Universidad de Córdoba

Escuela Politécnica Superior de Córdoba

máster universitario en inteligencia computacional e internet de las cosas

Clasificación No Convencional (CNC)

Práctica del Bloque 2: Métodos de Clasificación Multi-Etiqueta.

Estudiante:
Mabrouka SALMI

Correo electrónico:
z12salsm@uco.es

03 de febrero de 2024

Índice

Índice de figuras

Listings

1 Ejercicio 1

Seguimos el cuaderno mencionado y ejecutamos los pasos para familiarizarnos con problemas de multietiqueta en scikit-multilearn. El código reproducido se encuentra en este cuaderno de Colab.

2 Ejercicio 2

Listing 1: Información sobre el grupo de conjuntos de datos multietiqueta

```
from skmultilearn.dataset import load_dataset

# List of available datasets
datasets_list = ['scene', 'Corel5k', 'bibtex', 'enron',
'rcv1subset5', 'tmc2007_500', 'rcv1subset3', 'rcv1subset1',
'delicious', 'rcv1subset4', 'genbase','birds', 'emotions',
'rcv1subset2', 'mediamill', 'medical', 'yeast']

# Loop through each dataset and explore its characteristics
for dataset_name in datasets_list:
    # Load the dataset
    X, y, feature_names, label_names = load_dataset(dataset_name, '
    undivided')

    # Basic dataset information
    print(f"Dataset: {dataset_name}")
    print(f"Number of instances: {X.shape[0]}")
    print(f"Number of features: {X.shape[1]}")
    print(f"Number of labels: {y.shape[1]}")
```

2.1 Información sobre el conjunto de datos *scene*

```
Dataset: scene
Number of instances: 2407
Number of features: 294
Number of labels: 6
```

2.2 Información sobre el conjunto de datos *Corel5k*

```
Dataset: Corel5k
Number of instances: 5000
Number of features: 499
Number of labels: 374
```

2.3 Información sobre el conjunto de datos *bibtex*

```
Dataset: bibtex
Number of instances: 7395
Number of features: 1836
Number of labels: 159
```

2.4 Información sobre el conjunto de datos *enron*

```
Dataset: enron
Number of instances: 1702
Number of features: 1001
Number of labels: 53
```

2.5 Información sobre el conjunto de datos *rcv1subset5*

```
Dataset: rcv1subset5
Number of instances: 6000
Number of features: 47235
Number of labels: 101
```

2.6 Información sobre el conjunto de datos *tmc2007_500*

```
Dataset: tmc2007_500
Number of instances: 28596
Number of features: 500
Number of labels: 22
```

2.7 Información sobre el conjunto de datos *rcv1subset3*

```
Dataset: rcv1subset3
Number of instances: 6000
Number of features: 47236
Number of labels: 101
```

2.8 Información sobre el conjunto de datos *rcv1subset1*

```
Dataset: rcv1subset1
Number of instances: 6000
Number of features: 47236
Number of labels: 101
```

2.9 Información sobre el conjunto de datos *delicious*

```
Dataset: delicious
```

```
Number of instances: 16105
Number of features: 500
Number of labels: 983
```

2.10 Información sobre el conjunto de datos *rcv1subset4*

```
Dataset: rcv1subset4
Number of instances: 6000
Number of features: 47229
Number of labels: 101
```

2.11 Información sobre el conjunto de datos *genbase*

```
Dataset: genbase
Number of instances: 662
Number of features: 1186
Number of labels: 27
```

2.12 Información sobre el conjunto de datos *birds*

```
Dataset: birds
Number of instances: 645
Number of features: 260
Number of labels: 19
```

2.13 Información sobre el conjunto de datos *emotions*

```
Dataset: emotions
Number of instances: 593
Number of features: 72
Number of labels: 6
```

2.14 Información sobre el conjunto de datos *rcv1subset2*

```
Dataset: rcv1subset2
Number of instances: 6000
Number of features: 47236
Number of labels: 101
```

2.15 Información sobre el conjunto de datos *mediamill*

```
Dataset: mediamill
Number of instances: 43907
Number of features: 120
Number of labels: 101
```

2.16 Información sobre el conjunto de datos *medical*

```
Dataset: medical
Number of instances: 978
Number of features: 1449
Number of labels: 45
```

2.17 Información sobre el conjunto de datos *yeast*

```
Dataset: yeast
Number of instances: 2417
Number of features: 103
Number of labels: 14
```

3 Ejercicio 3

Esta sección muestra el código utilizado para calcular las estadísticas que caracterizan los conjuntos de datos mencionados anteriormente. A continuación, se presentan las características de cada conjunto de datos por separado.

Listing 2: Código utilizado en el cálculo de estadísticas de conjuntos de datos.

```python
from skmultilearn.dataset import load_dataset
from itertools import combinations
import numpy as np
from scipy.stats import chi2_contingency

# Define functions for calculations
def number_of_instances(X):
    return X.shape[0]

def number_of_attributes(X):
    return X.shape[1]

def number_of_labels(y):
    return y.shape[1]

# Function to calculate cardinality
def calculate_cardinality(y):
    return (1 / y.shape[0]) * np.sum(np.abs(y))
import pandas as pd
# Function to calculate density
def calculate_density(y):
    return np.mean(y)

# Function to calculate diversity
import scipy.sparse as sp
def calculate_diversity(y):
```

```
    unique_label_sets = len({tuple(row) for row in y.toarray()})  # Convert lil_matrix to array for set operations
    possible_label_sets = 2 ** y.shape[1]
    print("unique_label_sets: ", unique_label_sets)
    diversity = unique_label_sets / possible_label_sets
    return diversity

# Function to calculate imbalance ratio for a label
def calculate_imbalance_ratio(y):
    label_counts = np.sum(y, axis=0)
    max_label_count = np.max(label_counts)
    imbalance_ratios = max_label_count / label_counts
    avgIR = np.mean(imbalance_ratios)
    return avgIR
# Function to convert label sets to integers
def label_sets_to_integers(y):
    return y.dot(1 << np.arange(y.shape[-1] - 1, -1, -1))

# Function to calculate ratio of unconditionally dependent label pairs using chi-square test
def calculate_dependent_label_pairs(y):
    print("shape of y", y.shape)
    label_pairs = list(combinations(range(y.shape[1]), 2))
    dependent_pairs = 0

    for pair in label_pairs:
        contingency_table = np.zeros((2, 2), dtype=np.int64)

        label_1 = y[:, pair[0]].toarray().flatten()
        label_2 = y[:, pair[1]].toarray().flatten()

        contingency_table[0, 0] = np.sum((label_1 == 0) & (label_2 == 0))  # Both labels absent
        contingency_table[0, 1] = np.sum((label_1 == 0) & (label_2 == 1))  # Label 1 absent, Label 2 present
        contingency_table[1, 0] = np.sum((label_1 == 1) & (label_2 == 0))  # Label 1 present, Label 2 absent
        contingency_table[1, 1] = np.sum((label_1 == 1) & (label_2 == 1))  # Both labels present

        _, p, _, _ = chi2_contingency(contingency_table)

        if p < 0.01:  # Considering 99% confidence level
            dependent_pairs += 1

    total_pairs = len(label_pairs)
    return dependent_pairs / total_pairs
```

La segunda parte del programa *car.py* crea una función 3, basada en las funciones

previamente definidas 2, que itera sobre la lista de conjuntos de datos de entrada y calcula las estadísticas de cada uno. A continuación, se muestra el código utilizado y los resultados para cada uno de los conjuntos de datos.

Listing 3: Estadísticos de datasets

```
# List of available datasets
datasets_list = ['scene', 'Corel5k', 'bibtex', 'enron', '
    rcv1subset5', 'tmc2007_500', 'rcv1subset3', 'rcv1subset1', '
    delicious', 'rcv1subset4', 'genbase', 'birds', 'emotions', '
    rcv1subset2', 'mediamill', 'medical', 'yeast']

def car(datasets_list):
# Iterate through datasets and calculate measures
  for dataset_name in datasets_list:
    # Load the dataset
      X, y, _, _ = load_dataset(dataset_name, 'undivided')

      # Calculate measures
      n_instances = number_of_instances(X)
      n_attributes = number_of_attributes(X)
      n_labels = number_of_labels(y)
      car = calculate_cardinality(y)
      den = calculate_density(y)
      div = calculate_diversity(y)
      avgIR = calculate_imbalance_ratio(y)
      rDep = calculate_dependent_label_pairs(y)

      # Print results
      print(f"Dataset: {dataset_name}")
      print(f"Number of instances (n): {n_instances}")
      print(f"Number of attributes (f): {n_attributes}")
      print(f"Number of labels (l): {n_labels}")
      print(f"Cardinality (car): {car}")
      print(f"Density (den): {den}")
      print(f"Diversity (div): {div}")
      print(f"Average Imbalance Ratio per label (avgIR): {avgIR}")
      print(f"Ratio of unconditionally dependent label pairs (rDep)
    : {rDep}")
      print("\n")  # Add a separator between datasets
car(datasets_list)
```

3.1 Características del conjunto de datos *scene*

```
Dataset: scene
Number of instances (n): 2407
Number of attributes (f): 294
Number of labels (l): 6
```

```
Cardinality (car): 1.0739509763190693
Density (den): 0.17899182938651154
Diversity (div): 0.234375
Average Imbalance Ratio per label (avgIR): 1.2537840597429668
Ratio of unconditionally dependent label pairs (rDep): 0.9333333333333333
```

3.2 Características del conjunto de datos *Corel5k*

```
Dataset: Corel5k
Number of instances (n): 5000
Number of attributes (f): 499
Number of labels (l): 374
Cardinality (car): 3.5220000000000002
Density (den): 0.00941711229946524
Diversity (div): 8.251356501480639e-110
Average Imbalance Ratio per label (avgIR): 189.56755995212163
Ratio of unconditionally dependent label pairs (rDep): 0.021275680635403076
```

3.3 Características del conjunto de datos *bibtex*

```
Dataset: bibtex
Number of instances (n): 7395
Number of attributes (f): 1836
Number of labels (l): 159
Cardinality (car): 2.401893171061528
Density (den): 0.015106246358877536
Diversity (div): 3.908308998155935e-45
Average Imbalance Ratio per label (avgIR): 12.498259172453173
Ratio of unconditionally dependent label pairs (rDep): 0.0990367008996099
```

3.4 Características del conjunto de datos *enron*

```
Dataset: enron
Number of instances (n): 1702
Number of attributes (f): 1001
Number of labels (l): 53
Cardinality (car): 3.378378378378378
Density (den): 0.06374298827129016
Diversity (div): 8.359979375427429e-14
Average Imbalance Ratio per label (avgIR): 73.95279305225448
Ratio of unconditionally dependent label pairs (rDep): 0.10304789550072568
```

3.5 Características del conjunto de datos *rcv1subset5*

```
Dataset: rcv1subset5
Number of instances (n): 6000
Number of attributes (f): 47235
Number of labels (l): 101
Cardinality (car): 2.6414999999999997
Density (den): 0.02615346534653465
Diversity (div): 3.731312081695386e-28
Average Imbalance Ratio per label (avgIR): 69.68150575183465
Ratio of unconditionally dependent label pairs (rDep): 0.14376237623762375
```

3.6 Características del conjunto de datos *tmc2007_500*

```
Dataset: tmc2007_500
Number of instances (n): 28596
Number of attributes (f): 500
Number of labels (l): 22
Cardinality (car): 2.2196111344243947
Density (den): 0.10089141520110885
Diversity (div): 0.00027942657470703125
Average Imbalance Ratio per label (avgIR): 17.13430534368369
Ratio of unconditionally dependent label pairs (rDep): 0.8008658008658008
```

3.7 Características del conjunto de datos *rcv1subset3*

```
Dataset: rcv1subset3
Number of instances (n): 6000
Number of attributes (f): 47236
Number of labels (l): 101
Cardinality (car): 2.6141666666666667
Density (den): 0.02588283828382838
Diversity (div): 3.7037019500126504e-28
Average Imbalance Ratio per label (avgIR): 68.33256867076253
Ratio of unconditionally dependent label pairs (rDep): 0.15762376237623762
```

3.8 Características del conjunto de datos *rcv1subset1*

```
Dataset: rcv1subset1
Number of instances (n): 6000
Number of attributes (f): 47236
Number of labels (l): 101
Cardinality (car): 2.8796666666666666
Density (den): 0.028511551155115507
```

```
Diversity (div): 4.054745052836001e-28
Average Imbalance Ratio per label (avgIR): 54.49225821406634
Ratio of unconditionally dependent label pairs (rDep): 0.17445544554455444
```

3.9 Características del conjunto de datos *delicious*

```
Dataset: delicious
Number of instances (n): 16105
Number of attributes (f): 500
Number of labels (l): 983
Cardinality (car): 19.019993790748217
Density (den): 0.01934892552466757
Diversity (div): 1.9334646666443984e-292
Average Imbalance Ratio per label (avgIR): 71.1347977589772866
Ratio of unconditionally dependent label pairs (rDep): 0.1272363374929815
```

3.10 Características del conjunto de datos *rcv1subset4*

```
Dataset: rcv1subset4
Number of instances (n): 6000
Number of attributes (f): 47229
Number of labels (l): 101
Cardinality (car): 2.4836666666666667
Density (den): 0.02459075907590759
Diversity (div): 3.218552493301728e-28
Average Imbalance Ratio per label (avgIR): 89.37133217881669
Ratio of unconditionally dependent label pairs (rDep): 0.13603960396039605
```

3.11 Características del conjunto de datos *genbase*

```
Dataset: genbase
Number of instances (n): 662
Number of attributes (f): 1186
Number of labels (l): 27
Cardinality (car): 1.2522658610271904
Density (den): 0.046380217075081116
Diversity (div): 2.384185791015625e-07
Average Imbalance Ratio per label (avgIR): 37.31458058800413
Ratio of unconditionally dependent label pairs (rDep): 0.1282051282051282
```

3.12 Características del conjunto de datos *birds*

```
Dataset: birds
Number of instances (n): 645
```

```
Number of attributes (f): 260
Number of labels (l): 19
Cardinality (car): 1.013953488372093
Density (den): 0.05336597307221542
Diversity (div): 0.0002536773681640625
Average Imbalance Ratio per label (avgIR): 5.406995586615484
Ratio of unconditionally dependent label pairs (rDep): 0.0935672514619883
```

3.13 Características del conjunto de datos *emotions*

```
Dataset: emotions
Number of instances (n): 593
Number of attributes (f): 72
Number of labels (l): 6
Cardinality (car): 1.8684654300168633
Density (den): 0.31141090500281055
Diversity (div): 0.421875
Average Imbalance Ratio per label (avgIR): 1.4780684597524212
Ratio of unconditionally dependent label pairs (rDep): 0.9333333333333333
```

3.14 Características del conjunto de datos *rcv1subset2*

```
Dataset: rcv1subset2
Number of instances (n): 6000
Number of attributes (f): 47236
Number of labels (l): 101
Cardinality (car): 2.6341666666666668
Density (den): 0.026080858085808574
Diversity (div): 3.7628665179042263e-28
Average Imbalance Ratio per label (avgIR): 45.5138129116238
Ratio of unconditionally dependent label pairs (rDep): 0.15306930693069307
```

3.15 Características del conjunto de datos *mediamill*

```
Dataset: mediamill
Number of instances (n): 43907
Number of attributes (f): 120
Number of labels (l): 101
Cardinality (car): 4.375566538365181
Density (den): 0.04332244097391268
Diversity (div): 2.5854916168618662e-27
Average Imbalance Ratio per label (avgIR): 256.404697696982
Ratio of unconditionally dependent label pairs (rDep): 0.32594059405940595
```

3.16 Características del conjunto de datos *medical*

```
Dataset: medical
Number of instances (n): 978
Number of attributes (f): 1449
Number of labels (l): 45
Cardinality (car): 1.245398773006135
Density (den): 0.02767552828902522
Diversity (div): 2.6716406864579767e-12
Average Imbalance Ratio per label (avgIR): 89.50136135813314
Ratio of unconditionally dependent label pairs (rDep): 0.029292929292929294
```

3.17 Características del conjunto de datos *yeast*

```
Dataset: yeast
Number of instances (n): 2417
Number of attributes (f): 103
Number of labels (l): 14
Cardinality (car): 4.237070748862226
Density (den): 0.3026479106330161
Diversity (div): 0.0120849609375
Average Imbalance Ratio per label (avgIR): 7.196811040164219
Ratio of unconditionally dependent label pairs (rDep): 0.6703296703296703
```

4 Ejercicio 4

Realizamos el siguiente código para cargar el conjunto de datos, calcular las estadísticas de cada conjunto de datos:

Listing 4: El código para cargar los conjuntos de datos

```
import pandas as pd
import numpy as np
import arff, numpy as np
# Specify the correct path to your .dat file
file_path = 'bookmarks.dat'
# Load ARFF file using arff module
with open(file_path, 'r') as file:
    dataset = arff.load(file)
# Extract data and column names
BookmarksData = pd.DataFrame(dataset['data'], columns=[attr[0] for attr in dataset['attributes']])
BookmarksData = BookmarksData.apply(pd.to_numeric, errors='coerce').fillna(0)
# Display the first few rows of the DataFrame
#print(BookmarksData.head())
```

```
# Specify the correct path to your .arff file
file_path = 'CAL500.arff'
# Load ARFF file using arff module
with open(file_path, 'r') as file:
    dataset = arff.load(file)
# Extract data and column names
CAL500Data = pd.DataFrame(dataset['data'], columns=[attr[0] for attr in dataset['attributes']])
CAL500Data= CAL500Data.apply(pd.to_numeric, errors='coerce').fillna(0)
```

Listing 5: El código Python para obtener cada estadística

```
from itertools import combinations
import numpy as np
from scipy.stats import chi2_contingency
# Define functions for calculations
def number_of_instances(X):
    return X.shape[0]
def number_of_attributes(X):
    return X.shape[1]
def number_of_labels(y):
    return y.shape[1]
# Function to calculate cardinality
def calculate_cardinality(y):
        # Convert DataFrame to numeric (replace non-numeric values with NaN)
    y_numeric =  y.apply(pd.to_numeric, errors='coerce')
    n = y.shape[0] # Assuming each row is an example
    # Calculate cardinality
    return (1 / n) * np.nansum(np.abs(y_numeric))
# Function to calculate density
def calculate_density(y):
    return np.mean(y)
# Function to calculate diversity
def calculate_diversity(y):
    # Convert to numeric and handle non-numeric values
    y_numeric = pd.to_numeric(y.stack(), errors='coerce')
    # Handle NaN values (you can choose a different approach based on your requirements)
    y_numeric = y_numeric.dropna()  # Drop NaN values
    # Convert the Series to a NumPy array before applying set
    unique_label_sets = {tuple([label]) for label in y_numeric.astype(int)}
    possible_label_sets = 2 ** y.shape[1]
    print("unique_label_sets: ", len(unique_label_sets))
    diversity = len(unique_label_sets) / possible_label_sets
    return diversity
# Function to calculate imbalance ratio for a label
```

```
def calculate_imbalance_ratio(y):
    label_counts = np.sum(y, axis=0)
    max_label_count = np.max(label_counts)
    imbalance_ratios = max_label_count / label_counts
    avgIR = np.mean(imbalance_ratios)
    return avgIR
# Function to convert label sets to integers
def label_sets_to_integers(y):
    return y.dot(1 << np.arange(y.shape[-1] - 1, -1, -1))
# Function to calculate ratio of unconditionally dependent label pairs using chi-square test
def calculate_dependent_label_pairs(y):
    label_pairs = list(combinations(range(y.shape[1]), 2))
    dependent_pairs = 0
    for pair in label_pairs:
        contingency_table = np.zeros((2, 2), dtype=np.float64)
        # Access columns using iloc and to_numpy
        label_1 = y.iloc[:, pair[0]].to_numpy()
        label_2 = y.iloc[:, pair[1]].to_numpy()
        contingency_table[0, 0] = np.sum((label_1 == 0) & (label_2 == 0))
        contingency_table[0, 1] = np.sum((label_1 == 0) & (label_2 == 1))
        contingency_table[1, 0] = np.sum((label_1 == 1) & (label_2 == 0))
        contingency_table[1, 1] = np.sum((label_1 == 1) & (label_2 == 1))
        contingency_table += 1e-10
        _, p, _, _ = chi2_contingency(contingency_table)
        if p < 0.01:
            dependent_pairs += 1
    total_pairs = len(label_pairs)
    return dependent_pairs / total_pairs
```

Listing 6: La función para obtener las estadísticas de cada conjunto de datos

```
def carFunction(dataset, n_inputs, m_labels, dataset_name):
    # Load the dataset
    X = dataset.iloc[:,:n_inputs]
    y = dataset.iloc[:,n_inputs:m_labels+n_inputs]
    # Calculate measures
    n_instances = number_of_instances(X)
    n_attributes = number_of_attributes(X)
    n_labels = number_of_labels(y)
    car = calculate_cardinality(y)
    den = calculate_density(y)
    div = calculate_diversity(y)
    avgIR = calculate_imbalance_ratio(y)
    rDep = calculate_dependent_label_pairs(y)
```

```
    # Print results
    print(f"Dataset: {dataset_name}")
    print(f"Number of instances (n): {n_instances}")
    print(f"Number of attributes (f): {n_attributes}")
    print(f"Number of labels (l): {n_labels}")
    print(f"Cardinality (car): {car}")
    print(f"Density (den): {den}")
    print(f"Diversity (div): {div}")
    print(f"Average Imbalance Ratio per label (avgIR): {avgIR}")
    print(f"Ratio of unconditionally dependent label pairs (rDep):
   {rDep}")
    print("\n")  # Add a separator between datasets
# we know the number of inputs in CAL500Data is 68 and the number
   of labels is 174
carFunction(CAL500Data,68, 174, "CAL500")
carFunction(BookmarksData, 2150, 208, "Bookmarks")
```

4.1 Conjunto de datos *CAL500*

Descripción y Recolección Es un conjunto de datos musicales compuesto por 502 canciones. Cada una fue anotada manualmente por al menos tres anotadores humanos, quienes emplearon un vocabulario de 174 etiquetas relacionadas con conceptos semánticos. Estas etiquetas abarcan 6 categorías semánticas: instrumentación, características vocales, géneros, emociones, calidad acústica de la canción y términos de usoTurnbull et al. (2008); Moyano (2017).

Características del conjunto de datos Después de descargar el conjunto de datos, utilizamos el código en Python en 4, 5, y 6 para cargar los datos y obtener las estadísticas relacionadas, que se muestran a continuación:

```
Dataset: CAL500
Number of instances (n): 502
Number of attributes (f): 68
Number of labels (l): 174
Cardinality (car): 26.04382470119522
Density (den): 0.14967715345514493
Diversity (div): 8.352389719038111e-53
Average Imbalance Ratio per label (avgIR): 20.577783236245043
Ratio of unconditionally dependent label pairs (rDep): 0.16404225632848315
```

4.2 Conjunto de datos *Bookmarks*

Descripción y Recopilación Está basado en los datos del desafío de descubrimiento ECML/PKDD 2008 y contiene entradas de marcadores del sistema BibsonomyKatakis et al. (2008); Moyano (2017).

Características del conjunto de datos Después de obtener el conjunto de datos, extrajimos estadísticas útiles que se muestran a continuación:

```
Dataset: Bookmarks
Number of instances (n): 87856
Number of attributes (f): 2150
Number of labels (l): 208
Cardinality (car): 2.02814833363686
Density (den): 0.009750713142484905
Diversity (div): 4.861730685829017e-63
Average Imbalance Ratio per label (avgIR): 12.30801607208126
Ratio of unconditionally dependent label pairs (rDep): 0.28911185432924563
```

5 Ejercicio 5

5.1 Métodos de transformación de problemas

5.1.1 BinaryRlevance basada en clasificador SVM

El código de la aplicación

Listing 7: Ejemplo de uso de BinaryRelevance en Python

```
from skmultilearn.problem_transform import BinaryRelevance
from sklearn.svm import SVC
# initialize Binary Relevance multi-label classifier with an SVM
    classifier
classifier = BinaryRelevance(classifier = SVC(), require_dense = [
    False, True])
classifier.fit(X_train, y_train) # train
predictions_BinaryRelevance = classifier.predict(X_test) # predict
```

Resultados en el conjunto de datos *scene*

```
Hamming Loss:  0.06673582295988935
Recall (micro):  0.697265625
Recall (macro):  0.701266581400048
Accuracy:  0.6680497925311203
F1-score (micro) 0.7872105843439912
F1-score (macro) 0.7812045513402271
Precision score (micro):  0.9037974683544304
Precision score (macro):  0.8940930533172172
```

5.1.2 MultiOutputClassifier para entrenar un clasificador multietiqueta

En Scikit-Learn, utilizamos el objeto MultiOutputClassifier para entrenar el modelo de clasificación multietiqueta. La estrategia detrás de este modelo es entrenar un clasificador por etiqueta. Cada etiqueta tiene su propio clasificador.

Utilizamos el clasificador RandomForest en este caso, y MultiOutputClassifier se utilizó para extenderlo a todas las etiquetas.

El código de la aplicación

Listing 8: Ejemplo de uso de MultiOutputClassifier en Python

```
from sklearn.multioutput import MultiOutputClassifier
from sklearn.ensemble import RandomForestClassifier
clf = MultiOutputClassifier(RandomForestClassifier(n_estimators
    =100, random_state=42)).fit(X_train, y_train)
prediction_MultiOutputClassifier = clf.predict(X_test)
```

Resultados en el conjunto de datos *scene*

```
Hamming Loss:  0.07434301521438451
Recall (micro):  0.61328125
Recall (macro):  0.6164588148972981
Accuracy:  0.6058091286307054
F1-score (micro) 0.7449584816132859
F1-score (macro) 0.7260366915668488
Precision score (micro):  0.9486404833836858
Precision score (macro):  0.9428213416074726
```

5.1.3 Algoritmo ClassifierChain

El código de la aplicación

Listing 9: Ejemplo de uso de ClassifierChain en Python

```
from skmultilearn.problem_transform import ClassifierChain
from sklearn.ensemble import RandomForestClassifier
# Initialize classifier chains multi-label classifier
classifier = ClassifierChain(RandomForestClassifier(n_estimators
    =100, random_state=42))
# Training Random Forest model on train data
classifier.fit(X_train, y_train)
# predict
predictions_ClassifierChain = classifier.predict(X_test)
```

Resultados en el conjunto de datos *scene*

```
Hamming Loss:  0.07088520055325034
Recall (micro):  0.646484375
Recall (macro):  0.6545052595427095
Accuracy:  0.6452282157676349
F1-score (micro) 0.7635524798154555
F1-score (macro) 0.7526488069212586
Precision score (micro):  0.9323943661971831
Precision score (macro):  0.9248703933866308
```

5.1.4 Algoritmo LabelPowerSet

El código de la aplicación

Listing 10: Ejemplo de uso de LabelPowerSet en Python

```
from skmultilearn.problem_transform import LabelPowerset
# initialize label powerset multi-label classifier
classifier = LabelPowerset(RandomForestClassifier(n_estimators=100,
    random_state=42))
# train
classifier.fit(X_train, y_train)
# predict
predictions_LabelPowerset = classifier.predict(X_test)
```

Resultados en el conjunto de datos *scene*

```
Hamming Loss:  0.07330567081604426
Recall (micro):  0.765625
Recall (macro):  0.7728928209297797
Accuracy:  0.7572614107883817
F1-score (micro) 0.7871485943775101
F1-score (macro) 0.7836813962181335
Precision score (micro):  0.8099173553719008
Precision score (macro):  0.8035240794451557
```

5.1.5 RakelD

Distinct RAndom k-labELsets multi-label classifier RakelD divide el espacio de etiquetas en particiones iguales de tamaño k, entrena un clasificador Label Powerset por partición y realiza predicciones sumando los resultados de todos los clasificadores entrenados.

Código de la aplicación

Listing 11: Ejemplo de uso de RakelD en Python

```
from sklearn.naive_bayes import GaussianNB
from skmultilearn.ensemble import RakelD
classifier = RakelD( base_classifier=GaussianNB(),
   base_classifier_require_dense=[True, True], labelset_size=4 )
classifier.fit(X_train, y_train)
predictions_RakelD = classifier.predict(X_test)
```

Los resultados en el conjunto de datos *scene*

```
Hamming Loss:  0.14453665283540804
Recall (micro):  0.720703125
Recall (macro):  0.7168710874462804
Accuracy:  0.43775933609958506
F1-score (micro) 0.6384083044982698
F1-score (macro) 0.6378130316121121
```

```
Precision score (micro):  0.5729813664596274
Precision score (macro):  0.5887296533366185
```

5.1.6 MajorityVoting

Código de la aplicación

Listing 12: MajorityVoting en Python

```
from skmultilearn.ensemble import MajorityVotingClassifier
from skmultilearn.cluster import FixedLabelSpaceClusterer
from skmultilearn.problem_transform import ClassifierChain
from sklearn.naive_bayes import GaussianNB
classifier = MajorityVotingClassifier(
    clusterer = FixedLabelSpaceClusterer(clusters = [[1,2,3], [0, 2, 5], [4, 5]]),
    classifier = ClassifierChain(classifier=GaussianNB())
)
classifier.fit(X_train,y_train)
predictions_MajorityVotingClassifier = classifier.predict(X_test)
```

Resultados en el conjunto de datos *scene*

```
Hamming Loss:  0.21092669432918396
Recall (micro):  0.666015625
Recall (macro):  0.6478996185442473
Accuracy:  0.2842323651452282
F1-score (micro) 0.5278637770897833
F1-score (macro) 0.4908348980240542
Precision score (micro):  0.4371794871794872
Precision score (macro):  0.45098640073902446
```

5.2 Métodos de adaptación de algoritmos:

Recorremos algunos de los métodos de adaptación de algoritmos que existen en el paquete *scikit – multilearn* en PythonSzymański and Kajdanowicz (2017), e implementamos estos utilizando el conjunto de datos *scene*. A continuación, mostramos el código de la aplicación de estos métodos y su evaluación utilizando las métricas mencionadas en el curso.

5.2.1 MLkNN

El código de la aplicación

Listing 13: MLkNN en Python

```
from skmultilearn.adapt import MLkNN
classifier = MLkNN(k=3)
        # train
classifier.fit(X_train, y_train)
```

```
        # predict
predictions_MLkNN = classifier.predict(X_test)
import sklearn.metrics as metrics
print("_________________MLkNN Algorithm_________________")
print("Hamming Loss: ", metrics.hamming_loss(y_test,
    predictions_MLkNN))
print("Recall (micro): ", metrics.recall_score(y_test,
    predictions_MLkNN, average ='micro' ))
print("Recall (macro): ", metrics.recall_score(y_test,
    predictions_MLkNN, average ='macro' ))
print("Accuracy: ",metrics.accuracy_score(y_test, predictions_MLkNN
    ))
print("F1-score (micro)", metrics.f1_score(y_test,
    predictions_MLkNN, average='micro'))
print("F1-score (macro)", metrics.f1_score(y_test,
    predictions_MLkNN, average='macro'))
print("Precision score (micro): ", metrics.precision_score(y_test,
    predictions_MLkNN, average='micro'))
print("Precision score (macro): ", metrics.precision_score(y_test,
    predictions_MLkNN, average='macro'))
```

Resultados en el conjunto de datos *scene*

```
_________________MLkNN Algorithm_________________
Hamming Loss:  0.08402489626556017
Recall (micro):  0.734375
Recall (macro):  0.7457439097228975
Accuracy:  0.6804979253112033
F1-score (micro) 0.7557788944723618
F1-score (macro) 0.7545788383245644
Precision score (micro):  0.7784679089026915
Precision score (macro):  0.7898300771853067
```

5.2.2 MLTSVM

El código de la aplicación

Listing 14: MLTSVM in Python

```
from skmultilearn.adapt import MLTSVM
classifier = MLTSVM(c_k = 2**-1)
        # train
classifier.fit(np.array(X_train) ,np.array( y_train))
        # predict
predictions_MLTSVM = classifier.predict(np.array(X_test))
print("_________________MLTSVM Algorithm_________________")
print("Hamming Loss: ", metrics.hamming_loss(y_test,
    predictions_MLTSVM))
```

```
print("Recall (micro): ", metrics.recall_score(y_test,
    predictions_MLTSVM, average ='micro' ))
print("Recall (macro): ", metrics.recall_score(y_test,
    predictions_MLTSVM, average ='macro' ))
print("Accuracy: ",metrics.accuracy_score(y_test,
    predictions_MLTSVM))
print("F1-score (micro)", metrics.f1_score(y_test,
    predictions_MLTSVM, average='micro'))
print("F1-score (macro)", metrics.f1_score(y_test,
    predictions_MLTSVM, average='macro'))
print("Precision score (micro): ", metrics.precision_score(y_test,
    predictions_MLTSVM, average='micro'))
print("Precision score (macro): ", metrics.precision_score(y_test,
    predictions_MLTSVM, average='macro'))
```

Resultados en el conjunto de datos *scene*

```
_________________MLTSVM Algorithm_________________
Hamming Loss:  0.14349930843706776
Recall (micro):  0.427734375
Recall (macro):  0.43005195938661833
Accuracy:  0.2987551867219917
F1-score (micro) 0.5134818288393904
F1-score (macro) 0.5182870219927463
Precision score (micro):  0.6422287390029325
Precision score (macro):  0.6912230228193034
```

5.2.3 BRkNN

El código de la aplicación

Listing 15: BRkNN in Python

```
from skmultilearn.adapt import BRkNNaClassifier
classifier = BRkNNaClassifier(k=3)
        # train
classifier.fit(X_train, y_train)
        # predict
predictions_BRkNN = classifier.predict(X_test)
print("_________________BRkNN Algorithm_________________")
print("Hamming Loss: ", metrics.hamming_loss(y_test,
    predictions_BRkNN))
print("Recall (micro): ", metrics.recall_score(y_test,
    predictions_BRkNN, average ='micro' ))
print("Recall (macro): ", metrics.recall_score(y_test,
    predictions_BRkNN, average ='macro' ))
print("Accuracy: ",metrics.accuracy_score(y_test, predictions_BRkNN
    ))
```

```
print("F1-score (micro)", metrics.f1_score(y_test,
    predictions_BRkNN, average='micro'))
print("F1-score (macro)", metrics.f1_score(y_test,
    predictions_BRkNN, average='macro'))
print("Precision score (micro): ", metrics.precision_score(y_test,
    predictions_BRkNN, average='micro'))
print("Precision score (macro): ", metrics.precision_score(y_test,
    predictions_BRkNN, average='macro'))
```

Resultados en el conjunto de datos *scene*

```
_________________BRkNN Algorithm_____________________
Hamming Loss:  0.08540802213001383
Recall (micro):  0.7109375
Recall (macro):  0.7210525516982061
Accuracy:  0.6846473029045643
F1-score (micro) 0.7466666666666667
F1-score (macro) 0.7469859559273001
Precision score (micro):  0.7861771058315334
Precision score (macro):  0.8049417279982324
```

5.3 Análisis de los resultados en el conjunto de datos *scene*

BinaryRelevance:

- Fortalezas: Logra el menor Hamming Loss (0.066736) y una alta precisión (0.903797), lo que indica predicciones precisas de las etiquetas.
- Consideraciones: Un recuerdo moderado (0.697266) y un F1-Score (0.787211) sugieren que hay margen para mejorar la captura de todas las instancias positivas.

MultiOutputClassifier:

- Fortalezas: Demuestra un rendimiento competitivo en términos de Pérdida de Hamming (0.074343), precisión (0.948640) y F1-Score (0.744958).
- Consideraciones: Recuperación ligeramente menor (0.613281) en comparación con otros modelos, lo que podría afectar la predicción general de instancias positivas.

ClassifierChain:

- Fortalezas: Logra un buen rendimiento con una Pérdida de Hamming de 0.070885, alta precisión (0.932394) y F1-Score (0.763552).
- Consideraciones: La recuperación moderada (0.646484) indica potencial para capturar más instancias positivas.

LabelPowerset:

- Fortalezas: Rendimiento destacado en general, con la mayor recuperación (0.765625), F1-Score (0.787149) y precisión (0.809917).
- Consideraciones: Tiene un Hamming Loss más alto (0.073306), pero el rendimiento equilibrado en otras métricas lo convierte en una opción sólida.

RakellD:

- Consideraciones: Muestra un rendimiento relativamente inferior en la mayoría de las métricas, con un Hamming Loss más alto (0.144537) y una precisión más baja (0.572981).

MajorityVotingClassifier:

- Consideraciones: Muestra un rendimiento más bajo en todas las métricas, especialmente con el mayor Hamming Loss (0.210927) y una menor precisión (0.437179).

MLkNN:

- Consideraciones: Logra un equilibrio razonable con una pérdida de Hamming moderada (0.084025) y un rendimiento competitivo en precisión (0.778468) y F1-Score (0.755779).

MLTSVM:

- Consideraciones: Muestra un menor recall (0.427734) y precisión (0.642229), lo que afecta la predicción general de instancias positivas.

BRkNN:

- Consideraciones: Se desempeña bien con un conjunto equilibrado de métricas, incluyendo recall (0.710938), F1-Score (0.746667) y precision (0.786177).

Conclusión:

LabelPowerset: Destaca como el modelo más equilibrado y efectivo para el conjunto de datos multietiqueta "scene", ofreciendo un rendimiento competitivo en precisión, recall y F1-Score. Consideraciones para otros modelos: Algunos modelos, como BinaryRelevance y MLkNN, muestran fortalezas en métricas específicas pero podrían beneficiarse de mejoras en otras áreas para obtener un rendimiento más completo.

6 Ejercicio 6

La validación cruzada es crucial en Machine Learning para evaluar modelos y evitar sobreajuste. En el contexto multietiqueta, donde las instancias pueden tener múltiples etiquetas, esta técnica es invaluable. Distribuir equitativamente las instancias entre conjuntos de entrenamiento y prueba en cada iteración garantiza una evaluación exhaustiva, mejorando la capacidad del modelo para generalizar y manejar múltiples etiquetas.

Además, para nuestras experimentaciones, cargamos un conjunto de datos llamado 'scene'. A través de este conjunto, exploraremos las funciones clave: $KFold()$, $cross\_validate()$, $cross\_val\_score$ y $make\_scorer()$ para entender su aplicación en el contexto multietiqueta.

Listing 16: Cargando el conjunto de datos de *scene*

```
from skmultilearn.dataset import load_dataset
from itertools import combinations
import numpy as np
import pandas as pd
# scene
X, y, feature_names, label_names = load_dataset('scene', 'undivided')
X = pd.DataFrame(X.toarray())
y = y.toarray()
```

6.1 Función $KFold()$

La función $KFold()$ es una herramienta fundamental para dividir los datos en K particiones, permitiendo la realización de K iteraciones en el proceso de validación cruzada. Esto asegura que cada partición sirva tanto como conjunto de entrenamiento como de prueba, logrando una evaluación exhaustiva del modelo.

6.1.1 Ejemplo de uso

Listing 17: Multilabel Evaluation using $Kfold()$ with BRkNNaClassifier

```
from sklearn.model_selection import KFold
from skmultilearn.adapt import BRkNNaClassifier
import sklearn.metrics as metrics
kf = KFold(n_splits=5, shuffle=True, random_state=42)
# Initialize an array to store predictions
all_predictions_BRkNN = np.empty_like(y, dtype=int)
for train_index, test_index in kf.split(X,y):
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test =  y[train_index], y[test_index]
    # Initialize the classifier
    classifier = BRkNNaClassifier(k=3)
    # Train the classifier
    classifier.fit(X_train, y_train)
    # Predict on the test set
    predictions_BRkNN = classifier.predict(X_test).toarray()  # Convert predictions to array
    # Store the predictions for each label in the corresponding test indices
    for label_index in range(y.shape[1]):
        all_predictions_BRkNN[test_index, label_index] = predictions_BRkNN[:, label_index]
print("Hamming Loss: ", metrics.hamming_loss(y, all_predictions_BRkNN))
print("Recall (micro): ", metrics.recall_score(y, all_predictions_BRkNN, average ='micro' ))
```

```
print("Recall (macro): ", metrics.recall_score(y,
    all_predictions_BRkNN, average ='macro' ))
print("Accuracy: ",metrics.accuracy_score(y, all_predictions_BRkNN)
    )
print("F1-score (micro)", metrics.f1_score(y, all_predictions_BRkNN
    , average='micro'))
print("F1-score (macro)", metrics.f1_score(y, all_predictions_BRkNN
    , average='macro'))
print("Precision score (micro): ", metrics.precision_score(y,
    all_predictions_BRkNN, average='micro'))
print("Precision score (macro): ", metrics.precision_score(y,
    all_predictions_BRkNN, average='macro'))
```

```
scene:undivided - exists, not redownloading
Hamming Loss:  0.0970779670405761
Recall (micro):  0.6758220502901354
Recall (macro):  0.6855487276547443
Accuracy:  0.6393851267137516
F1-score (micro) 0.7136437908496732
F1-score (macro) 0.7233946466000575
Precision score (micro):  0.7559498052790999
Precision score (macro):  0.7907272479579843
```

6.2 Función *cross_validate*()

La función *cross_validate*() simplifica el proceso de entrenamiento y evaluación en la validación cruzada al devolver métricas de rendimiento y tiempos de entrenamiento en cada iteración. Facilita la obtención de información detallada sobre el comportamiento del modelo en diferentes particiones del conjunto de datos.

6.2.1 Ejemplo de uso

Listing 18: Multilabel Evaluation using *cross_validate* with BRkNNaClassifier

```
from sklearn.model_selection import cross_validate
from skmultilearn.adapt import BRkNNaClassifier
from sklearn.metrics import precision_recall_fscore_support,
    hamming_loss
# Define a custom scorer for micro and macro averaging
def custom_scorer(estimator, X, y):
    # Predict on the input data
    y_pred = estimator.predict(X)
    # Ensure that both y_true and y_pred are numpy arrays
    y_true = np.array(y)
    # Compute precision, recall, and f1-score for each label
    precision, recall, f1, _ = precision_recall_fscore_support(
   y_true, y_pred, average=None, zero_division=1)
```

```
    # Compute micro-averaged precision, recall, and f1-score
    micro_precision, micro_recall, micro_f1, _ =
   precision_recall_fscore_support(y_true, y_pred, average='micro',
   zero_division=1)
    # Compute macro-averaged precision, recall, and f1-score
    macro_precision, macro_recall, macro_f1, _ =
   precision_recall_fscore_support(y_true, y_pred, average='macro',
    zero_division=1)
    # Compute Hamming loss
    hamming_loss_value = hamming_loss(y_true, y_pred)
    return {
        'precision_micro': micro_precision,
        'recall_micro': micro_recall,
        'f1_micro': micro_f1,
        'precision_macro': macro_precision,
        'recall_macro': macro_recall,
        'f1_macro': macro_f1,
        'hamming_loss': hamming_loss_value
    }
# Initialize the classifier
classifier = BRkNNaClassifier(k=3)
# Specify the custom scorer in the cross_validate function
scoring = custom_scorer
cv_results = cross_validate(classifier, X, y, scoring=scoring, cv
   =5)
# Print the results
for metric, values in cv_results.items():
    print(f"{metric}: {np.mean(values)}")
```

```
scene:undivided - exists, not redownloading
fit_time: 0.008177995681762695
score_time: 0.16674981117248536
test_precision_micro: 0.6775012535002303
test_recall_micro: 0.5873561497404121
test_f1_micro: 0.6289322932517416
test_precision_macro: 0.4564394034917223
test_recall_macro: 0.6798462269547165
test_f1_macro: 0.38477734787641904
test_hamming_loss: 0.12365576556447926
```

6.3 Función *cross_val_score*

La función *cross_val_score* proporciona una forma simplificada de obtener las puntuaciones de rendimiento para cada iteración de la validación cruzada. Es especialmente útil cuando solo se necesitan las métricas de evaluación.

6.3.1 Ejemplo de uso

Listing 19: Multilabel Cross-Validation with *cross_val_score*

```
from sklearn.model_selection import cross_val_score
from skmultilearn.adapt import BRkNNaClassifier
from sklearn.metrics import precision_recall_fscore_support,
    hamming_loss
# define the custom scoring to use it in computing the metric of
    interest
def custom_scorer(estimator, X, y):
    # Predict on the input data
    y_pred = estimator.predict(X)
    # Ensure that both y_true and y_pred are numpy arrays
    y_true = np.array(y)
    # Compute precision, recall, and f1-score for each label
    precision, recall, f1, _ = precision_recall_fscore_support(
    y_true, y_pred, average=None, zero_division=1)
    # Compute micro-averaged precision, recall, and f1-score
    micro_precision, micro_recall, micro_f1, _ =
    precision_recall_fscore_support(y_true, y_pred, average='micro',
    zero_division=1)
    # Compute macro-averaged precision, recall, and f1-score
    macro_precision, macro_recall, macro_f1, _ =
    precision_recall_fscore_support(y_true, y_pred, average='macro',
     zero_division=1)
    # Compute Hamming loss
    hamming_loss_value = hamming_loss(y_true, y_pred)
    return  micro_precision
# using BRkNNaWrapper in cross_val_score
classifier = BRkNNaClassifier(k=3)
scoring = custom_scorer
cv_scores = cross_val_score(classifier, X, y, scoring=scoring, cv
    =5)
print(f"Precision (micro): {cv_scores.mean()}")
```

```
scene:undivided - exists, not redownloading
Precision (micro): 0.6775012535002303
```

6.4 Función *make_scorer*()

La función *make_scorer*() permite la creación de métricas personalizadas, lo que es útil cuando se desea evaluar el rendimiento del modelo utilizando métricas específicas de interés. Esto proporciona flexibilidad para adaptar la validación cruzada a las necesidades particulares del problema.

6.4.1 Ejemplo de uso

Listing 20: GridSearchCV with Custom Accuracy for MLkNN

```
from sklearn.metrics import make_scorer, accuracy_score
from skmultilearn.adapt import MLkNN
from sklearn.model_selection import GridSearchCV
# define the custom scoring metric 'accuracy'
def custom_accuracy_score(y_true, y_pred, **kwargs):
    return accuracy_score(y_true, y_pred, **kwargs)

# Use MLkNN in GridSearchCV with custom scoring
parameters = {'k': range(1, 3), 's': [0.5, 0.7, 1.0]}
scorer = make_scorer(custom_accuracy_score, greater_is_better=True)
clf = GridSearchCV(MLkNN(), parameters, scoring=scorer)
clf.fit(X, y)

print("GridSearchCV Best Parameters:", clf.best_params_)
print("GridSearchCV Best Score_accuracy:", clf.best_score_)
```

```
scene:undivided - exists, not redownloading
GridSearchCV Best Parameters: {'k': 1, 's': 0.5}
GridSearchCV Best Score_accuracy: 0.5708508380707551
```

Listing 21: Custom F1-score Scoring with MLkNN using Cross-Validation

```
from sklearn.metrics import make_scorer, f1_score
from skmultilearn.adapt import MLkNN
from sklearn.model_selection import cross_val_score
# define the custom f1-metric for scoring
def custom_f1_macro(y_true, y_pred, **kwargs):
    return f1_score(y_true, y_pred, average='macro', **kwargs)

# Use MLkNN in cross_val_score
classifier = MLkNN()
scorer = make_scorer(custom_f1_macro, greater_is_better=True)
cv_scores = cross_val_score(classifier, X, y, scoring=scorer, cv=5)
print("Cross-Validation Scores:", cv_scores)
print("Mean CV Score_f1-score (macro):", cv_scores.mean())
```

```
scene:undivided - exists, not redownloading
Cross-Validation Scores: [0.42612187 0.38574456 0.43256734
0.44493297 0.3439875 ]
Mean CV Score_f1-score (macro): 0.406670847512362
```

6.5 Conclusión: Aplicación de Validación Cruzada en el Contexto ML con Funciones Clave

La evaluación del rendimiento del MLkNN en el conjunto de datos multietiqueta *scene* a través de las cuatro funciones de validación cruzada revela hallazgos significativos. En

primer lugar, la aplicación de la metodología $KFold$ con 5 divisiones y mezcla de instancias demuestra una precisión superior en comparación con el uso de GridSearch con *make_scorer* y MLkNN.

Además, al examinar la puntuación F1-macro, se observa que la estrategia $KFold$ supera a *cross_val_score* con *make_scorer* y MLkNN. Esto resalta la eficacia de la validación cruzada tradicional en este escenario multietiqueta.

En términos de precisión micro, la metodología $KFold$ demuestra un rendimiento superior a *cross_validate* y *cross_val_score* con $cv = 5$ y MLkNN. Estos resultados sugieren que la distribución equitativa de instancias en conjuntos de entrenamiento y prueba, combinada con la aleatorización y la repetición del proceso, influye positivamente en la capacidad del modelo para generalizar y manejar múltiples etiquetas en el contexto *scene*.

7 Ejercicio 7

7.1 Breve descripción de las métricas

Accuracy: Accuracy es una métrica fundamental que mide la corrección general de un modelo de clasificación. Se calcula como la proporción de instancias predichas correctamente respecto al total de instancias.
Hamming Loss: Hamming loss es especialmente útil en escenarios de clasificación multietiqueta. Representa la fracción de etiquetas incorrectamente predichas con respecto al número total de etiquetas. Cuanto menor sea la pérdida de Hamming, mejor será el rendimiento del modelo.
Precision: Precision es una medida de la capacidad del modelo para identificar correctamente las instancias positivas entre las instancias que predijo como positivas. Se calcula como la razón entre los verdaderos positivos y la suma de los verdaderos positivos y los falsos positivos.
Recall: Recall, También conocida como sensibilidad o tasa de verdaderos positivos, evalúa la capacidad del modelo para capturar todas las instancias positivas. Se calcula como la proporción de verdaderos positivos respecto a la suma de verdaderos positivos y falsos negativos.
F1-Score: F1-score es la media harmónica de la precisión y la sensibilidad. Proporciona una medida equilibrada entre precisión y sensibilidad. Una puntuación F1 alta indica un buen rendimiento tanto en precisión como en sensibilidad.

7.2 Información proporcionada por cada métrica

Accuracy: Proporciona una visión general de qué tan bien se desempeña el modelo en todas las clases. Sin embargo, puede no ser adecuado para conjuntos de datos desequilibrados.
Hamming Loss: Indica la fracción de etiquetas incorrectamente predichas, proporcionando información sobre el rendimiento del modelo en clasificación multietiqueta.

Precision: Destaca la precisión de las predicciones positivas, ayudando a evaluar la capacidad del modelo para evitar falsos positivos.
Recall: Mide la capacidad del modelo para capturar todas las instancias positivas, indicando su rendimiento en evitar falsos negativos.
F1-Score: Encuentra un equilibrio entre precisión y recall, proporcionando una evaluación integral del rendimiento del modelo.

7.3 Investigando el Uso

La aplicación de métricas de clasificación, que incluyen precisión, pérdida de Hamming, precisión, recuperación y puntuación F1, es crucial para evaluar el rendimiento de los modelos de clasificación. El módulo de métricas de clasificación en bibliotecas como scikit-learn proporciona un conjunto completo de herramientas para este propósito.

7.3.1 Módulo de métricas de clasificación de Scikit-Learn:

Scikit-learn, una popular biblioteca de aprendizaje automático en PythonPedregosa et al. (2011), ofrece un módulo dedicado para métricas de clasificación. Este módulo proporciona diversas funciones para calcular diferentes métricas que evalúan el rendimiento de modelos de clasificación. Por ejemplo, *accuracy_score* calcula la precisión, *precision_score* computa la precisión, *recall_score* calcula la recuperación, y *f1_score* computa el F1-Score.
Ejemplo de uso

Listing 22: Ejemplo de uso del Módulo de métricas

```
from skmultilearn.problem_transform import BinaryRelevance
from sklearn.svm import SVC
# initialize Binary Relevance multi-label classifier with an SVM
    classifier
classifier = BinaryRelevance( classifier = SVC(), require_dense = [
    False, True])
classifier.fit(X_train, y_train) # train
predictions_BinaryRelevance = classifier.predict(X_test) # predict
import sklearn.metrics as metrics
print("Hamming Loss: ", metrics.hamming_loss(y_test,
    predictions_BinaryRelevance))
"""f average='micro', it computes the recall globally by counting
    the total true positives, false negatives, and false positives.
If average='macro', it computes the recall for each label
    independently and then takes the unweighted average."""
print("Recall (micro): ", metrics.recall_score(y_test,
    predictions_BinaryRelevance, average ='micro' ))
print("Recall (macro): ", metrics.recall_score(y_test,
    predictions_BinaryRelevance, average ='macro' ))
print("Accuracy: ",metrics.accuracy_score(y_test,
    predictions_BinaryRelevance))
print("F1-score (micro)", metrics.f1_score(y_test,
    predictions_BinaryRelevance, average='micro'))
```

```
print("F1-score (macro)", metrics.f1_score(y_test,
    predictions_BinaryRelevance, average='macro'))
print("Precision score (micro): ", metrics.precision_score(y_test,
    predictions_BinaryRelevance, average='micro'))
print("Precision score (macro): ", metrics.precision_score(y_test,
    predictions_BinaryRelevance, average='macro'))
```

Aquí, mostramos todas las métricas para el modelo *BinaryRelevance* entrenado con el conjunto de datos *scene*:

```
Hamming Loss:  0.06673582295988935
Recall (micro):  0.697265625
Recall (macro):  0.701266581400048
Accuracy:  0.6680497925311203
F1-score (micro) 0.7872105843439912
F1-score (macro) 0.7812045513402271
Precision score (micro):  0.9037974683544304
Precision score (macro):  0.8940930533172172
```

7.3.2 Función *classification_report*():

Esta función genera un informe detallado que incluye precisión, recall, F1-score y soporte para cada clase. Proporciona una visión integral del rendimiento del modelo en diferentes clases.

Ejemplo de uso en el context de multilabel

En el contexto de la clasificación multietiqueta, el informe de clasificación *classification_report* Pedregosa et al. (2011) proporciona métricas adaptadas para manejar múltiples etiquetas para cada instancia.

Listing 23: Ejemplo de uso de *classification_report*()

```
from skmultilearn.problem_transform import BinaryRelevance
from sklearn.svm import SVC
# initialize Binary Relevance multi-label classifier with an SVM
    classifier
classifier = BinaryRelevance(
    classifier = SVC(),
    require_dense = [False, True])
# train
classifier.fit(X_train, y_train)
# predict
predictions_BinaryRelevance = classifier.predict(X_test)
from sklearn.metrics import classification_report

report = classification_report(y_test, predictions_BinaryRelevance)
print(report)
```

Aquí está el informe de clasificación para el modelo *BinaryRelevance* entrenado con el conjunto de datos *scene*:

	precision	recall	f1-score	support
0	0.92	0.65	0.76	91
1	0.93	0.84	0.88	81
2	0.96	0.87	0.92	93
3	0.94	0.81	0.87	79
4	0.79	0.44	0.57	99
5	0.82	0.59	0.69	69
micro avg	0.90	0.70	0.79	512
macro avg	0.89	0.70	0.78	512
weighted avg	0.89	0.70	0.78	512
samples avg	0.72	0.71	0.71	512

Precision, Recall, F1-Score, Support Para cada clase:

- Para cada clase (0, 1, 2, 3, 4, 5), el informe muestra precisión, recall y F1-score.
- Precisión: El número de instancias verdaderamente positivas dividido por el número total de instancias predichas como positivas.
- Recall: El número de instancias verdaderamente positivas dividido por el número total de instancias positivas reales.
- F1-Score: La media armónica de precisión y recall, proporcionando un equilibrio entre ambos.
- Soporte: El número de instancias reales para cada clase.

Estas métricas brindan una comprensión de qué tan bien está funcionando el clasificador para cada etiqueta individual.
Micro Promedio:

- La fila *micro avg* proporciona las métricas calculadas globalmente considerando cada elemento de la matriz indicadora de etiquetas como una instancia individual.
- Precisión Micro: El número total de verdaderos positivos dividido por el número total de instancias predichas como positivas.
- Recall Micro: El número total de verdaderos positivos dividido por el número total de instancias positivas reales.
- Puntuación F1 Micro: La media armónica de la precisión micro y el recall micro.
- Soporte Micro: El número total de instancias.

La micro-media es útil cuando queremos evaluar el rendimiento general en todas las etiquetas sin tener en cuenta las etiquetas individuales.
Promedio Macro:

- La fila *macroavg* proporciona las métricas calculadas de manera independiente para cada etiqueta y luego se promedian sin tener en cuenta el desequilibrio de clases.
- Precisión Macro: La precisión promedio en todas las clases.
- Recall Macro: El recall promedio en todas las clases.
- F1-Score Macro: El F1-Score promedio en todas las clases.
- Soporte Macro: Ignorado en la clasificación multietiqueta.

La macro-media es útil cuando queremos evaluar el rendimiento del modelo en todas las etiquetas tratando cada etiqueta por igual.
Promedio Weighted:

- La fila *weightedavg* proporciona las métricas calculadas considerando el número de muestras para cada etiqueta.
- Precisión ponderada: Es el promedio ponderado de la precisión en todas las clases, donde los pesos son el soporte para cada clase.
- Recall ponderado: Es el promedio ponderado del recall en todas las clases.
- Puntuación F1 ponderada: Es el promedio ponderado de la puntuación F1 en todas las clases.
- Soporte ponderado: No se tiene en cuenta en la clasificación multietiqueta.

The weighted average is useful when we want to account for class imbalance.
Promedio de Muestras:

- La fila *samplesavg* proporciona el promedio sobre todas las muestras (instancias).
- Precisión de las muestras: La precisión promedio en todas las instancias.
- Recall de las muestras: El recall promedio en todas las instancias.
- Puntuación F1 de las muestras: La puntuación F1 promedio en todas las instancias.
- Soporte de las muestras: El número total de instancias.

El promedio de las muestras proporciona una evaluación general al considerar cada instancia de manera igual, independientemente de las etiquetas.

El $classification\_report$ para la clasificación multietiqueta ofrece una evaluación integral al considerar métricas tanto a nivel de etiquetas como de instancias, lo que le permite evaluar el rendimiento del modelo desde diferentes perspectivas.

8 Ejercicio 8

Utilizamos los siguientes conjuntos de datos: bibtex, genbase, Corel5k, medical, enron. Para los algoritmos, seleccionamos dos métodos de transformación: RakelD y ClassifierChain, y un método de adaptación: MLkNN. Mostramos el código Python realizado para este paso.

8.1 Implementación de código: Evaluación de clasificación multietiqueta con varios métodos

El script en Python (cl-cv.py) muestra la implementación de los tres métodos de clasificación multietiqueta. El código demostrará cómo evaluar estos métodos utilizando validación cruzada K-fold en los cinco conjuntos de datos seleccionados.

Listing 24: Código para la Evaluación de Clasificación Multietiqueta utilizando MLkNN con Validación Cruzada de 5 Pliegues.

```
import skmultilearn
import pandas as pd
import numpy as np
from skmultilearn.dataset import load_dataset
# here are the datasets: [ 'bibtex', 'enron',  'genbase', 'medical', 'yeast','rcv1subset4', 'delicious']
X_bibtex, y_bibtex, feature_names, label_names = load_dataset('bibtex', 'undivided')
# 'bibtex'
X_bibtex = pd.DataFrame(X_bibtex.toarray())
y_bibtex = y_bibtex.toarray()
# X and y are  features and labels
""" Train MLkNN """
from skmultilearn.adapt import MLkNN
import time
from sklearn.model_selection import KFold

# bibtex
print ("start training MLkNN on bibtex")
start_time = time.time()    # Registra el tiempo de inicio
kf = KFold(n_splits=5, shuffle=True, random_state=42)
# Initialize an array to store predictions
predictions_MLkNN_bibtex = np.empty_like(y_bibtex, dtype=int)
for train_index, test_index in kf.split(X_bibtex,y_bibtex):
    X_train, X_test = X_bibtex.iloc[train_index], X_bibtex.iloc[test_index]
    y_train, y_test =  y_bibtex[train_index], y_bibtex[test_index]
    # Initialize the classifier
    classifier = MLkNN(k=3)
    # Train the classifier
    classifier.fit(X_train, y_train)
    # Predict on the test set
```

```
    predictions = classifier.predict(X_test).toarray()  # Convert predictions to array
    # Store the predictions for each label in the corresponding test indices
    for label_index in range(y_bibtex.shape[1]):
        predictions_MLkNN_bibtex[test_index, label_index] = predictions[:, label_index]
end_time = time.time()  # Registra el tiempo de finalizaci n
execution_time_bibtex_MLkNN = end_time - start_time    # Calcula la diferencia para obtener el tiempo de ejecuci n en segundos

# El mismo c digo ha sido utilizado para entrenar modelos en los otros conjuntos de datos.

import sklearn.metrics as metrics
# Assuming you have multiple models and their predictions stored in a list or dictionary
models = ["MLkNN 'bibtex'", "MLkNN 'genbase'","MLkNN 'enron","MLkNN 'medical'","MLkNN 'yeast'", "MLkNN 'Corel5k'"]  # model names

prediction_dict = {"MLkNN 'bibtex'":predictions_MLkNN_bibtex,
                   "MLkNN 'genbase'": predictions_MLkNN_genbase,
                   "MLkNN 'enron": predictions_MLkNN_enron,
                   "MLkNN 'medical'":predictions_MLkNN_medical,
                   "MLkNN 'yeast'": predictions_MLkNN_yeast,
                   "MLkNN 'Corel5k'": predictions_MLkNN_Corel5k}
# a DataFrame to store the metrics
metrics_MLkNN= pd.DataFrame(columns=[
    'Hamming Loss', 'Recall (micro)', 'Recall (macro)',
    'Accuracy', 'F1-score (micro)', 'F1-score (macro)',
    'Precision (micro)', 'Precision (macro)', 'execution time (s)'
    ])
execution_time_g = {"MLkNN 'bibtex'":execution_time_bibtex_MLkNN,
                    "MLkNN 'genbase'": execution_time_genbase_MLkNN,
                    "MLkNN 'enron": execution_time_enron_MLkNN,
                    "MLkNN 'medical'":execution_time_medical_MLkNN,
                    "MLkNN 'yeast'": execution_time_yeast_MLkNN,
                    "MLkNN 'Corel5k'": execution_time_Corel5k_MLkNN}
test_sets = {"MLkNN 'bibtex'":y_bibtex,
                   "MLkNN 'genbase'": y_genbase,
                   "MLkNN 'enron": y_enron,
                   "MLkNN 'medical'": y_medical,
                   "MLkNN 'yeast'": y_yeast,
                   "MLkNN 'Corel5k'": y_Corel5k}
# metrics for each model
for model in models:
    hamming_loss = metrics.hamming_loss(test_sets[model], prediction_dict[model])
```

```
    recall_micro = metrics.recall_score(test_sets[model],
   prediction_dict[model], average='micro', zero_division=1)
    recall_macro = metrics.recall_score(test_sets[model],
   prediction_dict[model], average='macro', zero_division=1)
    accuracy = metrics.accuracy_score(test_sets[model],
   prediction_dict[model])
    f1_micro = metrics.f1_score(test_sets[model], prediction_dict[
   model], average='micro',zero_division=1)
    f1_macro = metrics.f1_score(test_sets[model], prediction_dict[
   model], average='macro', zero_division=1)
    precision_micro = metrics.precision_score(test_sets[model],
   prediction_dict[model], average='micro', zero_division=1)
    precision_macro = metrics.precision_score(test_sets[model],
   prediction_dict[model], average='macro', zero_division=1)
    execution_time =execution_time_g[model]
    metrics_MLkNN.loc[model] = [
        hamming_loss, recall_micro, recall_macro,
        accuracy, f1_micro, f1_macro,
        precision_micro, precision_macro, execution_time]

```

Listing 25: Código para la Evaluación de Clasificación Multietiqueta utilizando RakelD y ChainClassifier con Validación Cruzada de 5 Pliegues.

```
# bibtex
import time
from sklearn.naive_bayes import GaussianNB
from skmultilearn.ensemble import RakelD
from sklearn.model_selection import KFold
import sklearn.metrics as metrics
# Registra el tiempo de inicio
start_time = time.time()
kf = KFold(n_splits=5, shuffle=True, random_state=42)
# Initialize an array to store predictions
predictions_bibtex_RakelD_Kf = np.empty_like(y_bibtex, dtype=int)
for train_index, test_index in kf.split(X_bibtex,y_bibtex):
    X_train, X_test = X_bibtex.iloc[train_index], X_bibtex.iloc[
   test_index]
    y_train, y_test =  y_bibtex[train_index], y_bibtex[test_index]
    # Initialize the classifier
    classifier = RakelD(
            base_classifier=GaussianNB(),
            base_classifier_require_dense=[True, True],
            labelset_size=4 )
    # Train the classifier
    classifier.fit(X_train, y_train)
    # Predict on the test set
    predictions_bibtex_Kf = classifier.predict(X_test).toarray()  #
    Convert predictions to array
```

```
    # Store the predictions for each label in the corresponding test indices
    for label_index in range(y_bibtex.shape[1]):
        predictions_bibtex_RakelD_Kf[test_index, label_index] = predictions_bibtex_Kf[:, label_index]
# Registra el tiempo de finalizaci n
end_time = time.time()
# Calcula la diferencia para obtener el tiempo de ejecuci n en segundos
execution_time_bibtex_RakelD_Kf = end_time - start_time
# using classifier chains
import time
from skmultilearn.problem_transform import ClassifierChain
from sklearn.ensemble import RandomForestClassifier
# Registra el tiempo de inicio
start_time = time.time()
kf = KFold(n_splits=5, shuffle=True, random_state=42)
# Initialize an array to store predictions
predictions_bibtex_ClassifierChain_Kf = np.empty_like(y_bibtex, dtype=int)
for train_index, test_index in kf.split(X_bibtex,y_bibtex):
    X_train, X_test = X_bibtex.iloc[train_index], X_bibtex.iloc[test_index]
    y_train, y_test =  y_bibtex[train_index], y_bibtex[test_index]
    # initialize classifier chains multi-label classifier
    classifier = ClassifierChain(RandomForestClassifier(n_estimators=100, random_state=42))
    # Train the classifier
    classifier.fit(X_train, y_train)
    # Predict on the test set
    predictions_bibtex_Kf = classifier.predict(X_test).toarray()  # Convert predictions to array
    # Store the predictions for each label in the corresponding test indices
    for label_index in range(y_bibtex.shape[1]):
        predictions_bibtex_ClassifierChain_Kf[test_index, label_index] = predictions_bibtex_Kf[:, label_index]
# Registra el tiempo de finalizaci n
end_time = time.time()
# Calcula la diferencia para obtener el tiempo de ejecuci n en segundos
execution_time_bibtex_ClassifierChain_Kf = end_time - start_time

# Evaluation of built models: ChainClassifier and RakelD
import pandas as pd
import sklearn.metrics as metrics
# multiple models and their predictions stored in a list or dictionary
```

```
models = ["RakelD-KFlold 'bibtex'", "ClassifierChain-KFold 'bibtex'"]  # model names
prediction_dict = {"RakelD-KFlold 'bibtex'": predictions_bibtex_RakelD_Kf,
                   "ClassifierChain-KFold 'bibtex'": predictions_bibtex_ClassifierChain_Kf,}
# a DataFrame to store the metrics
metrics_bibtex = pd.DataFrame(columns=[
    'Hamming Loss', 'Recall (micro)', 'Recall (macro)',
    'Accuracy', 'F1-score (micro)', 'F1-score (macro)',
    'Precision (micro)', 'Precision (macro)', "execution time (s)"
])
execution_time_m = {"RakelD-KFlold 'bibtex'": execution_time_bibtex_RakelD_Kf,
                    "ClassifierChain-KFold 'bibtex'": execution_time_bibtex_ClassifierChain_Kf}
# metrics for each model
for model in models:
    hamming_loss = metrics.hamming_loss(y_bibtex, prediction_dict[model])
    recall_micro = metrics.recall_score(y_bibtex, prediction_dict[model], average='micro')
    recall_macro = metrics.recall_score(y_bibtex, prediction_dict[model], average='macro')
    accuracy = metrics.accuracy_score(y_bibtex, prediction_dict[model])
    f1_micro = metrics.f1_score(y_bibtex, prediction_dict[model], average='micro')
    f1_macro = metrics.f1_score(y_bibtex, prediction_dict[model], average='macro')
    precision_micro = metrics.precision_score(y_bibtex, prediction_dict[model], average='micro', zero_division=1)
    precision_macro = metrics.precision_score(y_bibtex, prediction_dict[model], average='macro', zero_division=1)
    execution_time =execution_time_m[model]
    metrics_bibtex.loc[model] = [
        hamming_loss, recall_micro, recall_macro,
        accuracy, f1_micro, f1_macro,
        precision_micro, precision_macro, execution_time
    ]
metrics_bibtex
El mismo c digo ha sido utilizado para entrenar modelos en los otros conjuntos de datos.
```

8.2 Resultados e Interpretación: Métricas Validadas Cruzadamente para la Clasificación Multietiqueta

8.2.1 Resultados en el conjunto de datos *bibtex*

Algorithm	Hamming Loss	Recall (micro)	Recall (macro)	Accuracy	F1-score (micro)	F1-score (macro)	Precision (micro)	Precision (macro)	Execution Time (s)
MLkNN	0.016964	0.208535	0.131990	0.073158	0.270820	0.163107	0.386155	0.288927	121.055111
RakelD	0.038027	0.245749	0.140156	0.050845	0.163355	0.101468	0.122337	0.117773	102.763185
ClassifierChain	0.013088	0.151109	0.060604	0.112373	0.258612	0.081502	0.896160	0.659941	1388.489277

Figura 1: Resultados en el conjunto de datos *bibtex*

En la clasificación multietiqueta en *bibtex* con validación k-fold 1, MLkNN logra un rendimiento equilibrado con una pérdida de Hamming moderada y una recordación razonable. RakelD-KFold sacrifica precisión para mejorar la recordación, mientras que ClassifierChain-KFold optimiza la precisión a expensas de una recordación moderada.

8.2.2 Resultados en el conjunto de datos *yeast*

Algorithm	Hamming Loss	Recall (micro)	Recall (macro)	Accuracy	F1-score (micro)	F1-score (macro)	Precision (micro)	Precision (macro)	Execution Time (s)
RakelD	0.287783	0.598281	0.491573	0.092677	0.557203	0.442358	0.521402	0.416402	1.200228
ClassifierChain	0.192801	0.576018	0.347404	0.217625	0.643925	0.382425	0.729984	0.764834	135.028718
MLkNN	0.216177	0.605019	0.417145	0.201903	0.628812	0.439801	0.654553	0.563454	5.492321

Figura 2: Resultados de la clasificación multietiqueta con evaluación de validación cruzada en el conjunto de datos *yeast*

La figura 2 muestra los resultados de clasificación en los datos de *enron*. RakelD muestra un mayor Hamming Loss (0.28), indicando un número significativo de predicciones incorrectas de etiquetas. Exhibe buenas tasas de recuperación (tanto micro como macro), capturando instancias relevantes de manera efectiva, pero a costa de una precisión más baja.

El método ClassifierChain logra un Hamming Loss más bajo con un equilibrio entre recall y precisión, destacando la precisión general. MLkNN encuentra un punto intermedio entre ambos, ofreciendo un compromiso moderado entre precisión, recall y exactitud.

8.2.3 Resultados en el conjunto de datos *enron*

En la clasificación multietiqueta en el conjunto de datos *enron* 3, RakelD enfrenta desafíos con un alto Hamming Loss y una baja precisión, mostrando un recall moderado con baja precisión. ClassifierChain logra un rendimiento equilibrado con un Hamming Loss más bajo y una mayor precisión, mientras que MLkNN encuentra un equilibrio intermedio con un recall moderado y una precisión equilibrada.

Algorithm	Hamming Loss	Recall (micro)	Recall (macro)	Accuracy	F1-score (micro)	F1-score (macro)	Precision (micro)	Precision (macro)	Execution Time (s)
RakelD	0.151642	0.576000	0.274324	0.000000	0.326257	0.154439	0.227582	0.185185	3.822263
ClassifierChain	0.046427	0.495304	0.160167	0.131610	0.576285	0.199843	0.688921	0.390546	116.787480
MLkNN	0.060694	0.415304	0.160749	0.103995	0.465906	0.180091	0.530549	0.346595	10.505922

Figura 3: Resultados de la clasificación multietiqueta con evaluación de validación cruzada en el conjunto de datos *enron*

8.2.4 Resultados en el conjunto de datos *Corel5k*

Algorithm	Hamming Loss	Recall (micro)	Recall (macro)	Accuracy	F1-score (micro)	F1-score (macro)	Precision (micro)	Precision (macro)	Execution Time (s)
RakelD	0.036496	0.435889	0.086844	0.0008	0.183636	0.044291	0.116321	0.208425	63.627252
ClassifierChain	0.009355	0.025667	0.004883	0.0050	0.049138	0.008429	0.574333	0.937173	1101.677436
MLkNN	0.012338	0.093072	0.023859	0.0056	0.124402	0.029683	0.187529	0.360241	164.981008

Figura 4: Resultados de la clasificación multietiqueta con evaluación de validación cruzada en el conjunto de datos *Corel5k*

Como se observa en los resultados en el conjunto de datos *Corel5k* 4, RakelD muestra una precisión más baja pero un buen recall, lo que indica su efectividad para capturar instancias relevantes. ClassifierChain prioriza la precisión, lo que resulta en un recall más bajo. MLkNN logra un rendimiento equilibrado en todas las métricas.

8.2.5 Resultados en el conjunto de datos *genbase*

Algorithm	Hamming Loss	Recall (micro)	Recall (macro)	Accuracy	F1-score (micro)	F1-score (macro)	Precision (micro)	Precision (macro)	Execution Time (s)
RakelD	0.002741	0.972256	0.738697	0.933535	0.970500	0.741302	0.968750	0.970617	1.272757
ClassifierChain	0.001399	0.971049	0.741713	0.963746	0.984709	0.763947	0.998759	0.998911	29.745906
MLkNN	0.046604	0.302774	0.134023	0.323263	0.376030	0.162888	0.496047	0.520728	4.449108

Figura 5: Resultados de la clasificación multietiqueta con evaluación mediante validación cruzada en el conjunto de datos *genbase*

En la clasificación multietiqueta en el conjunto de datos "genbase-5, RakelD y ClassifierChain demuestran un rendimiento excepcional con un Hamming Loss mínimo, alta precisión, recall (tanto micro como macro), precisión (tanto micro como macro) y puntuaciones F1. MLkNN, aunque eficiente, muestra un rendimiento general más bajo en la captura de instancias relevantes.

8.3 Conclusión

Analizando detenidamente cada método, 'ClassifierChain'muestra un rendimiento sobresaliente en términos de pérdida de Hamming, precisión y recall en comparación con 'Ra-

kelD'y 'MLkNN'. Este patrón se mantiene en diferentes conjuntos de datos como *bibtex* y *genbase*. Por otro lado, 'RakelD'presenta buenos resultados en algunos conjuntos, pero su tiempo de ejecución es notablemente más rápido que 'ClassifierChain'. 'MLkNN', aunque competitivo, destaca menos en pérdida de Hamming y precisión, mostrando un rendimiento variable en diferentes conjuntos.

9 Ejercicio 9

9.1 A-Evaluación Comparativa de Métodos en Clasificación Multietiqueta

9.1.1 Rendimiento Global de Métodos de Clasificación

En la evaluación de los métodos de clasificación multietiqueta sobre diferentes conjuntos de datos, se observan tendencias y diferencias notables. En términos generales, el método 'ClassifierChain'muestra un rendimiento superior, exhibiendo menor pérdida de Hamming, mayor precisión y recall tanto en la media micro como macro, así como mayor velocidad de ejecución en comparación con *RakelD* y 'MLkNN'. Este rendimiento se destaca en conjuntos de datos como *bibtex* y *genbase*, donde la mejora es más significativa.

9.1.2 Impacto de la Elección del Clasificador Base en Métodos de Transformación

Caso del RakelD

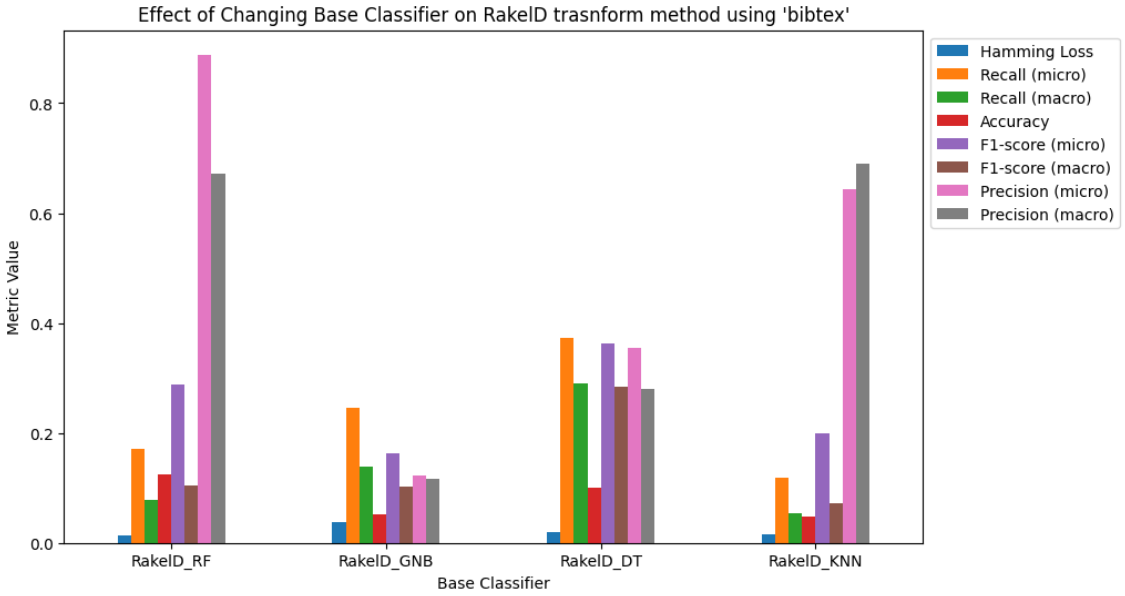

Figura 6: Rendimiento de RakelD según el clasificador base en el conjunto de datos *bibtex*

En el conjunto de datos *bibtex* (figura 6), *RakelD_RF* muestra un rendimiento equilibrado. *RakelD_GNB* y *RakelD_DT* presentan un rendimiento moderado con recall y precisión balanceados. *RakelD_KNN* muestra un bajo recall y precisión pero una precisión más alta.

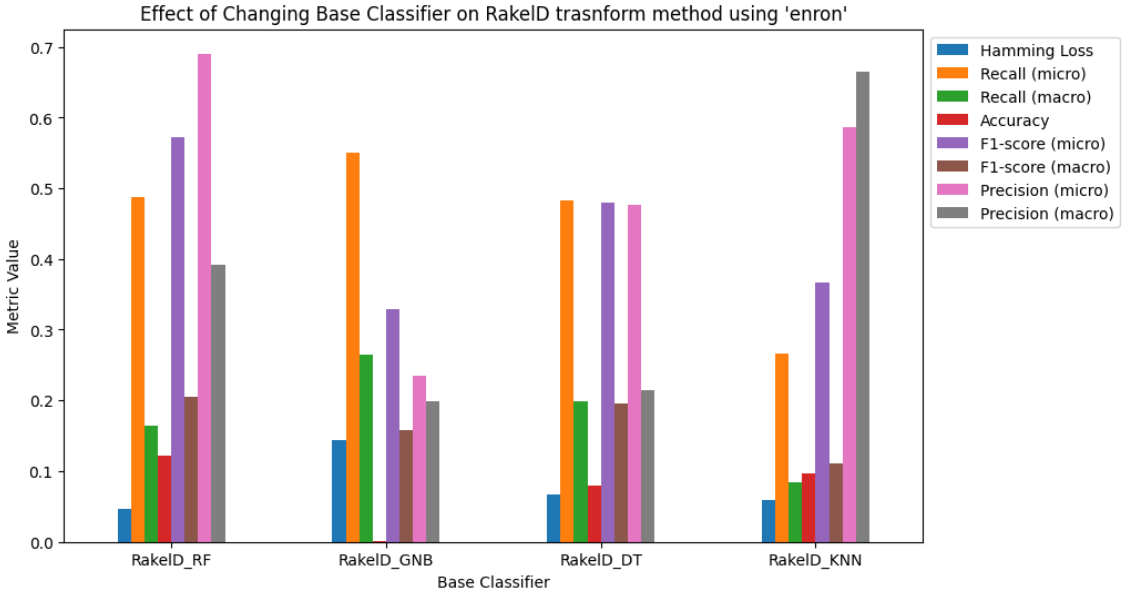

Figura 7: Rendimiento de RakelD según el clasificador base en el conjunto de datos *enron*

En el conjunto de datos *enron* (figura 7), *RakelD_RF* presenta un rendimiento equilibrado. *RakelD_GNB* se queda corto en precisión y métricas macro promedio. *RakelD_DT* y *RakelD_KNN* ofrecen resultados comparables y moderados.

En el conjunto de datos *yeast* (figura 8), *RakelD_RF* destaca con un rendimiento superior. *RakelD_KNN* le sigue de cerca, manteniendo un buen equilibrio en todas las métricas. *RakelD_DT* y *RakelD_GNB* muestran un rendimiento comparativamente más débil.

En *Corel5k* (figura 9), *RakelD_RF* muestra un bajo Hamming Loss y una precisión moderada. *RakelD_GNB* enfrenta desafíos con un alto Hamming Loss y precisión limitada. *RakelD_DT* logra un equilibrio entre recall y precisión. *RakelD_KNN* presenta un recall y precisión bajos, pero con un Hamming Loss relativamente bajo.

En genbase (figura 10), $RakelD_RF$ alcanza un rendimiento excepcional. $RakelD_DT$ se desempeña de manera sólida, mientras que $RakelD_KNN$ mantiene un equilibrio, y $RakelD_GNB$ muestra resultados competitivos en general.

Caso del ClassifierChain

La eficacia de ClassifierChain en *bibtex* 11 depende de la elección del clasificador base. RandomForest (RF) destaca al proporcionar un Hamming Loss mínimo y un rendimiento superior en Recall, Accuracy, F1-score y Precision en comparación con GNB, DT y KNN.

En el análisis del conjunto de datos Enron (figura 12), $ClassifierChain_RF$ destaca

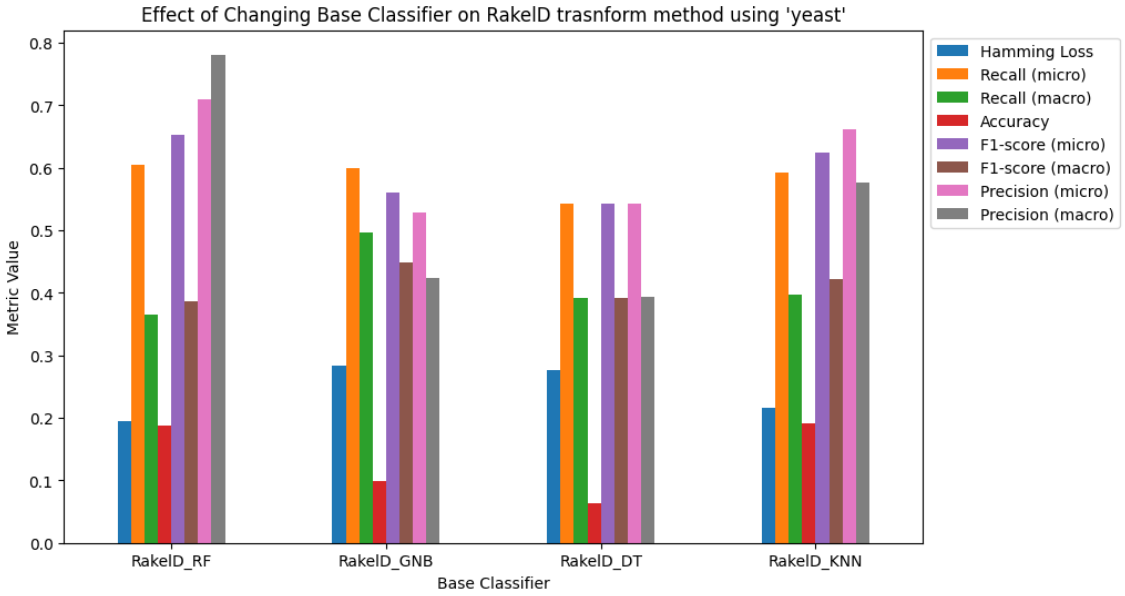

Figura 8: Rendimiento de RakelD según el Clasificador Base en el conjunto de datos *yeast*

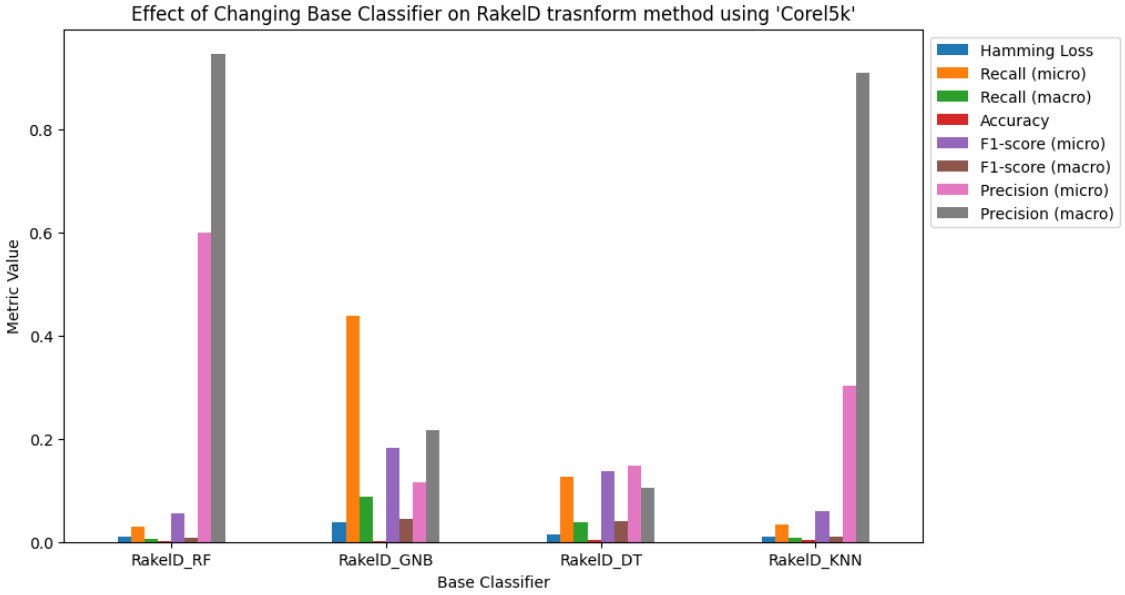

Figura 9: Rendimiento de RakelD según el clasificador base en el conjunto de datos *Corel5k*

como el más efectivo. Le sigue $ClassifierChain_DT$ con un rendimiento general sólido. Por otro lado, $ClassifierChain_KNN$ y $ClassifierChain_GNB$ muestran resultados comparativamente más débiles.

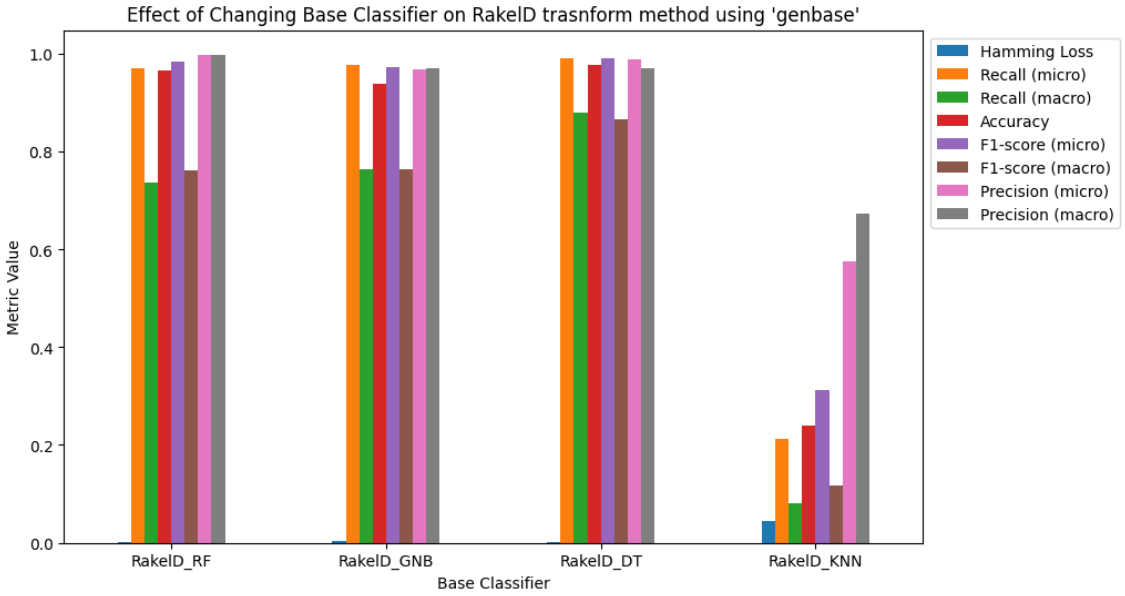

Figura 10: Rendimiento de RakelD según el clasificador base en el conjunto de datos *genbase*

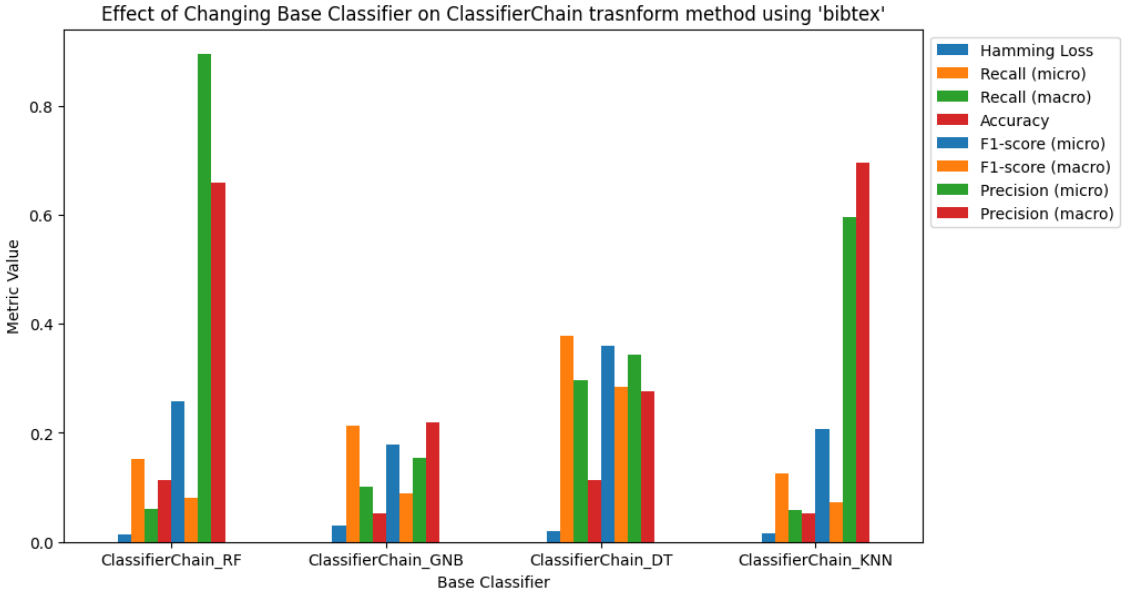

Figura 11: Rendimiento de ClassifierChain basado en el Clasificador Base en el conjunto de datos *bibtex*

En el conjunto de datos yeast (figura 13), RandomForest (RF) supera a todos los

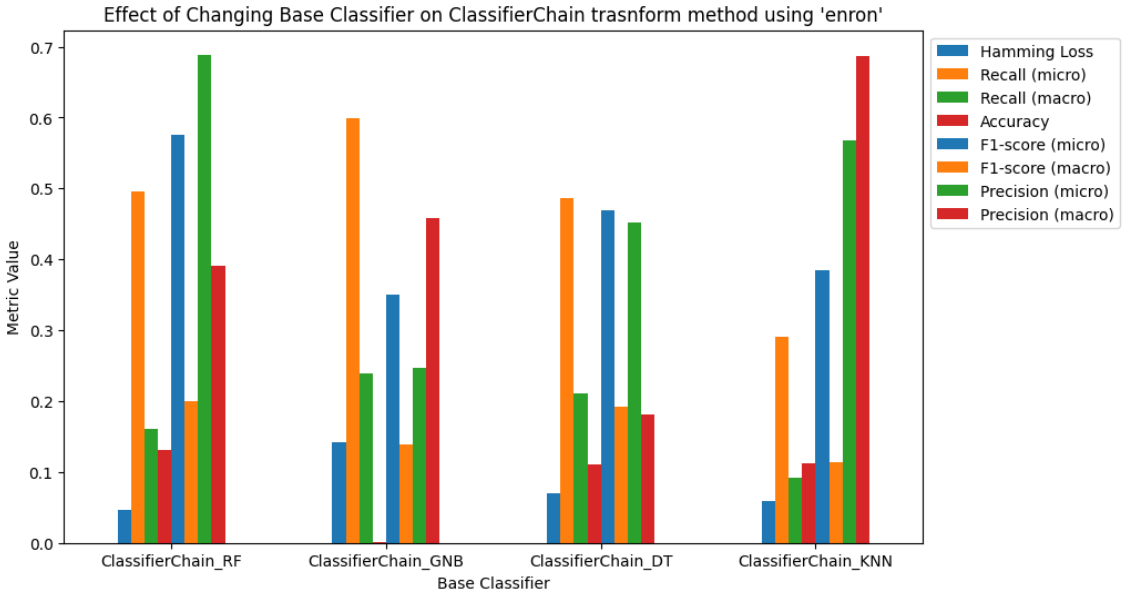

Figura 12: Desempeño de ClassifierChain según el clasificador base en el conjunto de datos *enron*

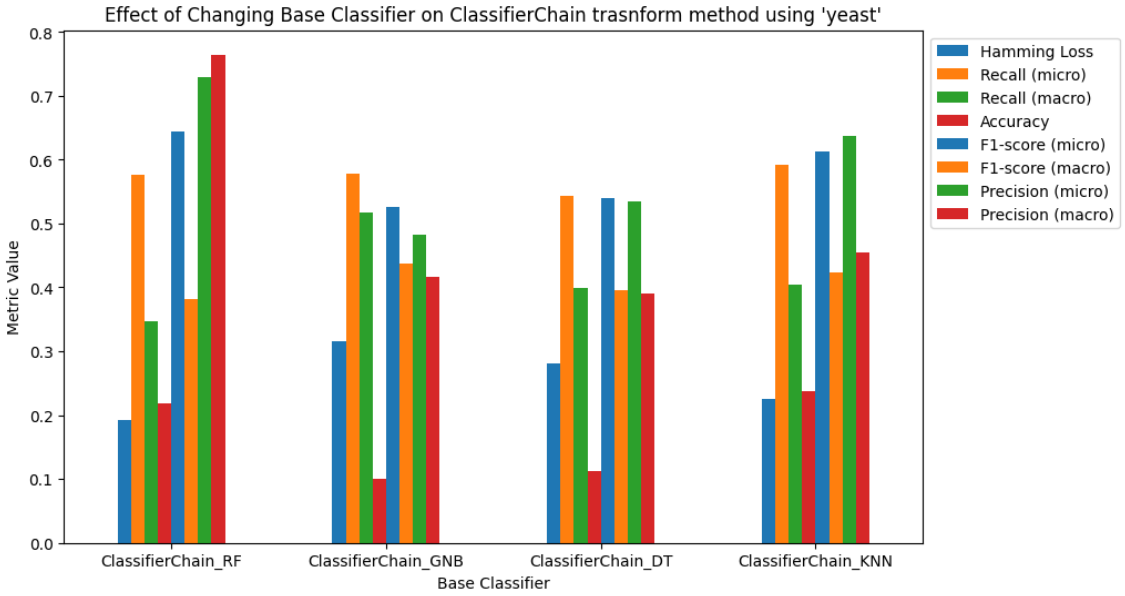

Figura 13: Rendimiento de ClassifierChain basado en el clasificador base en el conjunto de datos de *yeast*.

demás clasificadores base en ClassifierChain. Naive Bayes Gaussiano (GNB) está equili-

brado. Decision Tree (DT) demuestra un rendimiento moderado, mientras que k-Nearest Neighbors (KNN) muestra un rendimiento más bajo.

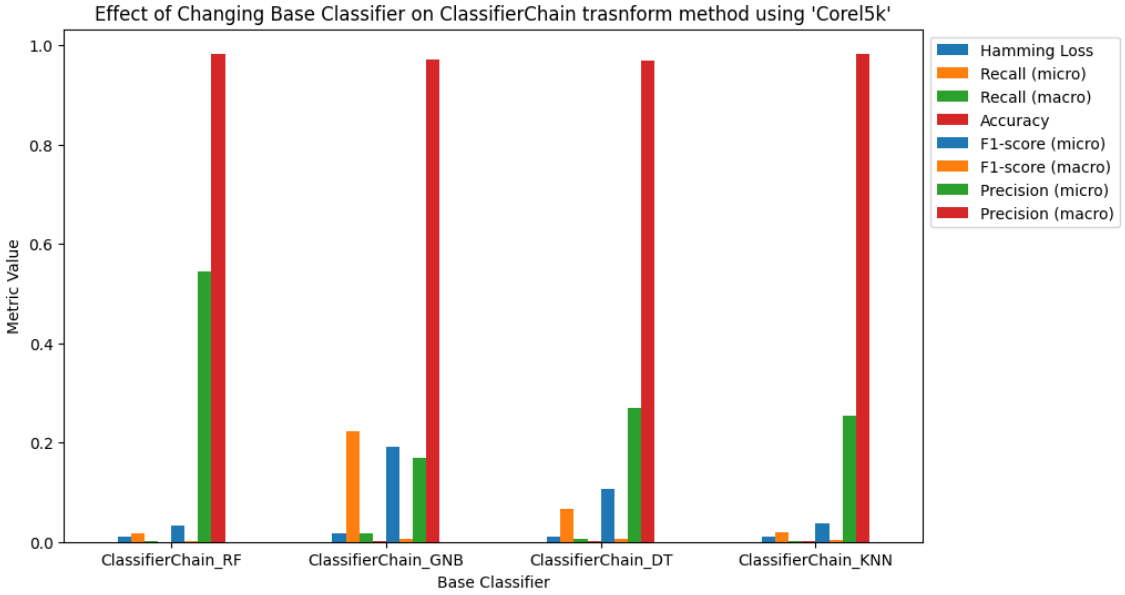

Figura 14: Rendimiento de ClassifierChain basado en el clasificador base en el conjunto de datos *Corel5k*

Según la figura 9, el método RakelD basado en RandomForest (RF) supera a otros modelos en varias métricas. GNB y DT muestran un rendimiento moderado, mientras que KNN tiene dificultades con la baja Recall y Precision.

En el conjunto de datos *genbase* (figura 15), $ClassifierChain\_RF$ supera de manera constante a otros clasificadores base. $ClassifierChain\_DT$ muestra un rendimiento excelente, especialmente en métricas de microavería. $ClassifierChain\_GNB$ le sigue con un rendimiento general sólido, mientras que $ClassifierChain\_KNN$ tiene un rendimiento débil.

Conclusiones

Observamos el efecto de la elección del clasificador base en el rendimiento de los algoritmos de transformación: RakelD y ClassifierChain en todos los conjuntos de datos multietiqueta. En particular, tanto RakelD como ClassifierChain basados en el clasificador RandomForest tienen un rendimiento destacado y superan a los algoritmos de transformación basados en Gaussian Naive Bayes, Decision Tree y k-Nearest Neighbors.

En contraste, los algoritmos de transformación basados en Decision Tree y Gaussian Naive Bayes muestran un rendimiento moderado en la mayoría de los conjuntos de datos, mientras que la transformación basada en kNN proporciona un rendimiento más bajo en la mayoría de los conjuntos de datos. En consecuencia, la elección del clasificador base afecta significativamente el rendimiento de los algoritmos de transformación, según nuestros experimentos.

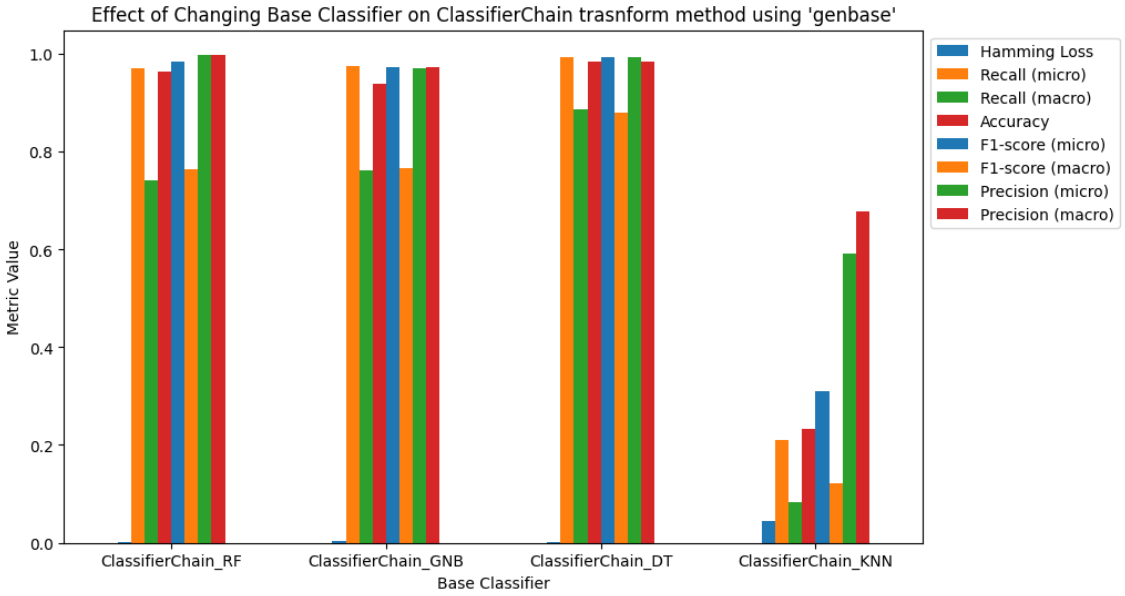

Figura 15: Rendimiento de ClassifierChain según el Clasificador Base en el conjunto de datos *genbase*

Además, esto puede estar relacionado con los datos utilizados, lo que requiere una investigación más profunda para comprender mejor las relaciones.

9.2 B-Tiempo de Ejecución de Métodos en Multietiquetado: Análisis y Diferencias

Al analizar los resultados de los métodos de clasificación multietiqueta en varios conjuntos de datos, se observan diferencias notables en los tiempos de ejecución. El modelo MLkNN tiende a tener tiempos de ejecución moderados, con un aumento notable en el conjunto de datos *Corel*5*k*, que tiene un mayor número de instancias, atributos y etiquetas.

En cambio, el clasificador Chain muestra tiempos variables, siendo más rápido en *genbase* y *enron*, pero notablemente más lento en *bibtex*, que tiene el mayor número de instancias, atributos y etiquetas entre los conjuntos de datos. RakelD destaca por su velocidad, especialmente en *genbase* y *yeast*; muestra consistentemente tiempos de ejecución rápidos, incluso en conjuntos de datos con características variables.

9.3 C-Análisis de Variaciones en Métricas de Desempeño entre Métodos de Clasificación Multietiqueta

Los resultados de cada algoritmo en la clasificación multietiqueta exhiben diferencias notables en varias métricas. En el método ClassifierChain, los conjuntos de datos *bibtex* y *genbase* muestran un Hamming Loss más bajo, así como recall, precisión, y puntuaciones

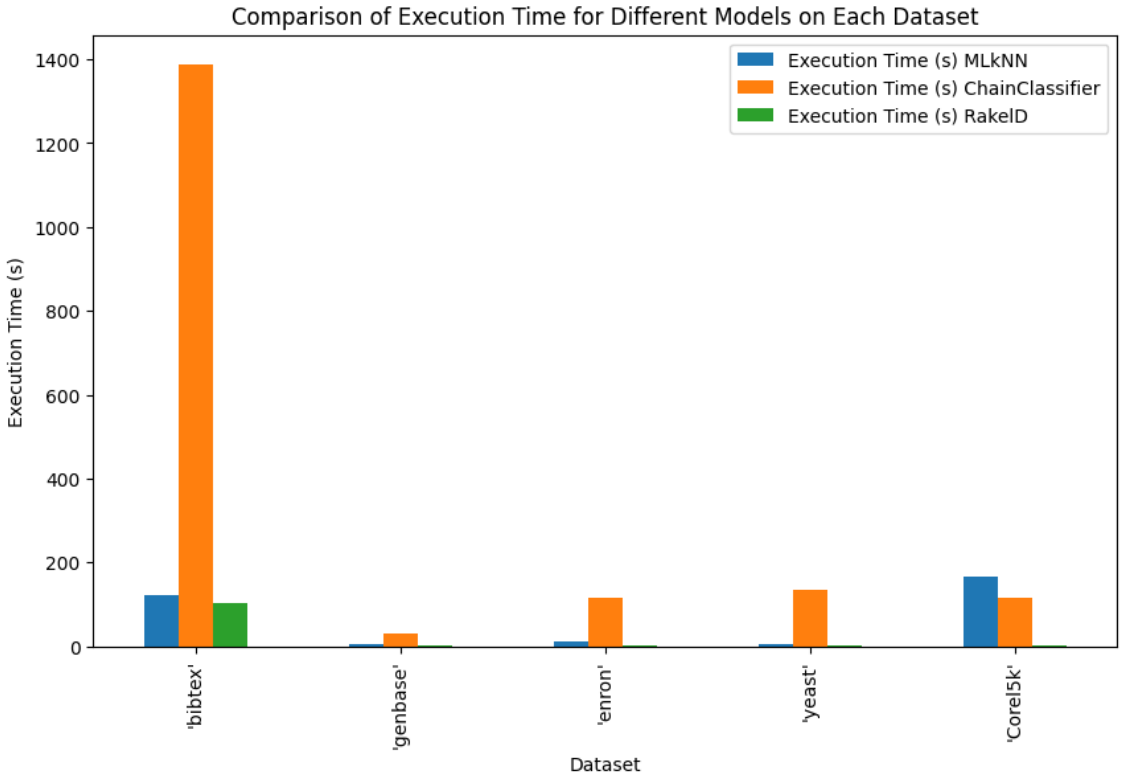

Figura 16: Métricas de Rendimiento del Algoritmo MLkNN en Conjuntos de Datos Multietiqueta

F1 más altas, indicando un rendimiento superior. Por otro lado, los conjuntos de datos *yeast*, *enron*, y *Corel5k* muestran valores más altos en estas métricas, lo que sugiere desafíos en la clasificación.

RakelD, por otro lado, demuestra fortalezas diferentes. Mientras que los conjuntos de datos *genbase* y *yeast* muestran un Hamming Loss relativamente más alto, *bibtex*, *enron*, y *Corel5k* exhiben valores más bajos. Se observan tendencias similares en recall, precisión y puntuaciones F1.

En comparación, MLkNN muestra resultados mixtos. Para *bibtex* y *yeast*, demuestra un rendimiento moderado con métricas equilibradas. *genbase* muestra un buen rendimiento, mientras que *enron* y *Corel5k* tienen un Hamming Loss más alto y una precisión más baja, lo que indica posibles desafíos.

	Hamming Loss	Recall (micro)	Recall (macro)	Accuracy	F1-score (micro)	F1-score (macro)	Precision (micro)	Precision (macro)	Execution Time (s)	Number of instances (n)	Number of attributes (f)	Number of labels (l)	Cardinality (car)	Density (den)	avgIR
RakelD 'bibtex'	0.038027	0.245749	0.140156	0.050845	0.163355	0.101468	0.122337	0.117773	102.763185	7395	1836	159	2.4019	0.0151	12.4983
RakelD 'genbase'	0.002741	0.972256	0.738697	0.933535	0.970500	0.741302	0.968750	0.970617	1.272757	662	1186	27	1.2523	0.0464	37.3146
RakelD 'enron'	0.151642	0.576000	0.274324	0.000000	0.326257	0.154439	0.227582	0.185185	3.822263	1702	1001	53	3.3784	0.0637	73.9528
RakelD 'yeast'	0.287783	0.598281	0.491573	0.092677	0.557203	0.442358	0.521402	0.416402	1.200228	2417	103	14	4.2371	0.3026	7.1968
RakelD 'Corel5k'	0.151642	0.576000	0.274324	0.000000	0.326257	0.154439	0.227582	0.185185	3.822263	5000	499	374	3.5220	0.0094	189.5676

Figura 17: Métricas de Rendimiento del Algoritmo RakelD en Conjuntos de Datos Multi-etiqueta

9.4 D-Analysis of Relationship Between Evaluation Metrics and ML Dataset Characteristics

9.4.1 Relación del rendimiento de RakelD con las características de los datos

Como se observa en la figura 17, aunque algunas métricas no están bien calculadas en la evaluación del modelo RakelD, notamos que a mayor número de instancias y cardinalidad, menor es la precisión. El efecto del número de atributos y el número de etiquetas en el rendimiento de RakelD está relacionado con el número de instancias, por lo que un buen equilibrio entre ellos, como en el caso del conjunto de datos *genbase*, podría proporcionar un excelente rendimiento de multietiquetado.

El efecto de la Tasa de Desbalance Promedio es notable en el rendimiento del modelo, donde un mayor $avgIR$ aumenta la pérdida de Hamming, como en el caso del conjunto de datos *yeast*.

9.4.2 Relación del Rendimiento de ClassifierChain con las Características de los Datos

	Hamming Loss	Recall (micro)	Recall (macro)	Accuracy	F1-score (micro)	F1-score (macro)	Precision (micro)	Precision (macro)	Execution Time (s)	Number of instances (n)	Number of attributes (f)	Number of labels (l)	Cardinality (car)	Density (den)	avgIR
ClassifierChain 'bibtex'	0.013088	0.151109	0.060604	0.112373	0.258612	0.081502	0.896160	0.659941	1388.489277	7395	1836	159	2.4019	0.0151	12.4983
ClassifierChain 'genbase'	0.001399	0.971049	0.741713	0.963746	0.984709	0.763947	0.998759	0.998911	29.745906	662	1186	27	1.2523	0.0464	37.3146
ClassifierChain 'enron'	0.046427	0.495304	0.160167	0.131610	0.576285	0.199843	0.688921	0.390546	116.787480	1702	1001	53	3.3784	0.0637	73.9528
ClassifierChain 'yeast'	0.192801	0.576018	0.347404	0.217625	0.643925	0.382425	0.729984	0.764834	135.028718	2417	103	14	4.2371	0.3026	7.1968
ClassifierChain 'Corel5k'	0.046427	0.495304	0.160167	0.131610	0.576285	0.199843	0.688921	0.390546	116.787480	5000	499	374	3.5220	0.0094	189.5676

Figura 18: Métricas de Rendimiento del Algoritmo ClassifierChain en Conjuntos de Datos Multietiqueta

El rendimiento del algoritmo ClassifierChain varía según los conjuntos de datos multietiqueta 18, mostrando dependencias en características como el número de instancias,

atributos y cardinalidad. Números más altos generalmente contribuyen a una menor Pérdida de Hamming y un mejor Recuerdo/Precisión, notablemente en el conjunto de datos 'genbase'con un rendimiento excepcional asociado a un mayor número de atributos.

Por otro lado, el impacto del número de etiquetas parece matizado, como se observa en *yeast*, donde un menor número de etiquetas no se traduce necesariamente en un mejor rendimiento. La densidad de los conjuntos de datos también juega un papel crucial, con una mayor densidad correlacionando positivamente con el rendimiento, ejemplificado por *yeast* y *bibtex*. Además, la proporción media de desequilibrio muestra una relación inversa con el rendimiento, siendo *genbase* y *yeast* ejemplos de un rendimiento degradado con una proporción de desequilibrio más alta.

9.4.3 Relación del rendimiento de MLkNN con las características de los datos

Algorithm	Hamming Loss	Recall (micro)	Recall (macro)	Accuracy	F1-score (micro)	F1-score (macro)	Precision (micro)	Precision (macro)	Execution Time (s)	Number of instances (n)	Number of attributes (f)	Number of labels (l)	Cardinality (car)	Density (den)	avgIR
MLkNN 'bibtex'	0.016964	0.208535	0.131990	0.073158	0.270820	0.163107	0.386155	0.288927	121.055111	7395	1836	159	2.4019	0.0151	12.4983
MLkNN 'genbase'	0.046604	0.302774	0.134023	0.323263	0.376030	0.162888	0.496047	0.520728	4.449108	662	1186	27	1.2523	0.0464	37.3146
MLkNN 'enron'	0.060694	0.415304	0.160749	0.103995	0.465906	0.180091	0.530549	0.346595	10.505922	1702	1001	53	3.3784	0.0637	73.9528
MLkNN 'yeast'	0.017678	0.629721	0.238680	0.470348	0.663495	0.260723	0.701097	0.688646	5.492321	2417	103	14	4.2371	0.3026	7.1968
MLkNN 'Corel5k'	0.012338	0.093072	0.023859	0.005600	0.124402	0.029683	0.187529	0.360241	164.981008	5000	499	374	3.5220	0.0094	189.5676

Figura 19: Métricas de rendimiento del algoritmo MLkNN en conjuntos de datos multietiqueta

La figura 19 muestra los resultados de MLkNN junto con las estadísticas de los datos. A diferencia de los conjuntos de datos transformados, los valores altos de densidad y avgIR se asocian con mejores valores de precisión, promedio micro de cada f1-score, recall y precisión, pero están relacionados con valores más bajos del promedio macro de cada uno de ellos.

El efecto de la densidad, cardinalidad, el número de atributos, el número de etiquetas y el número de instancias en el rendimiento de MLkNN varía y no es estacionario en este caso.

9.4.4 Conclusión

Los resultados sobre el efecto de la cardinalidad en los clasificadores multietiqueta en el artículo Bernardini et al. (2013) concuerdan significativamente con las percepciones que derivamos del efecto de las características del conjunto de datos en el rendimiento de RakelD, que se observa en los altos valores de F1-score (micro o macro) y Precision (micro o macro) asociados con los valores más bajos en la cardinalidad y densidad.

9.5 E-Visualización Gráfica del Comportamiento de Métodos en Clasificación multietiqueta

9.5.1 Ejemplo de código para visualización

Listing 26: Ejemplo de código para visualización

```
import matplotlib.pyplot as plt
import seaborn as sns
# Transpose the DataFrame to have models as columns and metrics as rows
transposed_metrics = conc_yeast.transpose()
exclude_metrics = ['execution time (s)']
m = ['Hamming Loss', 'Recall (micro)', 'Recall (macro)', 'Accuracy',
        'F1-score (micro)', 'F1-score (macro)', 'Precision (micro)',
        'Precision (macro)']
# Plotting using Matplotlib
plt.figure(figsize=(14, 8))

# Plot line for each model, excluding the specified metrics
for model in transposed_metrics.columns:
    plt.plot(m, transposed_metrics[model][0:-1], marker='o', label=model)
plt.title('Comparison of Models in yeast Dataset')
plt.xlabel('Metrics')
plt.ylabel('Metric Values')
plt.xticks(rotation=45, ha='right')  # Rotate x-axis labels for better readability
plt.legend()
plt.tight_layout()
plt.show()
# Plot the 'execution time (s)' metric separately
plt.figure(figsize=(10, 6))
plt.plot(transposed_metrics.columns, transposed_metrics.loc['execution time (s)'], color='red', marker='o', linestyle='-', alpha=0.7)
# Add text annotations for each point on the line
for i, value in enumerate(transposed_metrics.loc['execution time (s)']):
    plt.text(transposed_metrics.columns[i], value, f'{value:.4f}', ha='center', va='bottom', fontsize=9, color='black')
plt.title('Execution Time Comparison in yeast Dataset')
plt.xlabel('Models')
plt.ylabel('Execution Time (s)')
plt.xticks(rotation=45, ha='right')  # Rotate x-axis labels for better readability
plt.tight_layout()
plt.show()
```

9.5.2 Visualización de los resultados en los conjunto de datos

Sabiendo que los resultados de los algoritmos en cada conjunto de datos ya han sido interpretados en esta subsección 8.2.

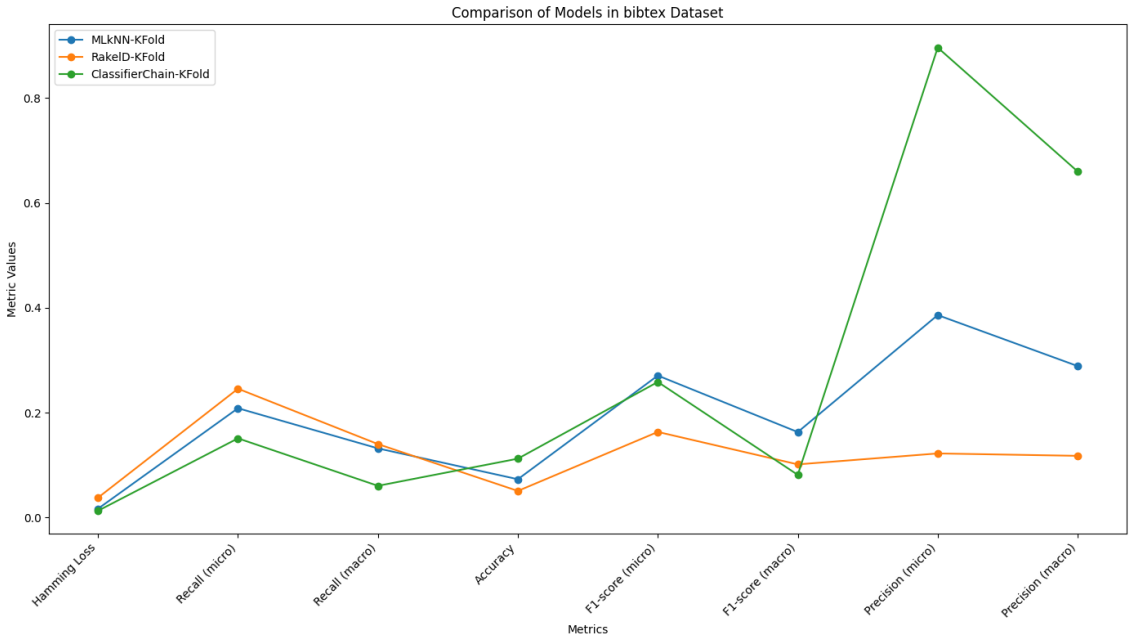

Figura 20: Visualización de Resultados de Algoritmos en *bibtex*

Las gráficas:figura 20, figura 21, figura 22, figura 23, and figura 24 de la clasificación multietiqueta utilizando MLkNN, ClassifierChain y RakelD destacan la variabilidad en el rendimiento de los clasificadores en relación con el conjunto de datos utilizado. Aunque los tres clasificadores se basan en enfoques diferentes, muestran una variabilidad coordinada, lo que significa que su rendimiento fluctúa consistentemente.

Observamos que los algoritmos responden de manera similar a las características específicas o cambios en los datos de entrada a lo largo de varias métricas. Este comportamiento sincronizado podría indicar que los algoritmos comparten patrones o respuestas comunes a las características subyacentes de los datos.

Referencias

Bernardini, F. C., Da Silva, R. B., Meza, E., and das Ostras-RJ-Brazil, R. (2013). Analyzing the influence of cardinality and density characteristics on multi-label learning. *Proc. X Encontro Nacional de Inteligencia Artificial e Computacional-ENIAC*, 2013.

Katakis, I., Tsoumakas, G., and Vlahavas, I. (2008). Multilabel text classification for automated tag suggestion. *ECML PKDD discovery challenge*, 75:2008.

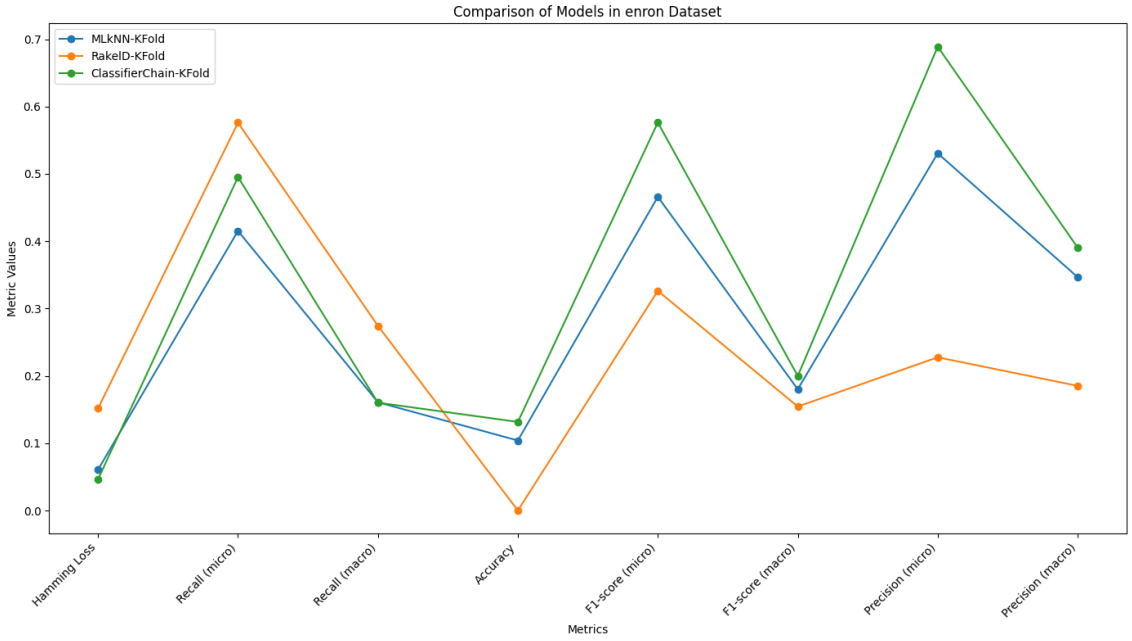

Figura 21: Visualización de Resultados de Algoritmos en *enron*

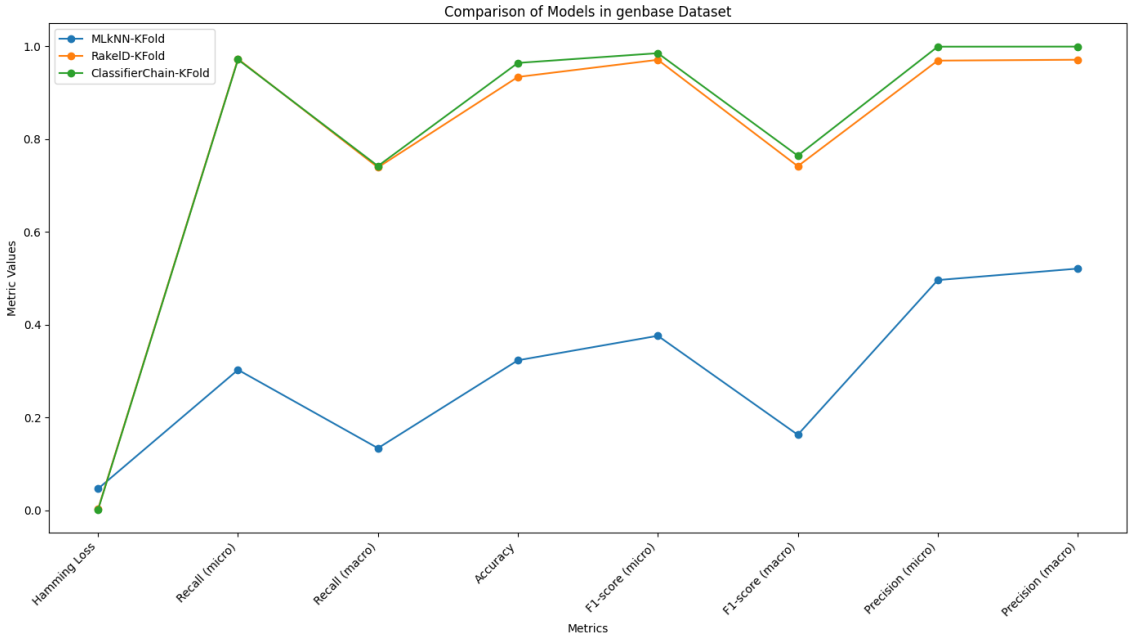

Figura 22: Visualización de Resultados de Algoritmos en *genbase*

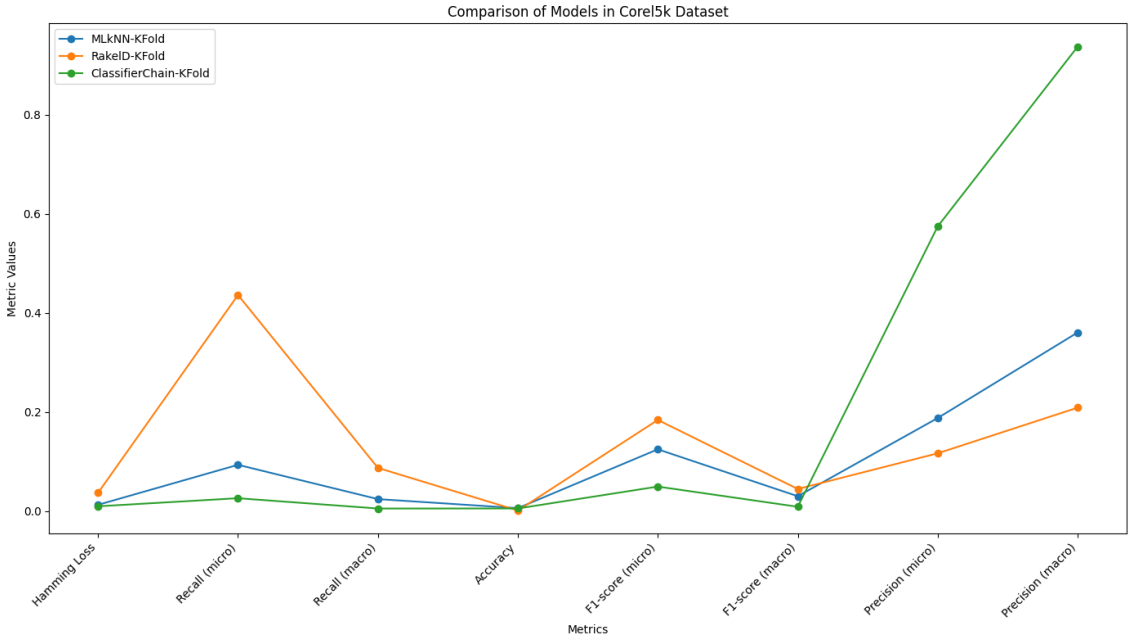

Figura 23: Visualización de Resultados de Algoritmos en *Corel5k*

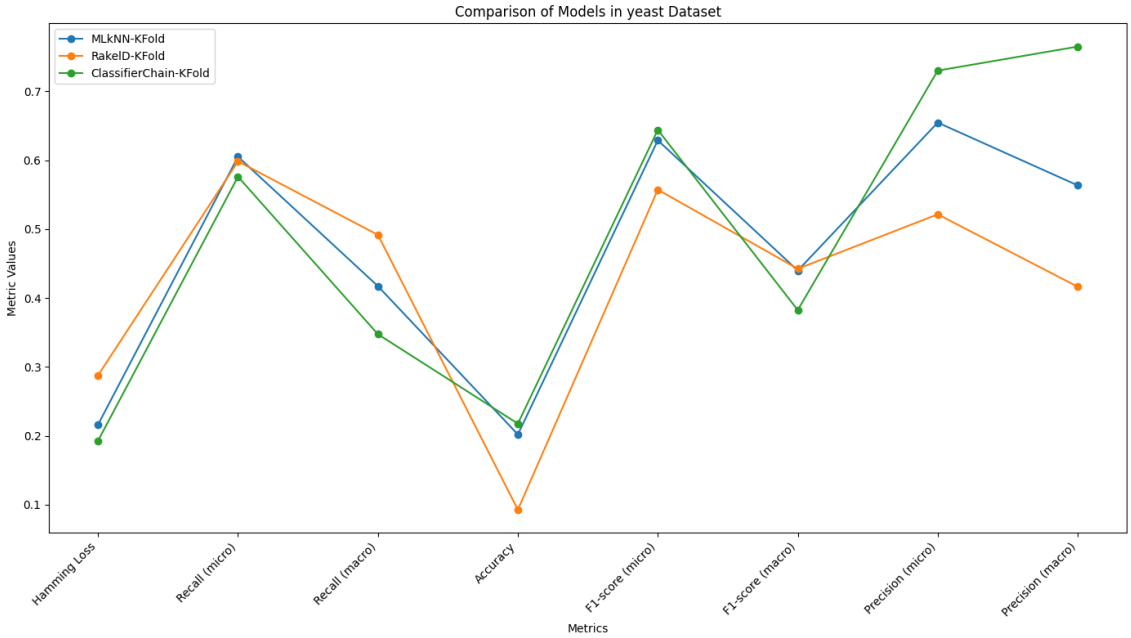

Figura 24: Visualización de Resultados de Algoritmos en *yeast*

Moyano, J. M. (2017). Multi-label classification dataset repository. Accessed on Enero 30, 2024.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

Szymański, P. and Kajdanowicz, T. (2017). A scikit-based Python environment for performing multi-label classification. *ArXiv e-prints*.

Turnbull, D., Barrington, L., Torres, D., and Lanckriet, G. (2008). Semantic annotation and retrieval of music and sound effects. *IEEE Transactions on Audio, Speech, and Language Processing*, 16(2):467–476.