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.

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

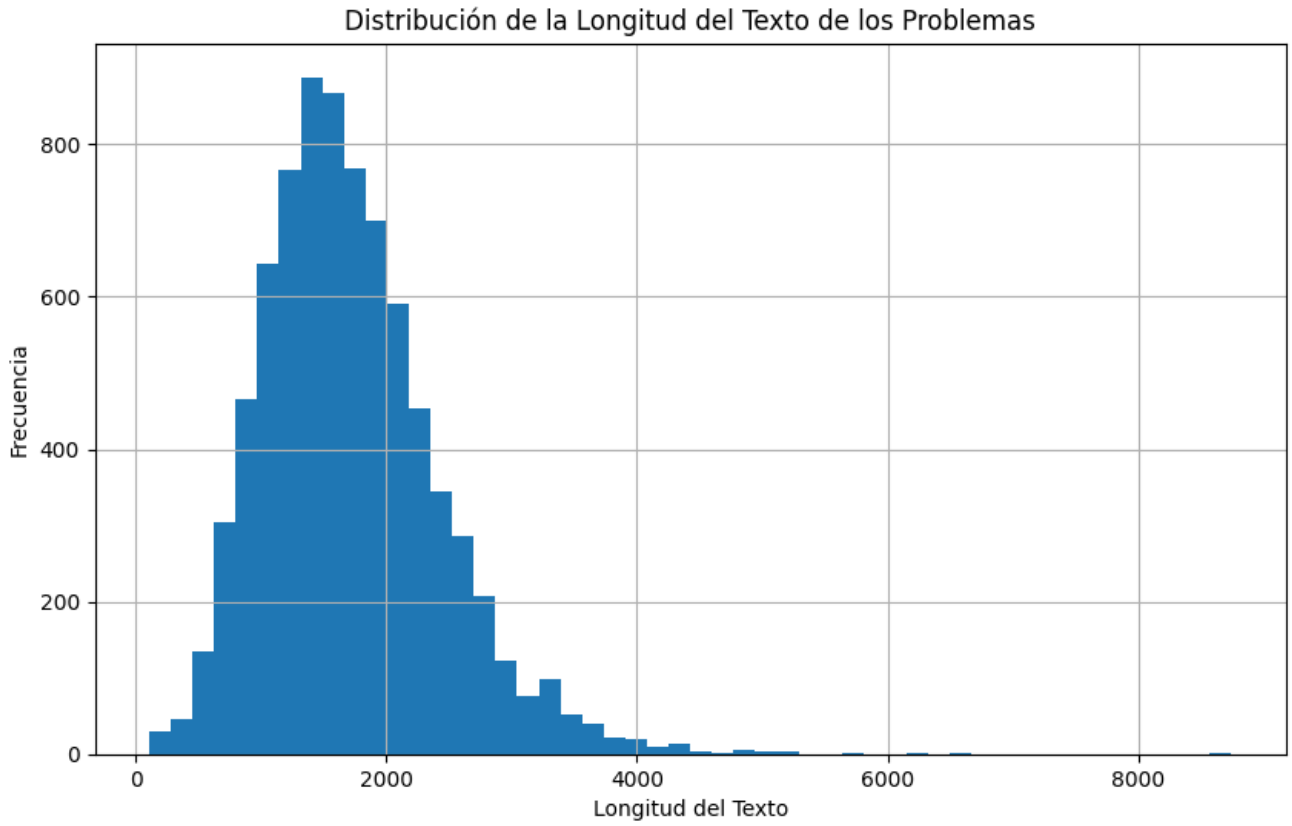
Para esto primero exploramos los datos buscando filas con ausencia de datos y eliminamos datos que no son de interés para nuestro problema.

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.

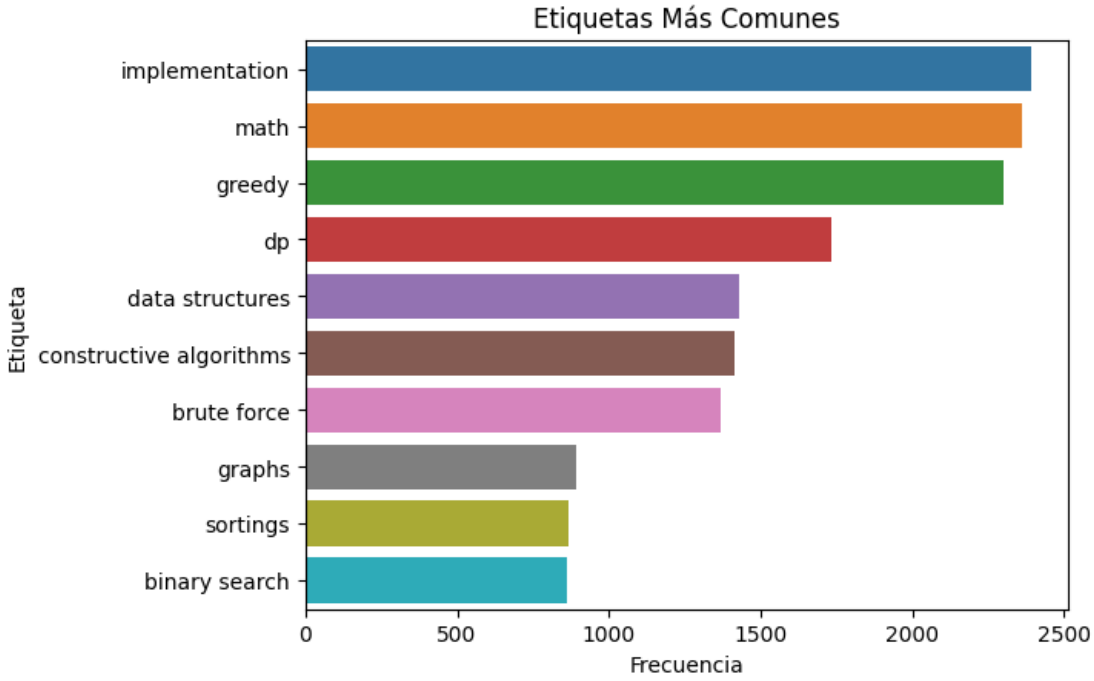
2.1.1. Información recopilada sobre los datos

- Distribución de la longitud del texto de los Problemas



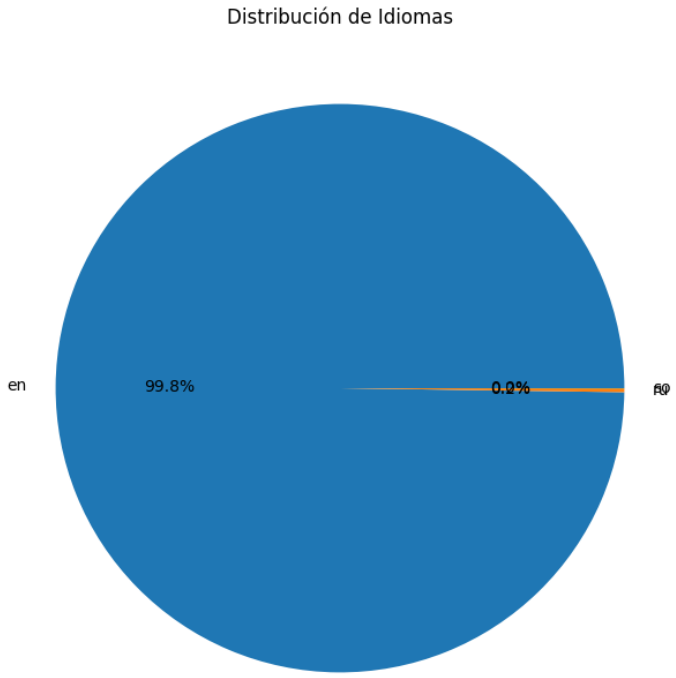
en la gráfica anterior se puede observar que la longitud de los problemas que más abunda sobre los 2000 caracteres.

- Frecuencia de las etiquetas más comunes



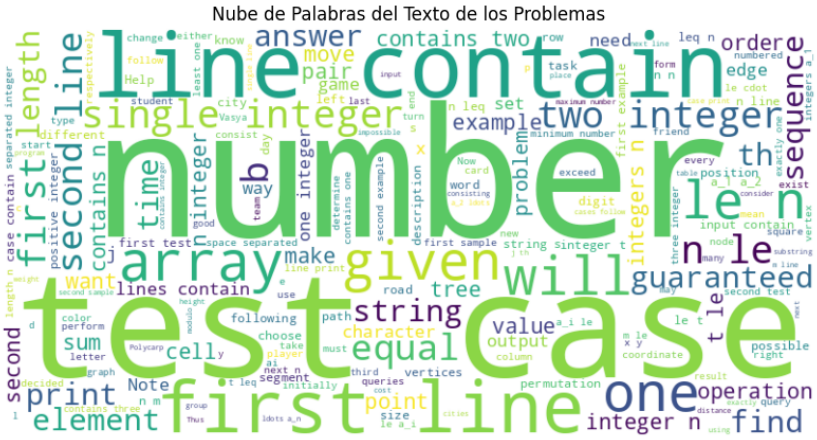
en la gráfica anterior se puede observar que la etiqueta más común es **implementation**, esto es bastante común, ya que la dificultad de muchos problemas además de la idea radica en la implementación de los algoritmos (sería interesante medir como se comporta la clasificación de un problema usando esta etiqueta si solo nos enfocáramos en el código del problema).

- Distribución de idiomas



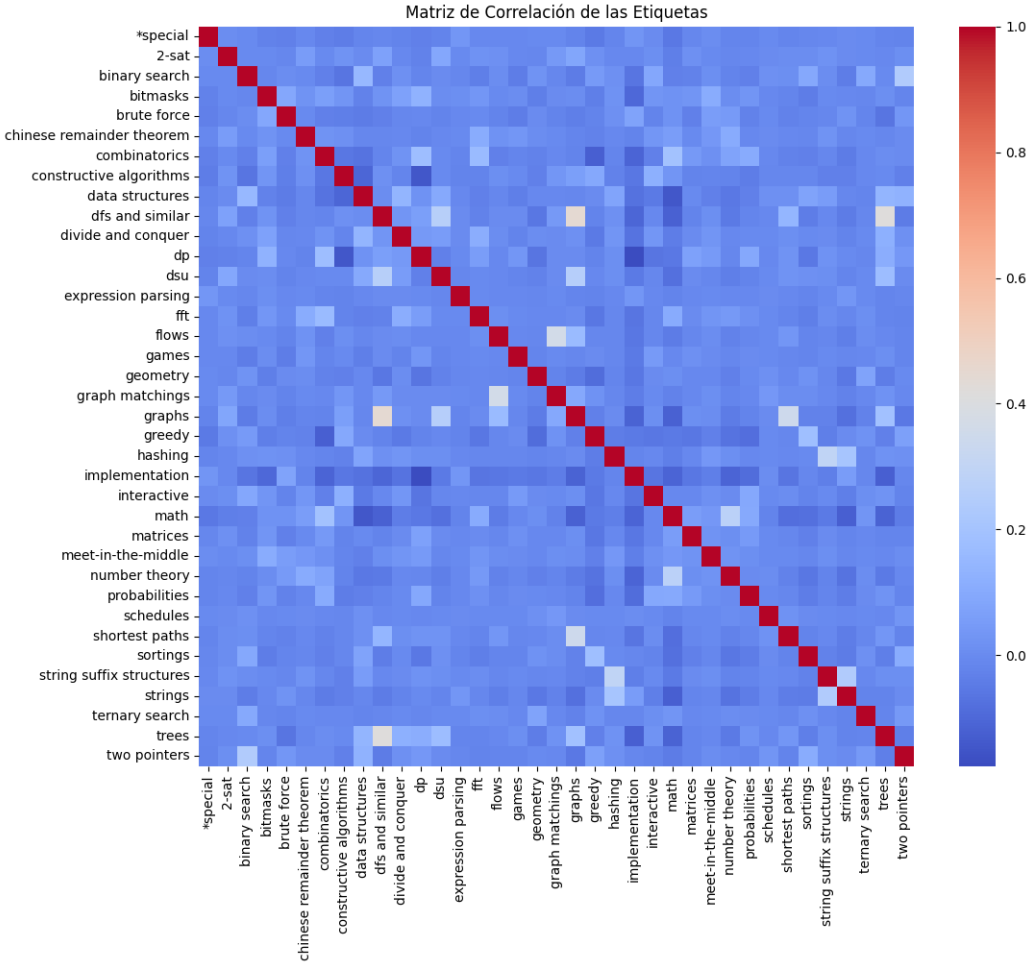
en la gráfica anterior se puede observar que el idioma más común es el **Inglés** y una pequeña parte en ruso, en este trabajo solo consideraremos los ejercicios en idioma **Inglés**.

- Nube de palabras asociada a la descripción de los problemas



en la gráfica anterior se pueden observar las palabras que más se repiten, la palabra más común es **Number**, lo que coincide con el tema fundamental de estos ejercicio ya que la mayoría tratan problemas relacionados con la Matemática. Después se observan las palabras **test** y **case**, título de la sección donde se describen los casos de prueba del problema.

- Matriz de correlación de las etiquetas



en la gráfica anterior se puede observar la matriz de correlación de las etiquetas que muestra para cada para de etiquetas la frecuencia de los problemas donde estas se aparecen a la vez. Se puede observar una estrecha relación entre los problemas con las etiquetas **dfs and similar**, **graphs** y **trees** (esto tomando cada par de etiquetas donde participan 2 de estas 3, con una mayor relación para el para **dfs and similar** y **graphs**), esto a raíz de que para los problemas de grafos y en un caso particular de árboles es muy común una vía de solución usando **DFS**.

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	**Año**	**Link**	**Modelos**	**Resultados**	**Dataset**	**Métodos**
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, clasificador dummy	Solamente Random Forest presentó mejoras sobre el clasificador dummy, el cual predijo el tag más común	Codeforces, considerando solo el primer tag para cada problema	Vectores word2vec pre entrenado y codificación one-hot para representar la entrada de datos
Predicting algorithm classes for programming word problems	2019	https://aclanthology.org/D19-5511/	Convolutional Neural Networks (CNN), ensemble de CNNs, predicciones humanas	Mejores resultados para el ensemble de CNNs; las predicciones humanas tuvieron mejores resultados que el resto de los modelos pero con una puntuación de 0.43 del macro-F1 en la clasificación multietiqueta de los primeros 20	Codeforces, prediciendo 10 y 20 de los tags más frequentes	Acercamientos de clasificación mulitclase y multietiqueta para la predicción de los tags
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)	Mejor puntuación F1 para LSTM sobre la codificación one-hot; mejor Weighted Hamming Score para LSTM sobre word2vec	Codeforces y TopCoder, tags ordenados en 9 clases	Doc2Vec, LSTM sobre word2vec, LSTM sobre codificación one-hot

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)	La precisión final no mejoró mucho en comparación con la línea de base de TF-IDF (0,86 frente a 0,88 de precisión); impacto positivo en kNN y MNB, negativo en RF		Modelado de temas (LDA, NMF) para vectorización; algoritmos de clasificación: kNN, RF, MNB, MLP

- Análisis de Código

Paper	**Año**	**Link**	**Modelos**	**Resultados**	**Dataset**	**Métodos**
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	Algoritmos de clasificación tradicionales	Aplicación exitosa de algoritmos de clasificación tradicionales y métricas de código para clasificar soluciones	Codeforces	Enfoque basado en métricas para la vectorización del código fuente; 30 métricas de software diferentes (por ejemplo, número de variables de tipos específicos, líneas de código, número de bucles, número de bucles anidados)
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 a nivel de caracteres	Logró clasificar las soluciones en cuatro clases; Combinación de información de todas las soluciones presentadas: clasificación mejorada	Codeforces	CNN a nivel de caracteres; propuso combinar la información de las clasificaciones de las soluciones individuales

4. Propuestas de solución

4.1. Clasificación Multietiqueta con TF-IDF y Naive Bayes

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.

4.2. Clasificación Multietiqueta con TF-IDF y KNN

Al igual que en la sección anterior, usaremos el enfoque de modelo de aprendizaje multietiquetas. La vectorización usada es la misma que en Naive-Bayes, TF-IDF para extraer características y un clasificador KNN para la clasificación. El objetivo poder comparar el rendimiento de este modelo con los otros que hemos usado.

4.3. 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. 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

5.1.2. Resultados

	auc-roc
brute force	0.50440
constructive algorithms	0.50000
data structures	0.51020
dfs and similar	0.50000
dp	0.50000
geometry	0.52941
greedy	0.59057
implementation	0.62189
math	0.52669
strings	0.62795

Cuadro 3: ROC-AUC Scores

	auc-roc
brute force	0.50062
constructive algorithms	0.50486
data structures	0.52040
dfs and similar	0.50000
dp	0.50000
geometry	0.58823
greedy	0.60907
implementation	0.59370
math	0.60561
strings	0.68182

Cuadro 4: ROC-AUC Scores (Al menos una etiqueta por problema)

- F1 Score:

```
F1 Score: 0.16300
F1 Score (Al menos una etiqueta por problema): 0.23954
```

- ROC-AUC Score:

```
ROC-AUC Score: 0.54111
ROC-AUC Score (Al menos una etiqueta por problema): 0.56043
```

- Umbral de Decisión usado (Ambos Casos):

```
Umbral de Decisión: 0.5
```

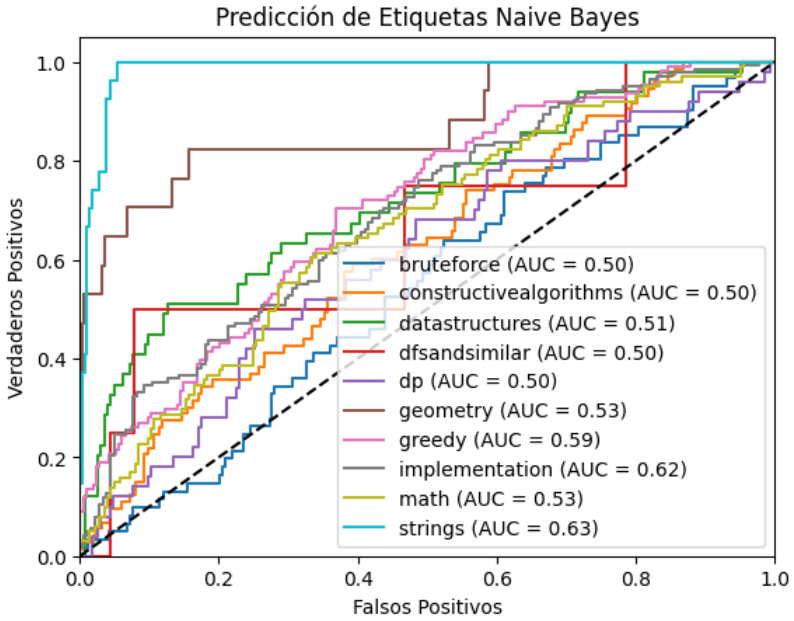


Figura 1: Etiquetas ROC-AUC

Como se pude apreciar en el Cuadro 3 y Cuadro 4 el agregarle la etiqueta más probable a aquellos problemas que no les fueron asignados por el modelo ninguna etiqueta hace una mejora en los resultados obtenidos en casi todas las métricas usadas, solo las etiqueta brute force e implementation obtienen un mejor valor en roc-auc sin hacer esto, lo cual está dado a que aquellos problemas sin etiquetar se le haya asignado su etiqueta incorrectamente. Con respecto a la Figura 1 podemos apreciar las curvas por encima de la diagonal, esta última que representa el azar ,lo que nos sugiere que nuestro modelo es capaz de predecir con un cierta confianza de que sea cierto.

5.2. Clasificación Multietiqueta con TF-IDF y KNN

5.2.1. Metodología

- Preprocesamiento de Datos

además de preprocesar la descripción de los problemas para usar TF-IDF, al ser KNN un modelo que solo se puede entrenar con valores numéricos, también se binarizaron los tags de los problemas, y se eliminaron columnas poco relevantes como el idioma y la identificación de los problemas.

- Entrenamiento del Modelo

La clasificación se realizó utilizando un clasificador KNN dentro de un marco MultiOutput. El conjunto de datos se dividió en conjuntos de entrenamiento, validación y prueba utilizando una división 60-20-20.
Pero cuál sería el mejor valor de K para el modelo, para esto se realizó una búsqueda de hiperparámetros con valores de K de 1 a 100 sobre el conjunto de validación, dando como resultado que K = 8 era el valor con mayor precisión en los resultados.

```
1
    neighbors = np.arange(1, 100)
2   reports = []
3   for neighbor in neighbors:
4       knn = KNeighborsClassifier(n_neighbors=neighbor)
5       mlb_knn = MultiOutputClassifier(knn)
6       mlb_knn.fit(X_train, y_train)
7       ypred = mlb_knn.predict(X_validation)
8       reports.append(classification_report(y_validation, ypred, zero_division = 0))
9
10
```

Listing 2: KNN

5.2.2. Resultados

	auc-roc
brute force	0.51701
constructiv ealgorithms	0.54449
data structures	0.60976
dfs and similar	0.50000
dp	0.50000
geometry	0.50000
greedy	0.54201
implementation	0.57758
math	0.52466
strings	0.73744

ROC AUC Scores: El valor ideal para ROC AUC es 1, mientras que un valor de 0.5 sugiere que el modelo no está mejor que el azar.

- **bruteforce**: 0.517 → Desempeño cercano al azar, pero ligeramente mejor.
- **greedy**: 0.542 → Ligeramente mejor que el azar, aunque lejos de ser excelente.
- **math**: 0.524 → Similar al azar, muestra poca capacidad del modelo para predecir correctamente.

- **datastructures**: 0.610 → Este es el primer puntaje que indica una mejora apreciable sobre el azar, sugiriendo que el modelo tiene mejor capacidad de clasificación en esta etiqueta.
- **implementation**: 0.577 → Un desempeño aceptable, pero aún limitado.
- **strings**: 0.737 → Buen desempeño, el modelo parece funcionar bastante bien en problemas de tipo "strings".
- **dp, geometry, dfsandsimilar**: 0.5 → Desempeño similar al azar, sin capacidad de predicción en estas categorías.
- **constructivealgorithms**: 0.544 → Ligeramente mejor que el azar.

ROC AUC Score (0.55529): Un ROC AUC Score de 0.555 indica que el modelo tiene un desempeño apenas mejor que el azar (0.5), pero muy lejos de ser un buen clasificador. Aunque hay una ligera capacidad de discriminación entre las clases, es bastante limitada y sugiere que el modelo no está capturando adecuadamente las características de los datos para predecir con precisión.

F1 Score (0.208): Un F1 Score bajo sugiere que el modelo está teniendo problemas para balancear la precisión y el *recall*, lo que implica que la clasificación en general está lejos de ser efectiva.

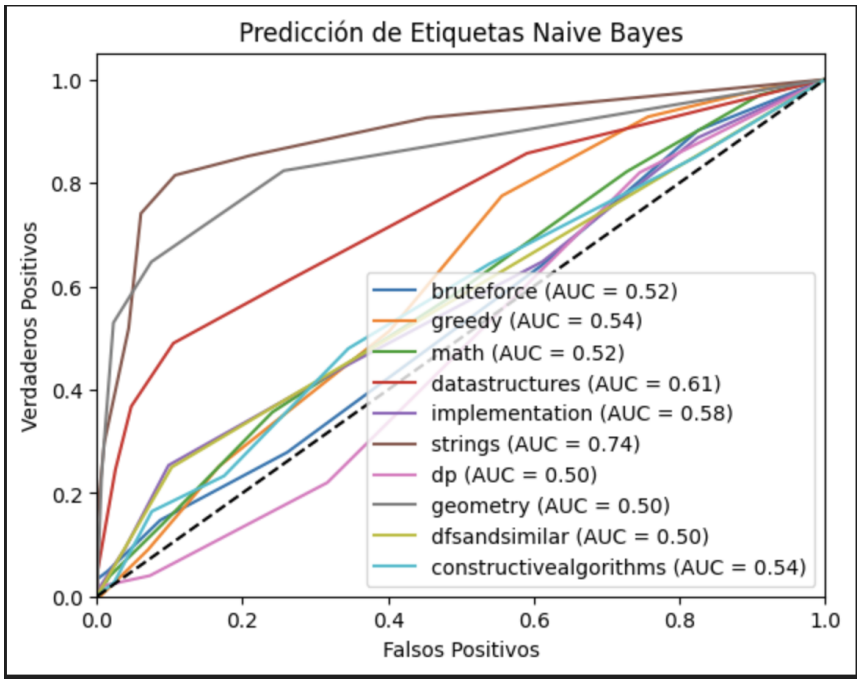


Figura 2: Etiquetas ROC-AUC

El modelo tiene un desempeño moderado en etiquetas como "datastructuresz "strings", pero lucha considerablemente con otras como "dp", "geometry", y "dfsandsimilar", donde su capacidad predictiva es casi nula.

5.3. 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
constructive algorithms	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.

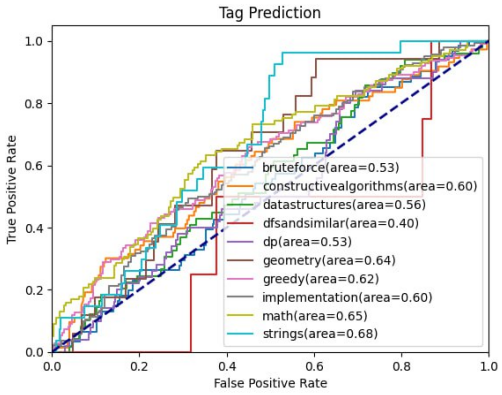


Figura 3: Tags ROC-AUC

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.

5.4. 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

	Metric	ChatGPT	Naive Bayes	KNN	Deep Learning
0	Accuracy	0.020000	0.146000	0.080000	0.12615384615384614
1	F1 (macro)	0.002431	0.240000	0.210000	0.3012989902488488
2	F1 (micro)	0.020000	0.390000	0.290000	0.4228110599078341
3	F1 (weighted)	0.017556	0.320000	0.260000	0.3973457965486768
4	F1 (samples)	0.002431	0.390000	0.240000	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

Los resultados obtenidos muestran una clara variabilidad en el rendimiento de los modelos utilizados para la tarea de etiquetado de problemas de Codeforces. A continuación se discuten las observaciones clave de las métricas presentadas:

- **Accuracy:** El modelo basado en ChatGPT presentó la menor precisión (0.020000), indicando que el modelo no fue efectivo en predecir las etiquetas correctas de los problemas. Por otro lado, el modelo de Deep Learning alcanzó la mayor precisión (0.12615384615384614), seguido por Naive Bayes (0.146000). Aunque estas precisiones son bajas, el modelo de Deep Learning muestra un rendimiento relativamente mejor.
- **F1 (macro):** Esta métrica mide el equilibrio entre la precisión y el recall para cada clase y promedia estos valores. Aquí, el modelo de Deep Learning obtuvo el valor más alto (0.3012989902488488), indicando un mejor desempeño general en todas las etiquetas. Naive Bayes y KNN también mostraron un rendimiento aceptable con valores de 0.240000 y 0.210000 respectivamente.
- **F1 (micro):** El F1 (micro) considera el total de verdaderos positivos, falsos negativos y falsos positivos. Los modelos Naive Bayes y Deep Learning destacaron con valores de 0.390000 y 0.4228110599078341 respectivamente. Estos resultados sugieren que, en términos de equilibrio global, ambos modelos gestionan bien la clasificación de las etiquetas.
- **F1 (weighted):** Similar al F1 (macro), pero ponderado por el soporte (número de verdaderos ejemplos en cada clase). Deep Learning nuevamente mostró el mejor desempeño (0.3973457965486768), seguido por Naive Bayes (0.320000). Este resultado indica que el modelo de Deep Learning maneja mejor las clases con un mayor número de ejemplos.

- **F1 (samples):** Esta métrica evalúa el rendimiento medio de la clasificación de etiquetas para cada muestra. El modelo de Deep Learning logró el mayor valor (0.4023785103785104), destacando su capacidad para predecir múltiples etiquetas por problema con mayor precisión. Naive Bayes también obtuvo un valor alto (0.390000), indicando un buen rendimiento en esta métrica.

En resumen, aunque ninguno de los modelos alcanzó un rendimiento excelente, el modelo de Deep Learning mostró un rendimiento superior en la mayoría de las métricas, seguido por Naive Bayes. Estos resultados sugieren que, con ajustes y mejoras adicionales, el enfoque de Deep Learning podría ser más efectivo para la tarea de etiquetado de problemas de Codeforces.

También debemos señalar que como repercusión ética este trabajo puede ser maliciosamente utilizado en concursos online de programación competitiva, donde se tiene tolerancia cero al fraude tagueando los problemas y de esta manera saber con que idea atacar, aunque cabe destacar que los problemas usualmente no son tan triviales. También positivamente este trabajo es una excelente forma de que en la preparación individual en vista a competencias se pueda utilizar para solo concentrarse en problemas que tengan una etiqueta específica, además creemos que los modelos están abiertos a problemas no solo del propio codeforce, sino de otras plataformas de programación competitiva. Las ideas abordadas en el trabajo estarán en disposición de todo aquel que quiera seguir avanzando con el tema en cuestión.

7. Conclusiones

En este proyecto, nos propusimos desarrollar un sistema de etiquetado automático para problemas de programación competitiva en la plataforma Codeforces utilizando técnicas de *machine learning*. Para ello, empleamos diversos modelos, incluyendo *k-Nearest Neighbors* (k-NN), Naive Bayes, ChatGPT 3.5 y un modelo basado en *embeddings* de texto usando Transformers. A lo largo del desarrollo, enfrentamos varios desafíos, entre ellos la naturaleza multi-etiqueta del problema y la diversidad de los enunciados de los problemas.

Nuestros resultados indican que, si bien los modelos implementados lograron etiquetar algunos problemas correctamente y los resultados son bastante superiores a los arrojados por ChatGPT 3.5, el rendimiento general no fue tan bueno como esperábamos. Las métricas de precisión, *recall* y F1-score fueron más bajas de lo deseado, reflejando la dificultad del problema. En particular, encontramos que:

- El modelo k-NN tuvo dificultades para manejar la alta dimensionalidad y diversidad de las características textuales.
- Naive Bayes, aunque rápido y eficiente, no capturó adecuadamente las complejidades y variaciones en el lenguaje de los enunciados.
- El modelo basado en *embeddings* usando Transformers mostró cierta promesa, pero requiere más ajuste y optimización para mejorar su desempeño.

Las limitaciones encontradas sugieren que los modelos actuales no son suficientemente sofisticados para capturar la variedad de etiquetas y la complejidad lingüística de los problemas de Codeforces.

8. Trabajo Futuro

Dado que nuestros resultados actuales no alcanzaron el rendimiento esperado, proponemos las siguientes áreas de trabajo futuro para mejorar el sistema de etiquetado:

1. **Aumento de Datos y Preprocesamiento**: Incrementar el volumen de datos de entrenamiento y aplicar técnicas de preprocesamiento más avanzadas para normalizar y limpiar los enunciados podría ayudar a mejorar el rendimiento de los modelos.

2. **Técnicas de Ensemble**: Utilizar técnicas de *ensemble*, combinando varios modelos para aprovechar sus fortalezas individuales y mitigar sus debilidades, podría conducir a un etiquetado más preciso y robusto.

3. **Evaluación y Validación Continua**: Implementar un sistema de evaluación continua y validación cruzada para ajustar y optimizar los modelos de manera iterativa, asegurando una mejora constante en el rendimiento.

4. **Colaboración con la Comunidad**: Involucrar a la comunidad de usuarios de Codeforces para obtener retroalimentación y posiblemente etiquetar manualmente un conjunto de datos, que podría ser utilizado para refinar y validar los modelos.

En conclusión, aunque nuestro sistema de etiquetado automático actual tiene limitaciones, este proyecto establece una base sólida para futuros desarrollos. Con más datos, modelos más avanzados y una validación continua, creemos que es posible mejorar significativamente la precisión y utilidad del etiquetado automático de problemas de programación competitiva.