Applied Machine Learning in Python
Module 4. Supervised Machine Learning — Part 2

Kevyn Collins-Thompson
University of Michigan

Contents

1 Naive Bayes Classifiers

Another family of supervised learning models that's related to linear classification models is the Naive Bayes family of classifiers, which are based on simple *probabilistic* models of how the data in each class might have been generated.

Naive Bayes classifiers are called naive because informally, they make the *simplifying assumption* that each feature of an instance is independent of all the others, given the class.

In practice, of course, this is not often the case, features often are somewhat correlated. For example, in predicting whether a house is likely to sell above the owner's asking price. Some features, such as the are of the interior rooms are likely to be correlated with other features, such as the size of the land that the house is built on or the number of bedrooms. And these features in turn might be correlated with the location of the property, and so on.

This naive simplifying assumption means on the one hand, that learning a Naive Bayes classifier is *very fast*. Because only simple per class statistics need to be estimated for each feature and applied for each feature independently.

On the other hand, the penalty for this efficiency is that the *generalization performance* of Naive Bayes Classifiers can often be a bit worse than other more sophisticated methods, or even linear models for classification.

Even so, especially for high dimensional data sets, Naive Bayes Classifiers can achieve *performance* that's often competitive to other more sophisticated methods, like support vector machines, for some tasks.

There are three flavors of Naive Bayes Classifier that are available in scikit learn.

1.1 Bernoulli Naive Bayes model

The Bernoulli Naive Bayes model uses a set of *binary* occurrence features. When classifying texts document for example, the Bernoulli Naive Bayes model is quit handy because we could represent the presence or the absence of the given word in the text with the binary feature.

Of course this doesn't take into account how often the word occurs in the text.

1.2 Multinomial Naive Bayes model

So the Multinomial Naive Bayes model uses a set of count base *discrete* features each of which does account for how many times a particular feature such as a word is observed in training example like a document.

In this lecture we won't have time to cover the Bernoulli or Multinomial Naive Bayes models. However, those models are particularly well suited to textual data, where each feature corresponds to an observation for a particular word. And so you'll see Naive Bayes again, including the Bernoulli and Multinomial models in more depth in the text mining part of this specialization.

1.3 Gaussian Naive Bayes model

This lecture will focus on Gaussian Naive Bayes classifiers which assume features that are *continuous or real-valued*. During training, the Gaussian Naive Bayes Classifier estimates for each feature the `mean` and `standard deviation` of the feature value for each class.

For prediction, the classifier compares the features of the example data point to be predicted with the feature statistics for each class and selects the class that best matches the data point.

More specifically, the Gaussian Naive Bayes Classifier assumes that the data for each class was generated by a simple class specific *Gaussian distribution*.

Predicting the class of a new data point corresponds mathematically to estimating the probability that each classes Gaussian distribution was most likely to have generated the data point. Classifier then picks the class that has the highest probability.

Without going into the mathematics involved, it can be shown that the decision boundary between classes in the two class Gaussian Naive Bayes Classifier. In general is a *parabolic* curve between the classes.

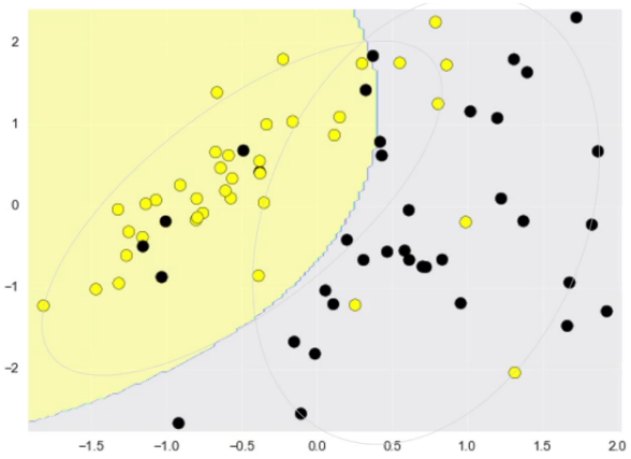
And in the special case where the variance of these feature is the same for both classes. The decision boundary will be linear.

Here's what that looks like, typically, on a simple binary classification data set.

The gray ellipses given idea of the shape of the Gaussian distribution for each class, as if we were looking down from above.

You can see the centers of the Gaussian's correspond to the mean value of each feature for each class.

More specifically, the gray ellipses show the contour line of the Gaussian distribution for each class, that corresponds to about two standard deviations from the mean.



The line between the yellow and gray background areas represents the decision boundary. And we can see that this is indeed parabolic.

To use the Gaussian Naive Bayes classifier in Python, we just instantiate an instance of the Gaussian NB class and call the fit method on the training data just as we would with any other classifier.

```
from sklearn.naive_bayes import GaussianNB
```

Note: the Naive Bayes models are among a few classifiers in scikit learn that support a method called `partial_fit`, which can be used instead of `fit` to train the classifier *incrementally* in case you're working with a huge data set that doesn't fit into memory.

For the `GaussianNB` class there are no special parameters to control the models complexity.

Looking at one example in the notebook from our synthetic two class dataset, we can see that, in fact, the Gaussian Naive Bayes classifier achieves quite good performance on this simple classification example. When the classes are no longer as easily separable as with this second, more difficult binary example here. Like linear models, Naive Bayes does not perform as well.

On a real world example, using the breast cancer data set, the Gaussian Naive Bayes Classifier also does quite well, being quite competitive with other methods, such as support vector classifiers.

```
X_train, X_test, y_train, y_test =
train_test_split(X_cancer, y_cancer, random_state = 0)

nbclf = GaussianNB().fit(X_train, y_train)
print('Breast cancer dataset')
print('Accuracy of GaussianNB classifier on training
  set: {:.2f}'.format(nbclf.score(X_train, y_train)))
```

```
print('Accuracy of GaussianNB classifier on test set:
 {:.2f}'.format(nbclf.score(X_test, y_test)))

Breast cancer dataset
Accuracy of GaussianNB classifier on training set: 0.95
Accuracy of GaussianNB classifier on test set: 0.94
```

Typically, Gaussian Naive Bayes is used for *high-dimensional* data, when each data instance has hundreds, thousands or maybe even more features. Likewise the Bernoulli and Nultinomial flavors of Naive Bayes are used for text classification where there are very large number of distinct words is features and where the future vectors are sparse because any given document uses only a small fraction of the overall vocabulary.

The Naive Bayes Classifiers are related mathematically to linear models, so many of the pros and cons of linear models also apply to Naive Bayes.

On the **positive** side Naive Bayes classifiers are:

- easy to understand;
- fast to train and use for prediction;
- well suitable to high dimensional data including text and the applications involving very large data sets, where efficiency is critical and computational costs rule out other classification approaches;
- often useful as a baseline comparison against more sophisticated methods.

On the **negative** side:

- assumption that features are conditionally independence are not realistic; for many real world datasets there's significant covariance among features;
- other classifier types often have better generalization performance;
- confidence estimates for predictions are not very accurate.

Other more sophisticated classification methods that can account for these dependencies are likely to outperform Naive Bayes.

And on a side note, when getting confidence or probability estimates associated with predictions, Naive Bayes classifiers produce unreliable estimates, typically.

Still, Naive Bayes Classifiers can perform very competitively on some tasks, and are also often very useful as baseline models against which more sophisticated models can be compared.

2 Random Forests

A widely used and effective method in machine learning involves creating learning models known as *ensembles*. An ensemble takes multiple individual learning models and combines them to produce an aggregate model that is more powerful than any of its individual learning models alone.

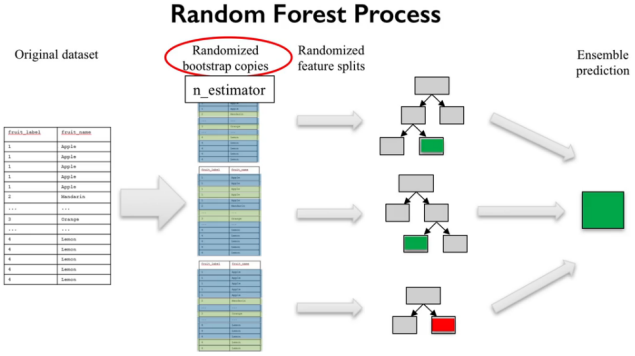
Why are ensembles effective? Well, one reason is that if we have different learning models, although each of them might perform well individually, they'll tend to make different kinds of mistakes on the data set. And typically, this happens because each individual model might overfit to a different part of the data. By combining different individual models into an ensemble, we can average out their individual mistakes to reduce the risk of overfitting while maintaining strong prediction performance.

Random forests are an example of the ensemble idea applied to decision trees. Random forests are widely used in practice and achieve very good results on a wide variety of problems. They can be used as classifiers via the sklearn `RandomForestClassifier` class or for regression using the `RandomForestRegressor` class both in the `sklearn.ensemble` module.

As we saw earlier, one disadvantage of using a *single* decision tree was that decision trees tend to be prone to overfitting the training data.

As its name would suggest, a random forest creates lots of individual decision trees on a training set, often on the order of tens or hundreds of trees. The idea is that each of the individual trees in a random forest should do reasonably well at predicting the target values in the training set but should also be constructed to be different in some way from the other trees in the forest.

Again, as the name would suggest this difference is accomplished by introducing random variation into the process of building each decision tree.

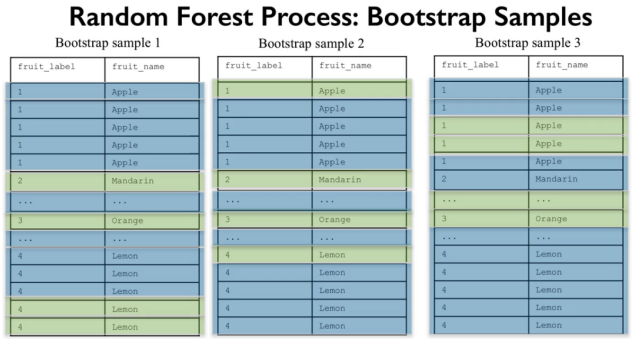


This random variation during tree building happens in two ways. First, the data used to build

each tree is selected randomly and second, the features chosen in each split tests are also randomly selected.

2.1 Random Forests Model Creation

To create a random forest model you first decide on how many trees to build. This is set using the `n_estimated` parameter for both RandomForestClassifier and RandomForestRegressor. Each tree were built from a different random sample of the data called the bootstrap sample. Bootstrap samples are commonly used in statistics and machine learning. If your training set has N instances or samples in total, a bootstrap sample of size N is created by just repeatedly picking one of the N dataset rows at random with replacement, that is, allowing for the possibility of picking the same row again at each selection.



You repeat this random selection process N times. The resulting bootstrap sample has N rows just like the original training set but with possibly some rows from the original dataset missing and others occurring multiple times just due to the nature of the random selection with replacement.

When building a decision tree for a random forest, the process is almost the same as for a standard decision tree but with one important *difference*. When picking the best split for a node, instead of finding the best split across all possible features, a *random subset of features* is chosen and the best split is found within that smaller subset of features. The number of features in the subset that are randomly considered at each stage is controlled by the `max_features` parameter.

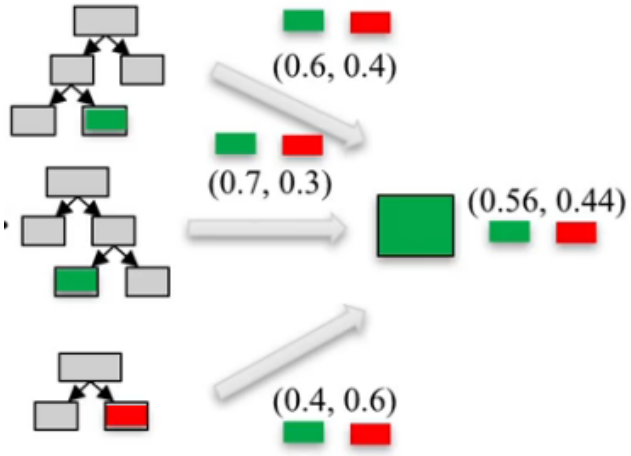
This randomness in selecting the bootstrap sample to train an individual tree in a forest ensemble, combined with the fact that splitting a node in the tree is restricted to random subsets of the features of the split, virtually guarantees that all of the decision trees and the random forest will be different.

The random forest model is quite sensitive to the `max_features` parameter. If `max_features = 1`, the random forest is limited to performing a split on the single feature that was selected randomly instead of being able to take the best split over several variables. This means the trees in the forest will likely be very different from each other and possibly with many levels in order to produce a good fit to the data.

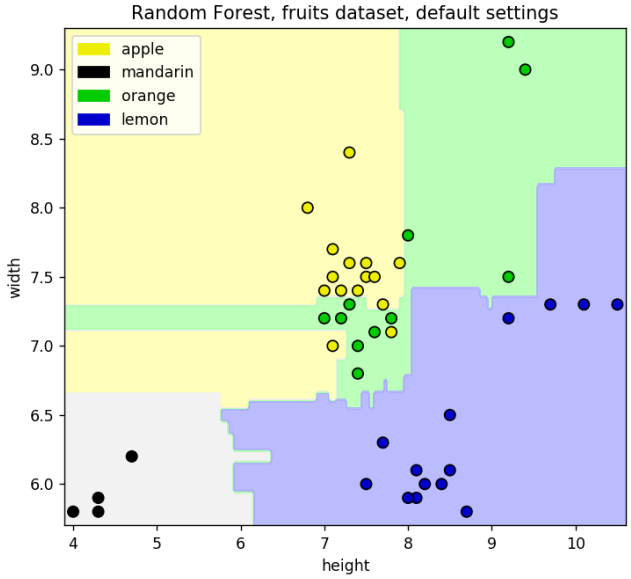
On the other hand if `max_features` is high, close to the total number of features that each instance has, the trees in the forest will tend to be similar and probably will require fewer levels to fit the data using the most informative features.

2.2 Prediction

Once a random forest model is trained, it predicts the target value for new instances by first making a prediction for every tree in the random forest.



For *regression* tasks the overall prediction is then typically the **mean** of the individual tree predictions. For *classification* the overall prediction is based on a **weighted vote**. Each tree gives a probability for each possible target class label then the probabilities for each class are averaged across all the trees and the class with the highest probability is the final predicted class.



Here's an example of learning a random forest of the example fruit dataset using two features, height and width. Here we're showing the training data plotted in terms of two feature values with height on the x axis and width on the y axis.

As usual, there are four categories of fruit to be predicted. Because the number of features is restricted to just two in this very simple example, the randomness in creating the tree ensemble is coming mostly from the bootstrap sampling of the training data. You can see that the decision boundaries overall have the *box like shape* that we associate with decision trees but with some additional detail variation to accommodate specific local changes in the training data.

Overall, you can get an impression of the increased complexity of this random forest model in capturing both the global and local patterns in the training data compared to the single decision tree model we saw earlier.

2.3 Implementation

Fruit dataset

Let's take a look at the notebook code that created and visualized this random forest on the fruit dataset.

```
from sklearn.ensemble import RandomForestClassifier
...
clf = RandomForestClassifier().fit(X, y)
clf = RandomForestClassifier(n_estimators = 10,
      random_state=0).fit(X_train, y_train)
...
Random Forest, Fruit dataset, default settings
Accuracy of RF classifier on training set: 1.00
Accuracy of RF classifier on test set: 0.80
```

To use the `RandomForestClassifier` we import the random forest classifier class from the `sklearn.ensemble` library.

For each pair of features we call the fit method on that subset of the training data X using the labels y. We then use the utility function plot class regions for classifier to visualize the training data and the random forest decision boundaries.

Let's apply random forest to a larger dataset with more features.

Breast Cancer dataset

For comparison with other supervised learning methods, we use the breast cancer dataset. We create a new random forest classifier and since there are about 30 features, we'll set `max_features = 8` to give a diverse set of trees that also fit the data reasonably well.

```
Breast cancer dataset
Accuracy of RF classifier on training set: 1.00
Accuracy of RF classifier on test set: 0.99
```

We can see that random forest with no feature scaling or extensive parameter tuning achieve very good test set performance on this dataset, in fact, it's as good or better than all the other supervised methods we've seen so far including current life support vector machines and neural networks that require more careful tuning.

Notice that we did not have to perform scaling or other pre-processing as we did with a number of other supervised learning methods. This is one *advantage* of using random forests.

2.4 Pros and Cons

On the **positive** side of Random Forests:

- widely used, excellent prediction performance on many problems;
- doesn't require careful normalization of features or extensive paramener tuning;
- like decision trees, handles a mixture of feature types;
- easily parallelized across multiple CPUs.

Even though building many different trees requires a corresponding increase in computation, building random forests is easily parallelized across multiple CPU's.

On the **negative** side:

- the resulting modela are often difficult for humans to interpret — difficult to see the predictive structure of the features or to know why a particular prediction was made;

- not a good choice for tasks that have very high dimensional sparse features like text classification.

2.5 Key Parameters

Here are some of the key parameters that you'll need for using random forests.

`N_estimators` sets the number of trees to use. The default value for `n_estimators` = 10 and increasing this number for larger data sets is almost certainly a good idea since ensembles that can average over more trees will reduce overfitting.

Just bear in mind that increasing the number of trees in the model will also increase the computational cost of training. You'll use more time and more memory. So in practice you'll want to choose the parameters that make best use of the resources available on your system.

The `max_features` parameter has a strong effect on performance. It has a large influence on how diverse the random trees in the forest are. Typically, the default setting of `max_features`, which for classification is the square root of the total number of features and for regression is the log base two of the total number of features, works quite well in practice although explicitly adjusting `max_features` may give you some additional performance gain with smaller values of max features tending to reduce overfitting.

The `max_depth` parameter controls the depth of each tree in the ensemble. The default setting for this is none, in other words, the nodes in a tree will continue to be split until all leaves contain the same class or have fewer samples than the minimum sample split parameter value, which is two by default.

The `n_jobs` parameter — how many cores to use in parallel to train the model. Generally, you can expect something close to a linear speed up. So, for example, if you have four cores, the training will be four times as fast as if you just used one. If you set end jobs to negative one it will use all the cores on your system and setting end jobs to a number that's more than the number of cores on your system won't have any additional effect.

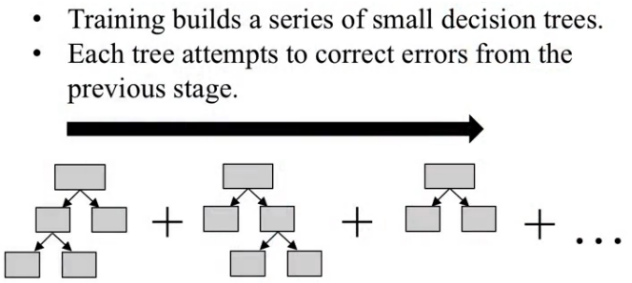
3 Gradient Boosted Decision Trees

Another tree based ensemble method that's gain wide use in real world application is gradient boosted decision trees — GBDT.

Like random forest, gradient boosted trees used an ensemble of multiple tress to create more powerful prediction models for classification and regression.

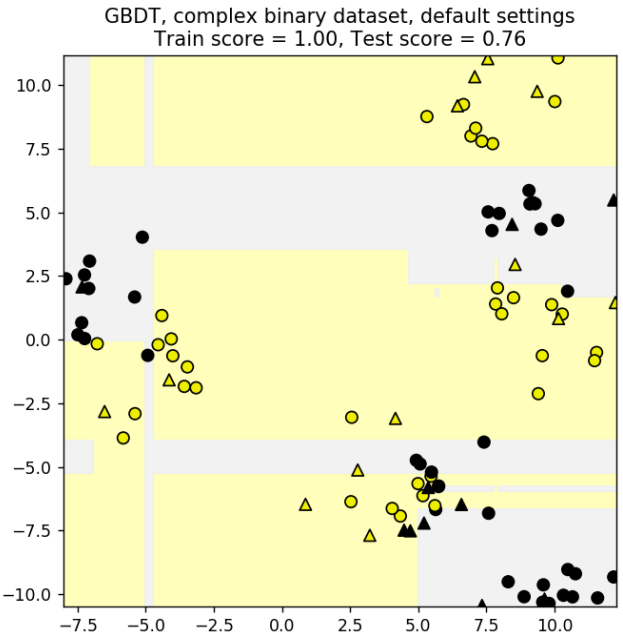
In this lecture, we'll provide a brief overview of gradient boosted decision trees, along with the discussion of their key parameters, the control model complexity.

Unlike the random forest method that builds and combines a forest of randomly different trees in parallel, the key idea of gradient boosted decision trees is that they build a *series* of trees. Where each tree is trained, so that it attempts to correct the mistakes of the previous tree in the series.



Typically, gradient boosted tree ensembles use lots of shallow trees known in machine learning as *weak learners*. Built in a *non-random* way, to create a model that makes fewer and fewer mistakes as more trees are added.

Once the model is built, making predictions with a gradient boosted tree models is fast and doesn't use a lot of memory.



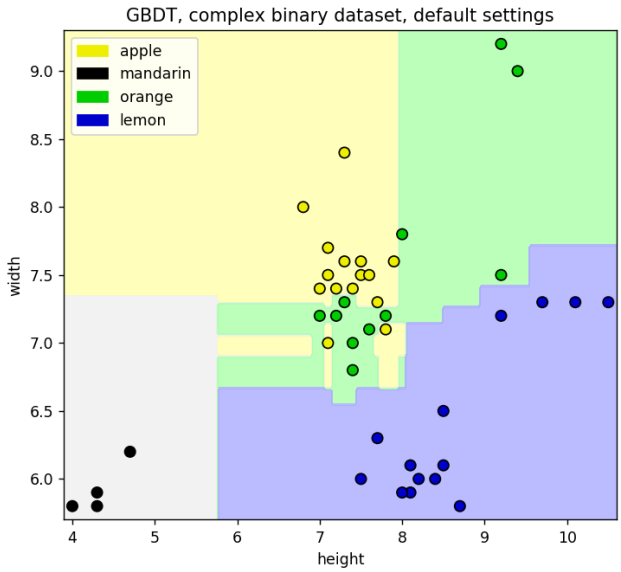
Like random forests, the `number of estimators` in the gradient boosted tree ensemble is an important parameter in controlling model complexity.

A new parameter that does not occur with random forest is something called the `learning rate`. The learning rate controls how the gradient boost the tree algorithms builds a series of collective trees. When the learning rate is *high*, each successive tree puts strong emphases on correcting the mistakes of its predecessor, and thus may result in a more complex individual tree, and those overall are more complex model. With *smaller* settings of the learning rate, there's less emphasis on thoroughly correcting the errors of the previous step, which tends to lead to simpler trees at each step.

```
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test =
    train_test_split(X_D2, y_D2, random_state = 0)

clf = GradientBoostingClassifier().fit(X_train, y_train)
...
GBDT, Fruit dataset, default settings
Accuracy of GBDT classifier on training set: 1.00
Accuracy of GBDT classifier on test set: 0.80
```



Here's an example showing how to use gradient boosted trees in scikit-learn on our sample fruit classification test, plotting the decision regions that result. The code is more or less the same as what we used for random forests.

But from the sklearn.ensemble module, we import the `GradientBoostingClassifier` class. We then create the `GradientBoostingClassifier` object, and fit it to the training data in the usual way. By default, the `learning rate` parameter is set to **0.1**, the `n_estimators` parameter giving the number of trees to use is set to **100**, and the **max_depth** is set to **3**. As with random forests, you can see the decision boundaries have that box-like shape that's characteristic of decision trees or ensembles of trees.

Now let's apply gradient boosted decision trees to the breast cancer dataset.

This code trains two different gradient boosted classifiers. The first one uses the default settings. We can see that the first result has perfect accuracy on the training set, which indicates the model is likely overfitting.

Two ways to learn a less complex gradient boosted tree model are, to reduce the learning rate, so that each tree doesn't try as hard to learn a more complex model, that fixes the mistakes of its predecessor. And to reduce the max_depth parameter for the individual trees in the ensemble.

The second classifier example makes these changes in the parameters. And you can see, that the training set accuracy does decrease, while the test set accuracy increases slightly.

Gradient boosted decision trees are among the best off-the-shelf supervised learning methods available. Achieving excellent accuracy with only modest memory and runtime requirements to perform prediction, once the model has been trained.

Some major commercial applications of machine learning have been based on gradient boosted decision trees.

Like other decision tree based learning methods, you don't need to apply feature scaling for the algorithm to do well. And the futures can be a mix of binary, categorical and continuous types.

Boosted decision trees do have several downsides. So like random forests, ensembles of trees are very difficult for people to interpret, compared to individual decision trees. However, this often may not matter for many applications where prediction accuracy is the most important goal.

Gradient boosted methods can require careful tuning of the learning rate and other parameters. And the training process can require a lot of computation.

And like the other tree based methods we saw, using gradient boosted methods for text classification or other scenarios. Where the featured space has thousands of features with sparse values, is usually not a good choice for accuracy and computational cost reasons.

The key parameters controlling model complexity for gradient boosted tree models are, n_estimators which sets the number of small decisions trees the week learns to use in the ensemble, and the learning rate.

Typically, these two parameters are tuned together. Since making the learning rates smaller, will require more trees to maintain model complexity.

Unlike random forest, increasing an n_estimators can lead to overfeeding. So typically, the n_estimators setting is chosen to best exploit the speed and memory capabilities of the system during the training. And other parameters like the learning rate are then adjusted, given that fixed an n_estimators setting.

The max_depth parameter can also have an effect of model complexity, but controlling the depth, and has a complexity of the individual trees. The gradient boosting method assumes, that each trees is a weak learner, and so the max_depth parameter is usually quite small, on the order of three to five, for most applications.