Lecture 7

1. Generalized additive models (GAM)
2. Tree-based methods
3. Boosting
4. Boosting trees

1 Generalized additive models (GAM)

In the previous lectures we talked about the generalized linear models (GLM), where we assumed that target variable y has a distribution from exponential family and the prediction is an expected value μ of this distribution. The link function g^{-1} has been introduced such that

$$g^{-1}(\mu) = f(\mu) = \theta^T x$$

(for convenience, we denoted $f = g^{-1}$). Recall that:

- if $f(\mu) = \mu$ and $y \sim N(\mu, \sigma^2)$, then the resulted model is a linear regression;
- if $f(\mu) = \ln\left(\dfrac{\mu}{1-\mu}\right)$ (or, $\mu = \dfrac{1}{1+e^{-\eta}}$) and $y \sim \text{Ber}(\mu)$, then the resulted model is a logistic regression.

The generalized additive models can be introduced as a modification of GLM, we assume that

$$f(\mu) = \alpha + f_1(x_1) + f_2(x_2) + \ldots + f_n(x_n),$$

where $f_j, j = 1, \ldots, n$, are nonlinear differentiable functions, α is a constant (notice that this is nonparametric model). Consider two cases: GAM for regression and GAM for classification.

1.1 GAM for regression

We assume that $f(\mu) = \mu$ and $y \sim N(\mu, \sigma^2)$, in other words,

$$y|x; \alpha; f_j \sim N(\alpha + f_1(x_1) + f_2(x_2) + \ldots + f_n(x_n), \sigma^2).$$

We find the best value for α using maximum likelihood estimation. The log-likelihood has a form

$$l(\alpha, f_1, \ldots, f_n) = \ln \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(- \frac{(y^{(i)} - \alpha - \sum\limits_{j=1}^{n} f_j(x_j^{(i)}))^2}{2\sigma^2} \right).$$

The derivative with respect to α is

$$l'_\alpha = \frac{1}{\sigma^2}\sum_{i=1}^{m}\left(y^{(i)} - \alpha - \sum_{j=1}^{n} f_j(x_j^{(i)})\right) = 0.$$

Because of the term $\sum_{j=1}^{n} f_j(x_j^{(i)})$, α cannot be found explicitly. Assuming the additional condition

$$\sum_{i=1}^{m} f_j(x_j^{(i)}) = 0 \text{ for any } j$$

(mean of f_j along the data is zero), we will find

$$\alpha = \frac{1}{m}\sum_{i=1}^{m} y^{(i)}$$

(average of all targets in the dataset).

The next step will be to find the estimations for f_j. If we have already found the estimations for $f_j, j \neq 1$ we will find the best estimation for f_1 (general case can be considered by analogy):

$$l(\alpha, f_j) = \ln \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y^{(i)} - \alpha - f_1(x_1^{(i)}) - \sum_{j=2}^{n} f_j(x_j^{(i)}))^2}{2\sigma^2}\right)$$

Denote

$$z^{(i)} = y^{(i)} - \alpha - \sum_{j=2}^{n} f_j(x_j^{(i)}),$$

then the log-likelihood equals to

$$l(\alpha, f_j) = \ln \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(z^{(i)} - f_1(x_1^{(i)}))^2}{2\sigma^2}\right).$$

We reformulate the problem: find the best estimation for f_1 with condition

$$z|x; f_1 \sim N(f_1(x_1), \sigma^2).$$

The advantage of this formulation is that the model with only one feature should be trained. It gives a rise to the **backfitting algorithm**.

Algorithm 1 Backfitting algorithm for GAM regression

```
1: set initial values $\alpha = \frac{1}{m}\sum_{i=1}^{m} y^{(i)}$, $f_j = 0$ for all $j = 1, \dots, n$
2: repeat
3:   for $j = 1$ to $n$ do
4:     evaluate working targets $z^{(i)} = y^{(i)} - \alpha - \sum_{k=1, k \neq j}^{n} f_k(x_k^{(i)})$
5:     train model with feature $x_j$ and target $z$ to estimate $f_j$
6: until convergence
7: return $\alpha$, $f_j$
```

For the line 5 of this algorithm we should choose a single variable model. Simple nonparametric approaches could be used (for example, weighted linear regression or cubic splines).

1.2 Python implementation

Import necessary libraries:

```
In [1]: import numpy as np
        import pandas as pd
        import random
        import math
        import sklearn.datasets as ds
        from sklearn.cross_validation import train_test_split
        from ml_metrics import rmse
        from matplotlib import cm
        import matplotlib.pyplot as plt
```

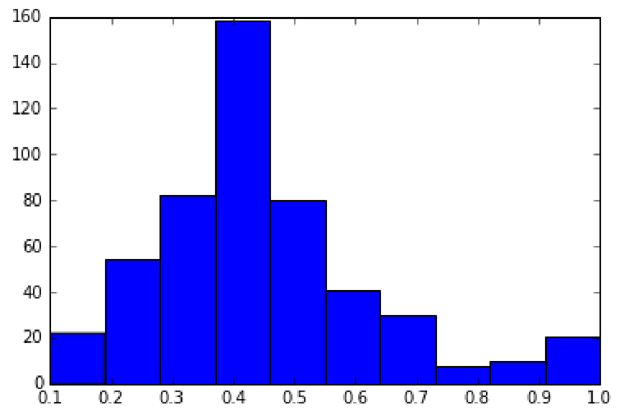
As a benchmark we will use `boston` data from the `scikit-learn` package.

```
In [2]: boston = ds.load_boston()
        X = boston.data
        y = boston.target/50.
        X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                            test_size=0.33,
                                                            random_state=42)
```

```
In [3]: print 'Number of training examples: ' + str(X.shape[0])
        print 'Number of features: ' + str(X.shape[1])
```

```
Number of training examples: 506
Number of features: 13
```

```
In [4]: plt.hist(y)
        plt.show()
```



To implement backfitting algorithm we will use cubic spline approximation as our building block algorithm (function `UnivariateSpline` from `scipy` package). Now we train generalized additive model with backfitting algorithm.

```
In [5]: from scipy.interpolate import UnivariateSpline

In [6]: # number of training examples
        m = X_train.shape[0]
        # number of features
        n = X_train.shape[1]
        # bias term for GAM
        alpha = np.mean(y_train)
        # list of trained splines
        fj = []
        # number of iterations
        niter = 100
        for it in range(niter):
            for var in range(n):
                # evaluate values for previous splines
                fj_eval = [f(X_train[:,j]) for j,f in enumerate(fj)
                           if j != var]
                # convert values to numpy array
                fj_eval = np.array(fj_eval).T.reshape((m, -1))
                # calculate working target
                z = y_train - alpha - np.sum(fj_eval, axis=1)
                # prepare var and z for UnivariateSpline:
                df = pd.DataFrame()
                df['x'] = np.round(X_train[:,var], 1)
                df['y'] = z
                df.sort_values(by='x', inplace=True)
                df = df.groupby('x')['y'].mean().reset_index()
                # fit UnivariateSpline model:
                if it == 0:
                    # add splines to fj
                    fj.append(UnivariateSpline(df.x.values,
                                               df.y.values,
                                               k=min(len(df)-1,3)))
                else:
                    # replace splines in fj
                    fj[var] = UnivariateSpline(df.x.values,
                                               df.y.values,
                                               k=min(len(df)-1,3))
```

The result will be the set of cubic splines for each variable saved in the list `fj`. Now we check the accuracy on the test set:

```
In [7]: # evaluate spline values for test points
        fj_eval = [f(X_test[:,j]) for j,f in enumerate(fj)]
        # convert spline values to numpy array
        fj_eval = np.array(fj_eval).T.reshape((X_test.shape[0], -1))
        # calculate predictions
        y_pred = alpha + np.sum(fj_eval, axis=1)
        # print accuracy
        print rmse(y_test, y_pred)
```

```
0.100600863408
```

1.3 GAM for classification

For the classification we assume $f(\mu) = \ln\left(\frac{\mu}{1-\mu}\right)$ and Bernoulli distribution for the target y, in other words:

$$y|x;\alpha;f_j \sim \mathrm{Ber}(g(\alpha + f_1(x_1) + f_2(x_2) + \ldots + f_n(x_n))),$$

where $g(z) = \frac{1}{1+e^{-z}}$ is a sigmoid function. Denote

$$\eta = \alpha + f_1(x_1) + f_2(x_2) + \ldots + f_n(x_n),$$

then the log-likelihood can be written as

$$l(\eta) = \sum_{i=1}^{m} y^{(i)} \ln \mu^{(i)} + (1 - y^{(i)}) \ln(1 - \mu^{(i)}).$$

To find the initial value for α we assume that $f_j = 0$ for all $j = 1, \ldots, n$:

$$l'_\eta = \sum_{i=1}^{m} y^{(i)}(1-\mu^{(i)}) - (1-y^{(i)})\mu^{(i)} = 0 \Rightarrow \alpha = \ln\left(\frac{\mu}{1-\mu}\right),$$

where

$$\mu = \frac{1}{m}\sum_{i=1}^{m} y^{(i)}.$$

To find f_j we will use the following procedure. First, we use Newton's method with respect to η:

$$\eta^{new} = \eta^{old} - \frac{l'_\eta(\eta^{old})}{l''_\eta(\eta^{old})} \text{(for each training example).}$$

But

$$\begin{aligned} l'(\eta) &= y(1-\mu) - (1-y)\mu = y - \mu, \\ l''(\eta) &= -\mu(1-\mu) \end{aligned}$$

(remember that $\mu = g(\eta)$). Then the updating formula for η is

$$\eta^{new} := \eta^{old} + \frac{y - \mu^{old}}{\mu^{old}(1-\mu^{old})}.$$

The second step will be to use backfitting algorithm for GAM regression (previous section) with target values η^{new}.

The important difference between linear case and logistic case is that in the linear case we assumed that variance σ^2 for all training examples is constant. In the logistic case the variance for the Bernoulli distribution is calculated as $\mu(1-\mu)$. It means that we should normalize variances for the training examples to the same value; it can be easily done by using weights $\mu(1-\mu)$ in backfitting algorithm (**weighted backfitting algorithm**):

Algorithm 2 GAM for classification with weighted backfitting algorithm

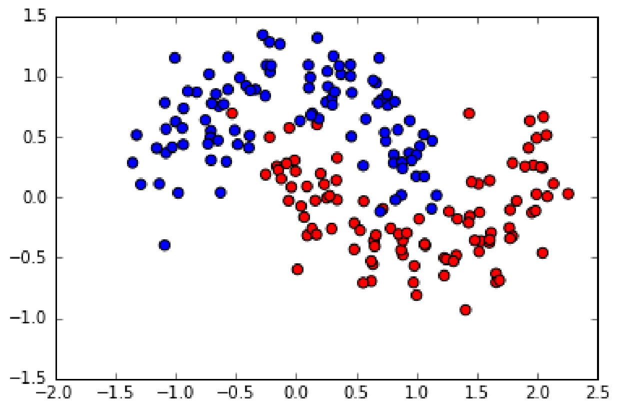
```
1: set initial values $\mu = \frac{1}{m}\sum_{i=1}^{m} y^{(i)}$, $\alpha = \ln\left(\frac{\mu}{1-\mu}\right)$, $f_j = 0$, $j = 1, \ldots, n$
2: repeat
3:     evaluate $\eta^{(i)} = \alpha + \sum_{j=1}^{n} f_j(x^{(i)})$ and $\mu^{(i)} = \frac{1}{1+e^{-\eta^{(i)}}}$
4:     make the Newton's step $\eta^{(i)} := \eta^{(i)} + \frac{y^{(i)} - \mu^{(i)}}{\mu^{(i)}(1-\mu^{(i)})}$
5:     evaluate weights $w^{(i)} = \mu^{(i)}(1-\mu^{(i)})$
6:     evaluate new value for $\alpha = \frac{1}{m}\sum_{i=1}^{m} \eta^{(i)}$
7:     for $j = 1$ to $n$ do
8:         evaluate working targets $z^{(i)} = \eta^{(i)} - \alpha - \sum_{k=1,k\neq j}^{n} f_k(x_k^{(i)})$
9:         train model with feature $x_j$, target $z$ and weights $w$ to estimate $f_j$
10: until convergence
11: return $\alpha$, $f_j$
```

As before for the step 9 of the algorithm we can choose some nonparametric model, for example, cubic spline fitting or weighted linear regression.

1.4 Python implementation

For classification we will use make_moons data from the scikit-learn package.

```
In [8]: np.random.seed(0)
        X_train, y_train = ds.make_moons(200, noise=0.20)
        plt.scatter(X_train[:,0], X_train[:,1],
                    s=40, c=y_train, cmap=cm.bwr)
        plt.show()
```

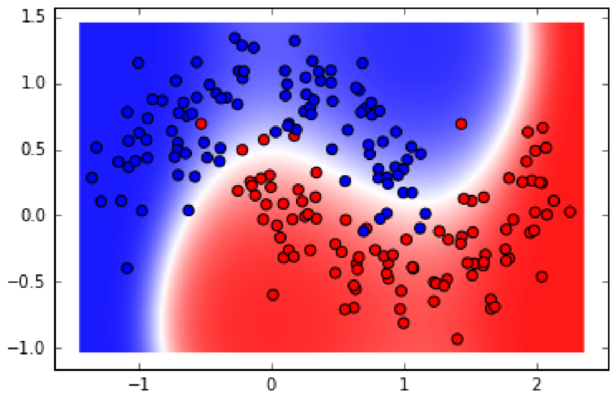


Now we implement GAM for logistic regression and weighted backfitting algorithm.

```
In [9]: # number of training examples
        m = X_train.shape[0]
        # number of features
        n = X_train.shape[1]
        # list of trained splines
        fj = []
        # number of iterations
        niter = 100
        for it in range(niter):
            y_mean = np.mean(y_train)
            alpha = np.log(y_mean/(1-y_mean))
            if it == 0:
                eta = np.array([alpha]*m)
            else:
                fj_eval = np.array([f(X_train[:,j])
                                    for j,f in enumerate(fj)
                                    if j != var]).T.reshape((m, -1))
                eta = np.array([alpha]*m) + np.sum(fj_eval, axis=1)
            mu = 1.0/(1 + np.exp(-1.0*eta))
            z = eta + (y_train - mu)/(mu*(1-mu))
            w = np.multiply(mu, 1.0-mu)
            alpha_z = np.mean(z)
            for var in range(n):
                fj_eval = np.array([f(X_train[:,j])
                                    for j,f in enumerate(fj)
                                    if j != var]).T.reshape((m, -1))
                zz = z - alpha_z - np.sum(fj_eval, axis=1)
                df = pd.DataFrame()
                df['x'] = np.round(X_train[:,var], 1)
                df['y'] = zz
                df['w'] = w
                df.sort_values(by='x', inplace=True)
                df = df.groupby('x')[['y', 'w']].mean().reset_index()
                # fit UnivariateSpline model:
                if it == 0:
                    # add splines to fj
                    fj.append(UnivariateSpline(df.x.values,
                                               df.y.values,
                                               df.w.values,
                                               k=min(len(df)-1,3)))
                else:
                    # replace splines in fj
                    fj[var] = UnivariateSpline(df.x.values,
                                               df.y.values,
                                               df.w.values,
                                               k=min(len(df)-1,3))
            alpha = alpha_z
```

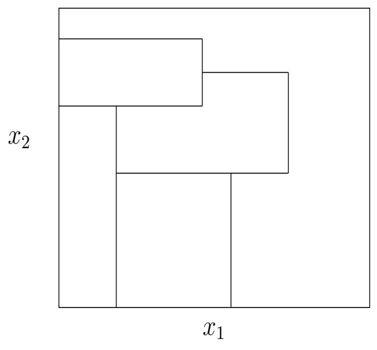
```
In [10]: def plot_decision_boundary(X, y, alpha, fj,
                                    xmin=-0.1, xmax=1.1,
                                    ymin=-0.1, ymax=1.1):
             fig = plt.figure(figsize=(6,6))
             x1, x2 = np.meshgrid(np.linspace(xmin, xmax, 200),
                                  np.linspace(ymin, ymax, 200))
             Xt = np.c_[x1.ravel(), x2.ravel()]
             fj_eval = np.array([f(Xt[:,j])
                                 for j,f in enumerate(fj)]).T.reshape((Xt.shape[0], -1))
             ypred = 1.0/(1.0 + np.exp(-alpha - np.sum(fj_eval, axis=1)))
             ypred = ypred.reshape(x1.shape)
             extent = xmin, xmax, ymin, ymax

             plt.imshow(ypred, cmap=cm.bwr, alpha=.9, interpolation='bilinear',
                        extent = extent, origin='lower')
             plt.scatter(X[:,0], X[:,1], s=40, c=y, cmap=cm.bwr)
```

```
In [11]: plot_decision_boundary(X_train, y_train, alpha, fj,
                                xmin=np.min(X_train[:,0])-0.1,
                                xmax=np.max(X_train[:,0])+0.1,
                                ymin=np.min(X_train[:,1])-0.1,
                                ymax=np.max(X_train[:,1])+0.1)
```

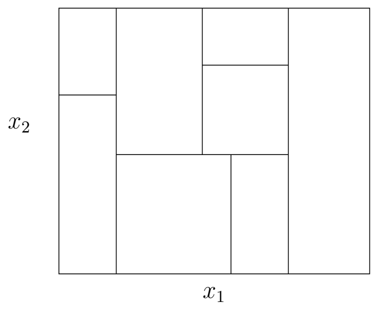


2 Tree-based methods

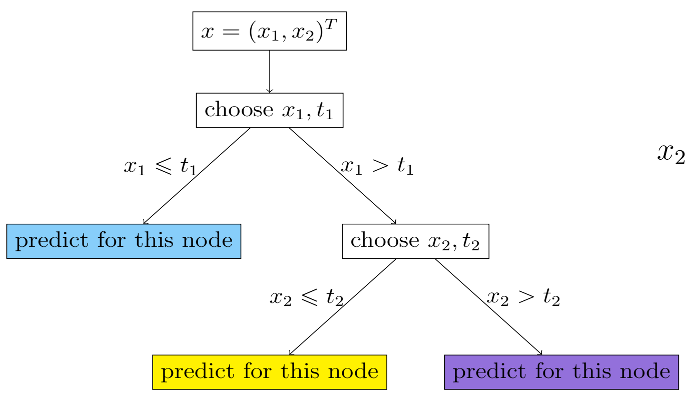
Another nonparametric approach utilizes the structure called decision tree. In many cases we should split the feature space in small regions and predict the target variable y for each region separately. The example of splitting is shown on the following picture:

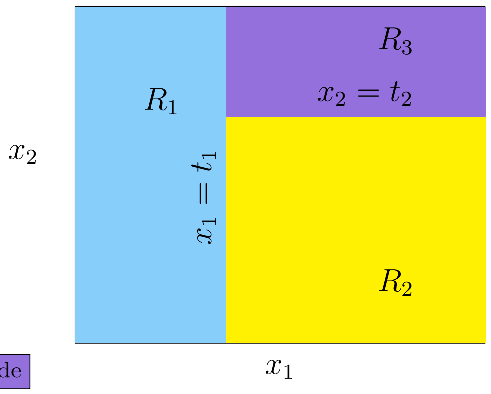


Unfortunately, it is difficult to describe such splitting analytically. We consider simpler type of partition - binary partitions:



We can describe the last splitting using **decision trees**. The methods that utilize the decision trees are called tree-based methods. Consider the problem with two features $x = (x_1, x_2)^T$, the simplest decision tree with according binary partition could look like this:





Usually, the prediction is constant for each **terminal node** or **leaves** (coloured nodes on the picture). The advantage of the decision tree structure is that the hypothesis function $h_{c,t}(x)$ can be expressed as a linear combination of indicator functions. For the previous example, the hypothesis can be written as

$$h_{c,t}(x) = c_1 \cdot \mathbb{1}\{x_1 \leqslant t_1\} + c_2 \cdot \mathbb{1}\{x_1 > t_1, x_2 \leqslant t_2\} + c_3 \cdot \mathbb{1}\{x_1 > t_1, x_2 > t_2\}.$$

In this model we have 5 parameters: c_1, c_2, c_3 are predictions for the terminal nodes, t_1, t_2 are splitting points.

In general case if we have n features $x = (x_1, \ldots, x_n)^T$, the hypothesis function can be written as

$$h_{c,t}(x) = \sum_{k=1}^{K} c_k \mathbb{1}\{x \in R_k\}.$$

Here K is a number of terminal nodes and R_k is the region of the feature space that corresponds to the terminal node k. The main advantage of this model is interpretability.

2.1 Regression trees

We consider in details how to build the decision trees for regression problems. Assuming that the prediction c_k for each region R_k is constant it is easy to prove that the best prediction would be the average of $y^{(i)}$ for the training examples in this region.

Indeed, we could use the probabilisitic approach in this case. Consider the simplest situation when we have one feature and one target. If the variable x is splitted by the number c and μ_1 is a constant prediction for $x < c$, μ_2 is a constant prediction for $x \geqslant c$, then

$$y|x, c, \mu_1, \mu_2 \sim N((\mu_2 - \mu_1)\mathbb{1}\{x \geqslant c\} + \mu_1, \sigma^2).$$

The log-likelihood

$$l(\mu_1, \mu_2, c) = m \ln \frac{1}{\sqrt{2\pi}\sigma} - \sum_{i=1}^{m} \frac{(y^{(i)} - (\mu_2 - \mu_1)\mathbb{1}\{x^{(i)} \geqslant c\} - \mu_1)^2}{2\sigma^2}.$$

Derivatives with respect to μ_1 and μ_2 give

$$\mu_1 = \frac{\sum\limits_{i=1}^{m} y^{(i)} \mathbb{1}\{x^{(i)} < c\}}{\sum\limits_{i=1}^{m} \mathbb{1}\{x^{(i)} < c\}},$$
$$\mu_2 = \frac{\sum\limits_{i=1}^{m} y^{(i)} \mathbb{1}\{x^{(i)} \geqslant c\}}{\sum\limits_{i=1}^{m} \mathbb{1}\{x^{(i)} \geqslant c\}}.$$

Differentiation with respect to c becomes more complicated, because the indicator function $\mathbb{1}$ is discontinuous. Another problem appears as in majority of cases there are more than one feature, and we should build log-likelihood with respect to all features. It becomes computationally infeasible to optimize the error function for the hypothesis $h_{c,t}(x) = \sum\limits_{k=1}^{r} c_k \mathbb{1}\{x \in R_k\}$ explicitly.

To resolve this problem we introduce the measure that will identify the best variable and the best split

$$Q_k(T) = \frac{1}{m_k} \sum_{i \in R_k} (y^{(i)} - \mu_k)^2,$$

where m_k is a number of training examples in the node R_k, and apply greedy algorithm that optimizes the tree step by step (not the hypothesis in general).

Algorithm 3 Greedy optimization algorithm for decision trees

```
1: initialize root node $R_1 = \mathbb{R}^n$, where $n$ is a number of features
2: initialize a list of terminal nodes $R = \{R_1\}$
3: repeat
4:   for each region $R_k$ from $R$ do
5:     for $j = 1$ to $n$ do
6:       for all splitting points $c$ do
7:         define
             $R_{kl} = \{x^{(i)} \in R_k \mid x_j^{(i)} < c\},$
             $R_{kr} = \{x^{(i)} \in R_k \mid x_j^{(i)} \geqslant c\}$
8:         evaluate $\mu_1 = \frac{1}{m_{kl}} \sum_{R_{kl}} y^{(i)}$, $\mu_2 = \frac{1}{m_{kr}} \sum_{R_{kr}} y^{(i)}$
9:         evaluate $\varepsilon = m_{kl} Q_{kl}(T) + m_{kr} Q_{kr}(T)$
10:    choose $j, c = \arg\min_{j,c} \varepsilon$
11:    remove $R_k$ from $R$; add $R_{kl}$, $R_{kr}$ to $R$
12: until convergence
13: return list of terminal nodes $R$
```

This algorithm when run till the end gives a huge decision tree with small number of training examples in the terminal nodes. Obviously, that the resulted error on the training set will be equal to zero. Remember that we called such situation overfitting (excellent predictions on the training set and very bad prediction on the test set). To avoid such situation we can introduce additional stopping criteria:

- minimal size of terminal node: for example, we can require to have at least 10 training examples in each terminal node
- minimal change of error: for example, we can require not to split the node in case if the error does not decrease more than 0.01

Then the above algorithm should be modified by adding one of these conditions (or both) before the lines 10 and 11.

Usually, the decision tree is constructed using two steps. First, we build the maximal decision tree and second, we start pruning the tree by removing terminal nodes one by one. The procedure of prunning is stopped when the following function reaches the minimal value:

$$C_\alpha(T) = \sum_{k=1}^{K} m_k Q_k(T) + \alpha K.$$

This function creates some trade-off between the tree size and its goodness of fit to the data. If α is zero then K increases till the errors $Q_k = 0$ for all k. But if $\alpha > 0$, then big value of K implies the big value for the second term in $C_\alpha(T)$, for example, if there exists big α such that the minimal value of C_α is obtained if $K = 1$. Usually, α is defined using cross-validation procedure.

2.2 Python implementation

First, we use the built-in implementation of the decision tree in Python.

```
In [12]: from sklearn.tree import DecisionTreeRegressor, export_graphviz
         from graphviz import Source
         import pydot
```

As a benchmark we use `boston` dataset from the `scikit-learn` package.

```
In [13]: boston = ds.load_boston()
         X = boston.data
         y = boston.target/50. # just for convenience
         X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                             test_size=0.33,
                                                             random_state=42)
```

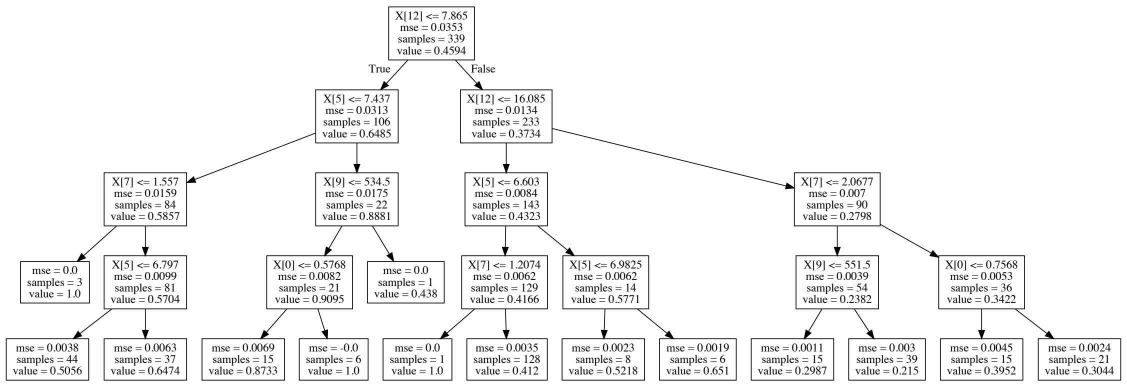
Fit the model (notice that we use parameter `max_depth` to restrict the tree size):

```
In [14]: clf = DecisionTreeRegressor(max_depth=4)
         clf = clf.fit(X_train, y_train)
```

And visualize the resulted tree:

```
In [15]: # produces dot file
         export_graphviz(clf, out_file='regression_tree_scikit.dot')
         dot = open('regression_tree_scikit.dot').read()
         dot = dot.replace('node [shape=box] ;',
              'node [shape=box] ;\ngraph [size="10.3,10.3!"];')
         dot = Source(dot)
         dot
```

```
Out[15]:
```



The predictions can be obtained as follows:

```
In [16]: y_pred = clf.predict(X_test)
         print rmse(y_test, y_pred)
```

```
0.0771597318202
```

We want to show a simple implementation of decision tree algorithm from scratch. There are different possibilities to implement the decision tree in Python. We will create two classes for node and tree, accordingly. `Node` class contains the description of each node and pointer to father and daughter nodes, and `RegressionTree` class contains all nodes and functions for splitting and predicton. We have added the function that saves tree structure in the dot file.

```
In [17]: class Node():
             def __init__(self, id, samples, prediction,
                          error, father_node=None):
                 if father_node is None: # the node is a root node
                     self.depth = 0
                 else:
                     self.depth = father_node.depth + 1
                 self.father_node = father_node
                 self.id = id
```

```
            self.samples = np.array(samples)
            self.prediction = prediction
            self.error = error
            self.j = None
            self.t = None

    class RegressionTree():
        def __init__(self, X, y, max_depth=5, min_samples_leaf = 1):
            self.X = X
            self.y = y
            self.max_depth = max_depth
            self.min_samples_leaf = min_samples_leaf
            self.nodes = []
            mu, err = self.error(y)
            self.nodes.append(Node(0, [x for x in range(X.shape[0])],
                                   mu, err))
            self.nodes_to_split = [0]

        def error(self, y):
            p = np.mean(y)
            return p, np.mean(np.power(y-p, 2))

        def split_next_node(self):
            if len(self.nodes_to_split) == 0:
                return
            node = self.nodes[self.nodes_to_split[0]]
            del self.nodes_to_split[0]
            if (node.depth >= self.max_depth or
                len(node.samples)<=2*self.min_samples_leaf-1):
                node.left = None
                node.right = None
                return
            best_j = -999
            best_t = -999
            best_se1 = node.error*len(node.samples)
            best_se2 = node.error*len(node.samples)
            best_mu1 = -999
            best_mu2 = -999
            for j in range(self.X.shape[1]):
                Xj = self.X[node.samples,j]
                Xj_min = np.min(Xj)
                Xj_max = np.max(Xj)
                for t in np.linspace(Xj_min, Xj_max, 300):
                    y_left = self.y[node.samples][Xj<t]
                    y_right = self.y[node.samples][Xj>=t]
                    if (len(y_left)<self.min_samples_leaf or
                        len(y_right)<self.min_samples_leaf):
                        continue
                    mu1, mse1 = self.error(y_left)
                    mu2, mse2 = self.error(y_right)
                    se1 = mse1*len(y_left)
                    se2 = mse2*len(y_right)
                    if (se1 + se2) < (best_se1 + best_se2):
                        best_j = j
                        best_t = t
                        samples_left = node.samples[Xj<t]
                        samples_right = node.samples[Xj>=t]
                        best_mu1 = mu1
                        best_mu2 = mu2
                        best_se1 = se1
                        best_se2 = se2
                        best_mse1 = mse1
                        best_mse2 = mse2
            if best_j == -999:
                node.left = None
                node.right = None
            else:
                node.left = Node(len(self.nodes), samples_left,
                                 best_mu1, best_mse1, node)
                node.right = Node(len(self.nodes)+1, samples_right,
                                  best_mu2, best_mse2, node)
                node.j = best_j
                node.t = best_t
                self.nodes.append(node.left)
                self.nodes.append(node.right)
                self.nodes_to_split.append(len(self.nodes)-2)
                self.nodes_to_split.append(len(self.nodes)-1)

        def predict(self, X):
            preds = []
            for i in range(X.shape[0]):
                node = self.nodes[0]
                while True:
                    if node.j is None:
                        preds.append(node.prediction)
                        break
                    if X[i, node.j] < node.t:
                        node = node.left
                    else:
                        node = node.right
            return preds

        def save_tree_to_dot(self, filename):
            def get_edge(x):
                if x.j is None:
                    label = 'mse = ' + str(np.round(x.error,3)) +\
                            '\n samples = ' + str(len(x.samples)) +\
                            '\n value = ' + str(np.round(x.prediction,3))
                else:
                    label = 'X[' + str(x.j) + '] < ' + str(np.round(x.t)) +\
                            '\n mse = ' + str(np.round(x.error,3)) +\
                            '\n samples = ' + str(len(x.samples)) +\
                            '\n value = ' + str(np.round(x.prediction,3))
                if x.father_node is None:
                    return str(x.id) + ' [ label="' + label + '"] ;\n'
                else:
                    return str(x.id) + ' [ label="' + label + '"] ;\n' +\
                                     str(x.father_node.id) +\
                                     ' -> ' + str(x.id) + ' ;\n'

            f = open(filename, 'w')
            f.write('digraph Tree {\n node [shape = box] ;\n')
            f.write(' graph [size="9.7,8.3!"];\n')

            for x in self.nodes:
                f.write(get_edge(x))
            f.write('}')
            f.close()
```

```
In [18]: tree = RegressionTree(X_train, y_train,
                               max_depth=4, min_samples_leaf = 4)
         while len(tree.nodes_to_split)>0:
             tree.split_next_node()
         tree.save_tree_to_dot('regression_tree_manual.dot')
```

Check the error on the test set:

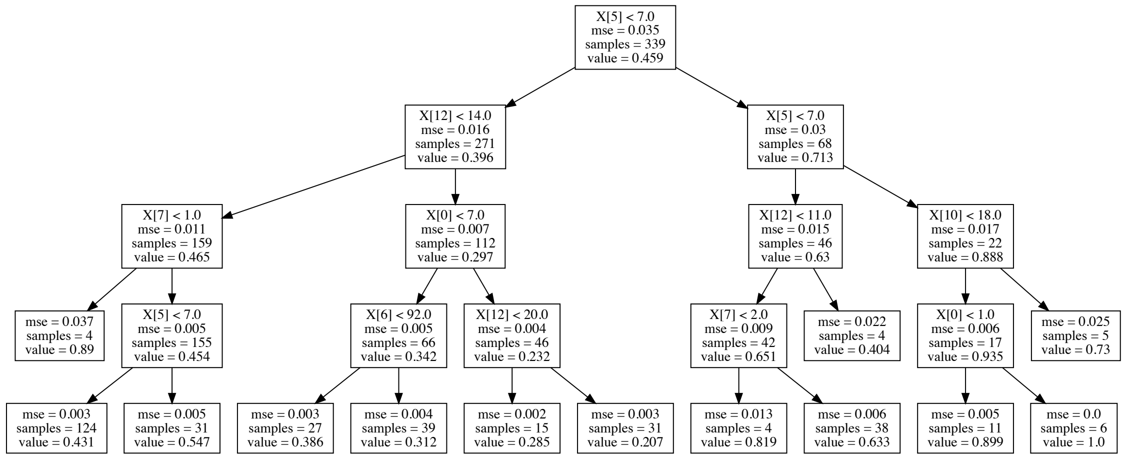
```
In [19]: rmse(y_test, tree.predict(X_test))
```

```
Out[19]: 0.07209792520517698
```

We can also draw the tree from the dot file (remember that we saved tree in the dot file already):

```
In [20]: dot = Source(open('regression_tree_manual.dot').read())
         dot
```

```
Out[20]:
```



2.3 Classification trees

The main difference between regression and classification decision trees is the way to define the quality of split. If we have several classes $1, 2, \dots, D$, then it does not make sense to calculate the mean squared error $Q_k(T)$ like we did for the regression trees. Consider the regions $R_1, R_2, \dots, R_K$, obtained by the decision tree. We could evaluate the quantities

$$p_{kd} = \frac{1}{m_k} \sum_{i \in R_k} \mathbb{1}\{y^{(i)} = d\} \text{ (percentage of class } d \text{ in the region } R_k),$$

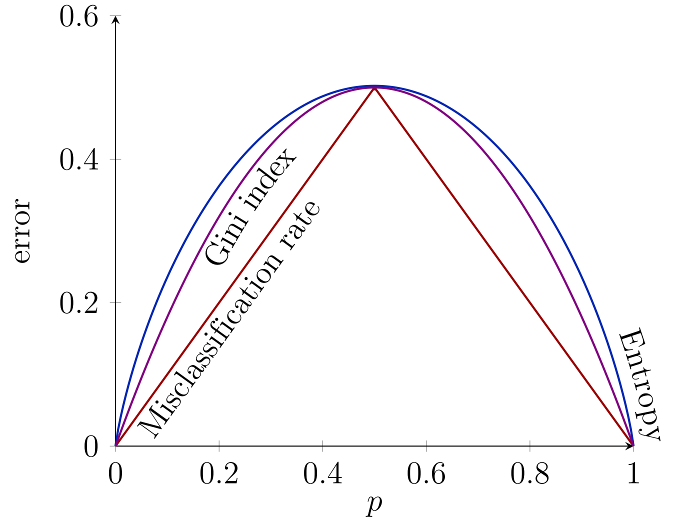
where as before m_k is a number of training examples in the region R_k, and $d \in \{1, 2, \dots, D\}$. There are several criteria for quality of split can be used:

- **Misclassification error**: denote $d(k) = \arg\max_d p_{kd}$ (the majority class in the region R_k), then
$$Q_k(T) = \frac{1}{m_k} \sum_{i \in R_k} \mathbb{1}\{y^{(i)} \neq d(k)\} = 1 - p_{k\,d(k)}.$$
- **Gini index**:
$$Q_k(T) = \sum_{d=1}^{D} p_{kd}(1 - p_{kd}).$$
- **Cross-entropy (or deviance)**:
$$Q_k(T) = -\sum_{d=1}^{D} p_{kd} \ln p_{kd}.$$

For example, if we have two classes only ($D = 2$) and $p_{k1} = p$, then

- Misclassification error: $\min(p, 1-p)$

- Gini index: $2p(1-p)$
- Cross-entropy: $-p\ln p-(1-p)\ln(1-p)$

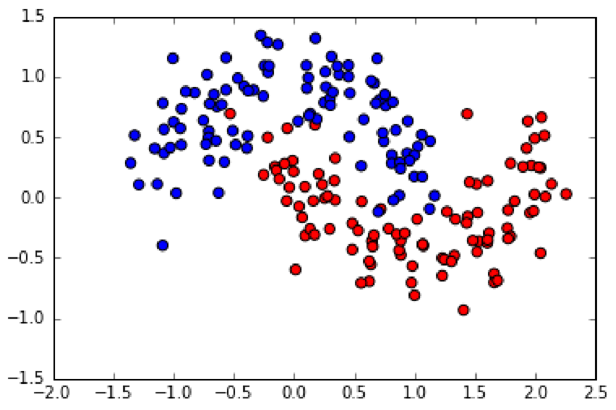


To build the classification decision tree we should choose the error $Q_k(T)$ and run the Algorithm 3.

2.4 Python implementation

As a benchmark we use `make_moons` data:

```
In [21]: np.random.seed(0)
         X, y = ds.make_moons(200, noise=0.20)
         plt.scatter(X[:,0], X[:,1], s=40, c=y, cmap=cm.bwr)
         X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                              test_size=0.33,
                                                              random_state=42)
```

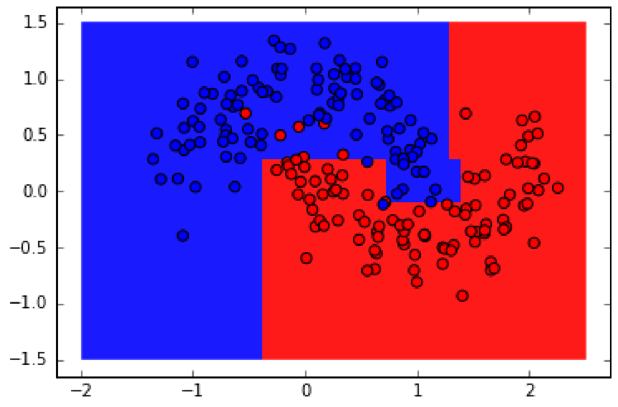


The following function `plot_decision_boundary()` visualizes the result of our classification:

```
In [22]: def plot_decision_boundary(model, xmin=-0.1, xmax=1.1,
                                    ymin=-0.1, ymax=1.1):
             fig = plt.figure(figsize=(6,6))
             x1, x2 = np.meshgrid(np.linspace(xmin, xmax, 500),
                                  np.linspace(ymin, ymax, 500))
             ypred = model.predict(np.c_[x1.ravel(), x2.ravel()])
             ypred = ypred.reshape(x1.shape)
             extent = xmin, xmax, ymin, ymax

             plt.imshow(ypred, cmap=cm.bwr, alpha=.9,
                        interpolation='bilinear',
                        extent = extent, origin='lower')
             plt.scatter(X[:,0], X[:,1], s=40, c=y, cmap=cm.bwr)
```

The `scikit-learn` has its own implementation of classification decision trees:

```
In [23]: from sklearn.tree import DecisionTreeClassifier

In [24]: clf = DecisionTreeClassifier(max_depth=10,
                                      min_samples_leaf=4)
         clf = clf.fit(X_train, y_train)

In [25]: plot_decision_boundary(clf, xmin=-2.0, xmax=2.5,
                                ymin=-1.5, ymax=1.5)
```



The main difference between regression and classification decision trees is the metrics used to estimate the quality of split. In regression decision tree we used a mean squared error, in classification decision tree we will use Gini index, cross-entropy or misclassification rate:

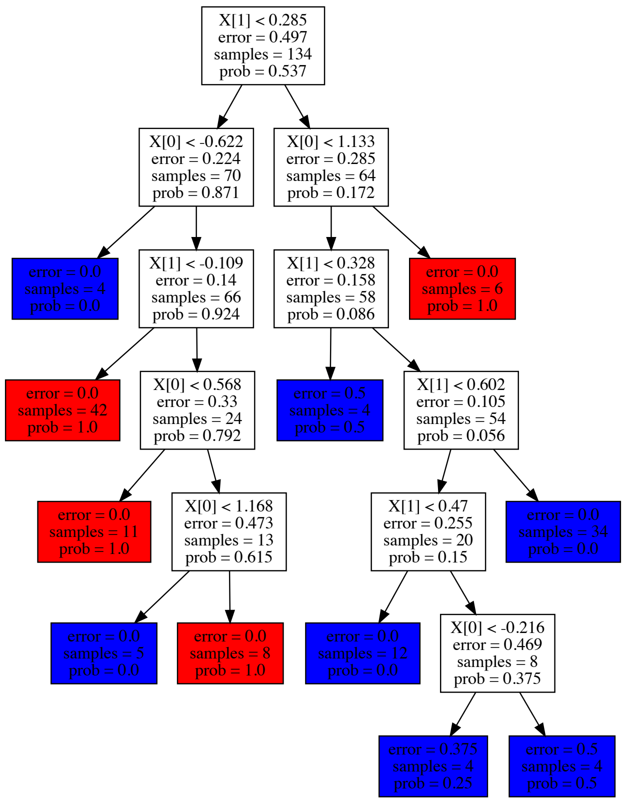
```
In [26]: class ClassificationTree():
             def __init__(self, X, y, error_type='gini',
                          max_depth=5, min_samples_leaf = 1):
                 self.X = X
```

```
        self.y = y
        self.error_type = error_type
        self.max_depth = max_depth
        self.min_samples_leaf = min_samples_leaf
        self.nodes = []
        mu, err = self.error(y)
        self.nodes.append(Node(0, [x for x in range(X.shape[0])],
                               mu, err))
        self.nodes_to_split = [0]

    def error(self, y):
        p = np.mean(y)
        if self.error_type == 'entropy':
            p = np.clip(p, 0.01, 0.99)
            return p, -1.0*p*np.log(p)-(1-p)*np.log(1-p)
        elif self.error_type == 'gini':
            return p, 2.0*p*(1.0-p)
        elif self.error_type == 'misclass':
            return p, min(p, 1-p)

    def split_next_node(self):
        if len(self.nodes_to_split) == 0:
            return
        node = self.nodes[self.nodes_to_split[0]]
        del self.nodes_to_split[0]
        if (node.depth >= self.max_depth
            or len(node.samples)<=2*self.min_samples_leaf-1
            or node.error == 0.0):
            node.left = None
            node.right = None
            return
        best_j = -999
        best_t = -999
        best_weighted_err1 = node.error*len(node.samples)
        best_weighted_err2 = node.error*len(node.samples)
        best_mu1 = -999
        best_mu2 = -999
        for j in range(self.X.shape[1]):
            Xj = self.X[node.samples,j]
            Xj_min = np.min(Xj)
            Xj_max = np.max(Xj)
            for t in np.linspace(Xj_min, Xj_max, 300):
                y_left = self.y[node.samples][Xj<t]
                y_right = self.y[node.samples][Xj>=t]
                if (len(y_left)<self.min_samples_leaf
                    or len(y_right)<self.min_samples_leaf):
                    continue
                mu1, err1 = self.error(y_left)
                mu2, err2 = self.error(y_right)
                weighted_err1 = err1*len(y_left)
                weighted_err2 = err2*len(y_right)
                if (weighted_err1 + weighted_err2 <
                    best_weighted_err1 + best_weighted_err2):
                    best_j = j
                    best_t = t
                    samples_left = node.samples[Xj<t]
                    samples_right = node.samples[Xj>=t]
                    best_mu1 = mu1
                    best_mu2 = mu2
                    best_err1 = err1
                    best_err2 = err2
                    best_weighted_err1 = weighted_err1
                    best_weighted_err2 = weighted_err2
        if best_j == -999:
            node.left = None
            node.right = None
        else:
            node.left = Node(len(self.nodes), samples_left,
                             best_mu1, best_err1, node)
            node.right = Node(len(self.nodes)+1, samples_right,
                              best_mu2, best_err2, node)
            node.j = best_j
            node.t = best_t
            self.nodes.append(node.left)
            self.nodes.append(node.right)
            self.nodes_to_split.append(len(self.nodes)-2)
            self.nodes_to_split.append(len(self.nodes)-1)

    def predict(self, X, probability = False):
        preds = []
        for i in range(X.shape[0]):
            node = self.nodes[0]
            while True:
                if node.j is None:
                    preds.append(node.prediction)
                    break
                if X[i, node.j] < node.t:
                    node = node.left
                else:
                    node = node.right
        if probability == False:
            preds = np.round(preds)
        return np.array(preds)

    def save_tree_to_dot(self, filename):
        def get_edge(x):
            if x.j is None:
                label = 'error = ' + str(np.round(x.error,3)) +\
                        '\n samples = ' + str(len(x.samples)) +\
                        '\n prob = ' + str(np.round(x.prediction,3))
                if x.prediction > 0.5:
                    color = 'red'
                else:
                    color = 'blue'
            else:
                label = 'X['+str(x.j)+'] < '+str(np.round(x.t,3))+\
                        '\n error = '+str(np.round(x.error,3))+\
                        '\n samples = '+str(len(x.samples))+\
                        '\n prob = '+str(np.round(x.prediction,3))
                color = 'white'
            if x.father_node is None:
                return str(x.id)+' [ style=filled fillcolor='+color+\
                       ' label="'+label+'"] ;\n'
            else:
                return str(x.id)+' [ style=filled fillcolor='+color+\
                       ' label="'+label+'"] ;\n'+\
                       str(x.father_node.id)+' -> '+str(x.id)+' ;\n'

        f = open(filename, 'w')
        f.write('digraph Tree {\n node [shape = box] ;\n')
        for x in tree.nodes:
            f.write(get_edge(x))
        f.write('}')
        f.close()
```

```
In [27]: tree = ClassificationTree(X_train, y_train,
                                   error_type='gini',
                                   max_depth=10, min_samples_leaf = 4)
         while len(tree.nodes_to_split)>0:
             tree.split_next_node()
         tree.save_tree_to_dot('classification_tree_manual.dot')
```

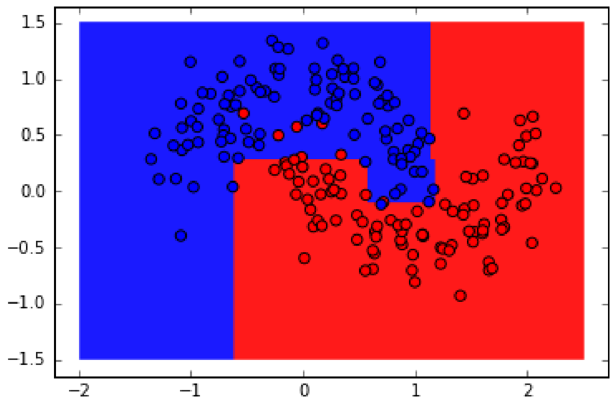
```
In [28]: dot = Source(open('classification_tree_manual.dot').read())
         dot
```

```
Out[28]:
```

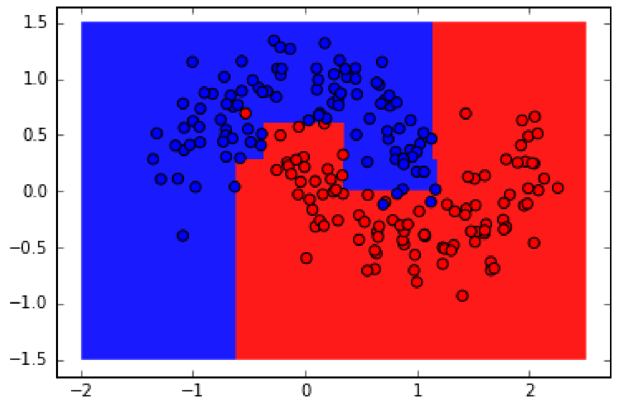


Compare the decision boundaries for different error types:

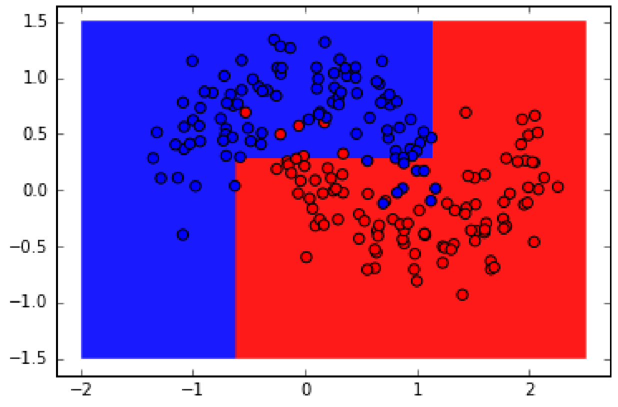
```
In [29]: tree = ClassificationTree(X_train, y_train,
                                   error_type='gini', max_depth=10,
                                   min_samples_leaf = 4)
         while len(tree.nodes_to_split)>0:
             tree.split_next_node()
         plot_decision_boundary(tree,
                                xmin=-2.0, xmax=2.5,
                                ymin=-1.5, ymax=1.5)
```



```
In [30]: tree = ClassificationTree(X_train, y_train,
                                   error_type='entropy', max_depth=10,
                                   min_samples_leaf = 4)
         while len(tree.nodes_to_split)>0:
             tree.split_next_node()
         plot_decision_boundary(tree,
                                xmin=-2.0, xmax=2.5,
                                ymin=-1.5, ymax=1.5)
```



```
In [31]: tree = ClassificationTree(X_train, y_train,
                                   error_type='misclass', max_depth=10,
                                   min_samples_leaf = 4)
         while len(tree.nodes_to_split)>0:
             tree.split_next_node()
         plot_decision_boundary(tree,
                                xmin=-2.0, xmax=2.5,
                                ymin=-1.5, ymax=1.5)
```



3 Boosting

3.1 Exponential loss

The natural generalization for the additive models can be obtained by

$$\eta = \sum_{k=1}^{K} \beta_k b_k(x, \theta_k). \tag{1}$$

For classification problems as before we find the probability by applying the sigmoid function:

$$\mu = g\left(\sum_{k=1}^{K} \beta_k b_k(x, \theta_k)\right) = g(\eta),$$

where

$$g(z) = \frac{1}{1+e^{-2z}}$$

(the coefficient 2 is introduced for convenience).

As before we assume

$$y \mid \beta_k; b_k; \theta_k \sim \mathrm{Ber}(\mu),$$

and log-likelihood is expressed as:

$$l(\beta_k, b_k, \theta_k) = \sum_{i=1}^{m} y^{(i)} \ln \mu^{(i)} + (1 - y^{(i)}) \ln(1 - \mu^{(i)}).$$

Consider the function

$$h(\mu) = y \ln \mu + (1-y)\ln(1-\mu).$$

We introduce the transformed target by $z = 2y - 1 \Leftrightarrow y = \dfrac{z+1}{2}$ (with this notation if $y \in \{0, 1\}$, then $z \in \{-1, 1\}$), then

$$\begin{aligned} h(\mu) &= \frac{z+1}{2} \ln \frac{1}{1+e^{-2\eta}} + \left(1 - \frac{z+1}{2}\right) \ln\left(1 - \frac{1}{1+e^{-2\eta}}\right) = \\ &= \frac{1}{2}\left(-z\ln(1+e^{-2\eta}) - \ln(1+e^{-2\eta}) - 2\eta + 2z\eta - \ln(1+e^{-2\eta}) + z\ln(1+e^{-2\eta})\right) = \\ &= -(\ln(1+e^{-2\eta}) + \eta(1-z)) = -\ln\left((1+e^{-2\eta})e^{\eta(1-z)}\right) = -\ln\left(1+e^{-2z\eta}\right). \end{aligned}$$

In the last transition we use the fact that $z \in \{-1, 1\}$:

$$\begin{aligned} &\text{if } z = 1, \text{ then } \ln\left((1+e^{-2\eta})e^{\eta(1-z)}\right) = \ln\left(1+e^{-2\eta}\right) = \ln\left(1+e^{-2z\eta}\right), \\ &\text{if } z = -1, \text{ then } \ln\left((1+e^{-2\eta})e^{\eta(1-z)}\right) = \ln\left(e^{2\eta}+1\right) = \ln\left(1+e^{-2z\eta}\right). \end{aligned}$$

With the obtained expression for the likelihood we can formulate the maximum likelihood optimization problem in the simpler way:

$$\arg\max_{\beta_k, b_k, \theta_k} l(\beta_k, b_k, \theta_k) = \arg\min_{\beta_k, b_k, \theta_k} e^{-z\eta},$$

where η is defined with (1) and $z = 2y - 1$ is transformed target. The function

$$L(z, \eta) = \sum_{i=1}^{m} \exp\left(-z^{(i)}\eta^{(i)}\right)$$

is called an **exponential loss**.

3.2 Adaboost

Boosting is the example of algorithm ensembling. The idea of ensembling is to combine different algorithms to increase the prediction accuracy. The simplest example of ensembling is to take the output from two algorithms (for example, SVM and neural net) and take the average of two predictions. Boosting is a very powerful algorithm that helps to combine "weak" models (the models with accuracy just a little bit higher than random guess).

Consider the equation (1) where we replace basis functions $b_k(x, \theta)$ by some classifiers $G_k(x) \in \{-1, 1\}$:

$$\eta = \sum_{k=1}^{K} \beta_k G_k(x). \tag{2}$$

We will use stagewise approach to find weights β_k and functions G_k in (2), the idea is similar to the backfitting algorithm for generalized additive models. Assuming that we have built the models $G_k(x)$ with weights β_k, $k = 1, \dots, K$, we try to find the next weight β and classifier $G(x)$ using exponential loss function:

$$\begin{aligned}
&\arg\min_{\beta,G} L(z, \eta) = \arg\min_{\beta,G} \sum_{i=1}^{m} \exp\left(-z^{(i)}\eta^{(i)}\right) = \\
&= \arg\min_{\beta,G} \sum_{i=1}^{m} \exp\left(-z^{(i)}\left(\sum_{k=1}^{K} \beta_k G_k(x^{(i)}) + \beta G(x^{(i)})\right)\right) = \\
&= \arg\min_{\beta,G} \sum_{i=1}^{m} w^{(i)} \exp\left(-z^{(i)}\beta G(x^{(i)})\right),
\end{aligned}$$

where

$$w^{(i)} = \exp\left(-z^{(i)} \sum_{k=1}^{K} \beta_k G_k(x^{(i)})\right) \tag{3}$$

are constants during the algorithm step and can be considered as weights.

If the classifier G predicts the training example i correctly, then

$$z^{(i)}G(x^{(i)}) = 1,$$

otherwise,

$$z^{(i)}G(x^{(i)}) = -1.$$

Hence,

$$\begin{aligned}
&\arg\min_{\beta,G}\left(\sum_{i=1}^{m} w^{(i)} \exp\left(-z^{(i)}\beta G(x^{(i)})\right)\right) = \\
&= \arg\min_{\beta,G}\left(\sum_{i:\, G(x^{(i)})=z^{(i)}} w^{(i)}e^{-\beta} + \sum_{i:\, G(x^{(i)})\neq z^{(i)}} w^{(i)}e^{\beta}\right) = \\
&= \arg\min_{\beta,G}\left((e^{\beta} - e^{-\beta}) \cdot \sum_{i=1}^{m} w^{(i)}\mathbb{1}\{z^{(i)} \neq G(x^{(i)}\} + e^{-\beta} \cdot \sum_{i=1}^{m} w^{(i)}\right).
\end{aligned}$$

This formula gives the following results:

- if β is fixed, then $G = \arg\min\limits_G \sum\limits_{i=1}^{m} w^{(i)} \mathbb{1}\{z^{(i)} \neq G(x^{(i)})\}$. In other words, our classifier G should minimize the sum of weights for wrongly predicted training examples.
- if G is fixed, then calculating the derivative with respect to β and assigning it to zero gives:

$$\beta = \frac{1}{2} \ln \frac{1-r}{r}, \text{ where } r = \frac{\sum\limits_{i=1}^{m} w^{(i)} \mathbb{1}\{z^{(i)} \neq G(x^{(i)})\}}{\sum\limits_{i=1}^{m} w^{(i)}}. \tag{4}$$

  This result has a perfect common sense: if $r \to \frac{1}{2}$ (random classifier), then $\beta \to 0$; if $r \to 0$ (perfect classifier), then $\beta \to \infty$. As we can see the algorithm will give higher weight β to more accurate classifier G.

We can also notice that weights $w^{(i)}$ can be updated from previous step with the formula (3). Indeed,

$$w_{new}^{(i)} = \exp\left(-z^{(i)} \sum_{k=1}^{K-1} \beta_k G_k(x^{(i)}) - z^{(i)} \beta_K G_K(x^{(i)})\right) =$$
$$= w_{old}^{(i)} \cdot \exp\left(-z^{(i)} \beta_K G_K(x^{(i)})\right),$$

but $-z^{(i)} G_K(x^{(i)}) = 2 \cdot \mathbb{1}\{z^{(i)} \neq G_K(x^{(i)})\} - 1$, thus

$$w_{new}^{(i)} = w_{old}^{(i)} \cdot e^{\alpha_K \mathbb{1}\{z^{(i)} \neq G_K(x^{(i)})\}} \cdot e^{-\beta_K},$$

where $\alpha_K = 2\beta_K$ and finally, we can remove the term $e^{-\beta_K}$, because it multiplies all weights by the same factor:

$$w_{new}^{(i)} = w_{old}^{(i)} \cdot e^{\alpha_K \mathbb{1}\{z^{(i)} \neq G_K(x^{(i)})\}} \tag{5}$$

If we start fitting with the constant weights for all training examples (for example, $w^{(i)} = \frac{1}{m}$ for all $i = 1, \ldots, m$, then after each iteration k we multiply weight for the training example i by the factor

$$e^{\alpha_k \mathbb{1}\{z^{(i)} \neq G_k(x^{(i)})\}} = \begin{cases} 1, & \text{if } G_k \text{ classifies the example } i \text{ correctly} \\ e^{\alpha_k}, & \text{otherwise.} \end{cases}$$

Now we can formulate the first boosting algorithm (Adaboost.M1):

Algorithm 4 Adaboost.M1

```
1: Initialize weights $w^{(i)} = \frac{1}{m}$, $i = 1, 2, \ldots, m$
2: for $k = 1$ to $K$ do
3:     Fit a classifier $G_k(x)$ to the training data with weights $w^{(i)}$
4:     Compute $r = \frac{\sum_{i=1}^{m} w^{(i)} \mathbb{1}\{z^{(i)} \neq G(x^{(i)})\}}{\sum_{i=1}^{m} w^{(i)}}$    ▷ (4)
5:     Compute $\alpha_k = \ln \frac{1-r}{r}$ (weight for the classifier $G_k$)
6:     Set $w^{(i)} := w^{(i)} \cdot \exp\left(\alpha_k \cdot \mathbb{1}\{z^{(i)} \neq G_k(x^{(i)})\}\right)$, $i = 1, \ldots, m$    ▷ (5)
7: return $G(x) = \text{sign}\left(\sum_{k=1}^{K} \alpha_k G_k(x)\right)$
```

4 Boosting trees

4.1 Gradient boosting

In this section we combine the ideas of backfitting algorithm and decision tree. First, we introduce some loss function $L(f)$, for different problems we can choose squared loss function (for regression)

$$L(f) = \frac{1}{2m} \sum_{i=1}^{m} (y^{(i)} - f(x^{(i)}))^2,$$

or cross-entropy loss (for classification)

$$L(f) = -\frac{1}{m} \sum_{i=1}^{m} \left(y^{(i)} \ln f(x^{(i)}) + (1 - y^{(i)}) \ln(1 - f(x^{(i)}))\right),$$

or any other convenient loss function. The comparison of the Algorithms 1 and 2 can give the following observations:

- In the Algorithm 2 we try to find minimum of the loss function as a function of f. Replacing the Newton's method by Gradient Descent gives rise to the updating step:

$$f^{new} = f^{old} - \alpha \frac{dL}{df}\bigg|_{f=f^{old}}, \tag{6}$$

  where f^{old} is a vector of current values of f on all training examples, $\frac{dL}{df}\Big|_{f=f^{old}}$ is a derivative of the loss function $L(f)$ with respect to f calculated at all current values f^{old}, α is a learning rate.

- In the Algorithm 1 we find the new value for f as

$$f^{new} = f^{old} + g, \tag{7}$$

where g is fitted using some building block algorithm (for example, univariate cubic splines) with previous residues as as working target.

The equations (6) and (7) give a new idea: find the function g by fitting the building block algorithm to the working target $r = -\left.\frac{dL}{df}\right|_{f=f^{old}}$ and find the coefficient α as

$$\alpha = \arg\min_{\alpha} L(f^{old} + \alpha g).$$

This idea is called **gradient boosting**.

4.2 Gradient tree boosting

Gradient boosting can be easily implemented when the building block algorithm is a classification or decision tree.

Algorithm 5 Gradient Tree Boosting

```
1: Initialize $f_0(x) = \arg\min_{\mu} \sum_{i=1}^{m} L(y^{(i)}, \mu)$
2: for $k = 1$ to $K$ do
3:     Compute working target $r_k^{(i)} = -\left.\left(\frac{dL}{df}\right)\right|_{f=f_{k-1}(x^{(i)})}$ for all $i = 1, \dots, m$
4:     Fit a regression tree to the targets $r_k^{(i)}$ with terminal nodes $R_{kj}$, $j = 1, \dots, J_k$
5:     Compute
           $$\gamma_{kj} = \arg\min_{\gamma} \sum_{x^{(i)} \in R_{kj}} L(y^{(i)}, f_{k-1}(x^{(i)}) + \gamma)$$
       for all $j = 1, \dots, J_k$
6:     Update $f_k(x) = f_{k-1}(x) + \sum_{j=1}^{J_k} \gamma_{kj} \mathbb{1}\{x \in R_{kj}\}$
7: return $f_K(x)$
```

In the line 5 of the algorithm we should find the best value for γ in each terminal node (for squared loss function it will be just an average of working target).

The classification algorithm is similar except that in the line 4 we should fit a classification tree and also we should repeat lines 3-6 for each class.

Exercise. Write down the pseudocode for the Gradient Tree Boosting for the classification problem with

a) two classes;

b) D classes.

4.3 Python implementation

As a benchmark we use the `boston` data from the `scikit-learn` package.

```
In [32]: boston = ds.load_boston()
         X = boston.data
         y = boston.target/50.
         X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                             test_size=0.33,
                                                             random_state=42)
```

```
In [33]: print 'Number of training examples: ' + str(X.shape[0])
         print 'Number of features: ' + str(X.shape[1])
```

```
Number of training examples: 506
Number of features: 13
```

Now we train the gradient trees boosting model implemented in the package `xgboost`:

```
In [34]: import xgboost as xgb

         param = {}
         param['objective'] = 'reg:linear'
         param['eta'] = 0.03
         param['max_depth'] = 10
         param['eval_metric'] = 'rmse'
         param['silent'] = 1
         param['nthread'] = 8
         param['gamma'] = 1.0
         param['lambda'] = 0.0
         param['min_child_weight'] = 1
         param['subsample'] = 0.8
         param['colsample_bytree'] = 1.0
         num_round = 1000
         param['seed'] = 2657
         plst = list(param.items())

         xgmat_train = xgb.DMatrix(X_train, label=y_train,
                                   missing = -999.0)
         bst = xgb.train( plst, xgmat_train, num_round );
         xgmat_test = xgb.DMatrix(X_test, missing = -999.0)
         y_pred = bst.predict( xgmat_test )
         print rmse(y_test, y_pred)
```

```
0.108290326706
```

We will implement gradient tree boosting using `DecisionTreeRegressor` class from the `scikit-learn`. Notice that we did not use a lot of tricks (like shrinkage, regularization and so on).

```
In [35]: from sklearn.tree import DecisionTreeRegressor
```

```
In [36]: def dLdf(X, y, f0, f):
             if len(f) > 0:
                 ypred = np.sum(np.array([fj.predict(X) for fj in f]).T,
                                axis=1) + f0
             else:
                 ypred = np.ones_like(y) * f0
             return y - ypred

         def gradient_boosting_trees(X, y, ntrees=100, max_depth=3):
             f0 = np.mean(y)
             nsamples = X.shape[0]
             f = []
             for k in range(ntrees):
                 # define the working target
                 r = dLdf(X, y, f0, f)
                 clf = DecisionTreeRegressor(max_depth=max_depth)
                 clf.fit(X, r)
                 f.append(clf)
             return f0, f
```

```
In [37]: f0, f = gradient_boosting_trees(X_train, y_train,
                                         ntrees=100, max_depth=2)
```

```
In [38]: ypred = np.sum(np.array([fj.predict(X_test) for fj in f]).T,
                        axis=1) + f0
         rmse(y_test, ypred)
```

```
Out[38]: 0.075087791829280118
```

Compared to the single decision tree, the accuracy is better:

```
In [39]: clf = DecisionTreeRegressor(max_depth=5)
         clf.fit(X_train, y_train)
         ypred = clf.predict(X_test)
         rmse(y_test, ypred)
```

```
Out[39]: 0.080012682023411025
```

References

[1] T. Hastie and R. Tibshirani. "Generalized additive models", *in Statistical Science, Vol.1, No.3*, 1998, pp. 297-318