CS229 H.W 2 Chen Cen

1. (a) Data set a converges in 30364 iterations

Data set b never converge. This means θ change is always fluctuating, it never drops below some threshold.

(b). The reason constant learning rate works for A is because θ has low norm (~ 35) and norm stays closer on each iteration. This means the algorithm's parameter is good enough to find a local min of the cost.

It's not working for B because the learning rate is too big in later iterations, making the algorithms jumps between different points of local min loss.

1.(c) The algorithm use

$$\frac{\partial J}{\partial \theta_j} = -\frac{1}{m}\sum_{i=1}^{m} \frac{y_i \theta x_i}{1+\exp(-y^{(i)}\theta^T x^{(i)})}$$

(i) Doesn't work, the more iteration the smaller learing rate is required, otherwise the estimated θ fluctuate around different optimal local mins or saddle points.

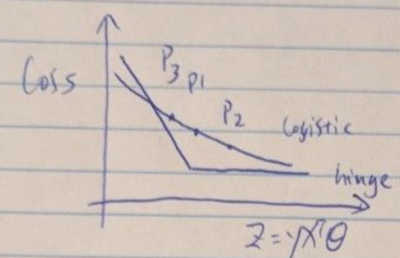
(ii) Works, converges in 27850000 iterations, it finds a local min saddle point.

(iii) works, adding the term reduces the norm of θ, which makes the algorithm converge.

(iv) Doesn't work with constant leaning rate the norm decrease ratio is the same with original X, the ~~the~~ norm delta will be fluctating around the same points.

(v) Doesn't work because the new term won't contribute to the gradient, so it has no impact on the algorithm.

1.(d)



Using hinge loss we will also face the issue, because the issue we have now is that adjusting θ won't guarantee a bigger Z, means we could jump from $P1 \to P2 \to P3$. The same issue will also affect the hinge loss line.

2.(a) Let $L(\theta) = \prod_{i=1}^{m} p(y^{(i)} | x^{(i)}; \theta)$

We can write the loglikelihood of θ as

$$\ell(\theta) = \log L(\theta) = \sum_{i=1}^{m} (y^{(i)} \log h_\theta(x) + (1-y^{(i)}) \log(1-h_\theta(x)))$$

to maximize to get θ, let it's gradient $=0$, so

$$\frac{\partial \ell}{\partial \theta_j} = \sum_{i=1}^{m} \left(y^{(i)} \cdot \frac{1}{h_\theta(x)} - (1-y^{(i)}) \cdot \frac{1}{1-h_\theta(x)}\right) \cdot \frac{\partial h_\theta}{\partial \theta}$$

$$= \sum_{i=1}^{m} \left(\frac{y^{(i)}}{h_\theta(x)} - \frac{1-y^{(i)}}{1-h_\theta(x)}\right) \cdot (1+\exp(-\theta^T x))^{-2} \cdot \exp(-\theta^T x) \cdot x_j \quad \cdots ☆$$

Let $H = h_\theta(x) = \frac{1}{1+\exp(-\theta^T x)}$, $E = \exp(-\theta^T x)$

we know $H = \frac{1}{1+E} \Rightarrow E = \frac{1-H}{H} \quad \cdots$ A

Plug A into ☆

$$\frac{\partial \ell}{\partial \theta_j} = \sum \left(\frac{y}{H} - \frac{1-y}{1-H}\right) \cdot H^2 \cdot \frac{1-H}{H} \cdot x_j$$

$$= \sum (y - yH - H + Hy) \cdot x_j$$

$$= \sum (y - H) x_j = \sum \left(y^{(i)} - \frac{1}{1+\exp(-\theta^T x)}\right) \cdot x_j$$

Let $\underline{\frac{\partial \ell}{\partial \theta_j} = 0}$ and $\underline{y^{(i)} = 1}$, considering the bias term where $\underline{\forall x_1^{(i)} = 1}$

We have $\underline{\sum_{i \in I_{a,b}} 1\{y^{(i)} = 1\} = \sum_{i \in I_{a,b}} P(y^{(i)} = 1 | x^{(i)}; \theta)}$

2. (b) If model is perfectly calibrated, it doesn't necessarily achieve perfect accuracy. E.g if the training Data has all $Y=1$, then the model would be perfectly calibrated, but it won't know the case where $Y=0$.

However the converse is true, if we can really find θ that perfectly predict any set of X, then the properties must hold.

3. $p(y^{(i)}|x;\theta) = h_\theta(x)^y \cdot (1-h_\theta(x))^{1-y}$

if $\theta \sim N(0, \tau^2 I)$, then $p(\theta) = \frac{1}{(2\pi)^{(n/2)} \cdot (\tau I)} \cdot \exp(-\frac{1}{2}\theta^T \tau^{-2} I \cdot \theta)$

We know $\ell_{ML}(\theta) = \sum_{i=1}^{m} (y \log h_\theta(x) + (1-y)\log(1-h_\theta(x)))$

we can also write $\ell_{MAP}(\theta) = \ell_{ML}(\theta) \underbrace{-\frac{n}{2}\log 2\pi - \log\tau}_{A} - \frac{1}{2\tau^2}\theta^T \cdot I \cdot \theta$

For A, its only affected by n, which is the length of θ, write it as A

So $\ell_{MAP}(\theta) = \ell_{ML}(\theta) - A - \frac{1}{2\tau^2}||\theta||_2$ --- ①

Say we have θ_{MAP} maximizes ℓ_{MAP} and θ_{ML} maximizes ℓ_{ML}

Plug both into ①

$$\ell_{MAP}(\theta_{MAP}) = \ell_{ML}(\theta_{MAP}) - A - \frac{1}{2\tau^2}||\theta_{MAP}||_2 \quad \text{--- ①}$$

$$\vee \qquad\qquad \wedge \qquad\qquad ?$$

$$\ell_{MAP}(\theta_{ML}) = \ell_{ML}(\theta_{ML}) - A - \frac{1}{2\tau^2}||\theta_{ML}||_2 \quad \text{--- ②}$$

We know ① $\underline{\ell_{ML}(\theta_{ML}) > \ell_{ML}(\theta_{MAP})}$ and ② $\underline{\ell_{MAP}(\theta_{MAP}) > \ell_{MAP}(\theta_{ML})}$

If $||\theta_{MAP}||_2 > ||\theta_{ML}||_2$, then it's impossible for ① > ②

therefor we must have $||\theta_{MAP}||_2 \leq ||\theta_{ML}||_2$

4. (a) $K(x,z) = k_1(x,z) + k_2(x,z)$ is kernel

Per Mercer's theorem, $\forall$ vector A,

we have $A^T K_1 A \geq 0$ ①

$A^T K_2 A \geq 0$ ②

Adding ① and ②

$$A^T K_1 A + A^T K_2 A \geq 0$$

So $A^T (K_1 + K_2) A \geq 0$

(b) $K(x,z) = K_1(x,z) - K_2(x,z)$ is not kernel

because $A^T K_1 A - A^T K_2 A$ is not guaranted to be non negative

e.g if $K_2 = 2K_1$, then it becomes $-A^T K_1 A \leq 0$

(c) $K(x,z) = aK_1(x,z)$ is kernel

Since a is positive scarlar,

$$A^T a K_1 A = a \cdot A^T K_1 A \geq 0$$

(d) $K(x,z) = -aK_1(x,z)$ is not kernel

if $a = 1$, then

$$aA^T K_1 A = -A^T K_1 A \leq 0$$

$k(x,z) = K_1(x,z) \cdot k_2(x,z)$

4. (e) To prove it's a kernel, expand it as the Σ form

$$k(x,z) = \sum_i \phi_i^1(x)^T \phi_i^1(z) \cdot \sum_j \phi_j^2(x)^T \phi_j^2(z)$$

$$= \sum_i \sum_j \phi_i^1(x)^T \phi_i^1(z)\, \phi_j^2(x)^T \phi_j^2(z)$$

$$= \sum_{i,j} \underbrace{\left(\phi_i^1(x)^T \phi_j^2(x)\right)}_{A} \cdot \underbrace{\left(\phi_i^1(z)^T \phi_j^2(z)\right)}_{B}$$

Where i, j is the length of ϕ^1 and ϕ^2

Note A and B are the same polynomial of x and z

that means we can define $\phi^0(x) = \phi^1(x) \cdot \phi^2(x)$

that maps x to the vector that's composed of terms of polynomial expression in A

So $k(x,z) = \phi_0(x)^T \phi_0(z)$

---

f) Similar to (e), $f(x): \mathbb{R}^n \to \mathbb{R}$ maps x_s to a polynomial expression

We can build $\phi(x)$ that's a vector containing each term in $f(x)$

So we can write $k(x,z) = \phi^T(x) \cdot \phi(z)$

proving it is a kernel

$k(x,z) = k_3(\phi(x), \phi(z))$ is a kernel

4.(g) Since k_3 is a kernel over $\mathbb{R}^d \times \mathbb{R}^d$

$$k_3(x,z) = \phi_3^T(x) \cdot \phi_3(z)$$

we know ϕ_3 is takig $\mathbb{R}^d$ as input and out put a polynomial expression

A

So $k(x,z) = k_3(\phi(x), \phi(z))$

$$= \phi_3^T(\phi(x)) \cdot \phi_3(\phi(z))$$

we know $\phi(x): \mathbb{R}^n \to \mathbb{R}^d$ and $\phi_3: \mathbb{R}^d \to$ some polynomial exp.

A

So we can write

$$k(x,z) = \phi_0^T(x) \cdot \phi_0(z)$$

where ϕ_0 ~~[illegible]~~ $: \mathbb{R}^n \to$ A

So $k(x,z)$ is a kernel

5. (a) we can implicitly define $\theta^{(i)}$ in $x^{(i)}$'s polynomial expression

Note since $\theta^{(i+1)} := \theta^{i} + \underbrace{\alpha 1\{g(\theta^{(i)T}\phi x^{(i+1)}))y^{(i+1)} < 0\} y^{(i+1)}}_{\boxed{A}} \underbrace{\phi(x^{(i+1)})}_{\boxed{B}}$

We know

$$\theta^{(i+1)} = \theta^{(0)} + a_1\phi(x^{(1)}) + a_2\phi(x^{(2)}) + \cdots a_i\phi(x^{(i)}) = \boxed{\sum_{j=1}^{i} a_j\phi(x^{(j)})}$$

$a_i = 1$ or -1 or 0 depending on correctness of $g(\theta^{(i)T}\phi(x^{(i)})$

(b) now $\theta^{(i)T}\phi(x^{(i+1)}) = \sum_{j=1}^{i} a_j \underline{\phi(x^{(j)}) \cdot \phi(x^{(i+1)})}$

$$= \sum_{j=1}^{i} a_j k(x^{(j)}, x^{(i+1)})$$

(c) So we can efficiently know $\mathrm{sign}(g(\theta^{(i)T}\phi(x^{(i+1)}))y^{(i+1)})$ (part $\boxed{A}$ from above)

then decide if we need to calculat $\boxed{B}$ on each iteration

Some explaination of my code: (Final error is 0.0262 ~~0.0260~~)

6.(a) For Laplace smoothing,

$$\phi_{j|y=1} = \frac{\sum_{i=1}^{m} 1\{x_j^{(i)}=1 \wedge y^{(i)}=1\}+1}{\sum_{i=1}^{m} 1\{y^{(i)}=1\}+2}$$

The goal is to calculate

phi_1 $= [\phi_{1|y=1}, \phi_{2|y=1} \cdots \phi_{n|y=1}]$

and

phi_0 $= [\phi_{1|y=0}, \phi_{2|y=0} \cdots \phi_{n|y=0}]$

Then for testing set, calculate if it has higher 1 or 0 probability

$$P(y=1|x) = \frac{(\prod_{i=1}^{n} P(x_i|y=1))\,P(y=1)}{(\prod_{i=1}^{n} P(x_i|y=1))P(y=1) + (\prod_{i=1}^{n} P(x_i|y=0))P(y=0)} \rightarrow A$$

& $P(y=0|x) = \text{- - - - - -}$

Note both $P(y=1|x)$ & $P(y=0|x)$ has same denometer A

so to compare them, we only need to compare the log of numerators

compare $\log(\prod_{i=1}^{n} P(x_i|y=1)\,P(y=1)) = \sum_i \log\phi_{i|y=1} + \log P(y=1)$

and

$\sum_i \log\phi_{i|y=0} + \log P(y=0)$

```python
from __future__ import division
import numpy as np

def readMatrix(file):
    fd = open(file, 'r')
    hdr = fd.readline()
    rows, cols = [int(s) for s in fd.readline().strip().split()]
    tokens = fd.readline().strip().split()
    matrix = np.zeros((rows, cols))
    Y = []
    for i, line in enumerate(fd):
        nums = [int(x) for x in line.strip().split()]
        Y.append(nums[0])
        kv = np.array(nums[1:])
        k = np.cumsum(kv[:-1:2])
        v = kv[1::2]
        matrix[i, k] = v
    return matrix, tokens, np.array(Y)

def nb_train(matrix, category):
    m, n = matrix.shape
    phi_1 = [None] * n
    phi_0 = [None] * n

    ones_count = np.count_nonzero(category)
    zeros_count = len(category) - ones_count

    ones_count_p = ones_count/len(category)
    zeros_count_p = 1 - ones_count_p

    # logP(y=1)
    log_ones = np.log(ones_count_p)
    # logP(y=0)
    log_zeros = np.log(zeros_count_p)

    ####################
    # use the laplace term
    # counts # of training examples where y = 1
    # counts # of training examples where y = 0
    for i in range(1, n):
        if i % 1000 == 0:
            print("training: " + str(i))
        token_i_column = matrix[:, i]

        non_zero_count_for_token_i = 0
        zeros_count_for_token_i = 0
        for j in range(1, m):
            if category[j] == 1 and token_i_column[j] != 0:
                non_zero_count_for_token_i += 1
            if category[j] == 0 and token_i_column[j] != 0:
                zeros_count_for_token_i += 1

        # +1 to avoid log(0)
        phi_0[i] = np.log(zeros_count_for_token_i + 1) - np.log(zeros_count + n)
        phi_1[i] = np.log(non_zero_count_for_token_i + 1) - np.log(ones_count + n)

    ####################
    return phi_1, phi_0, log_ones, log_zeros
```

```python
def nb_test(matrix, phi_1, phi_0, log_ones, log_zeros):
  m, n = matrix.shape
  output = np.zeros(m)
  ###################
  for i in range(1, m):
     if i % 1000 == 0:
        print("testing: " + str(i))
     token_i_row = matrix[i ,:]
     estimate_log_ones = 0
     estimate_log_zeros = 0
     for j in range(1, n):
        if token_i_row[j] > 0:
           estimate_log_ones += phi_1[j] * token_i_row[j]
           estimate_log_zeros += phi_0[j] * token_i_row[j]

     estimate_log_ones += log_ones
     estimate_log_zeros += log_zeros

     if estimate_log_zeros > estimate_log_ones:
        output[i] = 0
     else:
        output[i] = 1
  ###################
  return output

def evaluate(output, label):
  error = (output != label).sum() * 1. / len(output)
  print 'Error: %1.4f' % error

def find_bad_tokens(tokenlist, phi_1, phi_0):
  n = len(phi_1)

  output = [None] * n

  for i in range(1, n):
     # note phi are already log, so we just minus them here
     output[i] = phi_1[i] - phi_0[i]

  sorted_indices = sorted(range(n), key=lambda k: output[k], reverse=True)
  for i in range(5):
     print tokenlist[sorted_indices[i]]

def train_set(training_set_name):
  print('training: ' + str(training_set_name))
  trainMatrix, tokenlist, trainCategory = readMatrix(training_set_name)
  testMatrix, tokenlist, testCategory = readMatrix('MATRIX.TEST')

  phi_1, phi_0, log_ones, log_zeros = nb_train(trainMatrix, trainCategory)
  output = nb_test(testMatrix, phi_1, phi_0, log_ones, log_zeros)

  evaluate(output, testCategory)


  # find_bad_tokens(tokenlist, phi_1, phi_0)
  # result for MATRIX.TRAIN:
     # output:
     # spam
     # httpaddr
     # unsubscrib
     # websit
```

```python
    # lowest


def main():
  #training_datas = ['MATRIX.TRAIN']
  # output: Error: 0.0262

  training_datas = ['MATRIX.TRAIN.50', 'MATRIX.TRAIN.100', 'MATRIX.TRAIN.200', 'MATRIX.TRAIN.400',
                   'MATRIX.TRAIN.800', 'MATRIX.TRAIN.1400']
  for training_data in training_datas:
      train_set(training_data)

  # training: MATRIX.TRAIN.50
  # Error: 0.3475
  # training: MATRIX.TRAIN.100
  # Error: 0.2062
  # training: MATRIX.TRAIN.200
  # Error: 0.0725
  # training: MATRIX.TRAIN.400
  # Error: 0.0200
  # training: MATRIX.TRAIN.800
  # Error: 0.0213
  # training: MATRIX.TRAIN.1400
  # Error: 0.0238

  return

if __name__ == '__main__':
  main()
```

(b) since phi_1 and phi_0 are already logs,

we just need to minus them.

The output of first 5 words are:

spam, httpaddr, unsubscrib, websit, ~~lowest~~

(c) TRAIN.400 gives the best result.

plot attached

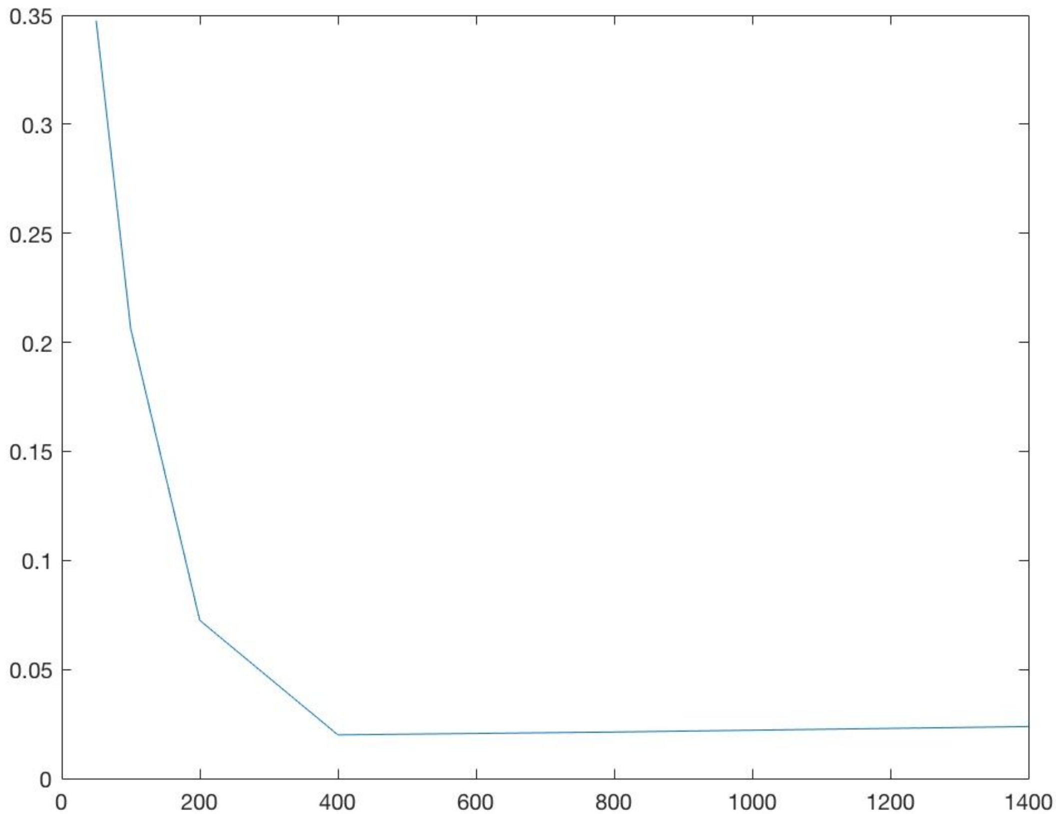
(d) plot attached

(e) compare from the 2 plots, both algorithms

achieves better results with more training data.

But anive bayes has higher "low bound" for error

than svm.

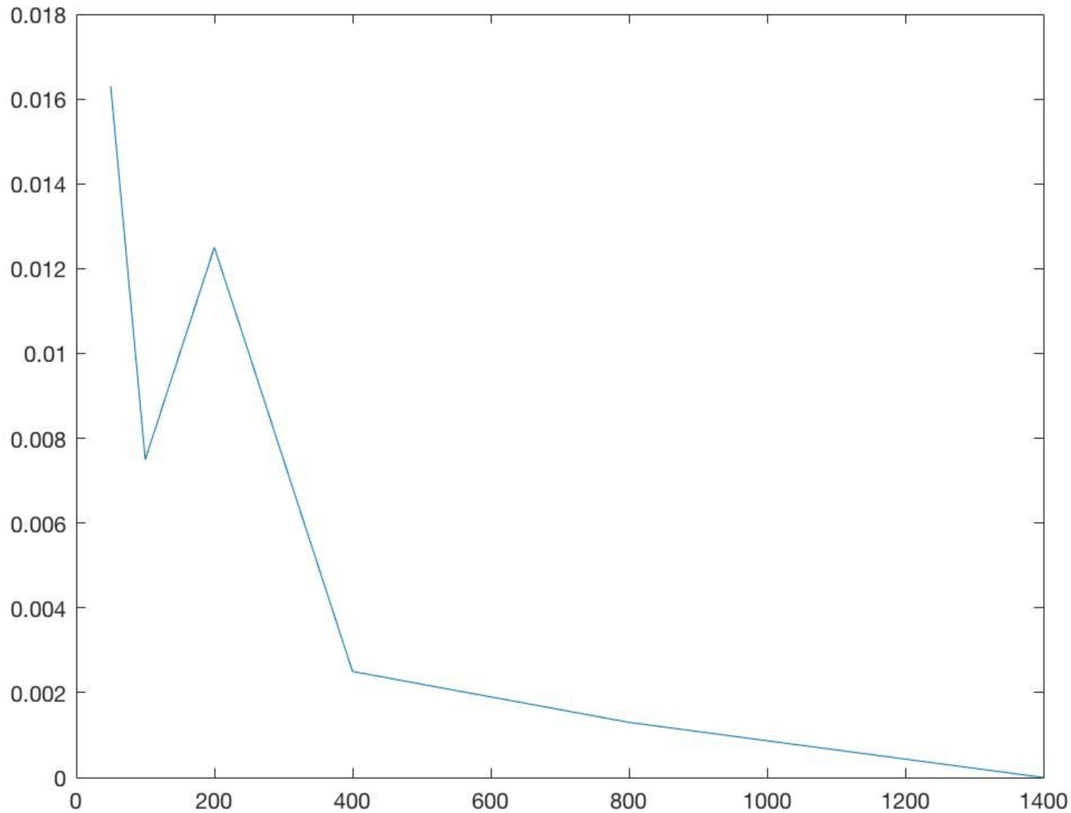


Comparison - naive bayes

```
def train_set(training_set_name):
  print("training: " + str(training_set_name))
  trainMatrix, tokenlist, trainCategory = readMatrix(training_set_name)
  testMatrix, tokenlist, testCategory = readMatrix('MATRIX.TEST')

  state = svm_train(trainMatrix, trainCategory)
  output = svm_test(testMatrix, state)
  evaluate(output, testCategory)

  # output
  # training: MATRIX.TRAIN.50
  # Error: 0.0163
  # training: MATRIX.TRAIN.100
  # Error: 0.0075
  # training: MATRIX.TRAIN.200
  # Error: 0.0125
  # training: MATRIX.TRAIN.400
  # Error: 0.0025
  # training: MATRIX.TRAIN.800
  # Error: 0.0013
  # training: MATRIX.TRAIN.1400
  # Error: 0.0000
```



Comparison - svm