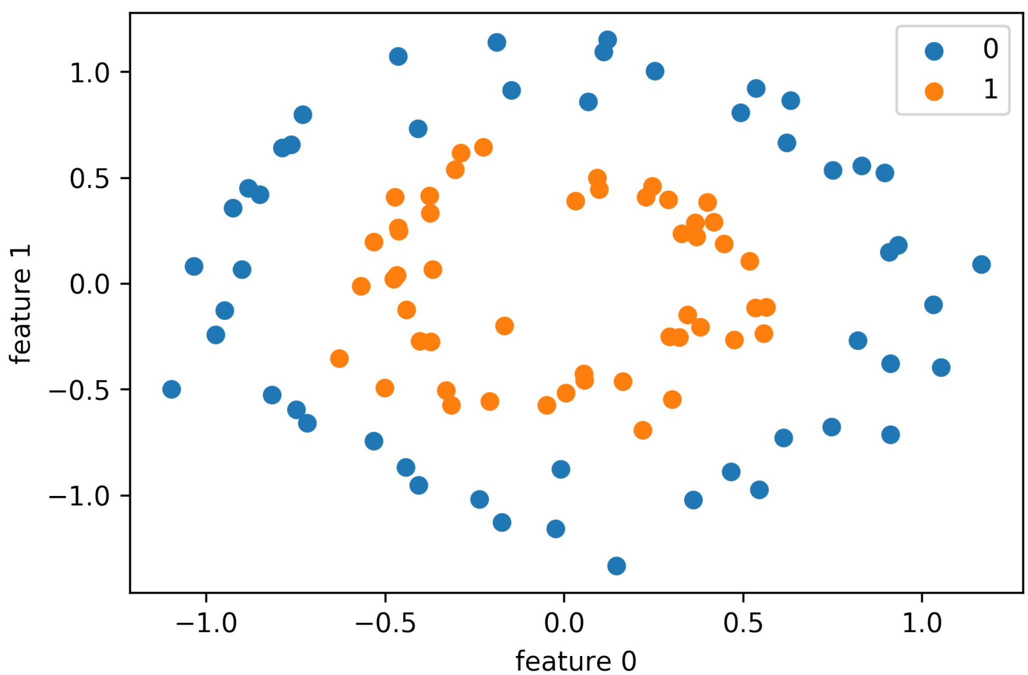
Random Forest Challenge

December 2, 2020

In [1]:
```python
import numpy as np  # to build the algorithm
import matplotlib.pyplot as plt  # to visualize
from sklearn.datasets import make_circles  # to generate a dataset
```

In [2]:
```python
# Generate a dataset
X, y = make_circles(n_samples=100, noise=0.1, factor=0.5, random_state=0)
plt.figure(dpi=200)
plt.scatter(X[:, 0][y == 0], X[:, 1][y == 0], label=0)
plt.scatter(X[:, 0][y == 1], X[:, 1][y == 1], label=1)
plt.xlabel('feature 0')
plt.ylabel('feature 1')
plt.legend()
```

Out[2]: <matplotlib.legend.Legend at 0x11ca3def0>



In [3]:
```python
# Train the model
from sklearn.ensemble import RandomForestClassifier

clf = RandomForestClassifier(n_estimators=30)
clf.fit(X, y)
```

Out[3]:
```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
            max_depth=None, max_features='auto', max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, n_estimators=30, n_jobs=None,
            oob_score=False, random_state=None, verbose=0,
            warm_start=False)
```

In [4]:
```python
# Test the model
X_test, y_test = make_circles(n_samples=100, noise=0.1, factor=0.5, random_state=42)
y_pred = clf.predict(X_test)
acc = sum(y_pred == y_test)/len(y_test)
print('Accuracy: ', acc)
```

```
Accuracy:  0.97
```

In [5]:
```python
clf.feature_importances_
```

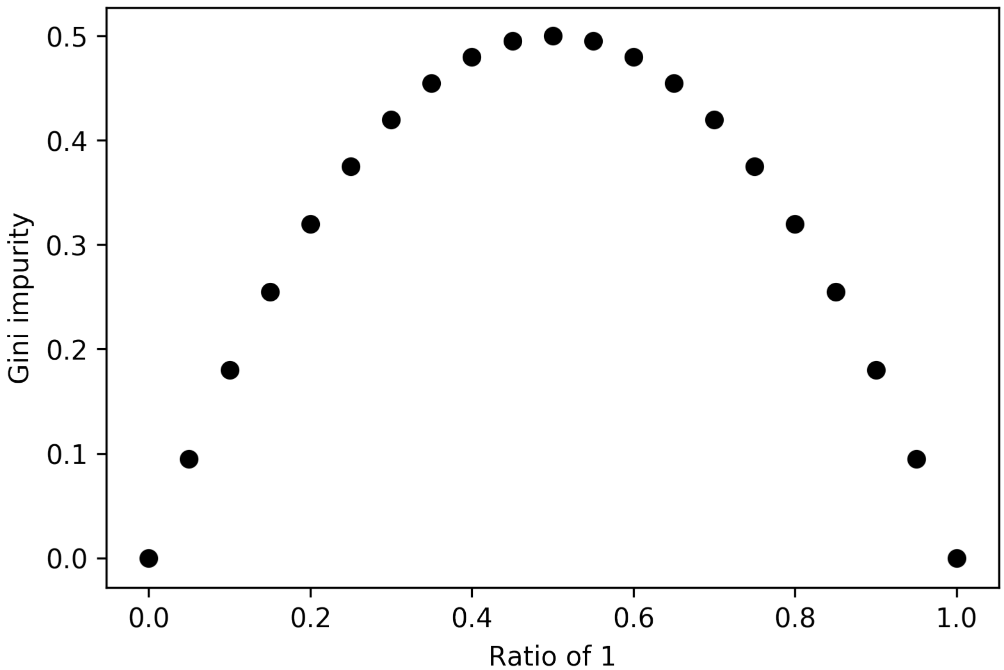
Out[5]:
```
array([0.48581498, 0.51418502])
```

In [6]:
```python
# Challenge 1
def gini_calculator(y):
    '''
    Calculates the gini impurity of a set
    Arguments
        y: np.array() containing the labels
    Returns
        gini: 1 - p0^2 - p1^2
            p0 ratio of class 0
            p1 ratio of class 1
    '''




    return gini
```

In [7]:
```python
# Sanity check
num_datapoints = 20
plt.figure(dpi=200)
for i in range(num_datapoints+1):
    num_ones = i
```

```
        num_zeros = num_datapoints-i
        p1 = num_ones/num_datapoints
        combined_set = np.concatenate((np.ones(num_ones), np.zeros(num_zeros)))
        gini = gini_calculator(combined_set)
        plt.scatter(p1, gini, color='k')
    plt.xlabel('Ratio of 1')
    plt.ylabel('Gini impurity')
```

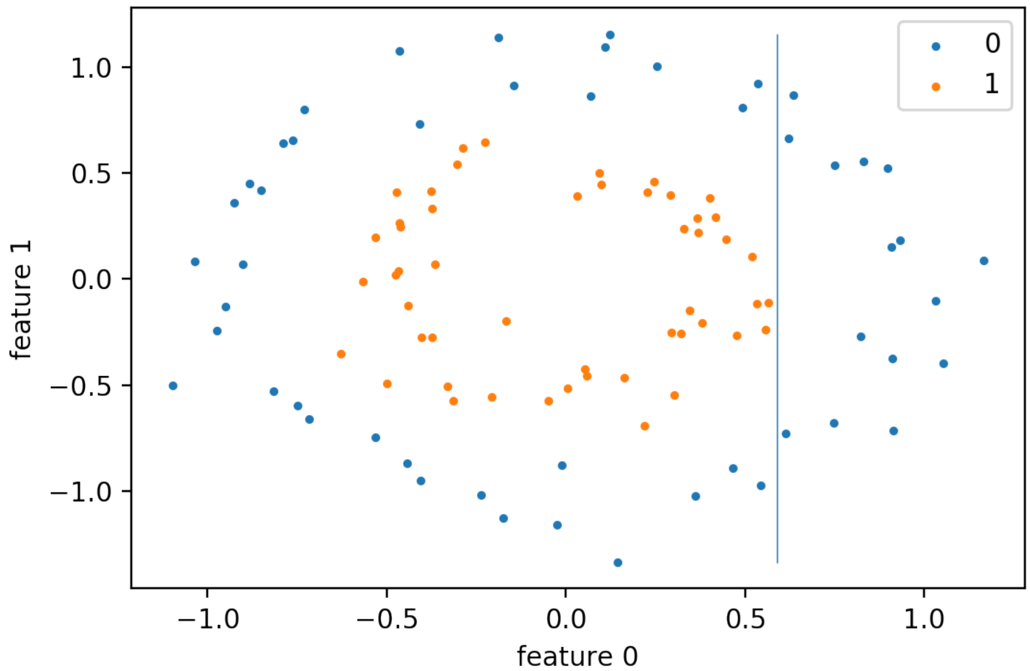
Out[7]: Text(0, 0.5, 'Gini impurity')




```
In [8]: def gini_of_a_split(y1, y2):
            '''
            Weighted average gini of two sets
            Arguments
                y1: np.array() containing the labels of set 1
                y2: np.array() containing the labels of set 2
            Returns
                avg_gini: Weighted average gini of y1 and y2
            '''
            g1 = gini_calculator(y1)
            w1 = len(y1)/(len(y1)+len(y2))
            g2 = gini_calculator(y2)
            w2 = len(y2)/(len(y1)+len(y2))
            avg_gini = g1*w1 + g2*w2
            return avg_gini
```

In [9]:
```python
# Challenge 2
from operator import itemgetter


def split_finder(X, y):
    '''
    Finds the best split that minimizes the average gini.
    Best split is defined by a feature index and its value.

    Arguments
        m = num_of_datapoints
        n = num_of_features
        X: np.array() shape (m, n)
        y: np.array() shape (m, 1)

    Returns
        best_split_feature: integer, best feature index
        best_split_value: float, best value

    '''





    return best_split_feature, best_split_value
```

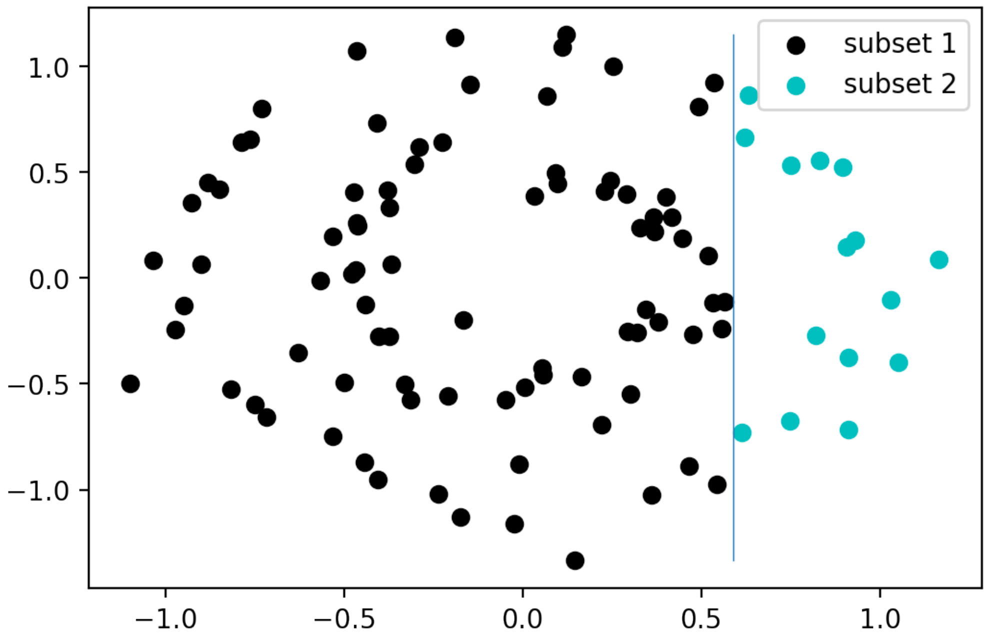
In [10]:
```python
# Visualize the first split
best_split = split_finder(X, y)
plt.figure(dpi=200)
plt.scatter(X[:, 0][y == 0], X[:, 1][y == 0], label=0, s=5)
plt.scatter(X[:, 0][y == 1], X[:, 1][y == 1], label=1, s=5)
plt.xlabel('feature 0')
plt.ylabel('feature 1')
plt.legend()

split_feature = best_split[0]
split_value = best_split[1]
boundary_limits = [min(X[:, 1]), max(X[:, 1])]

if split_feature == 0:
    plt.plot([split_value, split_value], boundary_limits, lw=.5)
if split_feature == 1:
    plt.plot(boundary_limits, [split_value, split_value], lw=.5)
```




```
In [11]: def splitter(X, y):
             '''
             This is one node split i.e. building block of a tree.

             Given X and y,
             - finds the split
             - splits the datasets into 2 subsets
             - returns the subsets and the split

             Arguments
                 m = num_of_datapoints
                 n = num_of_features
                 X: np.array() shape (m, n)
                 y: np.array() shape (m, 1)

             Returns
                 subset1: list of np arrays, [X1, y1]
                     X1: np.array() shape (m1, n)
                     y1: np.array() shape (m1, 1)
                 subset2: list of np arrays, [X2, y2]
                     X2: np.array() shape (m2, n)
                     y2: np.array() shape (m2, 1)
                 where m = m1 + m2

                 split: a tuple, (split_feature, split_value)
```

```
            split_feature: integer, best feature index
            split_value: float, best value
    '''
    split = split_finder(X, y)

    X1 = X[X[:, split[0]] < split[1]]
    y1 = y[X[:, split[0]] < split[1]]

    X2 = X[X[:, split[0]] > split[1]]
    y2 = y[X[:, split[0]] > split[1]]

    return (X1, y1), (X2, y2), split
```

In [12]:
```
# Visualize the split and the subsets
subset1, subset2, split = splitter(X, y)
X1 = subset1[0]
X2 = subset2[0]

plt.figure(dpi=200)
plt.scatter(X1[:, 0], X1[:, 1], color='k', label='subset 1')
plt.scatter(X2[:, 0], X2[:, 1], color='c', label='subset 2')
plt.legend()

split_feature = split[0]
split_value = split[1]
boundary_limits = [min(X[:, 1]), max(X[:, 1])]

if split_feature == 0:
    plt.plot([split_value, split_value], boundary_limits, lw=.5)
if split_feature == 1:
    plt.plot(boundary_limits, [split_value, split_value], lw=.5)
```



In [13]:
```python
# Challenge 3
def fit_tree(X, y):
    '''
    Repeat the splitter to fit a tree to X and y.
    return the tree i.e. the trained model

    Arguments
        m = num_of_datapoints
        n = num_of_features
        X: np.array() shape (m, n)
        y: np.array() shape (m, 1)

    Returns
        tree: a dictionary of nodes
            key: node name e.g. 'root', 'rootRL'
            value: node dictionary
                key:
                    'depth' int
                    'data' (subset_X, subset_y)
                    'class' majority class
                    'leaf' binary, leaf(1) or not(0)
                    'split_feature' 0 or 1
                    'split_value' float
    '''
```

```
        return tree
```

```
In [14]: tree = fit_tree(X, y)
```

```
In [15]: # Challenge 4
         def predict_tree(X, tree):
             '''
             Given X and the model (i.e. tree), return predictions

             Arguments
                 X: np.array() shape (m, n)
                 tree: a dictionary of nodes

             Returns
                 y_pred: a list of predictions for each row of X,
                         class labels 0 or 1.


             '''



             return y_pred
```

```
In [16]: y_pred = predict_tree(X, tree)
```

```
In [17]: def accuracy(y_pred, y):
             return sum(y_pred == y)/len(y)
```

```
In [18]: accuracy(y_pred, y)
```

```
Out[18]: 1.0
```

```
In [19]: # Putting all together
         # with Train/Test
         X_train, y_train = make_circles(n_samples=100, noise=0.1, factor=0.5, random_state=0)
         X_test, y_test = make_circles(n_samples=100, noise=0.1, factor=0.5, random_state=42)
         tree = fit_tree(X_train, y_train)
         y_pred_train = predict_tree(X_train, tree)
         y_pred_test = predict_tree(X_test, tree)

         print('Training acc:', accuracy(y_pred_train, y_train))
         print('Testing acc:', accuracy(y_pred_test, y_test))
```

```
Training acc: 1.0
Testing acc: 0.94
```

```
In [24]: # Challenge 5
         def fit_forest(X, y):
```

```
    '''
    Fit 30 trees
    by randomly sampling from
    X and y
    return 30 trees

    Arguments
        X: np.array() shape (m, n)
        y: np.array() shape (m, 1)

    Returns
        forest: a list of trees

    '''
    num_trees = 30
    forest = []
    for i in range(num_trees):
        X_sample =
        y_sample =
        tree = fit_tree(X_sample, y_sample)
        forest.append(tree)
    return forest
```

In [21]:
```
# Challenge 6
def predict_forest(X, forest):
    '''
    Predict the labels for X
    for all 30 trees
    calculate the average of 30 trees
    return avg. predictions

    Arguments
        X: np.array() shape (m, n)
        forest: a list of trees

    Returns
        y_pred: a list containing predicted classes
                for each row in X

    '''

    return y_pred
```

In [22]:
```
# Putting all together
# with Train/Test
X_train, y_train = make_circles(n_samples=100, noise=0.1, factor=0.5, random_state=0)
X_test, y_test = make_circles(n_samples=100, noise=0.1, factor=0.5, random_state=42)
forest = fit_forest(X_train, y_train)
```

```
y_pred_train = predict_forest(X_train, forest)
y_pred_test = predict_forest(X_test, forest)

print('Training acc:', accuracy(y_pred_train, y_train))
print('Testing acc:', accuracy(y_pred_test, y_test))
```

```
Training acc: 1.0
Testing acc: 0.96
```

In [ ]: