Lab 4 - ML Programming

December 3, 2021

1 EXERCISE 0

1.1 Dataset Preprocessing

[1]:
```
## Import dataset and any libraries
## Convert any non-numeric values to numeric values

import numpy as np
import pandas as pd
import csv
import matplotlib.pyplot as plt

tictactoe = pd.read_csv('tic-tac-toe.data', header=None)
tictactoe.replace('x',0,inplace=True)
tictactoe.replace('o',1,inplace=True)
tictactoe.replace('b',2,inplace=True)
tictactoe.replace('negative',0,inplace=True)
tictactoe.replace('positive',1,inplace=True)
```

[2]:
```
## Confirm that the dataset is unbalanced

## Data imbalance refers to an unequal distribution of classes within a dataset
## So to show that the data is imbalanced we need the count of positive and␣
↪negative endgames

tictactoe[9]. value_counts()
```

[2]:
```
1    626
0    332
Name: 9, dtype: int64
```

[3]:
```
## Explain what stratified sampling and implement a stratified sampler

## Stratified sampling is a method of partitioning a population into␣
↪subpopulations
## In this example we want a subpopulation which has an equal distribution␣
↪across classes
```

```python
## Pick the sampling number for each class
## Want to keep as much data as possible so make both classes around the size␣
↪of the smaller class (i.e. 330)
n0 = 330
n1 = 330
## Get indexes and lengths for the classes respectively
idx0 = tictactoe.index.values[tictactoe[9] == 0]
idx1 = tictactoe.index.values[tictactoe[9] == 1]
len0 = len(idx0)   ## 626
len1 = len(idx1)   ## 332
## Draw randomly from the indices
draw0 = np.random.permutation(len0)[:n0]
idx0 = idx0[draw0]
draw1 = np.random.permutation(len1)[:n1]
idx1 = idx1[draw1]
## Combine the drawn indexes
idx = np.hstack([idx0, idx1])
## Create new dataset
new_tictactoe = tictactoe.loc[idx, :]
## Check length of new dataset (should be 2x332)
new_tictactoe[9]. value_counts()
```

[3]:
```
0    330
1    330
Name: 9, dtype: int64
```

[4]:
```python
## Split the data into a train(80%) and test(20%)

tictactoe_train = new_tictactoe.sample(frac=0.8,random_state=42) ## random␣
↪state is just a seed value
tictactoe_test = new_tictactoe.drop(tictactoe_train.index)
```

2 EXERCISE 1

2.1 Logistic Regression w/ Gradient Descent

[5]:
```python
## Split data into features and targets
x_train = tictactoe_train.iloc[:,:-1].values
y_train = tictactoe_train.iloc[:,-1].values
x_test = tictactoe_test.iloc[:,:-1].values
y_test = tictactoe_test.iloc[:,-1].values
```

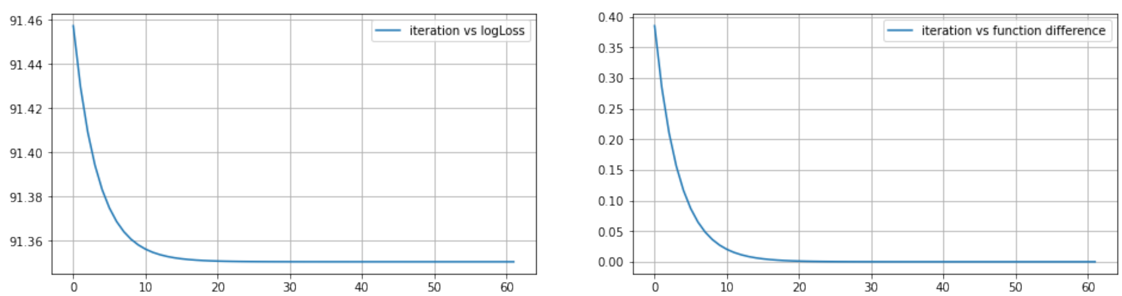
[63]:
```python
## Implement linear classification with stochastic gradient ascent algorithm
## Choose imax between 100 to 1000
## Use bolddriver as the step length controller
## In each iteration of the algorithm calculate |f(xi-1) - f(xi)| and logloss␣
↪on test set
```

```python
## Plot these against iteration number i and explain the graphs

## Code based on https://beckernick.github.io/logistic-regression-from-scratch/
def sigmoid(x):
    return 1.0/(1.0 + np.exp(-x))

def log_loss(y, p):
    return -(y * np.log(p) + (1 - y) * np.log(1 - p)).sum()

def logistic_regression(x, y, xTest, yTest, alpha, imax = 1000, precision = 0.
 ↪00000001):
    beta = np.zeros(x_train.shape[1])
    betaX = np.dot(beta, x.T)
    ## Calculating log likelihood
    log_likelihood = (y * betaX - np.log(1 + np.exp(betaX))).sum()
    ## For plotting
    iter_count = []
    absolute_dif = []
    loglosslist = []
    for i in range (0, imax):
        ## Calculate probability
        p = sigmoid(betaX)
        ## Add the gradient to maximize log likelihood
        gradient = np.dot(x.T, y - p)
        beta = beta + (alpha * gradient)
        betaX = np.dot(beta, x.T)
        log_likelihoodnew = (y * betaX - np.log(1 + np.exp(betaX))).sum()
        ## For plotting
        iter_count.append(i)
        absolute_dif.append(abs(log_likelihoodnew - log_likelihood))
        predictiontest = sigmoid(np.dot(beta, xTest.T))
        loglosslist.append(log_loss(yTest, predictiontest))
        ## Convergence condition
        if (abs(log_likelihoodnew - log_likelihood) < precision):
            return beta, iter_count, absolute_dif, loglosslist
        log_likelihood = log_likelihoodnew
        ## Bolddriver
        ## If the function values are decreasing, increase alpha. Otherwise,␣
 ↪decrease alpha
        if log_likelihoodnew < log_likelihood:
            alpha = 1.05 * alpha
        else:
            alpha = 0.75 * alpha
    return beta, iter_count, absolute_dif, loglosslist

beta, xaxis, absolute_dif, loglosstest = logistic_regression(x_train, y_train,␣
 ↪x_test, y_test, alpha=0.0001)
```

```python
fig, axs = plt.subplots(1, 2, figsize=(16, 4))
axs[0].plot(xaxis, loglosstest, label = "iteration vs logLoss")
axs[0].grid()
axs[0].legend()

axs[1].plot(xaxis, absolute_dif, label = "iteration vs function difference")
axs[1].grid()
axs[1].legend()

## The logloss is increasing which is what we want as we are trying to maximize our log
## The absolute difference as we update the betas is decreasing
```

[63]:

```
<matplotlib.legend.Legend at 0x1b0813a05b0>
```



3 EXERCISE 2

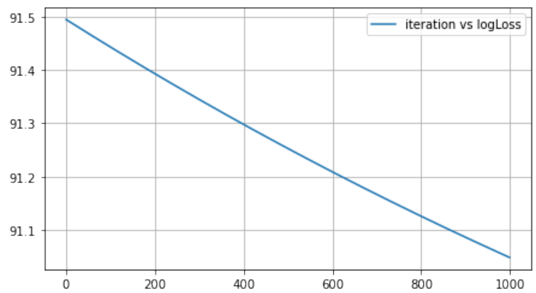
3.1 Logistic Regression w/ Newton's Algorithm

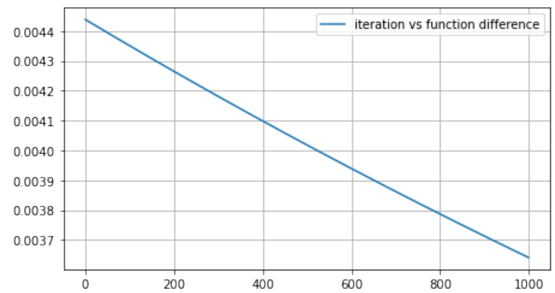
[66]:

```python
## Implement Newton Algorithm
## In each iteration calculate |f(xi-1)-f(xi)| and the logloss on test set
## Comment on the rate of convergence in the light of plots from above

def newton(x, y, xTest, yTest, alpha, imax = 1000, precision = 0.00000001):
    beta = np.zeros(x_train.shape[1])
    betaX = np.dot(beta, x.T)
    ## Calculating log likelihood
    log_likelihood = (y * betaX - np.log(1 + np.exp(betaX))).sum()
    ## For plotting
    iter_count = []
    absolute_dif = []
    loglosslist = []
    for i in range (0, imax):
        ## Calculate probability
```

```
        p = sigmoid(betaX)
        ## Make Hessian matrix
        w = np.diag(p*(1 - p))
        hessian = x.T.dot(w).dot(x)
        gradient = np.dot(x.T, y - p)
        term = np.dot(np.linalg.inv(hessian),gradient)
        beta = beta + (alpha * term)
        betaX = np.dot(beta, x.T)
        log_likelihoodnew = (y * betaX - np.log(1 + np.exp(betaX))).sum()
        ## For plotting
        iter_count.append(i)
        absolute_dif.append(abs(log_likelihoodnew - log_likelihood))
        predictiontest = sigmoid(np.dot(beta, xTest.T))
        loglosslist.append(log_loss(yTest, predictiontest))
        ## Convergence condition
        if (abs(log_likelihoodnew - log_likelihood) < precision):
            return beta, iter_count, absolute_dif, loglosslist
        log_likelihood = log_likelihoodnew
    return beta, iter_count, absolute_dif, loglosslist

beta, xaxis, absolute_dif, loglosstest = newton(x_train, y_train, x_test,␣
 ↪y_test, alpha=0.0001)

fig, axs = plt.subplots(1, 2, figsize=(16, 4))
axs[0].plot(xaxis, loglosstest, label = "iteration vs logLoss")
axs[0].grid()
axs[0].legend()

axs[1].plot(xaxis, absolute_dif, label = "iteration vs function difference")
axs[1].grid()
axs[1].legend()

## Since figure 3 tells us to use a fixed step length, we get straight lines␣
 ↪instead of curves
## However, just like the graphs using gradient ascent, we see that the␣
 ↪absolute difference decreases
## This means that the difference between the functions decreases as our betas␣
 ↪are optimized
## We can see that it is much slower in convergence here
## However, if we add in a steplength controller like bolddriver, convergence␣
 ↪is quicker with Newton
## See following cell for graphs
```

[66]: <matplotlib.legend.Legend at 0x1b083841430>





[57]:
```
## If we redo task 2 implementing the bolddriver like in task 1, we get very␣
↪similar graphs
## However, the Newton algorithm seems to converge faster than gradient ascent

def newton_bold(x, y, xTest, yTest, alpha, imax = 1000, precision = 0.00000001):
    beta = np.zeros(x_train.shape[1])
    betaX = np.dot(beta, x.T)
    ## Calculating log likelihood
    log_likelihood = (y * betaX - np.log(1 + np.exp(betaX))).sum()
    ## For plotting
    iter_count = []
    absolute_dif = []
    loglosslist = []
    for i in range (0, imax):
        ## Calculate probability
        p = sigmoid(betaX)
        ## Make Hessian matrix
        w = np.diag(p*(1 - p))
        hessian = x.T.dot(w).dot(x)
        gradient = np.dot(x.T, y - p)
        term = np.dot(np.linalg.inv(hessian),gradient)
        beta = beta + (alpha * term)
        betaX = np.dot(beta, x.T)
        log_likelihoodnew = (y * betaX - np.log(1 + np.exp(betaX))).sum()
        ## For plotting
        iter_count.append(i)
        absolute_dif.append(abs(log_likelihoodnew - log_likelihood))
        predictiontest = sigmoid(np.dot(beta, xTest.T))
        loglosslist.append(log_loss(yTest, predictiontest))
        ## Convergence condition
        if (abs(log_likelihoodnew - log_likelihood) < precision):
            return beta, iter_count, absolute_dif, loglosslist
        log_likelihood = log_likelihoodnew
        ## Bolddriver
```

```
        ## If the function values are decreasing, increase alpha. Otherwise,␣
 ↪decrease alpha
        if log_likelihoodnew < log_likelihood:
            alpha = 1.05 * alpha
        else:
            alpha = 0.75 * alpha
    return beta, iter_count, absolute_dif, loglosslist

beta, xaxis, absolute_dif, loglosstest = newton_bold(x_train, y_train, x_test,␣
 ↪y_test, alpha=0.0001)

fig, axs = plt.subplots(1, 2, figsize=(16, 4))
axs[0].plot(xaxis, loglosstest, label = "iteration vs logLoss")
axs[0].grid()
axs[0].legend()

axs[1].plot(xaxis, absolute_dif, label = "iteration vs function difference")
axs[1].grid()
axs[1].legend()
```

[57]: <matplotlib.legend.Legend at 0x1b085203100>