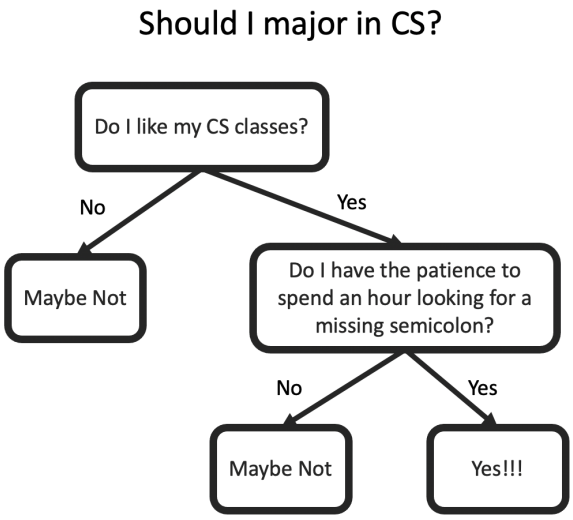
Lab 16

Today, we will continue our journey through the classification task within supervised learning. Today we will explore some of the ingredients of decision trees before learning how to implement them in `sklearn`. Today's goals are:

1. Explain why SVMs and decision trees are different approaches to classification
2. Develop intuition for how to build decision trees
3. Learn how to build and fit decision trees in `sklearn`



Imports for Today

We begin as usual, importing the packages and data that we need.

In [ ]:

```
## Import block
%matplotlib inline

import matplotlib.pyplot as plt
import seaborn as sns; sns.set()

import numpy as np
import pandas as pd

from sklearn.tree import DecisionTreeClassifier

from sklearn.tree import export_graphviz
# from sklearn.externals.six import StringIO
from IPython.display import Image
from pydot import graph_from_dot_data
from six import StringIO
```

Add a conda package!

If you got a (non-deprecation) error, you may need to exit jupyter and add `pydot` to your conda environment.

Helper Functions

We define some premade helper functions that will use to build your decision trees. These functions are heavily adapted from "Decision Tree Algorithm in Python From Scratch" by Eligijus Bujokas on TowardsDataScience.com.

In [ ]:

```python
def gini_index(data_pd: pd.DataFrame, class_var: str) -> float:
    """
    Given the observations of a binary class and the name of the binary class
    calculate the gini index
    """
    # count classes 0 and 1
    count_A = np.sum(data_pd[class_var] == 0)
    count_B = np.sum(data_pd[class_var])

    # get the total observations
    n = count_A + count_B

    # If n is 0 then we return the lowest possible gini impurity
    if n == 0:
        return 0.0

    # Getting the probability to see each of the classes
    p1 = count_A / n
    p2 = count_B / n

    # Calculating gini
    gini = 1 - (p1 ** 2 + p2 ** 2)

    # Returning the gini impurity
    return gini

def info_gain(data_pd: pd.DataFrame, class_var: str, feature: str) -> float:
    """
    Calculates how much info we gain from a split compared to info at the cur
    """
    # compute the base gini impurity (at the current node)
    gini_base = gini_index(data_pd, class_var)

    # split on the feature
    node_left, node_right = split_bool(data_pd, feature)

    # count datapoints in each split and the whole dataset
    n_left = node_left.shape[0]
    n_right = node_left.shape[0]
    n = n_left + n_right

    # get left and right gini index
    gini_left = gini_index(node_left, class_var)
    gini_right = gini_index(node_right, class_var)

    # calculate weight for each node
    # according to proportion of data it contains from parent node
    w_left = n_left / n
    w_right = n_right / n

    # calculated weighted gini index
    w_gini = w_left * gini_left + w_right * gini_right

    # calculate the gain of this split
    gini_gain = gini_base - w_gini

    # return the best feature
    return gini_gain
```

In [ ]:
```
# For function testing
```

Today's data

Today's data is adapted from a dog breed trait dataset posted by Al Chernyshev on Kaggle, as web scraped from breed profiles on https://dogtime.com. Each row represents a breed or breed mix. The dataset contains binary, numerical, and categorical data. For the first part of the lab, we use a subset of the available features to work with only binary variables.

Our goal is to classify dogs as "Good" or "Not Good" for Novice Owners.

In [ ]:
```
## Import Data
dog_pd = pd.read_csv("lab16data-binary.csv", sep = ",", index_col = "Breed Na

print(dog_pd.shape)

dog_pd.head()
```

From SVMs to Decision Trees

In our discussion of SVMs, we did not attempt to interpret nor to explain *why* one data point gets labeled. In SVM, we have a boundary and we might be able to twist ourselves into an "explanation." However, we may want a transparent set of questions that lead us to our classifications.

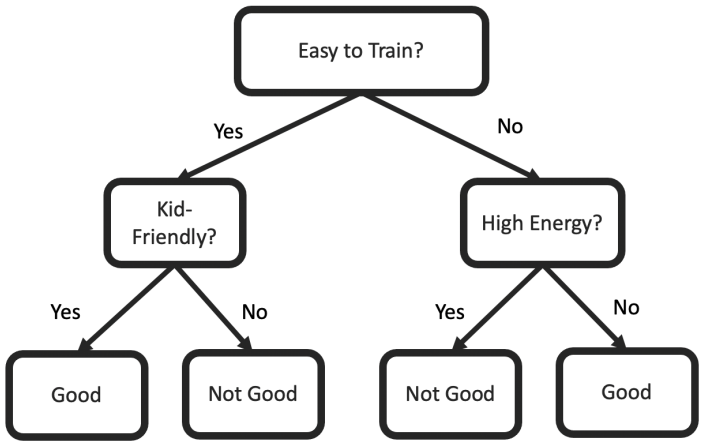
- In SVM, we could say a dog is in group A because it lies above the line `class = 4 + 3*Easy-to-Train - 7*High-Energy` .
- In Decision Trees, we would say that this dog is in group A because it is easy to train and has high energy.

In the SVM example, it can be hard to interpret to the average person "above the line" with some equation. While in comparison, the answering of yes/no questions for the decision tree provides a straight-forward way to understand *why* a dog is or is not in one group. This is the intuition behind **decision trees.**

Decision trees are trees where each **branch** represents a decision based on a specific variable. Consider the following situation: We're a novice dog owner who wants to get a dog. We decide to classify dog breeds as "good" or "not good" for novice owners.

Our goal is to build something like this infographic that tells us how to decide whether or not a dog breed is good for novice owners:

Is this dog breed good for novice owners?



At the top **node,** we represent our first decision. Then based on whether a dog is easy to train, we either examine the dog's energy levels or the dog's kid-friendliness.

Finding the Best Way to Split the Data

The central question of decision trees is: "What decision will give me the best split of my data?" To address this question we need to 1) understand how to split the data, and 2) how to evaluate the "goodness" of a split.

In our example about dogs, currently, we don't know whether a dog's size, energy, or some other characteristic will help us best sort dogs into "good" or "not good" for novice owners.

We start by trying to split the data into two groups based on one variable, then repeat this decision for each other variable in the dataset.

Let's start by splitting our dog breeds based on whether they are "Easy to Train".

In [ ]:
```
# observe that "Easy to Train" is a boolean variable - either True or False
dog_pd['Easy To Train'].head()
```

In [ ]:
```
# split our data into two dataframes: easy/not easy to train
easy_to_train = dog_pd[dog_pd['Easy To Train']]
hard_to_train = dog_pd[~dog_pd['Easy To Train']]
```

In [ ]:
```
# check our output shapes
print(dog_pd.shape)
print(easy_to_train.???)
print(???.???)
```

For our later convenience, let's functionalize the process of splitting a dataframe based on a binary variable:

In [ ]:
```
## split on a boolean/binary variable
## should look very similar to your code in the block above
def split_bool(data_pd, column_name):
    """Returns two pandas dataframes:
    one where the specified variable is true,
    and the other where the specified variable is false"""
    node_left = data_pd[data_pd[???]]
    node_right = ???

    return node_left, node_right

easy_to_train, hard_to_train = split_bool(dog_pd, 'Easy To Train')
```

In [ ]:
```
# For function testing
```

In [ ]:
```
# check our output shapes that use the above function
# Do they match the above?

print(dog_pd.shape)
print(easy_to_train.shape)
print(hard_to_train.shape)
```

As another example, let's repeat our split on the variable "Kid-Friendly" to split dog breeds by whether or not they are good with kids.

In [ ]:
```
kid_friendly, not_kid_friendly = split_bool(???, ???)
```

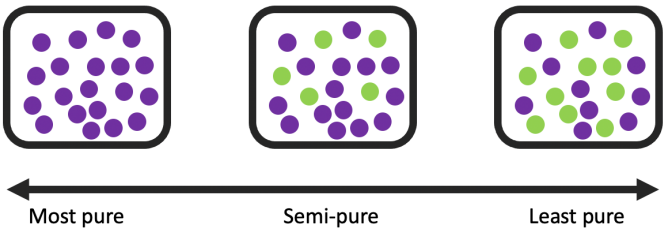
In [ ]:
```
# check our output shapes
```

How do we define the "Best" Split?

We need a way to measure whether "Easy to Train" or "Kid-Friendly" is better for splitting up dogs into groups of "Good" or "Not Good" for Novice Owners.

Note: *In this section, we will use the term "purity" as it is term that is often used in the literature. We—your instructor and pedagogical partner—do not like this term, but feel that you should hear it due to its existence in the decision tree space.*

In an ideal scenario, splitting on one of these variables would let us send all the dogs who are "Good" for Novice Owners to one leaf of the tree, and send all the dogs who are "Not Good" for Novice Owners to the other leaf of the tree. But in our imperfect world, we'll usually end up with *impure leaves*—leaves which are not perfectly separated by our target classification. We aim to quantify the "purity", also known as the "homogeneity", of each node on our tree.



Homogeneity

There are multiple ways to measure the homogeneity of a node, including:

- Gini Impurity Index
- Entropy
- Missclassification

In the helper functions at the top of the lab, we have an implementation of the Gini Impurity Index. For a given leaf's group of data, `gini_leaf()` returns a number between 0 and 0.5, where 0 means the data is completely homogeneous (all the same class) and 0.5 means the data as not homogeneous at all (there's the same amount of data of both classes at the

In [ ]:
```
## find gini impurity index for each of our split groups
## syntax: gini_index(leaf_data, class_var)
easy_to_train_gini = gini_index(easy_to_train, 'Good For Novice Owners')
hard_to_train_gini = gini_index(hard_to_train, ???)

kid_friendly_gini = gini_index(???, ???)
not_kid_friendly_gini = ???(???, ???)
```

In [ ]:
```
# print our gini values
```

Question: Among the "Easy to Train" and "Kid-Friendly" variables, do one decision's nodes seem to have lower Gini Impurities than the other decision?

Your Thoughts Here

Using our chosen metric of homogeneity, **information gain** (coded here as `info_gain()`) quantifies how much we learn from a given decision. (We learn more if we can split the data into more homogenous nodes.) We want to learn more, and thus have higher values for information gain. If you want to read more about the calculations behind the Gini Index and Information Gain, see the resources at the end of the lab.

For now, we'll run with applying `info_gain()` to our decisions, and we'll split on the variable that lets us gain the most information.

In [ ]:
```
## apply info_gain to each split
## syntax is: info_gain(data_decide, decision_var, class_var)
info_easy_to_train = info_gain(dog_pd, 'Easy To Train', 'Good For Novice Owne
info_kid_friendly = info_gain(dog_pd, ???, 'Good For Novice Owners')
```

In [ ]:
```
# print our info gain values
```

Question: Among "Easy To Train" and "Kid-Friendly", which variable gives us more information for classifying dog breeds as "Good for Novice Owners"? Does this correspond with the node which had lower Gini Impurities above?

Your Thoughts Here

The variable whose split gives us the most information is the best first split in our decision tree. In other words, splitting on this variable is the best first decision at the top of our decision tree.

Let's build a function which tests all possible features to split on and returns the best one.

A Note on Types in Python:

The functions in this lab specify input and output types in the function declarations. The function declaration
`def best_split(data_pd: pd.DataFrame, class_var: str) -> float:`
is equivalent to
`def best_split(data_pd, class_var):`
but gives us the additional information that `data_pd` is a Pandas DataFrame, `class_var` is a string, and the return from the function is a float.

We do not expect you to code with types in Python for this course, but this lab aims to give you some exposure in case you encounter them in a future context.

In [ ]:

```
def best_split(data_pd: pd.DataFrame, class_var: str, exclude_features: list
    """
    Returns the name of the best feature to split on at this node.
    If the current node contains the most info (all splits lose information),
    EXCLUDE_FEATURES is the list of variables we want to omit from our list o
    """
    # compute the base gini index (at the current node)
    gini_base = gini_index(data_pd, class_var)

    # initialize max_gain and best_feature
    max_gain = 0
    best_feature = None

    # create list of features of data_pd not including class_var
    features = ???

    # This line will be useful later - can skip for now
    # remove features we're excluding
    # (already made decision on this feature)
    features = [f for f in features if f not in exclude_features]

    # test a split on each feature
    for ??? in ???:
        info = info_gain(???, ???, ???)

        # check whether this is the greatest gain we've seen so far
        # and thus the best split we've seen so far
        if ??? > max_gain:
            best_feature = feature
            max_gain = info

    # return the best feature
    return best_feature
```

In [ ]:

```
# block for testing your code
```

In [ ]:

```
best_split(dog_pd, "Good For Novice Owners")
```

In [ ]:

```
# test for the best split if we exclude 'Kid-Friendly'
best_split(dog_pd, "Good For Novice Owners", exclude_features=['Kid-Friendly'
```

A Note on Splitting Non-Binary Data

Very few real data sets contain only binary variables. Whether a variable is categorical, ordinal, or numerical, the presence of more than two possible values of the variable in the data set means that there are multiple ways to split the data set into two groups.

For example:

- For a categorical variable with levels ABC, there are three potential ways to split the

data into two groups: AB|C, AC|B, or BC|A.
- For an ordinal variable with rankings 1-5, there are four potential decision boundaries: var <= 1, 2, 3, or 4 (Note that <= 5 is not a valid boundary, as the condition will be true for all data and thus we divide the data into only one group.)
- For a numerical variable with values 2.3, 2.6, 3.1, 3.7, and 4.1, we can form decision boundaries between each value of the data similar to the ordinal variable above, or based on moving averages of subsets of the data.

Note that the number of possible decisions on a variable grows rapidly as the number of distinct values of the variable increase.

We will not code decisions on non-binary data from scratch, but it's important to remember that when we talk about finding the "best possible decision" at each level of a decision tree,

Question (optional, but recommended):
If a dataset contains n variables, we've established that we have more than n options to consider for the "best possible decision." In the worst case, what is the order of decisions we need to consider?

Your Thoughts Here

Building a Tree of Decisions

After we decide on the best first decision in our tree, we have two nodes of data. We consider each node individually, and repeat the whole process again to see which of the remaining potential decisions is the next best split for that node. Often, the next best decision is different on each side of the tree.

In other words, we take the nodes that result from our first decision, and we recursively build a decision tree from each node.

Question: When we have a recursive process, we need both the recursive step and a base case (aka stopping condition). What are potential base cases for building a decision tree?

Your Thoughts Here

Let's start writing the **pseudocode** for a recursive decision tree algorithm. Our goal at each node is to:

1. (Stop if we reach the base case)
2. Determine which split/decision gives us the most information
3. Make the split on that variable
4. Continue recursively on each of the resulting two nodes

In [ ]:

```
## initial decision tree algorithm PSEUDOCODE
## this doesn't need to run
## just sketch out how you would follow the steps above

# you may need to pass additional parameters to this function call
# depending on your stopping condition
def decision_tree(node_data: pd.DataFrame, class_var: str):
    # 0. stop at the base case

    # 1. determine which decision gives us the most information

    # 2. make the split according to the best decision

    # 3. continue recursively on each of the resulting two nodes

    pass
```

Other Stopping Conditions

Common stopping conditions for a decision tree algorithm include, but are not limited to:

- When the tree has reached *n* levels of depth
- When none of the remaining possible decisions/splits at a node provide more information than the info we already have at the node
- When we run out of potential decisions**
- When our splits result in leaves smaller than size *n*

**Note that for binary data, we run out of potential decisions when we've split on every variable. For non-binary data, we could make multiple splits on a given variable between different levels of the numbers or among different combinations of categories—thus, we have many, many possible decisions to consider.

Question: Why would it be valuable to have shorter trees (Trees with depth <= *n*)?

Your Thoughts Here

(Hint: Think about overfitting...)

Building a Decision Tree

Now, let's turn your decision tree into working code. We'll be implementing a simplified version of the decision tree in this post. For the scope of this lab, we'll just build the structure of the decision tree—we will not add functionality to produce a classification from the tree. See the source post for an example of how to classify observations using your decision tree.

The outline provided is based on a max depth of 2 as a stopping condition. As a challenge, try to add another stopping condition as described in the above section.

As a reminder, here is our ***recursive*** decision tree algorithm:

1. (Stop if we reach the base case)
2. Determine which split/decision gives us the most information
3. Make the split on that variable
4. Continue recursively on each of the resulting two nodes

In [ ]:
```
## build a decision tree from a node of data
def build_decision_tree(node_data: pd.DataFrame, class_var: str, depth: int =
    """Build a decision tree for NODE_DATA with
    CLASS_VAR as the variable that stores the class assignments.
    EXCLUDE_FEATURES is the list of variables we want to omit from our list o
    # 0. stop at the base case
    max_depth = 2
    if depth >= max_depth:
        return

    # 1. determine which decision gives us the most information
    best_feature = ???
    print(f"{'>'*(depth+1)}Splitting {node_data.shape[0]} data points on {bes

    # 2a. if best_feature == None, don't split further
    if best_feature == None:
        print(f"{'>'*(depth+1)}No best next split.")
        return

    # 2b. else, make the split according to the best decision
    else:
        data_left, data_right = ???
        print(f"{'>'*(depth+1)}Produces {data_left.shape[0]} True data points

        # and exclude this feature at future levels of the tree
        exclude_features.append(???)

    # 3. continue recursively on each of the resulting two nodes
    build_decision_tree(data_left, class_var, depth + 1, exclude_features)
    build_decision_tree(???)
    return
```

In [ ]:
```
# block for testing your code
```

In [ ]:
```
build_decision_tree(dog_pd, "Good For Novice Owners")
```

Question: Can you describe in words how the tree splits the data?

Your Thoughts Here

In the previous tree, we were attempting to classify dogs into "good" for novice owners and not. To practice our understanding and interpretation of trees, build another tree using a different variable as the classifications.

In [ ]:
```

```

Question: Describe how this second tree splits the data:

Your thoughts here

Decision Trees in `sklearn`

Now that we have some basic intuition about decision trees, let's build a decision tree classifier using `sklearn`. As with other machine learning tools in `sklearn`, we will first specify our decision tree model, and then fit the decision tree model to some data—similar to other `sklearn` classifiers we used before.

Since the `sklearn` decision tree implementation can form decisions on non-binary features, let's import our full dog breed dataset.

In [ ]:
```
# import full dataset
dog_full = pd.read_csv("lab16data.csv", sep = ",", index_col = "Breed Name")
dog_data = dog_full.to_numpy(dtype = np.float16)

print(dog_full.shape)

## Look at some of the pandas version of the data
dog_full.head()
```

In [ ]:
```
## Look at some of the numpy version of the data
dog_data[:5, :]
```

In [ ]:
```
# Split the data into the input variables and the target classes
in_dog = dog_data[:,:???]
out_class = dog_data[:,???]

# Get the variable names
var_names = list(dog_full.columns)[:???]
print(var_names)
```

In [ ]:
```
# Specify our model
# note: the default for the `criterion` parameter is 'gini' - but 'entropy' i
dt = DecisionTreeClassifier()
```

In [ ]:
```
# Fit our model to the data
dt.fit(in_dog, out_class)
```

Umm... this is not easy to "look" at. Referring to this post, we adjust their code to create a visual output of our decision tree (double-click the image to zoom in):

In [ ]:
```
dot_data = StringIO()

export_graphviz(dt, out_file=dot_data, feature_names=var_names)
(dt_vis, ) = graph_from_dot_data(dot_data.getvalue())

Image(dt_vis.create_png())
```

Question What do you notice about this image? What is encoded? What is left out?

Your Thoughts Here

To wrap up, we'll predict whether the first five dogs in the dataset are good for novice owners. (This is bad practice to predict from our training data—don't use training data like this when you validate your models.)

In [ ]:
```
# Predict for the first five dogs
dt.predict(in_dog[:5,:???])
```

Next time

Next time, we will look at ensemble methods which aggregates several methods together. The specific example that we will explore are **random forests.** Random forests are based on a collection of decision trees.

Final Thoughts

In reality, every dog is unique and may not conform to the description of its breed. While breed descriptions may be useful guidelines for those hoping to adopt a new dog, it is best for each family to meet the individual dog to get a sense of whether the dog will be a good fit.

And remember, regardless of fit for an individual family, every dog is a good dog.

For this lab, you have two choices for how you finish up this lab. To get credit, you **only** need to do one of the following choices:

Choice 1 - Final Questions:

1. Save the image of your `sklearn` decision tree.
2. Choose a dog breed in the dataset.
3. Using your `sklearn` image of your decision tree, decide if the breed is good for novice owners.
4. Compare your result to the result from the classifier. Are they the same?
5. Compare your result and the classifier's result to the original classification in the dataset. Are they correct?
6. (Optional) If you have knowledge of this dog breed, would you agree with the result of the classification? Is this dog good for novice owners?

Choice 2 - Final Questions:

1. Explain at least one scenario in which use of this classifier could be potentially harmful.
2. Name at least one other scenario where forming decisions via a classifier could be potentially harmful and/or unethical. Explain why.
3. What are some considerations you would want to keep in mind when deciding whether use of a classifier is appropriate in a given scenario?

Share your thoughts in a post on the **#lab16_submission** channel on slack.

If your have questions from this lab, post them to **#lab_questions** with the same preamble (i.e. starting with **Lab16**). If you have the same question, please use one of the emoji's to upvote the question. If you would like to answer someone's question, please use the thread function. This will tie your answer to their question.

Resources consulted

1. Data Source - Dogs Breeds on Kaggle
2. Decision Tree Algorithm in Python From Scratch
3. SCIKIT-LEARN : DECISION TREE LEARNING I - ENTROPY, GINI, AND INFORMATION GAIN
4. `sklearn` documentation
5. `sklearn` further tree information
6. Google Developers: Let's Write a Decision Tree Classifier from Scratch - Machine Learning Recipes #8 (YouTube)
7. StatQuest: Decision Trees (YouTube)
8. ISLR
9. SDS 293 Notes by R. Jordan Crouser
10. Decision Tree In Python
11. Decision Tree Algorithm With Hands On Example

This lab was created by Kathleen R. Hablutzel, Smith College '23J, as part of a Pedagogical Partnership with Professor Katherine M. Kinnaird in Spring 2022. Parts of the introduction and conclusion to the lab are borrowed from previous versions of Lab 16. Much of the code has been adapted from this post.