Assignment 3

Bryan and Thanmaye

Part 1:

The spread on each of the classes (except group 1) is quite large, this would indicate that the pre-defined classification might not be the best. One thing to consider is that neither the inner product nor the polynomial kernel are adjusted for file size, so the grouping could still be fine.

Results

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

Part 2:

Some stemming was done, as you can see in the picture there was some stemming for various arithmetic operations, jump and move operations. This seems to have to have improved the average kernel by cluster as well as the standard deviation.

A screen shot of a computer program

Description automatically generated

Results

A screenshot of a computer

Description automatically generatedA screenshot of a computer program

Description automatically generated

Part 3:

A stronger remapping of the terms was applied in part 3. Though I don’t think this was a significant improvement

A computer screen shot of a program code

Description automatically generated

Results

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

Part 4:

In this part we analyze our own spectral clusters using silhouettes. Using each dictionary and both kernels here are the silhouettes. The interesting thing about the results is that the two best performing models are d1, inner product with k=2 and D2, polynomial with k=3. Another interesting note is that there doesn’t seem to be any correlation between the number of clusters and the performance, and that while the original had 5 different cluster, the best performance is with less clusters.

D0, inner product D0, polynomial

A computer screen shot of a number

Description automatically generatedA computer screen shot of numbers and symbols

Description automatically generated

D1, inner product D0, polynomial

A computer screen shot of a number

Description automatically generatedA screenshot of a computer program

Description automatically generated

D2, inner product D2, polynomial

A computer screen shot of a number

Description automatically generatedA computer screen shot of a number

Description automatically generated

Code

“””main.py”””

import kernel

import spectral

import vectorize

import vocab

import numpy as np

import os

from typing import List

def silhouette(K: np.ndarray, clusters: np.ndarray) -> float:

"""

Compute the silhouette score

"""

# compute average distane between points in the same cluster

d = np.zeros(K.shape[0])

for i in range(K.shape[0]):

d[i] = np.mean(K[i, clusters == clusters[i]])

# compute the smallest average distance between a point and any other cluster

D = np.zeros(K.shape[0])

for i in range(K.shape[0]):

D[i] = np.min(

[np.mean(K[i, clusters == j])

for j in set(clusters) if j != clusters[i]]

)

# compute the silhouette score

s = (D - d) / np.maximum(d, D)

return np.mean(s)

if \_\_name\_\_ == "\_\_main\_\_":

# list of filenames

files = [f"./data/{f}" for f in os.listdir("./data")]

files.remove("./data/.DS\_Store")

class\_sizes = vectorize.classCounts(files)

class\_keys = sorted(list(class\_sizes.keys()))

results = open("results.txt", "w")

"""part 1"""

results.writelines("PART 1!!:\n")

# dictionary of terms across all files

D0 = vocab.dictionary(files)

# re-vectorize with revised dictionay

vectors: List[np.ndarray] = [vectorize.vectorize(f, D0) for f in files]

# D is the Document-term matrix

D = np.vstack(vectors)

# K is the kernel matrix

K0 = kernel.buildKernel(D, "dot")

results.writelines("dot kernel\n")

sub\_matrices = kernel.extractSubmatrices(K0, class\_sizes, class\_keys)

[

kernel.descriptiveStats(kernel\_mat, class\_label, results)

for class\_label, kernel\_mat in sub\_matrices.items()

]

for k in range(2, 5):

clusters = spectral.spectralClustering(K0, k)

silhouette\_score = silhouette(K0, clusters)

results.writelines(f"spectral clustering k={k}\n")

results.writelines(f"silhouette score: {silhouette\_score}\n")

results.writelines("\n")

Kp = kernel.buildKernel(D, "poly")

results.writelines("poly kernel\n")

sub\_matrices = kernel.extractSubmatrices(Kp, class\_sizes, class\_keys)

[

kernel.descriptiveStats(kernel\_mat, class\_label, results)

for class\_label, kernel\_mat in sub\_matrices.items()

]

for k in range(2, 5):

clusters = spectral.spectralClustering(Kp, k)

silhouette\_score = silhouette(Kp, clusters)

results.writelines(f"spectral clustering k={k}\n")

results.writelines(f"silhouette score: {silhouette\_score}\n")

results.writelines("\n")

"""part 2"""

results.writelines("PART 2!!:\n")

# dictionary of terms across all files

D1 = vocab.dictionary(files, stem=True)

# re-vectorize with revised dictionay

vectors: List[np.ndarray] = [vectorize.vectorize(f, D1) for f in files]

# D is the Document-term matrix

D = np.vstack(vectors)

# K is the kernel matrix

K0 = kernel.buildKernel(D, "dot")

results.writelines("dot kernel\n")

sub\_matrices = kernel.extractSubmatrices(K0, class\_sizes, class\_keys)

[

kernel.descriptiveStats(kernel\_mat, class\_label, results)

for class\_label, kernel\_mat in sub\_matrices.items()

]

for k in range(2, 5):

clusters = spectral.spectralClustering(K0, k)

silhouette\_score = silhouette(K0, clusters)

results.writelines(f"spectral clustering k={k}\n")

results.writelines(f"silhouette score: {silhouette\_score}\n")

results.writelines("\n")

Kp = kernel.buildKernel(D, "poly")

results.writelines("poly kernel\n")

sub\_matrices = kernel.extractSubmatrices(Kp, class\_sizes, class\_keys)

[

kernel.descriptiveStats(kernel\_mat, class\_label, results)

for class\_label, kernel\_mat in sub\_matrices.items()

]

for k in range(2, 5):

clusters = spectral.spectralClustering(Kp, k)

silhouette\_score = silhouette(Kp, clusters)

results.writelines(f"spectral clustering k={k}\n")

results.writelines(f"silhouette score: {silhouette\_score}\n")

results.writelines("\n")

"""part 3"""

results.writelines("PART 3!!:\n")

# dictionary of terms across all files

D2 = vocab.dictionary(files, stem=True)

# re-vectorize with revised dictionay

vectors: List[np.ndarray] = [vectorize.vectorize(f, D2) for f in files]

# D is the Document-term matrix

D = np.vstack(vectors)

# K is the kernel matrix

K0 = kernel.buildKernel(D, "dot")

results.writelines("dot kernel\n")

sub\_matrices = kernel.extractSubmatrices(K0, class\_sizes, class\_keys)

[

kernel.descriptiveStats(kernel\_mat, class\_label, results)

for class\_label, kernel\_mat in sub\_matrices.items()

]

for k in range(2, 5):

clusters = spectral.spectralClustering(K0, k)

silhouette\_score = silhouette(K0, clusters)

results.writelines(f"spectral clustering k={k}\n")

results.writelines(f"silhouette score: {silhouette\_score}\n")

results.writelines("\n")

Kp = kernel.buildKernel(D, "poly")

results.writelines("poly kernel\n")

sub\_matrices = kernel.extractSubmatrices(Kp, class\_sizes, class\_keys)

[

kernel.descriptiveStats(kernel\_mat, class\_label, results)

for class\_label, kernel\_mat in sub\_matrices.items()

]

for k in range(2, 5):

clusters = spectral.spectralClustering(Kp, k)

silhouette\_score = silhouette(Kp, clusters)

results.writelines(f"spectral clustering k={k}\n")

results.writelines(f"silhouette score: {silhouette\_score}\n")

results.writelines("\n")

results.close()

“””kernel.py”””

import numpy as np

from collections import Counter

from typing import List, Literal, TextIO

def buildKernel(D: np.ndarray, type: Literal["dot", "poly"]):

if type == "dot":

return np.matmul(D, D.T)

else:

return (np.matmul(D, D.T) + 1) \*\* 2

def extractSubmatrices(

K: np.ndarray, class\_sizes: Counter, class\_keys: List[int]

) -> np.ndarray:

starts = [0]

for key in class\_keys[:-1]:

starts.append(starts[-1] + class\_sizes[key])

sub\_matrices = {

key: K[

starts[i]: starts[i] + class\_sizes[key],

starts[i]: starts[i] + class\_sizes[key],

]

for i, key in enumerate(class\_keys)

}

return sub\_matrices

def descriptiveStats(kernel\_mat: np.ndarray, class\_label: str, results: TextIO) -> None:

mean\_value = np.mean(kernel\_mat)

median\_value = np.median(kernel\_mat)

std\_deviation = np.std(kernel\_mat)

min\_value = np.min(kernel\_mat)

max\_value = np.max(kernel\_mat)

results.writelines(f"class\_matrix: {class\_label}\n")

results.writelines(f"Mean: {mean\_value}\n")

results.writelines(f"Median: {median\_value}\n")

results.writelines(f"Standard Deviation: {std\_deviation}\n")

results.writelines(f"Minimum Value: {min\_value}\n")

results.writelines(f"Maximum Value: {max\_value}\n")

results.writelines("\n")

“””spectral.py”””

import numpy as np

class Graph:

def \_\_init\_\_(self, K: np.ndarray):

self.W = self.similairyGraph(K)

def similairyGraph(self, K: np.ndarray, threshold: float = 0.5):

"""

Create a similarity graph from a kernel matrix

"""

distances = K.copy()

d\_max = distances.max()

distances = d\_max - distances

distances[distances < threshold \* d\_max] = 0

return distances

def spectralClustering(W: np.ndarray, k: int):

"""

Spectral Clustering

"""

sim\_graph = Graph(W)

W = sim\_graph.W

L = np.diag(W.sum(axis=1)) - W

# compute the eigenvalues and eigenvectors

eigvals, eigvecs = np.linalg.eigh(L)

U = eigvecs[:, :k]

clusters = kmeans(U, k)

return clusters

def kmeans(X: np.ndarray, k: int):

"""

K-means

"""

centroids = X[np.random.choice(X.shape[0], k, replace=False)]

prev\_centroids = np.zeros(centroids.shape)

clusters = np.zeros(X.shape[0])

error = np.linalg.norm(centroids - prev\_centroids)

while error != 0:

for i in range(X.shape[0]):

distances = np.linalg.norm(X[i] - centroids, axis=1)

clusters[i] = np.argmin(distances)

prev\_centroids = centroids

for i in range(k):

centroids[i] = np.mean(X[clusters == i], axis=0)

error = np.linalg.norm(centroids - prev\_centroids)

return clusters

“””spectral.py”””

import numpy as np

class Graph:

def \_\_init\_\_(self, K: np.ndarray):

self.W = self.similairyGraph(K)

def similairyGraph(self, K: np.ndarray, threshold: float = 0.5):

"""

Create a similarity graph from a kernel matrix

"""

distances = K.copy()

d\_max = distances.max()

distances = d\_max - distances

distances[distances < threshold \* d\_max] = 0

return distances

def spectralClustering(W: np.ndarray, k: int):

"""

Spectral Clustering

"""

sim\_graph = Graph(W)

W = sim\_graph.W

L = np.diag(W.sum(axis=1)) - W

# compute the eigenvalues and eigenvectors

eigvals, eigvecs = np.linalg.eigh(L)

U = eigvecs[:, :k]

clusters = kmeans(U, k)

return clusters

def kmeans(X: np.ndarray, k: int):

"""

K-means

"""

centroids = X[np.random.choice(X.shape[0], k, replace=False)]

prev\_centroids = np.zeros(centroids.shape)

clusters = np.zeros(X.shape[0])

error = np.linalg.norm(centroids - prev\_centroids)

while error != 0:

for i in range(X.shape[0]):

distances = np.linalg.norm(X[i] - centroids, axis=1)

clusters[i] = np.argmin(distances)

prev\_centroids = centroids

for i in range(k):

centroids[i] = np.mean(X[clusters == i], axis=0)

error = np.linalg.norm(centroids - prev\_centroids)

return clusters

“””vectorize.py”””

from collections import Counter

import numpy as np

from typing import List

import re

def classCounts(files: List[str]) -> Counter:

# Regular expression pattern to extract class information

pattern = r"(\d+)-"

# Use list comprehension to extract class information from each file name

class\_numbers = [int(re.search(pattern, file\_name).group(1))

for file\_name in files]

return Counter(class\_numbers)

def vectorize(filename: str, dict: List[str]) -> np.ndarray:

f = open(filename)

content = f.read().split()

vector = [content.count(word) for word in dict]

np\_vector = np.array(vector, dtype=np.float64)

return np\_vector

“””vocab.py”””

from typing import List, Optional

import numpy as np

import vectorize

def part2\_stem(vocab: List[str]):

add = [term for term in vocab if "add" in term]

sub = [term for term in vocab if "sub" in term]

mul = [term for term in vocab if "mul" in term]

div = [term for term in vocab if "div" in term]

jump = [term for term in vocab if term.startswith("j")]

push = [term for term in vocab if "push" in term]

mov = [term for term in vocab if "mov" in term]

num = [term for term in vocab if term.isnumeric()]

new\_vocab = {}

new\_vocab.update(dict.fromkeys(add, "add"))

new\_vocab.update(dict.fromkeys(sub, "sub"))

new\_vocab.update(dict.fromkeys(mul, "mul"))

new\_vocab.update(dict.fromkeys(div, "div"))

new\_vocab.update(dict.fromkeys(jump, "jump"))

new\_vocab.update(dict.fromkeys(push, "push"))

new\_vocab.update(dict.fromkeys(mov, "mov"))

new\_vocab.update(dict.fromkeys(num, "num"))

return new\_vocab

def part3\_stem(vocab: List[str]):

arith = [

term for term in vocab if any(s in term for s in ["add", "sub", "mul", "div"])

]

jump = [term for term in vocab if term.startswith("j")]

data = [term for term in vocab if any(s in term for s in ["mov", "push"])]

num = [term for term in vocab if term.isnumeric()]

new\_vocab = {}

new\_vocab.update(dict.fromkeys(num, "num"))

new\_vocab.update(dict.fromkeys(arith, "arith"))

new\_vocab.update(dict.fromkeys(jump, "jump"))

new\_vocab.update(dict.fromkeys(data, "data"))

return new\_vocab

def dictionary(

files: List[str],

stem: Optional[bool] = False,

adv\_stem: Optional[bool] = False,

):

assert type(stem) in [bool, False], "stem must be boolean"

assert type(adv\_stem) in [bool, False], "adv\_stem must be boolean"

# dictionary of terms across all files

vocab = set()

[vocab := vocab.union(set(open(f).read().split())) for f in files]

vocab = list(vocab)

vectors: List[np.ndarray] = [vectorize.vectorize(f, vocab) for f in files]

# revise dictionary

stop\_indexes = findStopWords(vectors)

vocab = [vocab[i] for i in range(len(vocab)) if i not in stop\_indexes]

if stem:

# TODO: implement stemming for part 2

vocab = part2\_stem(vocab)

elif adv\_stem:

# TODO: implement stemming for part 3

vocab = part3\_stem(vocab)

return vocab

def findStopWords(vectors: List[np.ndarray]) -> list:

"""returns the index of the stop words"""

top\_ten\_sets = [set(np.argsort(v)[:5]) for v in vectors]

common\_indexes = set.intersection(\*top\_ten\_sets)

return list(common\_indexes)