1. 集成学习
   集成学习(ensemble learning)通过构建并结合多个学习器来完成学习任务,有时也被称为多分类器系统(multi-classifier system)、基于委员会的学习(committee-based learning)等.集成学习通过将多个学习器进行结合，常可获得比单一学习器显著优越的泛化性能.这对“弱学习器”(weak learner)尤为明显，因此集成学习的很多理论研究都是针对弱学习器进行的，而基学习器有时也被直接称为弱学习器，但需注意的是，虽然从理论上来说使用弱学习器集成足以获得好的性能，但在实践中出于种种考虑，例如希望使用较少的个体学习器，或是重用关于常见学习器的一些经验，人们往往会使用比较强的学习器.
   在一般的经验中，如果把好坏不等的东西掺到一起，那么通常结果会是比最坏的要好一些，比最好的要坏一些.那集成学习如何能获得比最好的单一学习器更好的性能呢？
   考虑一个简单的例子：在而分类任务中，假定三个分类器在三个测试样本上的表现如图8.2所示，其中√表示分类正确，×表示分类错误，集成学习的结果通过投票法(voting)产生，即“少数服从多数”.

2. 个体学习器
   个体学习器(individual learner)通常由一个现有的学习算法从训练数据产生,例如C4.5决策树算法、BP神经网络算法等，此时集成中只包含同种类型的个体学习器，例如“决策树集成”中全是决策树，“神经网络集成中”全是神经网络，这样的集成是“同质”的(homogeneous).同质集成中的个体学习器亦称为“基学习器”(base learner),相应的学习算法称为“基学习算法”(base learning algorithm).集成也可包含不同类型的个体学习器，例如同时包含决策树和神经网络，这样的集成是“异质”的(heterogenous.)异质集成中的个体学习器由不同的学习算法生成，这时就不再有基学习算法；相应的个体学习器一般不称为基学习器，常称为“组件学习器”(component learner)或直接称为个体学习器.

3. 为什么集成学习有效? 考虑二分类问题y取值是-1或者1和真实函数f,假定基分类器的错误率为ϵ即对每个基分类器h_i有

$$P(h_i(x) = f(x)) = \epsilon$$

假设集成通过简单投票法结合T个基分类器,若有超过半数的基分类器正确，则集成分类就正确：

$$H(x) = sign(\sum_{i}^{T} h_i(x))$$

假设基分类器的错误率相互独立，则由Hoeffding不等式可知,集成的错误率为

$$\begin{aligned} P(H(x) = f(x)) &= \sum_{k=0}^{[T/2]} \binom{T}{k} (1-\epsilon)^k \epsilon^{T-k} \\ &\leq exp(-\frac{1}{2}T(1-2\epsilon)^2). \end{aligned}$$

上式显示出，随着集成中个体分类器数目T的增大，集成的错误率将指数级下降，最终趋向于0.
然而我们必须注意到，上面的分析有一个关键假设：基学习器的误差相互独立.在现实任务中，个体学习器是为解决同一个问题训练出来的，它们显然不可能相互独立！事实上，个体学习器的“准确性”和“多样性”本身就存在冲突.一般的，准确性很高之后，要增加多样性就需牺牲准确性.事实上，如何产生并结合“好而不同”的个体学习器，恰是集成学习研究的核心. 根据个体学习器的生成方式，目前集成学习方法大致可分为两大类，即个体学习器间存在强依赖关系、必须串行生成的序列化方法，及个体学习器不存在强依赖关系、可同时生成的并行化方法；前者是Boosting,后者的代表是Bagging和“随机森林”(Random Forest). 4. Boosting Boosting是一族可将弱学习器提升为强学习器的算法.这族算法的工作机制类似：先从初始训练数据集训练出一个基学习器，再根据基学习器的表现对训练样本分布进行调整，使得先前基学习器做错的训练样本在后续受到更多关注，然后基于调整后的样本分布来训练下一个基学习器；如此重复进行，直至基学习器数目达到事先指定的值T，最终将这T这个基学习器进行加权结合. Boosting族算法最著名的代表是AdaBoost。 AdaBoost算法有多种推导方式，比较容易理解的是基于“加性模型”，即基学习器的线性组合

$$H(x) = \sum_{t=1}^{T} a_t h_t(x)$$

来最小化指数损失函数

$$l_{exp}(H|D) = E_{x-D}[e^{-f(x)H(x)}]$$

对上式求偏导，并令等式为0可解得

$$H(x) = \frac{1}{2} ln \frac{(P(f(x) = 1|x)}{P(f(x) = -1|x)}$$

因此有 sigh(H(x))=$\arg\max_{y\in\{-1,1\}} P(f(x) = y|x)$ 这意味着sign(H(x))达到了贝叶斯最优错误率.换言之，若指数损失函数最小化，则分类错误率也将最小化；这说明指数损失函数是分类任务原本0/1损失函数的一致的替代损失函数.由于这个替代函数有更好的数学性质，例如它是连续可微函数，因此我们用它替代0/1损失函数作为优化目标 在AdaBoost算法中，第一个基分类器h_1是通过直接将基学习算法用于初始数据分布而得；此后迭代地生成h_t和α_t，当基分类器h_t基于分布D_t产生后，该基分类器的权重α_t应使得$\alpha_t h_t$最小化指数损失函数。 指数损失函数的导数是

$$-e^{-\alpha_t}(1-\epsilon_t) + e^{\alpha_t}\epsilon_t$$

令上式为0，解得

$$\alpha_t = \frac{1}{2} ln(\frac{1-\epsilon_t}{\epsilon_t})$$

这就是分类器权重的更新公式 理想的h_t将在分布D_t下最小化分类误差.因此，弱分类器将基于分布D_t来训练，且针对D_t的分类误差应小于0.5，这在一定程序上类似残差逼近的思想
Boosting算法要求基学习器能对特定的数据分布进行学习，这可通过“重赋权法”实施，即在训练过程的每一轮中，根据样本分布为每个训练样本重新赋予一个权重。对无法接受带权样本的基学习算法，则可通过“重采样法”来处理，即在每一轮学习中，根据样本分布对训练集重新进行采样，再用重采样而得的样本集对基学习器进行训练。一般而言，这两种做法没有显著的优劣差别。需注意的是，Boosting算法在训练的每一轮都要检查当前生成的基学习器是否满足基本条件，是否比随机猜测好，一旦条件不满足，则当前基学习器即被抛弃，且学习过程停止。在此种情形下，初始设置的学习轮数T也许还远未达到，可能导致最终集成中包含很少的基学习器而性能不佳。若采用“重采样法”，则可获得“重启动”机会以避免训练过程过早停止，即在抛弃不满足条件的当前基学习器之后，可根据当前分布重新对训练样本进行采样，再基于新的采样结果重新训练出基学习器，从而使得学习过程可以持续到预设的T轮完成。 从偏差-方差分解的角度看，Boosting主要关注降低偏差，因此Boosting能基于泛化性能相对弱的学习器构建出很强的集成。

5. bagging 由前面可知，欲得到泛化性能强的集成，集成中的个体学习器应尽可能相互独立；虽然“独立”在现实任务中无法做到，但可以设法使基学习器尽可能具有较大的差异，给定一个训练数据集，一种可能的做法是对训练样本进行采样，产生出若干个不能的子集，再从每个数据子集中训练出一个基学习器。这样，由于训练数据不同，我们获得的基学习器可望具有比较大的差异，然而，为获得好的集成，我们同时还希望个体学习器不能太差。如果采样出的每个子集都完全不同，则每个基学习器只用到了一部分训练数据，甚至不足以进行有效学习，这显然无法确保产生比较好的基学习器。未解决这个问题，我们可考虑使用相互有交叠的采样子集。
   Bagging(Boostrap AGGregatING)是并行式集成学习方法最著名的代表。从名字即可看出，它基于自助采样法。给定包含m个样本的数据集，我们先随机取出一个样本放入采样集中，再把该样本放回初始数据集，使得下次采样时该样本仍有可能被选中，这样，经过m次随机采样操作，我们得到含m个样本的采样集，初始训练集中有的样本在采样集类里多次出现，有的则从未出现。 照这样，我们可采样出T个含m个训练样本的采样集，然后基于每个采样集训练出一个基学习器，再将这些基学习器进行结合。这就是Bagging的基本流程。在对预测输出进行结合时，Bagging通常对分类任何使用简单投票法，对回归任务使用简单平均法。若分类预测时出现两个类收到同样票数的情形，则最简单的做法是随机选择一个，也可进一步考察学习器投票的置信度来确定最终胜者。 训练一个Bagging集成与直接使用基学习算法训练一个学习器的复杂度同阶，这说明Bagging是一个很高效的集成学习算法。另外，与标准AdaBoost只适用于二分类任务不同，Bagging能不经修改地用于多分类、回归等任务。
   自助采样过程还给Bagging带来了另一个优点：由于每个基学习器只使用了初始训练集中约63.2%的样本，剩下约36.8%的样本可用作验证集来对泛化性能进行“包外估计”。为此需记录每个基学习器所使用的训练样本。 包外样本还有许多其他用途，例如当基学习器是决策树时，可使用包外样本来辅助剪枝，或用于估计决策树中各结点的后验概率以辅助对零训练样本结点的处理；当基学习器是神经网络时，可使用包外样本来辅助早期停止以减小过拟合风险。 从偏差-方差分解的角度看，Bagging主要关注降低方差，因此它在不剪枝决策树、神经网络

等易受样本扰动的学习器上效果更为明显。

6. 随机森林 随机森林是Bagging的一个拓展辩题，RF在以决策树为基学习器构建Bagging集成的基础上，进一步在决策树的训练过程中引入了随机属性选择。具体来说，传统决策树在选择划分属性时是在当前结点的属性集合(假设有d个属性)中选择一个最优属性；而在RF中，对基决策树的每个结点，先从该结点的属性集合中随机选择一个包含k个属性的子集，然后再从这个子集中选择一个最优属性用于划分。这里的参数k控制了随机性的引入程度：若令k=d，则基决策树的构建与传统决策树相同；若令k=1，则是随机选择一个属性用于划分；一般情况下，推荐值k=$\log_2 d$ 随机森林简单、容易实现、计算开销小，令人惊奇的是，它在很多现实任务中展现出强大的性能，被誉为"代表集成学习技术水平的方法"。可以看出，随机森林对Bagging只做了小改动,但是与Bagging中基学习器的"多样性"仅通过样本扰动(通过对初始训练集采样)而来不同，随机森林中基学习器的多样性不仅来自样本扰动，还来自属性扰动，这就使得最终集成的泛化性能可通过个体学习器之间差异度的增加而进一步提升。 随机森林的收敛性与Bagging相似，随机森林的起始性能往往相对较差，特别是在集成中只包含一个基学习器时。这很容易理解，因为通过引入属性扰动，随机森林中个体学习器的性能往往有所降低。然而，随着个体学习器数目的增加，随机森林通常会收敛到更低的泛化误差。值得一提的是，随机森林的训练效率常优于Bagging，因为在个体决策树的构建过程中，Bagging使用的是"确定型"决策树，在选择划分属性时要对结点的所有属性进行考察，而随机森林使用的"随机型"决策树则只需考察一个属性子集。
7. 结合策略 学习器结合可能会从三个方面带来好处：首先，从统计的方面来看，由于学习任务的假设空间往往很大，可能有多个假设在训练集上达到同等性能，此时若使用单学习器可能因误选而导致泛化性能不佳，结合多个学习器则会减小这一风险；第二，从计算的方面来看，学习算法往往会陷入局部极小，有的局部极小点所对应的泛化性能可能很糟糕，而通过多次运行之后进行结合，可降低陷入糟糕局部极小点的风险；第三，从表示的方法来看，某些学习任务的真实假设可能不在当前学习算法所考虑的假设空间中，此时若使用但学习器则肯定无效，而通过结合多个学习器，由于相应的假设空间有所扩大，有可能学得更好的近似。
8. 平均法

- 简单平均法

$$H(x) = \frac{1}{T}\sum_{i=1}^{T} h_i(x).$$

- 加权平均法

$$H(x) = \sum_{i=1}^{T} w_i h_i(x).$$

其中w_i是个体学习器h_i的权重，通常要求$w_i \geq 0, \sum_{i=1}^{T} w_i = 1$. 加权平均法的权重一般是从训练数据中学习而得，现实任务中的训练样本通常不充分或存在噪声，这将使得学出的权重不完全可靠。尤其是对规模比较大的集成来说，要学习的权重比较多，较容易导致过拟合。一般而言，在个体学习器性能相差较大时宜使用加权平均法，而在个体学习器性能相近时宜使用简单平均法。 9. 投票法 对分类任务来说，学习器h_i将从类别标记集合{c_1,c_2,...,c_N}中预测出一个标记，最常见的结合策略是使用投票法(voting).

- 绝对多数投票法 若某标记得票数过半数，则预测为该标记；否则拒绝预测
- 相对多数投票法 预测为得票最多的标记，若同时有多个标记获最高票，则从中随机选取一个。
- 加权投票法 与加权平均法类似，w_i是h_i的权重

10. 学习法 当训练数据很多时，一种更为强大的结合策略是使用"学习法"，即通过另一个学习器来进行结合。Stacking是学习法的典型代表。这里我们把个体学习器称为初级学习器，用于结合的学习器称为次级学习器或元学习器。 Stacking先从初始数据集训练出初级学习器，然后"生成"一个新数据集用于训练次级学习器。 在训练阶段，次级训练集是利用初级学习器产生的，若直接用初级学习器的训练集来产生次级训练集，则过拟合风险会比较大；因此，一般是通过使用交叉验证或留一法这样的方式，用训练初级学习器未使用的样本来产生次级学习器的训练样本。以k折交叉验证为例，初始训练集D被随机划分为k个大小相似的集合$D_1, D_2, ..., D_k$.将预测值和真实值结合起来训练次级学习器，有研究表明，将初级学习器的输出类概率作为次级学习器的输入属性，用多响应线性回归作为次级学习算法效果较好，在MLR中使用不同的属性集更佳。 贝叶斯模型平均基于后验概率来为不同模型赋予权重，可视为加权平均法的一种特殊实现，对Stacking和BMA进行了比较，理论上来说，若

数据生成模型恰在当前考虑的模型中，且数据噪声很少，则BMA不差于Stacking；然而，在现实应用中无法确保数据生成模型一定在当前考虑的模型中，甚至可能难以用当前考虑的模型来进行近似，因为，Stacking通常优于BMA，因为其鲁棒性比BMA更好，而且BMA对模型近似误差非常敏感。

```
"""A random forest classifier.

    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).

    Read more in the :ref:`User Guide <forest>`.

    Parameters
    ----------
    n_estimators : integer, optional (default=10)
        The number of trees in the forest.

        .. versionchanged:: 0.20
           The default value of ``n_estimators`` will change from 10 in
           version 0.20 to 100 in version 0.22.

    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_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 fraction and
          `ceil(min_samples_split * n_samples)` are the minimum
          number of samples for each split.

        .. versionchanged:: 0.18
           Added float values for fractions.

    min_samples_leaf : int, float, optional (default=1)
        The minimum number of samples required to be at a leaf node.
        A split point at any depth will only be considered if it leaves at
        least ``min_samples_leaf`` training samples in each of the left and
        right branches.  This may have the effect of smoothing the model,
        especially in regression.

        - If int, then consider `min_samples_leaf` as the minimum number.
        - If float, then `min_samples_leaf` is a fraction and
          `ceil(min_samples_leaf * n_samples)` are the minimum
          number of samples for each node.

        .. versionchanged:: 0.18
```

```
       Added float values for fractions.

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_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 fraction 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_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_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.

    .. versionadded:: 0.19

min_impurity_split : float, (default=1e-7)
    Threshold for early stopping in tree growth. A node will split
    if its impurity is above the threshold, otherwise it is a leaf.

    .. deprecated:: 0.19
       ``min_impurity_split`` has been deprecated in favor of
       ``min_impurity_decrease`` in 0.19. The default value of
       ``min_impurity_split`` will change from 1e-7 to 0 in 0.23 and it
       will be removed in 0.25. Use ``min_impurity_decrease`` instead.


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 : int or None, optional (default=None)
    The number of jobs to run in parallel for both `fit` and `predict`.
    ``None`` means 1 unless in a :obj:`joblib.parallel_backend` context.
    ``-1`` means using all processors. See :term:`Glossary <n_jobs>`
    for more details.

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 when fitting and predicting.

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. See :term:`the Glossary <warm_start>`.

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.

Examples
--------
>>> from sklearn.ensemble import RandomForestClassifier
>>> from sklearn.datasets import make_classification

>>> X, y = make_classification(n_samples=1000, n_features=4,
...                            n_informative=2, n_redundant=0,
...                            random_state=0, shuffle=False)
>>> clf = RandomForestClassifier(n_estimators=100, max_depth=2,
...                              random_state=0)
>>> clf.fit(X, y)
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
            max_depth=2, 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=100, n_jobs=None,
            oob_score=False, random_state=0, verbose=0, warm_start=False)
>>> print(clf.feature_importances_)
[0.14205973 0.76664038 0.0282433  0.06305659]
>>> print(clf.predict([[0, 0, 0, 0]]))
[1]

Notes
-----
The default values for the parameters controlling the size of the trees
(e.g. ``max_depth``, ``min_samples_leaf``, etc.) lead to fully grown and
unpruned trees which can potentially be very large on some data sets. To
reduce memory consumption, the complexity and size of the trees should be
controlled by setting those parameter values.

The features are always randomly permuted at each split. Therefore,
the best found split may vary, even with the same training data,
``max_features=n_features`` and ``bootstrap=False``, if the improvement
of the criterion is identical for several splits enumerated during the
search of the best split. To obtain a deterministic behaviour during
fitting, ``random_state`` has to be fixed.
```

```
    References
    ----------

    .. [1] L. Breiman, "Random Forests", Machine Learning, 45(1), 5-32, 2001.

    See also
    --------
    DecisionTreeClassifier, ExtraTreesClassifier
    """
    def __init__(self,
                 n_estimators='warn',
                 criterion="gini",
                 max_depth=None,
                 min_samples_split=2,
                 min_samples_leaf=1,
                 min_weight_fraction_leaf=0.,
                 max_features="auto",
                 max_leaf_nodes=None,
                 min_impurity_decrease=0.,
                 min_impurity_split=None,
                 bootstrap=True,
                 oob_score=False,
                 n_jobs=None,
                 random_state=None,
                 verbose=0,
                 warm_start=False,
                 class_weight=None):
        super(RandomForestClassifier, self).__init__(
            base_estimator=DecisionTreeClassifier(),
            n_estimators=n_estimators,
            estimator_params=("criterion", "max_depth", "min_samples_split",
                              "min_samples_leaf", "min_weight_fraction_leaf",
                              "max_features", "max_leaf_nodes",
                              "min_impurity_decrease", "min_impurity_split",
                              "random_state"),
            bootstrap=bootstrap,
            oob_score=oob_score,
            n_jobs=n_jobs,
            random_state=random_state,
            verbose=verbose,
            warm_start=warm_start,
            class_weight=class_weight)

        self.criterion = criterion
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.min_samples_leaf = min_samples_leaf
        self.min_weight_fraction_leaf = min_weight_fraction_leaf
        self.max_features = max_features
        self.max_leaf_nodes = max_leaf_nodes
        self.min_impurity_decrease = min_impurity_decrease
        self.min_impurity_split = min_impurity_split
```

11. 应用场景 查不到具体应用场景，决策树可以拿来刚的地方，都可以试下随机森林，对噪声较大的数据，容易过拟。