classification-toy-example-copy

March 9, 2024

1 Lab 3: Introducing Classification

Objectives: - To gain hands-on experience classifying small dataset - To implement concepts related to Decision Tree classifier (i.e. Entropy, Information Gain), along with the Decision Tree algorithm

[ ]:
```
import math
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
# Read the data
df = pd.read_csv('toy_data.csv')
df
```

[ ]:
```
print(df.info())
```

[ ]:
```
df_num = df.copy()
d = {'<=30':0, '31-40':1, '>40':2}
df_num['age'] = df_num['age'].map(d)
d = {'high':2,'medium':1,'low':0}
df_num['income'] = df_num['income'].map(d)
d = {'yes':1,'no':0}
df_num['student'] = df_num['student'].map(d)
df_num['buys computer'] = df_num['buys computer'].map(d)
d = {'excellent':1,'fair':0}
df_num['credit rating'] = df_num['credit rating'].map(d)
df_num
```

2 Calculate Gain

2.0.1 entropy(t)

[ ]:
```
entropy_t = 0
n=df_num.shape[0]
countt = df_num['buys computer'].value_counts()
entropy_t = -countt[1]*math.log2(countt[1]/n)/n - countt[0]*math.log2(countt[0]/
 ↪n)/n
print(entropy_t)
```

2.0.2 entropy(i)

[ ]:
```
def entropy (df, target_att):
    ans = 0
    # print(f"n = {n}")
    for data in df[target_att].unique():
        print("data",data)
        n_i = df[(df[target_att] == data)].shape[0]
        # print("size",n_i)
        p_jyes = df[(df[target_att] == data) & (df['buys computer'] == 1)].
↪shape[0]/n_i
        # print(f"p(j=yes|{data})={p_jyes}")
        p_jno = df[(df[target_att] == data) & (df['buys computer'] == 0)].
↪shape[0]/n_i
        # print(f"p(j=no|{data})={p_jno}")
        if (p_jyes == 0 or p_jno == 0):
            log_j = 0
        else:
            log_j = -(p_jyes * math.log2(p_jyes) + p_jno * math.log2(p_jno))
        # print(f"log(j|{data})={log_j}")
        ans += (n_i*log_j/n)
        # print("\n")
    return ans
```

2.0.3 Gain fn

[ ]:
```
def gain_fn (target_att):
    return entropy_t - entropy(df_num,target_att)
```

3 Calculation

[ ]:
```
gain_fn("age")
```

[ ]:
```
gain_fn("income")
```

[ ]:
```
gain_fn("student")
```

[ ]:
```
gain_fn("credit rating")
```

4 Decision Tree

make it all numerical

[ ]:
```
features = ['age','income','student','credit rating']
X = df_num[features]
y = df_num['buys computer']
```

```
print(X)
print(y)
```

[ ]:
```
print(pd.concat([X,y],axis=1))
```

[ ]:
```
class Node():
    """
    A class representing a node in a decision tree.
    """

    def __init__(self, feature=None, threshold=None, left=None, right=None,␣
↪gain=None, value=None):
        """
        Initializes a new instance of the Node class.

        Args:
            feature: The feature used for splitting at this node. Defaults to␣
↪None.
            threshold: The threshold used for splitting at this node. Defaults␣
↪to None.
            left: The left child node. Defaults to None.
            right: The right child node. Defaults to None.
            gain: The gain of the split. Defaults to None.
            value: If this node is a leaf node, this attribute represents the␣
↪predicted value
                for the target variable. Defaults to None.
        """
        self.feature = feature
        self.threshold = threshold
        self.left = left
        self.right = right
        self.gain = gain
        self.value = value
```

[70]:
```
class DecisionTree():
    """
    A decision tree classifier for binary classification problems.
    """

    def __init__(self, min_samples=2, max_depth=2):
        """
        Constructor for DecisionTree class.

        Parameters:
            min_samples (int): Minimum number of samples required to split an␣
↪internal node.
```

```
            max_depth (int): Maximum depth of the decision tree.
        """
        self.min_samples = min_samples
        self.max_depth = max_depth

    def split_data(self, dataset, feature, threshold):
        """
        Splits the given dataset into two datasets based on the given feature␣
↪and threshold.

        Parameters:
            dataset (ndarray): Input dataset.
            feature (int): Index of the feature to be split on.
            threshold (float): Threshold value to split the feature on.

        Returns:
            left_dataset (ndarray): Subset of the dataset with values less than␣
↪or equal to the threshold.
            right_dataset (ndarray): Subset of the dataset with values greater␣
↪than the threshold.
        """
        # Create empty arrays to store the left and right datasets
        left_dataset = []
        right_dataset = []

        # Loop over each row in the dataset and split based on the given␣
↪feature and threshold
        for row in dataset:
            if row[feature] <= threshold:
                left_dataset.append(row)
            else:
                right_dataset.append(row)

        # Convert the left and right datasets to numpy arrays and return
        left_dataset = np.array(left_dataset)
        right_dataset = np.array(right_dataset)
        return left_dataset, right_dataset

    def entropy(self, y):
        """
        Computes the entropy of the given label values.

        Parameters:
            y (ndarray): Input label values.

        Returns:
            entropy (float): Entropy of the given label values.
```

```
        """
        entropy = 0

        # Find the unique label values in y and loop over each value
        # THIS IS FUCKING CLEVERRRR!!!
        # AND IT'S THE SAME AS PROF SAID
        labels = np.unique(y)
        for label in labels:
            # Find the examples in y that have the current label
            label_examples = y[y == label]
            # Calculate the ratio of the current label in y
            pl = len(label_examples) / len(y)
            # Calculate the entropy using the current label and ratio
            entropy += -pl * np.log2(pl)

        # Return the final entropy value
        return entropy

    def information_gain(self, parent, left, right):
        """
        Computes the information gain from splitting the parent dataset into␣
↪two datasets.

        Parameters:
            parent (ndarray): Input parent dataset.
            left (ndarray): Subset of the parent dataset after split on a␣
↪feature.
            right (ndarray): Subset of the parent dataset after split on a␣
↪feature.

        Returns:
            information_gain (float): Information gain of the split.
        """
        # set initial information gain to 0
        information_gain = 0
        # compute entropy for parent
        parent_entropy = self.entropy(parent)
        # calculate weight for left and right nodes
        weight_left = len(left) / len(parent)
        weight_right= len(right) / len(parent)
        # compute entropy for left and right nodes
        entropy_left, entropy_right = self.entropy(left), self.entropy(right)
        # calculate weighted entropy
        weighted_entropy = weight_left * entropy_left + weight_right *␣
↪entropy_right
        # calculate information gain
        information_gain = parent_entropy - weighted_entropy
```

```
        return information_gain


    def best_split(self, dataset, num_samples, num_features):
        """
        Finds the best split for the given dataset.

        Args:
        dataset (ndarray): The dataset to split.
        num_samples (int): The number of samples in the dataset.
        num_features (int): The number of features in the dataset.

        Returns:
        dict: A dictionary with the best split feature index, threshold, gain,
              left and right datasets.
        """
        # dictionary to store the best split values
        best_split = {'gain':- 1, 'feature': None, 'threshold': None}
        # loop over all the features
        for feature_index in range(num_features):
            #get the feature at the current feature_index
            feature_values = dataset[:, feature_index]
            #get unique values of that feature
            thresholds = np.unique(feature_values)
            # loop over all values of the feature
            for threshold in thresholds:
                # get left and right datasets
                left_dataset, right_dataset = self.split_data(dataset,␣
↪feature_index, threshold)
                # check if either datasets is empty
                if len(left_dataset) and len(right_dataset):
                    # get y values of the parent and left, right nodes
                    y, left_y, right_y = dataset[:, -1], left_dataset[:, -1],␣
↪right_dataset[:, -1]
                    # compute information gain based on the y values
                    information_gain = self.information_gain(y, left_y, right_y)
                    # update the best split if conditions are met
                    if information_gain > best_split["gain"]:
                        best_split["feature"] = feature_index
                        best_split["threshold"] = threshold
                        best_split["left_dataset"] = left_dataset
                        best_split["right_dataset"] = right_dataset
                        best_split["gain"] = information_gain
        return best_split


    def calculate_leaf_value(self, y):
```

```
        """
        Calculates the most occurring value in the given list of y values.

        Args:
            y (list): The list of y values.

        Returns:
            The most occurring value in the list.
        """
        y = list(y)
        #get the highest present class in the array
        most_occuring_value = max(y, key=y.count)
        return most_occuring_value

    def build_tree(self, dataset, current_depth=0):
        """
        Recursively builds a decision tree from the given dataset.

        Args:
        dataset (ndarray): The dataset to build the tree from.
        current_depth (int): The current depth of the tree.

        Returns:
        Node: The root node of the built decision tree.
        """
        # split the dataset into X, y values
        X, y = dataset[:, :-1], dataset[:, -1]
        n_samples, n_features = X.shape
        # keeps spliting until stopping conditions are met
        if n_samples >= self.min_samples and current_depth <= self.max_depth:
            # Get the best split
            best_split = self.best_split(dataset, n_samples, n_features)
            # Check if gain isn't zero
            if best_split["gain"]:
                # continue splitting the left and the right child. Increment␣
↪current depth
                left_node = self.build_tree(best_split["left_dataset"],␣
↪current_depth + 1)
                right_node = self.build_tree(best_split["right_dataset"],␣
↪current_depth + 1)
                # return decision node
                return Node(best_split["feature"], best_split["threshold"],
                            left_node, right_node, best_split["gain"])

        # compute leaf node value
        leaf_value = self.calculate_leaf_value(y)
        # return leaf node value
```

```
        return Node(value=leaf_value)

    def fit(self, X, y):
        """
        Builds and fits the decision tree to the given X and y values.

        Args:
        X (ndarray): The feature matrix.
        y (ndarray): The target values.
        """
        # print(X.shape, y.shape)
        # dataset = np.concatenate((X, y), axis=1)
        dataset = pd.concat([X,y],axis=1)
        dataset = dataset.to_numpy()
        self.root = self.build_tree(dataset)

    def predict(self, X):
        """
        Predicts the class labels for each instance in the feature matrix X.

        Args:
        X (ndarray): The feature matrix to make predictions for.

        Returns:
        list: A list of predicted class labels.
        """
        # Create an empty list to store the predictions
        predictions = []
        X= X.to_numpy()
        # For each instance in X, make a prediction by traversing the tree
        for x in X:
            prediction = self.make_prediction(x, self.root)
            # Append the prediction to the list of predictions
            predictions.append(prediction)
        # Convert the list to a numpy array and return it
        predictions = np.array(predictions)
        return predictions

    def make_prediction(self, x, node):
        """
        Traverses the decision tree to predict the target value for the given␣
↪feature vector.

        Args:
        x (ndarray): The feature vector to predict the target value for.
        node (Node): The current node being evaluated.
```

```
        Returns:
        The predicted target value for the given feature vector.
        """
        # if the node has value i.e it's a leaf node extract it's value
        if node.value != None:
            return node.value
        else:
            #if it's node a leaf node we'll get it's feature and traverse␣
↪through the tree accordingly
            feature = x[node.feature]
            if feature <= node.threshold:
                return self.make_prediction(x, node.left)
            else:
                return self.make_prediction(x, node.right)
```

[71]:

```
def train_test_split(X, y, test_size=0.2):
  """
  Splits data into training and testing sets without shuffling.

  Args:
      X (ndarray): The feature matrix.
      y (ndarray): The target values.
      test_size (float, optional): The proportion of data to be used for the␣
↪test set. Defaults to 0.2.

  Returns:
      tuple: A tuple containing the training and testing data (X_train, X_test,␣
↪y_train, y_test).
  """

  if not isinstance(test_size, (float, int)):
    raise ValueError("test_size should be a float or an integer")
  elif test_size < 0 or test_size > 1:
    raise ValueError("test_size should be between 0.0 and 1.0")

  n_samples = X.shape[0]
  test_index = int(n_samples * test_size)

  X_train, X_test = X[:test_index], X[test_index:]
  y_train, y_test = y[:test_index], y[test_index:]

  return X_train, X_test, y_train, y_test
```

[72]:

```
def accuracy(y_true, y_pred):
    """
    Computes the accuracy of a classification model.
```

```
    Parameters:
    ----------
        y_true (numpy array): A numpy array of true labels for each data point.
        y_pred (numpy array): A numpy array of predicted labels for each data␣
↪point.

    Returns:
    ----------
        float: The accuracy of the model
    """
    y_true = y_true.flatten()
    total_samples = len(y_true)
    correct_predictions = np.sum(y_true == y_pred)
    return (correct_predictions / total_samples)
```

[73]:
```
def balanced_accuracy(y_true, y_pred):
    """Calculate the balanced accuracy for a multi-class classification problem.

    Parameters
    ----------
        y_true (numpy array): A numpy array of true labels for each data point.
        y_pred (numpy array): A numpy array of predicted labels for each data␣
↪point.

    Returns
    -------
        balanced_acc : The balanced accuracyof the model

    """
    y_pred = np.array(y_pred)
    y_true = y_true.flatten()
    # Get the number of classes
    n_classes = len(np.unique(y_true))

    # Initialize an array to store the sensitivity and specificity for each␣
↪class
    sen = []
    spec = []
    # Loop over each class
    for i in range(n_classes):
        # Create a mask for the true and predicted values for class i
        mask_true = y_true == i
        mask_pred = y_pred == i

        # Calculate the true positive, true negative, false positive, and false␣
↪negative values
        TP = np.sum(mask_true & mask_pred)
```

```
        TN = np.sum((mask_true != True) & (mask_pred != True))
        FP = np.sum((mask_true != True) & mask_pred)
        FN = np.sum(mask_true & (mask_pred != True))

        # Calculate the sensitivity (true positive rate) and specificity (true␣
↪negative rate)
        sensitivity = TP / (TP + FN)
        specificity = TN / (TN + FP)

        # Store the sensitivity and specificity for class i
        sen.append(sensitivity)
        spec.append(specificity)
    # Calculate the balanced accuracy as the average of the sensitivity and␣
↪specificity for each class
    average_sen =  np.mean(sen)
    average_spec =  np.mean(spec)
    balanced_acc = (average_sen + average_spec) / n_classes

    return balanced_acc
```

[74]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y,test_size=0.8)
print(X_train)
# print(X_test)
print(y_train)
# print(y_test)
```

```
    age  income  student  credit rating
0     0       2        0              0
1     0       2        0              1
2     1       2        0              0
3     2       1        0              0
4     2       0        1              0
5     2       0        1              1
6     1       0        1              1
7     0       1        0              0
8     0       0        1              0
9     2       1        1              0
10    0       1        1              1
0     0
1     0
2     1
3     1
4     1
5     0
6     1
7     0
8     1
```

```
9     1
10    1
Name: buys computer, dtype: int64
```

[78]:
```
#create model instance
model = DecisionTree(2,10)

# Fit the decision tree model to the training data.
model.fit(X_train, y_train)

# Use the trained model to make predictions on the test data.
predictions = model.predict(X_test)

# Calculate evaluating metrics
print(f"Model's Accuracy: {accuracy(y_test.to_numpy(), predictions)}")
print(f"Model's Balanced Accuracy: {balanced_accuracy(y_test.to_numpy(),␣
↪predictions)}")
```

```
Model's Accuracy: 1.0
Model's Balanced Accuracy: 1.0
```

[ ]: