import numpy as np

from scipy.special import gammaln, psi, polygamma

import sys

from sam.pickle\_file\_io import PickleFileIO

from sam.math\_util import \*

import sam.optimize as optimize

import sam.log as log

class VEMModel(PickleFileIO):

def \_\_init\_\_(self, reader=None, T=None):

assert reader is not None

self.T = T if T is not None else 10 # Number of topics

self.iteration = 0 # Number of iterations completed

# Set up a reader to access the corpus. This gives the data dimensionality (D), the number of documents,

# the number of data, the number of document classes, and the document labels (as well as whether a

# document label was observed).

self.reader = reader

self.corpus\_file = self.reader.filename

self.V = self.reader.dim # Vocab size

self.D = self.reader.num\_docs

self.num\_docs = self.reader.num\_docs

# For efficiency, read the corpus into memory

self.load\_corpus\_as\_matrix()

# Variational parameters

self.alpha = np.ones(self.T) \* 1.0 + 1.0

self.m = l2\_normalize(np.ones(self.V)) # Parameter to p(mu)

self.kappa0 = 10.0

self.kappa1 = 5000.0

self.xi = 5000.0

self.vm = l2\_normalize(np.random.rand(self.V))

self.vmu = l2\_normalize(np.random.rand(self.V, self.T))

# Initialize vAlpha

self.valpha = np.empty((self.T, self.num\_docs))

for d in range(self.num\_docs):

distances\_from\_topics = np.abs(cosine\_similarity(self.v[:, d], self.vmu)) + 0.01

self.valpha[:, d] = distances\_from\_topics / sum(distances\_from\_topics) \* 3.0

def \_\_setstate\_\_(self, state):

self.\_\_dict\_\_.update(state)

# Reset corpus\_file just in case the reader loads the corpus from a different directory

self.corpus\_file = self.reader.filename

self.load\_corpus\_as\_matrix()

def \_\_getstate\_\_(self):

state = self.\_\_dict\_\_.copy()

del state['v']

return state

def load\_corpus\_as\_matrix(self):

self.v = np.empty((self.V, self.num\_docs))

for d in xrange(self.num\_docs):

self.v[:, d] = self.reader.read\_doc(d).T

def l\_valpha(self):

alpha0 = np.sum(self.alpha)

psi\_valpha = psi(self.valpha)

valpha0s = self.valpha.sum(axis=0)

psi\_valpha0s = psi(valpha0s)

sum\_of\_rhos = np.sum(self.rho\_batch())

# (alpha\_minus\_one\_matrix \* psi\_valpha).sum() \

like = np.dot(asrowvector(self.alpha - 1.0), psi\_valpha).sum() \

- (alpha0 - self.T)\*psi\_valpha0s.sum() \

+ self.D\*gammaln(alpha0) \

- self.D\*gammaln(self.alpha).sum() \

+ self.kappa1 \* sum\_of\_rhos \

- np.sum((self.valpha - 1.0) \* psi\_valpha) \

+ np.sum(psi\_valpha0s \* (valpha0s - self.T)) \

- np.sum(gammaln(valpha0s)) \

+ np.sum(gammaln(self.valpha))

return like

def l\_alpha(self):

alpha0 = np.sum(self.alpha)

psi\_valpha = psi(self.valpha)

psi\_valpha0s = psi(np.sum(self.valpha, axis=0))

likelihood = np.sum( ascolvector(self.alpha - 1) \* psi\_valpha ) \

- (alpha0 - self.T)\*np.sum(psi\_valpha0s) \

+ self.D\*gammaln(alpha0) \

- self.D\*np.sum(gammaln(self.alpha))

return likelihood

def l\_vmu(self):

a\_xi = avk(self.V, self.xi)

a\_k0 = avk(self.V, self.kappa0)

sum\_of\_rhos = sum(self.rho\_batch())

vm\_dot\_sum\_of\_vmu = np.dot(self.vm.T, np.sum(self.vmu, axis=1))

likelihood = a\_xi\*a\_k0\*self.xi\*vm\_dot\_sum\_of\_vmu + self.kappa1\*sum\_of\_rhos

return likelihood

def l\_xi(self):

a\_xi = avk(self.V, self.xi)

a\_k0 = avk(self.V, self.kappa0)

sum\_of\_rhos = sum(self.rho\_batch())

return a\_xi\*self.xi \* (a\_k0\*np.dot(self.vm.T, np.sum(self.vmu, axis=1)) - self.T) \

+ self.kappa1\*sum\_of\_rhos

def grad\_l\_valpha(self):

alpha0 = np.sum(self.alpha)

valpha0s = np.sum(self.valpha, axis=0)

grad\_psi\_valpha = polygamma(1, self.valpha)

grad\_psi\_valpha0s = polygamma(1, valpha0s)

grads\_of\_rho = self.grad\_rho\_valpha\_batch()

grad = ascolvector(self.alpha - 1.0) \* grad\_psi\_valpha \

+ self.kappa1\*grads\_of\_rho \

- (self.valpha - 1)\*grad\_psi\_valpha

addToEachRow = -grad\_psi\_valpha0s\*(alpha0 - self.T) \

+ grad\_psi\_valpha0s\*(valpha0s - self.T)

grad += asrowvector(addToEachRow)

return grad

def grad\_l\_vmu(self):

#avk = mean resultant length VMF

a\_xi = avk(self.V, self.xi)

a\_xi\_squared = a\_xi\*\*2

a\_k0 = avk(self.V, self.kappa0)

esns = self.e\_squared\_norm\_batch()

valpha0s = np.sum(self.valpha, axis=0)

# For single d: aXi/vAlphaD0 /sqrt(esn) \* vd \* vAlphaD'

first\_term = np.dot(self.v, (self.valpha \* a\_xi / asrowvector(valpha0s \* np.sqrt(esns))).T)

# Weights per document for everything that was in GradESN; from second term in GradRhoVMu\_BatchT

# For a single d: aXi/vAlphaD0 / (2\*esn^(3/2)) \* dot(model.vMu\*vAlphaD, vd)

per\_doc\_weights = a\_xi / valpha0s / (2\*esns \*\* (3./2.)) \

\* (self.valpha \* np.dot(self.vmu.T, self.v)).sum(axis=0).T

second\_term\_doc\_weights = 2\*(1-a\_xi\_squared) / (valpha0s\*(valpha0s+1)) # Same dim as valpha0s

second\_term = np.sum(per\_doc\_weights \* second\_term\_doc\_weights) \* self.vmu

# Last term in GradESN times per-doc factors from GradRhoVMu...

third\_term\_doc\_weights = per\_doc\_weights \* 2\*a\_xi\_squared / (valpha0s\*(valpha0s+1)) # From GradESN

# Instead of

#rescaled\_valphas = self.valpha \* asrowvector(np.sqrt(third\_term\_doc\_weights))

#third\_term = np.dot(self.vmu, np.dot(rescaled\_valphas, rescaled\_valphas.T))

third\_term = np.dot(self.vmu, np.dot(self.valpha \* asrowvector(third\_term\_doc\_weights), self.valpha.T))

sum\_over\_documents = first\_term - second\_term - third\_term

return ascolvector(a\_xi\*a\_k0\*self.xi\*self.vm) + self.kappa1\*sum\_over\_documents

def grad\_l\_alpha(self):

alpha0 = np.sum(self.alpha)

valpha0s = np.sum(self.valpha, axis=0)

return np.sum(psi(self.valpha), axis=1) - np.sum(psi(valpha0s)) \

+ self.D\*psi(alpha0) - self.D\*psi(self.alpha)

def grad\_l\_xi(self):

a\_xi = avk(self.V, self.xi)

a\_prime\_xi = deriv\_avk(self.V, self.xi)

a\_k0 = avk(self.V, self.kappa0)

sum\_over\_documents = sum(self.deriv\_rho\_xi())

# (a\_k0\*np.dot(self.vm.T, np.sum(self.vmu, axis=1)) - self.T)

return (a\_prime\_xi\*self.xi + a\_xi) \* (a\_k0\*np.dot(self.vm.T, np.sum(self.vmu, axis=1)) - self.T) \

+ self.kappa1\*sum\_over\_documents

def tangent\_grad\_l\_vmu(self):

"""

The gradient of the likelihood bound with respect to vMu, projected into the tangent space of the hypersphere.

"""

grad = self.grad\_l\_vmu()

# Project the gradients into the tangent space at each topic

for t in range(self.T):

vmu\_t = self.vmu[:, t]

grad[:, t] = grad[:, t] - np.dot(vmu\_t, np.dot(vmu\_t.T, grad[:, t]))

return grad

def rho\_batch(self):

esns = self.e\_squared\_norm\_batch()

valpha0s = self.valpha.sum(axis=0)

# vmu: V by T

# v: V by D

# vmu' \* v: T by D

vmu\_times\_v = self.vmu.T.dot(self.v)

return np.sum(self.valpha \* asrowvector(1.0/valpha0s/np.sqrt(esns)) \* vmu\_times\_v, axis=0) \

\* avk(self.V, self.xi)

def grad\_rho\_valpha\_batch(self):

valpha0s = np.sum(self.valpha, axis=0)

a\_xi = avk(self.V, self.xi)

derivsOfESquaredNorm = self.grad\_e\_squared\_norm\_batch()

esns = self.e\_squared\_norm\_batch()

vMuDotVd = np.dot(self.vmu.T, self.v) # T by D

vMuTimesVAlphaDotVd = np.sum(self.valpha \* vMuDotVd, axis=0)

grad = vMuDotVd / asrowvector(valpha0s)

# Subtract a constant from each column

grad -= asrowvector(vMuTimesVAlphaDotVd / (valpha0s\*\*2))

# Divide each column by sqrt(esns(d))

grad /= asrowvector(np.sqrt(esns))

s = vMuTimesVAlphaDotVd / valpha0s / (2\*esns\*\*(3./2.))

grad = a\_xi \* (grad - derivsOfESquaredNorm \* asrowvector(s))

return grad

def e\_squared\_norm\_batch(self):

valpha0s = self.valpha.sum(axis=0)

valpha\_squares = np.sum(self.valpha\*\*2, axis=0)

a\_xi\_squared = avk(self.V, self.xi) \*\* 2

vMuDotVMu = np.dot(self.vmu.T, self.vmu) # T by T

vMuVAlphaVMuVAlpha = np.sum(

np.dot(self.valpha.T, vMuDotVMu).T \* self.valpha,

axis=0)

esns = (valpha0s + (1.0-a\_xi\_squared)\*valpha\_squares + a\_xi\_squared\*vMuVAlphaVMuVAlpha) / (valpha0s \* (valpha0s + 1))

return esns

def grad\_e\_squared\_norm\_batch(self):

valpha0s = np.sum(self.valpha, axis=0)

a\_xi\_squared = avk(self.V, self.xi) \*\* 2

esns = self.e\_squared\_norm\_batch()

vMuTimesVAlphaTimesVMu = np.dot(self.valpha.T, np.dot(self.vmu.T, self.vmu)) # D by T

per\_doc\_weights = 1./(valpha0s \* (valpha0s + 1))

grad = (1 + 2\*(1-a\_xi\_squared)\*self.valpha + 2\*a\_xi\_squared\*vMuTimesVAlphaTimesVMu.T)

grad -= esns \* asrowvector(2\*valpha0s + 1)

grad \*= asrowvector(per\_doc\_weights)

return grad

def deriv\_rho\_xi(self):

""" Gradient of each Rho\_d with respect to xi. """

a\_xi = avk(self.V, self.xi)

deriv\_a\_xi = deriv\_avk(self.V, self.xi)

valpha0s = np.sum(self.valpha, axis=0)

esns = self.e\_squared\_norm\_batch()

deriv\_e\_squared\_norm\_xis = self.grad\_e\_squared\_norm\_xi()

vMuTimesVAlphaDotDoc = np.sum(self.valpha \* np.dot(self.vmu.T, self.v), axis=0)

deriv = deriv\_a\_xi \* vMuTimesVAlphaDotDoc / (valpha0s \* np.sqrt(esns)) \

- a\_xi/2 \* vMuTimesVAlphaDotDoc / (valpha0s \* esns\*\*1.5) \* deriv\_e\_squared\_norm\_xis

return deriv

def grad\_e\_squared\_norm\_xi(self):

""" Gradient of the expectation of the squared norms with respect to xi """

a\_xi = avk(self.V, self.xi)

deriv\_a\_xi = deriv\_avk(self.V, self.xi)

valpha0s = np.sum(self.valpha, axis=0)

sum\_valphas\_squared = np.sum(self.valpha\*\*2, axis=0)

vMuVAlphaVMuVAlpha = np.sum(np.dot(self.valpha.T, np.dot(self.vmu.T, self.vmu)).T \* self.valpha, axis=0)

deriv = 2\*a\_xi\*deriv\_a\_xi\*(vMuVAlphaVMuVAlpha - sum\_valphas\_squared) / (valpha0s \* (valpha0s + 1))

return deriv

def update\_vm(self):

self.vm = l2\_normalize(

self.kappa0\*self.m + avk(self.V, self.xi)\*self.xi\*np.sum(self.vmu, axis=1)

)

def update\_m(self):

self.m = l2\_normalize(np.sum(self.vmu, axis=1)) # Sum across topics

def update\_valpha(self):

optimize.optimize\_parameter(self, 'valpha', self.l\_valpha, self.grad\_l\_valpha)

def update\_alpha(self):

optimize.optimize\_parameter(self, 'alpha', self.l\_alpha, self.grad\_l\_alpha)

def update\_xi(self):

optimize.optimize\_parameter(self, 'xi', self.l\_xi, self.grad\_l\_xi)

def update\_vmu(self):

# XXX: The topics (vmus) must lie on the hypersphere, i.e. have unit L2 norm. I'm not sure if scipy has

# an optimization method that can accommodate this type of constraint, so instead, I'm encoding

# it here a Lagrange multiplier. This should at least push the optimizer towards solutions close to the

# L2 constraint.

# Set the strength of the Lagrange multipler to something much larger than the objective

LAMBDA = 10.0\*self.l\_vmu()

def f():

squared\_norms = np.sum(self.vmu \*\* 2, axis=0)

return self.l\_vmu() - LAMBDA\*np.sum((squared\_norms - 1.0)\*\*2)

def g():

squared\_norms = np.sum(self.vmu \*\* 2, axis=0)

return self.tangent\_grad\_l\_vmu() - LAMBDA\*2.0\*(squared\_norms - 1.0)\*(2.0\*self.vmu)

optimize.optimize\_parameter(self, 'vmu', f, g, bounds=(-1.0, 1.0))

self.vmu = l2\_normalize(self.vmu) # Renormalize

def run\_one\_iteration(self):

# theta - vector of topic proportions for a particular document

# phi - topics -- size vocab-1; we have k topics

# E - expectation step - how likely is each point to fit this distribution

# variational alpha - parameter vector for Dirichlet document-level topics distribution

# of topics x # of documents

log.debug('Updating vAlpha')

self.update\_valpha()

# variational mu - global topic distribution parameter - variational topic means

log.debug('Updating vMu')

self.update\_vmu()

# variational M - global topic distribution parameter - variational topic means

log.debug('Updating vM')

self.update\_vm()

# M - maximization step - fix the distributions to better match expectations

log.debug('Updating M')

self.update\_m()

# xi - global topic distribution parameter - variational topic means

log.debug('Updating xi')

self.update\_xi()

# vector dimension = topics

log.debug('Updating alpha')

self.update\_alpha()

self.iteration += 1

def write\_topics(self, f=None, num\_top\_words=10, num\_bottom\_words=10):

if f is None:

f = sys.stdout

wordlist = open(self.corpus\_file + '.wordlist').readlines() # TODO: not hardcode this?

wordlist = np.array([each.strip() for each in wordlist], str)

for t in range(self.T):

print >>f, 'Topic %d' % t

print >>f, '--------'

sorted\_indices = np.argsort(self.vmu[:, t])

sorted\_weights = self.vmu[sorted\_indices, t]

sorted\_words = wordlist[sorted\_indices]

print >>f, 'Top weighted words:'

for word, weight in zip(sorted\_words[:-num\_top\_words:-1], sorted\_weights[:-num\_top\_words:-1]):

print >>f, ' %.4f %s' % (weight, word)

print >>f

print >>f, 'Bottom weighted words:'

for word, weight in zip(sorted\_words[:num\_bottom\_words], sorted\_weights[:num\_bottom\_words]):

print >>f, ' %.4f %s' % (weight, word)

print >>f

print >>f

if f is not sys.stdout:

f.close()

def write\_topic\_weights\_arff(self, f=None):

if f is None:

f = sys.stdout

mean\_topic\_weights = self.valpha / asrxowvector(np.sum(self.valpha, axis=0))

print >>f, '@RELATION topicWeights'

for t in range(self.T):

print >>f, '@ATTRIBUTE topic%d NUMERIC' % t

print >>f, '@ATTRIBUTE class {%s}' % ','.join(self.reader.class\_names)

print >>f, '@DATA'

for d in range(self.num\_docs):

weights\_string = ', '.join([str(each) for each in mean\_topic\_weights[:, d]])

label = self.reader.raw\_labels[d]

print >>f, '%s, %s' % (weights\_string, label)

if f is not sys.stdout:

f.close()