Data Mining
Assignment 2

Ali Gholami

Department of Computer Engineering & Information Technology
Amirkabir University of Technology

http://ceit.aut.ac.ir/~aligholamee
aligholamee@aut.ac.ir

Abstract

A tree has many analogies in real life, and turns out that it has influenced a wide area of machine learning, covering both classification and regression. In decision analysis, a decision tree can be used to visually and explicitly represent decisions and decision making. As the name goes, it uses a tree-like model of decisions. Though a commonly used tool in data mining for deriving a strategy to reach a particular goal, its also widely used in machine learning, which will be the main focus of this article.

Keywords. *Decision Tree, Normalization, Generalization, Preprocessing, Feature Engineering, Scikit-Learn, Pandas, Numpy, Python 3.5.*

1 Data Preprocessing

In this section, we'll be looking at our training data from different aspects. First, we need to get a quick intuition of how data looks like, how is that distributed and what to do with that! To do this, we'll be using some functions as described below.

```
separate_output('Training Data Types')
print(train_data.dtypes)
```

In this part, we have printed the data types of our training set. Note that *separate-output* is a self-defined function to make thing more clear in the terminal. Now, its time for some statistics. To get a full understanding of how our numerical data is distributed, we use the following code.

```
separate_output('Statistical Information')
print(train_data.describe())
```

The result of this part of code will be some statistical parameters such as: *variance, mean, max, min, counts*. These can be useful in the future to make decisions about *data normalization*. Another amazing feature that *Pandas* has provided for us is the ability to separately describe each column in the dataset. As an example, the first column contains 686 missing values. Use the following code to believe this fact.

```
separate_output('Counts Values on a Column')
print(train_data['col_1'].value_counts())
```

This column may not be seem much useful at the first glance, but we keep it since there are some good values for that column which might make it useful while we go further in the classification task. Some of these columns are completely useless. Let's find them. The following function will return a dictionary consisting of number of missing values of each column.

```
def compute_nans(df):

nans_dict = {}

for col in df:
        nan_col_counter = 0
        for row in df[col]:
                if(row == '?'):
                        nan_col_counter += 1
        nans_dict[str(col)] = [nan_col_counter]

return nans_dict
```

We can obviously remove the following columns with more than 500 missing values.

```
cols_to_drop = [key for key, value in nan_cols.items() if value > 500]
train_data = train_data.drop(cols_to_drop, axis=1)
```

Now its time to look for some *correlation* between the features. We try to remove as much as correlated features as we can. There is a good reason for that. If two numerical features are perfectly correlated, then one doesn't add any additional information (it is determined by the other). So if the number of features is too high (relative to the training sample size), then it is beneficial to reduce the number of features. It's also important to mention that machine learning algorithms are very computationally intensive, and reducing the the features to independent components (or at least principal components) can greatly reduce the amount of resources required. Before implementing a dimensionality reduction approach on our data, let's make sure that the data is in the numeric form. But, before that, it is better to fill in the missing values with proper values.

Here is the number of missing values for each column left in the training set.

```
{
        'col_12': 217,
        'col_2': 0,
        'col_3': 70,
        'col_32': 0,
        'col_33': 0,
        'col_34': 0,
        'col_35': 0,
        'col_37': 0,
        'col_39': 0,
        'col_4': 0,
        'col_5': 0,
        'col_7': 271,
        'col_8': 282,
        'col_9': 0
}
```

There is only one numeric feature which is column 8 that its missing values can be filled using the mean of itself (column). To obtain this, we can use the following function to fill the missing values of an specific column.

```
train_data = train_data.replace('?', np.NaN)
train_data.col_8 = train_data.col_8.astype(float)
train_data['col_8'].fillna(train_data['col_8'].mean(), inplace=True)
```

The first line simply fixes the non-standard missing values given by dear T.A.s (just kidding bro :)). Then we change the type of column 8 to the float since mean function does not work for the *int* types. Then we use the *fillna* class member to fill the *NaN* values with the average of that column. Note that before continuing we shall extract the labels from our training data. We obtain this using the following code.

```
separate_output('Separated Training Labels')
train_labels = train_data['col_39']
train_data = train_data.drop(['col_39'], axis=1)
print(train_labels)
```

Now the time for converting the categorical data to numeric form has come. We use the following function to 1. Convert the object types to categorical types 2. Convert all categorical types to numeric format.

```
def obj_to_num(df):

for column in df.columns:
        if(df[column].dtype == 'object'):
```

```
                df[column] = df[column].astype('category')
                df[column] = df[column].cat.codes

    return df
```

Now we perform a handy task called *Standardization*. We can use either *Standard Scaler* or *Normalizer* to bring a unit *L1* norm to the dataframe *columns* or *rows* respectively. In this case, we are going to use the *Standard Scaler*.

```
    train_data = StandardScaler().fit_transform(train_data)
```

This snippet will return a *numpy* array of train data. Now we perform a *Principal Component Analysis* to extract the best features depicting our dataset.

```
    pca = PCA(n_components=5)
    pca.fit(train_data)
    train_data = pca.transform(train_data)
    test_data = pca.transform(test_data)
```

Now enough for the *preprocessing* phase. In the next phase, I'll be digging through the *Classification* using a *Decision Tree* classifier.

2 Classification

2.1 Decision Tree Classifier

In this section, we first apply a simple decision tree on our training data.

```
    decision_tree = DecisionTreeClassifier()
    decision_tree.fit(train_data, train_labels)
```

Of course, this is the simplest decision tree we can obtain right now. Nonetheless, we'll export the learned decision tree to watch it little bit! In the figure 2.1, we have provided a this decision tree. So, the time for evaluation has arrived. Let's evaluate the trained model on the training set first, then on the test set. We then perform various tweaks with the help of *gridsearch* of *Scikit* library. Note that the test set does not contain labels for the data, thus we are urged to use the *train-test-split* function in order to break down the training set into 2 parts.

```
    train_data, test_data, train_labels, test_labels =
    train_test_split(train_data, train_labels, test_size=0.2)
```

2.2 Model Evaluation

We fall into the evaluation section. Don't get nervous, we'll get back and try *k-fold* cross validation also to improve the test results. Here is the demonstration of how we implemented the evaluation metrics.

```
prediction = decision_tree.predict(test_data)
precision, recall, fscore, support = score(test_labels, prediction)
classes = []
[classes.append(x) for x in test_labels if x not in classes]
for i in range(0, len(classes)):
print('\nClass ', classes[i])
print('     precision = {prec:4.2f}\%'.format(prec=precision[i]*100))
print('     recall = {rc:4.2f}\%'.format(rc=recall[i]*100))
print('     fscore = {fsc:4.2f}\%'.format(fsc=fscore[i]*100))
```

We first predict the labels for our test split of data. Then we calculate the scores using *Sklearn.metrics*. We'll need to find the unique values in the test labels to display them in a beautiful way. We are also planned to work with *search-grid*. The evaluation metrics for the initial decision tree is given in the table 2.1.

Metric/Class	**'1'**	**'2'**	**'3'**	**'4'**	**'5'**	**'U'**
Precision	100%	90%	100%	NaN	100%	99%
Recall	83%	100%	100%	NaN	100%	99%
F-Score	90%	94%	100%	NaN	100%	99%
Support	6	9	2	NaN	17	126

Table 2.1: Measurements of the initial decision tree classifier without cross validation.

2.3 K-Fold Cross Validation

As there is never enough data to train your model, removing a part of it for validation poses a problem of underfitting. By reducing the training data, we risk losing important patterns in dataset, which in turn increases error induced by bias. So, what we require is a method that provides ample data for training the model and also leaves ample data for validation. K-Fold cross validation does exactly that. In K-Fold cross validation, the data is divided into k subsets. Now the holdout method is repeated k times, such that each time, one of the k subsets is used as the test set/ validation set and the other k-1 subsets are put together to form a training set. The error estimation is averaged over all k trials to get total effectiveness of our model. As can be seen, every data point gets to be in a validation set exactly once, and gets to be in a training set k-1 times. This significantly reduces bias as we are using most of the data for fitting, and also significantly reduces variance as most of the data is also being used in validation set. Interchanging the training and test sets also adds to the effectiveness of this method. As a general rule and empirical evidence, K = 5 or 10 is generally preferred, but nothing's fixed and it can take any value [1]. K-Fold splits the data to k parts and then for $i = 1..k$ iterations does this: takes all parts except i'th part as the training data, fits the model with them and then predicts labels for i'th part (test data). In each iteration, label of i'th part of data gets predicted. In the end cross-val-predict merges all partially predicted labels and returns them as a whole.

```
decision_tree = DecisionTreeClassifier()
NUM_FOLDS = 5
predicted = cross_val_predict(decision_tree, train_data, train_labels, cv=NUM_FOLDS)
scores = cross_val_score(decision_tree, train_data, train_labels, cv=NUM_FOLDS)
```

The accuracy of prediction is calculated and stored in the *scores* variable. If we output the scores variable we'll get the following.

```
[0.95031056    0.99378882    0.99375    0.99367089    0.96815287]
```

This array displays the accuracy of prediction on each of these 5 folds. This part of the code integrates these values.

```
print("Accuracy: %0.2f (+/— %0.2f)" % (scores.mean(), scores.std() * 2))
```

The mean of the accuracy is 98%.

2.4 Hyper Parameter Tuning

2.4.1 Parameters

The charming section has arrived! Here is given a simple and useful function to generate the list of possible parameters for an estimator [2].

```
print(decision_tree.get_params())
```

The full list of possible hyper parameters is provided here.

- **Class Weight** Weights associated with classes in the following form.

  ```
  {class_label: weight}
  ```

- **Criterion** – The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "entropy" for the information gain.
- **Max_Depth** – 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.
- **Max_Features** – The number of features to consider when looking for the best split. It can obtain values of "auto", "sqrt" and "log2".
- **Max_Leaf_Nodes** – Grow a tree with Max Leaf Nodes in best-first fashion.
- **Min_Impurity_Decrease** – A node will be split if this split induces a decrease of the impurity greater than or equal to this value.
- **Min_Impurity_Split** – Threshold for early stopping in tree growth. A node will split if its impurity is above the threshold, otherwise it is a leaf.

- **Min_Samples_Leaf** – The minimum number of samples required to be at a leaf node.
- **Min_Samples_Split** – The minimum number of samples required to split an internal node.
- **Min_Weight_Fraction_Leaf** – 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.
- **Presort** – Whether to presort the data to speed up the finding of best splits in fitting. For the default settings of a decision tree on large datasets, setting this to true may slow down the training process.
- **Random_State** – The seed used by the random number generator.
- **Splitter** – 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.

2.4.2 Grid Search

It is possible and recommended to search the hyper-parameter space for the best cross validation score. Any parameter provided when constructing an estimator may be optimized in this manner [3]. A search consists of the following:

- **Estimator**
- **Parameter Space**
- **Search Method**
- **Cross Validation Scheme**
- **Score Function**

Let's play with the parameter **max_depth** a little bit. Here is the code playing with the **max_depth** parameter.

```
param_grid = {
        'max_depth': np.arange(3, 10)
}
NUM_FOLDS = 5
decision_tree = GridSearchCV(DecisionTreeClassifier(), param_grid, cv=NUM_FOLDS)
decision_tree.fit(train_data, train_labels)
separate_output('Mean Accuracy of 5 Folds')
```

We can obtain the best results using the following code.

```
separate_output('Best Parameters')
print(decision_tree.best_params_)
```

Which it was 6 in this case. The fitted decision tree with maximum depth of 6 is given in the figure (2.2). Now let's try something more comprehensive. The following dictionary depicts a more comprehensive parameter tuning for the decision tree. Note that maximum depth of 6 yielded the same accuracy of 98% after cross validation and prediction.

```
param_grid = {
        'max_depth': np.arange(3, 10),
        'splitter': ['random', 'best'],
        'max_features': ['log2', 'sqrt', 'auto'],
}
```

The results for these settings is given below.

```
{'max_depth': 9, 'max_features': 'log2', 'splitter': 'best'}
```

The decision tree is depicted in figure (2.3). The mean accuracy of folds reduced to 96% with these settings.

3 Results & Conclusion

3.1 Parameters

3.1.1 Effect of Maximum Depth

The first parameter to tune is max_depth. This indicates how deep the tree can be. The deeper the tree, the more splits it has and it captures more information about the data.

3.1.2 Effect of Minimum Samples Split

min_samples_split represents the minimum number of samples required to split an internal node. This can vary between considering at least one sample at each node to considering all of the samples at each node. When we increase this parameter, the tree becomes more constrained as it has to consider more samples at each node.

3.1.3 Effect of Maximum Features

max_features represents the number of features to consider when looking for the best split. Using less features will speed up the training process. But, it will considerably lowers the accuracy of prediction.

3.2 Prediction of Test Data

The output of my code is provided in the *results.cv* file along with this report.

References

[1] Prashant Gupta, *Cross-Validation in Machine Learning*. Towards Data Science, Jun 5, 2017.

[2] Scikit-Learn, *sklearn.tree.DecisionTreeClassifier*. http://scikit-learn.org.

[3] Scikit-Learn, *Tuning the hyper-parameters of an estimator*. http://scikit-learn.org.

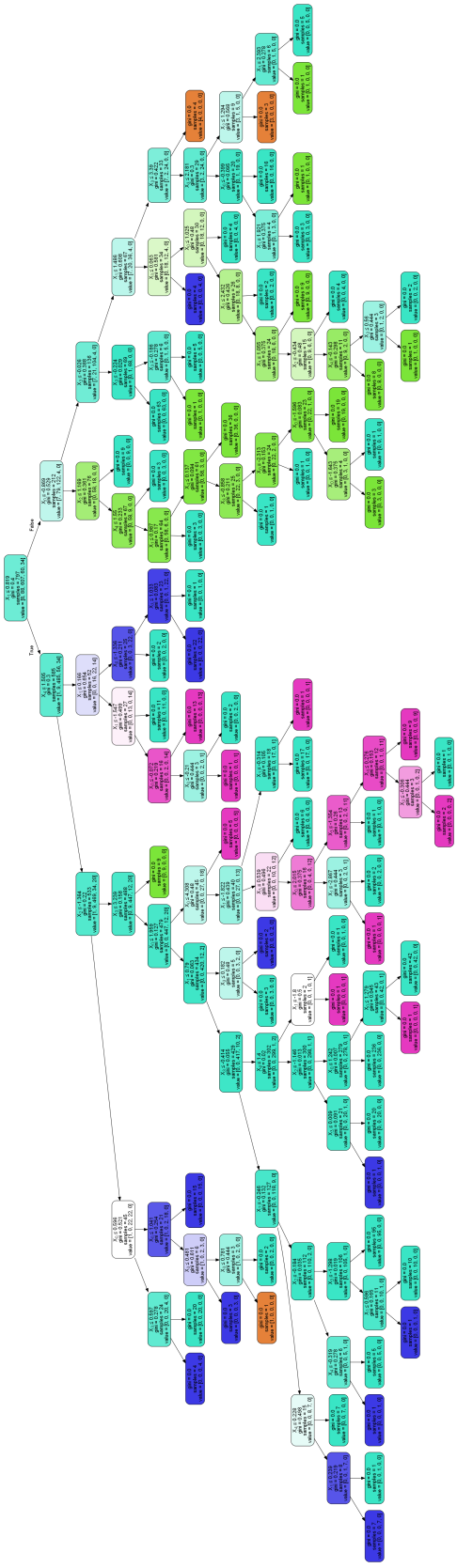

Figure 3.1: The First Lucky Decision Tree Trained on the Annealing Dataset.

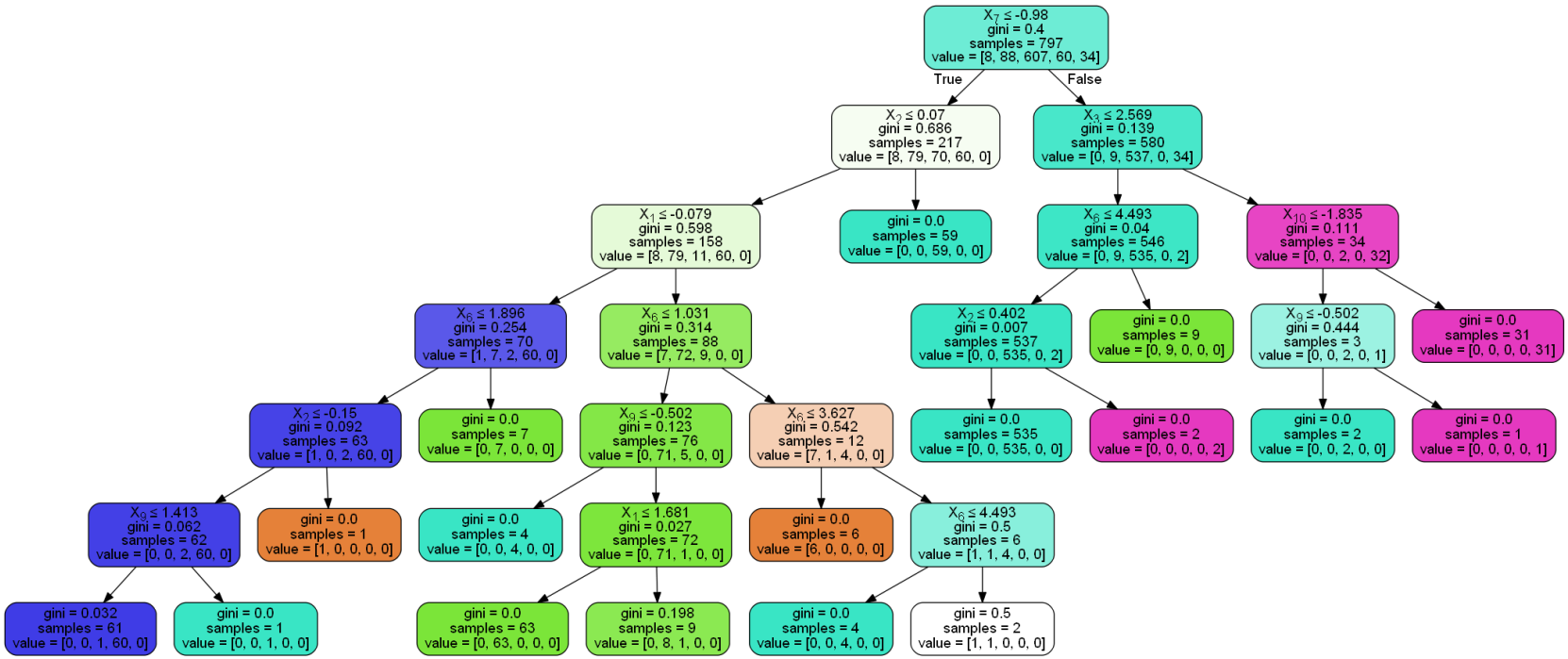

Figure 3.2: Decision Tree Learned with Maximum Depth of 6.

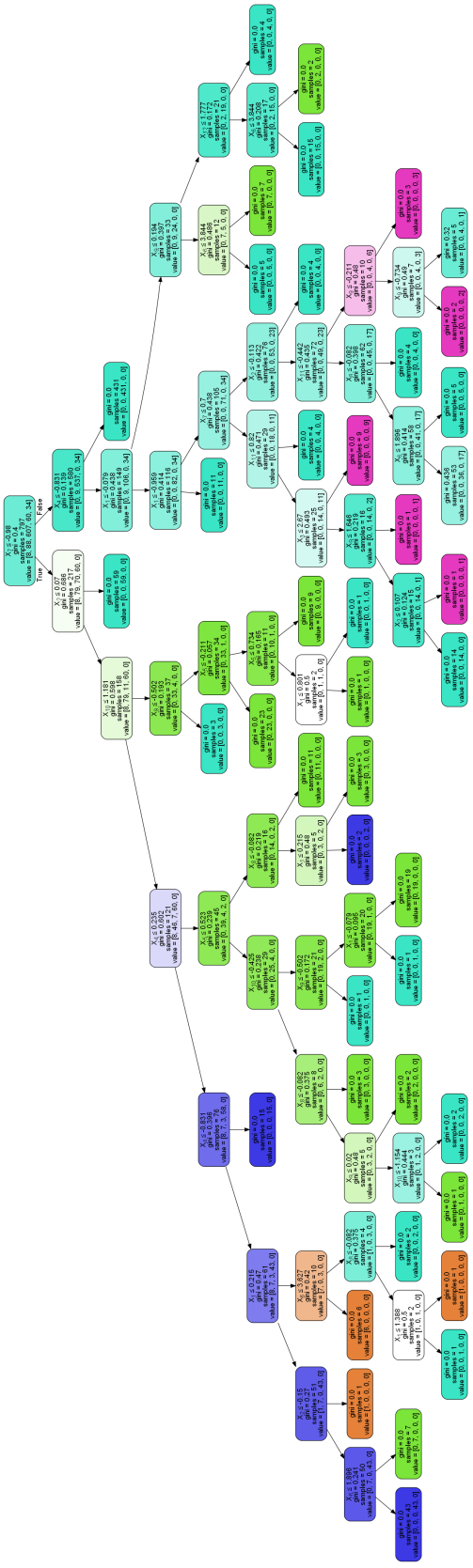

Figure 3.3: Decision Tree Learned with Final Settings.