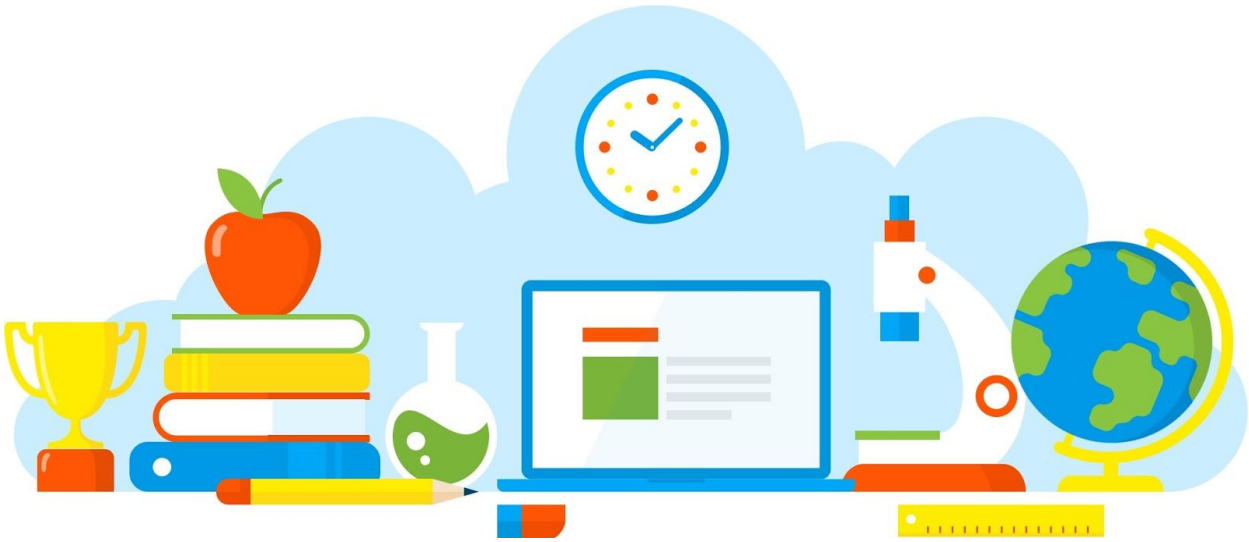


Machine Learning HW3 Report

Teacher: Dr. Zare

Mahdi Sadeghi
830598035
1398/9/9

Loading And Cleaning Data

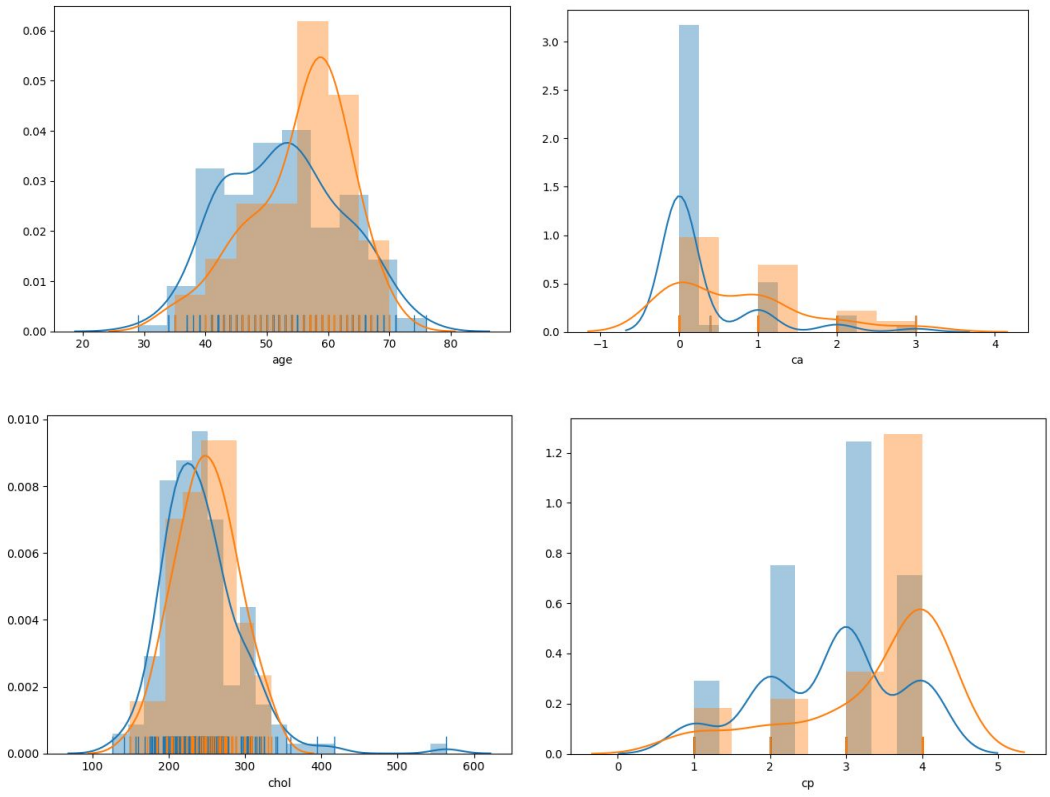
At first we should load the dataset and clean the data. The code for this purpose is located at dataset.py file. I used pandas read_csv function for loading the data and then cleaned the data by first removing rows with invalid “num” column (“num” other than 0 and 1) and replacing NaN values with their respective column mean.

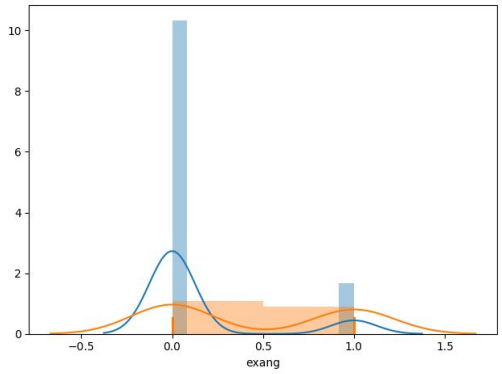
```
def _load_dataset_from_file(self):
   self.main_dataset = pd.read_csv(self.dataset_file_location,
header=None, names=[
       'age', 'sex', 'cp', 'trestbps', 'chol', 'fbs', 'restecg',
'thalach', 'exang', 'oldpeak', 'slope', 'ca',
       'thai', 'num'], index_col=False)

def _clean(self):
   self.main_dataset = self.main_dataset[self.main_dataset['num'] <
2]
   self.main_dataset.fillna(self.main_dataset.mean(), inplace=True)
```

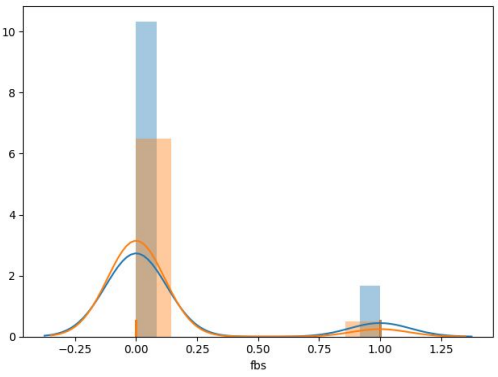
Drawing Data Distributions

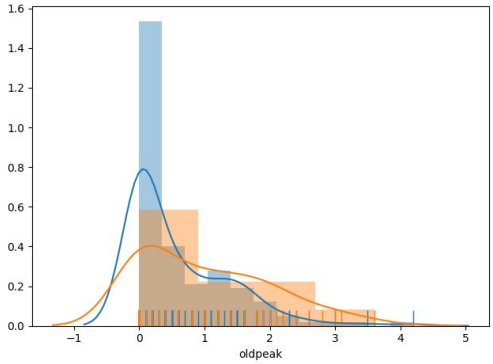
The code used to draw distributions is as follows:

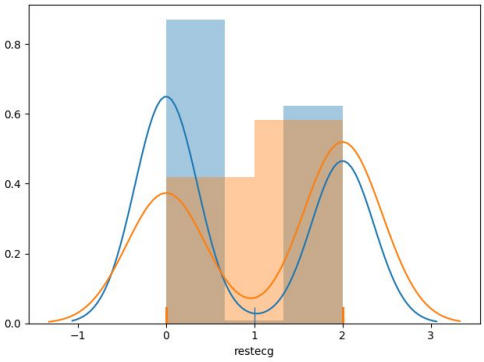
```
def draw_distribution(self, column):
    t0_dataset = self.main_dataset.loc[self.main_dataset['num'] ==
0].loc[:, column]
    t1_dataset = self.main_dataset.loc[self.main_dataset['num'] ==
1].loc[:, column]
    print(t0_dataset)
    sns.distplot(a=t0_dataset, hist=True, rug=True, label='0')
    sns.distplot(a=t1_dataset, hist=True, rug=True, label='1')
    plt.show()
```

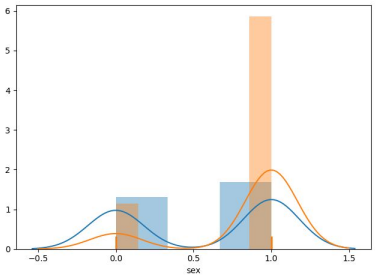


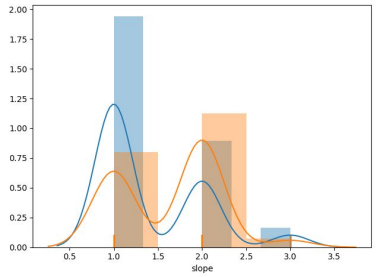

exang

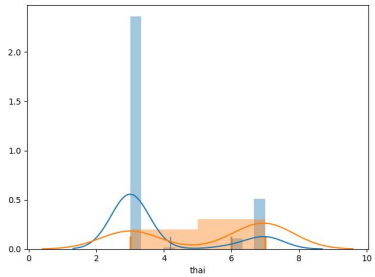

fbs

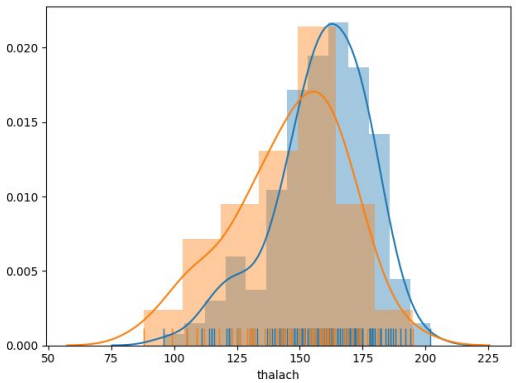

oldpeak

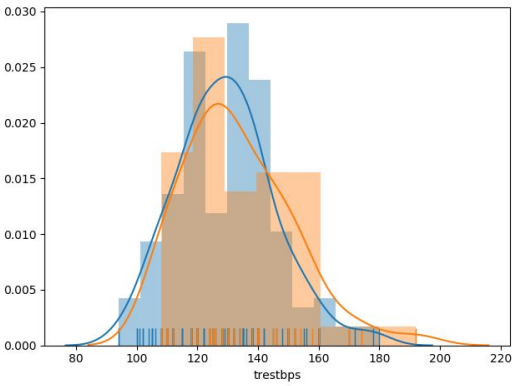

restecg


sex


slope


thai


thalach


trestbps

Generating Test, Training and KFold Validation Data

```
def _generate_training_and_test(self):
   mask = np.random.rand(len(self.main_dataset)) < 0.8
   self.training_dataset = self.main_dataset[mask]
   self.test_dataset = self.main_dataset[~mask]
   self.validation_folds = KFold(n_splits=5, shuffle=True)

def get_main_dataset(self):
   return self.main_dataset

def get_training_dataset(self):
   return self.training_dataset

def get_test_dataset(self):
   return self.test_dataset

def get_cross_validation_indexes(self):
   return self.validation_folds.split(self.training_dataset)
```

Creating the Decision Tree

For generating the decision tree we must first find a way to introduce our attributes to the system. There is an abstract class called “Attribute” in attribute.py which we can implement to describe each of our dataset features separately. To make things easier there is an AttributeFactory class that can help us create Attribute Implementations without implementing it ourselves. The abstract factory supports discrete with custom mapping function and continuous attributes for now. For custom attributes, one must derive and implement Attribute class herself.

To convert continuous feature to list of separate buckets, AttributeFactory uses Fayyad-Irani method. First we sort the data by the feature we want to discretize, then we add a stop point for each change of the target attribute:

```
class Attribute(ABC):
    def __init__(self, name, all_classes):
        self.name = name
        self.all_classes = all_classes

    def get_all_classes(self):
        return self.all_classes

    @abstractmethod
    def get_class(self, sample):
        """
        get class of the input sample for the attribute
        :param sample:
        :return: class (int)
        """
        pass

    def filter(self, samples, cls):
        mask = [(self.get_class(x) == cls) for i, x in samples.iterrows()]
        return samples.loc[mask, :]


class AttributeFactory:
    @staticmethod
    def create_discrete(name, all_classes, get_class_fn):
        class DiscreteAttributeImpl(Attribute):
```

```
        def __init__(self):
            super().__init__(name, all_classes)

        def get_class(self, sample):
            return get_class_fn(sample)

    return DiscreteAttributeImpl()

@staticmethod
def create_continuous(attrname, samples, source_column, target_column):
    class ContinuousAttributeImpl(Attribute):
        def __init__(self, threshold):
            self.threshold = threshold
            name = '{0} < {1}'.format(attrname, threshold)
            super().__init__(name, ['Yes', 'No'])

        def get_class(self, sample):
            if sample[source_column] < self.threshold:
                return 'Yes'

            return 'No'

    sorted_samples = samples.sort_values(source_column)
    attributes = []

    last_target = sorted_samples.iloc[0][target_column]
    last_source = sorted_samples.iloc[0][source_column]

    last_min = -math.inf

    for _, sample in sorted_samples.iterrows():
        if sample[target_column] != last_target:
            threshold = (sample[source_column] + last_source) / 2
            if threshold > last_min:
                attribute = ContinuousAttributeImpl(threshold)
                attributes.append(attribute)
                last_min = threshold

        last_target = sample[target_column]
        last_source = sample[source_column]

    return attributes
```

After describing our attributes, we may start creating the decision tree itself. In the DecisionTreeFactory class, there is a static method called “create” which takes samples, attributes and an attribute selector to create the decision tree. The algorithm is the same of the slides, only with a subtle difference, the factory method itself does not select the best attribute, instead it passes required data to attribute selector which in turn returns the best feature to use at that moment. This is used to allow multiple attribute selection algorithms (GiniIndex and Information Gain) with the same code base.

```
class DecisionTreeFactory:
   @staticmethod
   def _create_node(samples, attributes, target_attribute, attribute_selector):
       if len(samples) == 0:
           return DTreeTerminalNode('X')

       tattr = samples.loc[:, target_attribute.name].mode().iloc[0]

       # if there is no more attributes -> return terminal with most abundant
target value
       if len(attributes) == 0:
           return DTreeTerminalNode(tattr)

       # if all the samples have same target_attribute -> return terminal node
with target value
       if samples.loc[:, target_attribute.name].nunique() == 1:
           return DTreeTerminalNode(tattr)

       # select best attribute
       selected_attribute = attribute_selector.select(samples, attributes,
target_attribute)
       attributes.remove(selected_attribute)

       # create sub nodes
       links = []
       for attribute_class in selected_attribute.get_all_classes():
           child_node =
DecisionTreeFactory._create_node(selected_attribute.filter(samples,
attribute_class),
                                                            attributes[:],
target_attribute, attribute_selector)
           link = DTreeLink(attribute_class, child_node)
```

```
            links.append(link)

        return DTreeDecisionNode(selected_attribute, links, tattr)

    @staticmethod
    def create(samples, attributes, target_attribute, attribute_selector):
        root_node = DecisionTreeFactory._create_node(samples, attributes,
target_attribute, attribute_selector)
        return DecisionTree(root_node, target_attribute)
```

By passing different implementations of attribute selector, you can achieve different trees. Below is the two implementations of AttributeSelector provided by this project:

```
class AttributeSelector(ABC):
    @abstractmethod
    def select(self, samples, attributes, target_attribute):
        pass


class InformationGainAttributeSelector(AttributeSelector):
    def _compute_entropy(self, samples, target_attribute):
        entropy = 0

        if len(samples) == 0:
            return entropy

        for cls in target_attribute.get_all_classes():
            p = len(target_attribute.filter(samples, cls)) / len(samples)
            if p != 0:
                entropy += -1 * p * math.log(p, 2)

        return entropy

    def select(self, samples, attributes, target_attribute):
        current_entropy = self._compute_entropy(samples, target_attribute)
        max_gain = (None, -math.inf)
        for attribute in attributes:
            total_entropy = 0
            for cls in attribute.get_all_classes():
                filtered_samples = attribute.filter(samples, cls)
```

```
                factor = len(filtered_samples) / len(samples)
                total_entropy += factor *
self._compute_entropy(filtered_samples, target_attribute)

            total_entropy = current_entropy - total_entropy
            if total_entropy > max_gain[1]:
                max_gain = (attribute, total_entropy)

        return max_gain[0]


class GiniIndexAttributeSelector(AttributeSelector):
    def _compute_gindex(self, samples, target_attribute):
        gindex = 0

        if len(samples) == 0:
            return gindex

        for cls in target_attribute.get_all_classes():
            p = len(target_attribute.filter(samples, cls)) / len(samples)
            gindex += p ** 2

        return 1 - gindex

    def select(self, samples, attributes, target_attribute):
        current_gindex = self._compute_gindex(samples, target_attribute)
        max_gain = (None, -math.inf)
        for attribute in attributes:
            total_gindex = 0
            for cls in attribute.get_all_classes():
                filtered_samples = attribute.filter(samples, cls)
                factor = len(filtered_samples) / len(samples)
                total_gindex += factor * self._compute_gindex(filtered_samples,
target_attribute)

            total_gindex = current_gindex - total_gindex
            if total_gindex > max_gain[1]:
                max_gain = (attribute, total_gindex)

        return max_gain[0]
```

Pruning

After the decision tree was created, we can prune it by removing each decision node by replacing it with the dominant target attribute of samples in that point. After that, we calculate the error rate of the tree. If the error was increased we bring the node back.

```
def prune(self, validation_samples):
        all_nodes = self._get_all_nodes(0, self.root_node, True, False, False)
        all_nodes = sorted(all_nodes, key=lambda x: x[0], reverse=True)
        error_rate = self.evaluate(validation_samples)

        for _, node in all_nodes:
            node.deactivate()
            pruned_error_rate = self.evaluate(validation_samples)

            if pruned_error_rate > error_rate:
                node.activate()
            else:
                error_rate = pruned_error_rate
```

Cross Validation

For generating the best tree, we divide the training dataset to 5 folds. After that we can generate 5 separate trees and compute their error rates with the single validation folds. The tree with best performance shall be selected as the final tree.

```
min_error_tree = (math.inf, None)
for training_indices, validation_indices in
dataset.get_cross_validation_indexes():
   dtree =
dt.DecisionTreeFactory.create(dataset.get_training_dataset().iloc[tra
ining_indices, :],
                                          attributes_list,
                                          attribute_num,
attribute_selector)

dtree.prune(dataset.get_training_dataset().iloc[validation_indices,
:])
   error_rate =
dtree.evaluate(dataset.get_training_dataset().iloc[validation_indices
, :])
   if error_rate <= min_error_tree[0]:
       min_error_tree = (error_rate, dtree)
```

Results

The trees can be found as pdf files in “docs” folder of the project.

Gini Index Tree Error Rate: 0.32

Information Gain Tree Error Rate: 0.23