homework4

November 17, 2018

Edward Mattia: emattia@ucsc.edu
Weiting Zhan: wzhan83@ucsc.edu
Akila de Silva: audesilv@ucsc.edu

Resources:
https://www.kaggle.com/bibs2091/diabetes-database-analysis-and-model-choosing/notebook

Problem 1

In [189]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from time import time
from sklearn.model_selection import train_test_split
from sklearn import preprocessing
from sklearn import tree
from sklearn.ensemble import RandomForestClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import classification_report, accuracy_score
```

In [190]:
```
df = pd.read_csv('diabetes.csv')
```

In [191]:
```
df.head()
```

Out[191]:
```
   Pregnancies  Glucose  BloodPressure  SkinThickness  Insulin   BMI  \
0            6      148             72             35        0  33.6
1            1       85             66             29        0  26.6
2            8      183             64              0        0  23.3
3            1       89             66             23       94  28.1
4            0      137             40             35      168  43.1

   DiabetesPedigreeFunction  Age  Outcome
0                     0.627   50        1
1                     0.351   31        0
2                     0.672   32        1
3                     0.167   21        0
4                     2.288   33        1
```

```
In [192]: train, test = train_test_split(df, test_size=.2, random_state=7, stratify=df['Outcome
```

```
In [193]: train = train.values
          test = test.values
          min_max_scaler = preprocessing.MinMaxScaler()
          train_normalized = min_max_scaler.fit_transform(train)
          test_normalized = min_max_scaler.fit_transform(test)
```

Problem 2

```
In [194]: def sigmoid(x):
              return 1/(1+np.exp(-x))
```

```
In [195]: def sgd(X, labels, step_size, regularization=False, =0.01, print_=True):

              learning_rate = step_size
               = 1e-12 # to avoid 0 in log

              np.random.seed(1)

              converged = False
              epoch = 0
              error_rates = []
              NLLs = []
              # weights = (np.random.rand(X.shape[1])-0.5)*5e-8
              weights = np.zeros(X.shape[1])

              t0 = time()
              while not converged:
                  epoch += 1
                  epoch_errors = 0
                  epoch_NLL = 0
                  for x,t in zip(X, labels):
                      y = sigmoid(np.dot(weights,x))
                      if ((t==1) and (y<0.5)) or ((t==0) and (y>=0.5)):
                          epoch_errors+=1
                      NLL = - ((t)*np.log(y-) + (1-t)*np.log(1-y+))
                      grad = (y-t)*x
                      if regularization:
                          weights = weights - (learning_rate*grad + *weights)
                      else:
                          weights = weights - learning_rate*grad
                      epoch_NLL += NLL
                  error_rates.append(epoch_errors/X.shape[0])
                  NLLs.append(epoch_NLL)

                  # CONVERGENCE THRESHOLD
                  if epoch > 100 and np.abs(NLLs[-1] - NLLs[-100]) < NLLs[-1]*1e-5:
```

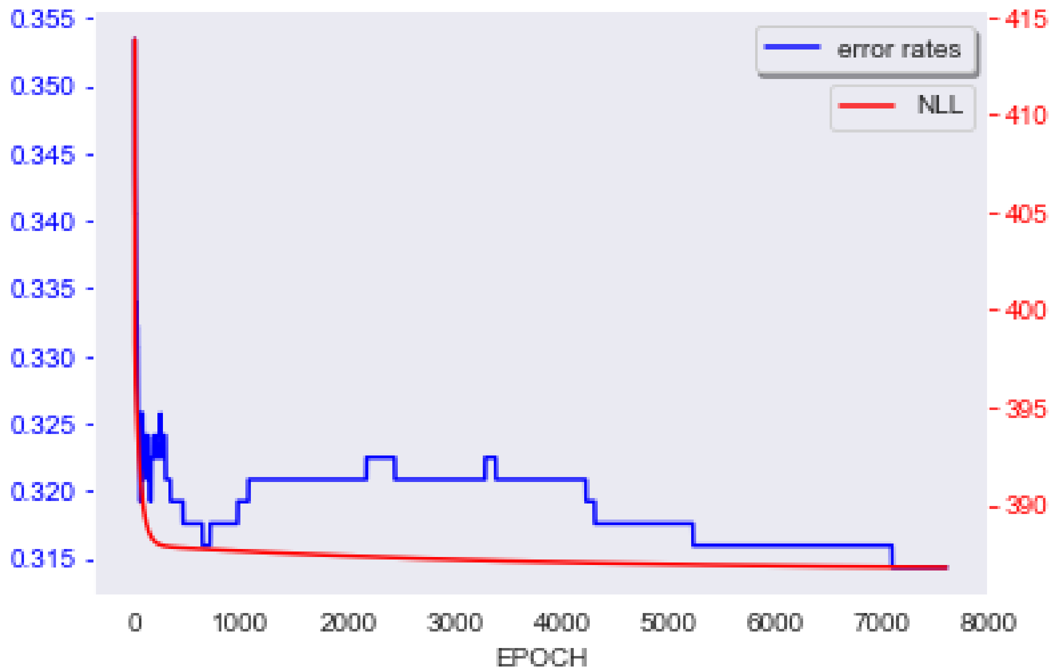
```
                converged = True

        if print_:
            print("Converged in {} epochs and {:.3f} seconds with learning rate of {}".fo
            print("Weight Values at Convergence:\n",weights)

        return weights, error_rates, NLLs
```

```
In [196]: def sgd_test(X,labels,weights, data_set):
              errors = 0
              total_NLL = 0
              for x,t in zip(X, labels):
                  y = sigmoid(np.dot(weights,x))
                  if ((t==1) and (y<=0.5)) or ((t==0) and (y>0.5)):
                      errors+=1
                  NLL = - (t*np.log(y) + (1-t)*np.log((1-y)))
                  total_NLL += NLL
              print(str(data_set)+" Set Errors: {} ({:.2f}% accuracy)".format(errors, (100-100
              print(str(data_set)+" Set NLL: {:.3f}".format(total_NLL))
```

```
In [197]: def plot_sgd(error_rates, NLLs):
              sns.set_style('dark')
              fig, ax1 = plt.subplots()
              t = np.arange(0, len(error_rates))

              ax1.plot(t, error_rates, 'b', label='error rates')
              ax1.set_xlabel('EPOCH')
              ax1.tick_params('y', colors='b')
              ax1.spines['top'].set_visible(False)
              ax1.legend(bbox_to_anchor=(1, 1), fancybox=True, shadow=True) # center

              ax2 = ax1.twinx()
              ax2.plot(t, NLLs, 'r', label='NLL')
              ax2.tick_params('y', colors='r')
              ax2.spines['top'].set_visible(False)
              ax2.legend(bbox_to_anchor= (1, 0.9), fancybox=True) # center right
              plt.show()
```

```
In [198]: learning_rate = 1e-5
          unnormalized_weights, error_rates, NLLs = sgd(train[:,:-1], train[:,-1], step_size=le
          plot_sgd(error_rates, NLLs)
```

```
Converged in 7609 epochs and 49.503 seconds with learning rate of 1e-05
Weight Values at Convergence:
 [ 0.14068024  0.01092138 -0.02877692 -0.00172508 -0.00063759 -0.00253689
  0.30257327 -0.00988826]
```




```
In [199]: sgd_test(test[:,:-1], test[:,-1], unnormalized_weights, "Test")

Test Set Errors: 52 (66.23% accuracy)
Test Set NLL: 95.768


In [200]: sgd_test(train[:,:-1], train[:,-1], unnormalized_weights, "Train")

Train Set Errors: 190 (69.06% accuracy)
Train Set NLL: 375.756
```

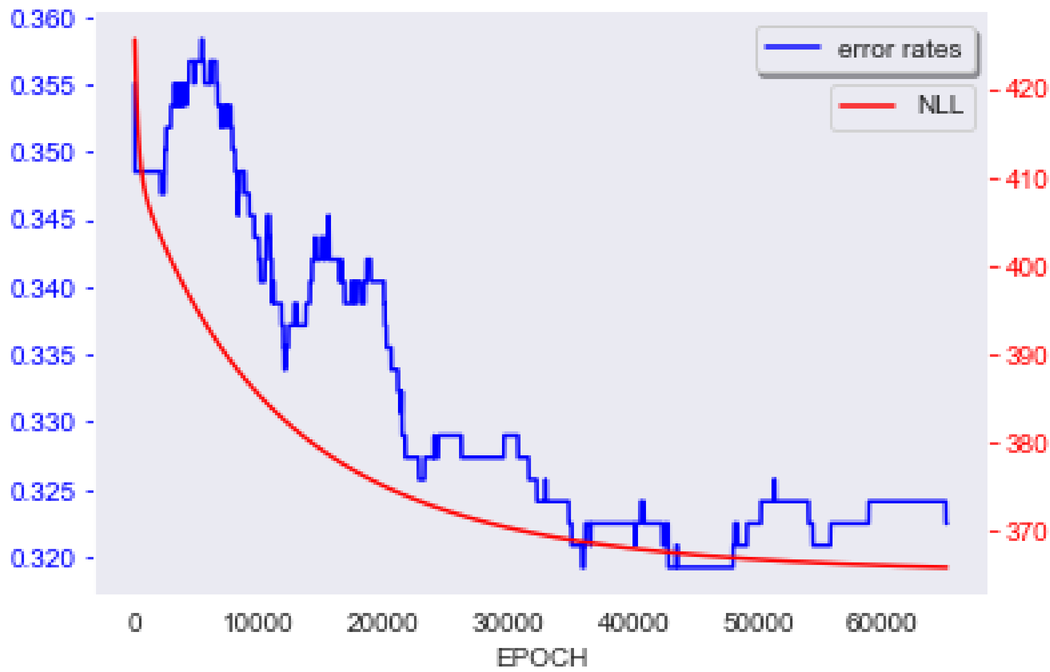
Problem 2 part A Small step sizes seem to converge faster. Step sizes in the range 1e-5 to 1e-7 seem to lead to convergence with the convergence happening quickest around 1e-6. The time for convergence was about 90 seconds on most runs on my ancient 2011 macbook pro. For reference, when the convergence threshold defined in the sgd() function was set to 1e-4 instead of 1e-5, convergence happened within 10 seconds or less almost every time. The error rates and NLL are plotted above for the training data as are the results when ran on the held out test data.

The highest positive weight by a full order of magnitude was for the seventh feature, which corresponds to the diabetes pedegree function that the patient has had. The largest negative weight assinged was that corresponding to the Blood Pressure feature.

Problem 2 part B

```
In [208]: learning_rate_normalized = 1e-5
          normalized_weights, error_rates, NLLs = sgd(train_normalized[:,:-1], train_normalized
          plot_sgd(error_rates, NLLs)
```

```
Converged in 65167 epochs and 429.183 seconds with learning rate of 1e-05
Weight Values at Convergence:
 [ 1.24278302  1.22360235 -3.33495651  0.21507972  1.55373068 -0.35648428
  0.90116595  1.46609395]
```




```
In [209]: sgd_test(test_normalized[:,:-1], test_normalized[:,-1], normalized_weights, "Test No
```

```
Test Normalized Set Errors: 57 (62.99% accuracy)
Test Normalized Set NLL: 97.215
```

```
In [210]: sgd_test(train_normalized[:,:-1], train_normalized[:,-1], normalized_weights, "Train
```

```
Train Normalized Set Errors: 198 (67.75% accuracy)
Train Normalized Set NLL: 365.806
```

```
In [211]: (unnormalized_weights - unnormalized_weights.min())/unnormalized_weights.max()
```

```
Out[211]: array([0.5600533 , 0.13120227, 0.        , 0.08940591, 0.09300006,
                 0.0867229 , 1.09510728, 0.06242674])
```

```
In [212]: (normalized_weights - normalized_weights.min())/normalized_weights.max()

Out[212]: array([2.94628894, 2.93394403, 0.        , 2.28484658, 3.14641865,
                 1.91698102, 2.72642003, 3.09001458])
```

The learned weights are of course on a different scale, but other than the surprisingly high value for the weight corresponding pregnanices in the unnormalized model, they seem to be picking up on the same features. This can be seen by rescaling the weights and observing that other than that future, the rescaled weights are shifted by a roughly constant factor of ~10.

Problem 2 part C

```
In [180]: # UNNORMALIZED
          alphas = [1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1] # note: 10 may cause numerical over

          for a in alphas:

              weights, error_rates, NLLs = sgd(train[:,:-1],
                                               train[:,-1],
                                               learning_rate,
                                               regularization=True,
                                               =a,
                                               print_=False)
              print('alpha:',a)
              print(weights)
              sgd_test(test[:,:-1], test[:,-1], weights, "Test Unnormalized")
              sgd_test(train[:,:-1], train[:,-1], weights, "Train Unnormalized")
              print()

alpha: 1e-06
[ 0.12985581  0.01111057 -0.02859385 -0.00116133 -0.00049154 -0.0011609
  0.07236405 -0.00790811]
Test Unnormalized Set Errors: 51 (66.88% accuracy)
Test Unnormalized Set NLL: 95.592
Train Unnormalized Set Errors: 189 (69.22% accuracy)
Train Unnormalized Set NLL: 376.017

alpha: 1e-05
[ 0.08487217  0.01064195 -0.02771484 -0.0013201  -0.00035795 -0.00140828
  0.00860553 -0.00124679]
Test Unnormalized Set Errors: 48 (68.83% accuracy)
Test Unnormalized Set NLL: 95.276
Train Unnormalized Set Errors: 198 (67.75% accuracy)
Train Unnormalized Set NLL: 377.312

alpha: 0.0001
[ 0.02021907  0.00944341 -0.02388453 -0.00241247 -0.0001468  -0.00218294
  0.00077269  0.0049574 ]
```

```
Test Unnormalized Set Errors: 49 (68.18% accuracy)
Test Unnormalized Set NLL: 95.796
Train Unnormalized Set Errors: 197 (67.92% accuracy)
Train Unnormalized Set NLL: 383.125

alpha: 0.001
[ 2.51806930e-03  4.77636984e-03 -1.19309641e-02 -2.83248993e-03
 -5.31322740e-05 -1.49744592e-03  5.78899135e-05  8.73255853e-04]
Test Unnormalized Set Errors: 51 (66.88% accuracy)
Test Unnormalized Set NLL: 98.672
Train Unnormalized Set Errors: 213 (65.31% accuracy)
Train Unnormalized Set NLL: 393.838

alpha: 0.01
[ 3.18133582e-04  8.97468533e-05 -1.79065401e-03 -6.11138663e-04
 -1.00166128e-03 -1.28687537e-04 -3.64670701e-06 -1.07297025e-04]
Test Unnormalized Set Errors: 54 (64.94% accuracy)
Test Unnormalized Set NLL: 102.694
Train Unnormalized Set Errors: 217 (64.66% accuracy)
Train Unnormalized Set NLL: 417.143

alpha: 0.1
[ 9.58257719e-05 -2.43476326e-04 -2.73695390e-04 -2.41876571e-04
 -2.23635132e-03 -8.26202716e-05  5.11257323e-07 -1.16098958e-05]
Test Unnormalized Set Errors: 54 (64.94% accuracy)
Test Unnormalized Set NLL: 102.494
Train Unnormalized Set Errors: 214 (65.15% accuracy)
Train Unnormalized Set NLL: 428.075

alpha: 1
[ 0.00000000e+00 -5.03760855e-04 -3.17182761e-04 -9.32890473e-05
  0.00000000e+00 -1.27339550e-04 -3.67092401e-06 -1.49262476e-04]
Test Unnormalized Set Errors: 54 (64.94% accuracy)
Test Unnormalized Set NLL: 105.555
Train Unnormalized Set Errors: 214 (65.15% accuracy)
Train Unnormalized Set NLL: 419.980
```

In [179]:
```python
# NORMALIZED
alphas = [1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1] # note: 10 may cause numerical over

for a in alphas:

    weights, error_rates, NLLs = sgd(train_normalized[:,:-1],
                                     train_normalized[:,-1],
                                     learning_rate_normalized,
                                     regularization=True,
```

```
                                                               =a,
                                                               print_=False)
                print('alpha:',a)
                sgd_test(test_normalized[:,:-1], test_normalized[:,-1], weights, "Test Normalize
                sgd_test(train_normalized[:,:-1], train_normalized[:,-1], weights, "Train Normali
                print(weights)
                print()
```

```
alpha: 1e-06
Test Normalized Set Errors: 57 (62.99% accuracy)
Test Normalized Set NLL: 97.041
Train Normalized Set Errors: 199 (67.59% accuracy)
Train Normalized Set NLL: 366.264
[ 1.15204065  1.13593609 -3.21274279  0.20194926  1.47896807 -0.25124867
  0.82578488  1.3950256 ]

alpha: 1e-05
Test Normalized Set Errors: 56 (63.64% accuracy)
Test Normalized Set NLL: 98.349
Train Normalized Set Errors: 206 (66.45% accuracy)
Train Normalized Set NLL: 382.193
[ 0.57834984  0.36073139 -1.45709786 -0.02441676  0.5686757  -0.31275062
  0.32080795  0.64867536]

alpha: 0.0001
Test Normalized Set Errors: 54 (64.94% accuracy)
Test Normalized Set NLL: 102.983
Train Normalized Set Errors: 213 (65.31% accuracy)
Train Normalized Set NLL: 405.995
[ 0.06277776 -0.08032464 -0.35968836 -0.06591976  0.05610435 -0.16282228
  0.00831361  0.0730203 ]

alpha: 0.001
Test Normalized Set Errors: 54 (64.94% accuracy)
Test Normalized Set NLL: 105.485
Train Normalized Set Errors: 214 (65.15% accuracy)
Train Normalized Set NLL: 419.244
[-4.57330963e-03 -4.04188062e-02 -6.93731548e-02 -1.86810445e-02
 -2.08486454e-05 -4.16507379e-02 -8.49896849e-03 -3.68774870e-03]

alpha: 0.01
Test Normalized Set Errors: 54 (64.94% accuracy)
Test Normalized Set NLL: 106.635
Train Normalized Set Errors: 214 (65.15% accuracy)
Train Normalized Set NLL: 425.036
[ 3.05362518e-04 -3.36690176e-03 -5.64943269e-03 -1.49608447e-03
 -8.13285872e-05 -2.99623444e-03 -8.92893150e-04 -5.27178155e-04]
```

```
alpha: 0.1
Test Normalized Set Errors: 100 (35.06% accuracy)
Test Normalized Set NLL: 106.749
Train Normalized Set Errors: 400 (34.85% accuracy)
Train Normalized Set NLL: 425.618
[ 6.89533581e-04  2.76104970e-04  7.37064347e-05 -1.13558065e-04
 -1.38480512e-04  1.69251348e-04  1.69356078e-04  2.92170449e-04]

alpha: 1
Test Normalized Set Errors: 54 (64.94% accuracy)
Test Normalized Set NLL: 106.738
Train Normalized Set Errors: 214 (65.15% accuracy)
Train Normalized Set NLL: 425.556
[ 0.00000000e+00 -2.71271063e-04 -2.98151399e-04 -1.00978192e-04
  0.00000000e+00 -2.03363458e-04 -1.51318538e-04 -9.16377095e-05]
```

When the regularization parameter α is small, the weights that are learned are closer to the weights learned by the unregularized models above. For large α, the weights begin to approach zero as the regularizer acts to dampen overfitting. Interestingly, both the normalized and unnormalized models go down in test set accuracy when the regularization parameter goes from 0.01 to 0.1, but then increase again when the parameter goes to 1. The cell above this shows that the normalized model is actually setting some of the feature importances to 0, encouraging sparsity as LASSO regularization does (it basically is LASSO when a=1).

0.0.1 3. Trees and Forests

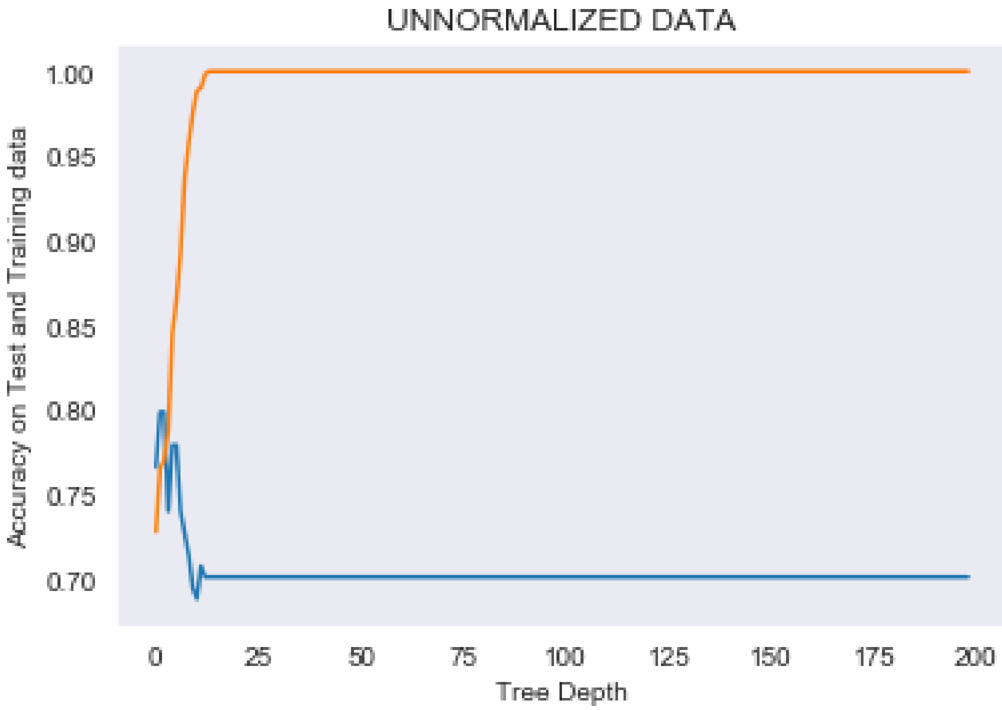
I experimented going all the way down to depths of 200, and found that the depths < 10 were consistently performing the best for both models.

```
In [102]: for depth in range(1,7):
              clf = tree.DecisionTreeClassifier(max_depth=depth, random_state=7)
              t0 = time()
              clf.fit(train[:,:-1],train[:,-1])
              train_time = time() - t0
              clf_pred = clf.predict(test[:,:-1])
              clf_pred_train = clf.predict(train[:,:-1])
              print("Unnormalized - Max Depth:",depth)
              print("Train time:", round(train_time,6))
              print("Accuracy Test:",round(accuracy_score(test[:,-1],clf_pred),3))
              print("Accuracy Train:",round(accuracy_score(train[:,-1],clf_pred_train),3))
              print("======================")

          accuracies = []
          accuracies_train = []
          for depth in range(1,200):
              clf = tree.DecisionTreeClassifier(max_depth=depth, random_state=7)
```

```
        t0 = time()
        clf.fit(train[:,:-1],train[:,-1])
        train_time = time() - t0
        clf_pred = clf.predict(test[:,:-1])
        clf_pred_train = clf.predict(train[:,:-1])
        accuracies.append(accuracy_score(test[:,-1],clf_pred))
        accuracies_train.append(accuracy_score(train[:,-1],clf_pred_train))

    plt.plot(accuracies)
    plt.plot(accuracies_train)
    plt.title("UNNORMALIZED DATA")
    plt.xlabel('Tree Depth')
    plt.ylabel('Accuracy on Test and Training data')
```

```
Unnormalized - Max Depth: 1
Train time: 0.001134
Accuracy Test: 0.766
Accuracy Train: 0.728
======================
Unnormalized - Max Depth: 2
Train time: 0.001329
Accuracy Test: 0.799
Accuracy Train: 0.765
======================
Unnormalized - Max Depth: 3
Train time: 0.001909
Accuracy Test: 0.799
Accuracy Train: 0.77
======================
Unnormalized - Max Depth: 4
Train time: 0.002216
Accuracy Test: 0.74
Accuracy Train: 0.788
======================
Unnormalized - Max Depth: 5
Train time: 0.001879
Accuracy Test: 0.779
Accuracy Train: 0.845
======================
Unnormalized - Max Depth: 6
Train time: 0.002425
Accuracy Test: 0.779
Accuracy Train: 0.865
======================
```

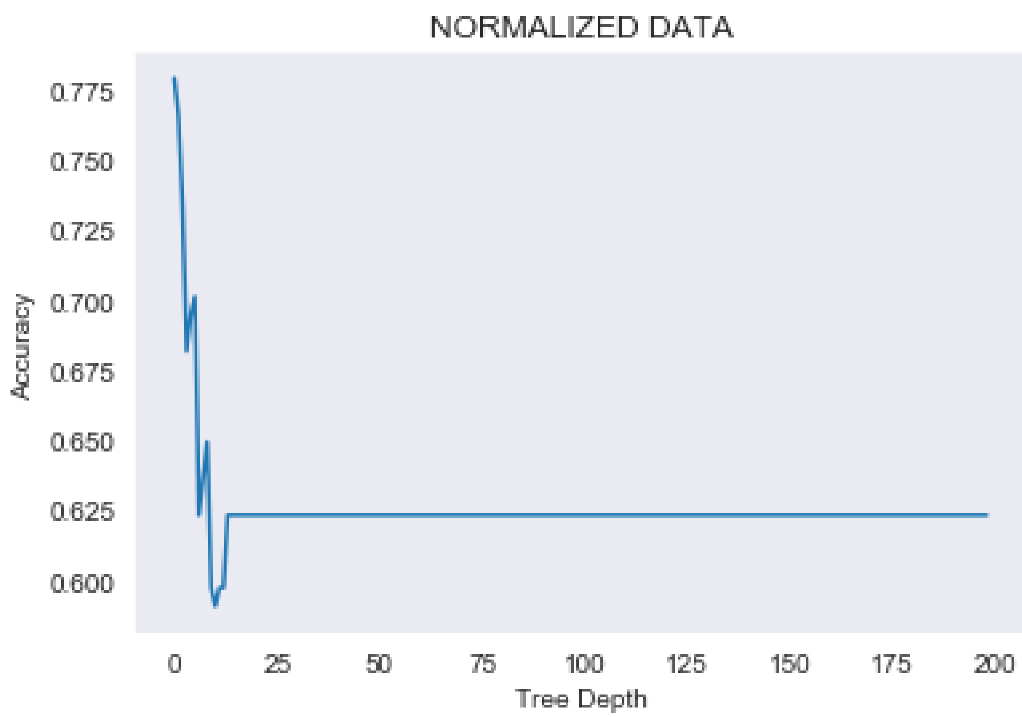
```
Out[102]: Text(0, 0.5, 'Accuracy on Test and Training data')
```




```
In [90]: for depth in range(1,5):
             clf = tree.DecisionTreeClassifier(max_depth=depth, random_state=7)
             t0 = time()
             clf.fit(train_normalized[:,:-1],train_normalized[:,-1])
             train_time = time() - t0
             clf_pred = clf.predict(test_normalized[:,:-1])
             print("Normalized - Max Depth:",depth)
             print("Train time:", round(train_time,6))
             print("Accuracy:",round(accuracy_score(test_normalized[:,-1],clf_pred),3))
             print("======================")

         accuracies = []
         for depth in range(1,200):
             clf = tree.DecisionTreeClassifier(max_depth=depth, random_state=7)
             t0 = time()
             clf.fit(train_normalized[:,:-1],train_normalized[:,-1])
             train_time = time() - t0
             clf_pred = clf.predict(test_normalized[:,:-1])
             accuracies.append(accuracy_score(test_normalized[:,-1],clf_pred))

         plt.plot(accuracies)
         plt.title("NORMALIZED DATA")
         plt.xlabel('Tree Depth')
         plt.ylabel('Accuracy')
```

```
Normalized - Max Depth: 1
Train time: 0.001443
Accuracy: 0.779
======================
Normalized - Max Depth: 2
Train time: 0.001827
Accuracy: 0.766
======================
Normalized - Max Depth: 3
Train time: 0.001846
Accuracy: 0.734
======================
Normalized - Max Depth: 4
Train time: 0.002144
Accuracy: 0.682
======================
```

Out[90]: Text(0, 0.5, 'Accuracy')



The decision trees perform better on this dataset than the logistic regression model. This is likely because there are a few features in this dataset that are highly relelvant predictors. Thus, if the decision tree can make high information splits on those few futures, it can very quickly fit a good model at shallow depth, which is also likely to prevent it from overfitting.

Should the training or test set accuracies be the same on the unnormalized data as the normalized data? Why or why not?

The training and test set accuracies should be the same on the unnormalized data as normalized data. This is because decision trees make "decisions" about the import feature splits based on information crietria (entropy in this case). These splits should be the same so long as the scaling process preserves the ordering of the points along any given dimension of the feature space. The reason that the accuracies in our case are very close however, is due to the randomness introduced by how we split our training and testing data. These two distribution are scaled using a built in sklearn method but each of them has a slightly different distribution and is thus scaled via slightly different transformations. So if we would have scaled the training data and testing data using only transformation properties of one of the two distributions (since seperating them randomly to create two different datasets), then the model would have predicted the exact same accuracies.

In [96]:
```
n_estimators = [5,25,50,100]
max_depths = [2,4]

print("\n##########  For unnormalized data  ################\n")

for n in n_estimators:
    for depth in max_depths:
        clf = RandomForestClassifier(n_estimators=n, max_depth=depth, random_state=7)
        t0 = time()
        clf.fit(train[:,:-1],train[:,-1])
        train_time = time() - t0
        clf_pred = clf.predict(test[:,:-1])
        clf_pred_train = clf.predict(train[:,:-1])
        print("Unormalized - Max Depth: {}\tNumber of Estimators: {}".format(depth,n)
        print("Train time:", round(train_time,6))
        print("Accuracy Test:",round(accuracy_score(test[:,-1],clf_pred),3))
        print("Accuracy Train:",round(accuracy_score(train[:,-1],clf_pred_train),3))
        print("======================")


print("\n##########  For normalized data  ################\n")

for n in n_estimators:
    for depth in max_depths:
        clf = RandomForestClassifier(n_estimators=n, max_depth=depth, random_state=7)
        t0 = time()
        clf.fit(train_normalized[:,:-1],train_normalized[:,-1])
        train_time = time() - t0
        clf_pred = clf.predict(test_normalized[:,:-1])
        clf_pred_train = clf.predict(train_normalized[:,:-1])
        print("Normalized - Max Depth: {}\tNumber of Estimators: {}".format(depth,n))
        print("Train time:", round(train_time,6))
        print("Accuracy Test:",round(accuracy_score(test_normalized[:,-1],clf_pred),3
        print("Accuracy Train:",round(accuracy_score(train_normalized[:,-1],clf_pred_
```

```
            print("======================")
```

```
##########  For unnormalized data  ################

Unormalized - Max Depth: 2        Number of Estimators: 5
Train time: 0.008078
Accuracy Test: 0.688
Accuracy Train: 0.735
======================
Unormalized - Max Depth: 4        Number of Estimators: 5
Train time: 0.00829
Accuracy Test: 0.74
Accuracy Train: 0.798
======================
Unormalized - Max Depth: 2        Number of Estimators: 25
Train time: 0.022296
Accuracy Test: 0.74
Accuracy Train: 0.754
======================
Unormalized - Max Depth: 4        Number of Estimators: 25
Train time: 0.033887
Accuracy Test: 0.779
Accuracy Train: 0.831
======================
Unormalized - Max Depth: 2        Number of Estimators: 50
Train time: 0.041441
Accuracy Test: 0.74
Accuracy Train: 0.757
======================
Unormalized - Max Depth: 4        Number of Estimators: 50
Train time: 0.045726
Accuracy Test: 0.766
Accuracy Train: 0.832
======================
Unormalized - Max Depth: 2        Number of Estimators: 100
Train time: 0.09582
Accuracy Test: 0.74
Accuracy Train: 0.761
======================
Unormalized - Max Depth: 4        Number of Estimators: 100
Train time: 0.0849
Accuracy Test: 0.786
Accuracy Train: 0.824
======================

##########  For normalized data  ################
```

```
Normalized - Max Depth: 2        Number of Estimators: 5
Train time: 0.007464
Accuracy Test: 0.708
Accuracy Train: 0.735
======================
Normalized - Max Depth: 4        Number of Estimators: 5
Train time: 0.007972
Accuracy Test: 0.695
Accuracy Train: 0.798
======================
Normalized - Max Depth: 2        Number of Estimators: 25
Train time: 0.028676
Accuracy Test: 0.734
Accuracy Train: 0.754
======================
Normalized - Max Depth: 4        Number of Estimators: 25
Train time: 0.022455
Accuracy Test: 0.786
Accuracy Train: 0.829
======================
Normalized - Max Depth: 2        Number of Estimators: 50
Train time: 0.041347
Accuracy Test: 0.76
Accuracy Train: 0.757
======================
Normalized - Max Depth: 4        Number of Estimators: 50
Train time: 0.051917
Accuracy Test: 0.766
Accuracy Train: 0.831
======================
Normalized - Max Depth: 2        Number of Estimators: 100
Train time: 0.080451
Accuracy Test: 0.76
Accuracy Train: 0.761
======================
Normalized - Max Depth: 4        Number of Estimators: 100
Train time: 0.083245
Accuracy Test: 0.779
Accuracy Train: 0.824
======================
```

Question 4

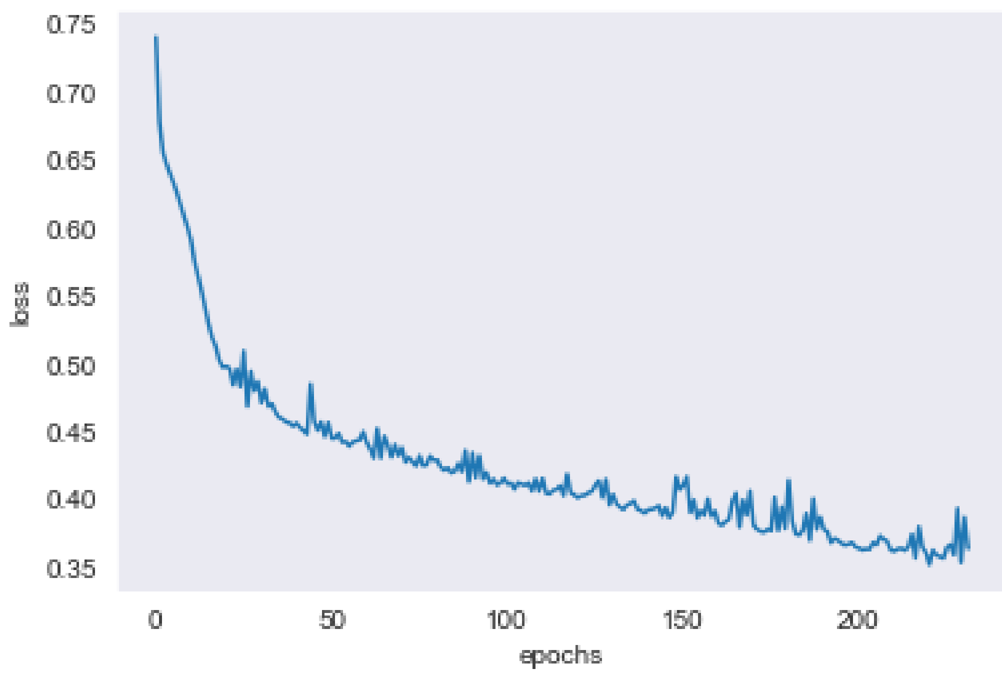
In [186]:
```
hidden_layer_configs = [(100,100,100), (50,50,50),
                         (20,20,20), (100,50,20),
                         (20,50,100), (25,75,25),
                        ]
```

```python
alphas = [1e-5,1e-4,1e-3,1e-2]
learning_rates = [1e-3,1e-2,1e-1]
solvers = ['sgd', 'adam']
model_id = 0

model_results = {}

for hidden_layer_config in hidden_layer_configs:
    for alpha in alphas:
        for learning_rate in learning_rates:
            for solver in solvers:
                model_id += 1

                # print(model_id)

                t0 = time()
                mlp = MLPClassifier(hidden_layer_sizes=hidden_layer_config,
                        alpha=alpha, solver=solver, tol=1e-4, random_state=1,
                        learning_rate_init=learning_rate, max_iter=1000)
                mlp.fit(train_normalized[:,:-1], train_normalized[:,-1])
                train_time = time() - t0
                y_pred_train = mlp.predict(train_normalized[:,:-1])
                y_pred_test = mlp.predict(test_normalized[:,:-1])
                acc_test = accuracy_score(train_normalized[:,-1], y_pred_train)
                acc_train = accuracy_score(test_normalized[:,-1], y_pred_test)


                best_loss = mlp.best_loss_
                model_results[model_id] = {}
                model_results[model_id]['layers'] = hidden_layer_config
                model_results[model_id]['alpha'] = alpha
                model_results[model_id]['solver'] = solver
                model_results[model_id]['best_loss'] = best_loss
                model_results[model_id]['loss_curve_'] = mlp.loss_curve_
                model_results[model_id]['learning_rate'] = learning_rate
                model_results[model_id]['train_time'] = train_time
                model_results[model_id]['acc_test'] = acc_test
                model_results[model_id]['acc_train'] = acc_train


best_model = model_results[1]
for i in range(1,145):
    if best_model['best_loss'] > model_results[i]['best_loss']:
        best_model = model_results[i]

print("layers:", best_model['layers'])
print("alpha:",best_model['alpha'])
```

```
print("solver:", best_model['solver'])
print("train time:", round(best_model['train_time'],3))
print("learning_rate:",best_model['learning_rate'])
print("accuracy on test normalized data:",round(best_model['acc_test'],3))
print("accuracy on train normalized data:",round(best_model['acc_train'],3))

plt.plot(best_model['loss_curve_'])
plt.xlabel('epochs')
plt.ylabel('loss')
```

```
layers: (100, 100, 100)
alpha: 1e-05
solver: adam
train time: 1.701
learning_rate: 0.001
accuracy on test normalized data: 0.85
accuracy on train normalized data: 0.74
```

```
Out[186]: Text(0, 0.5, 'loss')
```



Problem 5 INIT

$(w_{31}, w_{32}) = (1, 1)$
$(w_{41}, w_{42}) = (1, -1)$
$(w_{51}, w_{52}) = (-1, -1)$

$(w_{63}, w_{64}, w_{65}) = (1, 1, 1)$
$x_1 = 1, x_2 = 2$
$t = 2$

First we need to compute the forward pass:

$a_3 = w_{31} * x_1 + w_{32} * x_2 = 1 * 1 + 1 * 2 = 3$
$a_4 = w_{41} * x_1 + w_{42} * x_2 = 1 * 1 + (-1) * 2 = -1$
$a_5 = w_{51} * x_1 + w_{52} * x_2 = (-1) * 1 + (-1) * 2 = -3$
$z_3 = ReLU(a_3) = 3$
$z_4 = ReLU(a_4) = 0$
$z_5 = ReLU(a_5) = 0$
$a_6 = w_{63} * z_3 + w_{64} * z_4 + w_{65} * z_5 = 1 * 3 + 1 * 0 + 1 * 0 = 3$
$Error = \frac{1}{2}(a_6 - t)^2 = \frac{1}{2}(3 - 2)^2 = \frac{1}{2}$

Now we can backpropagate:

$\frac{\partial E}{\partial a_6} = \delta_6 = (a_6 - t) = 3 - 2 = 1$
$\frac{\partial E}{\partial w_{65}} = \delta_6 z_5 = 1 * 0 = 0$
$\frac{\partial E}{\partial w_{64}} = \delta_6 z_4 = 1 * 0 = 0$
$\frac{\partial E}{\partial w_{63}} = \delta_6 z_3 = 1 * 3 = 3$

Notice that we want $\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial a_j} z_i$, and since $z_4 = z_5 = 0$, the error derivatives with respect to the weights for hidden nodes that depend on nodes 4 and 5 will have no updates. Also, since $a_4 < 0$ and $a_5 < 0$, we have $\frac{\partial z_4}{\partial a_4} = 0$ and $\frac{\partial z_5}{\partial a_5} = 0$ and therefore $\delta_4 = \delta_5 = 0$. Because of this and the fact that $\frac{\partial E}{\partial w_{ji}} = \delta_j z_i$, $\frac{\partial E}{\partial w_{51}} = \frac{\partial E}{\partial w_{52}} = \frac{\partial E}{\partial w_{41}} = \frac{\partial E}{\partial w_{42}} = 0$.

The only weights left to compute are $\frac{\partial E}{\partial w_{31}}$ and $\frac{\partial E}{\partial w_{32}}$. We have:

$\frac{\partial E}{\partial w_{31}} = \frac{\partial E}{\partial a_3} z_1 = \frac{\partial E}{\partial z_3} \frac{\partial z_3}{\partial a_3} z_1$

We know $z_1 = 1$ and since $a_3 > 0$ that $\frac{\partial z_3}{\partial a_3} = 1$. Thus, we have $\frac{\partial E}{\partial a_j} = \sum_k \frac{\partial E}{\partial a_k} \frac{\partial a_k}{\partial z_j} = \sum_k \frac{\partial E}{\partial a_k} w_{kj}$ which in the case of a_3 yields:

$\frac{\partial E}{\partial a_6} w_{63} = (3 - 2)1 = 1$

Finally, we use this to compute $\frac{\partial E}{\partial w_{31}}$ and $\frac{\partial E}{\partial w_{32}}$.

$\frac{\partial E}{\partial w_{31}} = \frac{\partial E}{\partial a_3} z_1 = 1 * 1 = 1$
$\frac{\partial E}{\partial w_{32}} = \frac{\partial E}{\partial a_3} z_2 = 1 * 2 = 2$

Now we just need to update the weights with learning rate $\eta = .1$

$w_{31} = w_{31} - .1(1) = 1 - 0.1 = 0.9$
$w_{32} = w_{32} - .1(2) = 1 - 0.2 = 0.8$
$w_{41} = w_{41} - .1(0) = 1 - 0 = 1$
$w_{42} = w_{42} - .1(0) = -1 - 0 = -1$
$w_{51} = w_{51} - .1(0) = -1 - 0 = -1$
$w_{52} = w_{52} - .1(0) = -1 - 0 = -1$
$w_{63} = w_{63} - .1(3) = 1 - 0.3 = 0.7$
$w_{64} = w_{64} - .1(0) = 1 - 0 = 1$
$w_{65} = w_{65} - .1(0) = 1 - 0 = 1$

```
In [ ]:
```