Universidad de La Habana

Facultad de Matemática y Computación

Predicción de Problemas de Codeforces

Juan Carlos Espinosa Delgado C-411
Raudel Alejandro Gómez Molina C-411
Alex Sierra Alcalá C-411
Yoan René Ramos Corrales C-412

Proyecto en github

1. Introducción

1.1. Motivación

La plataforma Codeforces es una herramienta fundamental en la comunidad de programación competitiva, diseñada para desarrollar y entrenar habilidades de resolución de problemas. Los problemas en Codeforces no solo desafían a los competidores, sino que también proporcionan una base sólida para aprender y aplicar diversos algoritmos y estructuras de datos. Cada problema está asociado a una serie de categorías o etiquetas que ayudan a identificar los tipos de algoritmos y técnicas necesarias para resolverlos. Pero este proceso suele estar muy apegado a la a la solución final del ejercicio por lo que la correcta identificación de los tags suele ser un gran reto para los competidores.

Es válido aclarar que la automatización de este proceso no es interés de la programación competitiva, ya que en estos escenarios lo que se busca es el razonamiento lógico de los competidores, en este contexto si pudiera ser interesante el desarrollo de un mecanismo de generación de problemas, pero este no es el objetivo de este proyecto.

Sin embargo la experimentación con la detección de tags en escenarios controlados como este, manteniéndonos en el entorno de que esta tarea es útil para la posterior solución del problema pudiera servir de base para proponer mecanismo de detección de tags en problemas reales orientados al campo de la generación de algoritmos.

1.2. Problemática

Cada problema en la plataforma Codeforces y un problema de programación competitiva en general cuenta primeramente con un título y una descripción en la cual se plantea el objetivo y la problemática del mismo, este segundo componente del problema es la principal fuente de interés para la detección de tags ya que estos se encuentran explícitos en forma de lenguaje natural en dicha descripción.

Pero además de la descripción los problemas cuentan con una restricción de tiempo y espacio de memoria donde se deben desenvolver los algoritmos que den solución al problema (un algoritmo correcto para todos los casos de prueba que no este dentro de los límites establecidos no es considerado como solución). Ahora bien esta restricción también influye en los tags del algoritmo ya el conjunto de tags del problema que satisfacen estas restricciones y la descripción del problema será subconjunto del conjunto de tags solo asociados a la descripción.

Otra característica interesante que nos pudiera aportar información sobre la naturaleza del problema es analizar el código de una solución aceptada del problema, ya que es posible identificar los tags asociados a dicho código y por tanto un subconjunto de los tags del problema. Es importante analizar que con este enfoque solo podemos obtener un subconjunto de los tags ya que un mismo problema puede tener multiples soluciones y cada solución puede tener asociada distintos tipos de tags.

Por tanto como tenemos de por medio un problema asociado a lenguaje natural y actualmente no hay ningún método asociado medianamente eficaz asociado a este campo que no lleve Machine Learning, la propuesta de la solución descrita en este trabajo empleará algoritmos de Machine Learning los cuales iremos introduciendo a lo largo de este trabajo.

1.3. Objetivos Generales y Específicos

1.3.1. Objetivos Generales

El objetivo general de este proyecto es desarrollar un sistema automatizado utilizando técnicas de Machine Learning para detectar y asignar etiquetas (tags) a los problemas de programación en Codeforces.

1.3.2. Objetivos Específicos

- **Recolección y Preprocesamiento de Datos:**
  - Recolectar un conjunto representativo de problemas de Codeforces, incluyendo sus descripciones, restricciones de tiempo y memoria, y etiquetas existentes.

- Realizar el preprocesamiento del texto de las descripciones para normalizar y limpiar los datos, facilitando su análisis.

- **Desarrollo del Modelo de Machine Learning:**
  - Investigar y seleccionar algoritmos de Machine Learning apropiados para la tarea de clasificación de texto, como KNN, Naive Bayes, y redes neuronales.
  - Entrenar varios modelos utilizando el conjunto de datos preprocesado, ajustando hiperparámetros para optimizar el rendimiento.
- **Evaluación del Modelo:**
  - Evaluar los modelos entrenados utilizando métricas de rendimiento como precisión, recall, F1-score y exactitud.
  - Comparar los resultados de diferentes modelos para identificar el más eficaz en la detección de etiquetas.
  - Comparar los resultados con un modelo de lenguaje (en este caso se usará Chat-GPT 3.5).
- **Implementación y Prueba:**
  - Implementar el modelo seleccionado en un entorno de prueba para evaluar su rendimiento en condiciones reales.
  - Realizar pruebas adicionales para validar la consistencia y robustez del sistema en la asignación de etiquetas.
- **Documentación y Propuesta de Mejora:**
  - Documentar el proceso completo de desarrollo, incluyendo la recolección de datos, preprocesamiento, desarrollo del modelo, evaluación e implementación.
  - Proponer mejoras y futuras líneas de investigación basadas en los resultados obtenidos y las limitaciones encontradas durante el desarrollo del proyecto.

1.3.3. Hipótesis

- **Hipótesis Principal:** Un modelo de Machine Learning bien entrenado puede detectar y asignar etiquetas (tags) a los problemas de programación de Codeforces con una precisión comparable a la de un humano experto.
- **Hipótesis Secundarias:**
  - Los algoritmos de clasificación de texto basados en redes neuronales (como LSTM o Transformers) ofrecerán un mejor rendimiento en la detección de etiquetas en comparación con algoritmos más tradicionales como Naive Bayes o KNN.
  - El análisis y utilización de las restricciones de tiempo y memoria, así como del código de soluciones aceptadas, pueden mejorar significativamente la precisión del modelo en la asignación de etiquetas.

1.3.4. Preguntas Científicas

- ¿Qué algoritmos de Machine Learning son más efectivos para la tarea de detección de etiquetas en problemas de programación competitiva?
- ¿Cómo influye la calidad y cantidad de los datos de entrenamiento en el rendimiento del modelo?
- ¿En qué medida las restricciones de tiempo y memoria impactan en la precisión de la detección de etiquetas?
- ¿Es posible mejorar la detección de etiquetas utilizando información adicional del código de soluciones aceptadas?
- ¿Cuáles son las principales limitaciones y desafíos al aplicar Machine Learning en la detección de etiquetas en problemas de programación competitiva?

2. Análisis de los datos

Como dataset usaremos un conjunto de problemas de Codeforces con su identificador, descripción, conjunto de tags, puntos y rating.

2.1. Exploración de datos

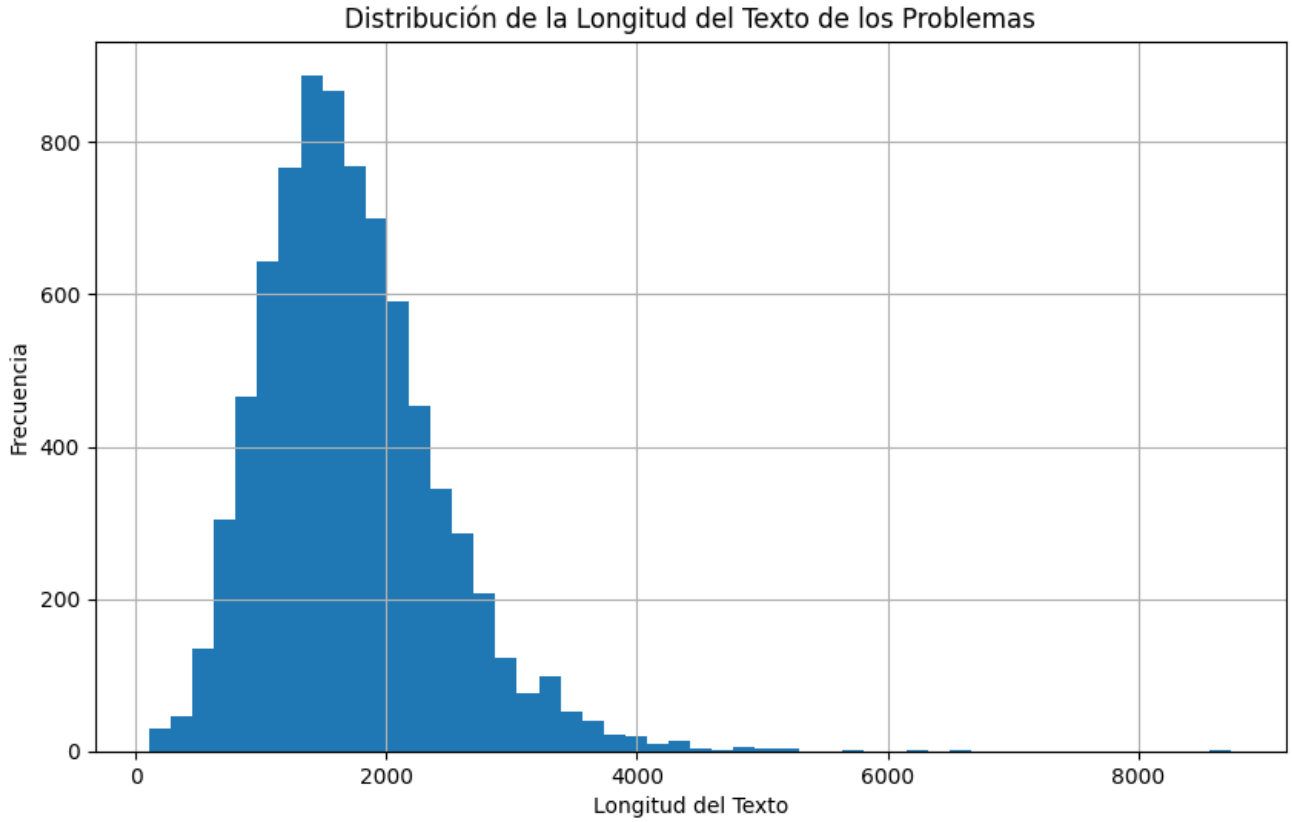
Para esto primero exploramos los datos buscando filas con ausencia de datos y eliminamos datos que no son de interés para nuestro problema.

En este caso hemos identificado que las columnas asociadas a los puntos y el rating cuentan con valores neuronales y no representan interés para nuestro problema por lo que hemos decidido eliminarlas.

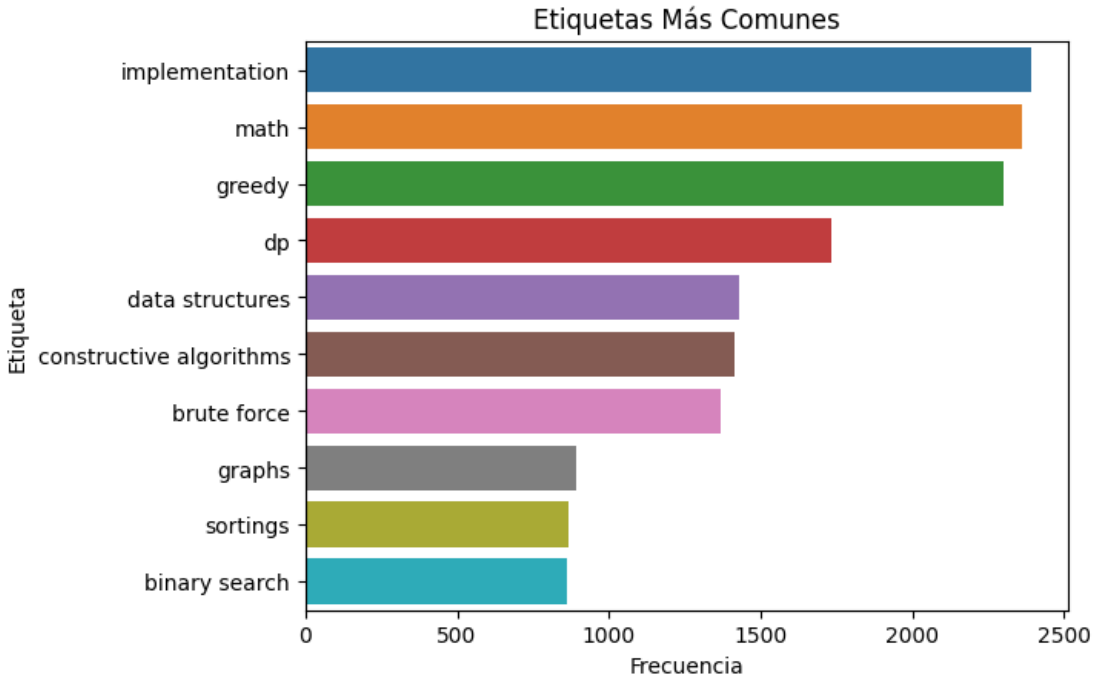
Luego analizamos el contenido de la descripción de los problemas y identificamos el idioma de las mismas ya que esta información puede ser útil para el futuro análisis empleando modelos de lenguaje.

2.1.1. Información recopilada sobre los datos

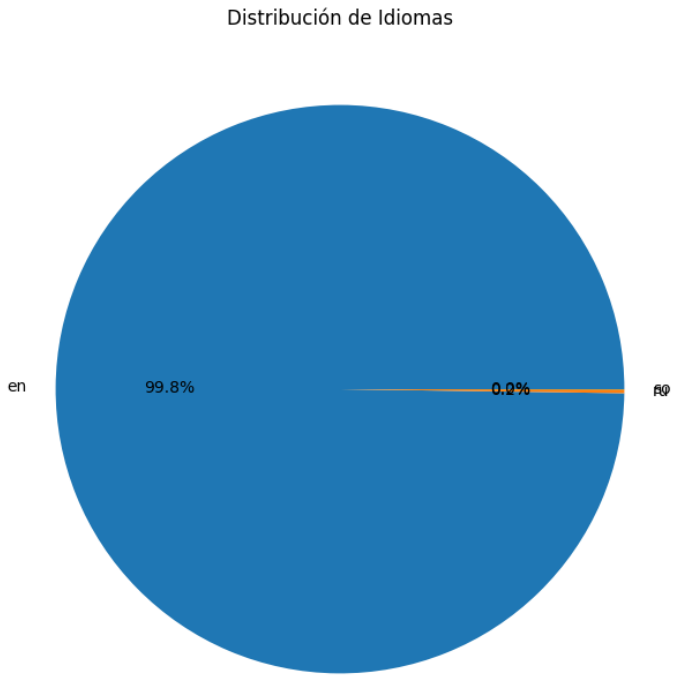
- Distribución de la longitud del texto de los Problemas



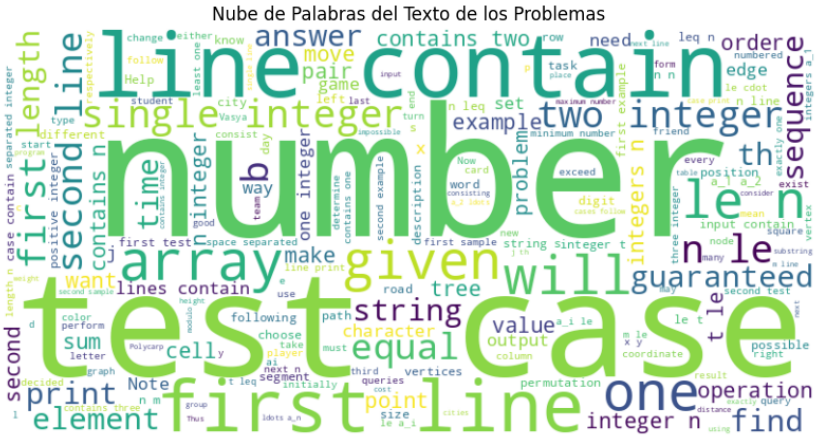
- Frecuencia de las etiquetas más comunes



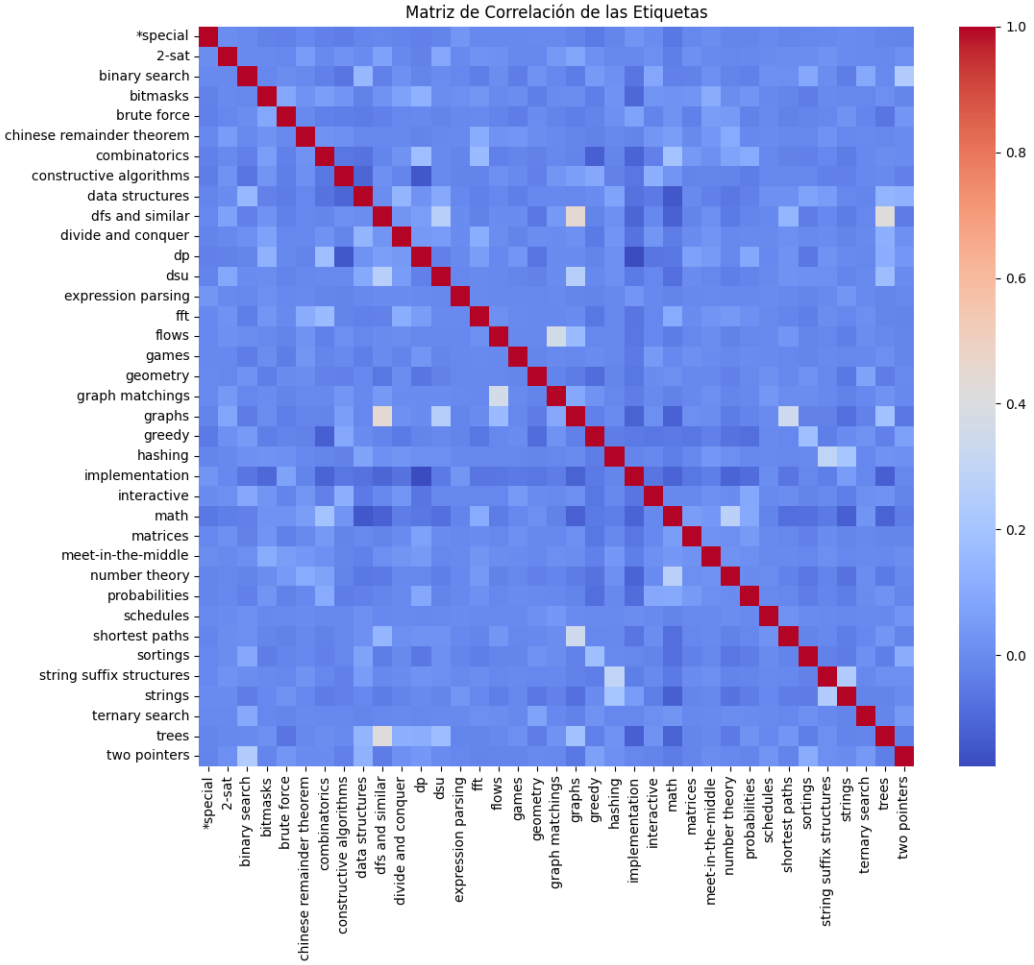
- Distribución de idiomas



- Nube de palabras asociada a la descripción de los problemas



- Matriz de correlación de las etiquetas



2.2. Preprocesamiento de datos

En el preprocesamiento de datos, se realizaron varios pasos para limpiar y normalizar los textos de las descripciones y otros campos del conjunto de datos. A continuación, se describen las principales etapas y funciones utilizadas:

1. **Corrección de capitalización después de puntos**
   - **Función:** `processing_dot_capitalize`
   - **Descripción:** Añade un espacio antes de una letra mayúscula si viene después de un punto, para corregir errores de capitalización en los textos.
2. **Reemplazo de notación exponencial**
   - **Función:** `replace_exponent_notation`
   - **Descripción:** Convierte notaciones como 10^6 a su equivalente numérico 1000000 usando expresiones regulares.
3. **Espaciado entre signos de dólar**
   - **Función:** `add_spacing_between_dollar_signs`
   - **Descripción:** Añade espacios alrededor de los símbolos $$$ para facilitar el procesamiento posterior.
4. **Conversión a minúsculas**
   - **Función:** `convert_to_lowercase`
   - **Descripción:** Convierte todo el texto a minúsculas para una normalización uniforme.
5. **Cálculo de multiplicaciones en el texto**
   - **Función:** `calculate_multiplication`
   - **Descripción:** Evalúa y reemplaza expresiones de multiplicación como 2 \cdot 100000 por su resultado 200000.
6. **Preprocesamiento general**
   - **Función:** `preprocessing`
   - **Descripción:** Aplica todas las funciones anteriores secuencialmente para limpiar y normalizar el texto.
7. **Tokenización y lemmatización**
   - **Función:** `split_sentences, split_words, lemmatization`
   - **Descripción:** Divide el texto en oraciones y palabras, y aplica lematización para reducir las palabras a su forma base.
8. **Eliminación de stopwords**
   - **Función:** `remove_stopwords`
   - **Descripción:** Elimina palabras comunes (stopwords) para reducir el ruido en los datos.
9. **Procesamiento de descripciones**
   - **Función:** `get_preprocessed_sentence`
   - **Descripción:** Aplica todas las técnicas de preprocesamiento a cada descripción en el conjunto de datos.

10. **Procesamiento de límites de tiempo y memoria**

    - **Descripción:** Convierte los límites de tiempo y memoria a formatos numéricos apropiados, eliminando unidades innecesarias y valores nulos.

11. **Eliminación de columnas innecesarias**

    - **Descripción:** Se eliminan columnas no relevantes como `input_file` y `output_file` para simplificar el conjunto de datos.

En resumen, el preprocesamiento de datos incluyó la normalización y limpieza del texto, lematización, eliminación de stopwords, procesamiento de signos de dólar, tokenización y el manejo de valores en columnas específicas como límites de tiempo y memoria. Estas transformaciones aseguraron que los datos estuvieran en un formato adecuado para ser utilizados en el modelo de clasificación.

3. Estado del arte

3.1. Revisión bibliográfica

En la revisión bibliográfica analizado encontramos 2 líneas de investigación fundamentales, modelos que analizaban solo el lenguaje natural y por otro lado modelos que se enfocaban análisis en el análisis de código (es decir dado el texto o el ast de un código en un lenguaje específico daba como salida los tags asociados a dicho código). A continuación se relacionan los papers estudiados.

- Preprocesamiento de Lenguaje Natural

Paper	Year	Link	Models	Results	Dataset	Methods
Predicting algorithmic approach for programming problems from natural language problem description	2016	https://ashishbora.github.io/assets/projects/nlp/report.pdf	Long Short Term Memory (LSTM), Random Forest, dummy classifier	Only Random Forest outperformed the dummy classifier, which predicted the most popular class	Codeforces, considering only the first tag for each problem	Pre-trained word2vec vectors and one-hot encoding for input data representation
Predicting algorithm classes for programming word problems	2019	https://aclanthology.org/D19-5511/	Convolutional Neural Networks (CNN), ensemble of CNNs, human predictors	Best results by ensemble of CNNs; human predictors had better results than the best model but with an F1-macro score of 0.43 on top-20 multi-label classification	Codeforces, predicting 10 and 20 most frequent tags	Multi-class and multi-label classification approaches for tag prediction

Multi-label classification for automatic tag prediction in the context of programming challenges	2019	https://arxiv.org/abs/1911.12224	Long Short Term Memory (LSTM)	Best F1 score by LSTM over one-hot encoding; best Weighted Hamming Score by LSTM over word2vec	Codeforces and TopCoder, tags sorted into 9 classes	Doc2Vec, LSTM over word2vec, LSTM over one-hot encoding
Classification of Programming Problems based on Topic Modeling	2019	https://dl.acm.org/doi/10.1145/3323771.3323795	k-Nearest Neighbors (kNN), Random Forest (RF), Multinomial Naive Bayes (MNB), Multilayer Perceptron (MLP)	Final accuracy did not improve much compared to TF-IDF baseline (0.86 vs 0.88 accuracy); positive impact on kNN and MNB, negative on RF		Topic modeling (LDA, NMF) for vectorization; classification algorithms: kNN, RF, MNB, MLP

- Análisis de Código

Paper	**Year**	**Link**	**Models**	**Results**	**Dataset**	**Methods**
Automatic algorithm recognition of source-code using machine learning	2017	https://www.semanticscholar.org/paper/Automatic_Algorithm_Recognition_of_Source_Code_Shalaby_Mehrez/641beb8d201a9bda_27dd0b5a7727116_cd47c7cb9	Traditional classification algorithms	Successful application of traditional classification algorithms and code metrics for classifying solutions	Codeforces	Metric-based approach to source code vectorization; 30 different software metrics (e.g., number of variables of specific types, lines of code, number of loops, number of nested loops)

Classification and recommendation of competitive programming problems using CNN	2017	https://www.researchgate.net/publication/321868484_Classification_and_Recommendation_of_Competitive_Programming_Problems_Using_CNN	CNN character-wise	Managed to classify solutions into four classes; combining information from all submitted solutions improved classification	Codeforces	Character-wise CNN; proposed combining information from classifications of individual solutions

4. Propuestas de solución

- Clasificación Multietiqueta con TF-IDF y Naive Bayes

4.1. Clasificación Multietiqueta usando Deep Learning

Durante el estudio del estado del arte notamos que un clasificador de algoritmos dado el texto basado en CNN puede lograr un rendimiento casi humano. Sin embargo, estas las arquitecturas presentadas no manejan eficazmente secuencias largas, que son comunes en las descripciones de problemas de esta índole. Para abordar esta limitación, adoptamos arquitecturas recientes basadas en transformers, que son más adecuadas para manejar secuencias largas. Nuestro método aborda la clasificación de etiquetas múltiples, ya que cada problema de algoritmo puede pertenecer a varias etiquetas simultáneamente.

Para lograr esto se define una función F como un extractor de features que convierte un texto en un espacio de representaciones vectoriales (embeddings). Luego, utilizamos una cabeza de clasificación H sobre el extractor. Nuestro modelo resuelve el problema de la clasificación de etiquetas múltiples usando una función de pérdida de entropía cruzada binaria:

$$E_{(x,y)\in D_{train}}[l(H(F(x)), y)]$$

donde l es una pérdida de entropía cruzada binaria para las categorías y del problema x.

Recolectamos en total 7968 problemas de algoritmos de CodeForces, con 37 etiquetas distintas. Utilizamos un extractor de features basado en BERT y una red de cabeza de clasificación. Usamos la arquitectura BigBird en nuestra modelación que nos permite introducir extensas secuencias de tokens.

5. Experimentación y resultados

5.1. Clasificación Multietiqueta con TF-IDF y Naive Bayes

5.1.1. Introducción

La clasificación multietiqueta es una variante de la clasificación en la que cada instancia puede pertenecer a múltiples clases simultáneamente. En este estudio, utilizamos la vectorización TF-IDF para la extracción de características y

un clasificador Naive Bayes para la clasificación. El objetivo principal es evaluar el rendimiento del modelo en varias métricas de evaluación.

5.1.2. Metodología

- Preprocesamiento de Datos

El conjunto de datos preprocesado pasa a utilizar la vectorización TF-IDF para convertir los datos de texto en características numéricas. Las etiquetas se transformaron a un vector binario utilizando `MultiLabelBinarizer` para adaptarse a la naturaleza multietiqueta del problema.

- Entrenamiento del Modelo

La clasificación se realizó utilizando un clasificador Naive Bayes dentro de un marco One-vs-Rest. El conjunto de datos se dividió en conjuntos de entrenamiento y prueba utilizando una división 80-20. Una vez el modelo predecía habían problemas a los cuales no se les asignaba ninguna etiqueta, cosa que no sucede en el codeforces, por lo cual se le hace asignar a dichos problemas la etiqueta más probable, garantizando así que todo problema contenga al menos una etiqueta.

```
vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(text_data)
mlb = MultiLabelBinarizer()
y = mlb.fit_transform(labels)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

clf = OneVsRestClassifier(MultinomialNB())
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
```

Listing 1: Naive Bayes

5.1.3. Resultados

- Informe de Clasificación

```
classification_report(y_test, y_pred, target_names=mlb.classes_)
```

Listing 2: Informe de Clasificación

	precision	recall	f1-score	support
brute force	0.200000	0.016393	0.030303	61.0
constructiv ealgorithms	0.500000	0.013699	0.026667	73.0
data structures	1.000000	0.040816	0.078431	49.0
dfs and similar	0.000000	0.000000	0.000000	4.0
dp	0.000000	0.000000	0.000000	50.0
geometry	1.000000	0.176471	0.300000	17.0
greedy	0.522727	0.414414	0.462312	111.0
implementation	0.515789	0.690141	0.590361	142.0
math	0.630435	0.287129	0.394558	101.0
strings	0.833333	0.370370	0.512821	27.0

- Accuracy

`Overall Accuracy: 14.46\%`

- Matriz de Confusión

Se tiene un jupyter interactivo en el que se puede consultar la matriz de confusión para cada etiqueta

- Precision ,F1-Score ,Recall y Support para cada etiqueta

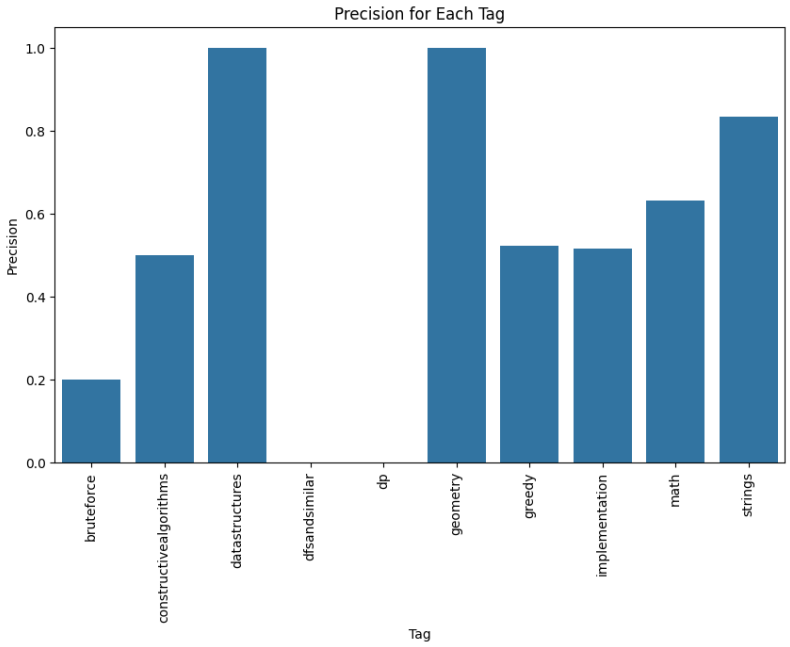


Figura 1: Precision

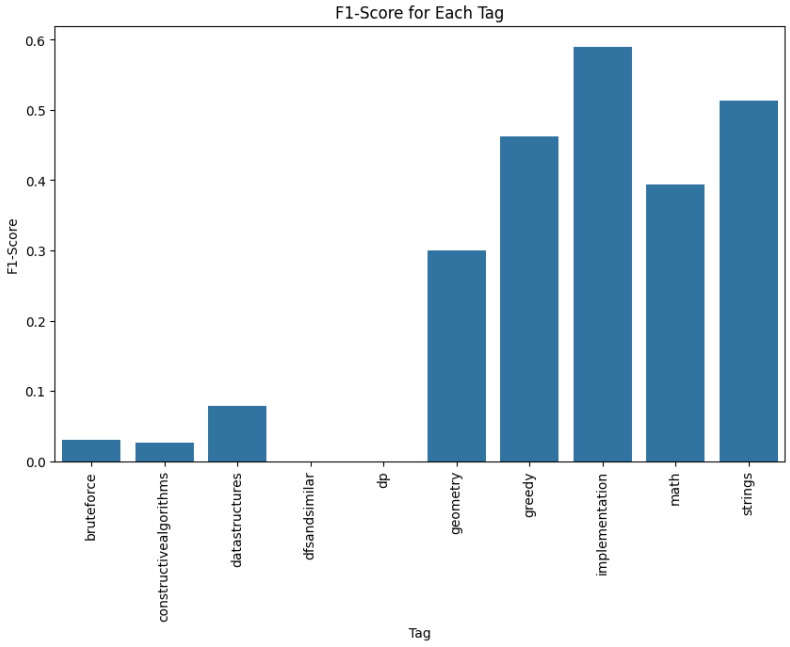


Figura 2: F1-Score

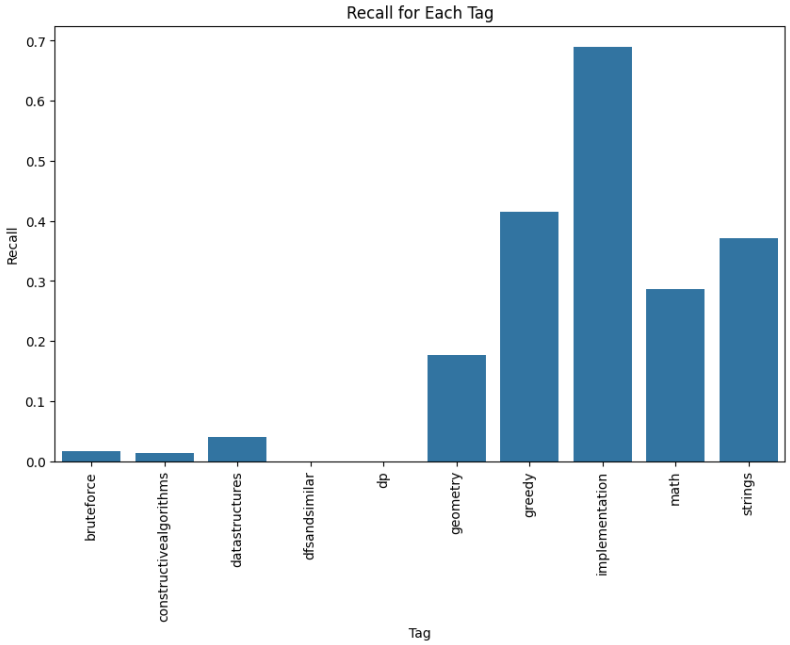


Figura 3: Recall

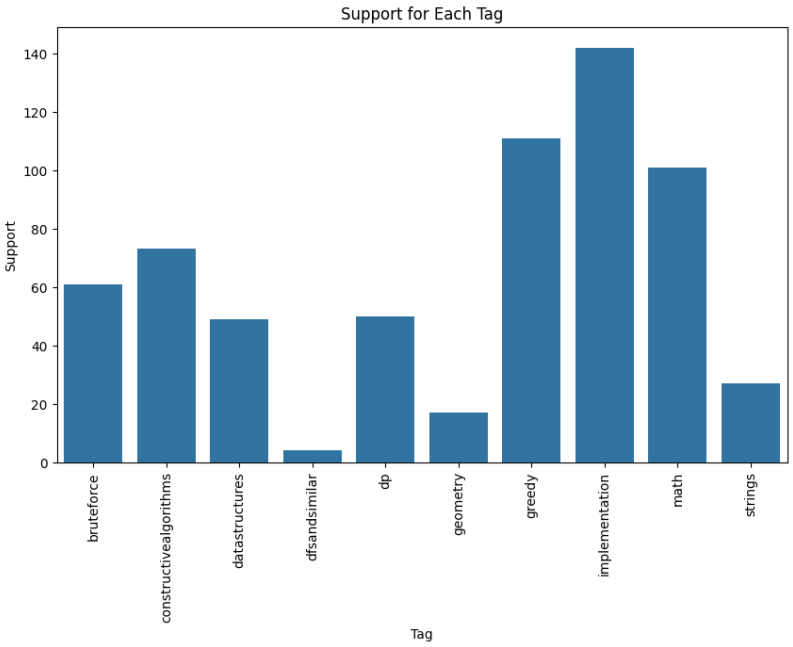


Figura 4: Support

5.2. Clasificación Multietiqueta usando Deep Learning

A pesar de haber recolectado un dataset de problemas tan extenso, por cuestiones de rapidez en el cómputo decidimos solo usar 1625 de ellos, la mitad del total teniendo en cuenta únicamente los problemas que solo están tagueados con las etiquetas más comunes. En nuestros experimentos, evaluamos varias configuraciones iniciales para el entrenamiento del modelo de clasificación de texto con el objetivo de validar la efectividad de nuestro conjunto de datos propuesto. Sin embargo, no nos fue posible concluirlo, pudiendo completar solo el 25 % de las epochs previstas. Estos fueron los resultados alcanzados en el último checkpoint de nuestro modelo:

	auc-roc
brute force	0.53433
constructiv ealgorithms	0.60461
data structures	0.55745
dfs and similar	0.39719
dp	0.53294
geometry	0.64132
greedy	0.62073
implementation	0.60420
math	0.65266
strings	0.67921

- **ROC-AUC Score** = 0,5824715558064865: Esta métrica indica que, en promedio, el modelo tiene una capacidad moderada para distinguir entre las clases. Un valor de 0.5 sugiere que el modelo no tiene mejor rendimiento que una clasificación aleatoria, mientras que un valor de 1 indica una clasificación perfecta. Por lo tanto, un AUC-ROC de aproximadamente 0.582 sugiere que hay margen para mejorar el modelo. AUC-ROC es el Área Bajo la Curva del Receptor Operador Característico.
- **F1 Score** = 0,3080225264908538: El F1 score combina precisión y recall en una sola métrica. Un valor de 0.308 indica que el modelo tiene un rendimiento bajo en términos de equilibrio entre estos dos aspectos. Nuevamente, esto sugiere que hay espacio para mejoras.
- **Umbral de decisión** = 0,001: Este valor indica el umbral utilizado para convertir las probabilidades de las etiquetas en predicciones binarias. Un umbral tan bajo podría estar indicando que el modelo está tratando de ser muy inclusivo (es decir, prefiere clasificar algo como positivo con una probabilidad muy baja), lo cual evidentemente está afectando la precisión de las predicciones.

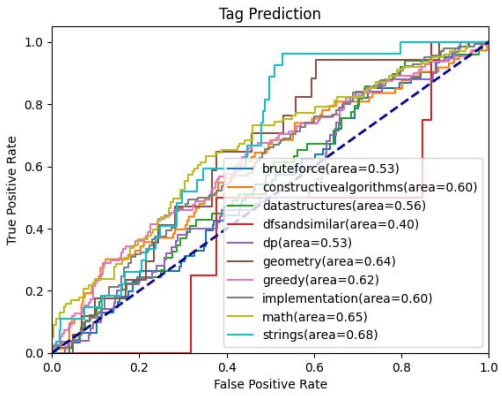


Figura 5: Tags ROC-AUC

Dado el rendimiento moderado en AUC-ROC y bajo en F1 score, además de la variabilidad notable en el rendimiento del modelo entre diferentes etiquetas se podría explorar mejorar el preprocesamiento de datos, el ajuste de hiperparámetros del modelo, o incluso el uso de técnicas de balanceo de clases para obtener mejores métricas.

5.3. Chat GPT

Para analizar que tan bien se comportaba Chat GPT 3.5, prediciendo los tags de Codeforces, usamos prompt engineering con un preprocesamiento sobre los problemas:

```
def prompt(description):
    all_tags_str = ', '.join(all_tags)
    return f'Give this set of {all_tags_str} tags and this problem ${description}, give me the set
    of problem tags in the following format: greedy, implementation, dp'
```

Listing 3: Prompt para tagear los problemas

5.3.1. Resultados y comparación con el resto de modelos

	Metric	ChatGPT	Naive Bayes	Resultados Reales
0	Accuracy	0.020000	0.146000	0.12615384615384614
1	F1 (macro)	0.002431	0.240000	0.3012989902488488
2	F1 (micro)	0.020000	0.390000	0.4228110599078341
3	F1 (weighted)	0.017556	0.320000	0.3973457965486768
4	F1 (samples)	0.002431	0.390000	0.4023785103785104

En el resto de los modelos comparados se obtuvieron mejores métricas en general, también vale destacar que no nos fue posible usar ChatGPT 4o, modelo de mayor complejidad con le que esperamos que se comporte mejor, además de que solo nos fue posible analizar con ChatGPT 100 problemas ya que no disponíamos de la api y todo el trabajo se realizo de manera manual. Otro aspecto a destacar, fue que en la mayoría de los problemas ChatGPT respondió con los tags que se le mostraban de ejemplo en el prompt lo que en alguna medida demuestra que el modelo en alguno o la mayoría de los casos alucinaba, algo que es común en estos modelos de lenguaje.

6. Discusión de los resultados

Debe incluir la repercusión ética de las soluciones.

7. Conclusiones y trabajo futuro