3.12

The dummy variable trap is a challenge that arises when a categorical variable is transformed into multiple binary variables through the process of hot encoding. This can result in a bias (constant term) that leads to multiple sets of parameters that provide equally good fits to the training data. This can cause confusion and make it difficult to determine the most appropriate parameters. The solution to this issue is through the use of regularization, which helps to overcome the bias and provide a more accurate representation of the data.

3.14

3.14 A)

The occurrence of default being treated as the desired result influences the maximum likelihood objective function, causing a shift in both the direction and magnitude of the bias and the weights. However, the calculated probabilities of either defaulting or not defaulting remain unaffected.

In [1]:

```
import warnings
warnings.filterwarnings('ignore')

import pandas as pd
import numpy as np

import seaborn as sns
import matplotlib.pyplot as plt

from sklearn.linear_model import LogisticRegression

from sklearn.metrics import accuracy_score, recall_score, precision_score, f1_score
from sklearn.metrics import confusion_matrix, classification_report, roc_curve, roc_auc_score
from sklearn.metrics import precision_recall_curve, auc, average_precision_score
```

In [2]:

```
train = pd.read_excel('lendingclub_traindata.xlsx')
validation=pd.read_excel('lendingclub_valdata.xlsx')
test = pd.read_excel('lendingclub_testdata.xlsx')

# 1 = good, 0 = default
cols = ['home_ownership', 'income', 'dti', 'fico', 'loan_status']

train.columns = validation.columns=test.columns = cols
```

In [3]:

```
X_train = train.drop('loan_status', 1)
X_val=validation.drop('loan_status', 1)
X_test = test.drop('loan_status', 1)

# Scale data
X_test=(X_test-X_train.mean())/X_train.std()
X_val=(X_val-X_train.mean())/X_train.std()
X_train=(X_train-X_train.mean())/X_train.std()

y_train = train['loan_status']
y_val=validation['loan_status']
y_test = test['loan_status']

print(X_train.shape, y_train.shape, X_val.shape,y_val.shape, X_test.shape, y_test.shape)
```

```
(7000, 4) (7000,) (3000, 4) (3000,) (2290, 4) (2290,)
```

In [4]:

```
#swapping 0 and 1
train_log = train
validation_log = validation
test_log = test

train_log.loc[train_log['loan_status'] == 1, 'reverse_loan_status'] = 0
train_log.loc[train_log['loan_status'] == 0, 'reverse_loan_status'] = 1

validation_log.loc[validation_log['loan_status'] == 1, 'reverse_loan_status'] = 0
validation_log.loc[validation_log['loan_status'] == 0, 'reverse_loan_status'] = 1

test_log.loc[test_log['loan_status'] == 1, 'reverse_loan_status'] = 0
test_log.loc[test_log['loan_status'] == 0, 'reverse_loan_status'] = 1

# store target column as y-variables
y_log_train=train_log['reverse_loan_status']
y_log_val=validation_log['reverse_loan_status']
y_log_test = test_log['reverse_loan_status']
```

In [5]:

```
#frequncy before the swap
freq = y_train.value_counts()
print('frequncy before the swap \n',freq/sum(freq)*100 )

#frequncy after the swap
freq_log = y_log_train.value_counts()
freq_log/sum(freq_log)*100
print('frequncy after the swap \n',freq_log/sum(freq_log)*100 )
```

```
frequncy before the swap
 1    79.171429
0    20.828571
Name: loan_status, dtype: float64
frequncy after the swap
 0.0    79.171429
1.0    20.828571
Name: reverse_loan_status, dtype: float64
```

In [6]:

```
#Log regression before the swap

lgstc_reg =  LogisticRegression(penalty="none",solver="newton-cg")
lgstc_reg.fit(X_train, y_train)
print('intercept and coefficient before the swap \n',lgstc_reg.intercept_, lgstc_reg.coef_)

#Log regression after the swap

lgstc_reg_log =  LogisticRegression(penalty="none",solver="newton-cg")
lgstc_reg_log.fit(X_train, y_log_train)
print('intercept and coefficient after the swap \n',lgstc_reg_log.intercept_, lgstc_reg_log.coef_)
```

```
intercept and coefficient before the swap
 [1.4162429] [[ 0.14531037  0.03366005 -0.32404502  0.36315462]]
intercept and coefficient after the swap
 [-1.4162429] [[-0.14531037 -0.03366005  0.32404502 -0.36315462]]
```

In [7]:

```python
#cost function beofore the swap
y_train_pred=lgstc_reg.predict_proba(X_train)
y_val_pred=lgstc_reg.predict_proba(X_val)
y_test_pred=lgstc_reg.predict_proba(X_test)

mle_vector_train = np.log(np.where(y_train == 1, y_train_pred[:,1], y_train_pred[:,0]))
mle_vector_val = np.log(np.where(y_val == 1, y_val_pred[:,1], y_val_pred[:,0]))
mle_vector_test = np.log(np.where(y_test == 1, y_test_pred[:,1], y_test_pred[:,0]))

cost_function_training=np.negative(np.sum(mle_vector_train)/len(y_train))
cost_function_val=np.negative(np.sum(mle_vector_val)/len(y_val))
cost_function_test=np.negative(np.sum(mle_vector_test)/len(y_test))

print('cost function beofore the swap:')
print('cost function training set =', cost_function_training)
print('cost function validation set =', cost_function_val)
print('cost function test set =', cost_function_test)


#cost function after the swap
y_log_train_pred=lgstc_reg_log.predict_proba(X_train)
y_log_val_pred=lgstc_reg_log.predict_proba(X_val)
y_log_test_pred=lgstc_reg_log.predict_proba(X_test)

mle_vector_train_log = np.log(np.where(y_log_train == 1, y_log_train_pred[:,1], y_log_train_pred[:,0]))
mle_vector_val_log = np.log(np.where(y_log_val == 1, y_log_val_pred[:,1], y_log_val_pred[:,0]))
mle_vector_test_log = np.log(np.where(y_log_test == 1, y_log_test_pred[:,1], y_log_test_pred[:,0]))

cost_function_training_log=np.negative(np.sum(mle_vector_train_log)/len(y_log_train))
cost_function_val_log=np.negative(np.sum(mle_vector_val_log)/len(y_log_val))
cost_function_test_log=np.negative(np.sum(mle_vector_test_log)/len(y_log_test))

print('\ncost function after the swap:')
print('cost function training set =', cost_function_training_log)
print('cost function validation set =', cost_function_val_log)
print('cost function test set =', cost_function_test_log)
```

```
cost function beofore the swap:
cost function training set = 0.49111143543170926
cost function validation set = 0.4860713220392096
cost function test set = 0.48467054875810117

cost function after the swap:
cost function training set = 0.49111143543170926
cost function validation set = 0.4860713220392096
cost function test set = 0.48467054875810117
```

In [8]:

```
#Confusion matrix before the swap
print('Confusion matrix before the swap')
THRESHOLD = [.75, .80, .85]
results = pd.DataFrame(columns=["THRESHOLD", "accuracy", "true pos rate", "true neg rate", "false pos rate", "precision", "f-

results['THRESHOLD'] = THRESHOLD

j = 0

for i in THRESHOLD:
    preds = np.where(lgstc_reg.predict_proba(X_test)[:,1] > i, 1, 0)
    cm = (confusion_matrix(y_test, preds,labels=[1, 0], sample_weight=None) / len(y_test))*100                   # confusion

    print('Confusion matrix for threshold =',i)
    print(cm)
    print(' ')

    TP = cm[0][0]                                                                                          # True Positives
    FN = cm[0][1]                                                                                          # False Positives
    FP = cm[1][0]                                                                                          # True Negatives
    TN = cm[1][1]                                                                                          # False Negatives

    results.iloc[j,1] = accuracy_score(y_test, preds)
    results.iloc[j,2] = recall_score(y_test, preds)
    results.iloc[j,3] = TN/(FP+TN)                                                                         # True negative ra
    results.iloc[j,4] = FP/(FP+TN)                                                                         # False positive r
    results.iloc[j,5] = precision_score(y_test, preds)
    results.iloc[j,6] = f1_score(y_test, preds)


    j += 1

print('ALL METRICS')
print( results.T)

#Confusion matrix after the swap
print('\nConfusion matrix after the swap')
THRESHOLD_log = [.25, .20, .15]
results_log = pd.DataFrame(columns=["THRESHOLD", "accuracy", "true pos rate", "true neg rate", "false pos rate", "precision",
results_log['THRESHOLD'] = THRESHOLD_log

j = 0
for i in THRESHOLD_log:
    preds_log = np.where(lgstc_reg_log.predict_proba(X_test)[:,1] > i, 1, 0)
    cm_log = (confusion_matrix(y_log_test, preds_log,labels=[1, 0], sample_weight=None) / len(y_log_test))*100

    print('Confusion matrix for threshold =',i)
    print(cm_log)
    print(' ')

    TP = cm_log[0][0]                                                                                          # True Positiv
    FN = cm_log[0][1]                                                                                          # False Positi
    FP = cm_log[1][0]                                                                                          # True Negativ
    TN = cm_log[1][1]                                                                                          # False Negati

    results_log.iloc[j,1] = accuracy_score(y_log_test, preds_log)
    results_log.iloc[j,2] = recall_score(y_log_test, preds_log)
    results_log.iloc[j,3] = TN/(FP+TN)                                                                         # True negativ
    results_log.iloc[j,4] = FP/(FP+TN)                                                                         # False positi
    results_log.iloc[j,5] = precision_score(y_log_test, preds_log)
    results_log.iloc[j,6] = f1_score(y_log_test, preds_log)


    j += 1

print('ALL METRICS')
print( results_log.T)
```

```
Confusion matrix before the swap
Confusion matrix for threshold = 0.75
[[60.82969432 18.34061135]
 [11.70305677  9.12663755]]

Confusion matrix for threshold = 0.8
[[42.70742358 36.4628821 ]
 [ 6.4628821  14.36681223]]

Confusion matrix for threshold = 0.85
[[22.7510917  56.41921397]
 [ 3.01310044 17.81659389]]

ALL METRICS
                       0         1         2
THRESHOLD           0.75       0.8      0.85
accuracy        0.699563  0.570742  0.405677
true pos rate    0.76834  0.539437  0.287369
true neg rate   0.438155  0.689727  0.855346
false pos rate  0.561845  0.310273  0.144654
precision       0.838651  0.868561  0.883051
f-score         0.801957  0.665532  0.433625

Confusion matrix after the swap
Confusion matrix for threshold = 0.25
[[ 9.12663755 11.70305677]
 [18.34061135 60.82969432]]

Confusion matrix for threshold = 0.2
[[14.36681223  6.4628821 ]
 [36.4628821  42.70742358]]

Confusion matrix for threshold = 0.15
[[17.81659389  3.01310044]
 [56.41921397 22.7510917 ]]

ALL METRICS
                       0         1         2
THRESHOLD           0.25       0.2      0.15
accuracy        0.699563  0.570742  0.405677
true pos rate   0.438155  0.689727  0.855346
true neg rate    0.76834  0.539437  0.287369
false pos rate   0.23166  0.460563  0.712631
precision       0.332273  0.282646      0.24
f-score         0.377939  0.400975  0.374828
```

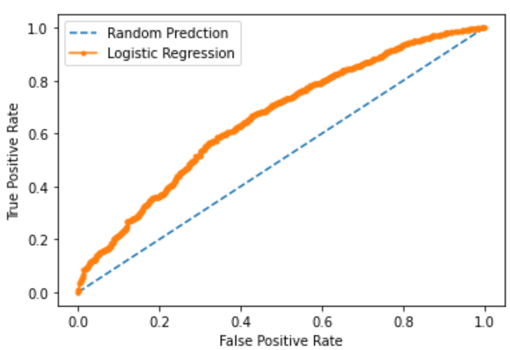
3.14 B)

please refer to the above matrix and metric above

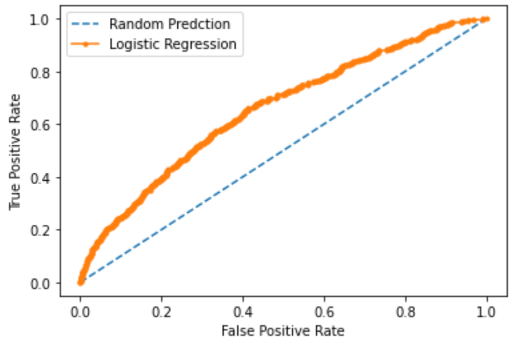
In [9]:

```
# Calculate the receiver operating curve and the AUC measure before swapping

print('Calculate the receiver operating curve and the AUC measure before swapping ')
lr_prob=lgstc_reg.predict_proba(X_test)
lr_prob=lr_prob[:, 1]
ns_prob=[0 for _ in range(len(y_test))]
ns_auc=roc_auc_score(y_test, ns_prob)
lr_auc=roc_auc_score(y_test,lr_prob)
print("AUC random predictions =", ns_auc)
print("AUC predictions from logistic regression model =", lr_auc)
ns_fpr,ns_tpr,_=roc_curve(y_test,ns_prob)
lr_fpr,lr_tpr,_=roc_curve(y_test,lr_prob)

plt.plot(ns_fpr,ns_tpr,linestyle='--',label='Random Predction')
plt.plot(lr_fpr,lr_tpr,marker='.',label='Logistic Regression')

plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend()
plt.show()



# Calculate the receiver operating curve and the AUC measure after swapping
print('Calculate the receiver operating curve and the AUC measure after swapping ')
lr_prob_log=lgstc_reg_log.predict_proba(X_test)
lr_prob_log=lr_prob_log[:, 1]
ns_prob_log=[0 for _ in range(len(y_log_test))]
ns_auc_log=roc_auc_score(y_log_test, ns_prob_log)
lr_auc_log=roc_auc_score(y_log_test,lr_prob_log)
print("AUC random predictions =", ns_auc_log)
print("AUC predictions from logistic regression model =", lr_auc_log)
ns_fpr_log,ns_tpr_log,_=roc_curve(y_log_test,ns_prob_log)
lr_fpr_log,lr_tpr_log,_=roc_curve(y_log_test,lr_prob_log)

plt.plot(ns_fpr_log,ns_tpr_log,linestyle='--',label='Random Predction')
plt.plot(lr_fpr_log,lr_tpr_log,marker='.',label='Logistic Regression')

plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend()
plt.show()
```

```
Calculate the receiver operating curve and the AUC measure before swapping
AUC random predictions = 0.5
AUC predictions from logistic regression model = 0.6577628841779786
```




```
Calculate the receiver operating curve and the AUC measure after swapping
AUC random predictions = 0.5
AUC predictions from logistic regression model = 0.6577628841779785
```



3.14 C)

when the new z value is minus the old z value, the new false positive rate is equal to the old true positive rate. It is also noted that the new true positive rate minus the new false positive rate is equal to the old true positive rate minus the old false positive rate; such symmetry leads to the same ROC curve and same AUC.

4.11

In [10]:

```
#Confusion matrix for Z = 0.9
THRESHOLD = [0.9]
results = pd.DataFrame(columns=["THRESHOLD", "accuracy", "true pos rate", "true neg rate", "false pos rate", "precision", "f-

results['THRESHOLD'] = THRESHOLD

j = 0

for i in THRESHOLD:
    preds = np.where(lgstc_reg.predict_proba(X_test)[:,1] > i, 1, 0)
    cm = (confusion_matrix(y_test, preds,labels=[1, 0], sample_weight=None) / len(y_test))*100               # confusion

    print('Confusion matrix for threshold =',i)
    print(cm)
    print(' ')
```

```
Confusion matrix for threshold = 0.9
[[ 7.6419214  71.52838428]
 [ 0.48034934 20.34934498]]
```

Predict that loans are good only if FICO > 722.5 and dti ≤ 19.85, confusion matrix plase refer to above

4.12

Conditional on a good loan, the probability density for a FICO score of 700 is

=1/(SQRT(2*PI())*32.85)*EXP(-((700-697.38)^2/(2*32.85^2))) =0.012105797

Conditional on a good loan, the probability density function for a dti of 10 is

=(1/(SQRT(2*PI())*8.72))*EXP(-((10-17.37)^2/(2*8.72^2)))=0.032009411

Conditional on a default, the probability density for a FICO score of 700 is

=1/(SQRT(2*PI())*24.26)*EXP(-((700-686.73)^2/(2*24.26^2)))=0.014159536

conditional on a default, the probability density for a dti of 10 is

=(1/(SQRT(2*PI())*9.11))*EXP(-((10-20.41)^2/(2*9.11^2)))=0.022795474

The probability of the loan being good is

=0.7917*0.012105797*0.032009411=0.000306783

where Q is the joint probability density of FICO =700 and dti =10. The probability of the loan defaulting is

=0.2083*0.014159536*0.022795474=6.72337E-05

The probability of the loan defaulting is

6.72337E-05/(0.000306783+6.72337E-05)=0.179761045

4.13

In [11]:
```
from sklearn.tree import DecisionTreeClassifier, plot_tree, export_graphviz, export_text


train_tree = pd.read_excel('lendingclub_traindata.xlsx')
validation_tree=pd.read_excel('lendingclub_valdata.xlsx')
test_tree=pd.read_excel('lendingclub_testdata.xlsx')

X_tree_train = train_tree.drop('loan_status',axis=1)
X_tree_val=validation_tree.drop('loan_status',axis=1)
X_tree_test=test_tree.drop('loan_status',axis=1)


y_tree_train = train_tree['loan_status']
y_tree_val=validation_tree['loan_status']
y_tree_test=test_tree['loan_status']


print(X_tree_train.shape, y_tree_train.shape, X_tree_val.shape,y_tree_val.shape,X_tree_test.shape,y_tree_test.shape)
```

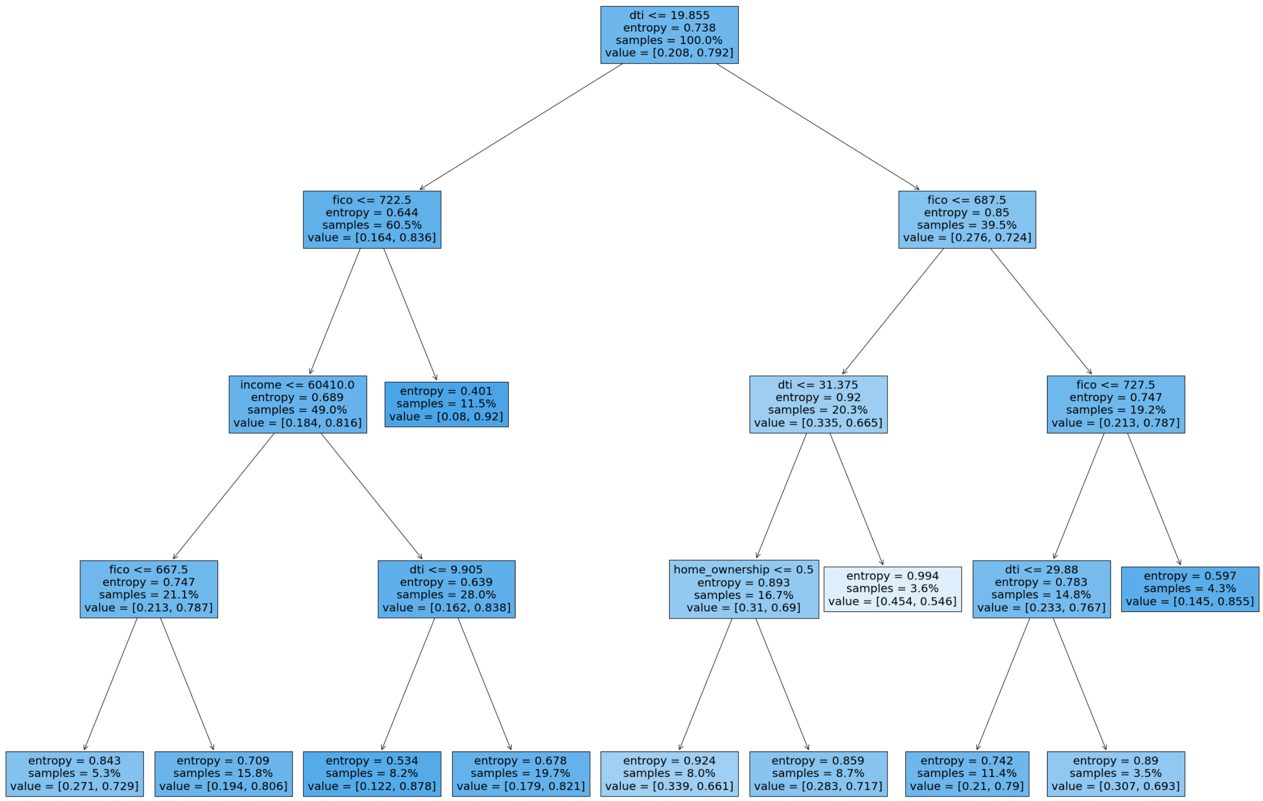
```
(7000, 4) (7000,) (3000, 4) (3000,) (2290, 4) (2290,)
```

In [12]:
```
X_tree_train.columns
```

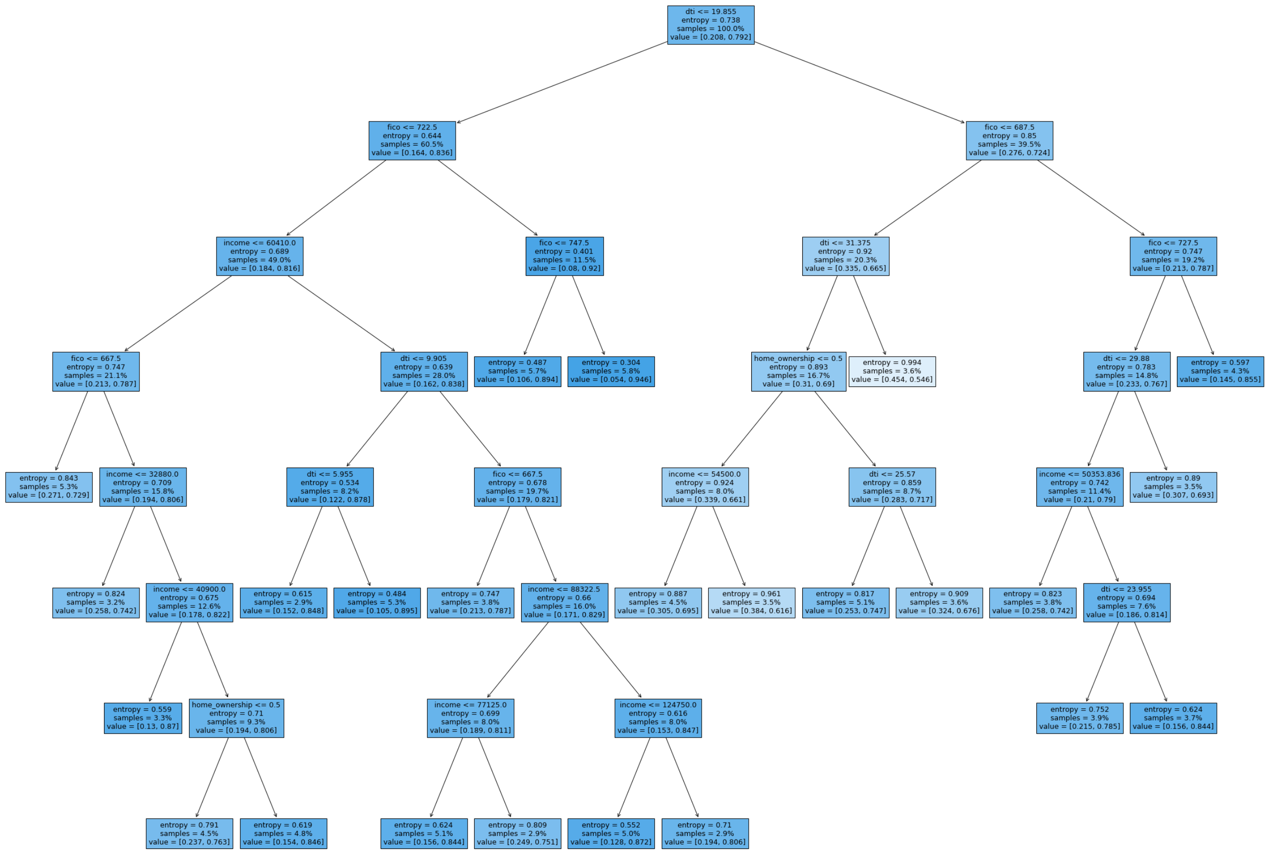
Out[12]:
```
Index(['home_ownership', 'income', 'dti', 'fico'], dtype='object')
```

In [13]:

```
clf = DecisionTreeClassifier(criterion='entropy',max_depth=4,min_samples_split=1000,min_samples_leaf=200,random_state=0)
clf = clf.fit(X_tree_train,y_tree_train)
fig, ax = plt.subplots(figsize=(40, 30))
plot_tree(clf, filled=True, feature_names=X_train.columns, proportion=True)
plt.show()
```



In [14]:
```
clf = DecisionTreeClassifier(criterion='entropy',max_depth=8,min_samples_split=500,min_samples_leaf=200,random_state=0)
clf = clf.fit(X_tree_train,y_tree_train)
fig, ax = plt.subplots(figsize=(40, 30))
plot_tree(clf, filled=True, feature_names=X_train.columns, proportion=True)
plt.show()
```



The maximum depth of a decision tree refers to the number of layers or levels in the tree. Changing the maximum depth of a decision tree affects the model in several ways:

Overfitting and underfitting: If the maximum depth of the tree is set too high, the model may overfit the training data, leading to poor generalization performance on unseen data. On the other hand, if the maximum depth is set too low, the model may underfit the data, meaning it won't capture the underlying patterns and relationships.

Complexity: Increasing the maximum depth increases the complexity of the model, making it harder to understand and interpret.

Training time: As the depth of the tree increases, the number of nodes and the computational cost of training the model also increases, leading to longer training times.

Accuracy: The accuracy of the model depends on the maximum depth of the tree. A model with a high maximum depth may produce accurate results, but the model with a lower maximum depth may generalize better to unseen data.

In conclusion, the optimal maximum depth of a decision tree should strike a balance between accuracy and complexity. It's a trade-off between fitting the data well and avoiding overfitting.

Changing the minimum number of samples necessary for a split in a decision tree can affect the structure and performance of the tree.

If the minimum number is set too high, the tree may become too shallow and not capture the complexities of the data, leading to overfitting or poor generalization to new data. On the other hand, if the minimum number is set too low, the tree may become too complex and difficult to interpret, leading to overfitting and a high risk of over-complicating the model.Adjusting the minimum number of samples necessary for a split in a decision tree can have a significant impact on the tree's performance and should be carefully considered in the model selection process.

In [ ]: