Web Mining Lab Assignment 7

Name: Om Ashish Mishra

Registration Number: 16BCE0789

Slot: F2

The Code:

from sklearn import model\_selection, preprocessing, linear\_model, naive\_bayes, metrics, svm

from sklearn.feature\_extraction.text import TfidfVectorizer, CountVectorizer

#from sklearn import decomposition, ensemble

import pandas, numpy, string

#from keras.preprocessing import text, sequence

#from keras import layers, models, optimizers

# load the dataset

data = open('d:/corpus',encoding="utf8").read()

labels, texts = [], []

for i, line in enumerate(data.split("\n")):

content = line.split()

labels.append(content[0])

texts.append(content[1])

# create a dataframe using texts and lables

trainDF = pandas.DataFrame()

trainDF['text'] = texts

trainDF['label'] = labels

#printing data frame

print(trainDF['text'])

print(trainDF['label'])

#Next, we will split the dataset into training and validation sets so that we can train and test classifier. Also, we will encode our target column so that it can be used in machine learning models.

# split the dataset into training and validation datasets

train\_x, valid\_x, train\_y, valid\_y = model\_selection.train\_test\_split(trainDF['text'], trainDF['label'])

# label encode the target variable

encoder = preprocessing.LabelEncoder()

train\_y = encoder.fit\_transform(train\_y)

valid\_y = encoder.fit\_transform(valid\_y)

print("\n\nencoder\n",train\_y)

# create a count vectorizer object

#Count Vector is a matrix notation of the dataset in which every row represents a document from the corpus, every column represents a term from the corpus,

#and every cell represents the frequency count of a particular term in a particular document

count\_vect = CountVectorizer(analyzer='word', token\_pattern=r'\w{1,}')

count\_vect.fit(trainDF['text'])

# transform the training and validation data using count vectorizer object

xtrain\_count = count\_vect.transform(train\_x)

xvalid\_count = count\_vect.transform(valid\_x)

# word level tf-idf

tfidf\_vect = TfidfVectorizer(analyzer='word', token\_pattern=r'\w{1,}', max\_features=5000)

tfidf\_vect.fit(trainDF['text'])

xtrain\_tfidf = tfidf\_vect.transform(train\_x)

xvalid\_tfidf = tfidf\_vect.transform(valid\_x)

# ngram level tf-idf

tfidf\_vect\_ngram = TfidfVectorizer(analyzer='word', token\_pattern=r'\w{1,}', ngram\_range=(2,3), max\_features=5000)

tfidf\_vect\_ngram.fit(trainDF['text'])

xtrain\_tfidf\_ngram = tfidf\_vect\_ngram.transform(train\_x)

xvalid\_tfidf\_ngram = tfidf\_vect\_ngram.transform(valid\_x)

# characters level tf-idf

tfidf\_vect\_ngram\_chars = TfidfVectorizer(analyzer='char', token\_pattern=r'\w{1,}', ngram\_range=(2,3), max\_features=5000)

tfidf\_vect\_ngram\_chars.fit(trainDF['text'])

xtrain\_tfidf\_ngram\_chars = tfidf\_vect\_ngram\_chars.transform(train\_x)

xvalid\_tfidf\_ngram\_chars = tfidf\_vect\_ngram\_chars.transform(valid\_x)

def train\_model(classifier, feature\_vector\_train, label, feature\_vector\_valid, is\_neural\_net=False):

# fit the training dataset on the classifier

classifier.fit(feature\_vector\_train, label)

# predict the labels on validation dataset

predictions = classifier.predict(feature\_vector\_valid)

if is\_neural\_net:

predictions = predictions.argmax(axis=-1)

return metrics.accuracy\_score(predictions, valid\_y)

# Naive Bayes on Count Vectors

accuracy = train\_model(naive\_bayes.MultinomialNB(), xtrain\_count, train\_y, xvalid\_count)

print ("NaiveBayes, Count Vectors: ", accuracy)

# Naive Bayes on Word Level TF IDF Vectors

accuracy = train\_model(naive\_bayes.MultinomialNB(), xtrain\_tfidf, train\_y, xvalid\_tfidf)

print ("NaiveBayes, WordLevel TF-IDF: ", accuracy)

# Naive Bayes on Ngram Level TF IDF Vectors

accuracy = train\_model(naive\_bayes.MultinomialNB(), xtrain\_tfidf\_ngram, train\_y, xvalid\_tfidf\_ngram)

print ("Naive Bayes, N-Gram Vectors: ", accuracy)

# Naive Bayes on Character Level TF IDF Vectors

accuracy = train\_model(naive\_bayes.MultinomialNB(), xtrain\_tfidf\_ngram\_chars, train\_y, xvalid\_tfidf\_ngram\_chars)

print ("NaiveBayes, CharLevel Vectors: ", accuracy)

The Output:

Assosiative property

import pandas as pd

import numpy as np

import sys

from itertools import combinations, groupby

from collections import Counter

from IPython.display import display

import matplotlib.pyplot as plt

orders = pd.read\_csv('d:/order\_products\_\_prior.csv')

#print('orders -- dimensions: {0}; size: {1}',format(orders.shape, size(orders)))

display(orders.head())

# Convert from DataFrame to a Series, with order\_id as index and item\_id as value

orders = orders.set\_index('order\_id')['product\_id'].rename('item\_id')

display(orders.head(10))

type(orders)

#print('dimensions: {0}; size: {1}; unique\_orders: {2}; unique\_items: {3}'

# .format(orders.shape, size(orders), len(orders.index.unique()), len(orders.value\_counts())))

# Returns frequency counts for items and item pairs

def freq(iterable):

if type(iterable) == pd.core.series.Series:

return iterable.value\_counts().rename("freq")

else:

return pd.Series(Counter(iterable)).rename("freq")

# Returns number of unique orders

def order\_count(order\_item):

return len(set(order\_item.index))

# Returns generator that yields item pairs, one at a time

def get\_item\_pairs(order\_item):

order\_item = order\_item.reset\_index().as\_matrix()

for order\_id, order\_object in groupby(order\_item, lambda x: x[0]):

item\_list = [item[1] for item in order\_object]

for item\_pair in combinations(item\_list, 2):

yield item\_pair

# Returns frequency and support associated with item

def merge\_item\_stats(item\_pairs, item\_stats):

return (item\_pairs

.merge(item\_stats.rename(columns={'freq': 'freqA', 'support': 'supportA'}), left\_on='item\_A', right\_index=True)

.merge(item\_stats.rename(columns={'freq': 'freqB', 'support': 'supportB'}), left\_on='item\_B', right\_index=True))

# Returns name associated with item

def merge\_item\_name(rules, item\_name):

columns = ['itemA','itemB','freqAB','supportAB','freqA','supportA','freqB','supportB',

'confidenceAtoB','confidenceBtoA','lift']

rules = (rules

.merge(item\_name.rename(columns={'item\_name': 'itemA'}), left\_on='item\_A', right\_on='item\_id')

.merge(item\_name.rename(columns={'item\_name': 'itemB'}), left\_on='item\_B', right\_on='item\_id'))

return rules[columns]

def association\_rules(order\_item, min\_support):

print("Starting order\_item: {:22d}".format(len(order\_item)))

# Calculate item frequency and support

item\_stats = freq(order\_item).to\_frame("freq")

item\_stats['support'] = item\_stats['freq'] / order\_count(order\_item) \* 100

# Filter from order\_item items below min support

qualifying\_items = item\_stats[item\_stats['support'] >= min\_support].index

order\_item = order\_item[order\_item.isin(qualifying\_items)]

print("Items with support >= {}: {:15d}".format(min\_support, len(qualifying\_items)))

print("Remaining order\_item: {:21d}".format(len(order\_item)))

# Filter from order\_item orders with less than 2 items

order\_size = freq(order\_item.index)

qualifying\_orders = order\_size[order\_size >= 2].index

order\_item = order\_item[order\_item.index.isin(qualifying\_orders)]

print("Remaining orders with 2+ items: {:11d}".format(len(qualifying\_orders)))

print("Remaining order\_item: {:21d}".format(len(order\_item)))

# Recalculate item frequency and support

item\_stats = freq(order\_item).to\_frame("freq")

item\_stats['support'] = item\_stats['freq'] / order\_count(order\_item) \* 100

# Get item pairs generator

item\_pair\_gen = get\_item\_pairs(order\_item)

# Calculate item pair frequency and support

item\_pairs = freq(item\_pair\_gen).to\_frame("freqAB")

item\_pairs['supportAB'] = item\_pairs['freqAB'] / len(qualifying\_orders) \* 100

print("Item pairs: {:31d}".format(len(item\_pairs)))

# Filter from item\_pairs those below min support

item\_pairs = item\_pairs[item\_pairs['supportAB'] >= min\_support]

print("Item pairs with support >= {}: {:10d}\n".format(min\_support, len(item\_pairs)))

# Create table of association rules and compute relevant metrics

item\_pairs = item\_pairs.reset\_index().rename(columns={'level\_0': 'item\_A', 'level\_1': 'item\_B'})

item\_pairs = merge\_item\_stats(item\_pairs, item\_stats)

item\_pairs['confidenceAtoB'] = item\_pairs['supportAB'] / item\_pairs['supportA']

item\_pairs['confidenceBtoA'] = item\_pairs['supportAB'] / item\_pairs['supportB']

item\_pairs['lift'] = item\_pairs['supportAB'] / (item\_pairs['supportA'] \* item\_pairs['supportB'])

# Return association rules sorted by lift in descending order

return item\_pairs.sort\_values('lift', ascending=False)

rules = association\_rules(orders, 0.01)

# Replace item ID with item name and display association rules

item\_name = pd.read\_csv('d:/order\_products\_\_prior.csv')

item\_name = item\_name.rename(columns={'product\_id':'item\_id', 'product\_name':'item\_name'})

rules\_final = merge\_item\_name(rules, item\_name).sort\_values('lift', ascending=False)

display(rules\_final)

Centrality

import networkx as nx

def degree\_centrality(G, nodes):

r"""Compute the degree centrality for nodes in a bipartite network.

The degree centrality for a node `v` is the fraction of nodes

connected to it.

Parameters

----------

G : graph

A bipartite network

nodes : list or container

Container with all nodes in one bipartite node set.

Returns

-------

centrality : dictionary

Dictionary keyed by node with bipartite degree centrality as the value.

Notes

-----

The nodes input parameter must conatin all nodes in one bipartite node set,

but the dictionary returned contains all nodes from both bipartite node

sets.

For unipartite networks, the degree centrality values are

normalized by dividing by the maximum possible degree (which is

`n-1` where `n` is the number of nodes in G).

In the bipartite case, the maximum possible degree of a node in a

bipartite node set is the number of nodes in the opposite node set

[1]\_. The degree centrality for a node `v` in the bipartite

sets `U` with `n` nodes and `V` with `m` nodes is

.. math::

d\_{v} = \frac{deg(v)}{m}, \mbox{for} v \in U ,

d\_{v} = \frac{deg(v)}{n}, \mbox{for} v \in V ,

where `deg(v)` is the degree of node `v`.

"""

top = set(nodes)

bottom = set(G) - top

s = 1.0/len(bottom)

centrality = dict((n,d\*s) for n,d in G.degree\_iter(top))

s = 1.0/len(top)

centrality.update(dict((n,d\*s) for n,d in G.degree\_iter(bottom)))

return centrality

G=nx.erdos\_renyi\_graph(100,0.5)

d=nx.degree\_centrality(G)

print(d)