After we use linear regression to fit our Boston housing Data and obtain the result (MSE and r square) as the benchmark for our project, we use neural network. However, there is an obvious problem that the amount of our data is not large enough, which may cause underfitting and return a bad result. Therefore, we try to use ensemble method to strength our learner to get a good result.

Ensemble methods aims to combine the predictions of several base estimators built with a given learning algorithm in order to improve generalizability / robustness over a single estimator. These base estimators, which called weak learner, generally do not have a good performance on a given data. But if they have strong dependency and we can use this dependency to update the weight, for example, Strengthen the learning weight of misclassified data or they don't have strong dependency and we can combine them, we can improve these weak learner to a strong learner. Boosting is the representative method of the former, and the latter is bagging method and random forest. They are two broad categories of ensemble methods.

In our project, we will try these two methods: Gradient Boosting and Random forest to improve the result. Actually, it turns out that with a small sample of data, the performance of ensemble methods is better

than our traditional machine learning models.

3.1 Random forest

Before we talk about random forest, we need to understand the core of this method—— resampling technique and randomly chooseing feature.

The first technique we also call it bootstrap methods:

Now Given data set D containing m samples, randomly take a sample from D with replacement into set D’ and sampled for m times to get the data set D’. Then, D’ have m samples.It's clear that some of the samples in D will show up multiple times, and some of the samples won't.

So we estimate the probability that the sample is never picked in m samples. It will be (1-1/m)^m. set m->∞:

$$\lim_{m \mapsto \infty} \left(1 - \frac{1}{m}\right)^m \mapsto \frac{1}{e} \approx 0.368$$

Which means there are about 36.8% samples in D will never show up in D’.

This is bootstrap, which we used to build our resampling data in random forest.

The second variant on random forest is when we choose a feature to classify our data on our base estimator, traditional decision tree will calculate all of information gain and choose the biggest one. In random forest, we first randomly choose k features from all of the features, and

use these k features to calculate information gain. This change improve the independency of each tree, which optimized the model. Generally, we will set k=log2 d, where d is the number of all features.

Now, let's summarize the algorithm of random forest:

First, use bootstrap build m number of subset.

For each subset, train a decision tree or regression tree as base estimator.

In each base estimator, randomly select “K” features from total “m” features where k << m

Among the “K” features, calculate the node “d” using the best split point

Split the node into daughter nodes using the best split

Build forest by training “m” number of trees.

3.2 Boosting

Boosting is a very powerful learning method, which is also a supervised categorization learning method.It combines many weak learners to produce a strong learner.The performance of a weak classifier is only slightly better than that of random selection, so it can be designed to be very simple and without too much computational cost.Many weak classifiers are combined to form an integrated strong classifier similar to SVM or neural network.

Now the question is how to use boosting to build a rule? We can do it in the following steps:

Step 1: The basic learner should have the same weight for each observation

Step 2: If the first basic learning algorithm makes a wrong prediction, the point has a higher weight in the next basic learning algorithm

Step 3: Iterate step 2 until a predetermined number of learners or a predetermined prediction accuracy is reached.

Finally, the output of a number of weak learner combined into a strong learner, improve the overall prediction accuracy of the model.Boosting is always more concerned with the weak rules that are misclassified.

Here, a famous algorithm of boosting method is Adaboost.

If we have a classification problem, given a training sample set, it is often much easier to derive a rough classification rule (weak classifier) from this classification problem than an accurate classification rule (strong classifier).The promotion method is to get a series of weak classifiers (also known as basic classifiers) by repeated learning from the weak learning algorithm, and then form a strong classifier by combining these weak classifiers.Most of the promotion methods are to change the probability distribution of training data (or the weight distribution of training data) to call the weak learning algorithm to learn a series of weak classifiers for different training distributions.As a result, there are

two issues that need to be addressed for the promotion approach:

How to change the weight or probability distribution of training data in each round?

How to combine a weak classifier into a strong classifier?

AdaBoost approach to the first problem is to raise the weights of samples that were misclassified by the previous round of weak classifiers and lower the weights of samples that were correctly classified.After a round of weight increase, the weak classifier will pay more attention to the samples that are not properly classified.As it goes on, the classification problem is "divided and ruled" by a series of weak classifiers.For the second problem, namely the combination of weak classifiers, AdaBoost adopts the weighted majority voting method. Specifically, it is to increase the weight of weak classifiers with small error rate so that they can play a greater role in voting; on the other hand, it is to reduce the weight of weak classifiers with large error rate so that they can play a smaller role in voting.

The next is what we used in our problem called Gradient boosting. It’s a generalization of AdaBoost.

Gradient boosting involves three elements:

A loss function to be optimized.

A weak learner to make predictions.

An additive model to add weak learners to minimize the loss function.

1. Loss Function

The loss function used depends on the type of problem being solved.

It must be differentiable, but many standard loss functions are supported and you can define your own.

For example, regression may use a squared error and classification may use logarithmic loss.

A benefit of the gradient boosting framework is that a new boosting algorithm does not have to be derived for each loss function that may want to be used, instead, it is a generic enough framework that any differentiable loss function can be used.

2. Weak Learner

Decision trees are used as the weak learner in gradient boosting.

Specifically regression trees are used that output real values for splits and whose output can be added together, allowing subsequent models outputs to be added and “correct” the residuals in the predictions.

Trees are constructed in a greedy manner, choosing the best split points based on purity scores like Gini or to minimize the loss.

Initially, such as in the case of AdaBoost, very short decision trees were used that only had a single split, called a decision stump. Larger trees can be used generally with 4-to-8 levels.

It is common to constrain the weak learners in specific ways, such as a maximum number of layers, nodes, splits or leaf nodes.

This is to ensure that the learners remain weak, but can still be constructed in a greedy manner.

3. Additive Model

Trees are added one at a time, and existing trees in the model are not changed.

A gradient descent procedure is used to minimize the loss when adding trees.

Traditionally, gradient descent is used to minimize a set of parameters, such as the coefficients in a regression equation or weights in a neural network. After calculating error or loss, the weights are updated to minimize that error.

Instead of parameters, we have weak learner sub-models or more specifically decision trees. After calculating the loss, to perform the gradient descent procedure, we must add a tree to the model that reduces the loss (i.e. follow the gradient). We do this by parameterizing the tree, then modify the parameters of the tree and move in the right direction by (reducing the residual loss.

Generally this approach is called functional gradient descent or gradient descent with functions.

4

After preprocessing, we can directly use the API from sklearn to run our model.

First, we use **from sklearn.model_selection import train_test_split** to split our data with test size as default, random_state=1. Then, use **RandomForestRegressor** and **GradientBoostingRegressor** API separately in **sklearn.ensemble** to run our model.

About the parameter selection, we use **from sklearn.model_selection import GridSearchCV** to choose the best combination of the number of estimators and max_depth.

After training, we will use r square and MSE to judge the performance of our model

5.1 random forest

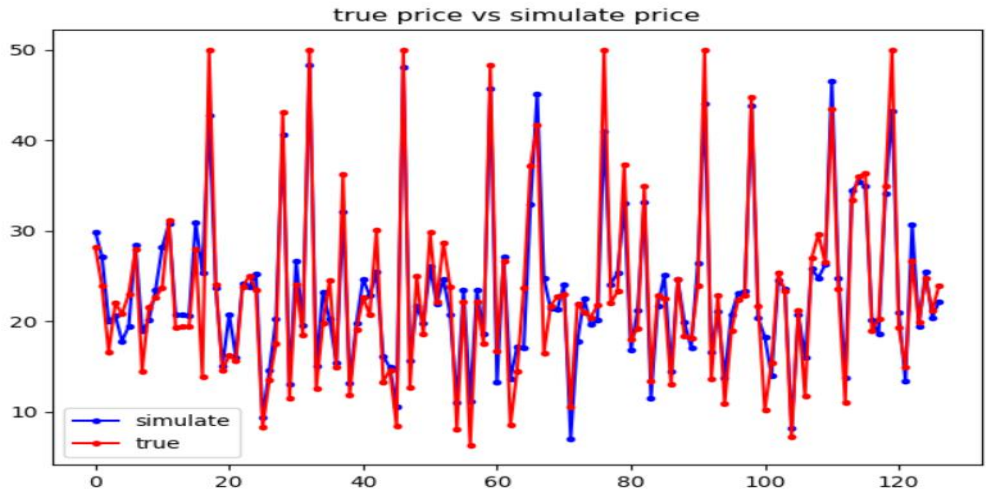
Through **GridSearchCV** we choose the combination of parameter like this:

```
{'max_depth': 10, 'n_estimators': 100}
```

And the Mean square error and R square are:

```
MSE: 9.3766947500555
R square: 0.905341820710926
```

What's more, we plot the true price vs simulate price:



5.2 gradient boosting

Through **GridSearchCV** we choose the combination of parameter like this:

```
{'min_samples_leaf': 3, 'n_estimators': 150}
```

And the Mean square error and R square are:

```
MSE: 9.126177702162598
R square: 0.9078708022194981
```

What's more, we plot the true price vs simulate price: