



Course name: **Python programming and analytics by Rahul Sir**

Topic name: **random forest classification (Sumit Batch)**

Video name: **random forest classification Sumit batch**

Video length: **53 minutes 58 seconds**

Importing Required Libraries and data

Let's first load the required libraries and the data

```
In [1]: import pandas as pd
```

```
In [2]: data = pd.read_csv("C:/Users/Sahibjot/Desktop/Data_set_telecom_churn.csv")
```

to see some of the core statistics about a particular column, you can use the 'describe' function.

- For numeric columns, describe() returns basic statistics: the value count, mean, standard deviation, minimum, maximum, and 25th, 50th, and 75th quantiles for the data in a column.
- For string columns, describe() returns the value count, the number of unique entries, the most frequently occurring value ('top'), and the number of times the top value occurs ('freq')

Select a column to describe using a string inside the [] braces, and call describe()

Many Data Frames have mixed data types, that is, some columns are numbers, some are strings, and some are dates etc. Internally, CSV files do not contain information on what data types are contained in each column; all of the data is just characters. Pandas infers the data



types when loading the data, e.g. if a column contains only numbers, pandas will set that column's data type to numeric: integer or float.

You can check the types of each column in our example with the `'dtypes'` property of the data frame.

```
In [8]: data.describe()
```

```
Out[8]:
```

	Account Length	VMail Message	Day Mins	Eve Mins	Night Mins	Intl Mins	CustServ Calls	Int'l Plan	VMail Plan	Day Calls	Day Charge	Eve
count	3333.000000	3333.000000	3333.000000	3333.000000	3333.000000	3333.000000	3333.000000	3333.000000	3333.000000	3333.000000	3333.000000	3333.000000
mean	101.064806	8.099010	179.775098	200.980348	200.872037	10.237294	1.562856	0.096910	0.276628	100.435644	30.562307	100.435644
std	39.822106	13.688365	54.467389	50.713844	50.573847	2.791840	1.315491	0.295879	0.447398	20.069084	9.259435	19.999999
min	1.000000	0.000000	0.000000	0.000000	23.200000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	74.000000	0.000000	143.700000	166.600000	167.000000	8.500000	1.000000	0.000000	0.000000	87.000000	24.430000	87.000000
50%	101.000000	0.000000	179.400000	201.400000	201.200000	10.300000	1.000000	0.000000	0.000000	101.000000	30.500000	100.435644
75%	127.000000	20.000000	216.400000	235.300000	235.300000	12.100000	2.000000	0.000000	1.000000	114.000000	36.790000	114.000000
max	243.000000	51.000000	350.800000	363.700000	395.000000	20.000000	9.000000	1.000000	1.000000	165.000000	59.640000	170.000000

```
In [49]: data.dtypes
```

```
Out[49]: Phone                object
Account Length              int64
VMail Message              int64
Day Mins                   float64
Eve Mins                   float64
Night Mins                 float64
Intl Mins                  float64
CustServ Calls             int64
Int'l Plan                 int64
VMail Plan                 int64
Day Calls                  int64
Day Charge                 float64
Eve Calls                  int64
Eve Charge                 float64
Night Calls                int64
Night Charge               float64
Intl Calls                 int64
Intl Charge                float64
State                      object
AreaCode                   int64
```

In the next step, we are going to drop or remove the "phone" column using `.drop()` function because we don't need the phone column in our data, this is not necessary if you want to keep the column, you can keep it. It is just a preprocessing step.

```
In [50]: data1 = data.drop(["Phone"], axis=1)
```

To delete rows and columns from DataFrames, Pandas uses the "[drop](#)" function.

To delete a column, or multiple columns, use the name of the column(s), and specify the "axis" as 1. The drop function returns a new Data Frame, with the columns removed.

Now, as you can see the area code column is denoted with integer data type, but the area code can't be an integer, it must be a character or string as you can't take the average of area code, so we need to change this integer column into string column.

In some cases, the automated inferring of data types can give unexpected results. Note that strings are loaded as 'object' datatypes.

To change the datatype of a specific column, use the [.astype\(\) function](#)

```
In [51]: data1['AreaCode'] = data1.AreaCode.astype(str)
```

In statistics, especially in regression models, we deal with various kind of data. The data may be quantitative (numerical) or qualitative (categorical). The numerical data can be easily handled in regression models but we can't use categorical data directly, it needs to be transformed in some way.



For transforming categorical attribute to numerical attribute, we can use label encoding procedure (label encoding assigns a unique integer to each category of data). But this procedure is not alone that much suitable, hence, **One hot encoding** is used in regression models following label encoding. This enables us to create new attributes according to the number of classes present in the categorical attribute i.e., if there are n number of categories in categorical attribute, n new attributes will be created. These attributes created are called **Dummy Variables**. Hence, dummy variables are "proxy" variables for categorical data in regression models.

These dummy variables will be created with *one hot encoding* and each attribute will have value either 0 or 1, representing presence or absence of that attribute.

```
In [52]: features = pd.get_dummies(data1)
```

```
In [53]: features
```

```
Out[53]:
```

	Account Length	VMail Message	Day Mins	Eve Mins	Night Mins	Intl Mins	CustServ Calls	Int'l Plan	VMail Plan	Day Calls	...	State_UT	State_VA	State_VT	State_WA	State_WI	State_WV	State_...
0	128	25	265.1	197.4	244.7	10.0	1	0	1	110	...	0	0	0	0	0	0	
1	107	26	161.6	195.5	254.4	13.7	1	0	1	123	...	0	0	0	0	0	0	
2	137	0	243.4	121.2	162.6	12.2	0	0	0	114	...	0	0	0	0	0	0	
3	84	0	299.4	61.9	196.9	6.6	2	1	0	71	...	0	0	0	0	0	0	
4	75	0	166.7	148.3	186.9	10.1	3	1	0	113	...	0	0	0	0	0	0	
5	118	0	223.4	220.6	203.9	6.3	0	1	0	98	...	0	0	0	0	0	0	
6	121	24	218.2	348.5	212.6	7.5	3	0	1	88	...	0	0	0	0	0	0	
7	147	0	157.0	103.1	211.8	7.1	0	1	0	79	...	0	0	0	0	0	0	
8	117	0	184.5	351.6	215.8	8.7	1	0	0	97	...	0	0	0	0	0	0	
9	141	37	258.6	222.0	326.4	11.2	0	1	1	84	...	0	0	0	0	0	0	1
10	65	0	129.1	228.5	208.8	12.7	4	0	0	137	...	0	0	0	0	0	0	



You can also use label encoding:

```
In [86]: from sklearn.preprocessing import LabelEncoder  
lb_make = LabelEncoder()  
data1["Area_code"] = lb_make.fit_transform(data1["AreaCode"])  
data1["State_code"] = lb_make.fit_transform(data1["State"])
```

Label Encoding refers to converting the labels into numeric form so as to convert it into the machine-readable form. Machine learning algorithms can then decide in a better way on how those labels must be operated. It is an important pre-processing step for the structured dataset in supervised learning. Label encoding convert the data in machine readable form, but it assigns a unique number(starting from 0) to each class of data.

If you want the column names or the variables in the list format:

```
In [54]: a = features.columns.values.tolist()
```

```
In [55]: print(a)
```

```
['Account Length', 'VMail Message', 'Day Mins', 'Eve Mins', 'Night Mins', 'Intl Mins', 'CustServ Calls', 'Int'l Plan', 'VMail P  
lan', 'Day Calls', 'Day Charge', 'Eve Calls', 'Eve Charge', 'Night Calls', 'Night Charge', 'Intl Calls', 'Intl Charge', 'Chur  
n', 'State_AK', 'State_AL', 'State_AR', 'State_AZ', 'State_CA', 'State_CO', 'State_CT', 'State_DC', 'State_DE', 'State_FL', 'St  
ate_GA', 'State_HI', 'State_IA', 'State_ID', 'State_IL', 'State_IN', 'State_KS', 'State_KY', 'State_LA', 'State_MA', 'State_M  
D', 'State_ME', 'State_MI', 'State_MN', 'State_MO', 'State_MS', 'State_MT', 'State_NC', 'State_ND', 'State_NE', 'State_NH', 'St  
ate_NJ', 'State_NM', 'State_NV', 'State_NY', 'State_OH', 'State_OK', 'State_OR', 'State_PA', 'State_RI', 'State_SC', 'State_S  
D', 'State_TN', 'State_TX', 'State_UT', 'State_VA', 'State_VT', 'State_WA', 'State_WI', 'State_WV', 'State_WY', 'AreaCode_408',  
'AreaCode_415', 'AreaCode_510']
```

Now, the next step is defining X and Y variables:

```
In [65]: x=features[cols]  
y=features['Churn']
```

Now, the next step is to split the data:

```
In [68]: import sklearn
         from sklearn.ensemble import RandomForestClassifier
         from sklearn import metrics
         from sklearn.cross_validation import train_test_split
```

```
In [69]: #Random forest model

         X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.3, random_state=0)
```

To understand model performance, dividing the dataset into a training set and a test set is a good strategy. Let's split dataset by using function `train_test_split()`.

Machine learning is about learning some properties of a data set and then testing those properties against another data set. A common practice in machine learning is to evaluate an algorithm by splitting a data set into two. We call one of those sets the **training set**, on which we learn some properties; we call the other set the **testing set**, on which we test the learned properties.

You can't possibly manually split the dataset into two. And you also have to make sure you split the data in a random manner. To help us with this task, the SciKit library provides a tool, called the Model Selection library. There's a class in the library which is, aptly, named ['train_test_split'](#). Using this we can easily split the dataset into the training and the testing datasets in various proportions.

- **test_size** — This parameter decides the size of the data that has to be split as the test dataset. This is given as a fraction. For example, if you pass 0.3 as the value, the dataset will be split 30% as the test dataset.

- **train_size** — You have to specify this parameter only if you're not specifying the test_size. This is the same as test_size, but instead you tell the class what percent of the dataset you want to split as the training set.
- **random_state** — Here you pass an integer, which will act as the seed for the random number generator during the split.

After this fit your model on the train set using fit()

Once the data has been divided into the training and testing sets, the final step is to train the decision tree algorithm on this data and make predictions. Scikit-Learn contains the tree library, which contains built-in classes/methods for various decision tree algorithms. Since we are going to perform a classification task here, we will use

```
In [73]: RFC = RandomForestClassifier()
         RFC.fit(X_train, y_train)
```

```
Out[73]: RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                                max_depth=None, max_features='auto', max_leaf_nodes=None,
                                min_impurity_decrease=0.0, min_impurity_split=None,
                                min_samples_leaf=1, min_samples_split=2,
                                min_weight_fraction_leaf=0.0, n_estimators=10, n_jobs=1,
                                oob_score=False, random_state=None, verbose=0,
                                warm_start=False)
```

A random forest is a meta estimator that fits a number of decision tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting. The sub-sample size is always the same as the original input sample size but the samples are drawn with replacement if bootstrap=True (default). In case of regression we used the RandomForestRegressor class of the sklearn.ensemble library. For

classification, we will RandomForestClassifier class of the sklearn.ensemble library. RandomForestClassifier class also takes n_estimators as a parameter. Like before, this parameter defines the number of trees in our random forest.

- **criterion:** It defines the function to measure the quality of a split. Sklearn supports "gini" criteria for Gini Index & "entropy" for Information Gain. By default, it takes "gini" value.
- **splitter:** It defines the strategy to choose the split at each node. Supports "best" value to choose the best split & "random" to choose the best random split. By default, it takes "best" value.
- **max_features:** It defines the no. of features to consider when looking for the best split. We can input integer, float, string & None value.
 - If an integer is inputted then it considers that value as max features at each split.
 - If float value is taken then it shows the percentage of features at each split.
 - If "auto" or "sqrt" is taken then $\text{max_features} = \sqrt{n_features}$.
 - If "log2" is taken then $\text{max_features} = \log_2(n_features)$.
 - If None, then $\text{max_features} = n_features$. By default, it takes "None" value.
- **max_depth:** The max_depth parameter denotes maximum depth of the tree. It can take any integer value or None. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples. By default, it takes "None" value.



```
Out[75]: array([[0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],  
                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1],  
                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0],  
                [0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],  
                [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0],  
                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0],  
                [0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],  
                [0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0],  
                [1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],  
                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],  
                [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],  
                [0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],  
                [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],  
                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],  
                [0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],  
                [0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0],  
                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0],  
                [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])
```

To check how much this model is accurate:


The function `accuracy_score()` will be used to print accuracy of Decision Tree algorithm. By accuracy, we mean the ratio of the correctly predicted data points to all the predicted data points. Accuracy as a metric helps to understand the effectiveness of our algorithm. It takes 4 parameters.

- `y_true`,
- `y_pred`,
- `normalize`,
- `sample_weight`.

Out of these 4, `normalize` & `sample_weight` are optional parameters. The parameter `y_true` accepts an array of correct labels and `y_pred` takes an array of predicted labels that are returned by the classifier. It returns accuracy as a float value.

```
In [76]: from sklearn.metrics import accuracy_score  
print('RF accuracy: {:.3f}'.format(accuracy_score(y_test, RFC.predict(X_test))))
```


RF accuracy: 0.933



The accuracy is 93.3%, if you use dummies

```
In [97]: print('RF accuracy: {:.3f}'.format(accuracy_score(y_test, RFC.predict(X_test))))
```

RF accuracy: 0.939



The accuracy is 93.9%, if we use label encoding



Model Evaluation using Confusion Matrix

A confusion matrix is a table that is used to evaluate the performance of a classification model. In the field of machine learning and specifically the problem of statistical classification, a confusion matrix, also known as an error matrix.

A confusion matrix is a table that is often used to describe the performance of a classification model (or "classifier") on a set of test data for which the true values are known. It allows the visualization of the performance of an algorithm.

It allows easy identification of confusion between classes e.g. one class is commonly mislabeled as the other. Most performance measures are computed from the confusion matrix.

- Positive (P) : Observation is positive (for example: is an apple).
- Negative (N) : Observation is not positive (for example: is not an apple).
- True Positive (TP) : Observation is positive, and is predicted to be positive.
- False Negative (FN) : Observation is positive, but is predicted negative.
- True Negative (TN) : Observation is negative, and is predicted to be negative.
- False Positive (FP) : Observation is negative, but is predicted positive.

At this point we have trained our algorithm and made some predictions. Now we'll see how accurate our algorithm is. For classification tasks some commonly used metrics are [confusion matrix](#), precision, recall, and [F1 score](#). Lucky for us Scikit-Learn's metrics library contains

the `classification_report` and `confusion_matrix` methods that can be used to calculate these metrics for us:

```
In [78]: from sklearn.metrics import classification_report
print(classification_report(y_test, RFC.predict(X_test)))
```

	precision	recall	f1-score	support
0	0.94	0.98	0.96	862
1	0.87	0.61	0.71	138
avg / total	0.93	0.93	0.93	1000

Classification Rate or Accuracy is given by the relation:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

Recall:

Recall can be defined as the ratio of the total number of correctly classified positive examples divide to the total number of positive examples. High Recall indicates the class is correctly recognized (small number of FN).

Recall is given by the relation:

$$\text{Recall} = \frac{TP}{TP + FN}$$

To get the value of precision we divide the total number of correctly classified positive examples by the total number of predicted positive examples. High Precision indicates an example labeled as positive is indeed positive (small number of FP).

Precision is given by the relation:

$$\text{Precision} = \frac{TP}{TP + FP}$$

High recall, low precision: This means that most of the positive examples are correctly recognized (low FN) but there are a lot of false positives.

Low recall, high precision: This shows that we miss a lot of positive examples (high FN) but those we predict as positive are indeed positive (low FP)

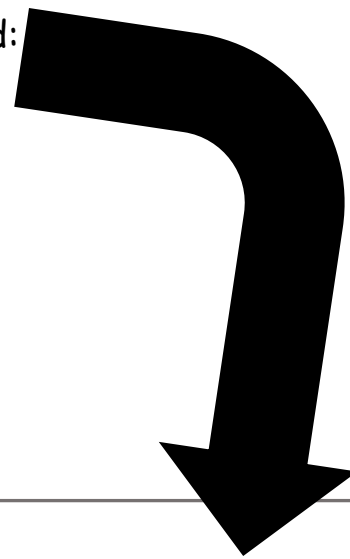
F-measure:

Since we have two measures (Precision and Recall) it helps to have a measurement that represents both of them. We calculate an F-measure which uses Harmonic Mean in place of Arithmetic Mean as it punishes the extreme values more.

The F-Measure will always be nearer to the smaller value of Precision or Recall.

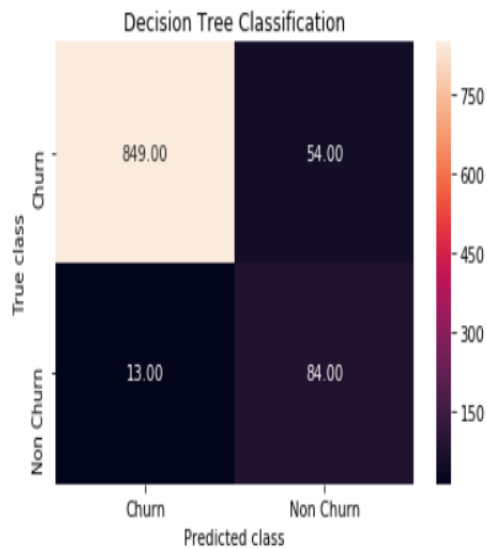
$$F - \text{measure} = \frac{2 * \text{Recall} * \text{Precision}}{\text{Recall} + \text{Precision}}$$

Confusion matrix when dummies method is used:

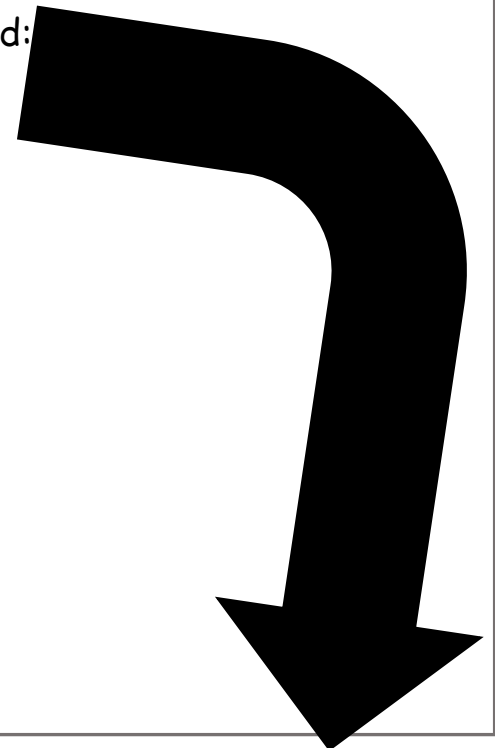


```
In [82]: %matplotlib inline
import matplotlib.pyplot as plt
pred = RFC.predict(X_test)
from sklearn.metrics import confusion_matrix
import seaborn as sns
forest_cm = metrics.confusion_matrix(pred, y_test)
sns.heatmap(forest_cm, annot=True, fmt='.2f', xticklabels = ["Churn", "Non Churn"], yticklabels = ["Churn", "Non Churn"] )
plt.ylabel('True class')
plt.xlabel('Predicted class')
plt.title('Decision Tree Classification')
```

Out[82]: Text(0.5,1,'Decision Tree Classification')



Confusion matrix when label encoding method is used:





```
In [98]: pred = RFC.predict(X_test)
from sklearn.metrics import confusion_matrix
import seaborn as sns
forest_cm = metrics.confusion_matrix(pred, y_test)
sns.heatmap(forest_cm, annot=True, fmt='.2f', xticklabels = ["Churn", "Non Churn"], yticklabels = ["Churn", "Non Churn"] )
plt.ylabel('True class')
plt.xlabel('Predicted class')
plt.title('Decision Tree Classification')
```

Out[98]: Text(0.5,1,'Decision Tree Classification')

