In [1]:
```
import pandas as pd
import numpy as np
import seaborn as sns
```

In [2]:
```
FraudC = pd.read_csv("Fraud_check.csv")
FraudC.head(10)
```

Out[2]:

	Undergrad	Marital.Status	Taxable.Income	City.Population	Work.Experience	Urban
0	NO	Single	68833	50047	10	YES
1	YES	Divorced	33700	134075	18	YES
2	NO	Married	36925	160205	30	YES
3	YES	Single	50190	193264	15	YES
4	NO	Married	81002	27533	28	NO
5	NO	Divorced	33329	116382	0	NO
6	NO	Divorced	83357	80890	8	YES
7	YES	Single	62774	131253	3	YES
8	NO	Single	83519	102481	12	YES
9	YES	Divorced	98152	155482	4	YES

In [3]:
```
FraudC
```

Out[3]:

	Undergrad	Marital.Status	Taxable.Income	City.Population	Work.Experience	Urban
0	NO	Single	68833	50047	10	YES
1	YES	Divorced	33700	134075	18	YES
2	NO	Married	36925	160205	30	YES
3	YES	Single	50190	193264	15	YES
4	NO	Married	81002	27533	28	NO
...	...	...	...	...	...	...
595	YES	Divorced	76340	39492	7	YES
596	YES	Divorced	69967	55369	2	YES
597	NO	Divorced	47334	154058	0	YES
598	YES	Married	98592	180083	17	NO
599	NO	Divorced	96519	158137	16	NO

600 rows × 6 columns

In [4]:
```
FraudC.shape
```

Out[4]: (600, 6)

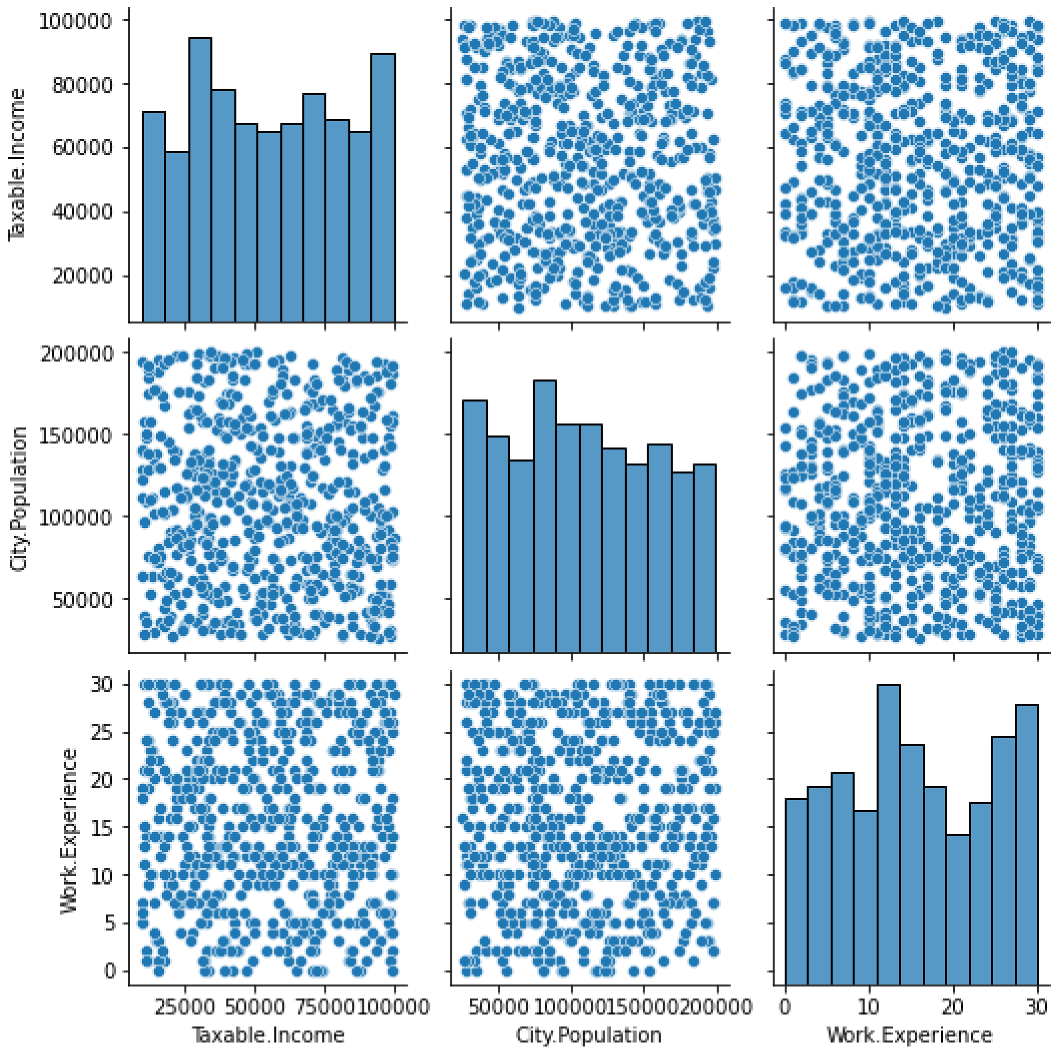
In [5]:
```
FraudC.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 600 entries, 0 to 599
Data columns (total 6 columns):
 #   Column           Non-Null Count  Dtype 
---  ------           --------------  ----- 
 0   Undergrad        600 non-null    object
 1   Marital.Status   600 non-null    object
 2   Taxable.Income   600 non-null    int64 
 3   City.Population  600 non-null    int64 
 4   Work.Experience  600 non-null    int64 
 5   Urban            600 non-null    object
dtypes: int64(3), object(3)
memory usage: 28.2+ KB
```

In [6]:
```
FraudC.columns
FraudC.isnull().sum()
```

Out[6]:
```
Undergrad          0
Marital.Status     0
Taxable.Income     0
City.Population    0
Work.Experience    0
Urban              0
dtype: int64
```

In [7]:
```
sns.pairplot(FraudC)
FraudC["TaxInc"] = pd.cut(FraudC["Taxable.Income"], bins = [10002,30000,99620], labe
FraudCheck = FraudC.drop(columns=["Taxable.Income"])
```



In [8]:
```
FCheck = pd.get_dummies(FraudCheck.drop(columns = ["TaxInc"]))
FraudC_final = pd.concat([FCheck, FraudCheck["TaxInc"]], axis = 1)
colnames = list(FraudC_final.columns)
colnames
```

Out[8]:
```
['City.Population',
 'Work.Experience',
 'Undergrad_NO',
 'Undergrad_YES',
 'Marital.Status_Divorced',
 'Marital.Status_Married',
 'Marital.Status_Single',
 'Urban_NO',
 'Urban_YES',
 'TaxInc']
```

In [9]:
```
predictors = colnames[:9]
predictors
target = colnames[9]
target
```

Out[9]:
```
'TaxInc'
```

In [10]:
```
X = FraudC_final[predictors]
X.shape
Y = FraudC_final[target]
```

Decision Tree Building

In [11]:
```
from sklearn.model_selection import train_test_split
train, test = train_test_split(FraudC_final, test_size = 0.3)
FraudC_final["TaxInc"].unique()
```

Out[11]:
```
['Good', 'Risky']
Categories (2, object): ['Risky' < 'Good']
```

In [12]:
```
from sklearn.tree import DecisionTreeClassifier
help(DecisionTreeClassifier)
modelTree = DecisionTreeClassifier(criterion = "entropy")
modelTree.fit(train[predictors], train[[target]])
type([target])
type(predictors)
```

```
Help on class DecisionTreeClassifier in module sklearn.tree._classes:

class DecisionTreeClassifier(sklearn.base.ClassifierMixin, BaseDecisionTree)
 |  DecisionTreeClassifier(*, criterion='gini', splitter='best', max_depth=None, min
_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features=Non
e, random_state=None, max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_s
plit=None, class_weight=None, ccp_alpha=0.0)
 |
 |  A decision tree classifier.
 |
 |  Read more in the :ref:`User Guide <tree>`.
 |
 |  Parameters
 |  ----------
 |  criterion : {"gini", "entropy"}, default="gini"
 |      The function to measure the quality of a split. Supported criteria are
```

```
 |      "gini" for the Gini impurity and "entropy" for the information gain.
 |  
 |  splitter : {"best", "random"}, default="best"
 |      The strategy used to choose the split at each node. Supported
 |      strategies are "best" to choose the best split and "random" to choose
 |      the best random split.
 |  
 |  max_depth : int, default=None
 |      The maximum depth of the tree. If None, then nodes are expanded until
 |      all leaves are pure or until all leaves contain less than
 |      min_samples_split samples.
 |  
 |  min_samples_split : int or float, default=2
 |      The minimum number of samples required to split an internal node:
 |  
 |      - If int, then consider `min_samples_split` as the minimum number.
 |      - If float, then `min_samples_split` is a fraction and
 |        `ceil(min_samples_split * n_samples)` are the minimum
 |        number of samples for each split.
 |  
 |      .. versionchanged:: 0.18
 |         Added float values for fractions.
 |  
 |  min_samples_leaf : int or float, default=1
 |      The minimum number of samples required to be at a leaf node.
 |      A split point at any depth will only be considered if it leaves at
 |      least ``min_samples_leaf`` training samples in each of the left and
 |      right branches.  This may have the effect of smoothing the model,
 |      especially in regression.
 |  
 |      - If int, then consider `min_samples_leaf` as the minimum number.
 |      - If float, then `min_samples_leaf` is a fraction and
 |        `ceil(min_samples_leaf * n_samples)` are the minimum
 |        number of samples for each node.
 |  
 |      .. versionchanged:: 0.18
 |         Added float values for fractions.
 |  
 |  min_weight_fraction_leaf : float, default=0.0
 |      The minimum weighted fraction of the sum total of weights (of all
 |      the input samples) required to be at a leaf node. Samples have
 |      equal weight when sample_weight is not provided.
 |  
 |  max_features : int, float or {"auto", "sqrt", "log2"}, default=None
 |      The number of features to consider when looking for the best split:
 |  
 |          - If int, then consider `max_features` features at each split.
 |          - If float, then `max_features` is a fraction and
 |            `int(max_features * n_features)` features are considered at each
 |            split.
 |          - If "auto", then `max_features=sqrt(n_features)`.
 |          - If "sqrt", then `max_features=sqrt(n_features)`.
 |          - If "log2", then `max_features=log2(n_features)`.
 |          - If None, then `max_features=n_features`.
 |  
 |      Note: the search for a split does not stop until at least one
 |      valid partition of the node samples is found, even if it requires to
 |      effectively inspect more than ``max_features`` features.
 |  
 |  random_state : int, RandomState instance or None, default=None
 |      Controls the randomness of the estimator. The features are always
 |      randomly permuted at each split, even if ``splitter`` is set to
 |      ``"best"``. When ``max_features < n_features``, the algorithm will
 |      select ``max_features`` at random at each split before finding the best
 |      split among them. But the best found split may vary across different
 |      runs, even if ``max_features=n_features``. That is the case, if the
 |      improvement of the criterion is identical for several splits and one
 |      split has to be selected at random. To obtain a deterministic behaviour
 |      during fitting, ``random_state`` has to be fixed to an integer.
```

```
 |      See :term:`Glossary <random_state>` for details.
 |  
 |  max_leaf_nodes : int, default=None
 |      Grow a tree with ``max_leaf_nodes`` in best-first fashion.
 |      Best nodes are defined as relative reduction in impurity.
 |      If None then unlimited number of leaf nodes.
 |  
 |  min_impurity_decrease : float, default=0.0
 |      A node will be split if this split induces a decrease of the impurity
 |      greater than or equal to this value.
 |  
 |      The weighted impurity decrease equation is the following::
 |  
 |          N_t / N * (impurity - N_t_R / N_t * right_impurity
 |                              - N_t_L / N_t * left_impurity)
 |  
 |      where ``N`` is the total number of samples, ``N_t`` is the number of
 |      samples at the current node, ``N_t_L`` is the number of samples in the
 |      left child, and ``N_t_R`` is the number of samples in the right child.
 |  
 |      ``N``, ``N_t``, ``N_t_R`` and ``N_t_L`` all refer to the weighted sum,
 |      if ``sample_weight`` is passed.
 |  
 |      .. versionadded:: 0.19
 |  
 |  min_impurity_split : float, default=0
 |      Threshold for early stopping in tree growth. A node will split
 |      if its impurity is above the threshold, otherwise it is a leaf.
 |  
 |      .. deprecated:: 0.19
 |         ``min_impurity_split`` has been deprecated in favor of
 |         ``min_impurity_decrease`` in 0.19. The default value of
 |         ``min_impurity_split`` has changed from 1e-7 to 0 in 0.23 and it
 |         will be removed in 1.0 (renaming of 0.25).
 |         Use ``min_impurity_decrease`` instead.
 |  
 |  class_weight : dict, list of dict or "balanced", default=None
 |      Weights associated with classes in the form ``{class_label: weight}``.
 |      If None, all classes are supposed to have weight one. For
 |      multi-output problems, a list of dicts can be provided in the same
 |      order as the columns of y.
 |  
 |      Note that for multioutput (including multilabel) weights should be
 |      defined for each class of every column in its own dict. For example,
 |      for four-class multilabel classification weights should be
 |      [{0: 1, 1: 1}, {0: 1, 1: 5}, {0: 1, 1: 1}, {0: 1, 1: 1}] instead of
 |      [{1:1}, {2:5}, {3:1}, {4:1}].
 |  
 |      The "balanced" mode uses the values of y to automatically adjust
 |      weights inversely proportional to class frequencies in the input data
 |      as ``n_samples / (n_classes * np.bincount(y))``
 |  
 |      For multi-output, the weights of each column of y will be multiplied.
 |  
 |      Note that these weights will be multiplied with sample_weight (passed
 |      through the fit method) if sample_weight is specified.
 |  
 |  ccp_alpha : non-negative float, default=0.0
 |      Complexity parameter used for Minimal Cost-Complexity Pruning. The
 |      subtree with the largest cost complexity that is smaller than
 |      ``ccp_alpha`` will be chosen. By default, no pruning is performed. See
 |      :ref:`minimal_cost_complexity_pruning` for details.
 |  
 |      .. versionadded:: 0.22
 |  
 |  Attributes
 |  ----------
 |  classes_ : ndarray of shape (n_classes,) or list of ndarray
 |      The classes labels (single output problem),
```

```
 |          or a list of arrays of class labels (multi-output problem).
 |
 |      feature_importances_ : ndarray of shape (n_features,)
 |          The impurity-based feature importances.
 |          The higher, the more important the feature.
 |          The importance of a feature is computed as the (normalized)
 |          total reduction of the criterion brought by that feature.  It is also
 |          known as the Gini importance [4]_.
 |
 |          Warning: impurity-based feature importances can be misleading for
 |          high cardinality features (many unique values). See
 |          :func:`sklearn.inspection.permutation_importance` as an alternative.
 |
 |      max_features_ : int
 |          The inferred value of max_features.
 |
 |      n_classes_ : int or list of int
 |          The number of classes (for single output problems),
 |          or a list containing the number of classes for each
 |          output (for multi-output problems).
 |
 |      n_features_ : int
 |          The number of features when ``fit`` is performed.
 |
 |      n_outputs_ : int
 |          The number of outputs when ``fit`` is performed.
 |
 |      tree_ : Tree instance
 |          The underlying Tree object. Please refer to
 |          ``help(sklearn.tree._tree.Tree)`` for attributes of Tree object and
 |          :ref:`sphx_glr_auto_examples_tree_plot_unveil_tree_structure.py`
 |          for basic usage of these attributes.
 |
 |      See Also
 |      --------
 |      DecisionTreeRegressor : A decision tree regressor.
 |
 |      Notes
 |      -----
 |      The default values for the parameters controlling the size of the trees
 |      (e.g. ``max_depth``, ``min_samples_leaf``, etc.) lead to fully grown and
 |      unpruned trees which can potentially be very large on some data sets. To
 |      reduce memory consumption, the complexity and size of the trees should be
 |      controlled by setting those parameter values.
 |
 |      The :meth:`predict` method operates using the :func:`numpy.argmax`
 |      function on the outputs of :meth:`predict_proba`. This means that in
 |      case the highest predicted probabilities are tied, the classifier will
 |      predict the tied class with the lowest index in :term:`classes_`.
 |
 |      References
 |      ----------
 |
 |      .. [1] https://en.wikipedia.org/wiki/Decision_tree_learning
 |
 |      .. [2] L. Breiman, J. Friedman, R. Olshen, and C. Stone, "Classification
 |             and Regression Trees", Wadsworth, Belmont, CA, 1984.
 |
 |      .. [3] T. Hastie, R. Tibshirani and J. Friedman. "Elements of Statistical
 |             Learning", Springer, 2009.
 |
 |      .. [4] L. Breiman, and A. Cutler, "Random Forests",
 |             https://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm
 |
 |      Examples
 |      --------
 |      >>> from sklearn.datasets import load_iris
 |      >>> from sklearn.model_selection import cross_val_score
 |      >>> from sklearn.tree import DecisionTreeClassifier
```

```
 |  >>> clf = DecisionTreeClassifier(random_state=0)
 |  >>> iris = load_iris()
 |  >>> cross_val_score(clf, iris.data, iris.target, cv=10)
 |  ...                             # doctest: +SKIP
 |  ...
 |  array([ 1.     ,  0.93...,  0.86...,  0.93...,  0.93...,
 |          0.93...,  0.93...,  1.     ,  0.93...,  1.      ])
 |
 |  Method resolution order:
 |      DecisionTreeClassifier
 |      sklearn.base.ClassifierMixin
 |      BaseDecisionTree
 |      sklearn.base.MultiOutputMixin
 |      sklearn.base.BaseEstimator
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(self, *, criterion='gini', splitter='best', max_depth=None, min_samples
_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features=None, rando
m_state=None, max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=Non
e, class_weight=None, ccp_alpha=0.0)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  fit(self, X, y, sample_weight=None, check_input=True, X_idx_sorted='deprecated')
 |      Build a decision tree classifier from the training set (X, y).
 |
 |      Parameters
 |      ----------
 |      X : {array-like, sparse matrix} of shape (n_samples, n_features)
 |          The training input samples. Internally, it will be converted to
 |          ``dtype=np.float32`` and if a sparse matrix is provided
 |          to a sparse ``csc_matrix``.
 |
 |      y : array-like of shape (n_samples,) or (n_samples, n_outputs)
 |          The target values (class labels) as integers or strings.
 |
 |      sample_weight : array-like of shape (n_samples,), default=None
 |          Sample weights. If None, then samples are equally weighted. Splits
 |          that would create child nodes with net zero or negative weight are
 |          ignored while searching for a split in each node. Splits are also
 |          ignored if they would result in any single class carrying a
 |          negative weight in either child node.
 |
 |      check_input : bool, default=True
 |          Allow to bypass several input checking.
 |          Don't use this parameter unless you know what you do.
 |
 |      X_idx_sorted : deprecated, default="deprecated"
 |          This parameter is deprecated and has no effect.
 |          It will be removed in 1.1 (renaming of 0.26).
 |
 |          .. deprecated :: 0.24
 |
 |      Returns
 |      -------
 |      self : DecisionTreeClassifier
 |          Fitted estimator.
 |
 |  predict_log_proba(self, X)
 |      Predict class log-probabilities of the input samples X.
 |
 |      Parameters
 |      ----------
 |      X : {array-like, sparse matrix} of shape (n_samples, n_features)
 |          The input samples. Internally, it will be converted to
 |          ``dtype=np.float32`` and if a sparse matrix is provided
 |          to a sparse ``csr_matrix``.
 |
```

```
 |      Returns
 |      -------
 |      proba : ndarray of shape (n_samples, n_classes) or list of n_outputs 
such arrays if n_outputs > 1
 |          The class log-probabilities of the input samples. The order of the
 |          classes corresponds to that in the attribute :term:`classes_`.
 |  
 |  predict_proba(self, X, check_input=True)
 |      Predict class probabilities of the input samples X.
 |      
 |      The predicted class probability is the fraction of samples of the same
 |      class in a leaf.
 |      
 |      Parameters
 |      ----------
 |      X : {array-like, sparse matrix} of shape (n_samples, n_features)
 |          The input samples. Internally, it will be converted to
 |          ``dtype=np.float32`` and if a sparse matrix is provided
 |          to a sparse ``csr_matrix``.
 |      
 |      check_input : bool, default=True
 |          Allow to bypass several input checking.
 |          Don't use this parameter unless you know what you do.
 |      
 |      Returns
 |      -------
 |      proba : ndarray of shape (n_samples, n_classes) or list of n_outputs 
such arrays if n_outputs > 1
 |          The class probabilities of the input samples. The order of the
 |          classes corresponds to that in the attribute :term:`classes_`.
 |  
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |  
 |  __abstractmethods__ = frozenset()
 |  
 |  ----------------------------------------------------------------------
 |  Methods inherited from sklearn.base.ClassifierMixin:
 |  
 |  score(self, X, y, sample_weight=None)
 |      Return the mean accuracy on the given test data and labels.
 |      
 |      In multi-label classification, this is the subset accuracy
 |      which is a harsh metric since you require for each sample that
 |      each label set be correctly predicted.
 |      
 |      Parameters
 |      ----------
 |      X : array-like of shape (n_samples, n_features)
 |          Test samples.
 |      
 |      y : array-like of shape (n_samples,) or (n_samples, n_outputs)
 |          True labels for `X`.
 |      
 |      sample_weight : array-like of shape (n_samples,), default=None
 |          Sample weights.
 |      
 |      Returns
 |      -------
 |      score : float
 |          Mean accuracy of ``self.predict(X)`` wrt. `y`.
 |  
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from sklearn.base.ClassifierMixin:
 |  
 |  __dict__
 |      dictionary for instance variables (if defined)
 |  
 |  __weakref__
```

```
 |      list of weak references to the object (if defined)
 |  
 |  ----------------------------------------------------------------------
 |  Methods inherited from BaseDecisionTree:
 |  
 |  apply(self, X, check_input=True)
 |      Return the index of the leaf that each sample is predicted as.
 |      
 |      .. versionadded:: 0.17
 |      
 |      Parameters
 |      ----------
 |      X : {array-like, sparse matrix} of shape (n_samples, n_features)
 |          The input samples. Internally, it will be converted to
 |          ``dtype=np.float32`` and if a sparse matrix is provided
 |          to a sparse ``csr_matrix``.
 |      
 |      check_input : bool, default=True
 |          Allow to bypass several input checking.
 |          Don't use this parameter unless you know what you do.
 |      
 |      Returns
 |      -------
 |      X_leaves : array-like of shape (n_samples,)
 |          For each datapoint x in X, return the index of the leaf x
 |          ends up in. Leaves are numbered within
 |          ``[0; self.tree_.node_count)``, possibly with gaps in the
 |          numbering.
 |  
 |  cost_complexity_pruning_path(self, X, y, sample_weight=None)
 |      Compute the pruning path during Minimal Cost-Complexity Pruning.
 |      
 |      See :ref:`minimal_cost_complexity_pruning` for details on the pruning
 |      process.
 |      
 |      Parameters
 |      ----------
 |      X : {array-like, sparse matrix} of shape (n_samples, n_features)
 |          The training input samples. Internally, it will be converted to
 |          ``dtype=np.float32`` and if a sparse matrix is provided
 |          to a sparse ``csc_matrix``.
 |      
 |      y : array-like of shape (n_samples,) or (n_samples, n_outputs)
 |          The target values (class labels) as integers or strings.
 |      
 |      sample_weight : array-like of shape (n_samples,), default=None
 |          Sample weights. If None, then samples are equally weighted. Splits
 |          that would create child nodes with net zero or negative weight are
 |          ignored while searching for a split in each node. Splits are also
 |          ignored if they would result in any single class carrying a
 |          negative weight in either child node.
 |      
 |      Returns
 |      -------
 |      ccp_path : :class:`~sklearn.utils.Bunch`
 |          Dictionary-like object, with the following attributes.
 |      
 |          ccp_alphas : ndarray
 |              Effective alphas of subtree during pruning.
 |      
 |          impurities : ndarray
 |              Sum of the impurities of the subtree leaves for the
 |              corresponding alpha value in ``ccp_alphas``.
 |  
 |  decision_path(self, X, check_input=True)
 |      Return the decision path in the tree.
 |      
 |      .. versionadded:: 0.18
 |      
```

```
 |      Parameters
 |      ----------
 |      X : {array-like, sparse matrix} of shape (n_samples, n_features)
 |          The input samples. Internally, it will be converted to
 |          ``dtype=np.float32`` and if a sparse matrix is provided
 |          to a sparse ``csr_matrix``.
 |
 |      check_input : bool, default=True
 |          Allow to bypass several input checking.
 |          Don't use this parameter unless you know what you do.
 |
 |      Returns
 |      -------
 |      indicator : sparse matrix of shape (n_samples, n_nodes)
 |          Return a node indicator CSR matrix where non zero elements
 |          indicates that the samples goes through the nodes.
 |
 |  get_depth(self)
 |      Return the depth of the decision tree.
 |
 |      The depth of a tree is the maximum distance between the root
 |      and any leaf.
 |
 |      Returns
 |      -------
 |      self.tree_.max_depth : int
 |          The maximum depth of the tree.
 |
 |  get_n_leaves(self)
 |      Return the number of leaves of the decision tree.
 |
 |      Returns
 |      -------
 |      self.tree_.n_leaves : int
 |          Number of leaves.
 |
 |  predict(self, X, check_input=True)
 |      Predict class or regression value for X.
 |
 |      For a classification model, the predicted class for each sample in X is
 |      returned. For a regression model, the predicted value based on X is
 |      returned.
 |
 |      Parameters
 |      ----------
 |      X : {array-like, sparse matrix} of shape (n_samples, n_features)
 |          The input samples. Internally, it will be converted to
 |          ``dtype=np.float32`` and if a sparse matrix is provided
 |          to a sparse ``csr_matrix``.
 |
 |      check_input : bool, default=True
 |          Allow to bypass several input checking.
 |          Don't use this parameter unless you know what you do.
 |
 |      Returns
 |      -------
 |      y : array-like of shape (n_samples,) or (n_samples, n_outputs)
 |          The predicted classes, or the predict values.
 |
 |  ----------------------------------------------------------------------
 |  Readonly properties inherited from BaseDecisionTree:
 |
 |  feature_importances_
 |      Return the feature importances.
 |
 |      The importance of a feature is computed as the (normalized) total
 |      reduction of the criterion brought by that feature.
 |      It is also known as the Gini importance.
 |
```

```
 |      Warning: impurity-based feature importances can be misleading for
 |      high cardinality features (many unique values). See
 |      :func:`sklearn.inspection.permutation_importance` as an alternative.
 |      
 |      Returns
 |      -------
 |      feature_importances_ : ndarray of shape (n_features,)
 |          Normalized total reduction of criteria by feature
 |          (Gini importance).
 |  
 |  ----------------------------------------------------------------------
 |  Methods inherited from sklearn.base.BaseEstimator:
 |  
 |  __getstate__(self)
 |  
 |  __repr__(self, N_CHAR_MAX=700)
 |      Return repr(self).
 |  
 |  __setstate__(self, state)
 |  
 |  get_params(self, deep=True)
 |      Get parameters for this estimator.
 |      
 |      Parameters
 |      ----------
 |      deep : bool, default=True
 |          If True, will return the parameters for this estimator and
 |          contained subobjects that are estimators.
 |      
 |      Returns
 |      -------
 |      params : dict
 |          Parameter names mapped to their values.
 |  
 |  set_params(self, **params)
 |      Set the parameters of this estimator.
 |      
 |      The method works on simple estimators as well as on nested objects
 |      (such as :class:`~sklearn.pipeline.Pipeline`). The latter have
 |      parameters of the form ``<component>__<parameter>`` so that it's
 |      possible to update each component of a nested object.
 |      
 |      Parameters
 |      ----------
 |      **params : dict
 |          Estimator parameters.
 |      
 |      Returns
 |      -------
 |      self : estimator instance
 |          Estimator instance.
```

Out[12]: list

In [13]:

```
#Prediction
preds = modelTree.predict(test[predictors])
preds
type(preds)
```

Out[13]: numpy.ndarray

In [14]:

```
pd.Series(preds).value_counts()
141/(141+39)
pd.crosstab(test[target],preds) #64%
temp = pd.Series(modelTree.predict(train[predictors])).reset_index(drop=True)
```

```
np.mean(pd.Series(train.TaxInc).reset_index(drop=True)==pd.Series(modelTree.predict(
np.mean(preds==test.TaxInc)
```

Out[14]: 0.6222222222222222

In [15]:

```
from sklearn.ensemble import RandomForestClassifier
rf = RandomForestClassifier(n_jobs = 3, oob_score = True, n_estimators = 15, criteri
```

In [16]:

```
np.shape(FraudC_final) # 600,100 => Shape
len(Y)
len(X)
```

Out[16]: 600

In [17]:

```
FraudC_final.describe()
```

Out[17]:

	City.Population	Work.Experience	Undergrad_NO	Undergrad_YES	Marital.Status_Divorced	M
count	600.000000	600.000000	600.000000	600.000000	600.000000	
mean	108747.368333	15.558333	0.480000	0.520000	0.315000	
std	49850.075134	8.842147	0.500017	0.500017	0.464903	
min	25779.000000	0.000000	0.000000	0.000000	0.000000	
25%	66966.750000	8.000000	0.000000	0.000000	0.000000	
50%	106493.500000	15.000000	0.000000	1.000000	0.000000	
75%	150114.250000	24.000000	1.000000	1.000000	1.000000	
max	199778.000000	30.000000	1.000000	1.000000	1.000000	

In [18]:

```
FraudC_final.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 600 entries, 0 to 599
Data columns (total 10 columns):
 #   Column                   Non-Null Count  Dtype   
---  ------                   --------------  -----   
 0   City.Population          600 non-null    int64   
 1   Work.Experience          600 non-null    int64   
 2   Undergrad_NO             600 non-null    uint8   
 3   Undergrad_YES            600 non-null    uint8   
 4   Marital.Status_Divorced  600 non-null    uint8   
 5   Marital.Status_Married   600 non-null    uint8   
 6   Marital.Status_Single    600 non-null    uint8   
 7   Urban_NO                 600 non-null    uint8   
 8   Urban_YES                600 non-null    uint8   
 9   TaxInc                   600 non-null    category
dtypes: category(1), int64(2), uint8(7)
memory usage: 14.3 KB
```

In [19]:

```
type([X])
type([Y])
Y1 = pd.DataFrame(Y)
type(Y1)
```

Out[19]: pandas.core.frame.DataFrame

In [20]:
```
rf.fit(X,Y1)
rf.estimators_
rf.classes_
rf.n_classes_
rf.n_features_
rf.n_outputs_
rf.oob_score_
rf.predict(X)
```

```
<ipython-input-20-7708b846b6bb>:1: DataConversionWarning: A column-vector y was pass
ed when a 1d array was expected. Please change the shape of y to (n_samples,), for e
xample using ravel().
  rf.fit(X,Y1)
```

Out[20]:
```
array(['Good', 'Good', 'Good', 'Good', 'Good', 'Good', 'Good', 'Good',
       'Good', 'Good', 'Risky', 'Good', 'Risky', 'Good', 'Good', 'Good',
       'Risky', 'Good', 'Good', 'Risky', 'Good', 'Risky', 'Good', 'Good',
       'Good', 'Good', 'Good', 'Good', 'Risky', 'Good', 'Good', 'Good',
       'Good', 'Good', 'Risky', 'Good', 'Risky', 'Good', 'Risky', 'Good',
       'Good', 'Good', 'Good', 'Good', 'Good', 'Good', 'Good', 'Good',
       'Risky', 'Good', 'Good', 'Good', 'Good', 'Risky', 'Good', 'Risky',
       'Good', 'Good', 'Risky', 'Good', 'Good', 'Risky', 'Good', 'Good',
       'Good', 'Good', 'Good', 'Good', 'Risky', 'Good', 'Good', 'Good',
       'Good', 'Good', 'Good', 'Good', 'Good', 'Good', 'Good', 'Good',
       'Good', 'Good', 'Good', 'Good', 'Good', 'Risky', 'Good', 'Risky',
       'Good', 'Risky', 'Good', 'Good', 'Good', 'Risky', 'Risky', 'Good',
       'Risky', 'Good', 'Risky', 'Good', 'Good', 'Good', 'Good', 'Good',
       'Good', 'Good', 'Risky', 'Good', 'Good', 'Good', 'Risky', 'Good',
       'Good', 'Good', 'Risky', 'Good', 'Good', 'Good', 'Risky', 'Good',
       'Good', 'Risky', 'Risky', 'Risky', 'Good', 'Good', 'Risky', 'Good',
       'Good', 'Good', 'Good', 'Good', 'Good', 'Good', 'Good', 'Risky',
       'Good', 'Risky', 'Good', 'Good', 'Risky', 'Good', 'Good', 'Good',
       'Good', 'Good', 'Good', 'Good', 'Good', 'Good', 'Good', 'Good',
       'Risky', 'Risky', 'Risky', 'Good', 'Good', 'Good', 'Risky', 'Good',
       'Good', 'Good', 'Good', 'Risky', 'Good', 'Good', 'Good', 'Good',
       'Risky', 'Good', 'Good', 'Risky', 'Risky', 'Good', 'Good', 'Good',
       'Good', 'Good', 'Good', 'Risky', 'Good', 'Risky', 'Risky', 'Good',
       'Good', 'Good', 'Good', 'Good', 'Risky', 'Good', 'Good', 'Good',
       'Good', 'Good', 'Risky', 'Good', 'Risky', 'Good', 'Good', 'Risky',
       'Good', 'Good', 'Good', 'Good', 'Risky', 'Risky', 'Good', 'Risky',
       'Good', 'Good', 'Good', 'Good', 'Good', 'Good', 'Good', 'Good',
       'Good', 'Good', 'Good', 'Risky', 'Good', 'Good', 'Good', 'Good',
       'Good', 'Good', 'Good', 'Good', 'Risky', 'Good', 'Risky', 'Good',
       'Risky', 'Risky', 'Good', 'Good', 'Good', 'Risky', 'Good', 'Good',
       'Good', 'Risky', 'Good', 'Good', 'Good', 'Good', 'Good', 'Good',
       'Risky', 'Good', 'Good', 'Good', 'Good', 'Good', 'Good', 'Good',
       'Good', 'Good', 'Good', 'Risky', 'Good', 'Good', 'Good', 'Good',
       'Good', 'Risky', 'Good', 'Risky', 'Good', 'Good', 'Good', 'Good',
       'Good', 'Good', 'Good', 'Good', 'Good', 'Good', 'Good', 'Good',
       'Risky', 'Good', 'Good', 'Good', 'Risky', 'Good', 'Good', 'Good',
       'Risky', 'Good', 'Good', 'Good', 'Risky', 'Good', 'Good', 'Good',
       'Good', 'Good', 'Good', 'Good', 'Good', 'Risky', 'Good', 'Good',
       'Good', 'Good', 'Good', 'Good', 'Good', 'Good', 'Good', 'Good',
       'Good', 'Good', 'Good', 'Risky', 'Good', 'Good', 'Good', 'Good',
       'Good', 'Good', 'Good', 'Good', 'Good', 'Good', 'Risky', 'Risky',
       'Good', 'Risky', 'Good', 'Risky', 'Good', 'Risky', 'Good', 'Good',
       'Good', 'Risky', 'Risky', 'Risky', 'Good', 'Good', 'Good', 'Good',
       'Good', 'Good', 'Risky', 'Risky', 'Good', 'Risky', 'Good', 'Good',
       'Risky', 'Good', 'Risky', 'Good', 'Risky', 'Good', 'Good', 'Good',
       'Good', 'Good', 'Risky', 'Risky', 'Good', 'Good', 'Good', 'Risky',
       'Good', 'Good', 'Good', 'Good', 'Good', 'Good', 'Good', 'Good',
       'Good', 'Good', 'Good', 'Risky', 'Good', 'Good', 'Good', 'Good',
       'Good', 'Good', 'Good', 'Good', 'Good', 'Good', 'Good', 'Good',
       'Risky', 'Good', 'Good', 'Good', 'Good', 'Good', 'Risky', 'Good',
       'Good', 'Good', 'Good', 'Good', 'Good', 'Good', 'Good', 'Good',
       'Good', 'Good', 'Good', 'Good', 'Good', 'Good', 'Good', 'Good',
```

```
       'Good', 'Good', 'Good', 'Good', 'Good', 'Good', 'Good', 'Good',
       'Good', 'Good', 'Good', 'Good', 'Risky', 'Good', 'Good', 'Good',
       'Good', 'Risky', 'Risky', 'Good', 'Good', 'Risky', 'Risky',
       'Risky', 'Good', 'Good', 'Good', 'Risky', 'Good', 'Good', 'Risky',
       'Good', 'Good', 'Good', 'Good', 'Good', 'Good', 'Good', 'Good',
       'Good', 'Good', 'Risky', 'Risky', 'Risky', 'Good', 'Good', 'Good',
       'Good', 'Good', 'Risky', 'Good', 'Good', 'Good', 'Good', 'Good',
       'Risky', 'Good', 'Good', 'Good', 'Good', 'Good', 'Good', 'Good',
       'Good', 'Good', 'Good', 'Good', 'Good', 'Good', 'Good', 'Good',
       'Good', 'Good', 'Good', 'Good', 'Risky', 'Good', 'Risky', 'Good',
       'Good', 'Good', 'Good', 'Good', 'Good', 'Good', 'Good', 'Good',
       'Good', 'Good', 'Good', 'Risky', 'Good', 'Good', 'Risky', 'Good',
       'Good', 'Good', 'Good', 'Good', 'Good', 'Good', 'Risky', 'Good',
       'Good', 'Good', 'Good', 'Good', 'Good', 'Risky', 'Good', 'Risky',
       'Good', 'Good', 'Good', 'Good', 'Good', 'Good', 'Risky', 'Good',
       'Good', 'Good', 'Good', 'Good', 'Good', 'Risky', 'Risky', 'Good',
       'Good', 'Risky', 'Good', 'Good', 'Good', 'Good', 'Good', 'Good',
       'Good', 'Good', 'Good', 'Good', 'Good', 'Good', 'Risky', 'Good',
       'Good', 'Good', 'Good', 'Good', 'Good', 'Good', 'Good', 'Good',
       'Risky', 'Good', 'Good', 'Good', 'Good', 'Good', 'Good', 'Good',
       'Risky', 'Good', 'Good', 'Good', 'Good', 'Good', 'Good', 'Good',
       'Good', 'Good', 'Good', 'Good', 'Good', 'Good', 'Good', 'Good',
       'Risky', 'Good', 'Good', 'Good', 'Good', 'Good', 'Good', 'Good',
       'Good'], dtype=object)
```

In [21]:

```python
FraudC_final['rf_pred'] = rf.predict(X)
cols = ['rf_pred','TaxInc']
FraudC_final[cols].head()
FraudC_final["TaxInc"]

from sklearn.metrics import confusion_matrix

confusion_matrix(FraudC_final['TaxInc'],FraudC_final['rf_pred']) # Confusion matrix

pd.crosstab(FraudC_final['TaxInc'],FraudC_final['rf_pred'])



print("Accuracy",(476+115)/(476+115+9+0)*100)
#98.5%

FraudC_final["rf_pred"]
```

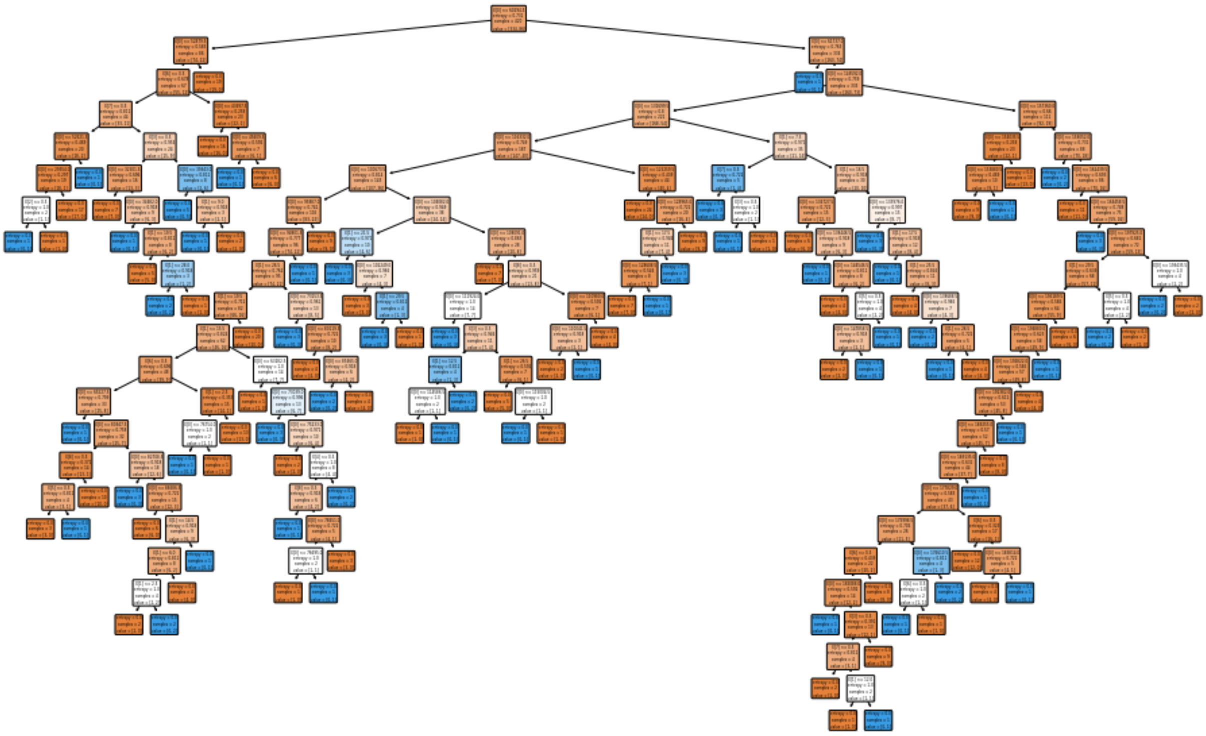
```
Accuracy 98.5
```

Out[21]:

```
0      Good
1      Good
2      Good
3      Good
4      Good
       ...
595    Good
596    Good
597    Good
598    Good
599    Good
Name: rf_pred, Length: 600, dtype: object
```

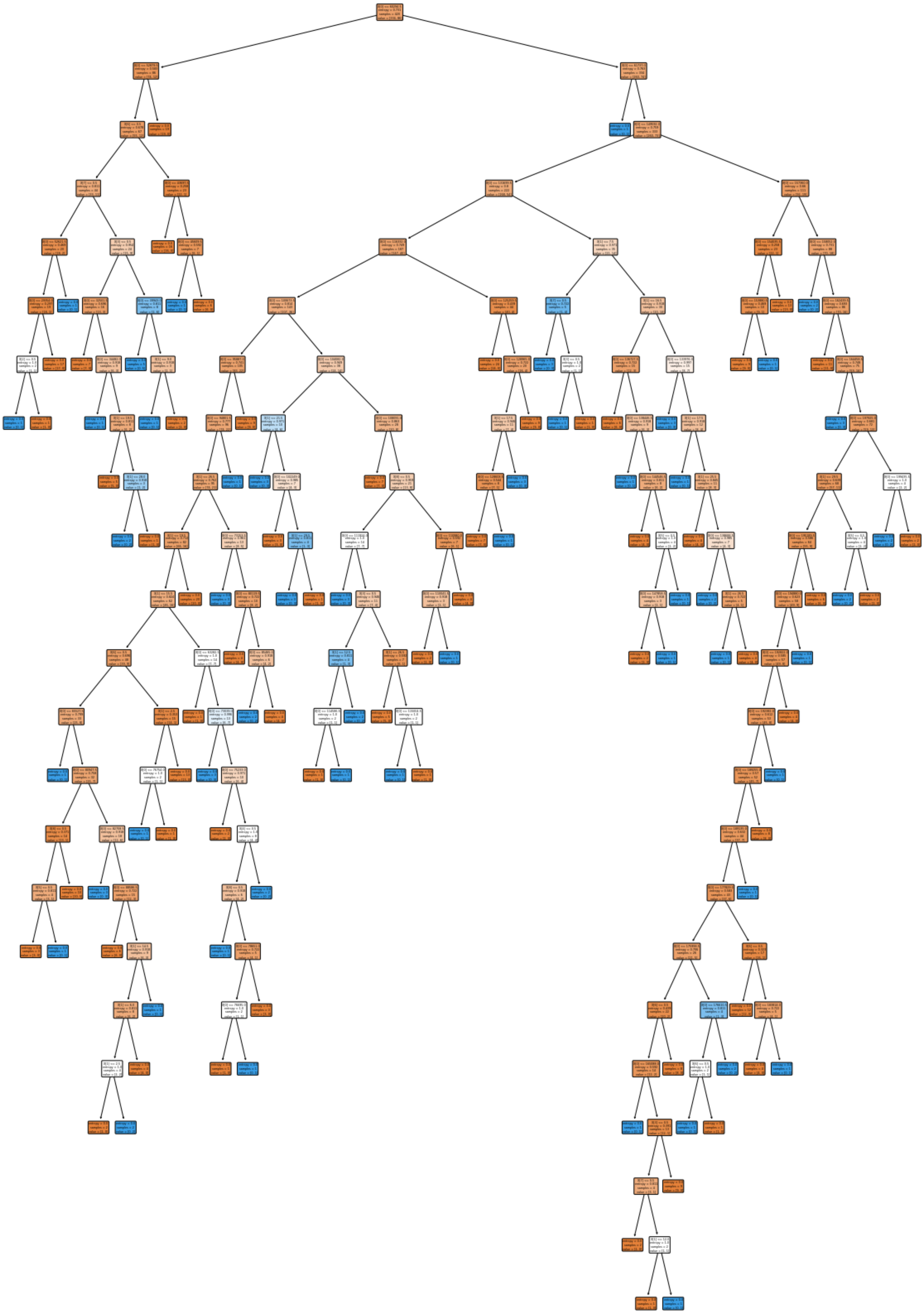
In [22]:

```python
# Prepare a plot figure with set size.
from sklearn.tree import plot_tree
from matplotlib import pyplot as plt
plt.figure(figsize = (16,10))
# Plot the decision tree.
plot_tree(modelTree,rounded = True,filled = True)# Display the tree plot figure.
plt.show()
```



In [28]:

```
plt.figure(figsize = (20,30))
# Plot the decision tree.
plot_tree(modelTree,rounded = True,filled = True)# Display the tree plot figure.
plt.show()
```



In [ ]: