



# Artificial intelligence

First project

Farnaz khoshdoost azad

99521253

```
▶ class Node:
    def __init__(self, feature = None, threshold = None, data_left = None , data_right = None,
                  gain = None, value = None, gini = None, entropy = None):
        self.feature = feature
        self.threshold = threshold
        self.data_left = data_left
        self.data_right = data_right
        self.gain = gain
        self.value = value
        self.gini = gini
        self.entropy = entropy
```


- ▶ this code defines a Node class with several attributes representing different properties of a node. The constructor initializes these attributes based on the provided arguments or assigns them default values of None.

```
▶ class DecisionTree:  
    def __init__(self, min_samples_split=25, max_depth=100):  
        self.min_samples_split = min_samples_split  
        self.max_depth = max_depth  
        self.root = None
```

- ▶ After that I implement decision tree class from scratch. This code defines the DecisionTree class. The constructor method `__init__` is called when a new object of the class is created. It takes two parameters: `min_samples_split` and `max_depth`, which have default values of 25 and 100, respectively. These parameters determine the minimum number of samples required to split a node (`min_samples_split`) and the maximum depth of the decision tree (`max_depth`). The constructor also initializes the root attribute as `None`.

```
@staticmethod
def Entropy(s):
    counts = np.bincount(np.array(s, dtype=np.int64))
    percentages = counts / len(s)
    ent = 0
    for pct in percentages:
        if pct > 0:
            ent -= pct * np.log2(pct)
    return ent
```

- Entropy is a measure of impurity in a set. The method uses numpy (np) to calculate the count of each unique value in s and then calculates the percentage of each value. It iterates over the percentages and calculates the entropy using the entropy formula. The calculated entropy is returned.




Best\_split method calculates the information a based on the parent set and its two child sets (left\_child and right\_child). Information gain measures the reduction in entropy or impurity achieved by splitting the data. The method calculates the proportion of samples in the left and right child sets (num\_left and num\_right) and uses the Entropy method to calculate the entropy of the parent, left child, and right child sets. It then subtracts the weighted average of the entropies of the child sets from the entropy of the parent set to obtain the information gain.

```
def _gini_impurity(self, y):  
    _, counts = np.unique(y, return_counts=True)  
    probabilities = counts / len(y)  
    gini = 1 - np.sum(probabilities ** 2)  
    return gini
```

```
def _gini_index(self, X, y, feature_index, threshold):  
    left_labels = y[X[:, feature_index] <= threshold]  
    right_labels = y[X[:, feature_index] > threshold]  
  
    left_gini = self._gini_impurity(left_labels)  
    right_gini = self._gini_impurity(right_labels)  
  
    num_left = len(left_labels)  
    num_right = len(right_labels)
```

```
    total_samples = len(X)  
    gini_index = (num_left / total_samples) * left_gini + (num_right / total_samples) * right_gini  
    return gini_index
```



These two methods are utility methods for calculating the Gini impurity and Gini index, which are alternative measures of impurity similar to entropy. The `gini_impurity` method takes a target vector  $y$  and calculates the Gini impurity by computing the probability of each class label and summing the squared probabilities. The `gini_index` method takes the feature matrix  $X$ , target vector  $y$ , feature index, and threshold, and calculates the Gini index for a split based on the given feature and threshold. It computes the Gini impurity of the left and right child sets and weights them by the proportion of samples in each child set to calculate the overall Gini index.

```
def fit(self, X, y):  
    self.root = self.build(X, y)
```

The fit method is used to train the decision tree classifier. It takes the feature matrix  $X$  and target vector  $y$  as input and calls the build method to construct the decision tree. The resulting tree is assigned to the root attribute of the DecisionTree object.





```
def _predict(self, x, tree):  
    if tree.value is not None:  
        return tree.value  
    feature_value = x[tree.feature]  
    if feature_value <= tree.threshold:  
        return self._predict(x, tree.data_left)  
    if feature_value > tree.threshold:  
        return self._predict(x, tree.data_right)  
  
def predict(self, X):  
    return [self._predict(x, self.root) for x in X]
```

These two methods are used for making predictions using the trained decision tree. The `_predict` method recursively traverses the tree starting from the tree node and returns the predicted class label for a given input `x`. If the current node is a terminal node (leaf), it returns the stored

- ▶ And after that I read the csv file and try to analyse the data.
- ▶ The first step is convert string columns to integer type to analyse easier. For this action I write below code.

```
▶ from sklearn.preprocessing import LabelEncoder  
columns_to_convert = ['Gender', 'Customer Type', 'Type of Travel', 'Class', 'satisfaction']  
label_encoder = LabelEncoder()  
for column in columns_to_convert:  
    data[column] = label_encoder.fit_transform(data[column])
```

I use Label\_encoder function. Instead of this I can choose another way like mapping. Replacing, OrdinalEncoder from sklearn.preprocessing.


```
from sklearn.tree import DecisionTreeClassifier
from sklearn.impute import KNNImputer
from sklearn.metrics import accuracy_score
X = data.iloc[:3000, :-1]
y = data.iloc[:3000, -1]
imputer = KNNImputer(n_neighbors=3)
X_imputed = pd.DataFrame(imputer.fit_transform(X), columns=X.columns)
from sklearn.model_selection import train_test_split
np.random.seed(42)

indices = np.arange(len(X))
np.random.shuffle(indices)


split_ratio = 0.8
train_size = int(split_ratio * len(X))

X_train = X_imputed.iloc[indices[:train_size]].values
y_train = y[indices[:train_size]].values
X_test = X_imputed.iloc[indices[train_size:]].values
y_test = y[indices[train_size:]].values
sk_model = DecisionTreeClassifier()
sk_model.fit(X_train, y_train)
sk_preds = sk_model.predict(X_test)
# print(type(X_train))
# print(type(X_train))
res_pack = accuracy_score(y_test, sk_preds)
print('the accuracy of decisionTree of sklearn is :', res_pack)
```

```
the accuracy of decisionTree of sklearn is : 0.9033333333333333
```



And after that I decide to change None value of dataframe to valuable data. For this I use KNNImputer. Instead of this I could even use simpleImputer or drop function. And split the data to train and test data with shuffling the data frame. 20 percent of it is for test data.



I use `accuracy_score` for get the accuracy of my method.  
and give my train and test data to my tree and  
`DecisionTreeClassifier` function and compare these to each other.

I wrote this code just for analyzing.

```
import matplotlib.pyplot as plt
for column in data.columns:
    plt.hist(data[column], bins=40, edgecolor='black')
    plt.xlabel(column)
    plt.ylabel('Frequency')
    plt.title(f'Histogram of {column}')
    plt.show()
```

## The comparison between scratch code and package

```
✓ 22s [598] model = DecisionTree()  
      model.fit(X_train, y_train)  
      preds = model.predict(X_test)
```

```
✓ 0s [599] res_mytree1 = accuracy_score(y_test, preds)
```

```
✓ 0s [600] print('the accuracy of my implementation tree is : ', res_mytree1)
```

```
the accuracy of my implementation tree is : 0.915
```

```
✓ 0s [601] if res_pack < res_mytree1:  
        print('the accuracy of decision tree from scratch is better than decision Tree from sklearn model.')  
    else:  
        print('the accuracy of decision Tree from sklearn model is better than decision tree from scratch.')
```

```
the accuracy of decision tree from scratch is better than decision Tree from sklearn model.
```

```
[ ] correlation_matrix = data.corr()

# Display the correlation matrix
print(correlation_matrix)
```

After this implementation I decided to more preprocessing on the data. At first I calculate the correlation between features and in the second way I show its chart with heatmap.

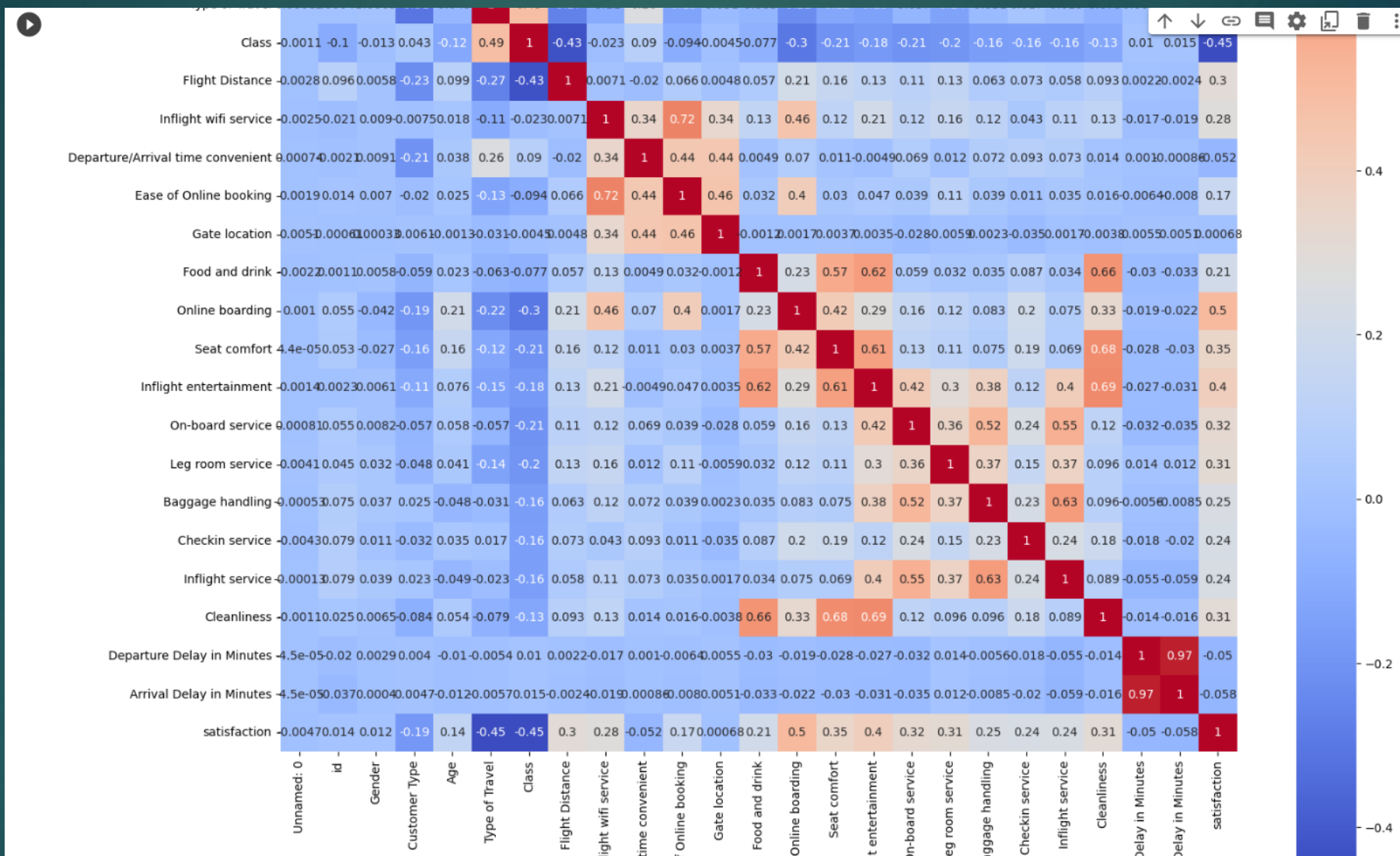
```
import seaborn as sns
import matplotlib.pyplot as plt


# Plot the correlation matrix heatmap
plt.figure(figsize=(25, 25))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', square=True)

# Add title
plt.title('Correlation Matrix')

# Show the plot
plt.show()
```







according to the heatmap we can delete the Gate Location feature and after that analyse it again.

# Another attempt for preprocessing: I use standardScaler to normalize my data

```
▶ import pandas as pd
   from sklearn.preprocessing import StandardScaler

   columns_to_standardize = data.columns
   scaler = StandardScaler()
   scaler.fit(data[columns_to_standardize])
   data[columns_to_standardize] = scaler.transform(data[columns_to_standardize])

   # Display the standardized DataFrame
   print(data)
```

I use SimpleImputer to convert my None value data to sth else and I compare the accuracy between scaled data and past data




```
from sklearn.model_selection import train_test_split
from sklearn.impute import SimpleImputer

X = data.iloc[:1000, :-1]
y = data.iloc[:1000, -1]
imputer = SimpleImputer(missing_values = np.nan, strategy = 'mean')
X_imputed = pd.DataFrame(imputer.fit_transform(X), columns=X.columns)

# Split the data into train and test sets
X_train1, X_test1, y_train1, y_test1 = train_test_split(X_imputed.values, y.values, test_size=0.2, random_state=42)

model2 = DecisionTree()
model2.fit(X_train1, y_train1)
preds1 = model2.predict(X_test1)
```



My scratch code has better accuracy but it is very slow and I could not run it on the google colab. So I just use the 3000 samples of my data.

I think if I had more time I could spend more time to clean and preprocess of my data and use matplotlib to analyze the data. But now don't have time.

And even I could test another way to split data for train and test data like split the whole data to 1000 samples and after that I replace each elements to mean and after that use unique data and use them and compare it to the previous data.