# Lecture 8: Binning, the DTree class

CS 167: Machine Learning

## Histograms with matplotlib

First, open german_credit.csv in a spreadsheet and familiarize yourself.

```
import matplotlib.pyplot as plt
%matplotlib inline
credit_data = pandas.read_csv('german_credit.csv')
print(credit_data['Duration in month'])
plt.hist(credit_data['Duration in month'], 12)
plt.xlabel('Duration in month')
plt.ylabel('frequency')
plt.show()
```

Try values other than 12 to see it with a different number of **bins**.

Try replacing the 12 with a list like [0,12,24,36,72] to see what it looks like with varied bin sizes



## Binning with cut

Pandas has a function that can be used to chop something up into bins

cut

Similar to hist, there are options for creating the bins.

```
#to make 4 bins
credit_data['Duration in month'] = \
    pandas.cut(credit_data['Duration in month'],4)
print(credit_data['Duration in month'])
```

```
#to make custom bins
credit_data['Duration in month'] = \
    pandas.cut(credit_data['Duration in month'],[0,12,24,36,72])
print(credit_data['Duration in month'])
```



## Decision Tree Code: node constructor

```
class DNode:
    def __init__(self,predictor_columns_data,target_column_data):

        self.__attribute = ''
        self.__predictor_columns = predictor_columns_data
        self.__target_column = target_column_data
        self.__child_nodes = {}
        self.__most_common_value_here = ''
```

## Sorting examples down the tree during training

```
class DNode:
    #...
    def train(self):
        self.choose_attribute() #'best' attribute at this node
        self.__most_common_value_here = self.__target_column.value_counts().idxmax()

        attribute_values_here = self.__predictor_columns[self.__attribute].unique()

        for value in attribute_values_here:

            examples_for_child_predictor_cols = self.__predictor_columns[self.__predictor_columns[self.__attribute] ==
            examples_for_child_target_col = self.__target_column[self.__predictor_columns[self.__attribute] == value]

            if examples_for_child_target_col.empty:
                print("error: we shouldn't get here")

            elif len(examples_for_child_predictor_cols.columns.values) == 1:
                leaf_child = DLeaf( self.__most_common_value_here )
                self.__child_nodes[value] = leaf_child

            elif len(examples_for_child_target_col.unique()) == 1:
                leaf_child = DLeaf( examples_for_child_target_col.unique()[0] )
                self.__child_nodes[value] = leaf_child

            else: #we have a regular decision node for this attribute value
                examples_for_child_predictor_cols = examples_for_child_predictor_cols.drop(self.__attribute,1)
                new_child = DNode(examples_for_child_predictor_cols,examples_for_child_target_col)
                new_child.train()
                self.__child_nodes[value] = new_child
```



## Selecting the best attribute

```
class DNode:
    #...
    def choose_attribute(self):
        #what a terrible way to choose the attribute!
        self.__attribute = random.choice\
            (self.__predictor_columns.columns.values)
```

You will fill this in by picking the best attribute based on information gain.



## Making predictions

```
class DNode:
    #...
    def predict(self,new_example):
        #look up the right branch in our dictionary of children
        if new_example[self.__attribute] in self.__child_nodes:
            node_on_corresponding_branch = self.__child_nodes[\
                new_example[self.__attribute]]
            return node_on_corresponding_branch.predict(new_example)
        else:
            return self.__most_common_value_here
```



## DLeaf: for representing leaves

```
class DLeaf:

    def __init__(self,val_in_target_col):
        self.__target_value = val_in_target_col


    def predict(self,new_example):
        return self.__target_value
```

## DTree: the wrapper

```
class DTree:
    def fit(self,predictor_columns_data,target_column_data):
        self.__root_node = DNode(predictor_columns_data,target_column_data)
        self.__root_node.train()

    def predict(self,df_of_new_examples):
        #apply the predict function to the whole series, one at a time, this returns the
        predictions = df_of_new_examples.apply(self.__root_node.predict,axis=1)
        return predictions

    def print_tree(self):
        self.__root_node.print_node()
```



## Try it out

```
credit_data = pandas.read_csv('german_credit.csv')

train_data, test_data = \
    cross_validation.train_test_split(credit_data,test_size = 0.1)

attributes_to_use = ['Status of existing checking account',\
    'Credit history','Purpose']

my_tree = DTree()

my_tree.fit(train_data[attributes_to_use],train_data['Creditability'])

my_tree.print_tree()

predictions = my_tree.predict(test_data[attributes_to_use])

print(accuracy(test_data['Creditability'],predictions))
```

**Exercise:** *What is its accuracy on the training set?*



## Things to do

**Mandatory:**

- Write functions for computing the computing the best attribute, you will need to compute
  - entropy for a set of examples: use the formula - based on the target column values
  - expected entropy for a set of examples if we split on a particular attribute
- Use all of the categorical attributes

**Do at least one:**

- Bin up the numerical columns, and use them
- Change the best_attribute code so that it can find a good set of bins based on entropy in the middle of training
- Implement some kind of early stopping that improves performance
- Implement some kind of pruning the improves performance



## How to get started

First, work on the function for finding `entropy` from some set of examples

- pass the set of examples to your `entropy` function
- **Note:** you only need to look at the target column to compute this!

Next, work on finding the `expected_entropy` from a particular column given a set of examples

- pass the set of examples to your `expected_entropy` function
- pass the column name (or number or the column data itself) of the attribute you want to split on to your `expected_entropy` function
- you will need to find subsets of the examples for each possible value the attribute can have
  - compute the entropy of each of these
  - take their weighted average based on how many examples were in each subset - this is the expected entropy

Then, work on integrating it into the `DNode` class