser la predicció de màscares, traducció o recuperació d'informació). Aquest model processa el codi per generar embeddings.
3. **Fine-tuning (si és necessari):** En alguns casos, els models preentrenats es reajusten o es tornen a entrenar amb dades específiques del domini per millorar la comprensió i generació del codi en un context particular.

Com s'ha mencionat anteriorment, hi ha diferents models preentrenats per generar Code Embeddings, com ara **CodeT5, Code2Vec, CodeBERT** o **CodeSearchNet.** Per aquest projecte hem destacat 2 models que hem fet servir per a la implementació i els experiments:

CodeT5
CodeT5 és un LLM(Large Language Model), adaptat de T5 (Text-to-Text Transfer Transformer), expecíficament dissenyat per tasques de generació de codi.

Utilitza un enfocament de “text-to-text” on el model rep entrades en forma de text i genera sortides també en format de text, permetent la generació de codi a partir de descripcions en llenguatge natural.

CodeBERT
CodeBERT és una adaptació de BERT (Bidirectional Encoder Representations from Transformers) per a tasques relacionades amb el codi.

El seu enfocament és de codificar el codi font i la informació contextual utilitzant una arquitectura de transformers preentrenada, permetent capturar la semàntica i l’estructura del codi per a tasques com classificació, recuperació d’informació o generació de codi.

4.1.2 Càlcul de distància entre Embeddings

Hi ha varies mètriques comunes per calcular la distància entre Embeddings, entre les més destacades podem trobar les següents:

Distància Euclidiana
La distància Euclidiana és la mesura directa entre dos punts en un espai euclidià. Per a dos vectors x i y de n dimensions, es calcula com:

$$distància\ euclidiana = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

Aquesta fórmula troba la longitud del segment de línia que conecta els dos punts a l’espai de n dimensions.

Distància Manhattan

També coneguda com a distància L1, la distància Manhattan mesura la suma de les diferències absolutes entre les coordenades dels vectors. Per a dos vectors x i y de n dimensions, es calcula com:

$$distància\ Manhattan = \sum_{i=1}^{n} \left| x_i - y_i \right|$$

Aquesta distància es basa en moure's entre punts només al llarg dels eixos, creant una ruta de "manhattan" o "reixa".

Distància Cosinus

La distància Cosinus mesura l'angle entre dos vectors. Per a dos vectors x i y, es calcula com:

$$distància\ Cosinus = \frac{x \cdot y}{||x||\ ||y||}$$

S'utilitza àmpliament en espais de alta dimensió ja que mesura la similitud direccional entre dos vectors independentment de la seva magnitud.

Per aquest projecte s'ha fet servir la **distància euclidiana** degut a la seva simplicitat i familiaritat.

4.1.3 Normalització de distància

Un cop obtinguda la distància de les tasques a evaluar, s'ha de processar per poder valorar-se com a puntuació en l'interval que es troben els resultats. Primer s'ha de normalitzar per a que estigui en un interval de 0 a 100. Per aconseguir això apliquem la següent formula:

$$Dnorm[i] = \frac{distancia[i] - min(distancia)}{max(distancia) - min(distancia)}$$

sent $distancia$ el vector que conté els valors de distància entre cada tasca i la solució, i l'índex de l'element d'aquest vector, $min(distancia)$ el valor mínim del vector, i $max(distancia)$ el valor màxim del vector.

4.1.4 Generació de qualificació estimada a partir de distància

Després d'obtenir el resultat, restem aquesta distància normalitzada a 1, per obtenir la similitud enlloc de la distància, així aconseguint que com a més similar sigui una tasca a la solució óptima, el valor obtingut s'aproparà més a 1. Finalment es multiplica aquest valor per 100 per obtenir una qualificació en el mateix rang que la qualificació manual del corrector.

La formula descrita és la següent:

$$P = (1 - Dnorm) \cdot 100$$

4.1.5 Càlcul de coeficient de correlació de Pearson

Per mesurar la similaritat entre la nota generada de la tasca de l'alumne i la solució, fem servir el coeficient de correlació de Pearson, que és una mesura estadística que avalua la relació lineal entre dues variables. És un nombre que oscil·la entre -1 i 1, on:

- 1 indica una correlació positiva perfecta, la qual cosa significa que a mesura que una variable aumenta, l'altra també augmenta en una relació lineal perfecta
- 0 indica una absencia de correlació lineal entre les variables.
- -1 indica una correlació negativa perfecta, on una variable agumenta a mesura que l'altra disminueix en una relació lineal perfecta.

El coeficient es calcula dividint la covariància de les dues variables pel producte de les seves desviacions típiques individuals, això s'expressa com:

$$\rho x, y = \frac{cov(x,y)}{\sigma x \sigma y}$$

És una mesura útil per comprendre com es relacionen conjunts de dades, però és important tenir en compte que la correlació no implica causalitat. És a dir, encara que dues variables estiguin correlacionades, no necessàriament una causa l'altra.

4.1.6 Entrenament de Xarxa Neuronal

Finalment, per obtenir una predicció més complexa, s'introdueixen les dades dels embeddings a una Xarxa Neuronal per predir la qualificació del corrector a base d'entrenament.

Les Xarxes Neuronals són models computacionals inspirats en el funcionament del cervell humà. Es basen en unitats interconnectades anomenades "neurones" que treballen col·lectivament per processar informació.

La seva estructura bàsica és la següent:

Neurones
En una xarxa neuronal, les "neurones" són les unitats bàsiques que reben entrades, realitzen càlculs i produeixen sortides. Cada neurona té pesos associats amb les seves entrades, que s'ajusten mitjançant l'aprenentatge.

Capes(Layers)
Les neurones estan organitzades en capes. Una xarxa neuronal pot tenir diverses capes:

- **Capa d'entrada:** Reben les dades inicials o les entrades.
- **Capa oculta:** Capes intermèdies entre la capa d'entrada i la de sortida. Processen les dades mitjançant càlculs complexos.
- **Capa de sortida:** Proporciona les prediccions o resultats finals.

Funcionament:

1. **Propagació cap endavant:**
   Les dades es mouen des de la capa d'entrada a través de les capes ocultes fins la capa de sortida. En cada neurona, les entrades es multipliquen per els pesos, se sumen i es passen a través de la funció d'activació
2. **Aprenentatge:**
   Els pesos de les connexions entre neurones es milloren durant l'entrenament de la xarxa neuronal mitjançant tècniques com la retropropagació (Backpropagation). Aquest procés ajusta els pesos per minimitzar l'error entre les prediccions i les sortides reals.

Tipus de Xarxes Neuronals

- **Xarxes Neuronals Convolucionals (CNNs):** Útils per a la visió artificial i anàlisi d'imatges.
- **Xarxes Neuronals Recurrents (RNNs):** Efectives per a dades seqüencials com text o dades temporals.
- **Xarxes Neuronals Generatives (GANs):** Creades per generar noves dades, com imatges realistes.

4.1.7 Avaluació d'eficàcia de Xarxa Neuronal

Hi ha diverses formes d'avaluar l'eficàcia d'una Xarxa Neuronal. Algunes de les principals formes són:

Precisió (Accuracy):
Mesura la proporció de prediccions correctes respecte al total de prediccions realitzades.

Pèrdua (Loss):
Representa la quantitat d'error del model d'entrenament. S'utilitza per ajustar els pesos de la xarxa neuronal. La pèrdua més baixa indica un millor ajust del model a les dades d'entrenament.

Matriu de confusió:
Proporciona una visió detallada de les prediccions correctes i incorrectes del model per a cada classe en problemes de classificació.

AUC-ROC (Àrea sota la corba ROC):
En cas de xarxes neuronals amb només 2 classes com per exemple predir si es passaràn els tests unitaris o no, Mesura la capacitat del model de distingir entre dues classes. La corba ROC és un gràfic de la relació entre la Tassa de vertaders positius (TPR) i la Tassa de falsos positius (FPR). Un AUC-ROC més alt indica un millor rendiment del model.

4.2 Implementació

4.2.1 Estructura del codi

El codi està estructurat en notebooks de python interactius (ipynb). Cada experiment té el seu propi notebook amb codi similar, fent petites variacions per a processar els diferents tipus d'embeddings dels diferents models utilitzats.

És necessari executar les Cel·les en ordre corresponent per carregar totes les variables correctament.
És important definir els paths on es troba el codi dels alumnes, el model a utilitzar i el path on es troba el codi de la solució, així com el nom dels embeddings generats.

Per altre banda, a mesura que s'executa el codi, es guarden diferents fitxers. Aquests fitxers contenen els valors numèrics dels embeddings obtinguts, ja que el procés de generar-los és llarg i necessita una gran quantitat de memòria RAM. Per agilitzar multiples execucions i experiments, la primera vegada que s'executa el codi per a un grup de tasques d'alumnes, es generen aquests fitxers, i si es vol tornar a executar, es poden llegir els embeddings enlloc de generar-los un altre cop, agilitzant el procés de manera eficient. Els fitxers tenen els següents noms:

- **embedding.txt:** Embedding obtingut al passar codi de l'alumne pel model
- **embedding_solution.txt:** Embedding obtingut al passar el codi de la solució pel model.

Els codis avaluats per aquest projecte són del tipus python en fitxers ".py".

4.2.2 Paquets i llibreries utilitzades

S'ha fet ús de diferents llibreries i paquets en el codi, llistades a continuació:

- **os[15]:** llibreria per interactuar amb el sistema operatiu. Proporciona funcionalitats per a tasques com gestionar rutes de fitxers, manipular el sistema de fitxers, obtenir informació sobre l'entorn i molt més.

- **unicodedata[16]:** Proporciona funcions per treballar amb dades Unicode. Ofereix eines per a normalitzar text en codificació unicode, com ara convertir caràcters a la seva forma canònica i treballar amb seqüències de caràcters combinats.

- **transformers[17]:** Eina potent per al processament de llenguatge natural (NLP) basada en models pre-entrenats de xarxes neuronals com els de l'arquitectura Transformer. Proporciona una interfície senzilla per a utilitzar, carregar i ajustar models de NLP de manera eficient. Principalment hem utilitzat dues funcions:
    - **AutoModel:** Facilita la càrrega de models pre-entrenats sense la necessitat de seleccionar manualment el model exacte.
    - **AutoTokenizer:** Permet carregar i utilitzar diferents tipus de tokenitzadors automàticament, adaptant-se als requisits específics del model que s'utilitza

- **numpy[18]:** Eina fonamental per al càlcul científic i computacional. Ofereix funcionalitats per a treballar amb matrius multidimensionals i altres estructures de dades numèriques. Proporciona un conjunt ampli de funcions d'àlgebra lineal, operacions matemàtiques avançades i manipulació de dades.

- **scipy.spatial[19]:** Proporciona eines per a operacions d'anàlisi espacial i càlculs geomètrics en Python. En concret s'ha utilitzat la funció "**distance**" per al càlcul de la distància euclidiana entre dos embeddings.

- **pandas[20]:** Eina per a l'anàlisi de dades que ofereix estructures de dades flexibles i funcionals. Facilita la manipulació i anàlisi de dades tabulars i etiquetades. El seu principal objecte és el DataFrame, que és una estructura bidimensional etiquetada que pot contenir dades de diversos tipus en columnes.

- **torch[21]:** Marc de treball de codi obert per a Deep Learning que ofereix eines per a la creació i entrenament de xarxes neuronals. Està dissenyada per oferir flexibilitat i eficiència en el desenvolupament d'aplicacions de Deep Learning. El nucli de PyTorch és el seu tensor, que és una estructura de dades similar a un array multidimensional que es comporta com les matrius de NumPy però amb la capacitat d'executar càlculs de GPU de manera eficient. Això permet accelerar les operacions matemàtiques necessàries per a l'entrenament de xarxes neuronals i altres tasques d'aprenentatge profund.

- **sklearn.neural_network[22]:** Forma part de la llibreria **scikit-learn** i ofereix eines per a la implementació de models de xarxes neuronals per a tasques de classificació i regressió. Dins d'aquesta llibreria, el **MLPClassifier** (Multilayer Preceptron Classifier) és un classificador basat en xarxes neuronals amb múltiples capes

ocultes. Aquest model és una implementació d'una xarxa neuronal artificial (ANN) que utilitza l'aprenentatge supervisat per a la classificació.

- **sklearn.model_selection[23]:** També part de la llibreria **scikit-learn**, ofereix eines per a la validació i selecció de models en l'aprenentatge automàtic. La funció clau utilitzada en aquest projecte és **train_test_split.** Aquesta funció és utilitzada per dividir un conjunt de dades en subconjunts d'entrenament i prova.

- **sklearn.metrics[24]:** Part de la llibreria **scikit-learn**, ofereix una sèrie de funcions per avaluar el rendiment de models d'aprenentatge automàtic utilitzant diferents mètriques i eines d'avaluació. Les funcions utilitzades en aquest projecte són:
  - **accuracy_score:** Calcula la precisió del model, és a dir, la proporció d'observacions classificades correctament sobre el total d'observacions. És una mètrica bàsica que es pot utilitzar per a models de classificació.
  - **roc_auc_score:** Calcula l'àrea sota la corba ROC (Reciever Operating Characteristic). Aquesta mètrica és utilitzada en problemes de classificació binària i avalua la capacitat discriminativa del model.
  - **confusion_matrix:** Aquesta funció crea una matriu de confusió que mostra el nombre d'observacions classificades correctament i incorrectament per a cada classe. És útil per a visualitzar errors específics de classificació.
  - **classification_report:** Aquesta funció genera un informe detallat amb diverses mètriques com la precisió, la recall, el valor F1 i el suport per a cada classe. Proporciona una visió completa del rendiment del model per a cada classe en problemes de classificació.

- **threading[25]:** Ofereix eines per a la programació concurrent mitjançant fils d'execució (Threads). Un Thread és una forma d'execució seqüencial d'instruccions dins d'un programa i permet la simultaneïtat de tasques.

- **matplotlib.pyplot[26]:** Eina per a la visualització de dades i gràfics. Es basa en **matplotlib**, que és una de les llibreries més populars per a la generació de gràfics en Python. Ofereix una interfície simple i flexible per a crear gràfics de diverses formes i estils. Es poden generar gràfics de línies, barres, de dispersió, histogrames, gràfics de torta i molts altres tipus de visualitzacions de dades.

.

4.2.3 Implementació del codi

Per entendre millor els passos seguits en la implementació, es seguirà l'arquitectura per blocs, amb una explicació prèvia i un exemple visual de cada apartat.

Import de llibreries i definició de paràmetres inicials

Per poder fer servir el codi, es defineixen els path on es troben els fitxers a avaluar, les seves solucions, i en el cas que no sigui la primera vegada que es fan servir aquests fitxers, els paths dels embeddings generats per prèvies execucions del codi per reduir temps d'execució.

```
import os
import unicodedata
from transformers import AutoModel, AutoTokenizer
import numpy as np
from scipy.spatial import distance
import pandas as pd
import torch
import threading
initial_directory = '/content/drive/MyDrive/TFG'
submissions_path = initial_directory + '/Data/submissions_exam'
solution_path_1 = initial_directory + '/Data/solutions_exam/solution_1.py'
solution_path_3 = initial_directory + '/Data/solutions_exam/solution_3.py'
embedded_solution_path_1 = initial_directory + '/Data/solutions_exam/embedding_solution_1.txt'
embedded_solution_path_3 = initial_directory + '/Data/solutions_exam/embedding_solution_3.txt'
```

Generació d’embeddings

Per generar embeddings i asserts, hem fet servir diferents models trobats a Huggingface, en concret CodeT5 i CodeBERT. A continuació es mostra un exemple simple del codi bàsic requerit per generar un Code Embedding fent servir el model preentrenat de CodeT5, extret de la pàgina oficial de HuggingFace[27], i l’aspecte del vector generat:

```
from transformers import AutoModel, AutoTokenizer

checkpoint = "Salesforce/codet5p-110m-embedding"
device = "cuda"  # for GPU usage or "cpu" for CPU usage

tokenizer = AutoTokenizer.from_pretrained(checkpoint,
trust_remote_code=True)
model = AutoModel.from_pretrained(checkpoint,
trust_remote_code=True).to(device)

inputs = tokenizer.encode("def print_hello_world():\tprint('Hello
World!')", return_tensors="pt").to(device)
embedding = model(inputs)[0]
print(f'Dimension of the embedding: {embedding.size()[0]}, with
norm={embedding.norm().item()}')
# Dimension of the embedding: 256, with norm=1.0
print(embedding)
# tensor([ 0.0185,  0.0229, -0.0315, -0.0307, -0.1421, -0.0575,
-0.0275,  0.0501,
#          0.0203,  0.0337, -0.0067, -0.0075, -0.0222, -0.0107,
-0.0250, -0.0657,
#          0.1571, -0.0994, -0.0370,  0.0164, -0.0948,  0.0490,
-0.0352,  0.0907,
#         -0.0198,  0.0130, -0.0921,  0.0209,  0.0651,  0.0319,
0.0299, -0.0173,
#         -0.0693, -0.0798, -0.0066, -0.0417,  0.1076,  0.0597,
-0.0316,  0.0940,
#         -0.0313,  0.0993,  0.0931, -0.0427,  0.0256,  0.0297,
-0.0561, -0.0155,
#         -0.0496, -0.0697, -0.1011,  0.1178,  0.0283, -0.0571,
-0.0635, -0.0222,
#          0.0710, -0.0617,  0.0423, -0.0057,  0.0620, -0.0262,
0.0441,  0.0425,
#         -0.0413, -0.0245,  0.0043,  0.0185,  0.0060, -0.1727,
-0.1152,  0.0655,
```

```
#          -0.0235, -0.1465, -0.1359,  0.0022,  0.0177, -0.0176,
-0.0361, -0.0750,
#          -0.0464, -0.0846, -0.0088,  0.0136, -0.0221,  0.0591,
0.0876, -0.0903,
#           0.0271, -0.1165, -0.0169, -0.0566,  0.1173, -0.0801,
0.0430,  0.0236,
#           0.0060, -0.0778, -0.0570,  0.0102, -0.0172, -0.0051,
-0.0891, -0.0620,
#          -0.0536,  0.0190, -0.0039, -0.0189, -0.0267, -0.0389,
-0.0208,  0.0076,
#          -0.0676,  0.0630, -0.0962,  0.0418, -0.0172, -0.0229,
-0.0452,  0.0401,
#           0.0270,  0.0677, -0.0111, -0.0089,  0.0175,  0.0703,
0.0714, -0.0068,
#           0.1214, -0.0004,  0.0020,  0.0255,  0.0424, -0.0030,
0.0318,  0.1227,
#           0.0676, -0.0723,  0.0970,  0.0637, -0.0140, -0.0283,
-0.0120,  0.0343,
#          -0.0890,  0.0680,  0.0514,  0.0513,  0.0627, -0.0284,
-0.0479,  0.0068,
#          -0.0794,  0.0202,  0.0208, -0.0113, -0.0747,  0.0045,
-0.0854, -0.0609,
#          -0.0078,  0.1168,  0.0618, -0.0223, -0.0755,  0.0182,
-0.0128,  0.1116,
#           0.0240,  0.0342,  0.0119, -0.0235, -0.0150, -0.0228,
-0.0568, -0.1528,
#           0.0164, -0.0268,  0.0727, -0.0569,  0.1306,  0.0643,
-0.0158, -0.1070,
#          -0.0107, -0.0139, -0.0363,  0.0366, -0.0986, -0.0628,
-0.0277,  0.0316,
#           0.0363,  0.0038, -0.1092, -0.0679, -0.1398, -0.0648,
0.1711, -0.0666,
#           0.0563,  0.0581,  0.0226,  0.0347, -0.0672, -0.0229,
-0.0565,  0.0623,
#           0.1089, -0.0687, -0.0901, -0.0073,  0.0426,  0.0870,
-0.0390, -0.0144,
#          -0.0166,  0.0262, -0.0310,  0.0467, -0.0164, -0.0700,
-0.0602, -0.0720,
#          -0.0386,  0.0067, -0.0337, -0.0053,  0.0829,  0.1004,
0.0427,  0.0026,
```

```
#          -0.0537,  0.0951,  0.0584, -0.0583, -0.0208,  0.0124,
0.0067,  0.0403,
#           0.0091, -0.0044, -0.0036,  0.0524,  0.1103, -0.1511,
-0.0479,  0.1709,
#           0.0772,  0.0721, -0.0332,  0.0866,  0.0799, -0.0581,
0.0713,  0.0218],
#         device='cuda:0', grad_fn=<SelectBackward0>)
```

Es pot veure l'aspecte dels Code embeddings generats per aquest codi per a l'exemple de la funció "print_hello_world()".

Per al nostre codi, s'ha partit d'aquest exemple i s'ha iterat sobre tots els subdirectoris continguts en el directori definit per on es troben les tasques dels alumnes. Per cada un d'aquests directoris, es busca un fitxer anomenat "submission.py", on es trobarà la tasca de l'alumne, i es llegeix aquest fitxer, generant l'embedding i guardant els valors del vector obtingut en un fitxer en el mateix subdirectori per a poder llegir els embeddings en un futur sense la necessitat de gastar memòria en la generació d'aquests:

```
submission_vectors_1 = []
submission_names = []
current_submission_code = ''

#Initialize model and tokenizer
checkpoint = "Salesforce/codet5p-110m-embedding"
device = "cpu"
tokenizer = AutoTokenizer.from_pretrained(checkpoint,
trust_remote_code=True)
model = AutoModel.from_pretrained(checkpoint,
trust_remote_code=True).to(device)

#RESETTING INITIAL DIRECTORY
print(initial_directory)
os.chdir(initial_directory)
print("changed path to: " , os.getcwd())

#change to submissions path
os.chdir(submissions_path)
print("changed path to: " , os.getcwd())

#iterate over all submissions and embed them
file_list = os.listdir()
```

```
for item in file_list:
    print(item)
    print("VECTOR LIST SIZE: " , len(submission_vectors_1))
    folder_path = os.path.join(submissions_path, item)

    if os.path.isdir(folder_path):
        submission_file_name = find_submission_file(folder_path,
"submission_1.py")
        with open(submission_file_name, "r", encoding="utf-8") as
sub_file:
            current_submission_code = sub_file.read()
        encoded_submission =
tokenizer.encode(current_submission_code,
return_tensors="pt").to(device)
        embedded_submission = model(encoded_submission)[0]
        submission_vectors_1.append(embedded_submission)
        submission_names.append(item)

        # Save embedded_submission to a file
        embedding_file_path = os.path.join(folder_path,
"embedding_submission_1.txt")
        with open(embedding_file_path, "w", encoding="utf-8") as
embedding_file:
            for value in
embedded_submission.detach().cpu().numpy():
                embedding_file.write(f"{value.item()}\n")
```

A continuació es fa el mateix amb el codi de la solució proposada per el corrector, que es troba en un directori diferent, definit prèviament. Es guarden tots els embeddings en fitxers de text ja que al tenir moltes tasques per avaluar, en generar-los s'omple ràpidament la memòria, fent difícil l’execució de qualsevol altre tipus de codi.

Lectura d’embeddings previament generats i obtenció d’asserts

Com s’ha vist en l’apartat anterior, és eficient llegir els embeddings dels fitxers “txt” per estalviar memòria i temps d’execució un cop han sigut generats. Per fer això es torna a iterar sobre els subdirectoris de les entregues, llegint els fitxers d’embeddings i guardant els vectors en una llista. També s’executen els fitxers “.py” que contenen el codi dels alumnes, per veure si es passen els “asserts” amb un timeout de 30s en cas que hi hagi loops infinits. Al final de la iteració es tenen llistes amb els vectors dels embeddings, els noms i cognoms dels alumnes i els resultats de l’execució dels asserts a partir del codi de les tasques:

```
submission_vectors_1 = []
submission_names = []
assert_list_1 = []

os.chdir(submissions_path)
print("changed path to: ", os.getcwd())

file_list = os.listdir()
for item in file_list:
    folder_path = os.path.join(submissions_path, item)
    embedded_submission = []
    assert_result = False
    if os.path.isdir(folder_path):
        submission_file_name = find_submission_file(folder_path,
"embedding_submission_1.txt")
        with open(submission_file_name, 'r') as file:
            for line in file:
                value = float(line)
                embedded_submission.append(value)

        python_script_file_name =
find_submission_file(folder_path, "submission_1.py")
        if os.path.exists(python_script_file_name):
            try:
                with open(python_script_file_name, 'r') as
script_file:
                    script_code = script_file.read()
                    local_namespace = {}
                    exec(script_code, local_namespace)

                    expected_results = [True, False, True]
```

```
                    # Run each test with a timeout of 30 seconds
                    tests = [
                        ('es_capicua_rotacions', ('ocattac',)),
                        ('es_capicua_rotacions', ('aabcdb',)),
                        ('es_capicua_rotacions', ('ccbaab',))
                    ]
                    results = [

run_test_with_timeout(local_namespace[test], args, 30) for test,
args in tests
                    ]

                    # Check the results of the tests
                    assert_result = all(result == expected for
result, expected in zip(results, expected_results))

            except Exception as e:
                print(f"Error running 'test()' function in
{python_script_file_name}")
                print(f"INDEX: {len(assert_list_1)}")
                print(f"Error type: {type(e).__name__}")

        embedded_submission = np.array(embedded_submission)
        submission_vectors_1.append(embedded_submission)
        submission_names.append(item)
        assert_list_1.append(assert_result)
```

Càlcul de distància entre embeddings i solució

Arribats a aquest punt, s'apliquen els càlculs comentats als anteriorment [apartat 4.1.2] per calcular la distància Euclidiana entre cada un dels embeddings de la llista de tasques i l'embedding de la solució, iterant sobre ells i guardant els valors. Un cop calculades les distàncies, es torna a iterar sobre elles, normalitzant-les [apartat 4.1.3] i generant una qualificació [apartat 4.1.4] en funció del valor obtingut.

```
submission_distance_values = []
submission_mapped_grades = []
normalized_distance_values = []
# Define the range for mapping (0 to 2 for cosine distance, 0 to
10 for the grade)


for i in range(0, len(submission_vectors)):
  print("NAME: " + submission_names[i])
  dist = np.linalg.norm(solution_embedding -
submission_vectors[i])
  submission_distance_values.append(dist)
  # Use linear interpolation to map the cosine distance to the
grade
  grade = (1 - dist) * 100
  print("Distance between Solution and " , submission_names[i],
"'s submission: ", dist)
  print("Mapped Grade:", grade)

max_dist = np.max(submission_distance_values)
min_dist = np.min(submission_distance_values)

for i in range(0, len(submission_distance_values)):
  print("distance: ", submission_distance_values[i])
  normalized_distance = (submission_distance_values[i] - min_dist)
/ (max_dist - min_dist)
  print("Normalized Distance: ", normalized_distance)
  grade = (1 - normalized_distance) * 100
  normalized_distance_values.append(normalized_distance)
  submission_mapped_grades.append(grade)
  print("GRADE: ", grade)
```

Estructuració de dades en DataFrames

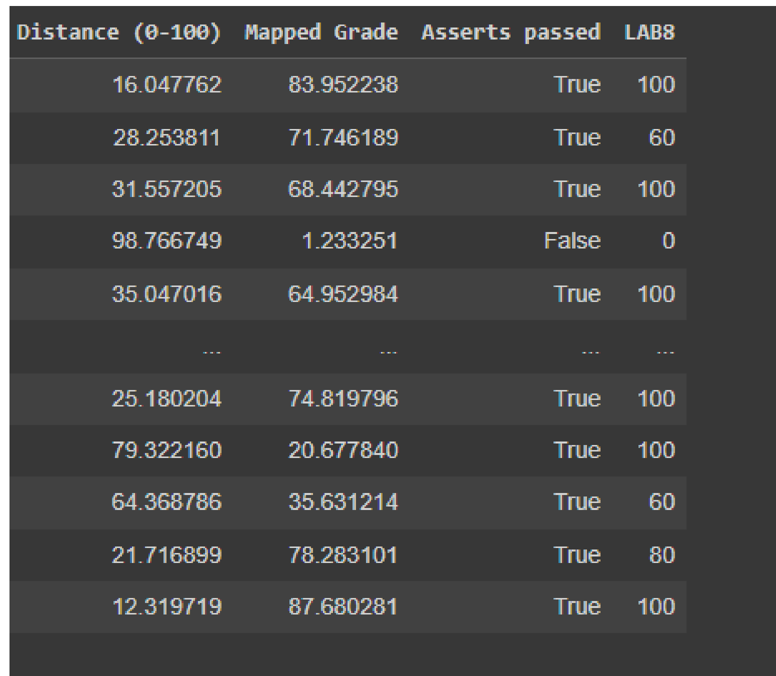
Un cop obtingudes totes les dades necessàries, s'estructuren en DataFrames per a poder comparar-les amb les qualificacions d'un corrector humà, que es troben en fitxers Excel/CSV. Com s'ha mencionat anteriorment, es fa servir la llibreria **pandas**[20] per llegir-los i poder gestionar les dades amb més facilitat degut a totes les utilitats i funcions dels DataFrames. Primer es normalitzen els noms dels alumnes obtinguts dels subdirectoris amb les tasques, ja que poden tenir diferents caràcters UNICODE respecte als noms continguts en els fitxers Excel, així que s'aplica una funció de normalització a cada un d'ells.

Un cop normalitzats, s'ajunten els noms, la distància normalitzada, la nota generada i el resultat dels Asserts de cada tasca en un DataFrame. L'aspecte d'aquest és el següent:

Distance (0-100)	Mapped Grade	Asserts passed
16.047762	83.952238	True
28.253811	71.746189	True
31.557205	68.442795	True
98.766749	1.233251	False
35.047016	64.952984	True
...	...	...
79.322160	20.677840	True
64.368786	35.631214	True
21.716899	78.283101	True
12.319719	87.680281	True
49.327306	50.672694	True

Figura 4. DataFrame generat per el codi

A continuació, es llegeix el fitxer amb les qualificacions assignades pel corrector humà, es normalitzen els noms d'aquest, i es fa un **inner merge** per aconseguir un DataFrame que contingui totes les dades, generades pel codi i pel corrector, per als alumnes que siguin als 2 DataFrames.



Distance (0-100)	Mapped Grade	Asserts passed	LAB8
16.047762	83.952238	True	100
28.253811	71.746189	True	60
31.557205	68.442795	True	100
98.766749	1.233251	False	0
35.047016	64.952984	True	100
...	...	...	...
25.180204	74.819796	True	100
79.322160	20.677840	True	100
64.368786	35.631214	True	60
21.716899	78.283101	True	80
12.319719	87.680281	True	100

Figure 5. DataFrame conjunt després de l'inner merge

En aquest exemple en concret, es pot veure a "Mapped Grade" la qualificació generada per el nostre codi, i a "LAB8", la donada pel corrector.

A partir d'aquestes dues columnes, es calcula el coeficient de correlació de Pearson per veure la seva similitud en puntuació.

Filtratge de dades i entrenament de MLP

Per crear i entrenar el classificador MLP, cal abans fer un filtratge de les dades, eliminant aquelles entregues les quals els alumnes hagin deixat en blanc/no entregat. En aquest projecte s'han fet 2 tipus de MLPs en funció de les dades utilitzades.

El primer tipus de MLP és un classificador de 2 classes per determinar si el codi d'un alumne passarà els asserts o no sense la necessitat de ser executat. Aquest s'ha fet servir en les dades les quals s'havien qualificat pel corrector de forma convencional, sense tenir en compte el projecte, amb el que només es tenia la nota del 0 al 10. Els "feature vectors" són els embeddings del codi.

L'altre tipus de MLP és un classificador de 5 classes. Aquestes classes són les mateixes a les establertes a l'article de Wang[3]:

- **5 - Correcte i elegant:** El codi passa tots els casos de prova i l'estil és net i clar. No hi ha variables ni línies de codi redundants, els noms de les variables estan molt estandarditzats.
- **4 - Correcte amb algunes imperfeccions:** Una implementació correcta però sovint acompanyada d'un estil deficient o una solució complexa.
- **3 - Gairebé correcte i net:** El codi no passa tots els casos de prova, però la lògica general és la mateixa que la solució correcta, i l'estil del codi és net i clar.
- **2 - Incorrecte i confús:** El codi és incorrecte i molt diferent de la solució correcta. L'estil de codi no és bo i la lògica sembla confusa.
- **1 - Incorrecte i pèssim:** El codi és incorrecte i l'estil de codi és horrible.

Per poder entrenar aquest classificador, han fet falta dades obtingudes per un corrector humà, que ha tingut en compte aquestes categories quan corregint exercicis, i no només ha assignat una qualificació del 0 al 10, sinó que també ha assignat una d'aquestes a cada una de les entregues dels alumnes.

S'han fet 2 diferents versions d'aquest classificador: Una versió universal per a qualsevol tipus de codi i una altra específica per a una funció/programa en concret, per veure si és millor tenir un classificador generalitzat amb moltes dades d'entrenament, o diferents classificadors especialitzats en problemes en concrets, tenint menys dades d'entrenament.

La primera versió del classificador rep com a **"feature vector",** els embeddings tant de l'entrega de l'alumne com els de la solució més eficient, concatenats ja que, al ser universal per a qualsevol tipus de problema, cal proporcionar dades extra per a cada cas en concret.

Per a la segona versió, especialitzada per a 1 problema en concret, només es proporcionen els embeddings de l'entrega de l'alumne, ja que totes les prediccions i dades d'entrenament, tindran la mateixa solució òptima degut a què són tots part del mateix exercici.

Obtenció de mètriques i dades dels MLP

Un cop entrenats tots els MLP, s'obtenen mètriques de cada un. Aquestes mètriques són les mencionades anteriorment [apartat 4.1.7].

Pels classificadors de 2 classes dels asserts, s'obté la Accuracy tant de training com de test, loss i l'AUC-ROC.

Pels classificadors de més de dues classes, s'obté a més a més la Matriu de confusió.

5 Resultats i discussió

En aquest apartat es parlarà de les dades utilitzades, mètriques, proves realitzades, i resultats obtinguts amb els que s'arribarà a les conclusions.

5.1 Dades utilitzades

Les dades que s'han utilitzat per les proves realitzades han sigut, principalment, entregues de pràctiques d'alumnes de l'assignatura d'Algorísmica i exercicis de l'examen parcial de la mateixa assignatura, corregits pel professor tenint en compte les categories mencionades anteriorment a sobre de la nota entre el 0 i el 10.

De les entregues de pràctiques, s'ha fet servir l'exercici 4.7 del Capítol 4, Algorismes i Text, que consisteix en crear una funció anomenada "**subcadena_mes_llarga**", que identifica la subcadena més llarga sense cap caràcter repetit. A continuació es mostra la solució proporcionada pel corrector, que es compara amb totes les entregues per determinar la qualificació d'aquestes:

```
def subcadena_mes_llarga(cadena):
    """
    Aquesta funció identifica la subcadena més llarga sense cap
caràcter repetit.

    Parameters
    ----------
    cadena: string
        Cadena donada

    Returns
    -------
    subcadena: string
        Subcadena més llarga sense caràcters repetits
    """
```

```
    n = len(cadena)

    # fem un diccionari amb els caràcters
    diccCaracters = {caracter: False for caracter in cadena}

    # la solució es basa en una finestra que es redefineix cada
cop
    # que troba un caràcter repetit
    iniciFinestra = 0

    # inici i final de la finestra, el final és el darrer caràcter
vist
    # el principi és el primer caràcter no repetit des del final
de la
    # finestra
    iniciSubcadena = 0
    finalSubcadena = 0

    # inici i final de la subcadena més llarga trobada fins al
moment
    for finalFinestra in range(0, n):

        if diccCaracters[cadena[finalFinestra]]:
            # si el caràcter ja hi era
            while (cadena[iniciFinestra] !=
cadena[finalFinestra]):
                diccCaracters[cadena[iniciFinestra]] = False
                iniciFinestra += 1
                # desplacem la finestra a partir de la
                # primera aparició del caràcter, sense incloure'l.
            iniciFinestra += 1

        else:
            diccCaracters[cadena[finalFinestra]] = True
            # anotem el caràcter com a existent
            finalFinestra += 1
            # com que no hi ha repeticions augmentem la finestra
            if finalSubcadena - iniciSubcadena < finalFinestra -
iniciFinestra:
                # revisem que la nova subcadena no sigui més
llarga
```

```
                # que la que teniem guardada
                iniciSubcadena = iniciFinestra
                finalSubcadena = finalFinestra

    return cadena[iniciSubcadena:finalSubcadena]
```

Per determinar el funcionament correcte del codi de cada alumne, s'han executat 3 asserts:

```
assert subcadena_mes_llarga('lacadenamesllarga') == 'namesl'
assert subcadena_mes_llarga('mesllarga') == 'mesl'
assert subcadena_mes_llarga('aaa') ==  'a'
```

Aquestes han sigut les dades en les quals s'ha basat la versió inicial del codi. Les entregues són d'anys anteriors i no van ser corregides tenint en compte aquest projecte, amb el que només disposem de la qualificació del 0 al 10 i el resultat dels asserts, pel qual l'únic classificador a entrenar per aquest Dataset és el que determina si es superaran els asserts o no. En total hi ha **108** entregues de diferents alumnes i el codi de referència que s'acaba de mostrar, utilitzat per calcular la distància entre els embeddings d'aquests. A partir d'aquestes dades i les qualificacions manuals del corrector humà per a cada entrega, s'han pogut obtenir uns resultats inicials sobre la correlació entre elles per poder refinar el codi creat.

Més endavant en el desenvolupament del projecte, durant els exàmens parcials del semestre, el tutor del projecte va corregir 2 exercicis de l'examen d'algorísmica, els quals demanaven la creació de funcions amb diferents objectius. Aquests exercicis han sigut corregits tenint en compte aquest projecte, amb el que a sobre de la qualificació entre el 0 i el 10, també se'ls hi va donar una categoria de les 5 definides anteriorment a l'article de Wang[3].

El primer exercici de l'examen utilitzat és el següent:

Escriu una funció que, donada una cadena de caràcters, determini si en aplicar rotacions de lletres, és capicua. Per resoldre aquest problema has d'anar creant paraules rotades progressivament i validant si són capicua amb un algorisme auxiliar.
Una rotació consisteix a situar l'última lletra en la primera posició o viceversa. Per exemple, en la cadena 'ABCD' les rotacions serien: ['DABC', 'CDAB', 'BCDA', 'ABCD'].

La solució proporcionada com a referència i utilitzada per determinar la distància entre els embeddings de les entregues ha sigut la següent:

```
def es_capicua(paraula):
    n = int(len(paraula)/2)
    capicua = True
    for i in range(n):
        if paraula[i] != paraula[len(paraula)-i-1]:
            return False

    return capicua

def es_capicua_rotacions(paraula):
    """
    Aquesta funció determina si una paraula és capicua aplicant
rotacions.

    Parameters
    ----------
    paraula: string
        La paraula a determinar

    Returns
    -------
    b: bool
        true si la paraula és capicua, false altrament
    """
    i, j = 0, len(paraula)
    trobada = False
    paraulanova = paraula + paraula

    while i <= j and not(trobada):
        trobada = es_capicua(paraulanova[i:i+j])
        i += 1

    if trobada:
        print(f"La paraula {paraula} és capicua rotada", end="")
        print(f" perquè {paraulanova[i-1:i+j-1]} és capicua")
```

```
    else:
        print(f"La paraula {paraula} no és capicua rotada")

    return trobada
```

Els asserts executats per determinar el correcte funcionament de les funcions entregades han sigut aquests:

```
assert es_capicua_rotacions("ocattac") == True
assert es_capicua_rotacions("aabcdb") == False
assert es_capicua_rotacions("ccbaab") == True
assert es_capicua_rotacions("aa") == True
assert es_capicua_rotacions("a") == True
assert es_capicua_rotacions("amormao")==False
```

L'altre exercici de l'examen utilitzat ha sigut l'exercici 3:

Run Length Encoding *(RLE) és un algorisme de compressió de dades sense pèrdua que agrupa els valors repetits amb el nombre de vegades que es repeteixen per optimitzar la memòria. Per exemple, suposem que tenim la cadena:*

BBBBBBBNBBBBBBBBBBBBBNNNBBBBBBBBBBBBNBBBBBBBBB

Aquesta cadena es pot comprimir en la cadena B7N1B13N3B12N1B9, i s'interpreta de la manera següent: 7 caràcters blancs, 1 de negre, 13 de blancs, 3 de negres, 12 de blancs, 1 de negre i 9 de blancs. D'aquesta manera podem reconstruir la cadena original sense pèrdua d'informació.

Escriu una funció rle que donat un text de lletres ASCII(A-Z) el codifiqui utilitzant l'algorisme RLE.

La solució proporcionada com a referència i utilitzada per determinar la distància entre els embeddings de les entregues ha sigut la següent:

```
def rle(text):
    """
    Aquesta funció retorna un text codificat segons run length
encoding.

    Parameters
    ----------
    text: string
        text a codificar

```

```
    Returns
    -------
    text: string
        text codificat
    """

    lletraactual = text[0]

    # Primera lletra
    comptador = 0

    # Iniciem el text codificat
    textcodificat = lletraactual

    for caracter in text:
        # Comptant quantes ocurrències hi ha
        if caracter == lletraactual:
            comptador += 1

        # resetegem i acumulem
        else:
            lletraactual = caracter
            textcodificat = textcodificat + str(comptador) +
lletraactual
            comptador = 1

    textcodificat = textcodificat + str(comptador)

    return textcodificat
```

Els asserts executats per determinar el correcte funcionament de les funcions entregades han sigut aquests:

```
assert rle("ABBBBNNNEEEDDDZZAAAAA") == 'A1B4N3E3D3Z2A5'
assert rle("BBBBBBBBBBBBBBBBBWWWWWWZAAA") == 'B17W6Z1A3'
```

Hi ha hagut un total de **140** entregues d'aquest examen parcial, el que ens dona **280** mostres entre els 2 exercicis. Com s'ha comentat, es disposa també de les qualificacions manuals realitzades pel corrector humà, a sobre de les categories assignades a cada una de les funcions per poder entrenar la Xarxa neuronal.

5.2 Proves i resultats

5.2.1 Proves amb el Dataset de pràctiques amb CodeT5

Les primeres proves realitzades s'han dut a terme amb el Dataset de pràctiques d'anys anteriors, ja que encara no s'havia fet el parcial d'algorísmica i no es disposava de mostres corregides i categoritzades amb aquest projecte en ment.

Predicció de qualificació

Un cop processats els embeddings de les 108 tasques i generades les seves qualificacions, s'obté el diagrama de dispersió següent respecte a les notes del corrector humà:

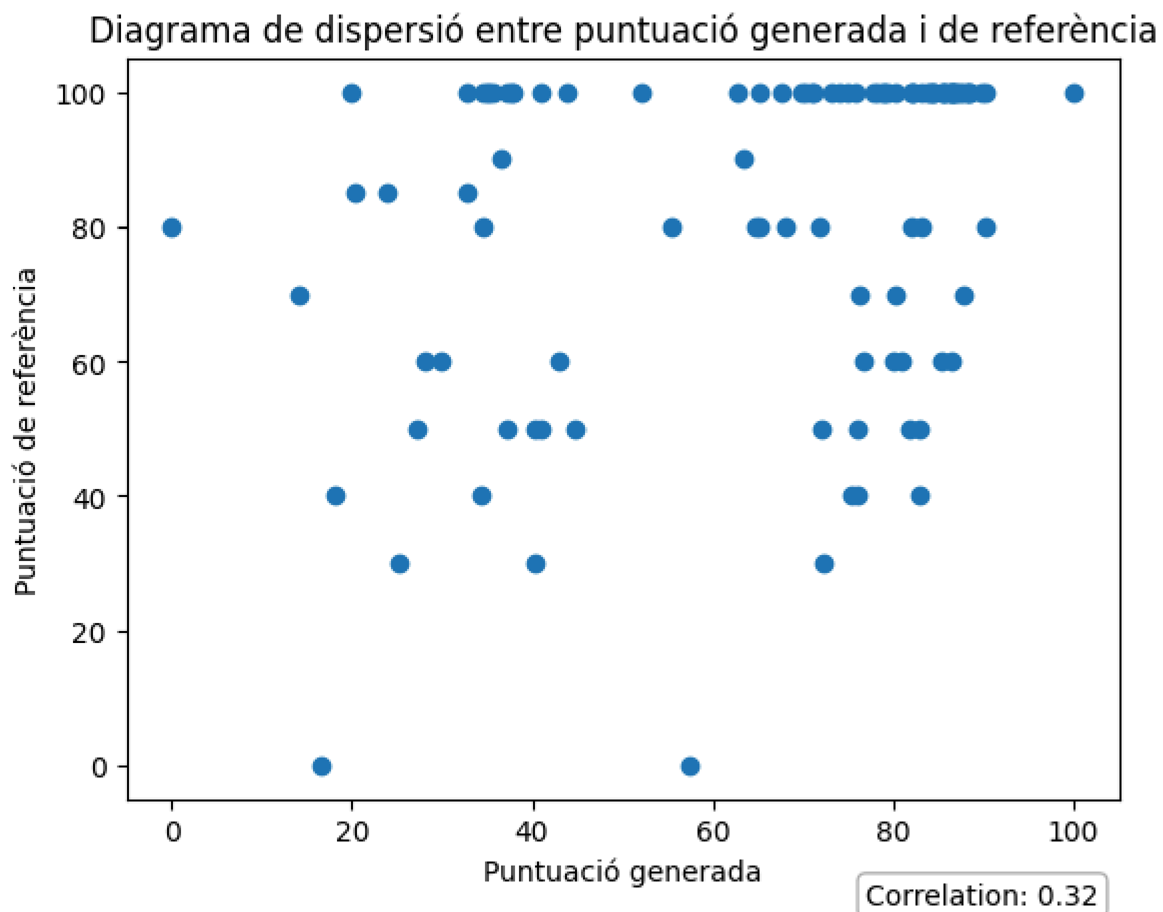


Figure 6. Diagrama de dispersió de les qualificacions de pràctiques

Podem veure també el valor del coeficient de correlació de Pearson entre les qualificacions generades i les de referència, que és 0.32, una correlació moderadament positiva.

Es pot veure que una gran quantitat de les mostres han obtingut una qualificació de "100" per part del corrector humà, i degut a com es generen les qualificacions en el codi, és molt difícil que assigni aquesta puntuació a una tasca.

Predicció de resultats d'asserts
Després d'obtenir les qualificacions, es procedeix a entrenar el clasificador MLP, però com que no es tenen les categories assignades a cada tasca pel corrector humà, aquests primers MLPs només determinaràn si un codi passarà o no els asserts sense ser executat. Per això dividim les mostres en datasets de training i de test, amb una proporció de 0.8 i 0.2 respectivament i entrenem el **MLPClassifier**.

Inicialment s'obté una **Accuracy** de 0.95 i una **Loss** de 0.006, però al avaluar les dades de les primeres 20 iteracions d'entrenament s'obtenen els següents resultats:

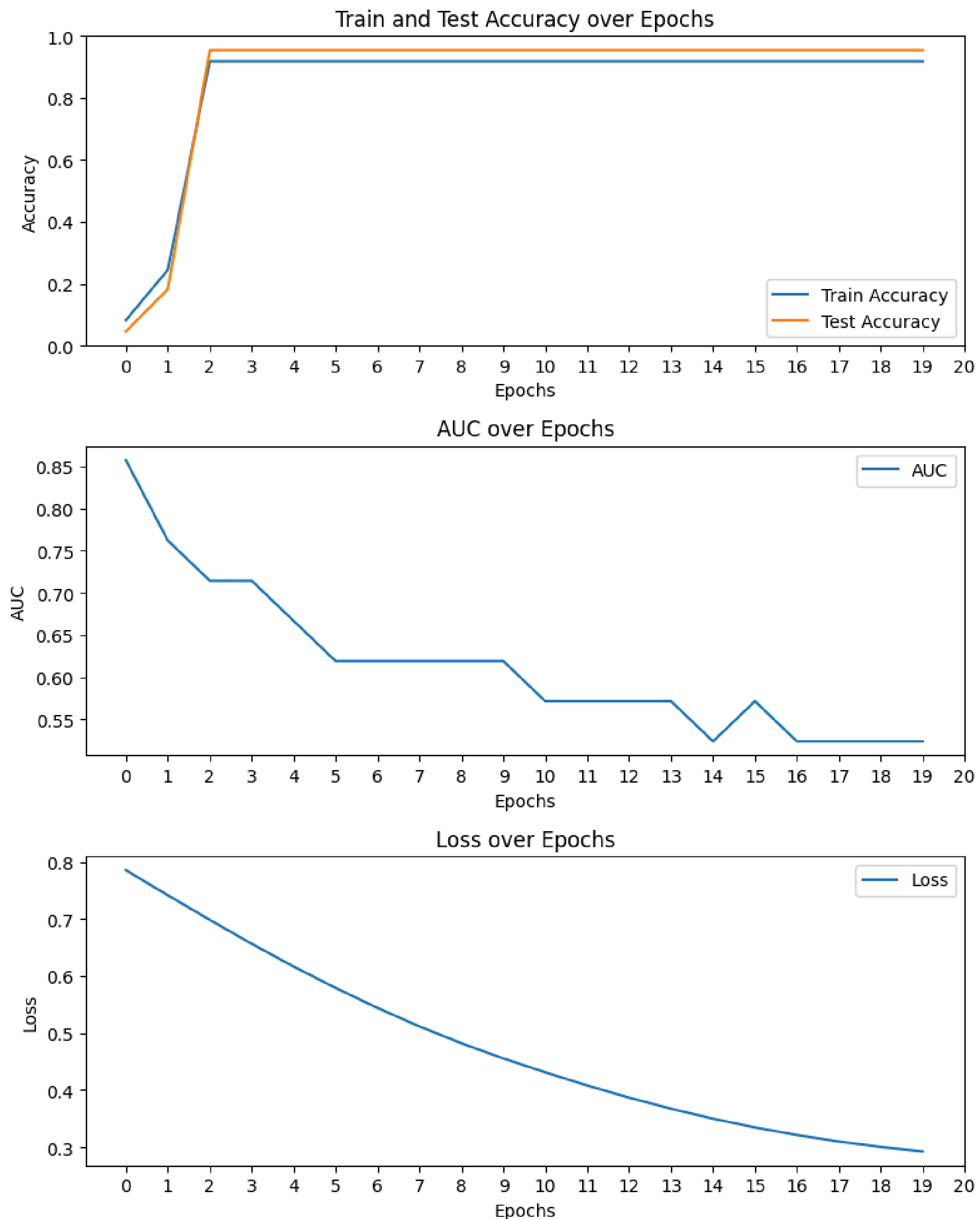


Figura 7. Dades de les primeres 20 iteracions del MLP per pràctiques

Es pot veure que a partir de la iteració entrenament número 3, no hi ha cap millora a l'accuracy tant de entrenament com de test, i la AUC s'estabilitza al voltant de 0.55, que vol dir que l'habilitat del model de discriminar entre les dues classes no es millor que si es determinés a l'atzar. Examinant les dades utilitzades com a mostres, es pot veure que la gran majoria passen els asserts: **100** tasques els passen i **8** no. Podem determinar que hi ha sobreentrenament de la xarxa neuronal degut al tamany de les mostres, de les quals la immensa majoria pertanyen a 1 categoria, mentre que menys del 8% pertanyen a l'altra.

Amb l'objectiu d'intentar millorar aquests resultats, s'ha entrenat un altre MLP que enlloc de rebre com a **feature vector** l'embedding del codi de la tasca, rep la resta entre aquest i l'embedding de la solució:

```
X_with_distance = [v - solution_embedding for v in
submission_vectors]
```

Els resultats es mostren a continuació:

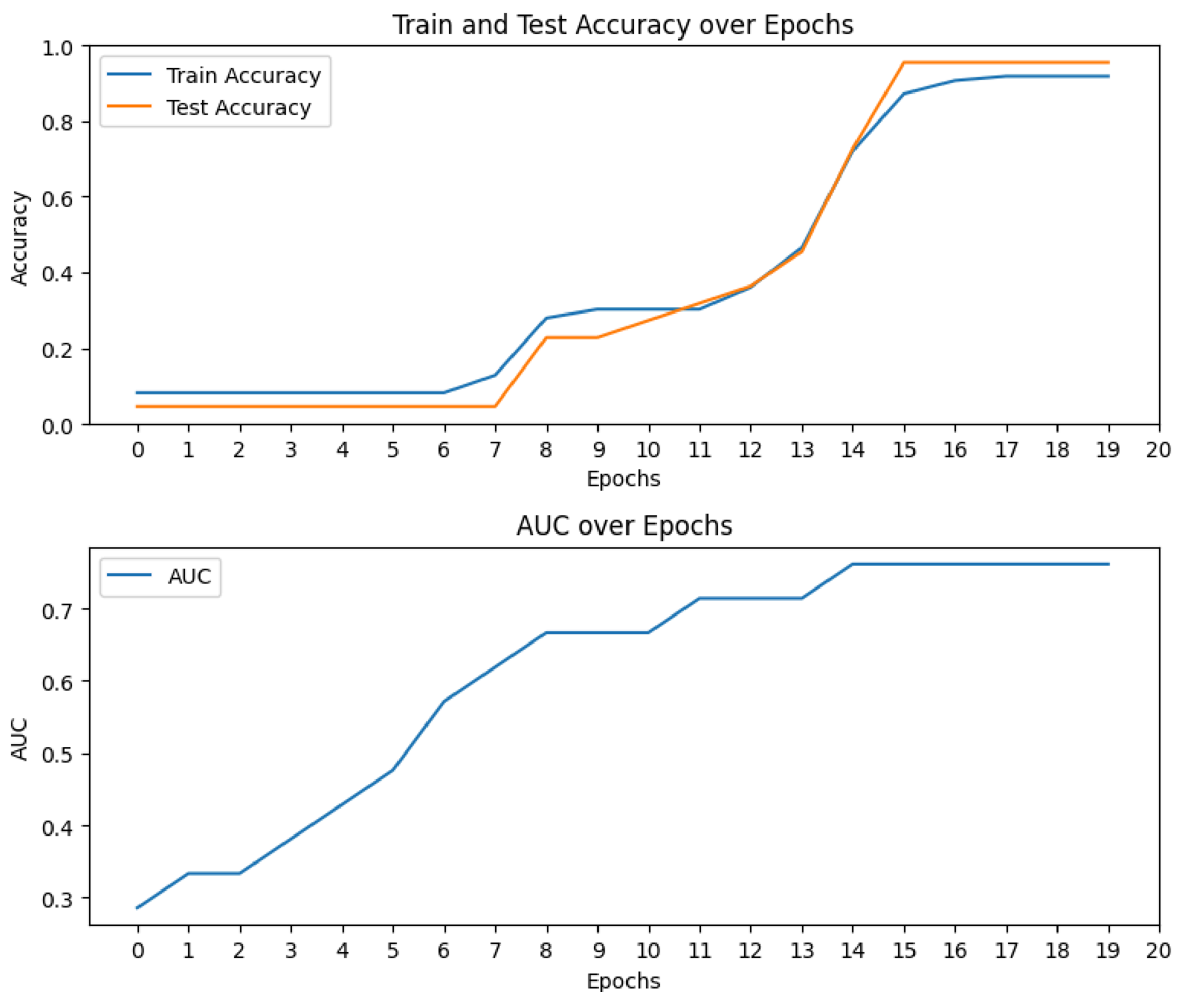


Figura 8. Dades de les primeres 20 iteracions del MLP per pràctiques amb distància

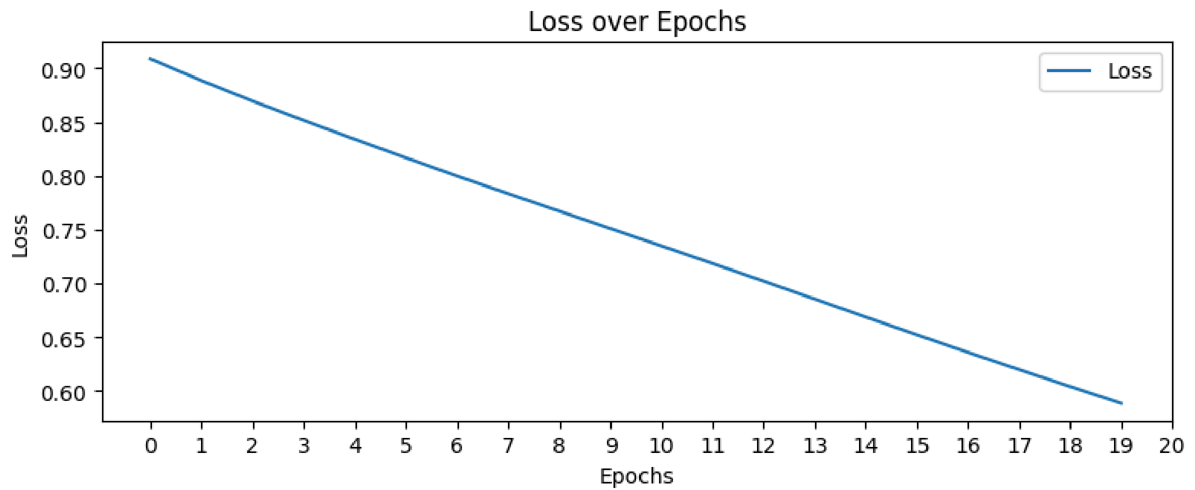


Figura 9. Dades de les primeres 20 iteracions del MLP per pràctiques amb distància

Es nota un increment en el valor de la **AUC** fins a 0.7, però també en la **Loss**, que no baixa de 0.65 en les primeres 20 iteracions, mentre que l'**Accuracy** no arriba al valor de 0.95 fins l'iteració numero 15.

5.2.2 Proves amb el Dataset de pràctiques amb CodeBERT

Un cop obtingudes les dades inicials, explicades a l'apartat anteriors, i com que encara no es tenien els resultats de l'examen parcial amb les categories adients assignades, es va decidir repetir les proves de [apartat 5.2.1] però fent servir el model preentrenat de CodeBERT enlloc de CodeT5 amb l'objectiu de veure la diferència entre aquests.

Predicció de qualificació

Un cop processats els embeddings de les 108 tasques i generades les seves qualificacions, s'obté el diagrama de dispersió següent respecte a les notes del corrector humà:

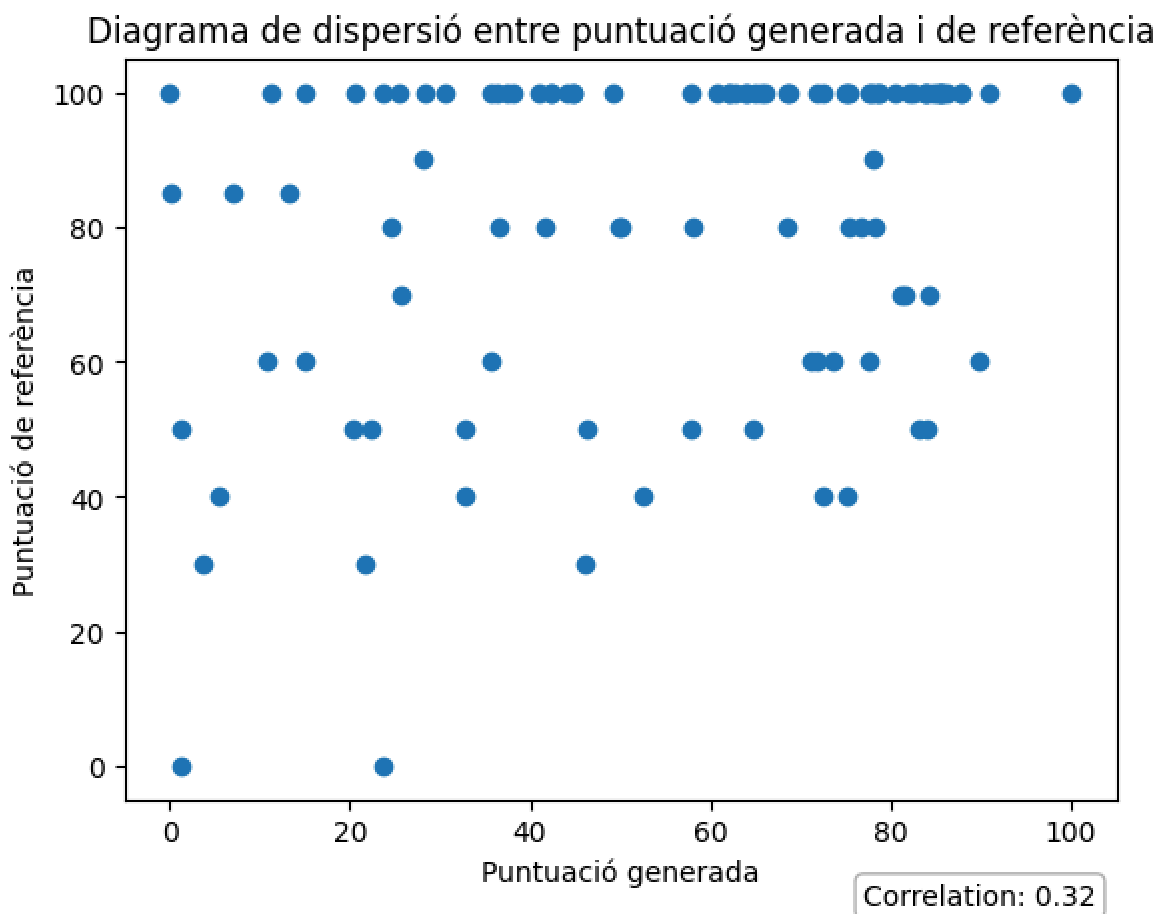


Figura 10. Diagrama de dispersió per qualificacions generades de pràctiques amb CodeBERT

Es pot veure que els resultats són bastant similars als obtinguts amb el model de CodeT5, amb una correlació quasi idèntica de 0.32.

Predicció de resultats d'asserts

S'entrena el **MLPClassifier** amb les mateixes proporcions de training y test datasets.

Inicialment s'obté una **Accuracy** de 0.95 i una **Loss** de 0.42, però al avaluar les dades de les primeres 20 iteracions d'entrenament s'obtenen els següents resultats:

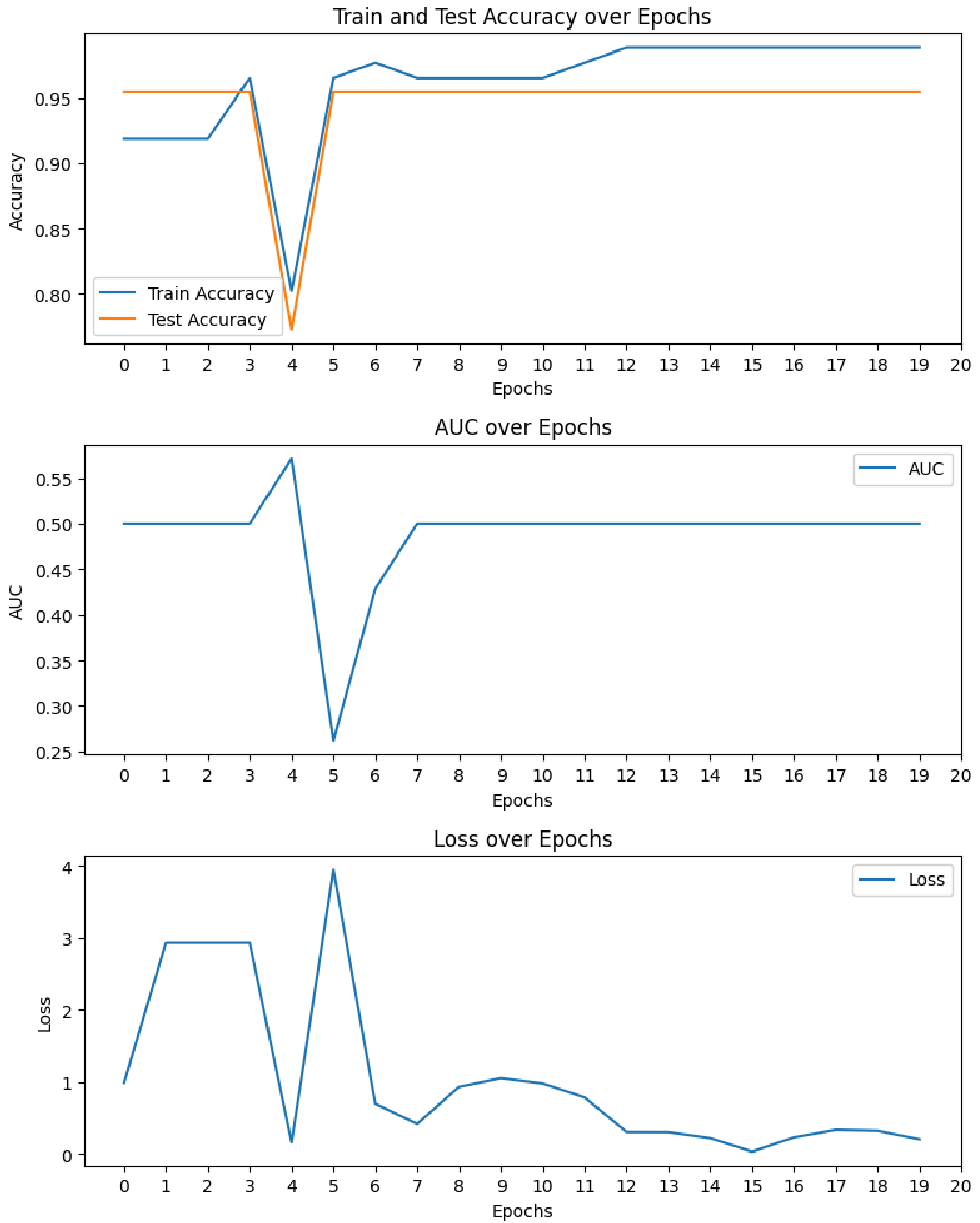


Figura 11. Dades de les primeres 20 iteracions del MLP per pràctiques amb CodeBERT

Els resultats d'aquest MLP són semblants als obtinguts amb el model previ, excepte la Loss, que és molt més alta. La **AUC** segueix tenint un valor al voltant del 0.5, que indica que aquest model no funciona millor que generar una categoria aleatòriament.

Seguint el mateix procés que pel model anterior, s'ha entrenat un segon MLP que rep com a **feature vector** la resta entre l'embedding de la tasca i la solució de referència. Aquests són els resultats:

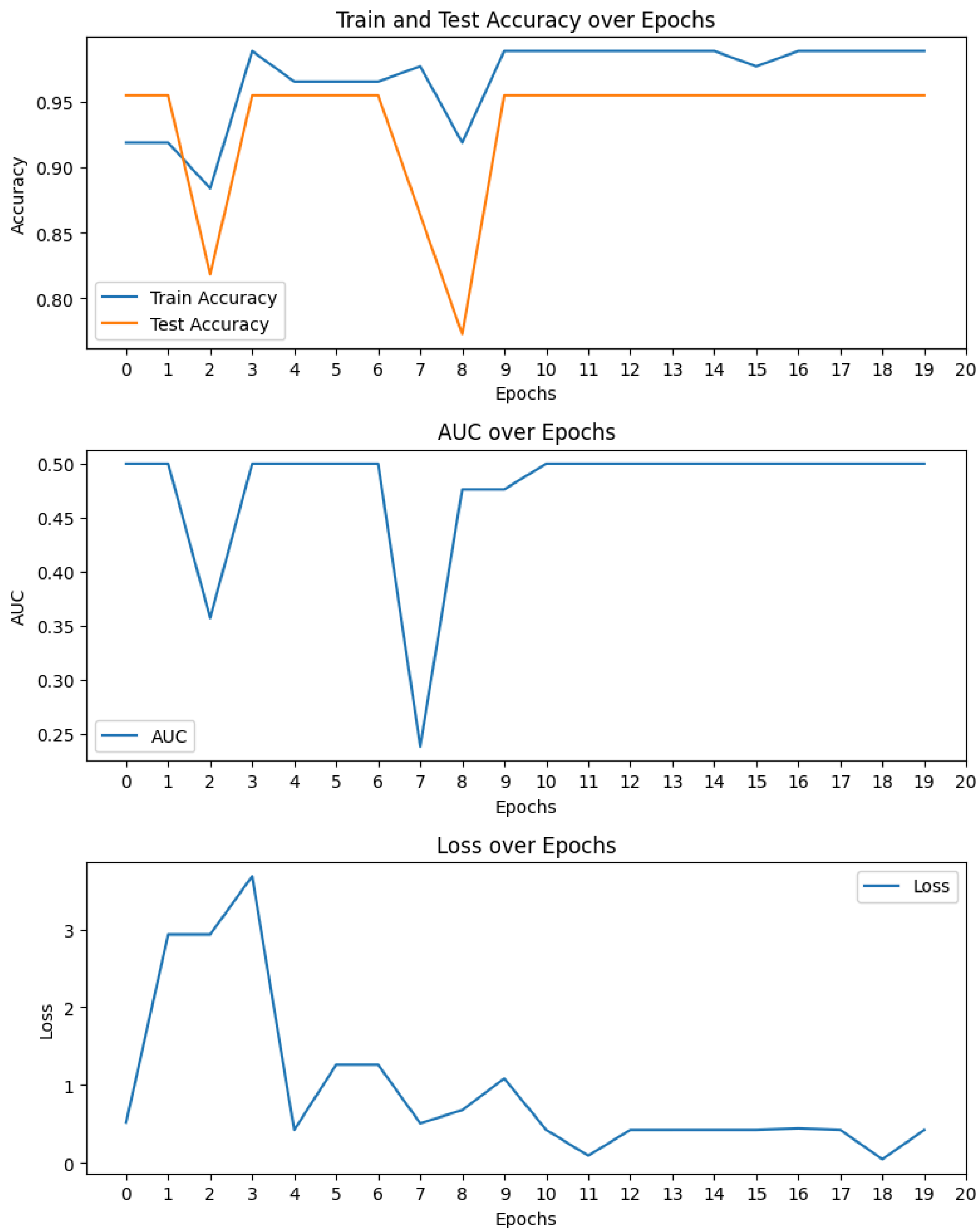


Figura 12. Dades de les primeres 20 iteracions del MLP per pràctiques amb distància i codeBERT

A diferència del model anterior, en aquest 2n experiment no s'observa una millora apreciable de l'**AUC**, mentre que l'accuracy es manté similar, igual que la Loss.

Després d'obtenir aquests resultats, s'ha decidit continuar fent servir **CodeT5** per a tots futurs experiments.

5.2.3 Proves amb el subconjunt d'exercici Capicua Rotacions

Després dels exàmens parcials del semestre, s'han pogut obtenir dades de l'examen d'Algorismica, corregides tenint en compte aquest projecte i assignant les categories referenciades anteriorment a cada exercici, a més a més de la qualificació sobre 10.

En total s'han corregit **140** examens. De cada examen, 2 exercicis han sigut categoritzats, amb el que tenim un total de **280** mostres per entrenar models de MLP. Primer hem començat tractant només l'exercici 1, i aplicant el mateix procés que a les pràctiques dels apartats anteriors.

Predicció de qualificació

Un cop processats els embeddings dels **140** exercicis i generades les seves qualificacions, s'obté el diagrama de dispersió següent respecte a les notes del corrector humà:

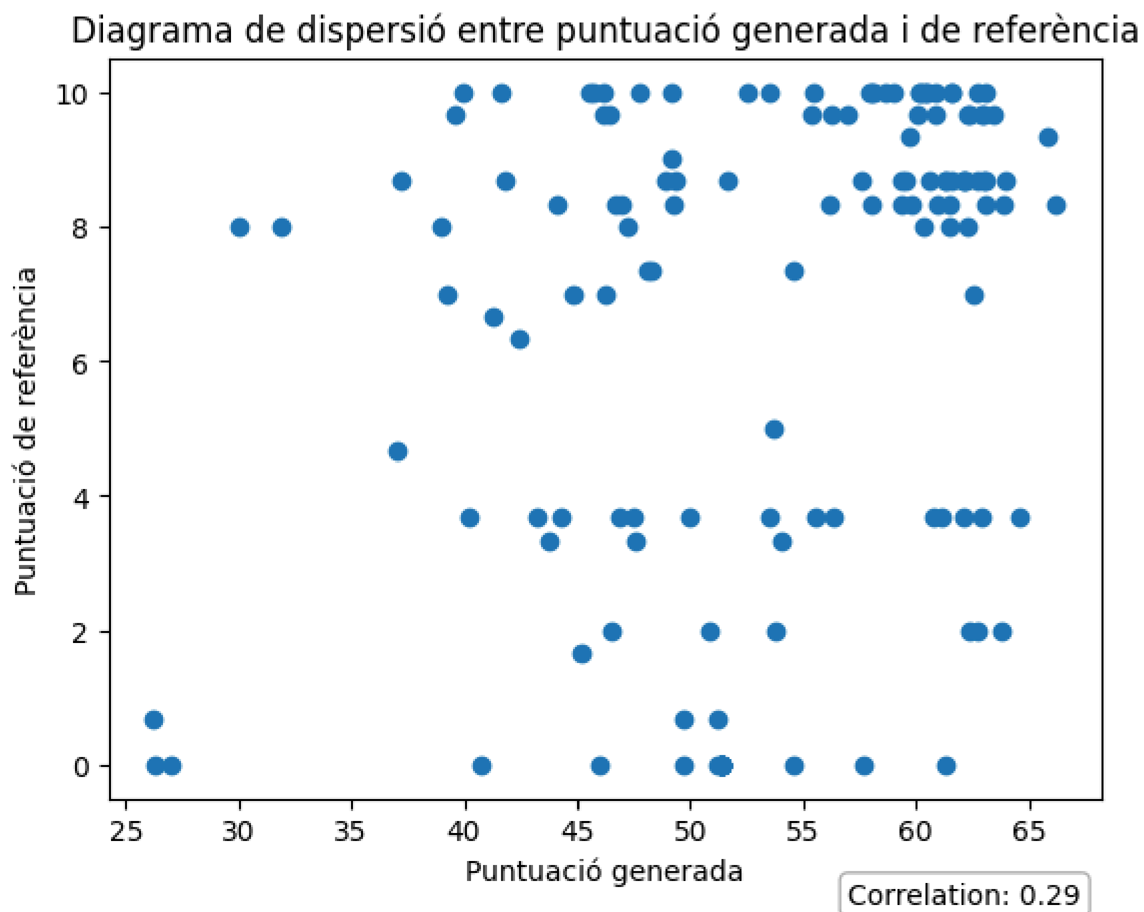


Figura 13. Diagrama de dispersió de qualificacions genrades d'Exercici 1 Parcial

Podem veure també el valor del coeficient de correlació de Pearson entre les qualificacions generades i les de referència, que és 0.29, una correlació moderadament positiva, similar a les correlacions obtingudes a les proves anteriors amb diferents tasques.

Es pot veure que en aquest exercici en particular, les puntuacions generades pel codi no passen del 6.5/10, mentre que les del corrector humà segueixen, generalment estant per sobre del 7.

Predicció de la etiqueta de classe
Després d'obtenir les qualificacions, es procedeix a entrenar el clasificador MLP que, a diferència de les proves anteriors, assigna 1 de les 5 categories de l'article de Wang, ja que per aquest **dataset** de mostres, si que tenim categories amb les quals entrenar el classificador.

Per això dividim les mostres en datasets de training i de test, amb una proporció de 0.8 i 0.2 respectivament i entrenem el **MLPClassifier**.

Inicialment s'obté una **Accuracy** de 0.375 i una **Loss** de 0.014, amb la següent matriu de confusió:

	C2	**C3**	**C4**	**C5**
C2	1	3	0	0
C3	0	2	0	2
C4	2	2	3	2
C5	1	0	3	3

Es pot veure que el classificador ha predit correctament:
- 1 instància de C2
- 2 instàncies de C3
- 3 instàncies de C4
- 3 instàncies de C5

S'observa que la dificultat principal la té a l'hora de predir C4.

Al avaluar les dades de les primeres 20 iteracions d’entrenament s’obtenen els següents resultats:

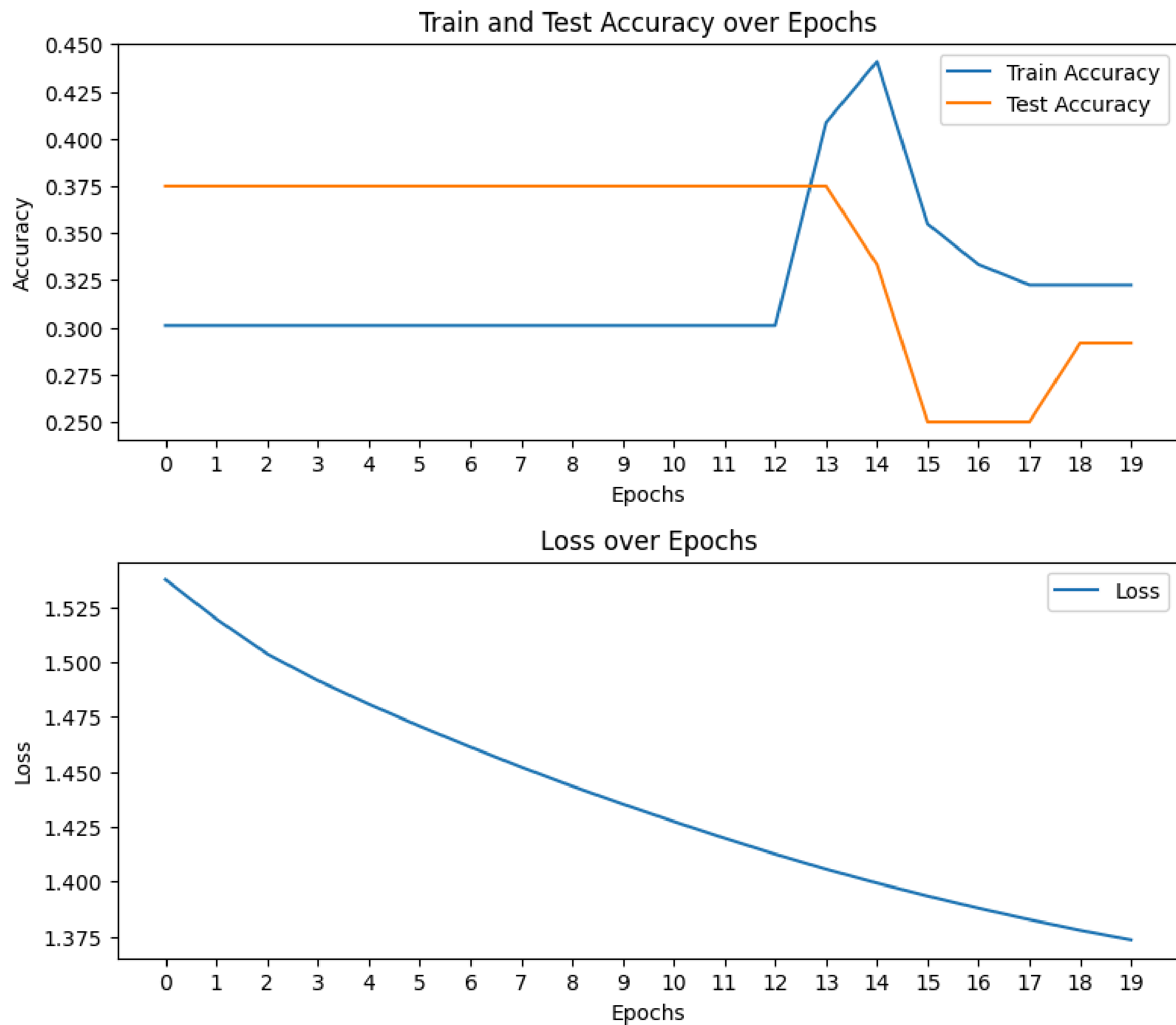


Figura 14. Dades de les primeres 20 iteracions de MLP de Exercici 1 Parcial

Es pot veure una accuracy de 0.3 i una loss de 1.38 després de 20 iteracions, però al contrari que fent servir les dades de les entregues de pràctiques dels anys anteriors, no hi ha signes de sobreentrenament.

5.2.4 Proves amb el subconjunt d'exercici RLE

Es repeteix el mateix procés amb l'exercici 3 de l'exàmen.

Predicció de qualificació

Un cop processats els embeddings de les **140** tasques i generades les seves qualificacions, s'obté el diagrama de dispersió següent respecte a les notes del corrector humà:

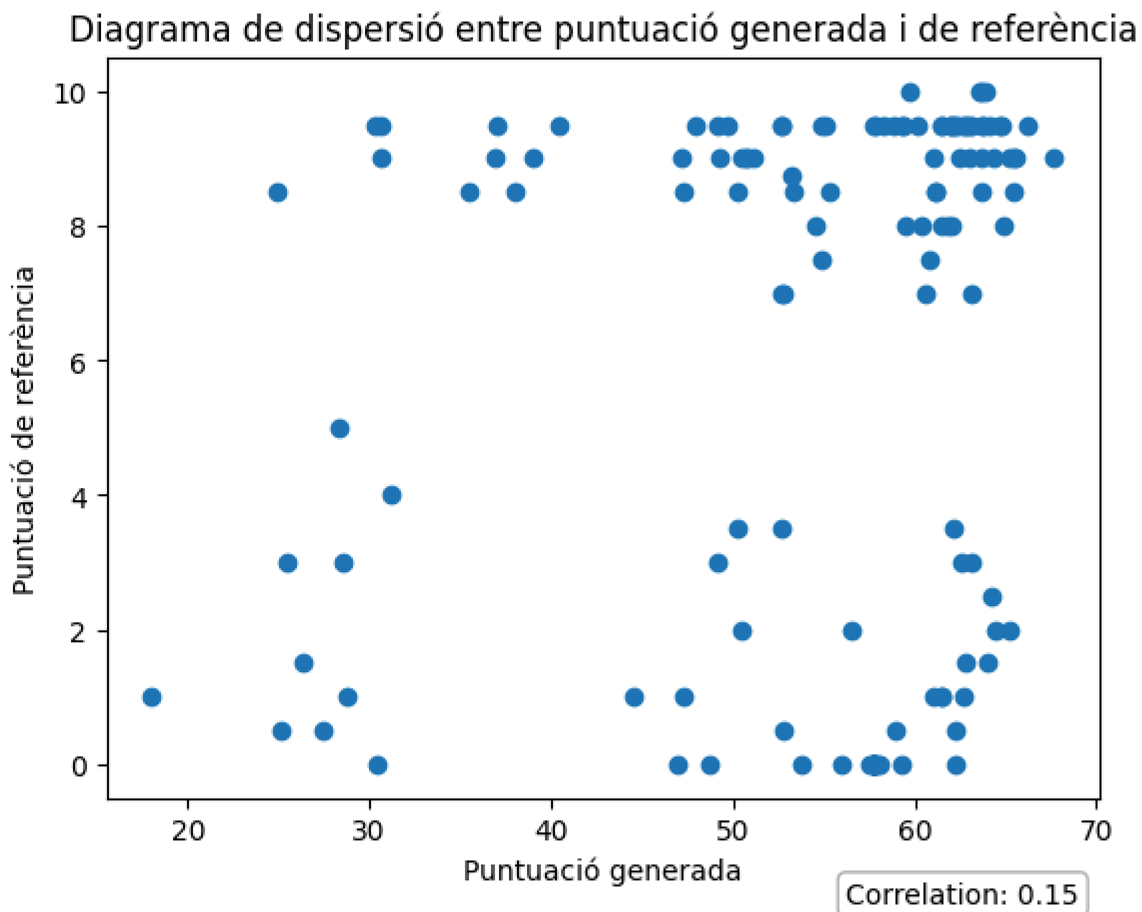


Figura 15. Diagrama de dispersió de qualificacions generades d'Exercici 3 Parcial

S'observa que el coeficient de correlació per aquest exercici és de 0.15, molt inferior al obtingut a l'exercici 1.

Es pot veure que en aquest exercici en particular, les puntuacions generades pel codi no passen del 7/10, mentre que les del corrector humà segueixen, generalment estant per sobre del 7.

Predicció de la etiqueta de classe
Després d'obtenir les qualificacions, es procedeix a entrenar el clasificador MLP que, a diferència de les proves anteriors, assigna 1 de les 5 categories de l'article de Wang, ja que per aquest **dataset** de mostres, si que tenim categories amb les quals entrenar el classificador.

Per això dividim les mostres en datasets de training i de test, amb una proporció de 0.8 i 0.2 respectivament i entrenem el **MLPClassifier**.

Inicialment s'obté una **Accuracy** de 0.173 i una **Loss** de 0.012, amb la següent matriu de confusió:

	C1	**C2**	**C3**	**C4**	**C5**
C1	0	0	0	2	0
C2	0	0	0	0	2
C3	0	1	0	1	1
C4	0	2	2	2	3
C5	0	2	0	3	2

Es pot veure que el classificador ha predit correctament:
- 2 instàncies de C4
- 2 instàncies de C5

S'observa molta més dificultat a l'hora de predir categories per aquest problema amb respecte a l'exercici 1.

Al avaluar les dades de les primeres 20 iteracions d'entrenament s'obtenen els següents resultats:

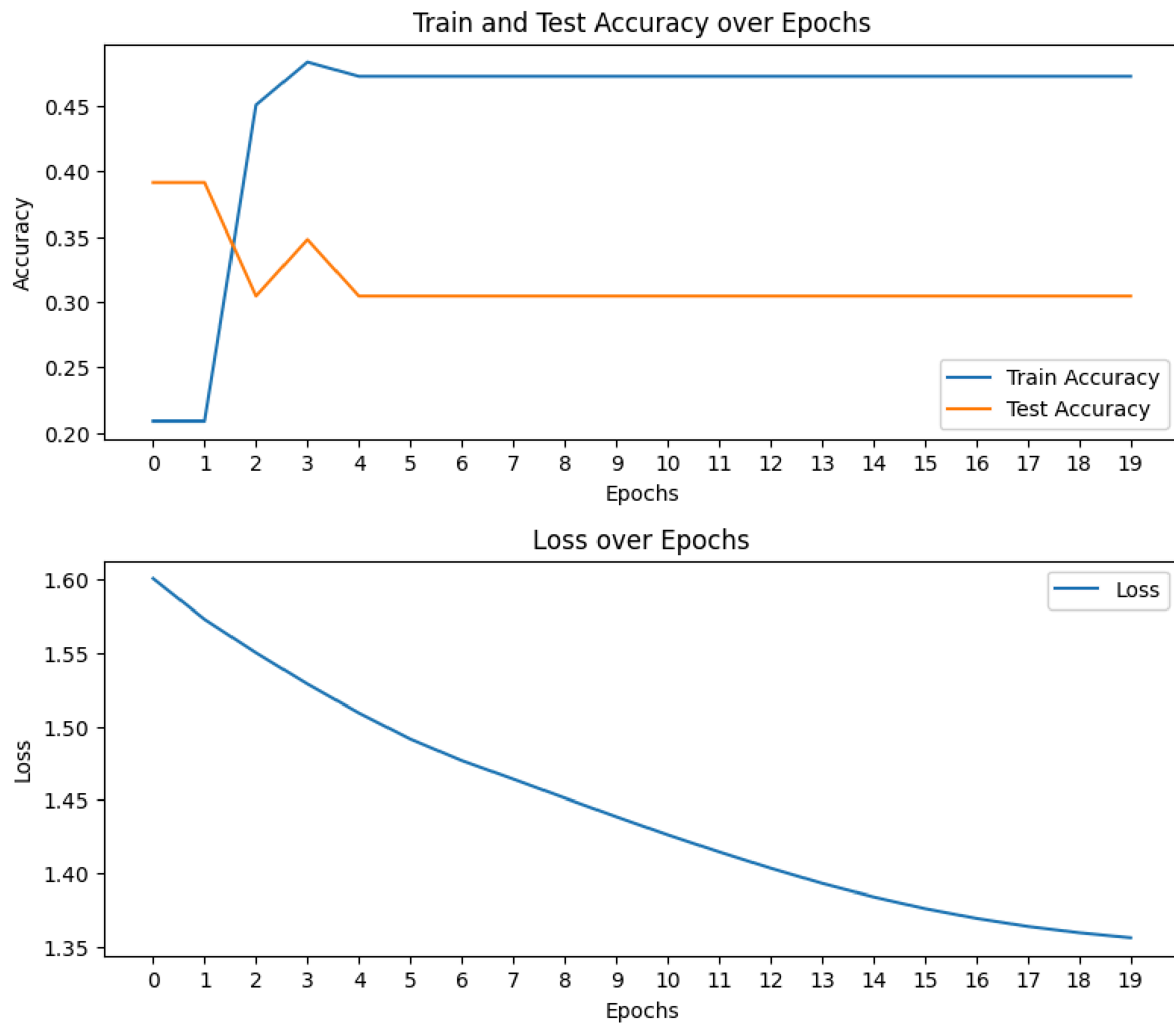


Figura 16. Dades de les 20 primeres iteracions del MLP Exercici 3 Parcial.

Es veu que inicialment en les iteracions, la Accuracy és de 0.3, però al seguir iterant baixa fins a 0.17.

5.2.5 Proves amb els Datasets combinats

Com a prova final, s'ha intentat combinar els Datasets dels exercicis 1 i 3, per veure si el problema podria ser generalitzat.

Predicció de qualificació

Un cop processats els embeddings de les **140** tasques i generades les seves qualificacions, s'obté el diagrama de dispersió següent respecte a les notes del corrector humà:

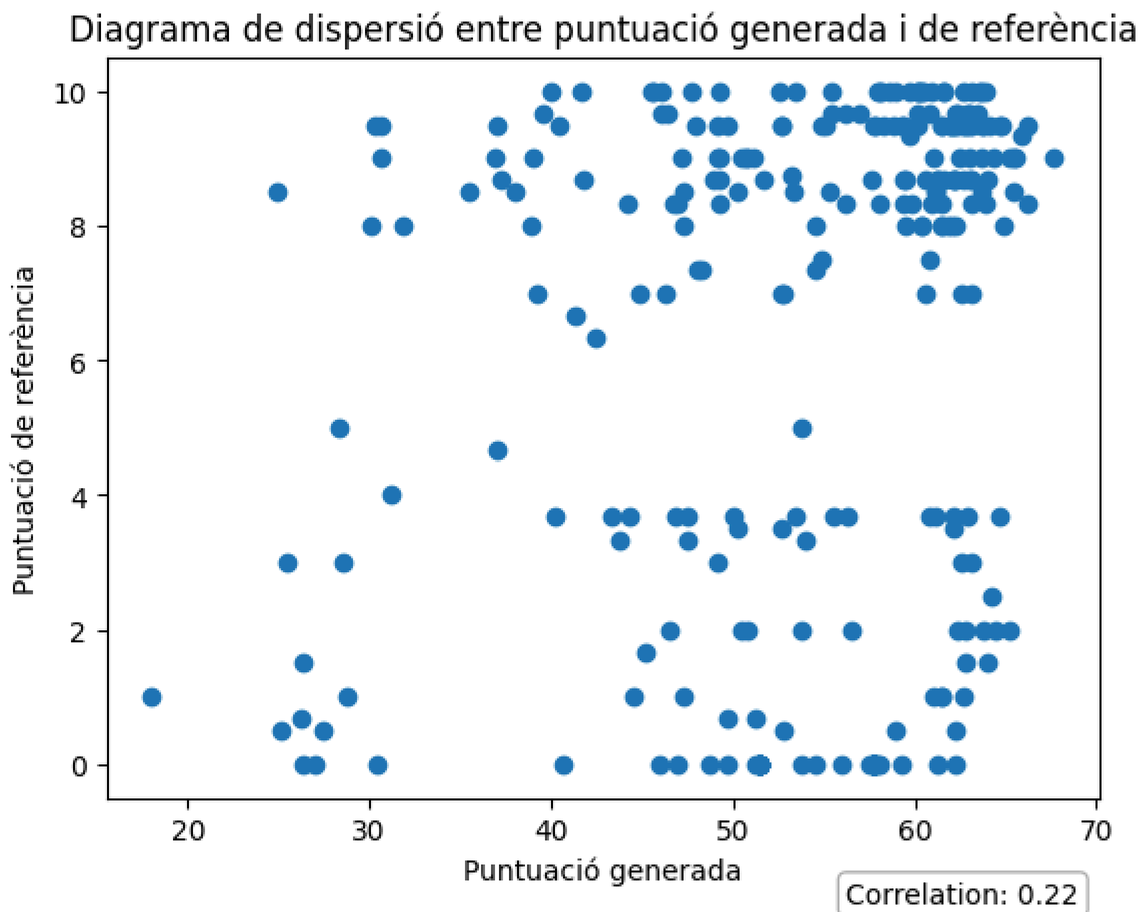


Figura 17. Diagrama de dispersió de qualificacions generades a partir de dades combinades Parcial

S'observa que el coeficient de correlació per aquesta prova és de 0.22, entre mig dels exercicis individuals.

Es pot veure que en aquest exercici en particular, les puntuacions generades pel codi no passen del 7/10, mentre que les del corrector humà segueixen, generalment estant per sobre del 7.

Predicció de la etiqueta de classe
Després d'obtenir les qualificacions, es procedeix a entrenar el clasificador MLP que, a diferència de les proves anteriors, assigna 1 de les 5 categories de l'article de Wang, ja que per aquest **dataset** de mostres, si que tenim categories amb les quals entrenar el classificador.

Per això dividim les mostres en datasets de training i de test, amb una proporció de 0.8 i 0.2 respectivament i entrenem el **MLPClassifier**.

Inicialment s'obté una **Accuracy** de 0.255 i una **Loss** de 0.02 amb la següent matriu de confusió:

	C1	**C2**	**C3**	**C4**	**C5**
C1	0	0	1	2	0
C2	2	0	2	1	1
C3	0	3	1	1	5
C4	1	6	2	5	6
C5	0	1	0	1	6

Es pot veure que el classificador ha predit correctament:

- 1 instància de C3
- 5 instàncies de C4
- 6 instàncies de C5

Com al classificador de l'exercici 1, s'observa gran dificultat a l'hora de distinguir la Categoria 4, ja que és la categoria que més assigna indiscriminadament.

Al avaluar les dades de les primeres 50 iteracions d’entrenament s’obtenen els següents resultats:

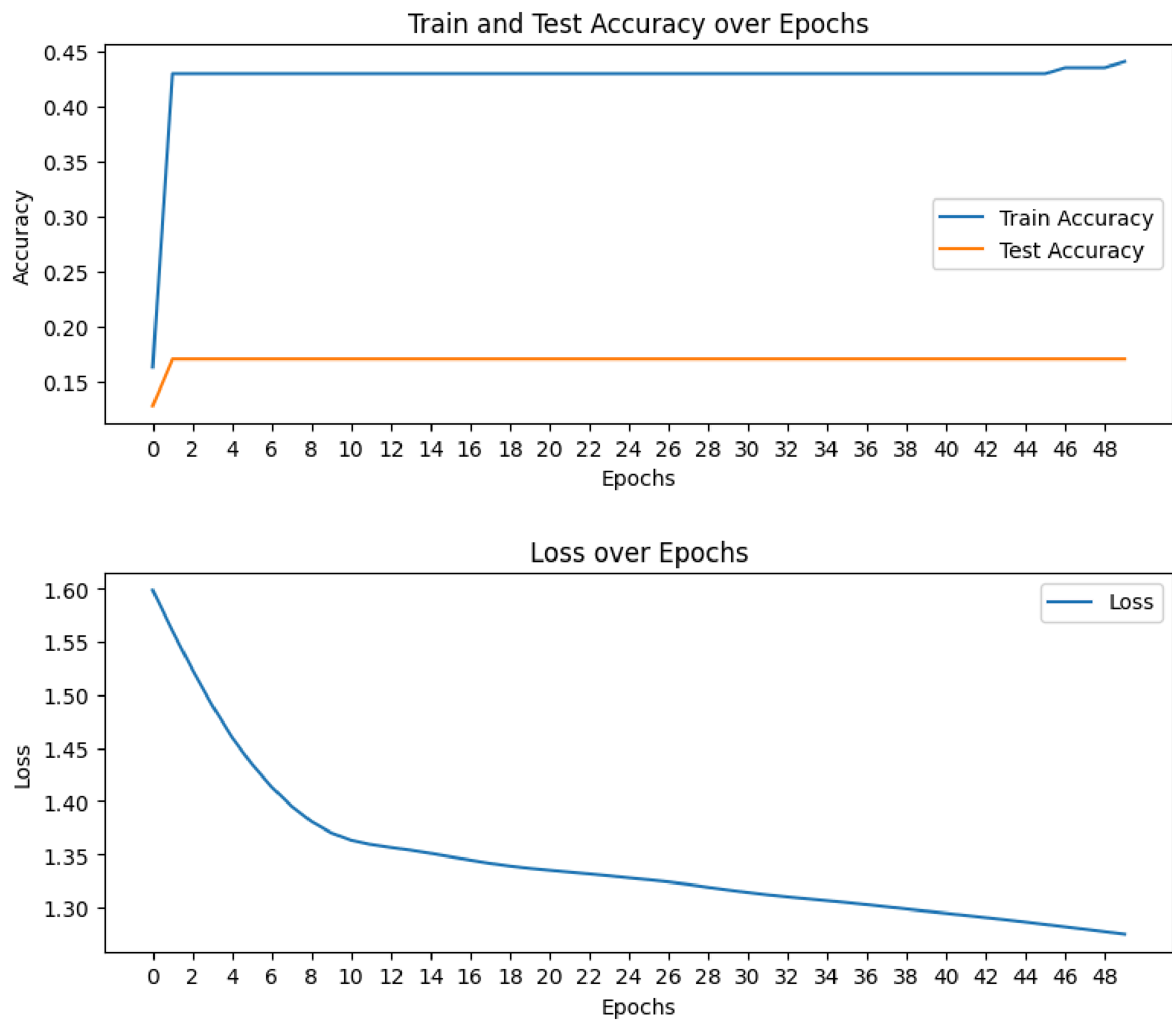


Figura 18. Dades de 20 primeres iteracions del MLP a partir de dades combinades Parcial

Es veu que inicialment l’Accuracy es manté al voltant de 0.17, però a mesura que es segueix iterant més endavant puja fins a 0.255.

6 Conclusions i feina futura

6.1 Conclusions

La conclusió principal d'aquest projecte és que mitjançant els embeddings dels models preentrenats i xarxes neuronals únicament, no es pot qualificar un codi correctament amb seguretat. Això indica que la hipòtesi plantejada on es comentava que la distància entre els embeddings de codi entregat per alumnes i la solució plantejada pel professor seria suficient per determinar una qualificació, no és del tot certa. Encara que hi ha un cert grau de similitud moderada, no es pot fer servir amb seguretat per avaluar codi. S'ha observat com la majoria de qualificacions atorgades pel corrector humà eren molt més extremes que les generades a partir d'embeddings. També s'ha vist dificultat a l'hora de distingir codis que poden ser similars quant a estructura, però un pot funcionar correctament i l'altre pot tenir un error de sintaxi que no el diferència gaire respecte al codi correcte, però provoca un error d'execució, requerint una qualificació de suspens que no és generada per l'autocorrector.

Quant a l'entrenament dels classificadors i xarxes neuronals, s'ha vist una accuracy molt elevada en els primers experiments a l'hora de predir els resultats d'asserts de tasques d'alumnes. Aquests resultats, però, són en gran part deguts a que el 93% de les 108 mostres utilitzades per entrenar el classificador (dataset de pràctiques d'anys anteriors) passen els asserts, amb el que el classificador aprèn a assignar sempre "True" com a classe i aconseguir una accuracy del 95%. El principal problema d'aquests experiments han sigut la quantitat i varietat de mostres.

Un cop obtingudes les mostres de l'examen parcial d'algorísmica, s'han entrenat els classificadors que inicialment es volia, amb les 5 categories, i s'ha vist una accuracy molt inferior, amb un rang del 0.375 fins al 0.17. Es creu que hi ha dues teories per les quals s'obtenen resultats tant poc favorables:

La primera hipòtesi segueix sent la manca d'una gran quantitat de mostres, pels classificadors específics d'exercicis només hi havia 140 mostres i pel classificador generalitzat n'hi havia la combinació d'aquestes amb un total de 280 mostres. Considerant que el **feature vector** de cada mostra té una longitud de 256 elements. Això pot provocar **Overfitting**, fent que el model memoritzi les dades d'entrenament en lloc d'aprendre els patrons.

L'altra teoria és que a aquests models preentrenats els hi falta un ingredient, que és el de comparació. Al model de Wang, se l'ensenya a comparar entre embeddings i aquests models no ho tenen generalitzat. No hi ha cap "fine tuning" o procés addicional d'adaptació del model que permeti que els embeddings siguin similars quan representin codi similar.

6.2 Feina futura

La feina futura podria tenir diferents camins de desenvolupament. La principal seria l'obtenció de més mostres d'entrenament, com les obtingudes de l'examen parcial d'algorísmica, tant amb qualificacions del 0 al 10 com classificades en les 5 categories com a referència per així millorar la precisió de les xarxes neuronals.

L'altre camí seria fer el desenvolupament d'aquest procés adicional d'adaptació del model que permeti la comparació entre embeddings amb més facilitat, mitjançant el procés de "fine-tuning".

7 Bibliografia i Referències

1. Automated Code Review:
   Wikipedia - Automated Code Review

2. Static Program Analysis:
   Wikipedia - Static Program Analysis

3. Dongxia Wang, En Zhang, and Xuesong Lu. Automatic Grading of Student Code with Similarity Measurement. East China Normal University, Shanghai, China:
   https://2022.ecmlpkdd.org/wp-content/uploads/2022/09/sub_828.pdf

4. Peter C Issacson and Terry A Scott. Automating the execution of student programs. ACM SIGCSE Bulletin, 1989:
   https://dl.acm.org/doi/10.1145/65738.65741

5. Abstract Syntax Tree:
   Wikipedia - Abstract Syntax Tree

6. Word Embedding:
   Wikipedia - Word embedding

7. Word Embedding Huggingface:
   https://huggingface.co/blog/getting-started-with-embeddings

8. OpenAI - Introducing text and code embeddings blog:
   https://openai.com/blog/introducing-text-and-code-embeddings

9. Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D.Q. Bui, Junnan Li, Steven C.H. Hoi. CodeT5+: Open Code Larga Language Model for Code Understanding and Generation, 2023:
   https://arxiv.org/pdf/2305.07922.pdf

10. Euclidean Distance:
    Wikipedia - Euclidean Distance

11. Cosine Similarity:
    Wikipedia - Cosine Similarity

12. Manhattan Distance:
    Wikipedia - Taxicab geometry

13. Pearson correlation:
    Wikipedia - Pearson correlation coefficient

14. Neural network:
    Wikipedia - Neural network

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

16. unicodedata - Unicode Database: https://docs.python.org/3/library/unicodedata.html

17. transformers: https://huggingface.co/docs/transformers/index

18. numpy: https://numpy.org/doc/stable/

19. scipy.spatial - Spatial algorithms and data structures: https://docs.scipy.org/doc/scipy/reference/spatial.html

20. pandas: https://pandas.pydata.org/docs/

21. Torch: https://pytorch.org/docs/stable/index.html

22. sklearn.neural_network: https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html

23. sklearn.model_selection: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html

24. sklearn.metrics: https://scikit-learn.org/stable/modules/model_evaluation.html

25. threading - Thread-based Parallelism: https://docs.python.org/3/library/threading.html

26. matplotlib.pyplot: https://matplotlib.org/3.5.3/api/_as_gen/matplotlib.pyplot.html

27. HuggingFace - CodeT5p: https://huggingface.co/Salesforce/codet5p-110m-embedding