DS-GA 1003 HW6

Yiyan Chen

April 2018

1 Reformulations of Multiclass Hinge Loss

1.1 Multiclass setting review

1.2 Two versions of multiclass hinge loss (or generalized hinge loss)

1.2.1

$$\begin{aligned}
l_2(h,(x_i,y_i)) &= max_{y\in Y}[\Delta(y_i,y)+h(x_i,y)-h(x_i,y_i)] \\
&= max(\Delta(y_i,y_i)+h(x_i,y_i)-h(x_i,y_i), max_{y\in Y-y_i}[\Delta(y_i,y)+h(x_i,y)-h(x_i,y_i)]) \\
&= max(0, max_{y\in Y-y_i}[\Delta(y_i,y)+h(x_i,y)-h(x_i,y_i)]) \\
&= max(0, max_{y\in Y-y_i}(\Delta(y_i,y)-m_{i,y}(h)) \\
&= max_{y\in Y-y_i}(max[0,\Delta(y_i,y),m_{i,y}(h)]) \\
&= l_1(h,(x_i,y_i))
\end{aligned}$$

1.2.2

(a):

$$\begin{aligned}
l_1(h,(x_i,y_i)) &= max_{y\in Y-y_i}(max[0,\Delta(y_i,y)+h(x_i,y)-h(x_i,y_i)]) \\
&= max_{y\in Y-y_i}(max[0]) \\
&= 0
\end{aligned}$$

$$\begin{aligned}
l_2(h,(x_i,y_i)) &= max(\Delta(y_i,y_i)+h(x_i,y_i)-h(x_i,y_i), max_{y\in Y-y_i}(\Delta(y_i,y)+h(x_i,y)-h(x_i,y_i))) \\
&= max(0, max_{y\in Y-y_i}(<0)) \\
&= 0
\end{aligned}$$

(b):

$$
\begin{aligned}
h(x_i, y) &\leq h(x_i, y_i) - \Delta(y_i, y) \\
\max h(x_i, y) &= h(x_i, y_i) - \Delta(y_i, y) \\
\max h(x_i, y) &= h(x_i, y_i) - 0 \\
\max h(x_i, y) &= h(x_i, y_i) \\
argmax_{y \in Y} h(x_i, y_i) &= y_i
\end{aligned}
$$

2 SGD for Multiclass Liner SVM

2.1 Optional

The first part of the function: $\lambda||w||^2$ is convex since its derivative$2\lambda w > 0$. The second part of the function is the sum of convex functions. Hence, two parts combined is the convex function.

2.2

Set g as the subgradient:

$$
\begin{aligned}
J(w) &= J_1(w) + J_2(w) \\
J_1(w) = \lambda||w||^2, J_2(w) &= \frac{1}{n}\sum_{i=1}^{n} max_{y\in Y}[\Delta(y_i, y) + < w, \Psi(x_i, y) - \Psi(x_i, y_i) >] \\
J_2(w_1) &= J_2(w_2) + g(w_1 - w_2) \\
\sum_{i=1}^{n}[\Delta(y_i, \hat{y}_i + < w_1, \Psi(x_i, y) - \Psi(x_i, y_i) >] &= \sum_{i=1}^{n}[\Delta(y_i, \hat{y}_i + < w_2, \Psi(x_i, \hat{y}) - \Psi(x_i, y_i) >] + ng(w_1 - w_2) \\
g &= \frac{\sum_{i=1}^{n} < w_1 - w_2, \Psi(x_i, \hat{y}) - \Psi(x_i, y_i) >}{n(w_1 - w_2)} \\
&= \frac{1}{n}\sum_{i=1}^{n}\frac{(w_1 - w_2)^T(\Psi(x_i, \hat{y}) - \Psi(x_i, y_i))}{w_1 - w_2} \\
&= \frac{1}{n}\sum_{i=1}^{n}[\Psi(x_i, \hat{y}) - \Psi(x_i, y_i)]
\end{aligned}
$$

$$
g = 2\lambda w + \frac{1}{n}\sum_{i=1}^{n}[\Psi(x_i, \hat{y}) - \Psi(x_i, y_i)]
$$

2.3

The stochastic subgradient iteration takes one point (x_i, y_i) at a time:

$$
\begin{aligned}
w &\longleftarrow w - \eta g \\
&\longleftarrow w - \eta(2\lambda w + [\Psi(x_i, \hat{y}) - \Psi(x_i, y_i)])
\end{aligned}
$$

2.4

Minibatch subgradient iteration takes $(x_i, y_y) \ldots (x_{i+m-1}, y_{i+m-1})$:

$$
\begin{aligned}
w &\longleftarrow w - \eta g \\
&\longleftarrow w - \eta(2\lambda w + \frac{1}{m}\sum_{i=j}^{j+m-1}[\Psi(x_i, \hat{y}) - \Psi(x_i, y_i)])
\end{aligned}
$$

3 [Optional]

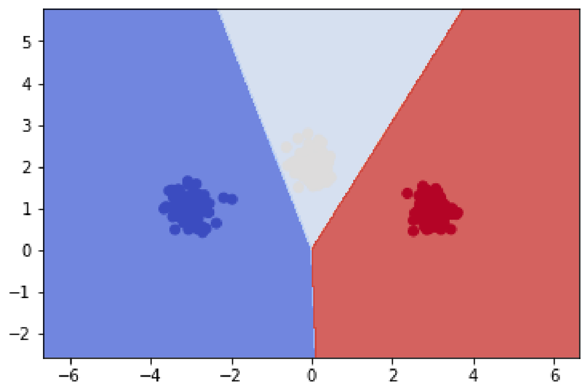
4 Muliclass Classification - Implementation

4.1 One-vs-All

```
from sklearn.base import BaseEstimator, ClassifierMixin, clone

class OneVsAllClassifier(BaseEstimator, ClassifierMixin):
    """
    One-vs-all classifier
    We assume that the classes will be the integers 0,..,(n_classes-1).
    We assume that the estimator provided to the class, after fitting, has a "decision_func
    returns the score for the positive class.
    """
    def __init__(self, estimator, n_classes):
        """
        Constructed with the number of classes and an estimator (e.g. an
        SVM estimator from sklearn)
        @param estimator : binary base classifier used
        @param n_classes : number of classes
        """
        self.n_classes = n_classes
        self.estimators = [clone(estimator) for _ in range(n_classes)]
        self.fitted = False

    def fit(self, X, y=None):
        """
        This should fit one classifier for each class.
        self.estimators[i] should be fit on class i vs rest
        @param X: array-like, shape = [n_samples,n_features], input data
        @param y: array-like, shape = [n_samples,] class labels
        @return returns self
        """
        #Your code goes here
        for i in range(self.n_classes):
            y_i = np.where(y==i, 1,0)
            self.estimators[i].fit(X,y_i)
        self.fitted = True
        return self


    def decision_function(self, X):
        """
        Returns the score of each input for each class. Assumes
```

```
        that the given estimator also implements the decision_function method (which sklearn
        and that fit has been called.
        @param X : array-like, shape = [n_samples, n_features] input data
        @return array-like, shape = [n_samples, n_classes]
        """
        if not self.fitted:
            raise RuntimeError("You must train classifer before predicting data.")

        if not hasattr(self.estimators[0], "decision_function"):
            raise AttributeError(
                "Base estimator doesn't have a decision_function attribute.")

        #Replace the following return statement with your code
        if len(self.estimators) == 1:
            score = self.estimator[0].decision_function(X)
        else:
            score = np.zeros([X.shape[0], self.n_classes])
            for i in range(self.n_classes):
                score.T[i] = self.estimators[i].decision_function(X)
        return score

    def predict(self, X):
        """
        Predict the class with the highest score.
        @param X: array-like, shape = [n_samples,n_features] input data
        @returns array-like, shape = [n_samples,] the predicted classes for each input
        """
        #Replace the following return statement with your code
        score = self.decision_function(X)
        pred = np.zeros(X.shape[0])
        for i in range(X.shape[0]):
            zipped = zip(score[i], range(self.n_classes))
            pred[i] = sorted(zipped, key=itemgetter(0))[-1][1]
        return pred
```

```
Coeffs 0
[[-1.05852747 -0.90296521]]
Coeffs 1
[[ 0.22117096 -0.38900908]]
Coeffs 2
[[ 0.89162796 -0.82467394]]

array([[100,   0,   0],
       [  0, 100,   0],
       [  0,   0, 100]])
```



4.2 Multiclass SVM

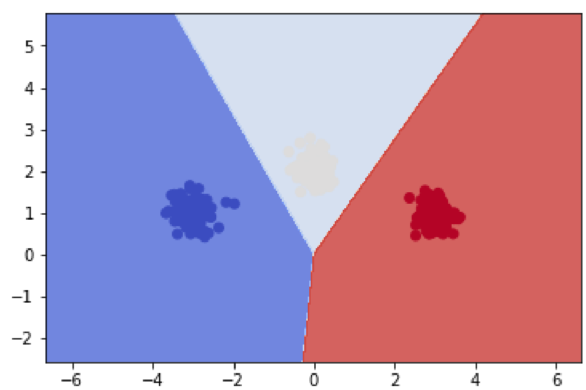
```python
def zeroOne(y,a) :
    '''
    Computes the zero-one loss.
    @param y: output class
    @param a: predicted class
    @return 1 if different, 0 if same
    '''
    return int(y != a)

def featureMap(X,y,num_classes) :
    '''
    Computes the class-sensitive features.
    @param X: array-like, shape = [n_samples,n_inFeatures] or [n_inFeatures,], input featur
    @param y: a target class (in range 0,..,num_classes-1)
    @return array-like, shape = [n_samples,n_outFeatures], the class sensitive features for
    '''
    #The following line handles X being a 1d-array or a 2d-array
    num_samples, num_inFeatures = (1,X.shape[0]) if len(X.shape) == 1 else (X.shape[0],X.sha
    #your code goes here, and replaces following return
    #x = (x1, x2), classes = (0,1,2)-> (x, 0) = (x1,x2, 0, 0, 0, 0)
    n_outFeatures = num_classes*num_inFeatures
    rn = np.zeros([num_samples, n_outFeatures])
    y=np.array([y])
    if num_samples == 1:
        X = X.reshape((1, -1))
    for i in range(num_samples):
        if len(y) > 1:
            y_i =y[i]
        else:
```

```
            y_i = y
        rn[i][int(y_i)*num_inFeatures:int(y_i)*num_inFeatures+num_inFeatures] = X[i]
    return rn

def sgd(X, y, num_outFeatures, subgd, eta = 0.001, T = 10000):
    '''
    Runs subgradient descent, and outputs resulting parameter vector.
    @param X: array-like, shape = [n_samples,n_features], input training data
    @param y: array-like, shape = [n_samples,], class labels
    @param num_outFeatures: number of class-sensitive features
    @param subgd: function taking x,y and giving subgradient of objective
    @param eta: learning rate for SGD
    @param T: maximum number of iterations
    @return: vector of weights
    '''
    num_samples = X.shape[0]
    #your code goes here and replaces following return statement
    w=np.zeros(num_outFeatures)
    for t in range(T):
        i = np.random.randint(num_samples)
        w = w-eta*subgd(X[i], y[i], w)
    return w


class MulticlassSVM(BaseEstimator, ClassifierMixin):
    '''
    Implements a Multiclass SVM estimator.
    '''
    def __init__(self, num_outFeatures, lam=1.0, num_classes=3, Delta=zeroOne, Psi=featureMa
        '''
        Creates a MulticlassSVM estimator.
        @param num_outFeatures: number of class-sensitive features produced by Psi
        @param lam: l2 regularization parameter
        @param num_classes: number of classes (assumed numbered 0,..,num_classes-1)
        @param Delta: class-sensitive loss function taking two arguments (i.e., target marg
        @param Psi: class-sensitive feature map taking two arguments
        '''
        self.num_outFeatures = num_outFeatures
        self.lam = lam
        self.num_classes = num_classes
        self.Delta = Delta
        self.Psi = lambda X,y : Psi(X,y,num_classes)
        self.fitted = False

    def subgradient(self,x,y,w):
        '''
```

```
        Computes the subgradient at a given data point x,y
        @param x: sample input
        @param y: sample class
        @param w: parameter vector
        @return returns subgradient vector at given x,y,w
        '''
        #Your code goes here and replaces the following return statement
        y_hat = 0
        w = w.reshape((1,-1))
        for i in range(self.num_classes):
            cal = self.Delta(y, i)+np.dot(w, (self.Psi(x,i)-self.Psi(x,y)).T)
            if cal>y_hat:
                y_hat = i
        g = 2*self.lam*w+self.Psi(x,y_hat)-self.Psi(x,y)
        return g


    def fit(self,X,y,eta=0.001,T=10000):
        '''
        Fits multiclass SVM
        @param X: array-like, shape = [num_samples,num_inFeatures], input data
        @param y: array-like, shape = [num_samples,], input classes
        @param eta: learning rate for SGD
        @param T: maximum number of iterations
        @return returns self
        '''
        self.coef_ = sgd(X,y,self.num_outFeatures,self.subgradient,eta,T)
        self.fitted = True
        return self

    def decision_function(self, X):
        '''
        Returns the score on each input for each class. Assumes
        that fit has been called.
        @param X : array-like, shape = [n_samples, n_inFeatures]
        @return array-like, shape = [n_samples, n_classes] giving scores for each sample,cl
        '''
        if not self.fitted:
            raise RuntimeError("You must train classifer before predicting data.")

        #Your code goes here and replaces following return statement
        score = np.zeros([X.shape[0], self.num_classes])
        for sample in range(X.shape[0]):
            for cls in range(self.num_classes):
                score[sample][cls] = np.dot(self.coef_, (self.Psi(X[sample], cls)).T)
        return score
```

```python

    def predict(self, X):
        '''
        Predict the class with the highest score.
        @param X: array-like, shape = [n_samples, n_inFeatures], input data to predict
        @return array-like, shape = [n_samples,], class labels predicted for each data poin
        '''
        #Your code goes here and replaces following return statement
        pred = np.zeros(X.shape[0])
        score = self.decision_function(X)
        for i in range(X.shape[0]):
            zipped = zip(score[i], range(self.num_classes))
            pred[i] = sorted(zipped, key=itemgetter(0))[-1][1]
        return pred
```

```
w:
[[-0.45890788 -0.0553265  -0.01807324  0.20763369  0.47698112 -0.15230719]]

array([[100,   0,   0],
       [  0, 100,   0],
       [  0,   0, 100]])
```



5 [Optional]

6 [Optional]

7 Gradient Boosting Machines

7.1

$$
\begin{aligned}
(-g_m)_i &= -\frac{\partial}{\partial f_{m-1}(x_i)} l(y_i, f_{m-1}(x_i)) \\
&= (y_i - f_{m-1}(x_i)) \\
h_m &= argmin_{h \in F} \sum_{i=1}^{n} [(-g_m)_i - h(x_i)]^2 \\
&= argmin_{h \in F} \sum_{i=1}^{n} [(y_i - f_{m-1}(x_i)) - h(x_i)]^2
\end{aligned}
$$

7.2

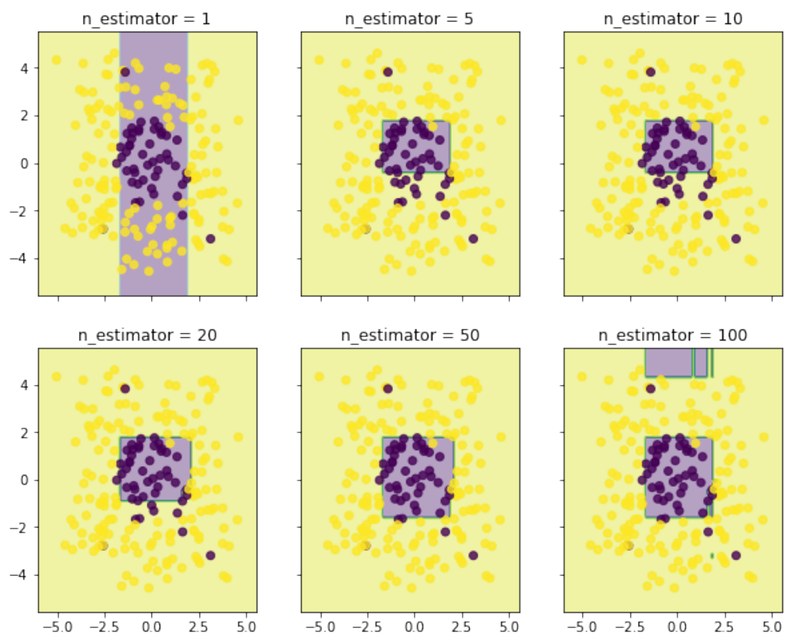
$$
\begin{aligned}
\frac{\partial ln(1 + e^{-yf(x)})}{\partial f(x)} &= \frac{-ye^{-yf(x)}}{1 + e^{-yf(x)}} \\
(-g_m)_i &= \frac{y_i}{e^{y_i f(x_i)} + 1} \\
h_m &= argmin_{h \in F} \sum_{i=1}^{n} [(-g_m)_i - h(x_i)]^2 \\
&= argmin_{h \in F} \sum_{i=1}^{n} [\frac{y_i}{e^{y_i f(x_i)} + 1} - h(x_i)]^2
\end{aligned}
$$

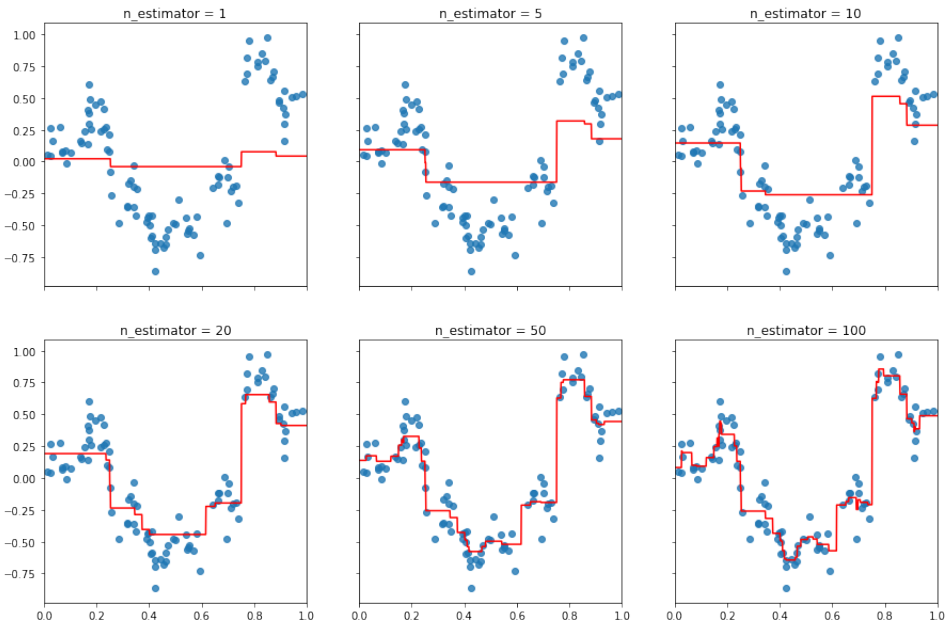
8 Gradient Boosting Implementation

8.1

```
class gradient_boosting():
    '''
    Gradient Boosting regressor class
    :method fit: fitting model
    '''
    def __init__(self, n_estimator, pseudo_residual_func, learning_rate=0.1, min_sample=5, m
        '''
        Initialize gradient boosting class

        :param n_estimator: number of estimators (i.e. number of rounds of gradient boosting
        :pseudo_residual_func: function used for computing pseudo-residual
        :param learning_rate: step size of gradient descent
        '''
        self.n_estimator = n_estimator
        self.pseudo_residual_func = pseudo_residual_func
        self.learning_rate = learning_rate
        self.min_sample = min_sample
        self.max_depth = max_depth


    def fit(self, train_data, train_target):
        '''
        Fit gradient boosting model
        '''
        # Your code goes here
        #n_estimator - M, n - train_data.shape[0]
        fm = np.zeros(train_data.shape[0])
#         print(train_target.shape)
        train_target = train_target.squeeze()
#         print(train_target.shape)
        self.hm = []
        for i in range(self.n_estimator):
            rgs = DecisionTreeRegressor(min_samples_split=self.min_sample,
            max_depth=self.max_depth)
            rgs.fit(train_data, self.pseudo_residual_func(train_target, fm))
            self.hm.append(rgs)
            h = rgs.predict(train_data)
            fm+=self.learning_rate*h
        return self

    def predict(self, test_data):
```

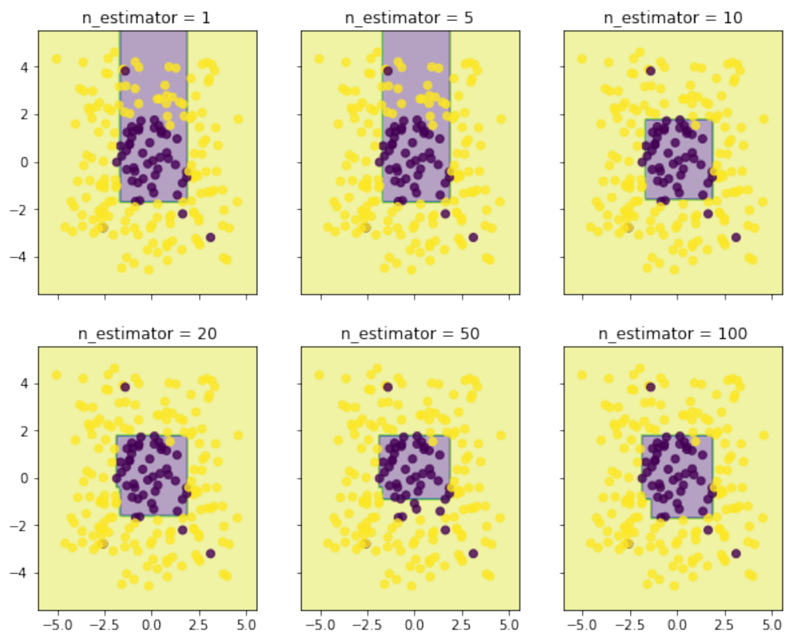
```
        '''
        Predict value
        '''
        # Your code goes here
        pred = np.zeros(test_data.shape[0])
        for i in range(self.n_estimator):
            rgs = self.hm[i]
            pev = rgs.predict(test_data)
            pred+= self.learning_rate*pev
        return pred
```

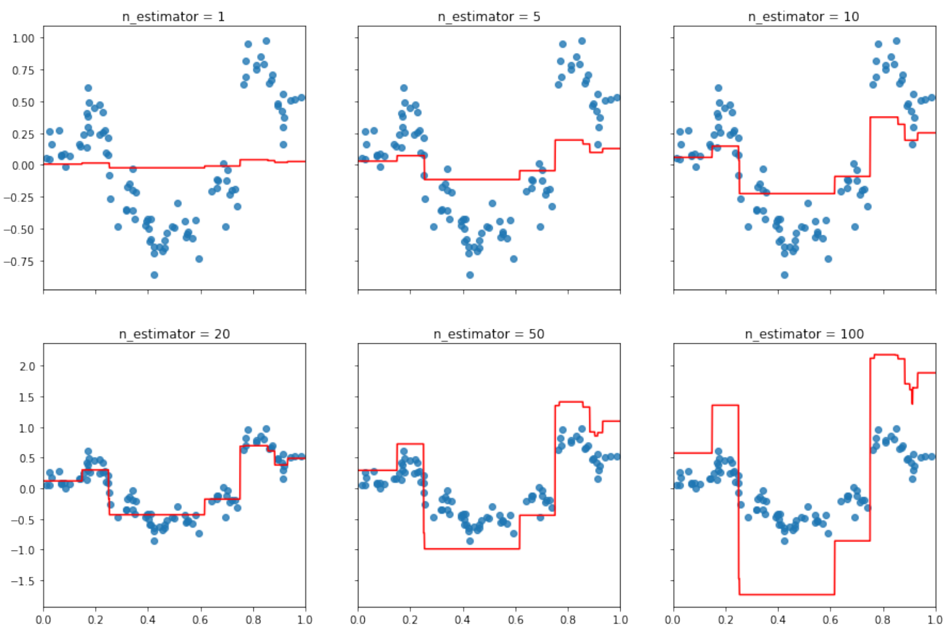



n_estimator = 1
n_estimator = 5
n_estimator = 10
n_estimator = 20
n_estimator = 50
n_estimator = 100
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
0.0
0.2
0.4
0.6
0.8
1.0

8.2 [Optional]

```python
def pseudo_residual_logistic(train_target, train_predict):
    '''
    Compute the pseudo-residual based on current predicted value.
    '''
    y= train_target
    f = train_predict
    return y/(np.exp(y*f)+1)
```




n_estimator = 1
n_estimator = 5
n_estimator = 10
n_estimator = 20
n_estimator = 50
n_estimator = 100