Lab 9 - ML Programming

January 21, 2022

1 EXERCISE 1

1.1 Implement Decision Trees

1.2 PART A: Basic Working with MCR

[4]:
```
## Pick one of 2 datasets. I'll be working with the iris dataset

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import random
from pprint import pprint

iris = pd.read_csv('iris.data',names=['sepal length','sepal width','petal␣
↪length','petal width','class'])
```

[94]:
```
##  Split data into three parts: train, validation and test (70%, 15% and 15%␣
↪respectively)

iris_train = iris.sample(frac=0.7,random_state=3116)
iris_leftover = iris.drop(iris_train.index)
iris_validation = iris_leftover.sample(frac=0.5,random_state=3116)
iris_test = iris_leftover.drop(iris_validation.index)

print(iris_train.shape,iris_validation.shape,iris_test.shape)
```

```
(105, 5) (22, 5) (23, 5)
```

[100]:
```
## Using the train data build a decision tree. Use Misclassification Rate (MCR)␣
↪as a Quality-criterion.

## Code follows examples provided at https://github.com/SebastianMantey/
↪Decision-Tree-from-Scratch/blob/master/notebooks/decision_tree_functions.py
def check_unique(data):
    label_column = data[:, -1]
    unique_classes = np.unique(label_column)
    if len(unique_classes) == 1:
```

```
        return True
    else:
        return False

def create_leaf(data):
    label_column = data[:, -1]
    unique_classes, counts_unique_classes = np.unique(label_column,␣
 ↪return_counts=True)
    index = counts_unique_classes.argmax()
    leaf = unique_classes[index]
    return leaf

def get_potential_splits(data):
    potential_splits = {}
    _, n_columns = data.shape
    for column_index in range(n_columns - 1):
        values = data[:, column_index]
        unique_values = np.unique(values)
        potential_splits[column_index] = unique_values
    return potential_splits

def misclassification_rate(data):
    actual_values = data[:, -1]
    if len(actual_values) == 0:
        misclassified = 0
    else:
        prediction = np.mean(actual_values)
        misclassifiedd = np.mean((actual_values - prediction) **2)
    return misclassified

def calculate_overall_metric(data_below, data_above, metric_function):
    n = len(data_below) + len(data_above)
    p_data_below = len(data_below) / n
    p_data_above = len(data_above) / n
    overall_metric =  (p_data_below * metric_function(data_below)
                     + p_data_above * metric_function(data_above))
    return overall_metric

def determine_best_split(data, potential_splits):
    first_iteration = True
    for column_index in potential_splits:
        for value in potential_splits[column_index]:
            data_below, data_above = split_data(data,␣
 ↪split_column=column_index, split_value=value)
            current_overall_metric = calculate_overall_metric(data_below,␣
 ↪data_above, metric_function=misclassification_rate)
            if first_iteration or current_overall_metric <= best_overall_metric:
```

```python
                first_iteration = False
                best_overall_metric = current_overall_metric
                best_split_column = column_index
                best_split_value = value
    return best_split_column, best_split_value

def split_data(data, split_column, split_value):
    split_column_values = data[:, split_column]
    type_of_feature = FEATURE_TYPES[split_column]
    if type_of_feature == "continuous":
        data_below = data[split_column_values <= split_value]
        data_above = data[split_column_values >  split_value]
    else:
        data_below = data[split_column_values == split_value]
        data_above = data[split_column_values != split_value]
    return data_below, data_above

def determine_type_of_feature(df):
    feature_types = []
    n_unique_values_treshold = 15
    for feature in df.columns:
        if feature != "label":
            unique_values = df[feature].unique()
            example_value = unique_values[0]
            if (isinstance(example_value, str)) or (len(unique_values) <=␣
 ↪n_unique_values_treshold):
                feature_types.append("categorical")
            else:
                feature_types.append("continuous")
    return feature_types

def decision_tree_algorithm(df, counter=0):
    if counter == 0:
        data = df.values
        global COLUMN_HEADERS, FEATURE_TYPES
        COLUMN_HEADERS = df.columns
    else:
        data = df
    if (check_unique(data)):
        leaf = create_leaf(data, ml_task)
        return leaf
    else:
        counter += 1
        potential_splits = get_potential_splits(data)
        split_column, split_value = determine_best_split(data, potential_splits)
        data_below, data_above = split_data(data, split_column, split_value)
        feature_name = COLUMN_HEADERS[split_column]
```

```
        question = "{} <= {}".format(feature_name, split_value)
        sub_tree = {question: []}
        yes_answer = dtree(data_below, counter)
        no_answer = dtree(data_above, counter)
        sub_tree[question].append(yes_answer)
        sub_tree[question].append(no_answer)
        return sub_tree
```

[102]:
```
## Plotting: At each decision step/split present probability of each class␣
↪using histogram (properly labeled figure)
```

[ ]:
```
## Plotting: Print your tree using a breath first tree traversal.
```

[ ]:
```
## On the validation-set measure the cross entropy loss (i.e. logloss)
```

1.3 PART B: Experimenting w/ Other Quality Criterion

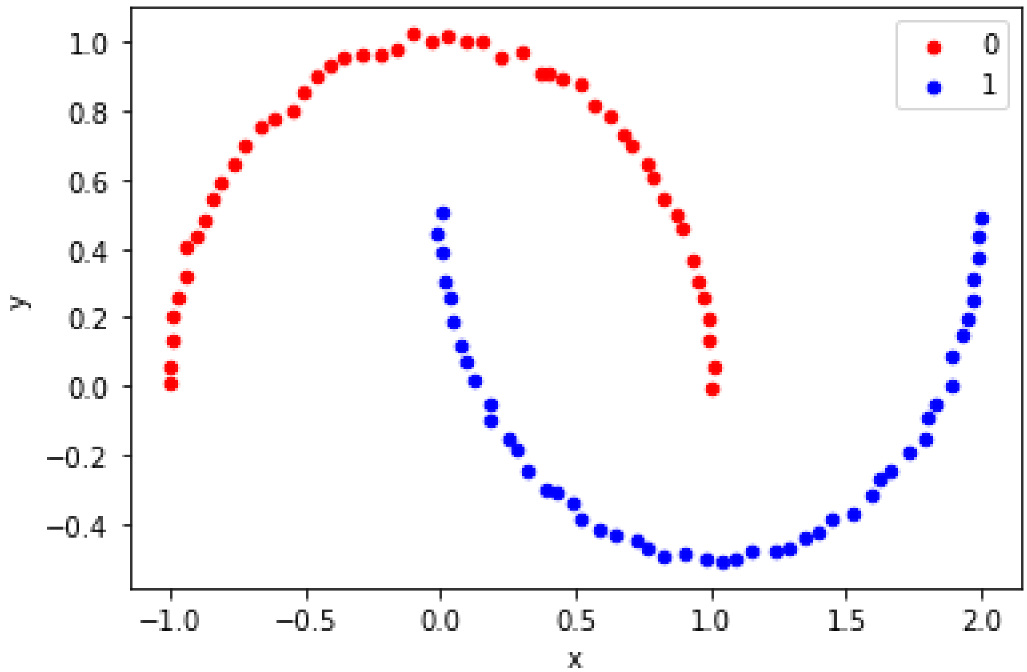
[ ]:
```
## Modify the Quality-criterion to Information Gain
## At each decision step, plot the Information Gain

def compute_entropy(data):
    label_column = data[:, -1]
    _, counts = np.unique(label_column, return_counts=True)
    probabilities = counts / counts.sum()
    entropy = sum(probabilities * -np.log2(probabilities))
    return entropy

def determine_best_split(data, potential_splits):
    first_iteration = True
    for column_index in potential_splits:
        for value in potential_splits[column_index]:
            data_below, data_above = split_data(data,␣
↪split_column=column_index, split_value=value)
            current_overall_metric = calculate_overall_metric(data_below,␣
↪data_above, metric_function=compute_entropy)
            if first_iteration or current_overall_metric <= best_overall_metric:
                first_iteration = False
                best_overall_metric = current_overall_metric
                best_split_column = column_index
                best_split_value = value
    return best_split_column, best_split_value
```

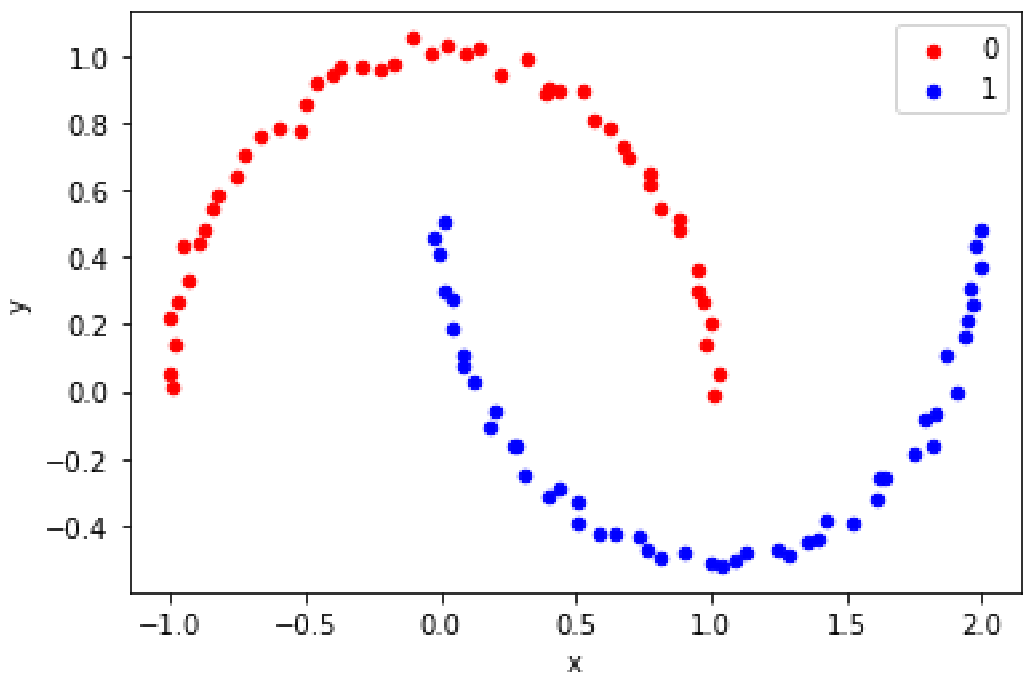
[ ]:
```
## Compare the validation set results for both Quality-criterion, output one␣
↪value for test-set.
```

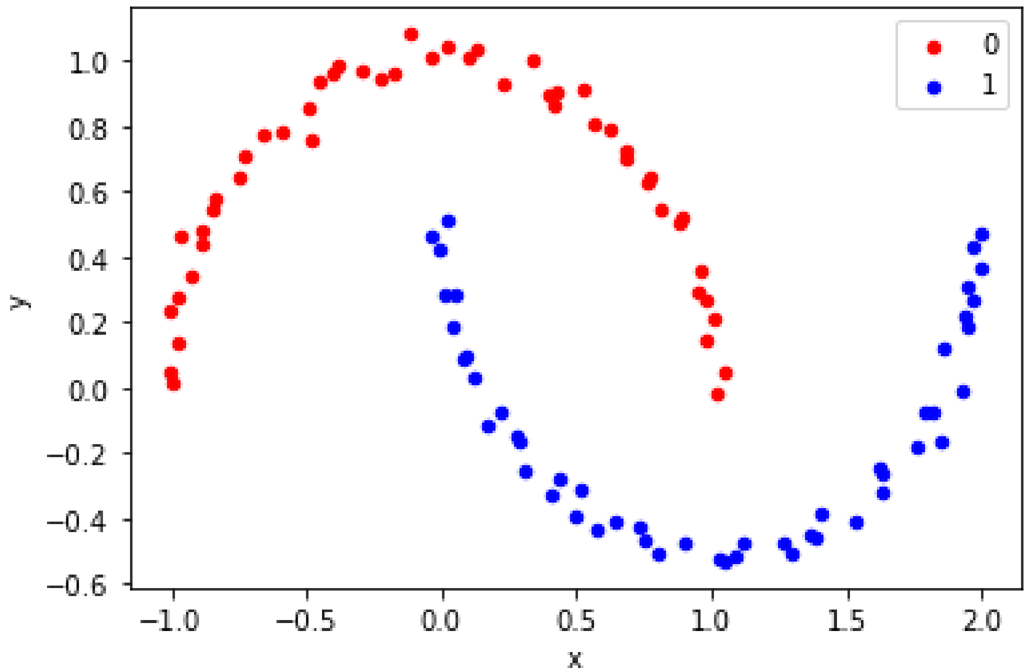
2 EXERCISE 2

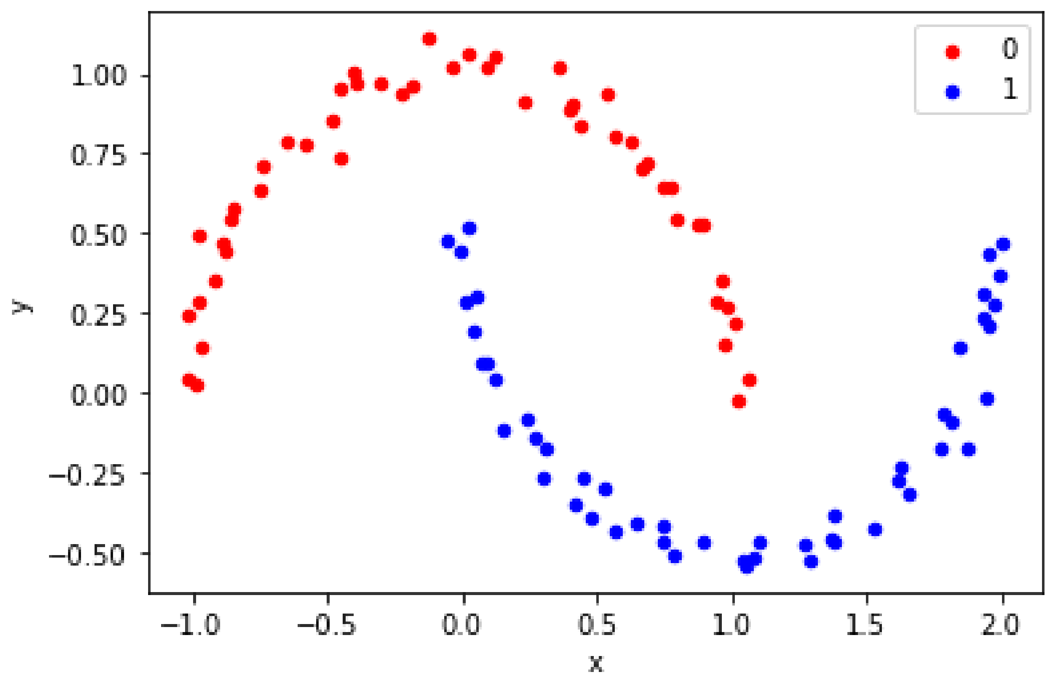
2.1 Gradient Boosted Decision Trees

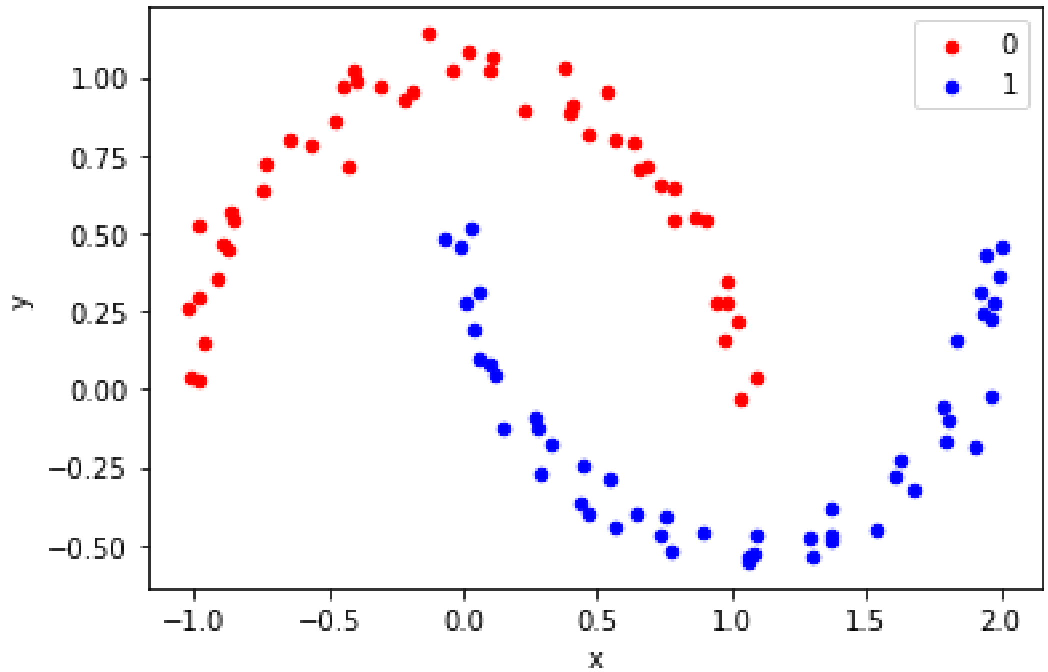
[60]:
```
## Generate a binary classification toy dataset from the scikit-learn utility␣
↪"make-moons"
## Generate 100 samples, for 10 different levels of noise which should give you␣
↪a toy-dataset of 1000 samples
## Visualize the 10 different pairs of so-called moons

from sklearn.datasets import make_moons

df_list = []

for i in range(1,11):
    X, y = make_moons(n_samples=100,noise=(0.01*i),random_state=3116)
    df = pd.DataFrame(dict(x=X[:,0],y=X[:,1],label=y))
    df_list.append(df)
    colors = {0:'red',1:'blue'}
    fig,axs = plt.subplots()
    grouped = df.groupby('label')
    for key,group in grouped:
        group.
↪plot(ax=axs,kind='scatter',x='x',y='y',label=key,color=colors[key])
    plt.show()

moons = pd.concat(df_list,ignore_index=True)
len(moons)
```


0
1
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
y
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
x

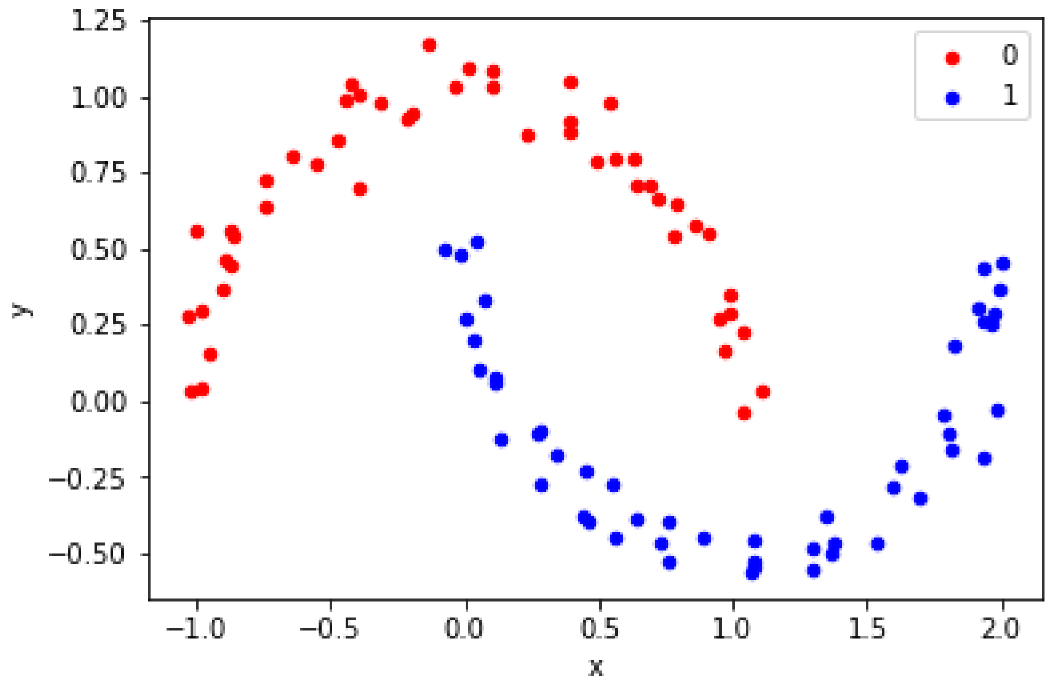

0
1
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
y
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
x


0
1
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
y
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
x

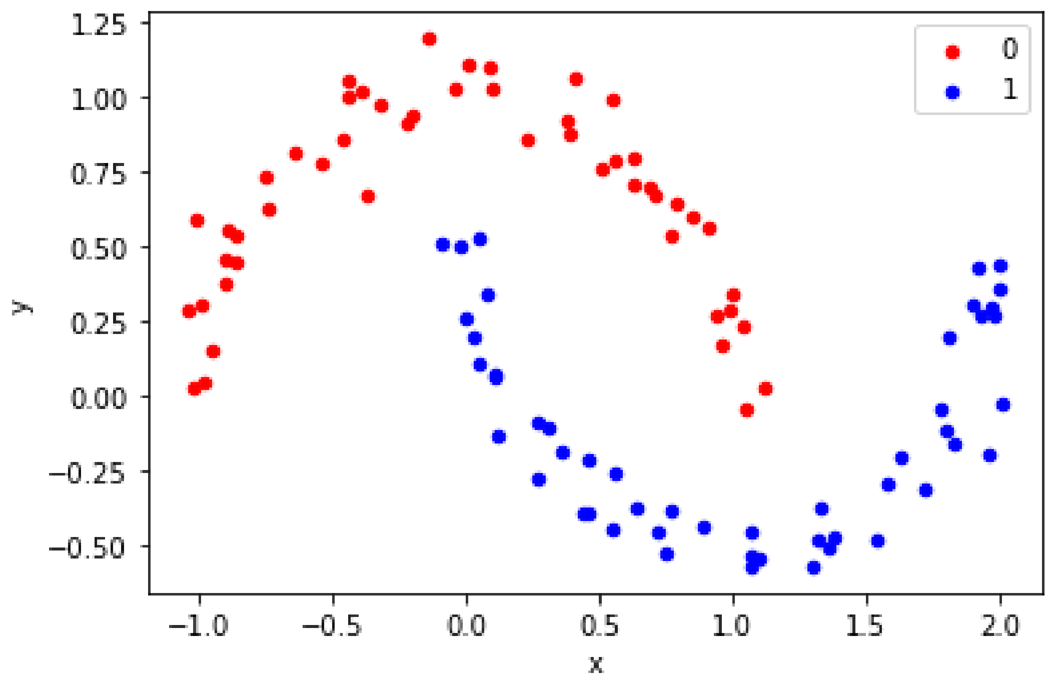

0
1
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
y
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
x



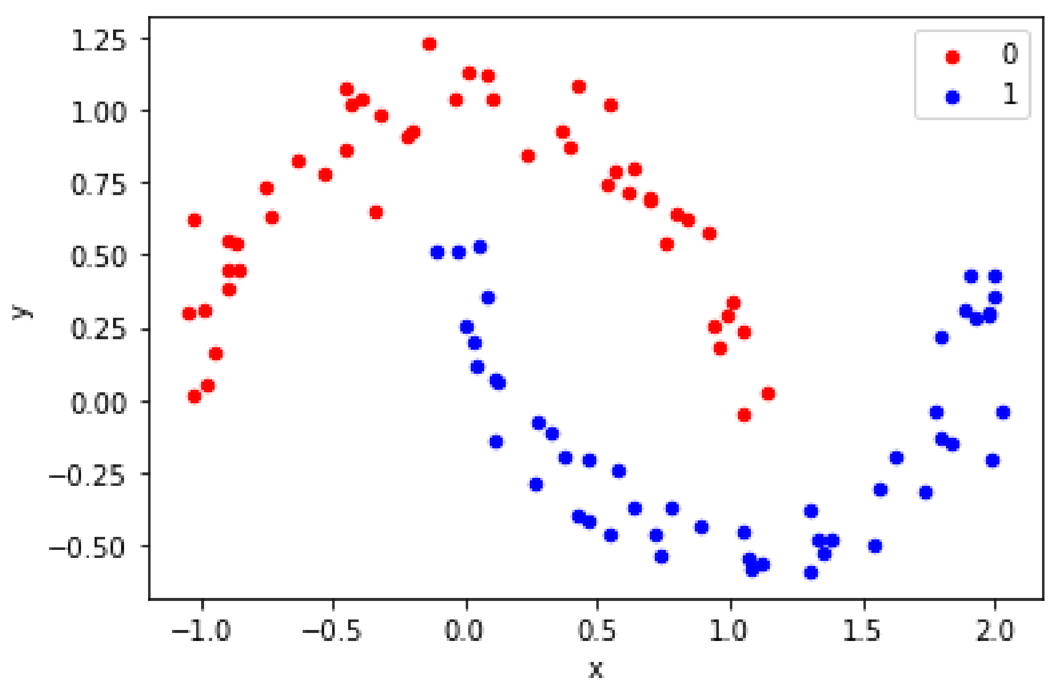
0
1
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
y
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
x



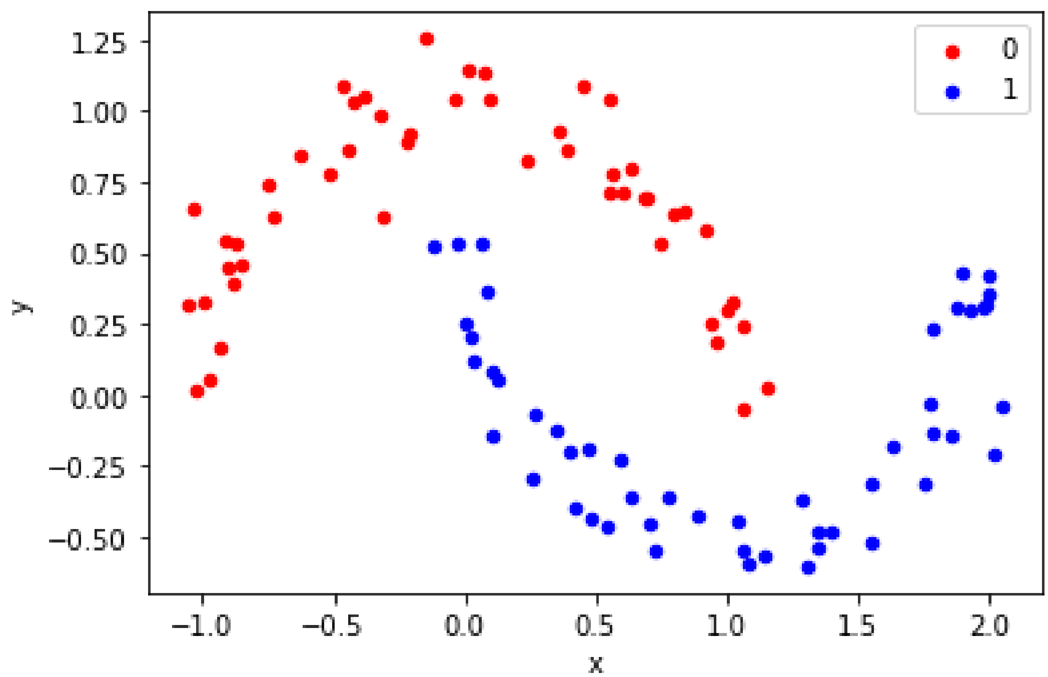
0
1
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
y
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
x



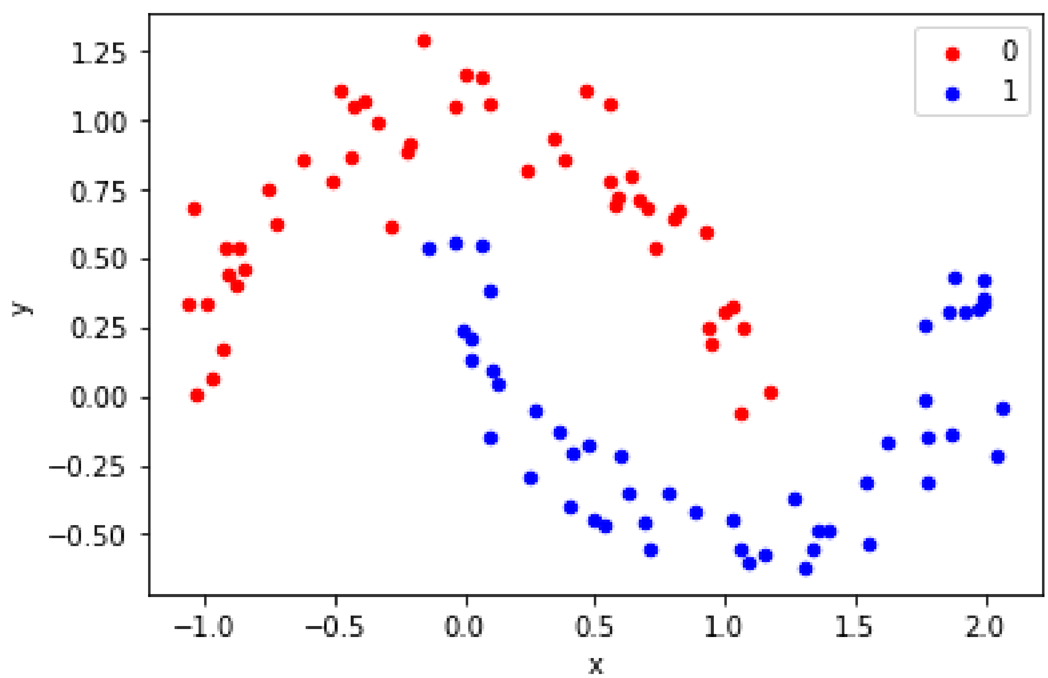
0
1
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
y
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
x



0
1
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
y
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
x



0
1
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
y
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
x



0
1
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
y
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
x

[60]: 1000

[85]:
```
## Generate train/validation/test splits with the ratios like before

moons_train = moons.sample(frac=0.7,random_state=3116)
moons_leftover = moons.drop(moons_train.index)
moons_validation = moons_leftover.sample(frac=0.5,random_state=3116)
moons_test = moons_leftover.drop(moons_validation.index)

print(moons_train.shape,moons_validation.shape,moons_test.shape)
```

```
(700, 3) (150, 3) (150, 3)
```

[88]:
```
## Keep max depth of trees to 2 (i.e root node then leaf nodes (also called␣
↪stumps))
## Tune number of trees in the ensemble on the validation set

def gd_boosted_tree(df, counter=0, min_samples=2, max_depth=2):
    if counter == 0:
        global COLUMN_HEADERS, FEATURE_TYPES
        COLUMN_HEADERS = df.columns
        FEATURE_TYPES = determine_type_of_feature(df)
        data = df.values
    else:
        data = df
    if (check_unique(data)) or (len(data) < min_samples) or (counter ==␣
↪max_depth):
        leaf = create_leaf(data)
        return leaf
    else:
        counter += 1
        potential_splits = get_potential_splits(data)
        split_column, split_value = determine_best_split(data, potential_splits)
        data_below, data_above = split_data(data, split_column, split_value)
        if len(data_below) == 0 or len(data_above) == 0:
            leaf = create_leaf(data)
            return leaf
        feature_name = COLUMN_HEADERS[split_column]
        type_of_feature = FEATURE_TYPES[split_column]
        if type_of_feature == "continuous":
            question = "{} <= {}".format(feature_name, split_value)
        else:
            question = "{} = {}".format(feature_name, split_value)
        sub_tree = {question: []}
        yes_answer = decision_tree_algorithm(data_below, counter, min_samples,␣
↪max_depth)
        no_answer = decision_tree_algorithm(data_above, counter, min_samples,␣
↪max_depth)
        if yes_answer == no_answer:
```

```
            sub_tree = yes_answer
        else:
            sub_tree[question].append(yes_answer)
            sub_tree[question].append(no_answer)
        return sub_tree
```

[92]:

```
## Report test-accuracy

## Code taken from ML homework trees.ipynb
def predict_example(example, tree):
    if not isinstance(tree, dict):
        return tree
    question = list(tree.keys())[0]
    feature_name, comparison_operator, value = question.split(" ")
    if comparison_operator == "<=":
        if example[feature_name] <= float(value):
            answer = tree[question][0]
        else:
            answer = tree[question][1]
    else:
        if str(example[feature_name]) == value:
            answer = tree[question][0]
        else:
            answer = tree[question][1]
    if not isinstance(answer, dict):
        return answer
    else:
        residual_tree = answer
        return predict_example(example, residual_tree)

def make_predictions(df, tree):
    if len(df) != 0:
        predictions = df.apply(predict_example, args=(tree,), axis=1)
    else:
        predictions = pd.Series()
    return predictions

def calculate_accuracy(df, tree):
    predictions = make_predictions(df, tree)
    predictions_correct = predictions == df.label
    accuracy = predictions_correct.mean()
    return accuracy

accuracy = calculate_accuracy(moons_validation,gd_boosted_tree)
print(accuracy)
```

```
93.7
```

[ ]:

```

```