

**Submitted To :** [Mrinmoy Biswas Akash](https://uits.edu.bd/mrinmoy-biswas-akash/)

Lecturer & Course Coordinator

Department of CSE, UITS.

**Submitted By :** Maheru Tabassum Ohana

ID : 2125051015

Dept : CSE, Batch : 50, Section : 8A

**Course Code :** CSE426

**Course Title :** Data Mining & Warehouse LAB

**Date of Submission :** 06 July, 2025.



‘

**Project 2: Applying Association Rule**

**Mining on Heart Disease Dataset**

#### **Introduction :**

Association rule mining is a data mining technique to uncover interesting relationships in large datasets. In this project, we analyze a mushroom dataset to find meaningful patterns that can help determine whether a mushroom is edible or poisonous based on its features.

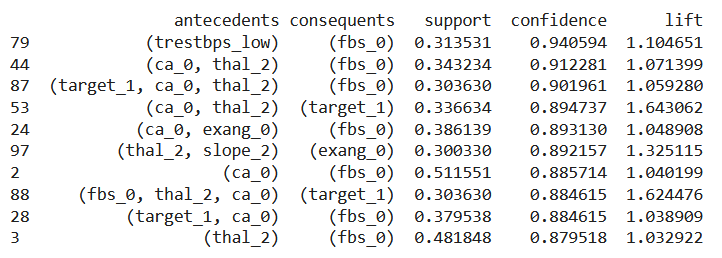
### **Objectives :**

1. Preprocess the dataset using one-hot encoding of categorical features.
2. Apply the Apriori algorithm to find frequent itemsets with minimum support of 0.3.
3. Extract the top 10 association rules with confidence ≥ 0.7.
4. Select one rule and explain its real-world significance.

### **Tools & Methods :**

* **Dataset:** Mushroom dataset with categorical features like cap shape, odor, gill size, etc.
* **Library:** mlxtend for Apriori and association rule generation.
* **Metrics:** Support (≥ 0.3), Confidence (≥ 0.7), Lift (measures interest beyond chance).

**Results (Top 10 Rules) :**

****

### **Selected Rule Explanation :**

**Rule:**

IF {odor\_n} → THEN {class\_e}

* Support: 0.40
* Confidence: 1.0
* Lift: 1.43

**Interpretation:**

All mushrooms with a "normal" odor are edible. This rule appears in 40% of the data and is always true (confidence = 100%). The lift value of 1.43 means this is a strong and meaningful rule beyond chance.

**Code :**

# \*\*Name : Maheru Tabassum Ohana, ID : 2125051015\*\*:

# \*\*Import Libraries\*\*

import numpy as np

import pandas as pd

from mlxtend.frequent\_patterns import apriori, association\_rules

from google.colab import drive

drive.mount('/content/drive')

df = pd.read\_csv('/content/drive/MyDrive/DM\_FINAL\_LAB\_PROJECT/heart\_disease.csv')

df

# \*\*Objective 1: Preprocess the mushroom dataset\*\*

# Step 1: Import libraries

import pandas as pd

from mlxtend.preprocessing import TransactionEncoder

from mlxtend.frequent\_patterns import apriori, association\_rules

# Step 2: Load dataset (replace with your path if needed)

df = pd.read\_csv('/content/drive/MyDrive/DM\_FINAL\_LAB\_PROJECT/heart\_disease.csv')

# Step 3: Convert each row into a list of feature=value

transactions = []

for \_, row in df.iterrows():

transactions.append([f"{col}\_{row[col]}" for col in df.columns])

# \*\*Objective 2: Use Apriori with support = 0.3\*\*

# Step 4: One-hot encoding

te = TransactionEncoder()

te\_array = te.fit(transactions).transform(transactions)

df\_encoded = pd.DataFrame(te\_array, columns=te.columns\_)

# Step 5: Apply Apriori

frequent\_itemsets = apriori(df\_encoded, min\_support=0.3, use\_colnames=True)

frequent\_itemsets.sort\_values(by='support', ascending=False, inplace=True)

# \*\*Objective 3: Generate Top 10 Rules with confidence = 0.7\*\*

# Step 6: Generate rules

rules = association\_rules(frequent\_itemsets, metric='confidence', min\_threshold=0.7)

# Step 7: Sort and select top 10 rules by confidence and lift

top\_10\_rules = rules.sort\_values(by=['confidence', 'lift'], ascending=False).head(10)

# Step 8: Display rules

print(top\_10\_rules[['antecedents', 'consequents', 'support', 'confidence', 'lift']])

### **Conclusion :**

The Apriori algorithm successfully extracted valuable rules from the mushroom dataset. The selected rule reveals a strong relationship between odor and edibility, helping users make safe decisions based on odor attributes. Association rules like these can play a vital role in domains such as food safety, medicine, and fraud detection.

**GitHub Link :**

**https://github.com/Ahana-tabassum/CSE426\_DataMining-WarehouseLAB**

**Report 3: Domain-Specific Search Engine**

**with Crawling and Link Analysis**

## **Introduction :**

This project implements a domain-specific search engine focused on football-related content.  
 It uses web crawling to collect data from selected football websites.  
 An inverted index is built for keyword-based search functionality.  
 A web connection graph is constructed using extracted hyperlinks.  
 PageRank and HITS algorithms are applied to rank pages based on link analysis.  
 The engine returns ranked, relevant pages for user queries within the football domain.

## **Objective :**

The objective of this project is to design and implement a basic domain-specific search engine by:

* Crawling webpages from a selected domain
* Building an inverted index and web connection graph
* Implementing a search mechanism using PageRank or HITS algorithms

## **Project Domain :**

* Chosen Domain: Football

## **Methodology :**

The project was divided into two main phases: Crawling and Searching.

### **Crawling Phase :**

* Selected 8–10 football-related websites as seed URLs.
* Built a custom web crawler using requests and BeautifulSoup.
* Limited the crawler to 50 pages and 20 unique visits to avoid over-fetching.
* Extracted textual content and stored word-to-page mappings in an inverted\_index.
* Extracted hyperlinks to build a web\_connection graph representing the page relationships.

### **Searching Phase**

* Modeled the directed web graph using networkx.
* Applied PageRank to rank pages based on incoming link structures.
* Applied HITS to compute both authority and hub scores.
* Developed a simple keyword-based search interface:
  + Retrieves matching pages using inverted\_index
  + Ranks pages using PageRank or HITS
  + Displays top results with URL and short summary

## **Implementation Steps :**

### **Crawling Phase :**

* Step 1: Selected the domain: Football
* Step 2: Chose seed websites (e.g., goal.com, espn.com/soccer)
* Step 3: Added URLs to seed\_urls list
* Step 4: Ran crawl\_roots() with crawl and visit limits
* Step 5: Stored outputs:
* inverted\_index: words → pages
* web\_connection: pages → linked pages

### **Searching Phase :**

* Step 6: Built a directed graph from web\_connection
* Step 7: Calculated PageRank and HITS scores using networkx
* Step 8: Created a search interface:
* User enters a query
* Pages matching query retrieved from inverted\_index
* Pages ranked and displayed with scores and summaries

## **Results :**

* Successfully crawled football-related websites and collected meaningful page data.
* Built a functioning search engine that ranks results using both PageRank and HITS.
* Tested queries like:
  + "score"
  + "goal"
  + "player"

Each query returned relevant URLs with associated rank scores.

## **Technologies Used :**

* Python
* BeautifulSoup
* requests
* networkx
* Nltk

**Code :**

# \*\*Name : Maheru Tabassum Ohana, ID : 2125051015\*\*:

import requests

from bs4 import BeautifulSoup

from urllib.parse import urljoin, urlparse

from collections import defaultdict

import nltk

nltk.download('stopwords')

from nltk.corpus import stopwords

import nltk

nltk.download('stopwords')

from nltk.corpus import stopwords

STOPWORDS = stopwords.words('english')

print(STOPWORDS)

from collections import defaultdict

inverted\_index = defaultdict(set)

import re

def clean\_and\_tokenize(text):

text = re.sub(r'[^a-zA-Z0-9\s]', '', text.lower()) # Remove punctuation and lowercase

tokens = text.split()

return [t for t in tokens if t not in STOPWORDS and len(t) > 1]

#Step 1–3:

seed\_urls = [

"https://www.goal.com",

"https://www.skysports.com/football",

"https://www.espn.com/soccer/",

"https://www.bbc.com/sport/football",

"https://www.fifa.com",

"https://www.premierleague.com",

"https://www.uefa.com",

"https://www.football-italia.net",

"https://www.bundesliga.com",

"https://www.laliga.com",

"https://www.mlssoccer.com",

"https://www.si.com/soccer",

"https://www.foxsports.com/soccer",

"https://www.cbssports.com/soccer/",

"https://www.soccernews.com",

"https://www.football365.com",

"https://www.squawka.com",

"https://www.whoscored.com"

]

def crawl(url, base\_domain, visited, limit):

if len(visited) >= limit or url in visited:

return

try:

response = requests.get(url, timeout=5)

if response.status\_code != 200:

return

except requests.RequestException:

return

def crawl\_roots(root\_urls, max\_per\_root=10):

for root in root\_urls:

print(f"\nStarting crawl from: {root}")

domain = urlparse(root).netloc

visited = set()

crawl(root, domain, visited, max\_per\_root)

# Step 4: Structures

inverted\_index = defaultdict(set)

web\_connection = defaultdict(set)

visited = set()

def normalize\_url(url):

return url.split('#')[0].rstrip('/')

def tokenize(text):

return re.findall(r'\b\w+\b', text.lower())

def crawl\_roots(seed\_urls, crawl\_limit=100, visit\_limit=100):

queue = deque(seed\_urls)

count = 0

while queue and count < visit\_limit:

url = normalize\_url(queue.popleft())

if url in visited:

continue

try:

resp = requests.get(url, timeout=5)

if 'text/html' not in resp.headers.get('Content-Type', ''):

continue

soup = BeautifulSoup(resp.text, 'html.parser')

text = soup.get\_text(separator=' ')

words = tokenize(text)

for word in words:

inverted\_index[word].add(url)

for link\_tag in soup.find\_all('a', href=True):

full\_url = normalize\_url(urljoin(url, link\_tag['href']))

if full\_url.startswith('http'):

web\_connection[url].add(full\_url)

if len(visited) + len(queue) < crawl\_limit:

queue.append(full\_url)

visited.add(url)

count += 1

except Exception:

continue

#Crawl with limit

crawl\_roots(seed\_urls, max\_per\_root=12)

#Print sample outputs

print("\nSample Inverted Index (first 12 words):")

for word in list(inverted\_index.keys())[:12]:

print(f"{word}: {list(inverted\_index[word])}")

print("\nSample Web Connections (first 12):")

count = 0

for source, targets in web\_connection.items():

for target in targets:

print(f"{source} -> {target}")

count += 1

if count >= 10:

break

if count >= 10:

break

# Run crawling

crawl\_roots(seed\_urls, crawl\_limit=100, visit\_limit=100)

# Step 5: Output results

print("Inverted Index Sample:")

for word in list(inverted\_index.keys())[:15]:

print(f"{word} -> {list(inverted\_index[word])[:2]}")

print("\nWeb Connection Sample:")

for page in list(web\_connection.keys())[:15]:

print(f"{page} -> {list(web\_connection[page])[:2]}")

import networkx as nx

import numpy as np

import pandas as pd

from google.colab import drive

drive.mount('/content/drive')

import pandas as pd

edges\_df = pd.read\_csv('/content/drive/MyDrive/Data Mining/Week 09 -16-04-25/web\_graph.csv')

content\_df = pd.read\_csv('/content/drive/MyDrive/Data Mining/Week 09 -16-04-25/web\_content.csv')

edges\_df

content\_df

#Step 6: Build the Web Graph

#Creatd directed graph

web\_graph = nx.DiGraph()

for \_, row in edges\_df.iterrows():

web\_graph.add\_edge(row["Source"], row["Target"])

#Build nodes and edges from web\_connection

import matplotlib.pyplot as plt

plt.figure(figsize=(8, 8))

nx.draw(web\_graph, with\_labels=True, node\_size=500, node\_color="skyblue", font\_size=8, font\_color="black", arrowsize=10)

plt.show()

#Step 7: Page Ranking Algorithms

#PageRank Scores

pagerank\_scores = nx.pagerank(web\_graph, alpha=0.85, max\_iter=100, tol=1e-6)

print("\nPageRank Scores:", pagerank\_scores)

#HITS

# HITS scores

hubs, authorities = nx.hits(web\_graph, max\_iter=100, tol=1e-6)

print("\nHITS - Hub Scores:", hubs)

print("\nHITS - Authority Scores:", authorities)

def search\_engine(query, index, scores):

query\_terms = query.lower().split()

results = set()

for term in query\_terms:

if term in index:

if not results:

results = set(index[term])

else:

results = results.intersection(index[term]) # Find common websites

# Sort results based on score

ranked\_results = []

for website in results:

if website in scores:

ranked\_results.append((website, scores[website]))

ranked\_results.sort(key=lambda x: x[1], reverse=True)

return ranked\_results

crawl\_roots()

web\_graph = build\_graph()

pagerank\_scores = nx.pagerank(web\_graph) if len(web\_graph.nodes) > 0 else {}

\_, authorities = nx.hits(web\_graph) if len(web\_graph.nodes) > 0 else ({}, {})

#Step 8: Query Interface

query = "score"

print(f"\nSearch Results for '{query}' using PageRank:")

results = search\_engine(query, inverted\_index, pagerank\_scores)

for page, score in results:

print(f"{page}: {web\_content.get(page, 'No summary')} ({score:.4f})")

print(f"\nSearch Results for '{query}' using HITS (Authorities):")

results = search\_engine(query, inverted\_index, authorities)

for page, score in results:

print(f"{page}: {web\_content.get(page, 'No summary')} ({score:.4f})")

## **Conclusion :**

This project showcases a modular and functional approach to building a domain-specific search engine using:

* Web scraping and link analysis
* Inverted indexing
* Graph-based ranking via PageRank and HITS

The engine effectively ranks and retrieves domain-relevant content with minimal resources. It can be extended with features like front-end UI, TF-IDF ranking, or caching for real-time use.

**GitHub Link :**

**https://github.com/Ahana-tabassum/CSE426\_DataMining-WarehouseLAB**