Machine Learning HW2

Chutian Chen cc4515

1

(a)

$$p(w) = (2\pi)^{-\frac{k}{2}} \det(\boldsymbol{\Sigma})^{-\frac{1}{2}} e^{-\frac{1}{2}(\mathbf{w}-\boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\mathbf{w}-\boldsymbol{\mu})}$$

$$\Sigma = \tau^2 I$$

$$Let\ L(w) = \left(\sum_{i=1}^{N} \ln p\left(y^{(i)} | \mathbf{x}^{(i)}; \mathbf{w}\right) \right) + \ln p(\mathbf{w})$$

$$\frac{\partial L}{\partial w} = -\sum_{i=1}^{N} \frac{w^T x^{(i)} - y^{(i)}}{\sigma^2} x^{(i)} - \frac{1}{\tau^2} w$$

$$\text{Let } A = (x^{(1)}, x^{(2)}, .., x^{(N)}),\ y = (y^{(1)}, y^{(2)}, .., y^{(N)})^T$$

$$\frac{\partial L}{\partial w} = -A(\frac{A^T w - y}{\sigma^2}) - \frac{w}{\tau^2} = 0$$

So

$$w_{MAP} = (AA^T + \frac{\sigma^2}{\tau^2} I)^{-1} Ay$$

It's same as the form of the solution of ridge regression.

(b)

$$p(w_i) = \frac{1}{2b} \exp\left(-\frac{|w_i|}{b}\right)$$

So

$$lnp(w_i) = -\frac{|w_i|}{b} - ln(2b)$$

$$\begin{aligned} \text{So } \ w_{MAP} &= \arg\max -\frac{1}{2\sigma^2} \sum_{i=1}^{N} \left(y^{(i)} - \hat{\mathbf{w}}^T \mathbf{x}^{(i)}\right)^2 - \frac{\|\hat{\mathbf{w}}\|_1}{b} \\ &= \arg\min \frac{1}{2\sigma^2} \sum_{i=1}^{N} \left(y^{(i)} - \hat{\mathbf{w}}^T \mathbf{x}^{(i)}\right)^2 + \frac{\|\hat{\mathbf{w}}\|_1}{b} \\ &= \arg\min \frac{1}{N} \sum_{i=1}^{N} \left(y^{(i)} - \hat{\mathbf{w}}^T \mathbf{x}^{(i)}\right)^2 + \frac{2\sigma^2}{Nb} \|\hat{\mathbf{w}}\|_1 \end{aligned}$$

It doesn't have a closed solution. But we can see it has the same form as lassor regression.

2

(a)

$$\hat{f}(\mathbf{x}) = \underset{y\in\{0,1\}}{\arg\max} \Pr(y|\mathbf{x}) = \underset{y\in\{0,1\}}{\arg\max} \underbrace{\pi_y}_{:=\Pr(y)} p(\mathbf{x}|y;\mu_y,\mathbf{\Sigma}_y)$$

$$p(\mathbf{x}|y;\mu_y,\mathbf{\Sigma}_y) = \frac{1}{(2\pi)^{d/2}|\Sigma_y|^{1/2}}\exp\left(\frac{-(\mathbf{x}-\mu_y)^T(\Sigma_y)^{-1}(\mathbf{x}-\mu_y)}{2}\right)$$

$$d_\Sigma(\mathbf{x},\mu) = (\mathbf{x}-\mu)^T(\Sigma)^{-1}(\mathbf{x}-\mu)$$

$$\hat{f}(\mathbf{x}) = 1[\Pr(y=1|\mathbf{x}) > \Pr(y=0|\mathbf{x})] = 1\left[\ln\frac{\Pr(y=1|\mathbf{x})}{\Pr(y=0|\mathbf{x})} > 0\right]$$

$$= 1\left[\ln\frac{\pi_1}{(1-\pi_1)} + \ln\frac{p(\mathbf{x}|y=1)}{p(\mathbf{x}|y=0)} > 0\right]$$

$$= 1\left[\ln\frac{\pi_1}{(1-\pi_1)} - \frac{1}{2}\ln\frac{|\Sigma_1|}{|\Sigma_0|} - \frac{1}{2}(d_{\Sigma_1}(\mathbf{x},\mu_1) - d_{\Sigma_0}(\mathbf{x},\mu_0)) > 0\right]$$

$$\text{Because } \Sigma_1 \neq \Sigma_0, \text{ the boundry is quadratic}$$

$$\text{If } \Sigma_1 = \Sigma_0,\ d_{\mathbf{\Sigma}_1}(\mathbf{x},\mu_1) - d_{\mathbf{\Sigma}_0}(\mathbf{x},\mu_0) = \mathbf{x}^T(\mathbf{\Sigma}_1)^{-1}(\mu_1-\mu_0) - \frac{1}{2}\mu_1^T(\mathbf{\Sigma}_1)^{-1}\mu_1 + \frac{1}{2}\mu_0^T(\mathbf{\Sigma}_1)^{-1}\mu_0$$

So the boundry becomes linear.

(b)

$$For\ (\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_d \end{bmatrix})where\ (x_k \in \{0,1\}\ ),\ assume$$

$$\Pr(\mathbf{x}|y) = \prod_{k=1}^{d}\Pr[x_k|y]$$

Then we can make the classifier a Gaussian Naive Bayes Classifier.

Bayes Classifier:

$$\hat{f}(\mathbf{x}) = \underset{y\in\mathcal{Y}}{\arg\max}\Pr(y|\mathbf{x}) = \underset{y\in\mathcal{Y}}{\arg\max}\prod_{k=1}^{d}\Pr[x_k|y]\cdot\Pr[y]$$

$$\text{Because every pair of } \mathrm{x_i} \text{ and } \mathrm{x_j} \text{ is independent on the condition y, } \Sigma \text{ is diagonal matrix.}$$

$$\Sigma_{\mathrm{i,i}} = \mathrm{Var}(\mathrm{x_i}) = \sigma_i^2$$

$$\Sigma_{ii}^{-1} = \frac{1}{\sigma_i^2}$$

$$\hat{f}(\mathbf{x}) = \underset{y\in\{0,1\}}{\arg\max}\Pr(y|\mathbf{x}) = \underset{y\in\{0,1\}}{\arg\max}\ \pi_y \frac{1}{(2\pi)^{d/2}|\Sigma_y|^{1/2}}\exp\left(\frac{-(\mathbf{x}-\mu_y)^T(\Sigma_y)^{-1}(\mathbf{x}-\mu_y)}{2}\right)$$

$$= \underset{y\in\{0,1\}}{\arg\max}\ \pi_y \frac{1}{(2\pi)^{d/2}\prod_{k=1}^{d}\sigma_k}\exp\left(\frac{-\sum_{k=1}^{d}\frac{(x_k-\mu_{yk})^2}{\sigma_i^2}}{2}\right)$$

$$= \underset{y\in\{0,1\}}{\arg\max}\ \pi_y \prod_{k=1}^{d}\frac{1}{\sqrt{2\pi}\sigma_k}\exp\left(-\frac{(x_k-\mu_{yk})^2}{2\sigma_i^2}\right)$$

$$= \underset{y\in\{0,1\}}{\arg\max}\prod_{k=1}^{d}\Pr[x_k|y]\Pr[y]$$

We can see two classifiers are equavalent.

3

(a)

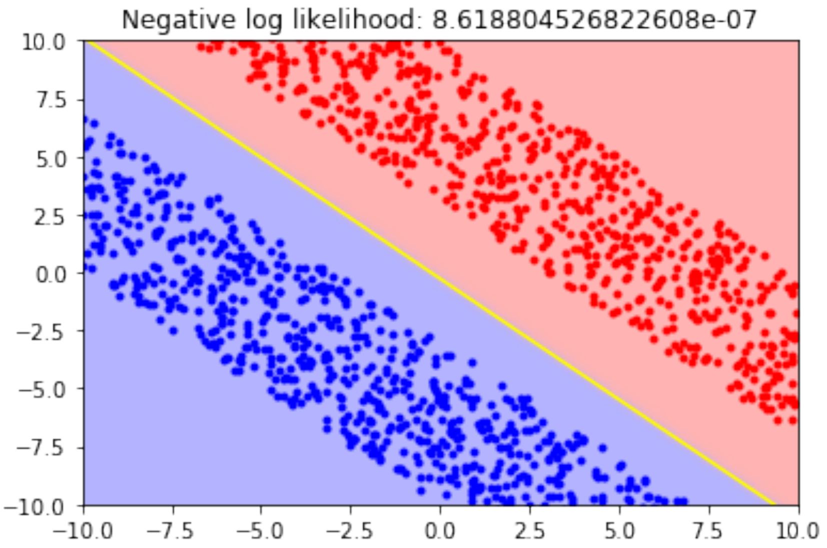
$$\text{Because } \lambda \text{ is very large, } \sum_{i=1}^{N}\log\left(P\left(y^{(i)}|x^{(i)};w_0,w_1,w_2\right)\right) - \lambda\cdot w_j^2 \approx -\lambda\cdot w_j^2$$

This function reach maximum when wj=0. From figure 1 we can see that when w2=0, the train error is smallest.

When w1=0, the train error is largest.

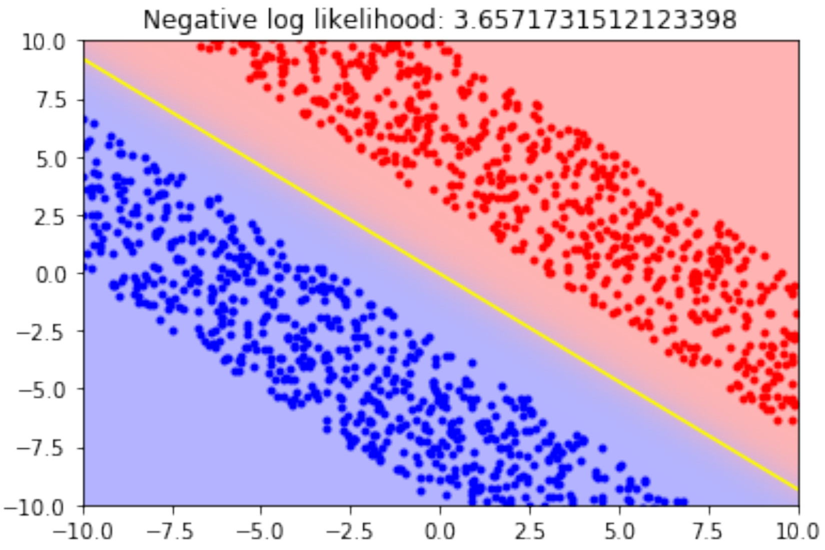
(b)

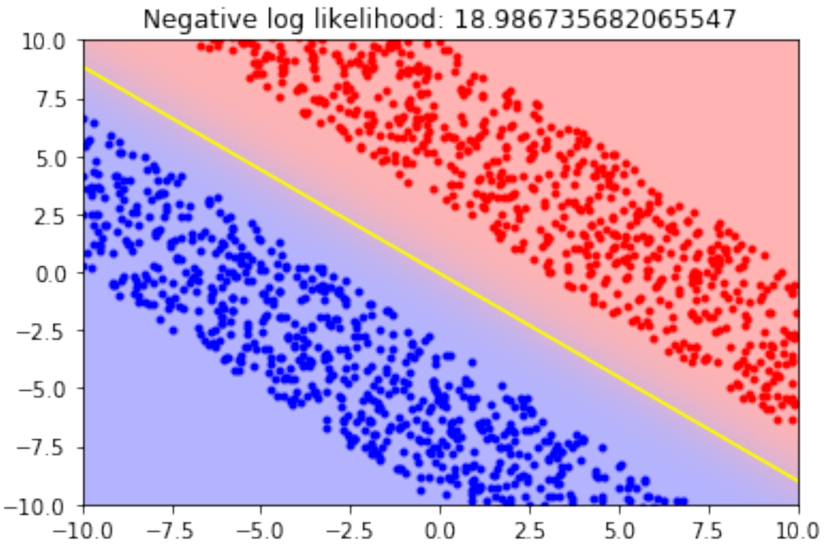
1. Without regularization

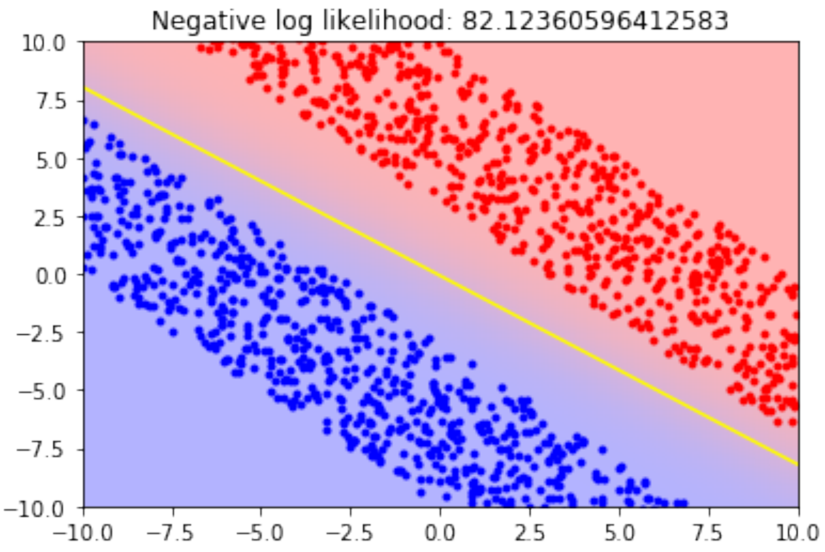


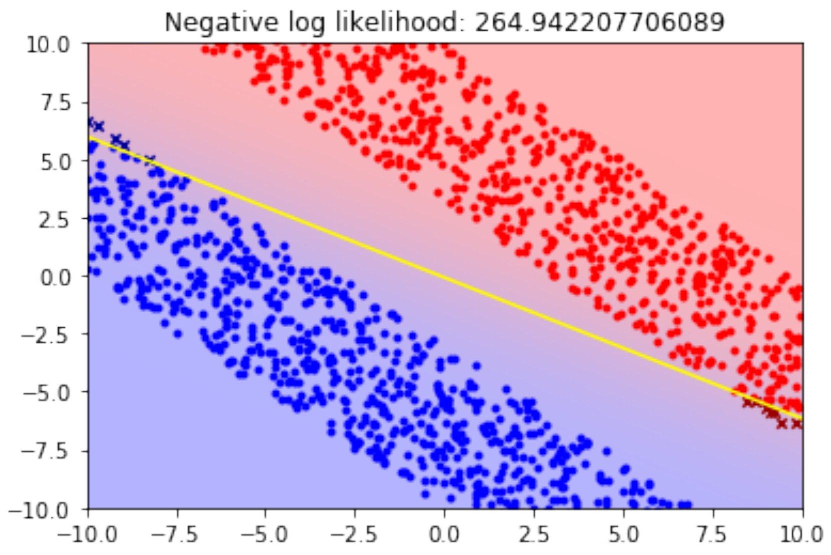
We can see that without regularization, the line classifies the points perfectly. Because the points in this graph is easy to divide by line, we don't need regularization to adjust the parameters.

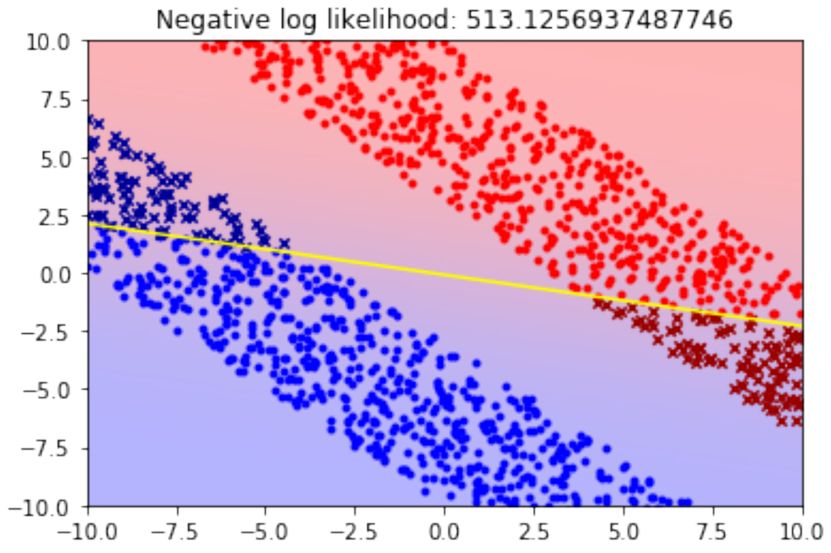
2. Regularize x1 (the order of pictures is the lambda from 1 to 100000)

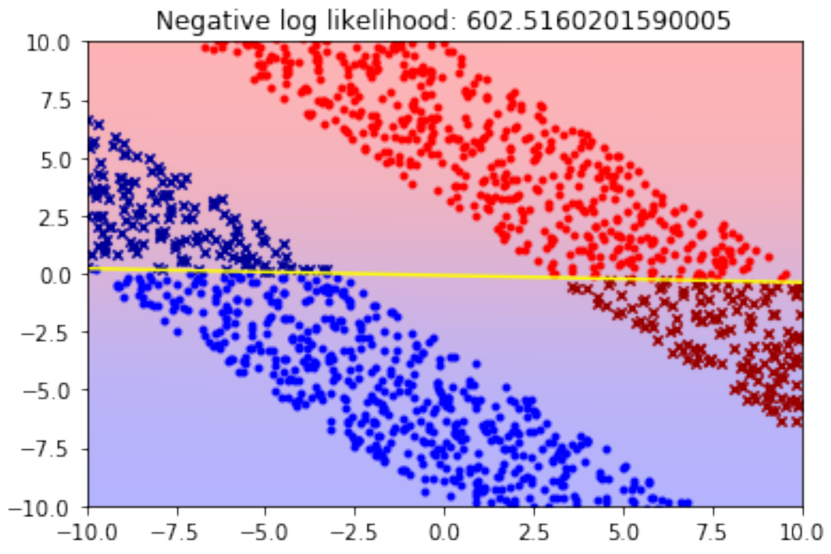






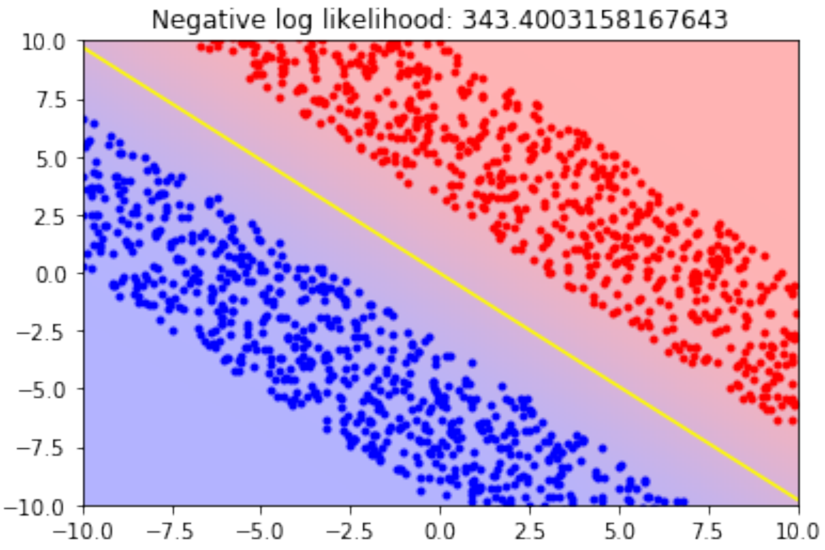


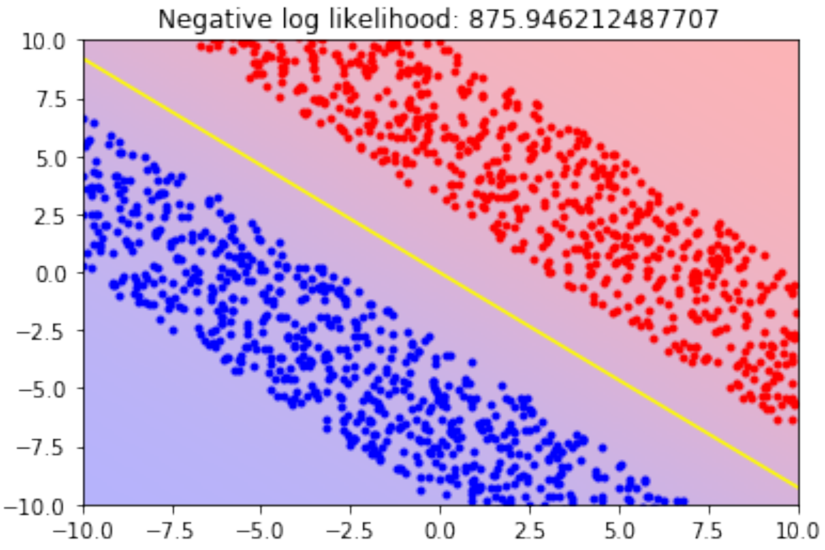


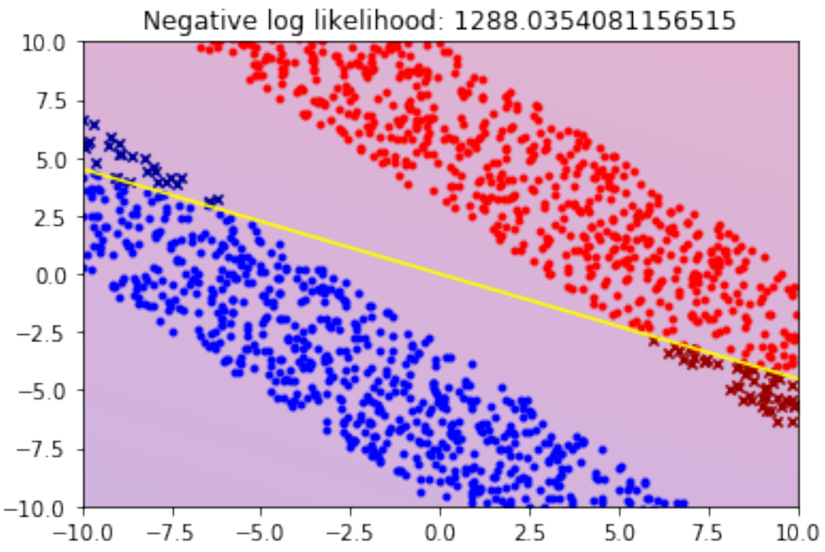


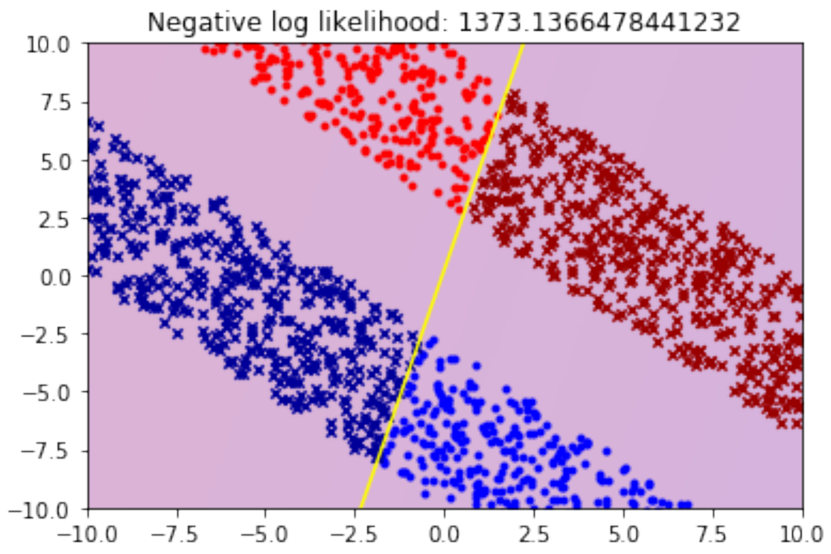
We can see as lambda increases, the line becomes horizontal. The reason is that w1 is closer to 0 as lambda increases.

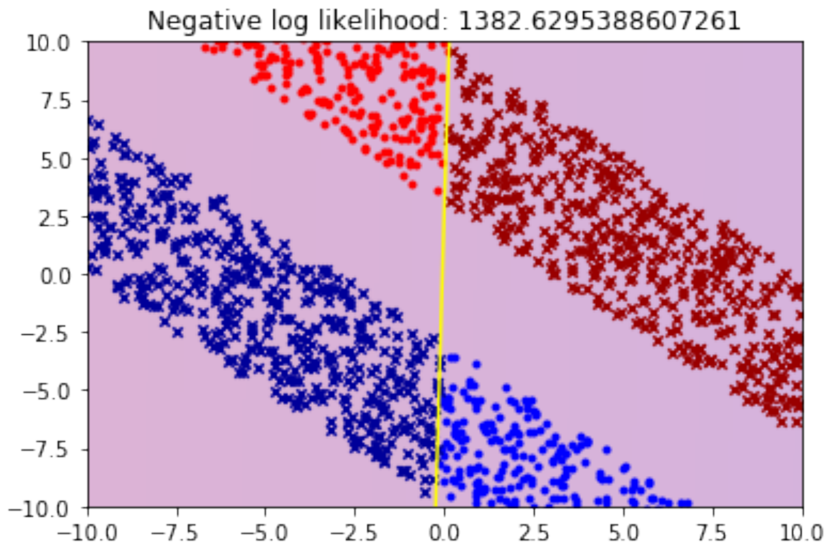
3. Regularize x2 (the order of pictures is the lambda from 1000 to 100000000)

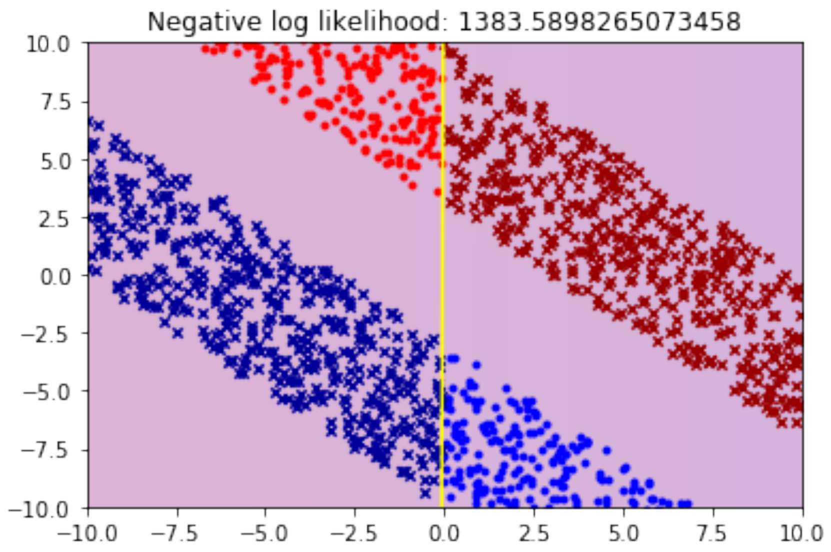












We can see as lambda increases, the line becomes vertical. The reason is that w2 is closer to 0 as lambda increases.

4

We can see that adding depth of the trees can improve the accuracy of the models. The accuracy of the models using Gini is close to that using Entropy. Because the two metrics have similar trends, the results of the two metrics are similar. What’s more, from the feature frequecy of the nodes we can see that the most useful features are 0 and 1. The least useful feature is 3.

HW2-MLDS-for-students

February 29, 2020

1 Hello and welcome to decision trees!

Decision trees are often pretty effect learning algorithms, and certainly serve as an interesting technical exercise in data preprocessing and recursion. This notebook will walk you through some of the basic notions of how the implementation should be executed, and also give you a chance to write some of your own code.

Suggested reading before you start: https://www.cs.princeton.edu/courses/archive/spring07/cos424/papers/mitch dectrees.pdf

Throughout the notebook you'll see TODO tags in the comments. This is where you should insert your own code to make the functions work! If you get stuck, we encourage you to come to office hours. You can also try to look at APIs and documentation online to try to get a sense how certain methods work. If you take inspiration from any source online other than official documentation, please be sure to cite the resource! Good luck!

[2]:
```
import pandas as pd
import scipy.io
import numpy as np
```

We will be using decision trees to classify if a banknote is fradulent (class 1) or not fradulent (class 0). Download data from https://archive.ics.uci.edu/ml/datasets/banknote+authentication#

1.1 Import data

[96]:
```
# TODO YOUR CODE HERE
def import_data(split = 0.8, shuffle=False):
    """Read in the data, split it by split percentage into train and test data,
    and return X_train, y_train, X_test, y_test as numpy arrays"""
    # TODO
    df = pd.read_csv('./data_banknote_authentication.txt',header = None,
                names = ['vairance','skewness','curtosis','entropy','class'])

    #shuffle the data
    df_shuffle = df.sample(frac=1)

    #split the data to get train and test data
    train = df_shuffle[:round(split*len(df))]
    test = df_shuffle[round(split*len(df)):]
    X_train = np.array(train.iloc[:,:-1])
    y_train = np.array(train.iloc[:,-1])
    X_test = np.array(test.iloc[:,:-1])
    y_test = np.array(test.iloc[:,-1])

    print("data imported")

    return X_train, y_train, X_test, y_test
```

[97]:
```
class Node:
    """Each node of our decision tree will hold values such as left and right␣
 ↪children,
    the data and labels being split on, the threshold value & index in the␣
 ↪dataframe for a particular feature,
    and the uncertainty measure for this node"""
    def __init__(self, data, labels, depth):
        """
        data: X data
        labels: y data
        depth: depth of tree
        """
        self.left = None
        self.right = None

        self.data = data
        self.labels = labels
        self.depth = depth

        self.threshold = None # threshold value
        self.threshold_index = None # threshold index
        self.feature = None # feature as a NUMBER (column number)
        self.label = None # y label
        self.uncertainty = None # uncertainty value
```

[198]:
```
class DecisionTree:
    def __init__(self, K=5, verbose=False):
        """
        K: number of features to split on
        """
        self.root = None
        self.K = K
        self.verbose = verbose

    def buildTree(self, data, labels, metric="entropy"):
        """Builds tree for training on data. Recursively called _buildTree"""
        self.root = Node(data, labels, 0)
        if self.verbose:
            print("Root node shape: ", data.shape, labels.shape)
        self._buildTree(self.root,metric)

    def _buildTree(self, node, metric="entropy"):

        # get uncertainty measure and feature threshold
        node.uncertainty = self.get_uncertainty(node.labels)
        self.get_feature_threshold(node,metric)

        index = node.data[:, node.feature].argsort()  # sort feature for return
        node.data = node.data[index]
        node.labels = node.labels[index]

        # check label distribution.
        label_distribution = np.bincount(node.labels)
        majority_label = node.labels[0] if len(label_distribution) == 1 else np.
↪argmax(label_distribution)

        if self.verbose:
            print("Node uncertainty: %f" % node.uncertainty)

        # Split left and right if threshold is not the min or max of the␣
↪feature or every point has the
        # same label.
        if node.threshold_index == 0 or node.threshold_index == node.data.
↪shape[0] or \
            len(label_distribution) == 1:
            node.label = majority_label
        else:
            node.left = Node(node.data[:node.threshold_index], node.labels[:
↪node.threshold_index], node.depth + 1)
            node.right = Node(node.data[node.threshold_index:], node.
↪labels[node.threshold_index:], node.depth + 1)
            node.data = None
            node.labels = None

            # If in last layer of tree, assign predictions
            if node.depth == self.K:
                if len(node.left.labels) == 0:
                    node.right.label = np.argmax(np.bincount(node.right.labels))
                    node.left.label = 1 - node.right.label
                elif len(node.right.labels) == 0:
                    node.left.label = np.argmax(np.bincount(node.left.labels))
                    node.right.label = 1 - node.left.label
                else:
                    node.left.label = np.argmax(np.bincount(node.left.labels))
                    node.right.label = np.argmax(np.bincount(node.right.labels))
                return

            else: # Otherwise continue training the tree by calling _buildTree
                self._buildTree(node.left)
                self._buildTree(node.right)

    def predict(self, data_pt):
        return self._predict(data_pt, self.root)

    def _predict(self, data_pt, node):
        feature = node.feature
        threshold = node.threshold
        if node.label is not None:
            return node.label
        elif data_pt[node.feature] < node.threshold:
            return self._predict(data_pt, node.left)
        elif data_pt[node.feature] >= node.threshold:
            return self._predict(data_pt, node.right)

    def get_feature_threshold(self, node,metric="entropy"):
        """ TODO Find the feature that gives the largest information gain.␣
↪Update node.threshold,
        node.threshold_index, and node.feature (a number representing the␣
↪feature. e.g. 2nd column feature would be 1)
        Make sure to sort the columns of data before you try to find the␣
↪threshold index (look at numpy argsort) and set the values
        for node.threshold, node.threshold_index, and node.feature
        return: None
        """
        node.threshold = 0
        node.threshold_index = 0
        node.feature = 0
        # TODO YOUR CODE HERE
        info_gain = []
        for col in range(node.data.shape[1]):
            node.feature = col
            info_gain.append([self.getInfoGain(node,i,metric) for i in␣
↪range(node.labels.shape[0])])

        info_gain = np.array(info_gain)
        max_index = np.unravel_index(info_gain.argmax(), info_gain.shape)
        node.feature = max_index[0]
        node.threshold_index = max_index[1]
        node.threshold = sorted(node.data[:,node.feature])[node.threshold_index]

    def getInfoGain(self, node, split_index,metric="entropy"):
        """
        TODO Get information gain using the variables in the parameters, \
        split_index: index in the feature column that you are splitting the␣
↪classes on
        return: information gain (float)
        """
        # TODO YOUR CODE HERE
        index = node.data[:, node.feature].argsort()
        labels = node.labels[index]
        return(self.get_uncertainty(labels,metric) -
               split_index/labels.shape[0]*self.get_uncertainty(labels[:
↪split_index],metric) -
               (labels.shape[0]-split_index)/labels.shape[0]*self.
↪get_uncertainty(labels[split_index:],metric))


    def get_uncertainty(self, labels, metric="entropy"):
        """
        TODO Get uncertainty. Implement entropy AND gini index metrics.
        np.bincount(labels) and labels.shape might be useful here
        return: uncertainty (float)
        """

        if labels.shape[0] == 0:
            return 1
        for y, n_k in zip(np.unique(labels),np.bincount(labels)):
            p[y] = n_k/labels.shape[0]
        if 0 in p.values(): return 0
        if metric == 'entropy':
            return(sum([-p_k*np.log(p_k) for p_k in p.values()]))
        if metric == 'gini':
            return(sum([p_k*(1-p_k) for p_k in p.values()]))

    def evaluation(self):
        feat_freq = {}
        feat_freq = self._evaluation(self.root,feat_freq)
        print("The feature frequency of depth {} is {}.".format(self.
↪K,feat_freq))

    def _evaluation(self,node,feat_freq):
        if node is not None:
            if node.label is None:
                if node.feature in feat_freq.keys(): feat_freq[node.feature] +=␣
↪1
                else: feat_freq[node.feature] = 1
            feat_freq = self._evaluation(node.left,feat_freq)
            feat_freq = self._evaluation(node.right,feat_freq)
        return feat_freq

    def printTree(self):
        """Prints the tree including threshold value and feature name"""
        self._printTree(self.root)

    def _printTree(self, node):
        if node is not None:
            if node.label is None:
                print("\t" * node.depth, "(%d, %d)" % (node.threshold, node.
↪feature))
            else:
                print("\t" * node.depth, node.label)
            self._printTree(node.left)
            self._printTree(node.right)

    def homework_evaluate(self, X_train, labels, X_test, y_test):
        n = X_train.shape[0]

        count = 0
        for i in range(n):
            if self.predict(X_train[i]) == labels[i]:
                count += 1

        print("The decision tree is %d percent accurate on %d training data" %␣
↪((count / n) * 100, n))

        n = X_test.shape[0]

        count = 0
        for i in range(n):
            if self.predict(X_test[i]) == y_test[i]:
                count += 1

        print("The decision tree is %d percent accurate on %d test data" %␣
↪((count / n) * 100, n))

        return count / n
```

1.2 Run the tree

Try different values of K (depth of tree a.k.a. number of features the tree will split on) and compare the performance. Which feature gives the largest information gain? Which feature is the least useful for the decision tree?

[262]:
```
X_train, y_train, X_test, y_test = import_data(split=0.8)

tree = DecisionTree(K=3, verbose=False)
tree.buildTree(X_train, y_train)

tree.homework_evaluate(X_train, y_train, X_test, y_test)
```

```
data imported
The decision tree is 98 percent accurate on 1098 training data
The decision tree is 95 percent accurate on 274 test data
```

[262]:
```
0.9562043795620438
```

[263]:
```
tree_entropy = {}
print('Use Entropy as metric:')
for k in range(3,8):
    tree_entropy[k] = DecisionTree(K=k, verbose=False)
    tree_entropy[k].buildTree(X_train, y_train)
    print('K={}:'.format(k))
    tree_entropy[k].homework_evaluate(X_train, y_train, X_test, y_test)
    tree_entropy[k].evaluation()
```

```
Use Entropy as metric:
K=3:
The decision tree is 98 percent accurate on 1098 training data
The decision tree is 95 percent accurate on 274 test data
The feature frequency of depth 3 is {0: 5, 1: 3, 2: 3, 3: 1}.
K=4:
The decision tree is 99 percent accurate on 1098 training data
The decision tree is 98 percent accurate on 274 test data
The feature frequency of depth 4 is {0: 6, 1: 3, 2: 5, 3: 1}.
K=5:
The decision tree is 99 percent accurate on 1098 training data
The decision tree is 98 percent accurate on 274 test data
The feature frequency of depth 5 is {0: 6, 1: 4, 2: 5, 3: 1}.
K=6:
The decision tree is 99 percent accurate on 1098 training data
The decision tree is 98 percent accurate on 274 test data
The feature frequency of depth 6 is {0: 6, 1: 4, 2: 6, 3: 1}.
K=7:
The decision tree is 99 percent accurate on 1098 training data
The decision tree is 98 percent accurate on 274 test data
The feature frequency of depth 7 is {0: 6, 1: 4, 2: 6, 3: 1}.
```

[264]:
```
tree_gini = {}
print('Use Gini as metric:')
for k in range(3,8):
    tree_gini[k] = DecisionTree(K=k, verbose=False)
    tree_gini[k].buildTree(X_train, y_train, metric='gini')
    print('K={}:'.format(k))
    tree_gini[k].homework_evaluate(X_train, y_train, X_test, y_test)
    tree_gini[k].evaluation()
```

```
Use Gini as metric:
K=3:
The decision tree is 98 percent accurate on 1098 training data
The decision tree is 95 percent accurate on 274 test data
The feature frequency of depth 3 is {0: 5, 1: 3, 2: 3, 3: 1}.
K=4:
The decision tree is 99 percent accurate on 1098 training data
The decision tree is 98 percent accurate on 274 test data
The feature frequency of depth 4 is {0: 6, 1: 3, 2: 5, 3: 1}.
K=5:
The decision tree is 99 percent accurate on 1098 training data
The decision tree is 98 percent accurate on 274 test data
The feature frequency of depth 5 is {0: 6, 1: 4, 2: 5, 3: 1}.
K=6:
The decision tree is 99 percent accurate on 1098 training data
The decision tree is 98 percent accurate on 274 test data
The feature frequency of depth 6 is {0: 6, 1: 4, 2: 6, 3: 1}.
K=7:
The decision tree is 99 percent accurate on 1098 training data
The decision tree is 98 percent accurate on 274 test data
The feature frequency of depth 7 is {0: 6, 1: 4, 2: 6, 3: 1}.
```

We can see that adding depth of the trees can improve the accuracy of the models. The accuracy of the models using Gini is close to that using Entropy. Because the two metrics have similar trends, the results of the two metrics are similar. What's more, from the feature frequecy of the nodes we can see that the most useful features are 0 and 1. The least useful feature is 3.

2

3 Optional Decision Tree Exercise

This section is designed to give you some exposure to typical preprocessing and allow you to run your decision tree code on another example.

All of the preprocessing has been done for you — there's nothing you need to fill in, but it may be worthwhile to tinker with some of the pieces to make sure you understand how everything fits together.

Otherwise, if your decision tree code works on the bank notes example, you should be able to run through this straight away.

3.0.1 Step 1: Data Preprocessing

To start, you'll need to download the data files from https://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer/. The file `breast-cancer.data` contains the actual data you'll need and the file `breast-cancer.names` gives some information about the researchers and the data types (it's probably worth looking at to give you a sense of what's going on).

Note: If you try to open `breast-cancer.data` or `breast-cancer.names` directly, your computer might not know how to handle the file format. To view them, you need to change the file types from .data and .names to .txt — this can be accomplished by simply changing the file name from `breast-cancer.data` to `breast-cancer.txt`

Now let's load the data file into the notebook. All you need to do is put it in the same directory as the notebook and the cell below should locate the appropriate file.

[246]:
```
import os
current_directory = os.listdir()

try:
  cancer_files = [file for file in current_directory if 'cancer' in file]
  data_file = [file for file in cancer_files if 'names' not in file][0]
except IndexError:
  print('The breast cancer files were not found. Please upload again.')
  data_file = None

if data_file:
  print(f'The data file has been located as {data_file}.')
```

```
The data file has been located as breast-cancer.data.
```

!! If the cell above returned a file that you don't recognize as the correct data file, you must go back to the upload step and ensure you have successfully uploaded your files before proceeding. !!

Now we'll try to read the sample into our environment:

[247]:
```
samples = []

with open(data_file, 'r') as file:
  samples = file.readlines()

samples = [sample.split(',') for sample in samples]

print(f'There are {len(samples)} samples in the dataset')
```

```
There are 286 samples in the dataset
```

We can now start defining how we want to map our sample values to numeric features that can be used in the decision tree algorithm below. Let's first store our samples in a pandas DataFrame so that it'll be easier to view and work with.

[248]:
```
import pandas as pd
import numpy as np

columns = ['class', 'age', 'menopause', 'tumor-size', 'inv-nodes',
           'node-caps', 'deg-malig', 'breast', 'breast-quad', 'irradiat']

samples_df = pd.DataFrame(samples, columns=columns)
samples_df['irradiat'] = samples_df['irradiat'].str.replace('\\n', '')

samples_df
```

[248]:
```
                    class    age menopause tumor-size inv-nodes node-caps  \
0    no-recurrence-events  30-39   premeno      30-34       0-2        no
1    no-recurrence-events  40-49   premeno      20-24       0-2        no
2    no-recurrence-events  40-49   premeno      20-24       0-2        no
3    no-recurrence-events  60-69      ge40      15-19       0-2        no
4    no-recurrence-events  40-49   premeno        0-4       0-2        no
..                    …      …       …        …       …         …
281     recurrence-events  30-39   premeno      30-34       0-2        no
282     recurrence-events  30-39   premeno      20-24       0-2        no
283     recurrence-events  60-69      ge40      20-24       0-2        no
284     recurrence-events  40-49      ge40      30-34       3-5        no
285     recurrence-events  50-59      ge40      30-34       3-5        no

    deg-malig breast breast-quad irradiat
0           3   left    left_low       no
1           2  right    right_up       no
2           2   left    left_low       no
3           2  right     left_up       no
4           2  right   right_low       no
..        …     …         …       …
281         2   left     left_up       no
282         3   left     left_up      yes
283         1  right     left_up       no
284         3   left    left_low       no
285         3   left    left_low       no

[286 rows x 10 columns]
```

Our DataFrame looks great! Now that we have the raw data stored in an interpretable way, it's good practice to make a copy before trying to encode everything — if something goes wrong, it'll be easy to just come back up here and reset the encoded DataFrame.

[249]:
```
encoded_samples_df = samples_df.copy()
```

In the following few cells, we're going to define the different types of variables that exist in our dataset and make sure we assign the appropriate type of encoding.

To start, we'll define our binary variables (0,1) and create Python dictionaries to map the string labels to binary integer values.

It's also helpful to store all of the columns and dictionaries in two lists so that we can easily reference them later on.

[250]:
```
binary_cols = ['class', 'breast', 'irradiat']

class_map = {'no-recurrence-events': 0, 'recurrence-events': 1}
breast_map = {'left': 0, 'right': 1}
irrad_map = {'yes': 1, 'no': 0}

binary_maps = [class_map, breast_map, irrad_map]
```

Next, we've defined our ordinal variables (those that have obvious ordered structure). Instead of hard-coding the maps here, it's nice to have a function that can take any number of ordinal columns and instantly create all of the corresponding maps. The code below does that exactly, relying on the integer value of the first number in the range (i.e. '50-65') as the sorting key.

[251]:
```
ordinal_cols = ['age', 'tumor-size', 'inv-nodes', 'deg-malig']

def first_val(x):
  return eval(x.split('-')[0].replace("'", ''))

ordinal_maps = {}

for col in ordinal_cols:
  uniques = sorted(list(encoded_samples_df[col].unique()), key=first_val)
  val_map = {}
  i = 1

  for val in uniques:
    val_map[val] = i
    i += 1

  ordinal_maps[col] = val_map
```

Finally, we have several columns that take n > 2 discrete values. Binary encoding won't work here, so we'll use one-hot encoding. Just like in the cell immediately above, we'll use a function here to take an arbitrary number of columns and encode their unique values as one-hot vectors.

[252]:
```
one_hot_cols = ['menopause', 'breast-quad', 'node-caps']
one_hot_maps = {}

def one_hot_map(col, df):
  one_hot = {}
  unique_vals = list(df[col].unique())
  one_hot_vecs = np.identity(len(unique_vals))
  for i in range(len(unique_vals)):
    one_hot[unique_vals[i]] = one_hot_vecs[i]

  return one_hot

for col in one_hot_cols:
  one_hot_maps[col] = one_hot_map(col, encoded_samples_df)
```

Now that we have all of our encoding maps set up, we'll zip them all up into a master dictionary so that we can easily iterate through them on our `encoded_samples_df` DataFrame.

[253]:
```
all_maps = dict(one_hot_maps, **ordinal_maps)
for col, val_map in zip(binary_cols, binary_maps):
  all_maps[col] = val_map
```

Before proceeding, let's double check to make sure you've captured all of the columns:

[254]:
```
if len(all_maps) != len(columns):
  print("You're missing a map! Go back and double check that you didn't miss a␣
 ↪column.")
else:
  print("Looks like you successfully created all the maps! Nice work!")
```

```
Looks like you successfully created all the maps! Nice work!
```

And now the moment of truth! Let's apply all of our encoding maps on the DataFrame to turn everything into useable features for our decision trees algorithm.

[255]:
```
for col in columns:
  encoded_samples_df[col] = encoded_samples_df[col].map(all_maps[col])

encoded_samples_df
```

[255]:
```
     class  age        menopause  tumor-size  inv-nodes        node-caps  \
0        0    2  [1.0, 0.0, 0.0]           7          1  [1.0, 0.0, 0.0]
1        0    3  [1.0, 0.0, 0.0]           5          1  [1.0, 0.0, 0.0]
2        0    3  [1.0, 0.0, 0.0]           5          1  [1.0, 0.0, 0.0]
3        0    5  [0.0, 1.0, 0.0]           4          1  [1.0, 0.0, 0.0]
4        0    3  [1.0, 0.0, 0.0]           1          1  [1.0, 0.0, 0.0]
..     …    …              …          …          …                …
281      1    2  [1.0, 0.0, 0.0]           7          1  [1.0, 0.0, 0.0]
282      1    2  [1.0, 0.0, 0.0]           5          1  [1.0, 0.0, 0.0]
283      1    5  [0.0, 1.0, 0.0]           5          1  [1.0, 0.0, 0.0]
284      1    3  [0.0, 1.0, 0.0]           7          2  [1.0, 0.0, 0.0]
285      1    4  [0.0, 1.0, 0.0]           7          2  [1.0, 0.0, 0.0]

     deg-malig  breast                     breast-quad  irradiat
0            3       0  [1.0, 0.0, 0.0, 0.0, 0.0, 0.0]         0
```

```
1             2       1  [0.0, 1.0, 0.0, 0.0, 0.0, 0.0]         0
2             2       0  [1.0, 0.0, 0.0, 0.0, 0.0, 0.0]         0
3             2       1  [0.0, 0.0, 1.0, 0.0, 0.0, 0.0]         0
4             2       1  [0.0, 0.0, 0.0, 1.0, 0.0, 0.0]         0
..          …     …                              …        …
281           2       0  [0.0, 0.0, 1.0, 0.0, 0.0, 0.0]         0
282           3       0  [0.0, 0.0, 1.0, 0.0, 0.0, 0.0]         1
283           1       1  [0.0, 0.0, 1.0, 0.0, 0.0, 0.0]         0
284           3       0  [1.0, 0.0, 0.0, 0.0, 0.0, 0.0]         0
285           3       0  [1.0, 0.0, 0.0, 0.0, 0.0, 0.0]         0

[286 rows x 10 columns]
```

Double check your `encoded_samples_df` here to make sure everything passes the sniff test!

Assuming it all looks good, the final step is to break up our DataFrame into individual samples, unpack the nested arrays into their constituent values, and concatenate everything together into a final sample vector…

```
[256]: sample_list = list(encoded_samples_df.to_numpy())
```

```
[257]: def unpack_nested_arrays(vec):
         final_vec = np.array([])

         for e in vec:
           if isinstance(e, int):
             e = np.array([e])
           final_vec = np.concatenate((final_vec, e))

         return final_vec
```

```
[258]: final_vecs = []
       for e in sample_list:
         final_vecs.append(unpack_nested_arrays(e))

       final_vecs = np.array(final_vecs, dtype=int)
```

Great! The `final_vecs` variable should now point to a 286x19 numpy.ndarray containing all of our samples. Let's finally move on to the machine learning!

3.0.2 Step 2: Decision Tree

We'll run the decision tree process here again.

```
[259]: def import_cancer_data(split=0.8, shuffle=True, CUTOFF=0, bins = 256):

           if shuffle:
             np.random.shuffle(final_vecs)
```

```
    num_samples = final_vecs.shape[0]
    num_train_samples = int(num_samples*split)

    train_data, test_data = final_vecs[:num_train_samples, :],␣
↪final_vecs[num_train_samples:, :]

    X_train = train_data[:, 1:]
    y_train = train_data[:, 0]
    X_test = test_data[:, 1:]
    y_test = test_data[:, 0]

    print(X_train.shape)
    print(y_train.shape)

    return X_train, y_train, X_test, y_test
```

Here's the final fit/evaluation code again, but this time using the breast cancer data. You should get somewhere around 0.75 accuracy for the decision tree on this dataset — not 100%, but not too bad for the humble decision tree!

[261]:
```
X_train, y_train, X_test, y_test = import_cancer_data(split=0.8)

tree = DecisionTree(K=3, verbose=False)
tree.buildTree(X_train, y_train)

tree.homework_evaluate(X_train, y_train, X_test, y_test)
```

```
(228, 18)
(228,)
The decision tree is 65 percent accurate on 228 training data
The decision tree is 68 percent accurate on 58 test data
```

[261]:
```
0.6896551724137931
```