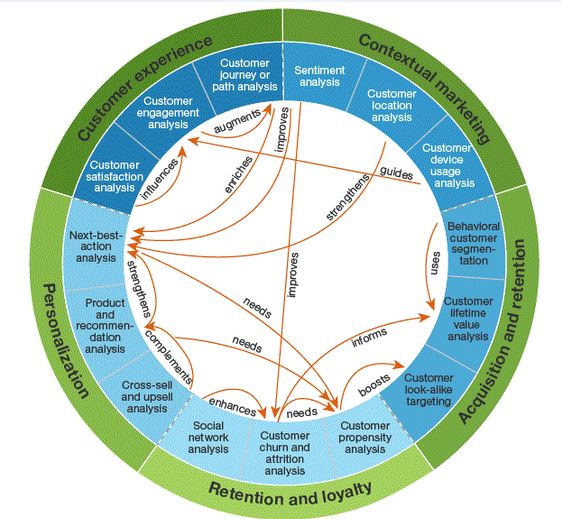
**Churn Modeling and Analysis**

Customer churn is a serious problem in the competitive world. Almost very organization face this issue of customer churn. Best way to reduce the churn is by identifying the customers who may churn and then take some action to retain the customer and keep them loyal and happy. Customer churn analysis is important part of customer analysis and we have various frame work available for the same. One of the most popular frame-work for customer analysis is given by Forrester.

**What is Forrester customer analytics framework?**

The scale and diversity of customer data provide rich new sources of insight, letting firms engage with customers in new ways and enable the digital disruption of entire industries. This will help us to understand customer and provide better understanding and insights. Forrester Research Inc. has provided with reference model called Forrester customer analytics model and Framework, which can be used as a reference for developing any solutions for customer analytics.

Shown below:



**Fig: Forrester customer analytics model**

The above model shows major domains in customer analytics, along with the dependencies between each domain. Each domain is further broken down to 15 sub domains and each sub domains places important role in customer analytics.

**1) Contextual Marketing:**

Customers are everything to any organization but there is very rare chance that customer will approach you first. In the completive world the most important part is to target best clients who will get engage with us in near future.

Contextual marketing refers to online and mobile marketing that provides targeted advertising based upon user information, such as the search terms they’re using or recent web-browsing activity (See also Computational Marketing).The goal is to present ads to customers representing products and services they are already interested in. For example, a customer performs an Internet search for cars and fuel efficiency. Afterwards, they check their daily news website and the ads which show up alongside of news includes for hybrid cars. The customer, already thinking about saving fuel on their commute, clicks on the ad to check out the latest hybrids.

**2) Acquisition and retention:**

Once we get the details of potential customer then the next task is to acquire the customer and get him engage with our products and services.

Best example will be POC done to our clients. Here we know what they require and now it is our task to acquire those clients by providing them confidence with our product and/or service.

**3) Retention and loyalty:**

Customer retention refers to the activities and actions companies and organizations take to reduce the number of customer defections. The goal of customer retention programs is to help companies retain as many customers as possible, often through customer loyalty and brand loyalty initiatives.

**4) Personalization:**

This involves generation of recommendation in which customer will be interested in, identifying the best possible action to make business from the customer along with keeping customer happy.

**5) Customer Experience:**

Customer experience is most important for the business. When customer has good experience they remain loyal and are retained with the business. We can understand customer experience by understanding customer engagement with the organization, satisfaction with the requirement fulfilled also by understanding the journey from the start of service.

**Current Problem Statement:**

There is a US based telecom company who face a problem with customer churn we need to make a predictive model so that company knows about the customers who will churn in near future so that they can take active steps for customer retention.

**Data Set Provided Overview:**

Company keeps track of following data and which I reflected in the data set provided.

* State: They are state code (2 character) representing the state from US.
* Account length: This represents customer timeline.
* Area code: It is to specify the exact location of customer, just like pin-code /postcode.
* Phone number: can be used as customer identifier.
* International plan: It states if the customer as opt for international service plan.
* Voice mail plan: It states if the customer as opt for voice mail service plan.
* Number vmail messages: Total voice-mail send.
* Total day minutes: Shows total minute spent in call during the day.
* Total day calls: Total number of calls during the day.
* Total day charge: Total charge the customer has to pay for all the calls during the day.
* Total eve minutes: Shows total minute spent in call during the evening.
* Total eve calls: Total number of calls during the evening.
* Total eve charge: Total charge the customer has to pay for all the calls during the evening.
* Total night minutes: Shows total minute spent in call during the night.
* Total night calls: Total number of calls during the night.
* Total night charge: Total charge the customer has to pay for all the calls during the night.
* Total intl minutes: Shows total minute spent in international calls.
* Total intl calls: Total number of international calls.
* Total intl charge: Total charge the customer has to pay for all the international calls.
* Customer service calls: Calls made to customer service center.
* Churn: Shows whether the customer has churned or not.

**Approach for getting solution**

1. **Understand the problem.**
2. **Understand the data.**
3. **Discover the relations among data.**
4. **Featuring engineering.**
5. **Modeling.**
6. **Predict the result.**

**Understand the problem**

Understanding the problem is very important as this helps in taking right step in getting accurate and desired result. The present problem is to identify and predict the customer who may churn in near future. We can do this by using Machine Learning (Classifier Problem).

**Understand the data**

This is one and most important step while working with discovery phase. This involves understanding of meanings and value of data provided. It also includes clarifying the doubts in data if present.

In our case we have a clear idea of data provided to us which are already described above.

**Discover the relations among data**

We gather, analyze relationship among the data. We can use different approach for the same.

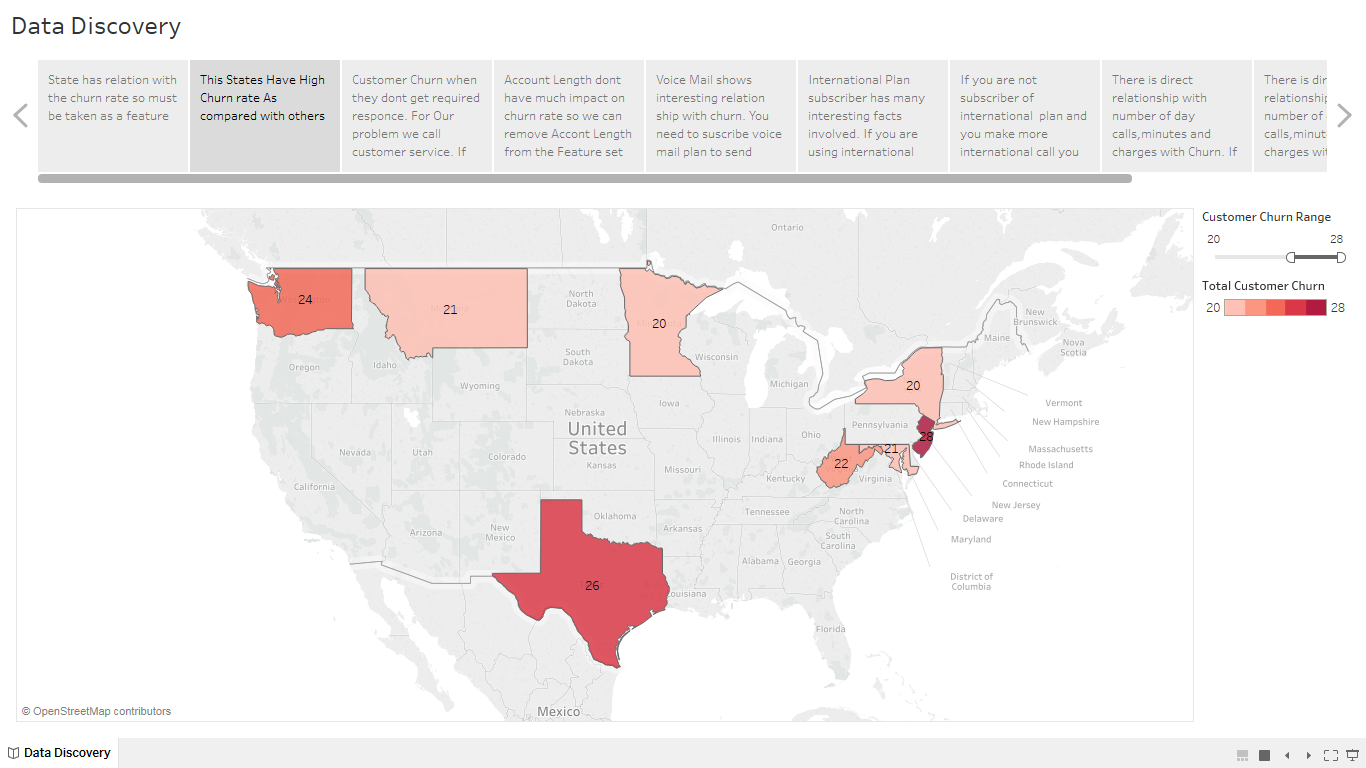
1. Using python libraries such as seaborn, matplotlib, etc.
2. Visualization tools.

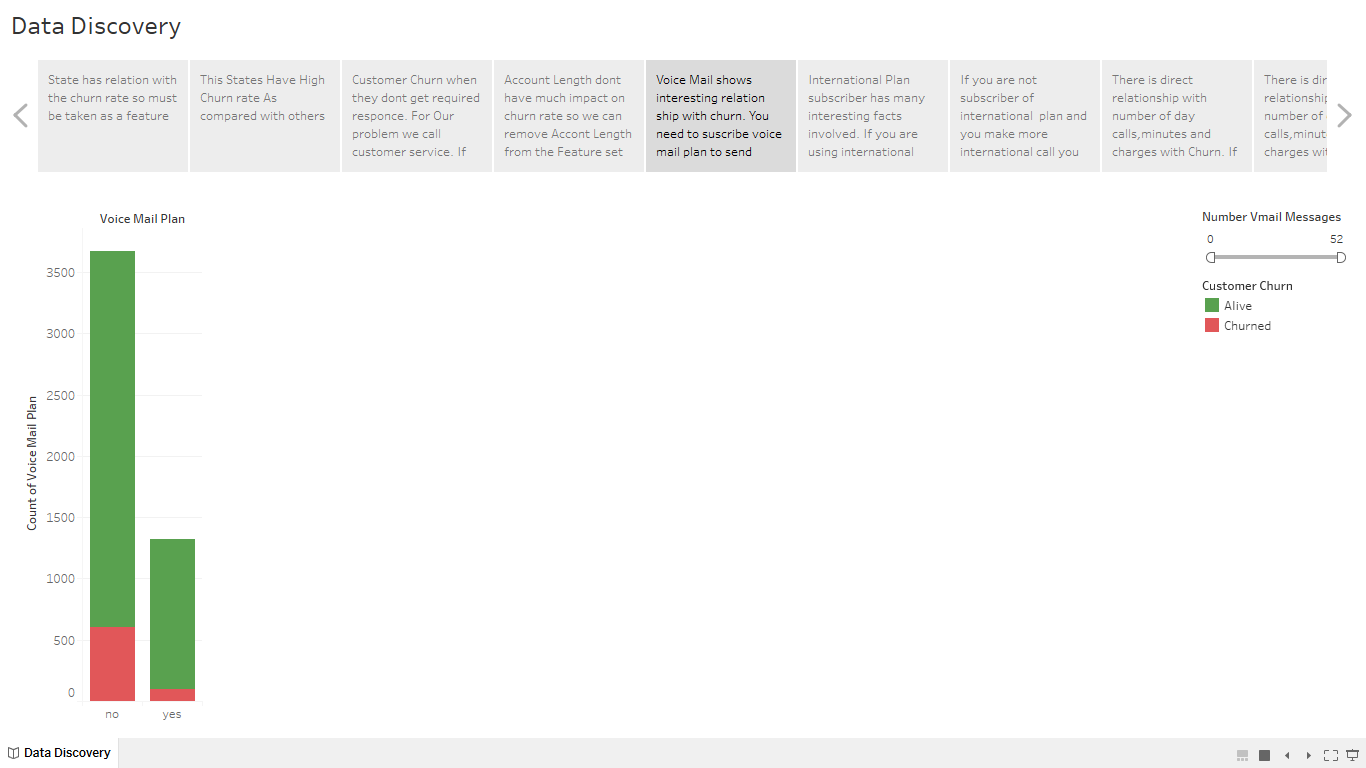
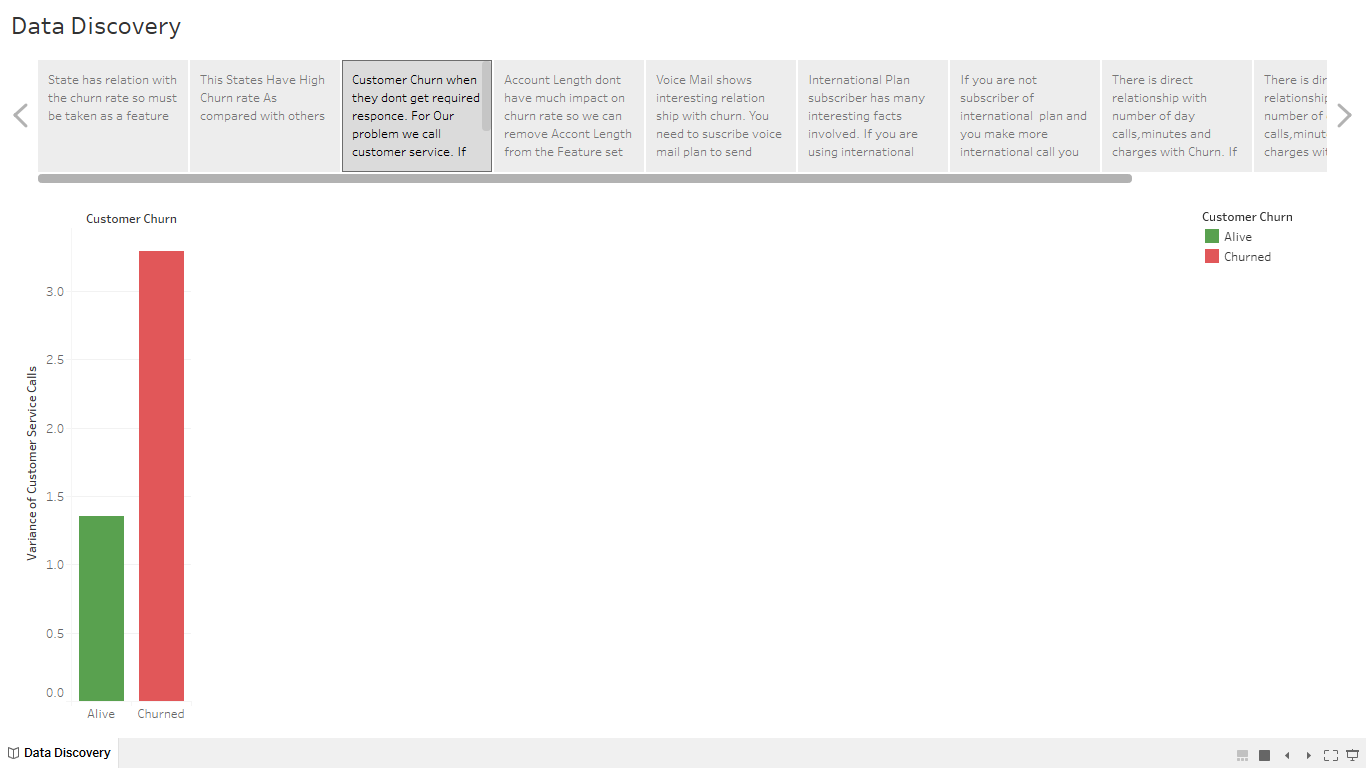
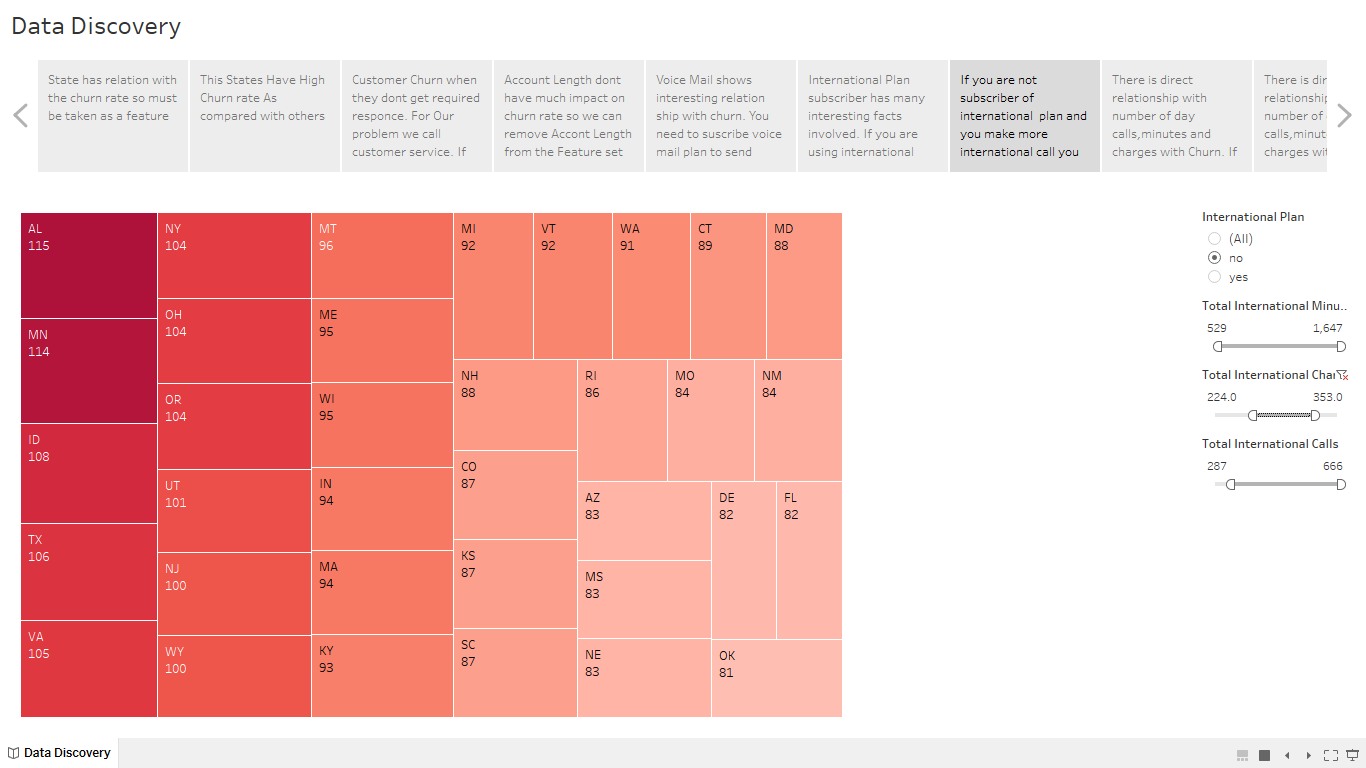
The best approach is to use visualization tools as it reduces the coding and also give better picture of data. We can track the changes in data using UI controllers and filters. This helps in understanding the story.

The data which has the highest relationship (dependency) with the problem are used as Features during the modeling phase.

For the given problem let’s use the visualization and reporting tool Tableau to determine the insights form the data.

**Few screenshot below:**



**Featuring engineering**

Once we know the relationship among the data we work for getting the Features for modeling.Feature engineering is the process of using domain knowledge of the data to create features that make machine learning algorithms work. Feature engineering is fundamental to the application of machine learning, and is both difficult and expensive. The need for manual feature engineering can be obviated by automated feature learning.

A feature is an attribute or property shared by all of the independent units on which analysis or prediction is to be done. Any attribute could be a feature, as long as it is useful to the model. The features in your data are important to the predictive models you use and will influence the results you are going to achieve. The quality and quantity of the features will have great influence on whether the model is good or not.

General approach for Featuring engineering:

1. Brainstorming or Testing features
2. Deciding what features to create
3. Creating features
4. Checking how the features work with your model
5. Improving your features if needed
6. Go back to brainstorming/creating more features until the work is done.

Features in our problem are:

* State
* International plan
* Voice mail plan
* Number vmail messages
* Total day minutes
* Total day calls
* Total day charge
* Total eve minutes
* Total eve calls
* Total eve charge
* Total night minutes
* Total night calls
* Total night charge
* Total intl minutes
* Total intl calls
* Total intl charge
* Customer service calls

**Modeling**

There are many models available for prediction and classification so we don’t need to make all of our own. In python we have sklearn library which provide almost all the models which we require for machine learning.

For churn modeling following models are mostly used:

1. Random Forest Classifier
2. Gradient Boosting Classifier
3. SVC – Linear SVC
4. Logistic Regression
5. Naive Bayes Classifier

Let’s have overview of the model along with function provided by sklearn library. We will also understand the parameters for the functions so as to optimize the code for optimization and better results.

**1)Random Forest Classifier**

**Random forests** or **random decision forests** are an ensemble learning method for classification, regression and other tasks, that operate by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees. Random decision forests correct for decision trees' habit of overfitting to their training set.

A random forest is a meta estimator that fits a number of decision tree classifiers on various sub-samples of the dataset and use 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).

Following is the explanation of parameter which we can pass:

|  |  |
| --- | --- |
| **Parameters:** | **n\_estimators** : integer, optional (default=10)  The number of trees in the forest.  **criterion** : string, optional (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. Note: this parameter is tree-specific.  **max\_features** : int, float, string or None, optional (default=”auto”)  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 percentage 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)* (same as “auto”). * 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.  **max\_depth** : integer or None, optional (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, float, optional (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 percentage and *ceil(min\_samples\_split \* n\_samples)* are the minimum number of samples for each split.   Changed in version 0.18: Added float values for percentages.  **min\_samples\_leaf** : int, float, optional (default=1)  The minimum number of samples required to be at a leaf node:   * If int, then consider *min\_samples\_leaf* as the minimum number. * If float, then *min\_samples\_leaf* is a percentage and *ceil(min\_samples\_leaf \* n\_samples)* are the minimum number of samples for each node.   Changed in version 0.18: Added float values for percentages.  **min\_weight\_fraction\_leaf** : float, optional (default=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\_leaf\_nodes** : int or None, optional (default=None)  Grow trees 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\_split** : float,  Threshold for early stopping in tree growth. A node will split if its impurity is above the threshold, otherwise it is a leaf.  Deprecated since version 0.19: min\_impurity\_split has been deprecated in favor of min\_impurity\_decrease in 0.19 and will be removed in 0.21. Use min\_impurity\_decrease instead.  **min\_impurity\_decrease** : float, optional (default=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.  New in version 0.19.  **bootstrap** : boolean, optional (default=True)  Whether bootstrap samples are used when building trees.  **oob\_score** : bool (default=False)  Whether to use out-of-bag samples to estimate the generalization accuracy.  **n\_jobs** : integer, optional (default=1)  The number of jobs to run in parallel for both *fit* and *predict*. If -1, then the number of jobs is set to the number of cores.  **random\_state** : int, RandomState instance or None, optional (default=None)  If int, random\_state is the seed used by the random number generator; If RandomState instance, random\_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.  **verbose** : int, optional (default=0)  Controls the verbosity of the tree building process.  **warm\_start** : bool, optional (default=False)  When set to True, reuse the solution of the previous call to fit and add more estimators to the ensemble, otherwise, just fit a whole new forest.  **class\_weight** : dict, list of dicts, “balanced”,  “balanced\_subsample” or None, optional (default=None) Weights associated with classes in the form {class\_label: weight}. If not given, 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))  The “balanced\_subsample” mode is the same as “balanced” except that weights are computed based on the bootstrap sample for every tree grown.  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. |
| **Attributes:** | **estimators\_** : list of DecisionTreeClassifier  The collection of fitted sub-estimators.  **classes\_** : array of shape = [n\_classes] or a list of such arrays  The classes labels (single output problem), or a list of arrays of class labels (multi-output problem).  **n\_classes\_** : int or list  The number of classes (single output problem), or a list containing the number of classes for each output (multi-output problem).  **n\_features\_** : int  The number of features when fit is performed.  **n\_outputs\_** : int  The number of outputs when fit is performed.  **feature\_importances\_** : array of shape = [n\_features]  The feature importances (the higher, the more important the feature).  **oob\_score\_** : float  Score of the training dataset obtained using an out-of-bag estimate.  **oob\_decision\_function\_** : array of shape = [n\_samples, n\_classes]  Decision function computed with out-of-bag estimate on the training set. If n\_estimators is small it might be possible that a data point was never left out during the bootstrap. In this case, *oob\_decision\_function\_* might contain NaN. |

**2)Gradient Boosting Classifier**

**Gradient boosting** is a machine learning technique for regression and classification problems, which produces a prediction model in the form of an ensemble of weak prediction models, typically decision trees. It builds the model in a stage-wise fashion like other boosting methods do, and it generalizes them by allowing optimization of an arbitrary differentiable loss function.

The boosting can be interpreted as an optimization algorithm on a suitable cost function. Algorithms that optimize a cost *function* over function space by iteratively choosing a function (weak hypothesis) those points in the negative gradient direction. This functional gradient view of boosting has led to the development of boosting algorithms in many areas of machine learning and statistics beyond regression and classification.

**Gradient Boosting for classification**

GB builds an additive model in a forward stage-wise fashion; it allows for the optimization of arbitrary differentiable loss functions. In each stage n\_classes\_ regression trees are fit on the negative gradient of the binomial or multinomial deviance loss function. Binary classification is a special case where only a single regression tree is induced.

|  |  |
| --- | --- |
| **Parameters:** | **loss** : {‘deviance’, ‘exponential’}, optional (default=’deviance’)  loss function to be optimized. ‘deviance’ refers to deviance (= logistic regression) for classification with probabilistic outputs. For loss ‘exponential’ gradient boosting recovers the AdaBoost algorithm.  **learning\_rate** : float, optional (default=0.1)  learning rate shrinks the contribution of each tree by *learning\_rate*. There is a trade-off between learning\_rate and n\_estimators.  **n\_estimators** : int (default=100)  The number of boosting stages to perform. Gradient boosting is fairly robust to over-fitting so a large number usually results in better performance.  **max\_depth** : integer, optional (default=3)  maximum depth of the individual regression estimators. The maximum depth limits the number of nodes in the tree. Tune this parameter for best performance; the best value depends on the interaction of the input variables.  **criterion** : string, optional (default=”friedman\_mse”)  The function to measure the quality of a split. Supported criteria are “friedman\_mse” for the mean squared error with improvement score by Friedman, “mse” for mean squared error, and “mae” for the mean absolute error. The default value of “friedman\_mse” is generally the best as it can provide a better approximation in some cases.  New in version 0.18.  **min\_samples\_split** : int, float, optional (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 percentage and *ceil(min\_samples\_split \* n\_samples)* are the minimum number of samples for each split.   Changed in version 0.18: Added float values for percentages.  **min\_samples\_leaf** : int, float, optional (default=1)  The minimum number of samples required to be at a leaf node:   * If int, then consider *min\_samples\_leaf* as the minimum number. * If float, then *min\_samples\_leaf* is a percentage and *ceil(min\_samples\_leaf \* n\_samples)* are the minimum number of samples for each node.   Changed in version 0.18: Added float values for percentages.  **min\_weight\_fraction\_leaf** : float, optional (default=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.  **subsample** : float, optional (default=1.0)  The fraction of samples to be used for fitting the individual base learners. If smaller than 1.0 this results in Stochastic Gradient Boosting. *subsample* interacts with the parameter *n\_estimators*. Choosing *subsample < 1.0* leads to a reduction of variance and an increase in bias.  **max\_features** : int, float, string or None, optional (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 percentage 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*.   Choosing *max\_features < n\_features* leads to a reduction of variance and an increase in bias.  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.  **max\_leaf\_nodes** : int or None, optional (default=None)  Grow trees 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\_split** : float,  Threshold for early stopping in tree growth. A node will split if its impurity is above the threshold, otherwise it is a leaf.  Deprecated since version 0.19: min\_impurity\_split has been deprecated in favor of min\_impurity\_decrease in 0.19 and will be removed in 0.21. Use min\_impurity\_decrease instead.  **min\_impurity\_decrease** : float, optional (default=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.  New in version 0.19.  **init** : BaseEstimator, None, optional (default=None)  An estimator object that is used to compute the initial predictions. init has to provide fit and predict. If None it uses loss.init\_estimator.  **verbose** : int, default: 0  Enable verbose output. If 1 then it prints progress and performance once in a while (the more trees the lower the frequency). If greater than 1 then it prints progress and performance for every tree.  **warm\_start** : bool, default: False  When set to True, reuse the solution of the previous call to fit and add more estimators to the ensemble, otherwise, just erase the previous solution.  **random\_state** : int, RandomState instance or None, optional (default=None)  If int, random\_state is the seed used by the random number generator; If RandomState instance, random\_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.  **presort** : bool or ‘auto’, optional (default=’auto’)  Whether to presort the data to speed up the finding of best splits in fitting. Auto mode by default will use presorting on dense data and default to normal sorting on sparse data. Setting presort to true on sparse data will raise an error.  New in version 0.17: *presort* parameter. |
| **Attributes:** | **feature\_importances\_** : array, shape = [n\_features]  The feature importances (the higher, the more important the feature).  **oob\_improvement\_** : array, shape = [n\_estimators]  The improvement in loss (= deviance) on the out-of-bag samples relative to the previous iteration. oob\_improvement\_[0] is the improvement in loss of the first stage over the init estimator.  **train\_score\_** : array, shape = [n\_estimators]  The i-th score train\_score\_[i] is the deviance (= loss) of the model at iteration i on the in-bag sample. If subsample == 1 this is the deviance on the training data.  **loss\_** : LossFunction  The concrete LossFunction object.  **init** : BaseEstimator  The estimator that provides the initial predictions. Set via the init argument or loss.init\_estimator.  **estimators\_** : ndarray of DecisionTreeRegressor, shape = [n\_estimators, loss\_.K]  The collection of fitted sub-estimators. loss\_.K is 1 for binary classification, otherwise n\_classes. |

**3)Linear SVC**

“Support Vector Machine” (SVM) is a supervised machine learning algorithm which can be used for both classification or regression challenges. However, it is mostly used in classification problems. In this algorithm, we plot each data item as a point in n-dimensional space (where n is number of features you have) with the value of each feature being the value of a particular coordinate. Then, we perform classification by finding the hyper-plane that differentiate the two classes very well (look at the below snapshot).

[](https://www.analyticsvidhya.com/wp-content/uploads/2015/10/SVM_1.png)

Support Vectors are simply the co-ordinates of individual observation. Support Vector Machine is a frontier which best segregates the two classes (hyper-plane/ line).

**Support Vector Classification**

The implementation is based on libsvm. The fit time complexity is more than quadratic with the number of samples which makes it hard to scale to dataset with more than a couple of 10000 samples.

The multiclass support is handled according to a one-vs-one scheme.

|  |  |
| --- | --- |
| **Parameters:** | **C** : float, optional (default=1.0)  Penalty parameter C of the error term.  **kernel** : string, optional (default=’rbf’)  Specifies the kernel type to be used in the algorithm. It must be one of ‘linear’, ‘poly’, ‘rbf’, ‘sigmoid’, ‘precomputed’ or a callable. If none is given, ‘rbf’ will be used. If a callable is given it is used to pre-compute the kernel matrix from data matrices; that matrix should be an array of shape (n\_samples, n\_samples).  **degree** : int, optional (default=3)  Degree of the polynomial kernel function (‘poly’). Ignored by all other kernels.  **gamma** : float, optional (default=’auto’)  Kernel coefficient for ‘rbf’, ‘poly’ and ‘sigmoid’. If gamma is ‘auto’ then 1/n\_features will be used instead.  **coef0** : float, optional (default=0.0)  Independent term in kernel function. It is only significant in ‘poly’ and ‘sigmoid’.  **probability** : boolean, optional (default=False)  Whether to enable probability estimates. This must be enabled prior to calling *fit*, and will slow down that method.  **shrinking** : boolean, optional (default=True)  Whether to use the shrinking heuristic.  **tol** : float, optional (default=1e-3)  Tolerance for stopping criterion.  **cache\_size** : float, optional  Specify the size of the kernel cache (in MB).  **class\_weight** : {dict, ‘balanced’}, optional  Set the parameter C of class i to class\_weight[i]\*C for SVC. If not given, all classes are supposed to have weight one. 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))  **verbose** : bool, default: False  Enable verbose output. Note that this setting takes advantage of a per-process runtime setting in libsvm that, if enabled, may not work properly in a multithreaded context.  **max\_iter** : int, optional (default=-1)  Hard limit on iterations within solver, or -1 for no limit.  **decision\_function\_shape** : ‘ovo’, ‘ovr’, default=’ovr’  Whether to return a one-vs-rest (‘ovr’) decision function of shape (n\_samples, n\_classes) as all other classifiers, or the original one-vs-one (‘ovo’) decision function of libsvm which has shape (n\_samples, n\_classes \* (n\_classes - 1) / 2).  Changed in version 0.19: decision\_function\_shape is ‘ovr’ by default.  New in version 0.17: *decision\_function\_shape=’ovr’* is recommended.  Changed in version 0.17: Deprecated *decision\_function\_shape=’ovo’ and None*.  **random\_state** : int, RandomState instance or None, optional (default=None)  The seed of the pseudo random number generator to use when shuffling the data. If int, random\_state is the seed used by the random number generator; If RandomState instance, random\_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*. |
| **Attributes:** | **support\_** : array-like, shape = [n\_SV]  Indices of support vectors.  **support\_vectors\_** : array-like, shape = [n\_SV, n\_features]  Support vectors.  **n\_support\_** : array-like, dtype=int32, shape = [n\_class]  Number of support vectors for each class.  **dual\_coef\_** : array, shape = [n\_class-1, n\_SV]  Coefficients of the support vector in the decision function. For multiclass, coefficient for all 1-vs-1 classifiers. The layout of the coefficients in the multiclass case is somewhat non-trivial. See the section about multi-class classification in the SVM section of the User Guide for details.  **coef\_** : array, shape = [n\_class-1, n\_features]  Weights assigned to the features (coefficients in the primal problem). This is only available in the case of a linear kernel.  *coef\_* is a readonly property derived from *dual\_coef\_* and *support\_vectors\_*.  **intercept\_** : array, shape = [n\_class \* (n\_class-1) / 2]  Constants in decision function. |

**Linear Support Vector Classification**

Similar to SVC with parameter kernel=’linear’, but implemented in terms of liblinear rather than libsvm, so it has more flexibility in the choice of penalties and loss functions and should scale better to large numbers of samples.

This class supports both dense and sparse input and the multiclass support is handled according to a one-vs-the-rest scheme.

|  |  |
| --- | --- |
| **Parameters:** | **penalty** : string, ‘l1’ or ‘l2’ (default=’l2’)  Specifies the norm used in the penalization. The ‘l2’ penalty is the standard used in SVC. The ‘l1’ leads to coef\_ vectors that are sparse.  **loss** : string, ‘hinge’ or ‘squared\_hinge’ (default=’squared\_hinge’)  Specifies the loss function. ‘hinge’ is the standard SVM loss (used e.g. by the SVC class) while ‘squared\_hinge’ is the square of the hinge loss.  **dual** : bool, (default=True)  Select the algorithm to either solve the dual or primal optimization problem. Prefer dual=False when n\_samples > n\_features.  **tol** : float, optional (default=1e-4)  Tolerance for stopping criteria.  **C** : float, optional (default=1.0)  Penalty parameter C of the error term.  **multi\_class** : string, ‘ovr’ or ‘crammer\_singer’ (default=’ovr’)  Determines the multi-class strategy if *y* contains more than two classes. "ovr" trains n\_classes one-vs-rest classifiers, while "crammer\_singer" optimizes a joint objective over all classes. While *crammer\_singer* is interesting from a theoretical perspective as it is consistent, it is seldom used in practice as it rarely leads to better accuracy and is more expensive to compute. If "crammer\_singer" is chosen, the options loss, penalty and dual will be ignored.  **fit\_intercept** : boolean, optional (default=True)  Whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (i.e. data is expected to be already centered).  **intercept\_scaling** : float, optional (default=1)  When self.fit\_intercept is True, instance vector x becomes [x, self.intercept\_scaling], i.e. a “synthetic” feature with constant value equals to intercept\_scaling is appended to the instance vector. The intercept becomes intercept\_scaling \* synthetic feature weight Note! the synthetic feature weight is subject to l1/l2 regularization as all other features. To lessen the effect of regularization on synthetic feature weight (and therefore on the intercept) intercept\_scaling has to be increased.  **class\_weight** : {dict, ‘balanced’}, optional  Set the parameter C of class i to class\_weight[i]\*C for SVC. If not given, all classes are supposed to have weight one. 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))  **verbose** : int, (default=0)  Enable verbose output. Note that this setting takes advantage of a per-process runtime setting in liblinear that, if enabled, may not work properly in a multithreaded context.  **random\_state** : int, RandomState instance or None, optional (default=None)  The seed of the pseudo random number generator to use when shuffling the data. If int, random\_state is the seed used by the random number generator; If RandomState instance, random\_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.  **max\_iter** : int, (default=1000)  The maximum number of iterations to be run. |
| **Attributes:** | **coef\_** : array, shape = [n\_features] if n\_classes == 2 else [n\_classes, n\_features]  Weights assigned to the features (coefficients in the primal problem). This is only available in the case of a linear kernel.  coef\_ is a readonly property derived from raw\_coef\_ that follows the internal memory layout of liblinear.  **intercept\_** : array, shape = [1] if n\_classes == 2 else [n\_classes]  Constants in decision function. |

**4)Logistic Regression**

Logistic regression is another technique borrowed by machine learning from the field of statistics. It is the go-to method for binary classification problems (problems with two class values).

This post was written for developers interested in applied machine learning, specifically predictive modeling. You do not need to have a background in linear algebra or statistics.

## Logistic Function: Logistic regression is named for the function used at the core of the method, the logistic function. The logistic function, also called the sigmoid function was developed by statisticians to describe properties of population growth in ecology, rising quickly and maxing out at the carrying capacity of the environment. It’s an S-shaped curve that can take any real-valued number and map it into a value between 0 and 1, but never exactly at those limits.

sigmoid function = 1 / (1 + e^-value)

Where e is the base of the natural logarithms (Euler’s number or the EXP() function in your spreadsheet) and value is the actual numerical value that you want to transform. Below is a plot of the numbers between -5 and 5 transformed into the range 0 and 1 using the logistic function.



**Fig: Sigmoid function**

**Logistic Regression (aka logit, MaxEnt) classifier**

In the multiclass case, the training algorithm uses the one-vs-rest (OvR) scheme if the ‘multi\_class’ option is set to ‘ovr’, and uses the cross- entropy loss if the ‘multi\_class’ option is set to ‘multinomial’. (Currently the ‘multinomial’ option is supported only by the ‘lbfgs’, ‘sag’ and ‘newton-cg’ solvers.)

This class implements regularized logistic regression using the ‘liblinear’ library, ‘newton-cg’, ‘sag’ and ‘lbfgs’ solvers. It can handle both dense and sparse input. Use C-ordered arrays or CSR matrices containing 64-bit floats for optimal performance; any other input format will be converted (and copied).

The ‘newton-cg’, ‘sag’, and ‘lbfgs’ solvers support only L2 regularization with primal formulation. The ‘liblinear’ solver supports both L1 and L2 regularization, with a dual formulation only for the L2 penalty.

|  |  |
| --- | --- |
| **Parameters:** | **penalty** : str, ‘l1’ or ‘l2’, default: ‘l2’  Used to specify the norm used in the penalization. The ‘newton-cg’, ‘sag’ and ‘lbfgs’ solvers support only l2 penalties.  New in version 0.19: l1 penalty with SAGA solver (allowing ‘multinomial’ + L1)  **dual** : bool, default: False  Dual or primal formulation. Dual formulation is only implemented for l2 penalty with liblinear solver. Prefer dual=False when n\_samples > n\_features.  **tol** : float, default: 1e-4  Tolerance for stopping criteria.  **C** : float, default: 1.0  Inverse of regularization strength; must be a positive float. Like in support vector machines, smaller values specify stronger regularization.  **fit\_intercept** : bool, default: True  Specifies if a constant (a.k.a. bias or intercept) should be added to the decision function.  **intercept\_scaling** : float, default 1.  Useful only when the solver ‘liblinear’ is used and self.fit\_intercept is set to True. In this case, x becomes [x, self.intercept\_scaling], i.e. a “synthetic” feature with constant value equal to intercept\_scaling is appended to the instance vector. The intercept becomes intercept\_scaling \* synthetic\_feature\_weight.  Note! the synthetic feature weight is subject to l1/l2 regularization as all other features. To lessen the effect of regularization on synthetic feature weight (and therefore on the intercept) intercept\_scaling has to be increased.  **class\_weight** : dict or ‘balanced’, default: None  Weights associated with classes in the form {class\_label: weight}. If not given, all classes are supposed to have weight one.  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)).  Note that these weights will be multiplied with sample\_weight (passed through the fit method) if sample\_weight is specified.  New in version 0.17: *class\_weight=’balanced’*  **random\_state** : int, RandomState instance or None, optional, default: None  The seed of the pseudo random number generator to use when shuffling the data. If int, random\_state is the seed used by the random number generator; If RandomState instance, random\_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*. Used when solver == ‘sag’ or ‘liblinear’.  **solver** : {‘newton-cg’, ‘lbfgs’, ‘liblinear’, ‘sag’, ‘saga’},  default: ‘liblinear’ Algorithm to use in the optimization problem.   * For small datasets, ‘liblinear’ is a good choice, whereas ‘sag’ and   ‘saga’ are faster for large ones.   * For multiclass problems, only ‘newton-cg’, ‘sag’, ‘saga’ and ‘lbfgs’   handle multinomial loss; ‘liblinear’ is limited to one-versus-rest schemes.   * ‘newton-cg’, ‘lbfgs’ and ‘sag’ only handle L2 penalty, whereas   ‘liblinear’ and ‘saga’ handle L1 penalty.  Note that ‘sag’ and ‘saga’ fast convergence is only guaranteed on features with approximately the same scale. You can preprocess the data with a scaler from sklearn.preprocessing.  New in version 0.17: Stochastic Average Gradient descent solver.  New in version 0.19: SAGA solver.  **max\_iter** : int, default: 100  Useful only for the newton-cg, sag and lbfgs solvers. Maximum number of iterations taken for the solvers to converge.  **multi\_class** : str, {‘ovr’, ‘multinomial’}, default: ‘ovr’  Multiclass option can be either ‘ovr’ or ‘multinomial’. If the option chosen is ‘ovr’, then a binary problem is fit for each label. Else the loss minimised is the multinomial loss fit across the entire probability distribution. Does not work for liblinear solver.  New in version 0.18: Stochastic Average Gradient descent solver for ‘multinomial’ case.  **verbose** : int, default: 0  For the liblinear and lbfgs solvers set verbose to any positive number for verbosity.  **warm\_start** : bool, default: False  When set to True, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution. Useless for liblinear solver.  New in version 0.17: *warm\_start* to support *lbfgs*, *newton-cg*, *sag*, *saga* solvers.  **n\_jobs** : int, default: 1  Number of CPU cores used when parallelizing over classes if multi\_class=’ovr’”. This parameter is ignored when the [``](http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html#id1)solver``is set to ‘liblinear’ regardless of whether ‘multi\_class’ is specified or not. If given a value of -1, all cores are used. |
| **Attributes:** | **coef\_** : array, shape (1, n\_features) or (n\_classes, n\_features)  Coefficient of the features in the decision function.  *coef\_* is of shape (1, n\_features) when the given problem is binary.  **intercept\_** : array, shape (1,) or (n\_classes,)  Intercept (a.k.a. bias) added to the decision function.  If *fit\_intercept* is set to False, the intercept is set to zero. *intercept\_* is of shape(1,) when the problem is binary.  **n\_iter\_** : array, shape (n\_classes,) or (1, )  Actual number of iterations for all classes. If binary or multinomial, it returns only 1 element. For liblinear solver, only the maximum number of iteration across all classes is given. |

**5)Naive Bayes Classifier**

### In machine learning, *naive Bayes classifiers* are a family of simple probabilistic classifiers based on applying Bayes' theorem with strong (naive) independence assumptions between the features. Naive Bayes classifiers are highly scalable, requiring a number of parameters linear in the number of variables (features/predictors) in a learning problem. Maximum-likelihood training can be done by evaluating a closed-form expression, which takes linear time, rather than by expensive iterative approximation as used for many other types of classifiers. Gaussian naive Bayes

When dealing with continuous data, a typical assumption is that the continuous values associated with each class are distributed according to a Gaussian distribution. For example, suppose the training data contains a continuous attribute. Another common technique for handling continuous values is to use binning to discretize the feature values, to obtain a new set of Bernoulli-distributed features; some literature in fact suggests that this is necessary to apply naïve. GaussianNB implements the Gaussian Naive Bayes algorithm for classification.

**Attributes**

|  |  |  |
| --- | --- | --- |
| *class\_prior\_* | array, shape = [n\_classes] | probability of each class. |
| *theta\_* | array, shape = [n\_classes, n\_features] | mean of each feature per class |
| *sigma\_* | array, shape = [n\_classes, n\_features] | variance of each feature per class |

**Methods**

|  |  |
| --- | --- |
| fit(X, y) | Fit Gaussian Naive Bayes according to X, y |
| get\_params([deep]) | Get parameters for the estimator |
| predict(X) | Perform classification on an array of test vectors X. |
| predict\_log\_proba(X) | Return log-probability estimates for the test vector X. |
| predict\_proba(X) | Return probability estimates for the test vector X. |
| score(X, y) | Returns the mean accuracy on the given test data and labels. |
| set\_params(\*\*params) | Set the parameters of the estimator. |

**Model Evaluation Basics**

The formula for **accuracy** is:   
A(M) = (TN+TP)/(TN+TP+FP+FN)

**TP** is the number of true positives : Predicted as churned and they do

**TN** is the number of true negatives: Predicted as remaining and they do

**FP** is the number of false positives: Predicted as they will churn, but they don’t

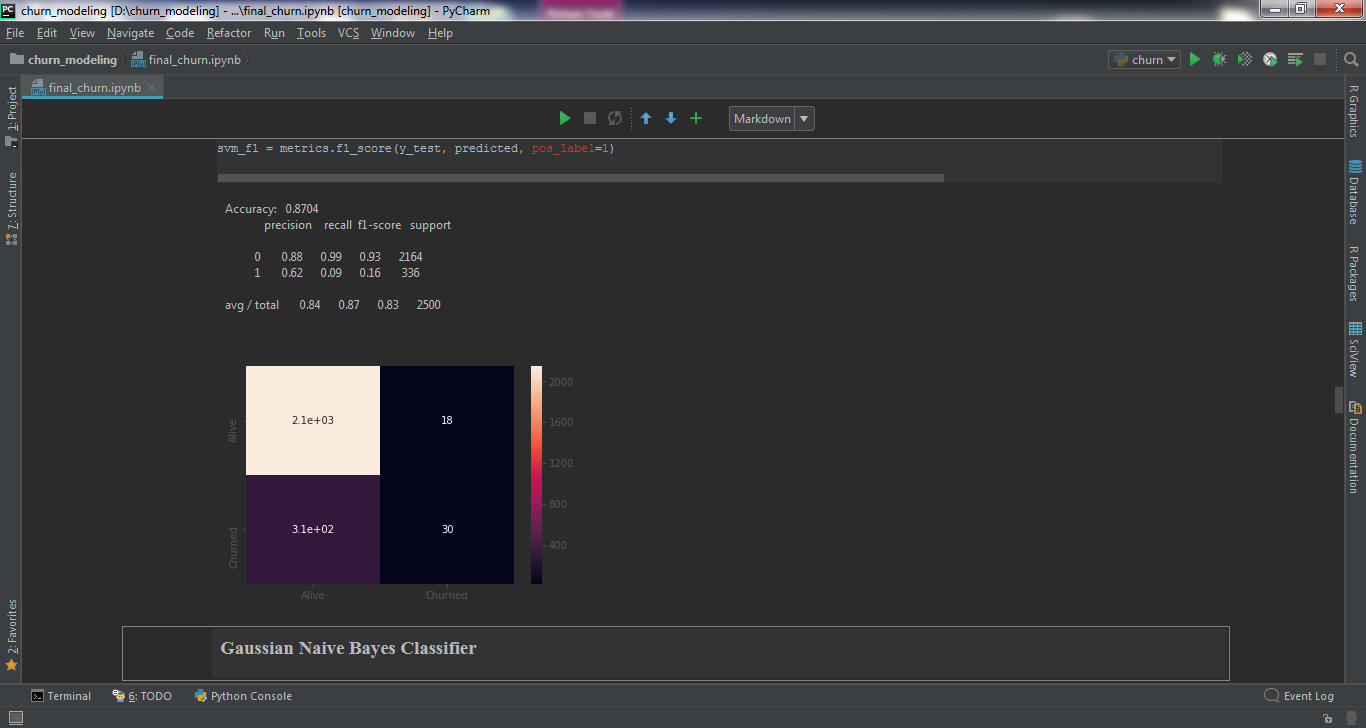
**FN** is the number of false negatives: Predicted as won’t churner but they churn

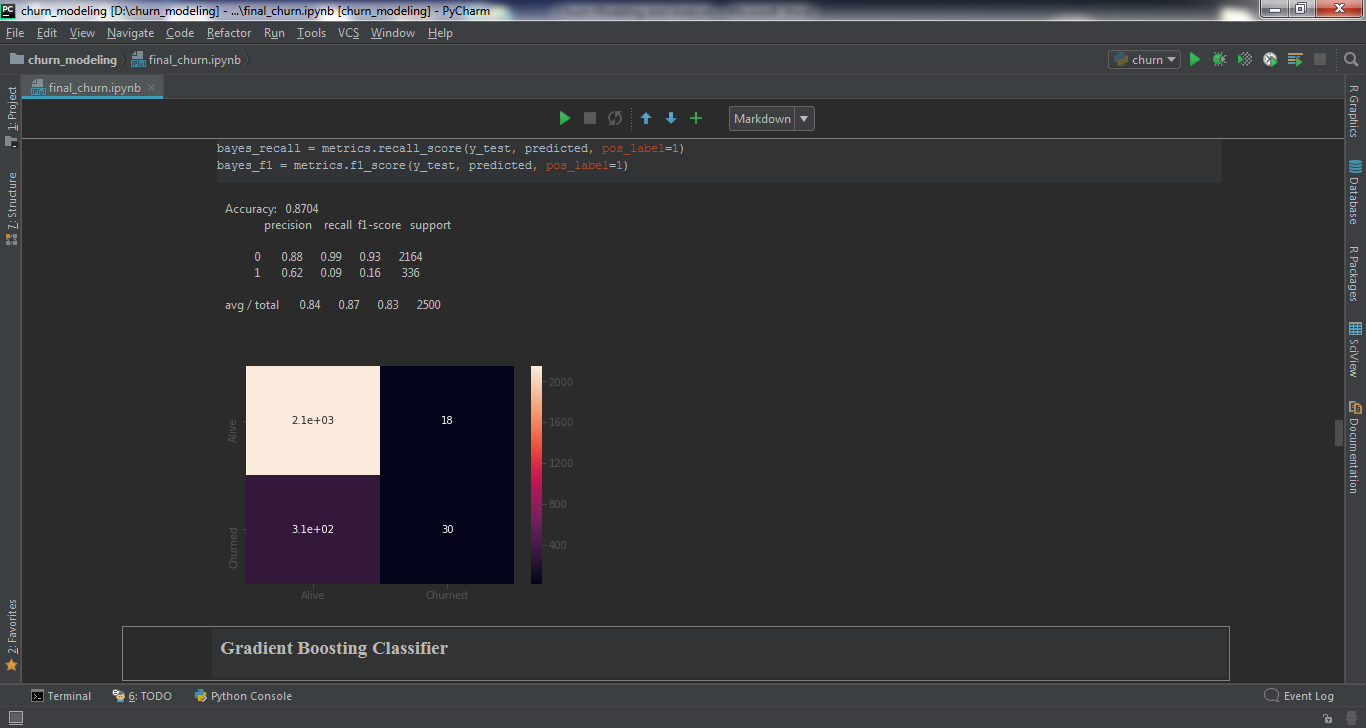
When working with unbalanced dataset we can’t completely depend on Accuracy alone, we need to check the result of other matrix along with accuracy and they are:

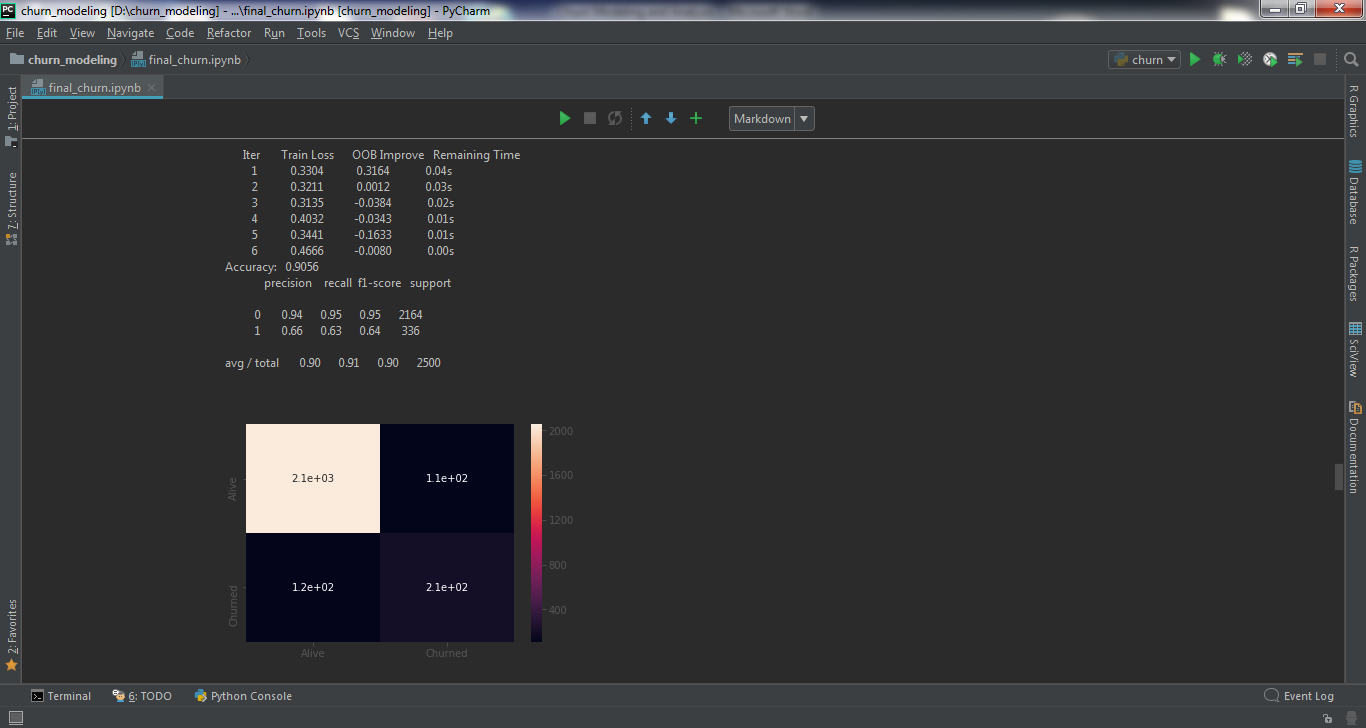
* **Sensitivity/Recall** - TP / (TP + FN) how well the model recalls/identifies those that will leave. AKA the true positive rate
* **Specificity** - TN / (TN + FP) how well the model identifies those that will stay.
* **Precision** TP / (TP + FP) how believable is the model? A low precision model will alarm you to those who are leaving that are actually staying.
* **F1 score** 2 \* (precision \* recall)/(precision + recall)is the harmonic mean between precision and recall or the balance.

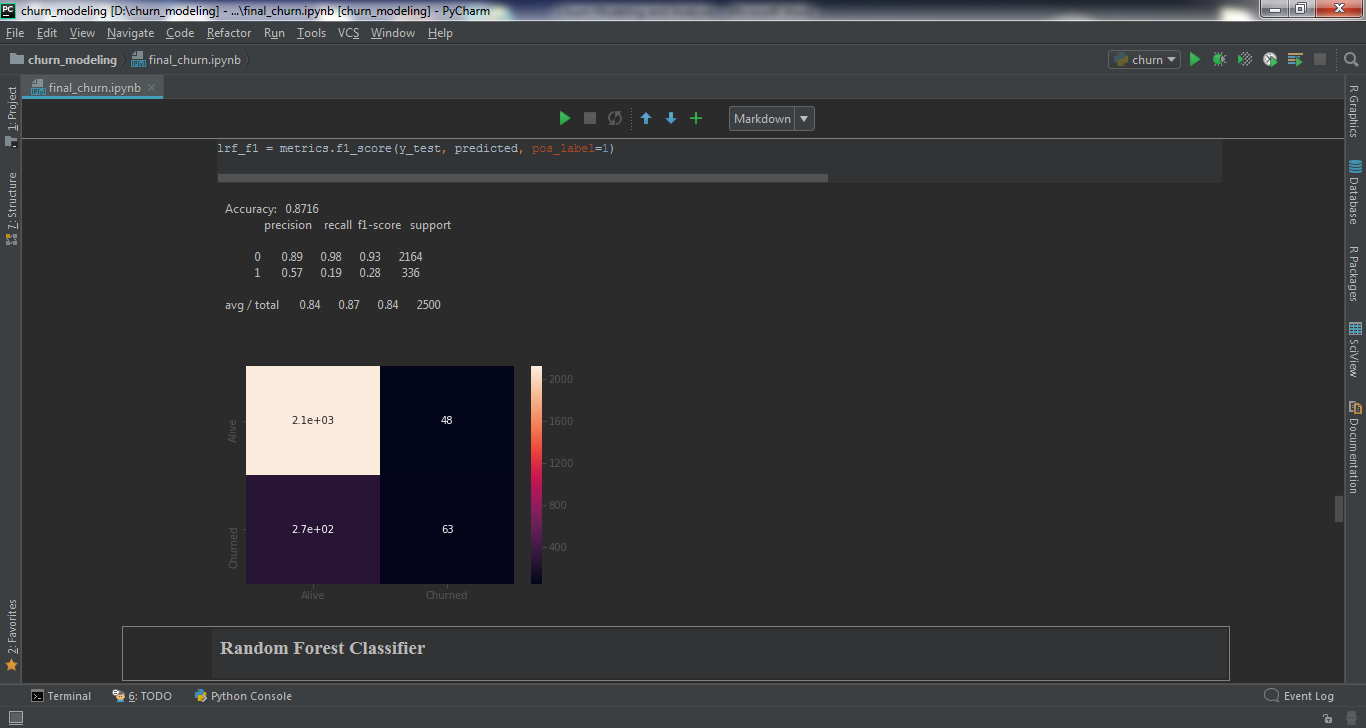
**Results of Models**

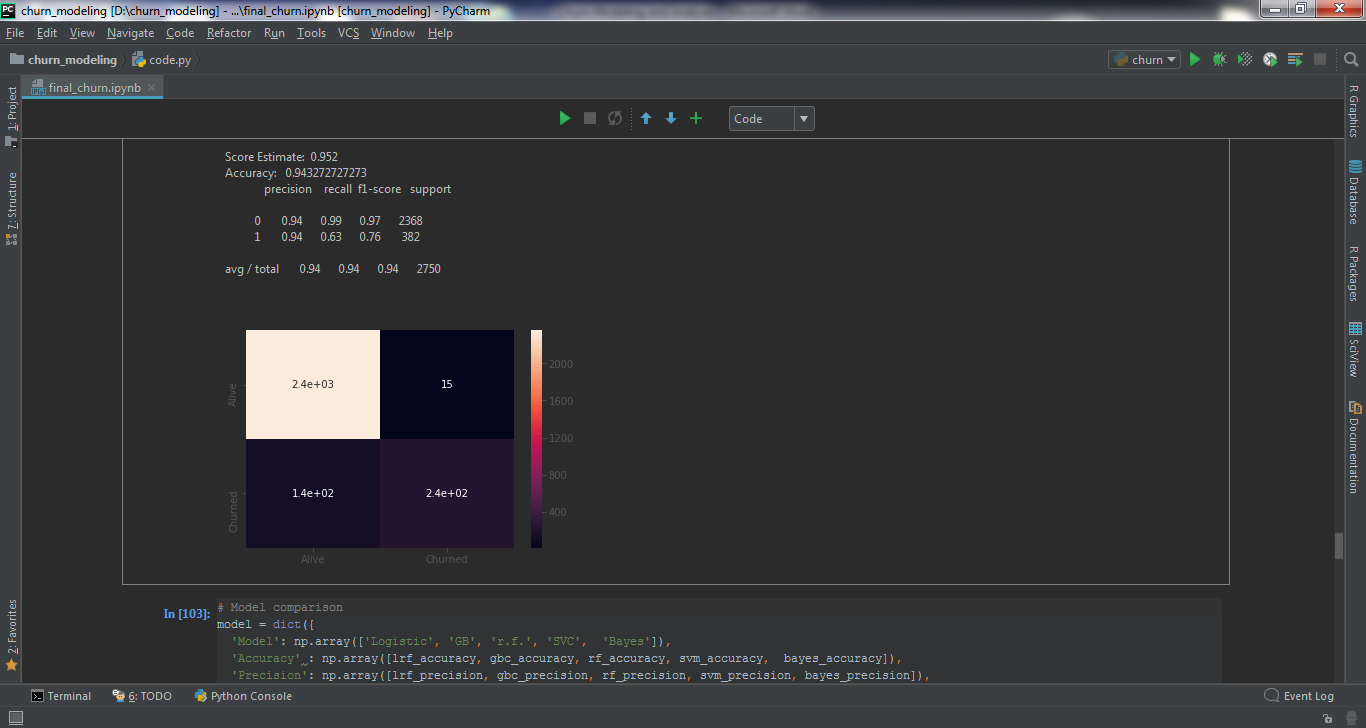
**As our data set id unbalanced so we can’t only relay on accuracy every time. So let’s check the values of precision, recall, F1-score for model evaluation.**

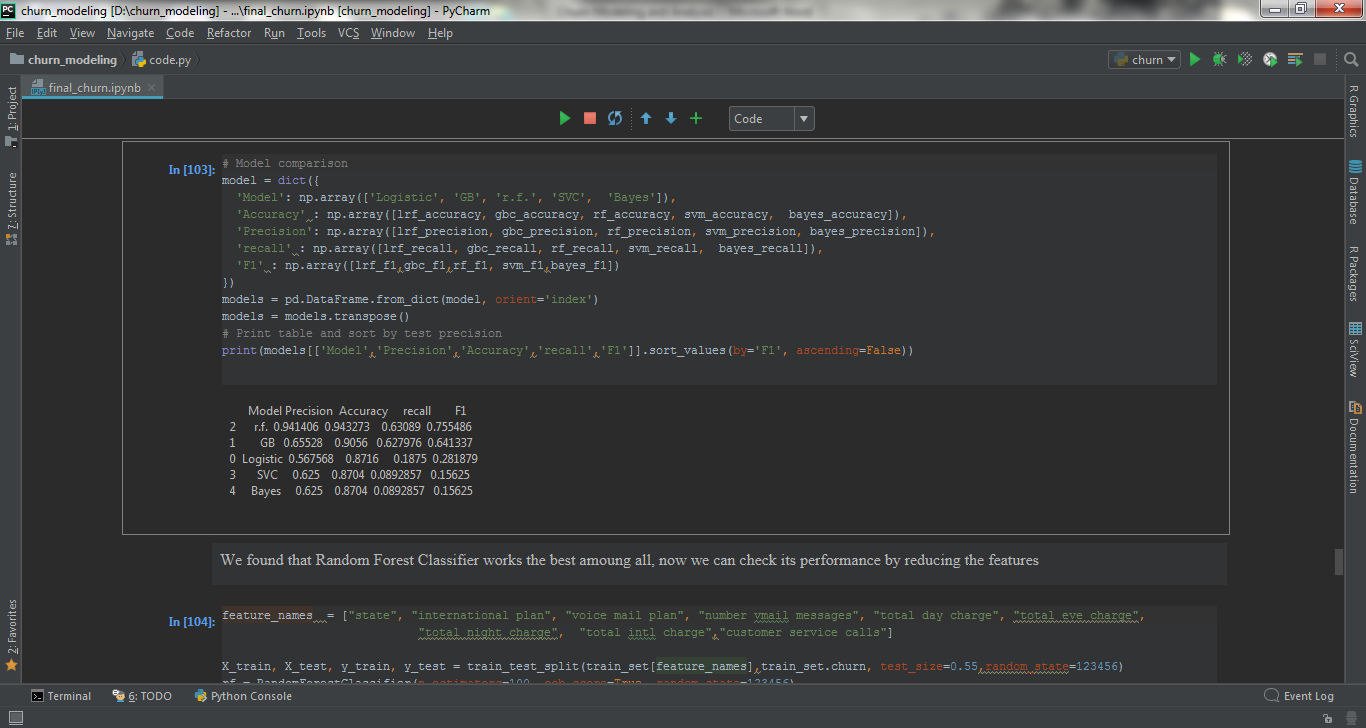
 **Linear SVC**

 **Naive Bayes Classifier**

 **Gradient Boosting Classifier**

 **Logistic Regression**

 **Random Forest Classifier**



**Models performance matrix**

**We can say that from all the models Random Forest Classifier works well with the above matrix.**

**Predict the result**

Once we have trained model we can use the same model to predict the churn and we have required solution for the given problem.

**B) When to use which type of models: Tips and Advice.**

**There is no fixed approach which can be used for churn modeling. As type of model and its parameters all depend on the Features and relationship of the data.**

**Mostly used models are**

**Random Forest Classifier:**

This model is best in most of the cases as it takes care of model overfitting which make the model bias to trained condition. Also it can easily balance the features by assigning weight to the branches. Random forest runtimes are quite fast, and they are able to deal with unbalanced and missing data.

Generally use when:

* + 1. Have missing data
    2. Training data set size is too large.
    3. Have unbalance data set
    4. Overfitting is major concerned.

**Logistic Regression**

Logistic regression is used to predict the odds of being a case based on the values of the independent variables (predictors). The odds are defined as the probability that a particular outcome is a case divided by the probability that it is a non-case. Thismodel is best when we can see log graph like pattern in relationship among data.

**Gradient Boosting Classifier**

This is also quite efficient as random forest and takes care of Unbalanced data structure. When the complexity of relationship increased Gradient Boosting Classifier works much better than others.

**Naive Bayes Classifier**

This is a simple classifier and works when the complexity is low with simple dataset. This is suitable for small sized dataset. This model do have issue with overfitting so not much used for real time solutions but all depends on type and structure of the data.