ECE 50024: Homework 3
Manish Kumar Krishne Gowda, 0033682812 (Spring 2023)

Exercise 1:

(i)
In logistic regression, we aim to find a hyperplane that separates the two classes of data. This hyperplane can be represented by the equation:
w^T x + w_0= 0
The output of the logistic regression model as:
y = sigmoid(w^T x + w_0)

If the output y is greater than or equal to 0.5, we predict that x belongs to the positive class, otherwise we predict that x belongs to the negative class.
if the two classes of data are linearly separable, it means that there exists a hyperplane that perfectly separates the two classes without any misclassifications. In other words, there exists a weight vector w and an intercept w_0 such that:
w^T x_i + w_0 > 0 if x_i belongs to Class1
w^T x_i + w_0 < 0 if x_i belongs to Class0

This is equivalent to
y_i = sigmoid(w^T x_i + w_0) > 0.5 if x_i belongs to Class1
y_i = sigmoid(w^T x_i + w_0) < 0.5 if x_i belongs to Class0

Since the classes are perfectly separable, we can choose the weight vector w and the intercept w_0 such that all the above conditions hold true. when we try to maximize the likelihood function of logistic regression to find the optimal values of w and w_0. The likelihood function can be written as:

$$= \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \prod_{n=1}^{N} \left\{ h_{\boldsymbol{\theta}}(\boldsymbol{x}_n)^{y_n} (1 - h_{\boldsymbol{\theta}}(\boldsymbol{x}_n))^{1-y_n} \right\}.$$

Which is equivalent to

$$= \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \sum_{n=1}^{N} -\left\{ y_n \log h_{\boldsymbol{\theta}}(\boldsymbol{x}_n) + (1 - y_n) \log(1 - h_{\boldsymbol{\theta}}(\boldsymbol{x}_n)) \right\}$$

When the classes are perfectly separable, it turns out that there exists no global minimum of the negative log-likelihood function. This is because we can always find a solution with a lower value of the negative log-likelihood function by increasing the magnitude of the weight vector w and the intercept w_0. As we increase the magnitude of w, the hyperplane becomes steeper and steeper, which means that the distance between the positive and negative examples becomes larger and larger. As we increase the magnitude of w_0, the hyperplane shifts farther away from the origin. As we keep increasing the magnitude of w and w_0, we can always find a solution that perfectly separates the two classes of data with a lower value of the negative log-likelihood function. This is because the sigmoid function saturates as its input approaches infinity, which means that increasing

the magnitude of w and w_0 beyond a certain point does not affect the output of the sigmoid function for any input.

We also have
$y \cdot w^{T}x \geq 0$ for every data point (x, y).
We can rewrite this as $y||w||_2||x||_2 \cos\theta \geq 0$ where θ is the angle between the vectors x and w. Therefore to minimize J(θ), we want the exponent to be as negative as possible (i.e. we want $y||w||_2||x||_2 \cos\theta$ as large and positive as possible). By separability of the data we know there is some vector w such that the exponent is positive for every data point. Thus $||w||_2$ is increased without bound.

ii)
If we restrict ‖w‖2 ≤ c1 and |w0| < c2 for some c1, c2 > 0, then we are adding a constraint on the magnitude of the weight vector w and the intercept w0. This means that we are limiting the "complexity" of the logistic regression model. In this case, the optimization problem for logistic regression becomes a constrained optimization problem, where we need to find the values of w and w0 that minimize the negative log-likelihood function subject to the constraints:
‖w‖2 ≤ c1
|w0| < c2

When we add constraints on the magnitude of w and w0, it can affect the performance of the logistic regression model. If the classes are perfectly separable, then it is possible that the constraint limits the ability of the model to find the optimal solution. This is because the optimal solution may require a larger magnitude of the weight vector and/or the intercept than the constraints allow. In this case, the model may not be able to achieve zero training error, even though the classes are perfectly separable.

On the other hand, if the classes are not perfectly separable, then the constraint can help prevent overfitting by limiting the complexity of the model. In this case, the model may not be able to achieve zero training error, but it may perform better on unseen data.

In general, the choice of the values of c1 and c2 would depend on the specific problem and the available data. If we have a small amount of data, it may be better to use a larger value of c1 and a smaller value of c2 to avoid overfitting. If we have a large amount of data, we may be able to use a smaller value of c1 and a larger value of c2 without overfitting.

If the logistic regression algorithm is not converging, there are several approaches that can be taken to counter the issue:

1. **Regularization**: Regularization can help prevent overfitting and improve convergence. L1 or L2 regularization can be used to add a penalty term to the loss function, which encourages the model to have smaller weights. This can lead to a simpler model and faster convergence.
2. **Different optimization algorithm**: The standard optimization algorithm used in logistic regression is gradient descent, but sometimes this algorithm may not converge. In such cases, other optimization algorithms such as stochastic gradient descent (SGD), mini-batch SGD, or Newton's method can be used.
3. **Initialization**: The initial values of the weight vector and intercept can affect the convergence of the logistic regression algorithm. Sometimes, starting with different initial values can help improve convergence.
4. **Adding more training data**: If the algorithm is not converging, adding more training data can help by reducing the effect of noise and outliers in the data.
5. Using different loss functions: Logistic regression uses the negative log-likelihood loss function. However, in some cases, using a different loss function such as hinge loss or squared loss may help improve convergence.

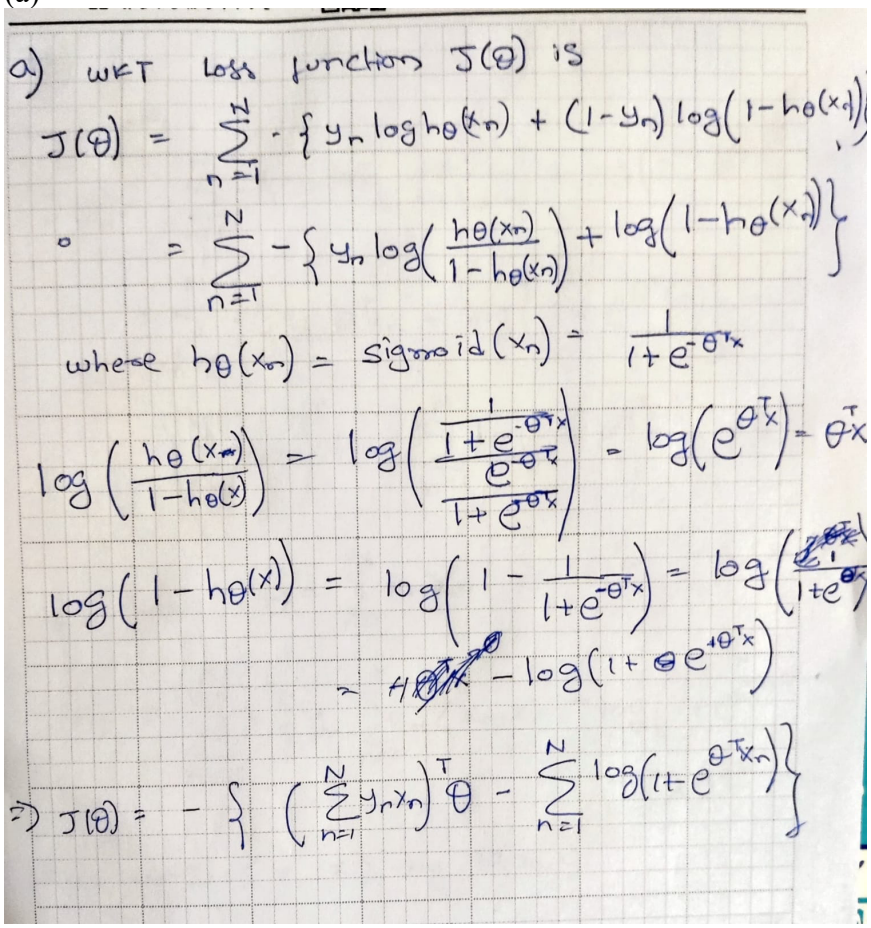
iii)

Linear separability of data may or may not cause convergence for the other linear classifiers. While the concept of linear separability is related to linear classifiers, it is not the only factor that determines the convergence of a linear classifier. Convergence depends on various factors such as the optimization algorithm, learning rate, initialization, regularization, and the complexity of the model.

For example, support vector machines (SVMs) are also linear classifiers that can handle linearly separable data. SVMs use a different loss function and optimization algorithm compared to logistic regression. Even when the data is linearly separable, it is possible for SVMs to have non-convergence issues if the hyperparameters such as the regularization parameter or kernel choice are not chosen appropriately.

Exercise 2:

(a)

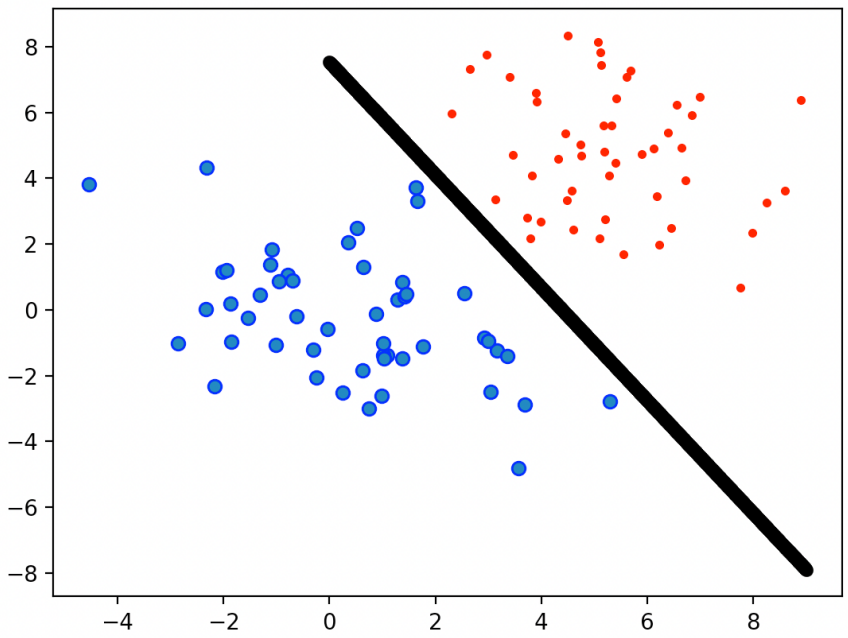


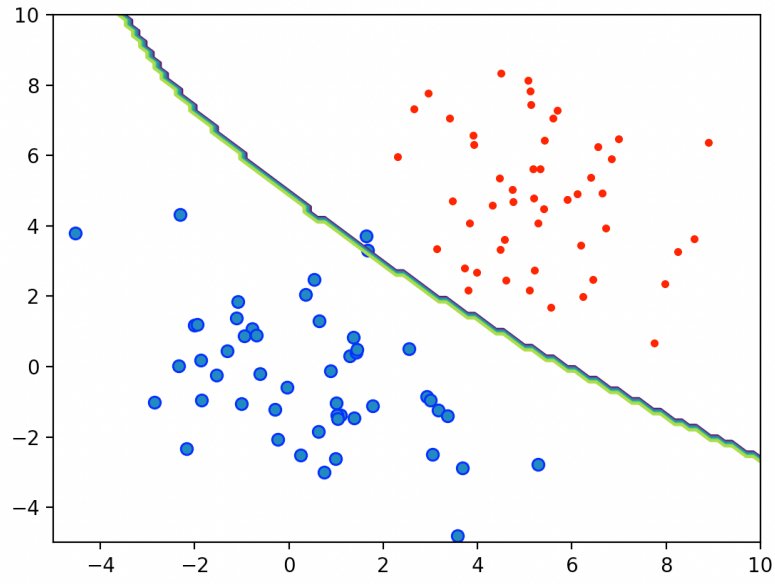
(b), (c) and (d)

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  import cvxpy as cp
4  import csv
5
6  class0 = []
7  class1 = []
8
9  # Reading csv file for male data
```

```
with open("data/homework4_class0.txt", "r") as csv_file:
  reader = csv.reader(csv_file, delimiter=' ')
  for row in reader:
    row = [i for i in row if i != '']
    class0.append(list(np.float_(row)))
  class0 = np.array(class0)
csv_file.close()
print(class0.shape)

with open("data/homework4_class1.txt", "r") as csv_file:
  reader = csv.reader(csv_file, delimiter=' ')
  for row in reader:
    row = [i for i in row if i != '']
    class1.append(list(np.float_(row)))
  class1 = np.array(class1)
csv_file.close()
print(class1.shape)

#least squares
N = class0.shape[0] + class1.shape[0]
d = 3
x = np.vstack((class0[:,0:2],class1[:,0:2]))
X = np.column_stack((x, np.ones(N)))  #consider the basis function as 1
+ x1 + x2
y =
np.vstack((np.zeros((class0.shape[0],1)),np.ones((class1.shape[0],1))))
lambd = 0.0001

theta = cp.Variable((d,1))
log_likelihood = cp.sum(cp.multiply(y, X @ theta) - cp.logistic(X @
theta))
prob = cp.Problem(cp.Maximize(log_likelihood/N - lambd * cp.norm(theta,
2)))
prob.solve()
theta_cap_cvx = theta.value
print(f"theta_cap by cvx method :\n {theta_cap_cvx}")


plt.scatter(class0[:,0], class0[:,1], edgecolor ="blue", marker ="o")
plt.scatter(class1[:,0], class1[:,1], c ="red", marker =".")
line_x = np.linspace(0, 9, 1000)
line_y = -theta_cap_cvx[2] / theta_cap_cvx[1] - theta_cap_cvx[0] /
theta_cap_cvx[1] * line_x
plt.scatter(line_x,line_y, c ="black", linewidths=0.1)
plt.show()


mu_class0 = np.mean(class0.T, axis=1)
sigma_class0 = np.cov(class0.T,bias=True)
mu_class1 = np.mean(class1.T, axis=1)
sigma_class1 = np.cov(class1.T,bias=True)
pi0 = class0.size/(class0.size+class1.size)
pi1 = class0.size/(class0.size+class1.size)


sigma_class0_inv = np.linalg.inv(sigma_class0)
```

```
sigma_class1_inv = np.linalg.inv(sigma_class1)
log_det_sigma_class0 = np.log(np.linalg.det(sigma_class0))
log_det_sigma_class1 = np.log(np.linalg.det(sigma_class1))

def dec_rule(x):
  x_class0_hat = x - mu_class0
  x_class0_hat_t = x_class0_hat.transpose()
  x_class1_hat = x - mu_class1
  x_class1_hat_t = x_class1_hat.transpose()
  param1_class0 = -0.5* x_class0_hat_t@ sigma_class0_inv @ x_class0_hat
  param1_class1 = -0.5*x_class1_hat_t @ sigma_class1_inv @ x_class1_hat
  param2_class0 = -0.5*log_det_sigma_class0
  param2_class1 = -0.5*log_det_sigma_class1
  return (param1_class0 + param2_class0 - param1_class1 -
param2_class1)


X1 = np.linspace(-5, 10, 100)
X2 = np.linspace(-5, 10, 100)
Z = np.zeros((100,100))

for i in range(0,len(X1)):
  for j in range(0,len(X2)):
      x = [X1[i], X2[j]]
      if(dec_rule(x) > 0):
        Z[i,j] = 1

print(Z)
  [X, Y] = np.meshgrid(X1, X2)
  fig, ax = plt.subplots(1, 1)
  plt.scatter(class0[:,0], class0[:,1], edgecolor ="blue", marker ="o")
  plt.scatter(class1[:,0], class1[:,1], c ="red", marker =".")
  ax.contour(X, Y, Z)
  plt.show()
```



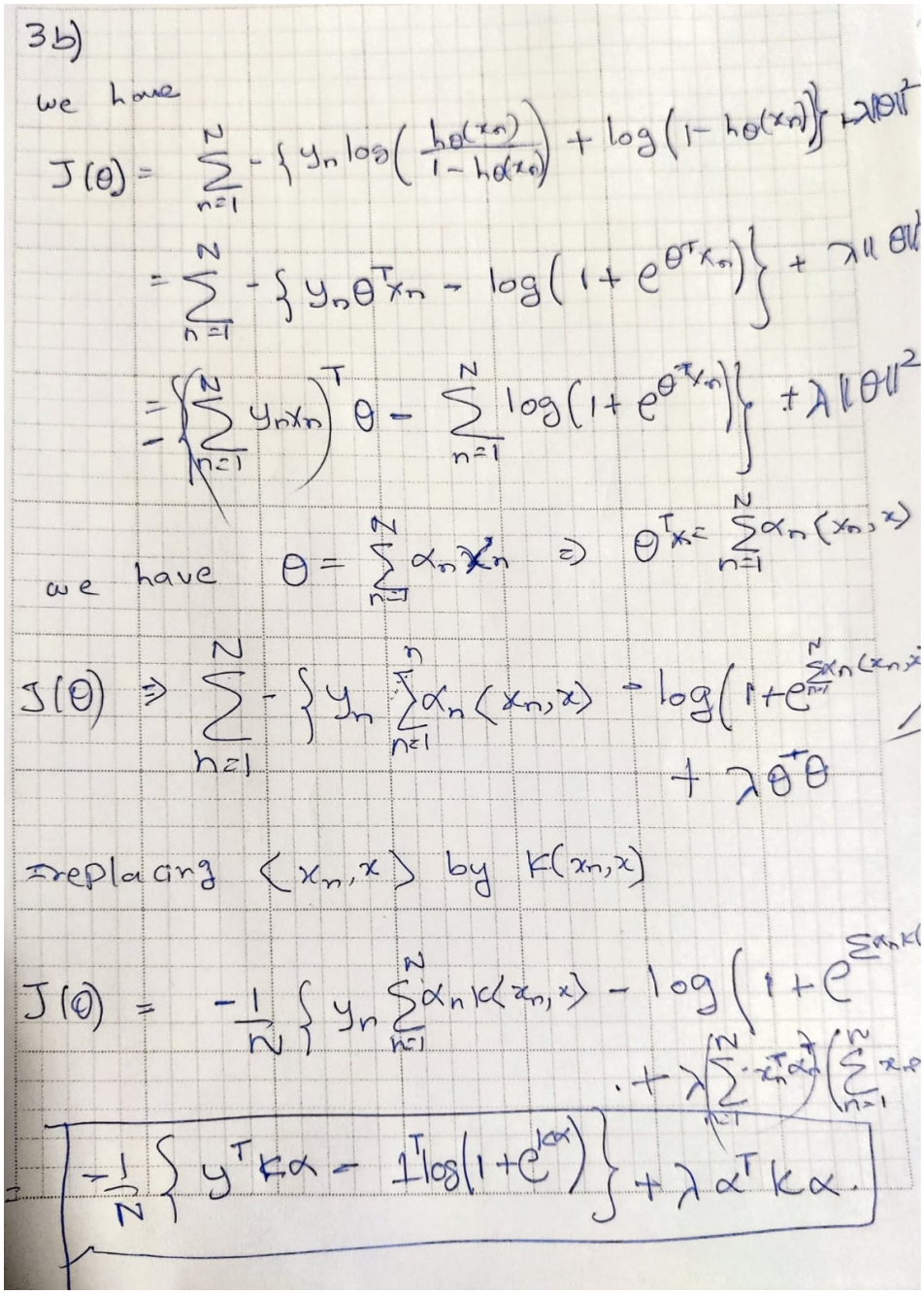


Exercise 3

a)

K[47:52,47:52] =

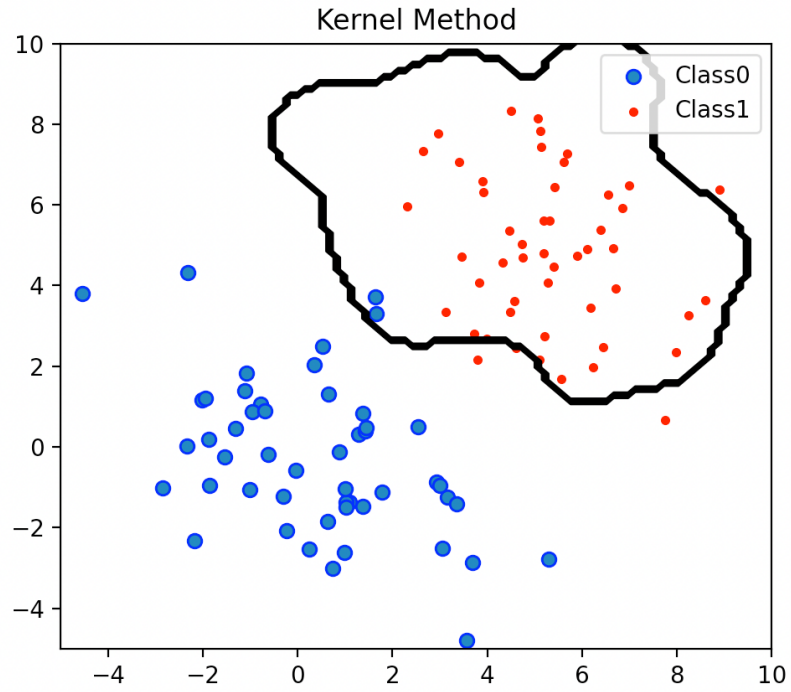
```
[[1.00000000e+00 5.64475274e-04 1.29143636e-03 3.10617742e-04
  2.14774769e-03]
 [5.64475274e-04 1.00000000e+00 4.77708365e-03 1.73395918e-04
  6.13460716e-03]
 [1.29143636e-03 4.77708365e-03 1.00000000e+00 5.03177733e-06
  1.54933077e-04]
 [3.10617742e-04 1.73395918e-04 5.03177733e-06 1.00000000e+00
  2.70068814e-02]
 [2.14774769e-03 6.13460716e-03 1.54933077e-04 2.70068814e-02
  1.00000000e+00]]
```



c)

```
first two values of alpha_cap by cvx method :
 [[-0.74267438]
 [-0.8716697 ]]
```

d)




```
import numpy as np
from numpy.matlib import repmat
import matplotlib.pyplot as plt
import cvxpy as cp
import csv

class0 = []
class1 = []

# Reading csv file for male data
with open("data/homework4_class0.txt", "r") as csv_file:
  reader = csv.reader(csv_file, delimiter=' ')
  for row in reader:
    row = [i for i in row if i != '']
    class0.append(list(np.float_(row)))
  class0 = np.array(class0)
csv_file.close()
print(class0.shape)

with open("data/homework4_class1.txt", "r") as csv_file:
  reader = csv.reader(csv_file, delimiter=' ')
  for row in reader:
    row = [i for i in row if i != '']
    class1.append(list(np.float_(row)))
  class1 = np.array(class1)
csv_file.close()
print(class1.shape)

#least squares
N = class0.shape[0] + class1.shape[0]
d = 3
x = np.vstack((class0[:,0:2],class1[:,0:2]))
X = np.column_stack((x, np.ones(N)))  #consider the basis function as 1
+ x1 + x2
```

```
y =
np.vstack((np.zeros((class0.shape[0],1)),np.ones((class1.shape[0],1))))
lambd = 0.0001
one_transpose = np.ones((1,N))

K = np.zeros((100,100))
for i in range(0,N):
  for j in range(0,N):
    K[i,j] = np.exp(-np.linalg.norm((x[i]-x[j]),2))

print(K[47:52,47:52])

alpha = cp.Variable((N,1))
log_likelihood = cp.sum(cp.multiply(y,K@alpha)) -
cp.sum(cp.log_sum_exp(cp.hstack([np.zeros((N,1)),K@alpha]),axis=1))
prob = cp.Problem(cp.Maximize(log_likelihood/N - lambd *
cp.quad_form(alpha, K)))
prob.solve()
alpha_cap_cvx = alpha.value
#print(alpha_cap_cvx)
print(f"first two values of alpha_cap by new cvx method :\n
{alpha_cap_cvx[:2]}")


X1 = np.linspace(-5, 10, 100)
X2 = np.linspace(-5, 10, 100)
Z = np.zeros((100,100))

for i in range(100):
  for j in range(100):
    data = repmat( np.array([X1[i], X2[j], 1]).reshape((1,3)), N, 1)
    s = data - X
    ks = np.exp(-np.sum(np.square(s), axis=1))
    Z[i,j] = np.dot(alpha_cap_cvx.T, ks).item()

plt.figure(figsize=(10,10))
plt.scatter(class0[:,0], class0[:,1], edgecolor ="blue", marker ="o")
plt.scatter(class1[:,0], class1[:,1], c ="red", marker =".")
plt.contour(X1, X2, Z>0.5, linewidths=1, colors='k')
plt.legend(['Class0', 'Class1'])
plt.title('Kernel Method')
plt.show()
```

Learning to Reweight Examples for Robust Deep Learning

Manish Kumar Krishne Gowda, Purdue University, ECE Department, On Campus MS Student[1]

Abstract

Deep Neural Networks easily overfit to training set biases and label noises.Regularizers and example reweighting algorithms are popular solutions to these problems. But these algorithms require careful tuning of additional hyperparameters. Traditionally, validation is performed at the end of training, which can be prohibitively expensive if the example weights are treated as some hyperparameters to optimize. To circumvent this, the paper proposes to perform validation at every training iteration to dynamically determine the example weights of the current batch. This approach significantly increases the robustness to training set biases.

1. Introduction

The paper proposes a learning algorithm that learns to assign weights to training examples based on their gradient directions. They perform gradient descent on the mini-batch example weights which are initialized to zero. Thereby they minimize the loss on clean unbiased validation set.

The model in the paper is derived from a meta-learning objective towards an online approximation that can fit into any regular supervised training. Instead of minimizing the expected loss for the training set each input example is weighted equally and this reweighted input is used to model the dataset. There are multiple examples of automatic differentiation techniques which will be the correct starting point as it is needed to compute the gradient of the validation loss. The paper itself for example references a couple of other papers which implements it using popular deep learning frameworks such as TensorFlow.

1.1. Unfamiliar Concepts

From last checkpoint I thoroughly read the paper and understood the concepts. This paper further narrows down the deep learning problem to batch learning which needs to be practically implemented.I understood concepts such as Hyperparameters, Meta-learning, Class imbalance Problem, Hard examples, Few-shot learning, Online and Offline training, Convergence rate, Backward-on-backward automatic differentiation, etc. I was also able to get a hang of the core algorithm that is proposed, i.e. "Learning to Reweight Examples using Automatic Differentiation"

2. Issues with Implementation

The core of the paper is the Algorithm 1 Learning to Reweight Examples using Automatic Differentiation in page 4. With internet support I was be able to replicate one of the two experiments mentioned in the paper i.e. MNIST data imbalance experiments. It is a computationally expensive algorithm, but I was able to replicate the results that was indicated in the dataset. The second CIFAR noisy label experiment is underway and once this is implemented the final project should be complete.

With deep neural network to be in agenda of discussion in class next week i hope to get more questions cleared.

3. Assumptions Made for Implementation

To test the effectiveness of the reweighting algorithm, i have to design both class imbalance and noisy label settings, and a combination of both, on standard MNIST and CIFAR benchmarks for image classification using deep CNNs in the same way the authors have designed. For example, the authors use the standard MNIST handwritten digit classification dataset and subsample the dataset to generate a class imbalance binary classification task. They select a total of 5,000 images of size 28×28 on class 4 and 9, where 9 dominates the training data distribution. They train a standard LeNet on this task and we compare our method with a suite of commonly used tricks for class imbalance. Assumption that the same pathway should also be followed by me while re-implementing the paper.

Further the results and graphs in the paper has to be replicated with same or similar degree of accuracy is also needed.

4. Code Structure

The main algoritm is clearly spelt out in the original paper and the same is shown in Figure 1

The summary of the algorithm is :

1. Initialize the weights for each example in the training data. The weights are initially set to be equal for all examples.
2. Train a deep learning model on the training data with the initial weights.
3. Step 3: Evaluate the performance of the model on a validation set. If the model's performance is good enough, stop training. Otherwise, go to the next step.
4. Calculate a weight update for each example in the training data. The weight update is based on how much the example contributes to the model's error on the validation set. Examples that contribute more to the error will have larger weight updates.
5. Update the weights for each example in the training data using the weight updates calculated in step 4.
6. Repeat steps 2-5 until the model's performance on the validation set is good enough or a maximum number of iterations is reached.
7. Use the final weights for each example in the training data to train a new deep learning model.

Algorithm 1 Learning to Reweight Examples using Automatic Differentiation

Require: $\theta_0, \mathcal{D}_f, \mathcal{D}_g, n, m$
Ensure: θ_T
for $t = 0$ ... $T-1$ **do**
  $\{X_f, y_f\} \leftarrow$ SampleMiniBatch($\mathcal{D}_f, n$)
  $\{X_g, y_g\} \leftarrow$ SampleMiniBatch($\mathcal{D}_g, m$)
  $\hat{y}_f \leftarrow$ Forward(X_f, y_f, θ_t)
  $\epsilon \leftarrow 0$; $l_f \leftarrow \sum_{i=1}^{n} \epsilon_i C(y_{f,i}, \hat{y}_{f,i})$
  $\nabla\theta_t \leftarrow$ BackwardAD(l_f, θ_t)
  $\hat{\theta}_t \leftarrow \theta_t - \alpha\nabla\theta_t$
  $\hat{y}_g \leftarrow$ Forward$(X_g, y_g, \hat{\theta}_t)$
  $l_g \leftarrow \frac{1}{m}\sum_{i=1}^{m} C(y_{g,i}, \hat{y}_{g,i})$
  $\nabla\epsilon \leftarrow$ BackwardAD(l_g, ϵ)
  $\tilde{w} \leftarrow \max(-\nabla\epsilon, 0)$; $w \leftarrow \frac{\tilde{w}}{\sum_j \tilde{w} + \delta(\sum_j \tilde{w})}$
  $\hat{l}_f \leftarrow \sum_{i=1}^{n} w_i C(y_i, \hat{y}_{f,i})$
  $\nabla\theta_t \leftarrow$ BackwardAD$(\hat{l}_f, \theta_t)$
  $\theta_{t+1} \leftarrow$ OptimizerStep$(\theta_t, \nabla\theta_t)$
end for

Figure 1. Algorithm

References

Langley, P. Crafting papers on machine learning. In Langley, P. (ed.), *Proceedings of the 17th International Conference on Machine Learning (ICML 2000)*, pp. 1207–1216, Stanford, CA, 2000. Morgan Kaufmann.

A. Do *not* have an appendix here

Do not put content after the references. Put anything that you might normally include after the references in a separate supplementary file.

We recommend that you build supplementary material in a separate document. If you must create one PDF and cut it up, please be careful to use a tool that doesn't alter the margins, and that doesn't aggressively rewrite the PDF file. pdftk usually works fine.

Please do not use Apple's preview to cut off supplementary material. In previous years it has altered margins, and created headaches at the camera-ready stage.