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

In this assignment, several paramount concepts of *Data Analysis* will be explained. we'll discuss the importance of metrics in the first theoretical problem. A quick review on the *Apriori* algorithm for the *Association Rule Mining* will be explained also. We'll also show how *Weka* can be used for *Association Rule Mining*. Furthermore, The effectiveness of *Normalization* concept is proposed. Finally, an *Statistical* point of view will help us to demonstrate and rationalize the relationship between the *Performance* of the *Learning Algorithm* and the amount of *Data* available. A chief section of this assignment is dedicated to solve the *Titanic* problem, which is a great practice of data mining concepts in production. We'll use *Python* programming language and three main libraries; *Scikit-Learn*, *Pandas* and *Numpy* to tackle this problem. The Python implementation of the Titanic problem is provided on a *Jupyter Notebook* attached with this report.

Keywords. *Apriori, Association Rule Mining, 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

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)
```

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. We are also planned to work with *search-grid*. The evaluation results for each class is given in the following pages.

```
Class  3
precision:  1.0
recall:  1.0
fscore:  1.0
support:  2

Class  5
precision:  1.0
recall:  1.0
fscore:  1.0
support:  17

Class  U
precision:  0.9920634920634921
recall:  0.9920634920634921
fscore:  0.9920634920634921
support:  126

Class  2
precision:  0.9
recall:  1.0
fscore:  0.9473684210526316
support:  9

Class  1
precision:  1.0
recall:  0.8333333333333334
fscore:  0.9090909090909091
support:  6
```

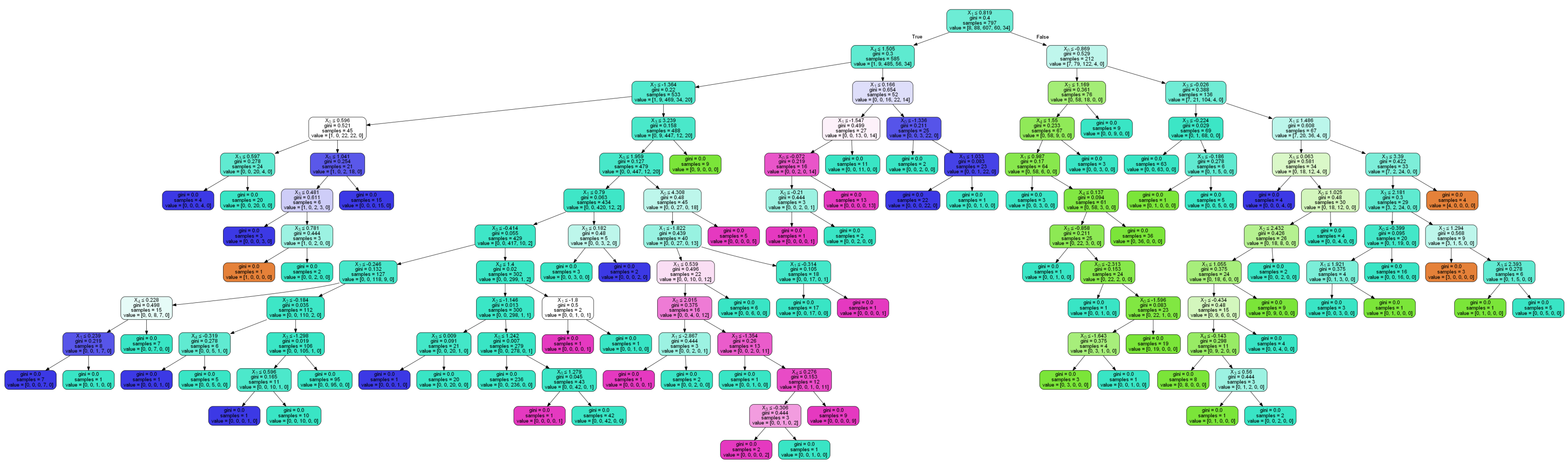

Figure 2.1: The First Lucky Decision Tree Trained on the Annealing Dataset.