lecture17-boosting

October 28, 2021

1 Lecture 17: Boosting

1.0.1 Applied Machine Learning

Volodymyr Kuleshov Cornell Tech

2 Part 1: Boosting and Ensembling

We are now going to look at ways in which multiple machine learning can be combined.

In particular, we will look at a way of combining models called *boosting*.

3 Review: Components of A Supervised Machine Learning Problem

At a high level, a supervised machine learning problem has the following structure:

$$\underbrace{\text{Training Dataset}}_{\text{Attributes + Features}} + \underbrace{\text{Learning Algorithm}}_{\text{Model Class + Objective + Optimizer}} \rightarrow \text{Predictive Model}$$

4 Review: Overfitting

Overfitting is one of the most common failure modes of machine learning. * A very expressive model (a high degree polynomial) fits the training dataset perfectly. * The model also makes wildly incorrect prediction outside this dataset, and doesn't generalize.

5 Review: Bagging

The idea of *bagging* is to reduce *overfitting* by averaging many models trained on random subsets of the data.

```
for i in range(n_models):
    # collect data samples and fit models
    X_i, y_i = sample_with_replacement(X, y, n_samples)
    model = Model().fit(X_i, y_i)
    ensemble.append(model)
```

```
# output average prediction at test time:
y_test = ensemble.average_prediction(y_test)
```

The data samples are taken with replacement and known as bootstrap samples.

6 Review: Underfitting

Underfitting is another common problem in machine learning. * The model is too simple to fit the data well (e.g., approximating a high degree polynomial with linear regression). * As a result, the model is not accurate on training data and is not accurate on new data.

7 Boosting

The idea of *boosting* is to reduce *underfitting* by combining models that correct each others' errors.

- As in bagging, we combine many models g_t into one *ensemble* f.
- Unlike bagging, the g_t are small and tend to underfit.
- Each g_t fits the points where the previous models made errors.

8 Weak Learners

A key ingredient of a boosting algorithm is a *weak learner*.

- Intuitively, this is a model that is slightly better than random.
- Examples of weak learners include: small linear models, small decision trees.

9 Structure of a Boosting Algorithm

The idea of *boosting* is to reduce *underfitting* by combining models that correct each others' errors.

1. Fit a weak learner g_0 on dataset $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}$. Let $f = g_0$.
2. Compute weights $w^{(i)}$ for each i based on model predictions $f(x^{(i)})$ and targets $y^{(i)}$. Give more weight to points with errors.
3. Fit another weak learner g_1 on $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}$ with weights $w^{(i)}$.
4. Set $f_1 = g_0 + \alpha_1 g$ for some weight α_1. Go to Step 2 and repeat.

In Python-like pseudocode this looks as follows:

```
weights, ensemble = np.ones(n_data,), Ensemble([])
for i in range(n_models):
    model = SimpleBaseModel().fit(X, y, weights)
    predictions = ensemble.predict(X)
    model_weight, weights = update_weights(weights, predictions)
    ensemble.add(model, model_weight)

# output consensus prediction at test time:
y_test = ensemble.predict(y_test)
```

10 Origins of Boosting

Boosting algorithms were initially developed in the 90s within theoretical machine learning.

- Originally, boosting addressed a theoretical question of whether weak learners with >50% accuracy can be combined to form a strong learner.
- Eventually, this research led to a practical algorithm called *Adaboost*.

Today, there exist many algorithms that are considered types of boosting, even though they're not derived from the perspective of theoretical ML.

11 Algorithm: Adaboost

- **Type**: Supervised learning (classification).
- **Model family**: Ensembles of weak learners (often decision trees).
- **Objective function**: Exponential loss.
- **Optimizer**: Forward stagewise additive model building.

12 Defining Adaboost

One of the first practical boosting algorithms was *Adaboost*.

We start with uniform $w^{(i)} = 1/n$ and $f = 0$. Then for $t = 1, 2, ..., T$:

1. Fit weak learner g_t on $\mathcal{D}$ with weights $w^{(i)}$.
2. Compute misclassification error $e_t = \frac{\sum_{i=1}^{n} w^{(i)} \mathbb{I}\{y^{(i)} \neq f(x^{(i)})\}}{\sum_{i=1}^{n} w^{(i)}}$
3. Compute model weight $\alpha_t = \log[(1 - e_t)/e_t]$. Set $f \leftarrow f + \alpha_t g_t$.
4. Compute new data weights $w^{(i)} \leftarrow w^{(i)} \exp[\alpha_t \mathbb{I}\{y^{(i)} \neq f(x^{(i)})\}]$.

13 Adaboost: An Example

Let's implement Adaboost on a simple dataset to see what it can do.

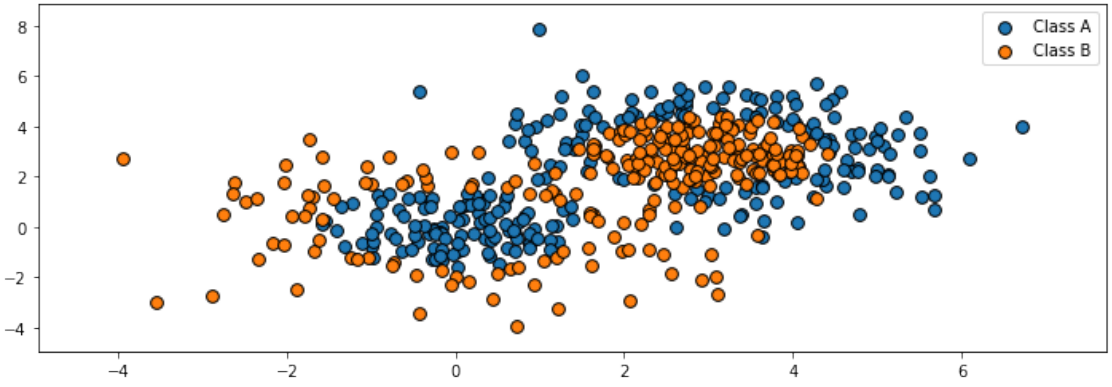
Let's start by creating a classification dataset.

[1]:
```
# https://scikit-learn.org/stable/auto_examples/ensemble/plot_adaboost_twoclass.
↪html
import numpy as np
from sklearn.datasets import make_gaussian_quantiles

# Construct dataset
X1, y1 = make_gaussian_quantiles(cov=2., n_samples=200, n_features=2,␣
↪n_classes=2, random_state=1)
X2, y2 = make_gaussian_quantiles(mean=(3, 3), cov=1.5, n_samples=300,␣
↪n_features=2, n_classes=2, random_state=1)
X = np.concatenate((X1, X2))
y = np.concatenate((y1, - y2 + 1))
```

We can visualize this dataset using `matplotlib`.

[15]:
```
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = [12, 4]

# Plot the training points
plot_colors, plot_step, class_names = "br", 0.02, "AB"
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1

for i, n, c in zip(range(2), class_names, plot_colors):
    idx = np.where(y == i)
    plt.scatter(X[idx, 0], X[idx, 1], cmap=plt.cm.Paired, s=60, edgecolor='k',␣
 ↪label="Class %s" % n)
plt.xlim(x_min, x_max)
plt.ylim(y_min, y_max)
plt.legend(loc='upper right')
```

[15]:
```
<matplotlib.legend.Legend at 0x12afda198>
```



Let's now train Adaboost on this dataset.

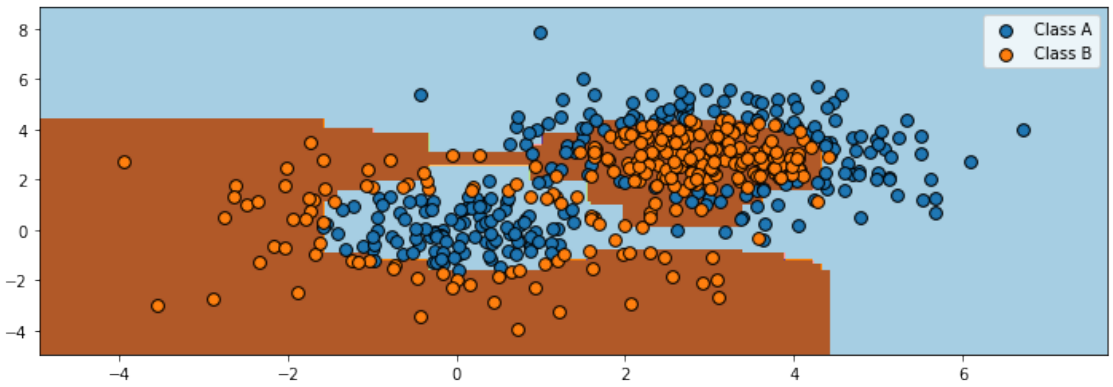
[12]:
```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier

# Create and fit an AdaBoosted decision tree
bdt = AdaBoostClassifier(DecisionTreeClassifier(max_depth=1),
                         algorithm="SAMME",
                         n_estimators=200)
bdt.fit(X, y)
```

[12]:
```
AdaBoostClassifier(algorithm='SAMME',
                   base_estimator=DecisionTreeClassifier(max_depth=1),
                   n_estimators=200)
```

Visualizing the output of the algorithm, we see that it can learn a highly non-linear decision boundary to separate the two classes.

[14]:
```
xx, yy = np.meshgrid(np.arange(x_min, x_max, plot_step), np.arange(y_min,␣
↪y_max, plot_step))

# plot decision boundary
Z = bdt.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
cs = plt.contourf(xx, yy, Z, cmap=plt.cm.Paired)

# plot training points
for i, n, c in zip(range(2), class_names, plot_colors):
    idx = np.where(y == i)
    plt.scatter(X[idx, 0], X[idx, 1], cmap=plt.cm.Paired, s=60, edgecolor='k',␣
↪label="Class %s" % n)
plt.xlim(x_min, x_max)
plt.ylim(y_min, y_max)
plt.legend(loc='upper right')
```

[14]:
```
<matplotlib.legend.Legend at 0x12b3b8438>
```



14 Ensembling

Boosting and bagging are special cases of *ensembling*.

The idea of ensembling is to combine many models into one. Bagging and Boosting are ensembling techniques to reduce over- and under-fitting.

- In stacking, we train m independent models $g_j(x)$ (possibly from different model classes) and then train another model $f(x)$ to prodict y from the outputs of the g_j.

- The Bayesian approach can also be seen as form of ensembling

$$P(y \mid x) = \int_{\theta} P(y \mid x, \theta) P(\theta \mid \mathcal{D}) d\theta$$

where we average models $P(y \mid x, \theta)$ using weights $P(\theta \mid \mathcal{D})$.

15 Pros and Cons of Ensembling

Ensembling is a useful tecnique in machine learning. * It often helps squeeze out additional performance out of ML algorithms. * Many algorithms (like Adaboost) are forms of ensembling.

Disadvantages include: * It can be computationally expensive to train and use ensembles.

Part 2: Additive Models

Next, we are going to see another perspective on boosting and derive new boosting algorithms.

16 The Components of A Supervised Machine Learning Algorithm

We can define the high-level structure of a supervised learning algorithm as consisting of three components: * A **model class**: the set of possible models we consider. * An **objective** function, which defines how good a model is. * An **optimizer**, which finds the best predictive model in the model class according to the objective function

17 Review: Underfitting

Underfitting is another common problem in machine learning. * The model is too simple to fit the data well (e.g., approximating a high degree polynomial with linear regression). * As a result, the model is not accurate on training data and is not accurate on new data.

18 Review: Boosting

The idea of *boosting* is to reduce *underfitting* by combining models that correct each others' errors.

- As in bagging, we combine many models g_i into one *ensemble* f.
- Unlike bagging, the g_i are small and tend to underfit.
- Each g_i fits the points where the previous models made errors.

19 Additive Models

Boosting can be seen as a way of fitting an *additive model*:

$$f(x) = \sum_{t=1}^{T} \alpha_t g(x; \phi_t).$$

- The main model $f(x)$ consists of T smaller models g with weights α_t and paramaters ϕ_t.
- The parameters are the α_t plus the parameters ϕ_t of each g.

This is more general than a linear model, because g can be non-linear in ϕ_t (therefore so is f).

20 Example: Boosting Algorithms

Boosting is one way of training additive models.

1. Fit a weak learner g_0 on dataset $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}$. Let $f = g_0$.
2. Compute weights $w^{(i)}$ for each i based on model predictions $f(x^{(i)})$ and targets $y^{(i)}$. Give more weight to points with errors.
3. Fit another weak learner g_1 on $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}$ with weights $w^{(i)}$.
4. Set $f_1 = g_0 + \alpha_1 g$ for some weight α_1. Go to Step 2 and repeat.

21 Forward Stagewise Additive Modeling

A general way to fit additive models is the forward stagewise approach.

- Suppose we have a loss $L : \mathcal{Y} \times \mathcal{Y} \to [0, \infty)$.
- Start with $f_0 = \arg\min_\phi \sum_{i=1}^n L(y^{(i)}, g(x^{(i)}; \phi))$.
- At each iteration t we fit the best addition to the current model.

$$\alpha_t, \phi_t = \arg\min_{\alpha,\phi} \sum_{i=1}^{n} L(y^{(i)}, f_{t-1}(x^{(i)}) + \alpha g(x^{(i)}; \phi))$$

22 Practical Considerations

- Popular choices of g include cubic splines, decision trees and kernelized models.
- We may use a fix number of iterations T or early stopping when the error on a hold-out set no longer improves.
- An important design choice is the loss L.

23 Exponential Loss

Give a binary classification problem with labels $\mathcal{Y} = \{-1, +1\}$, the exponential loss is defined as

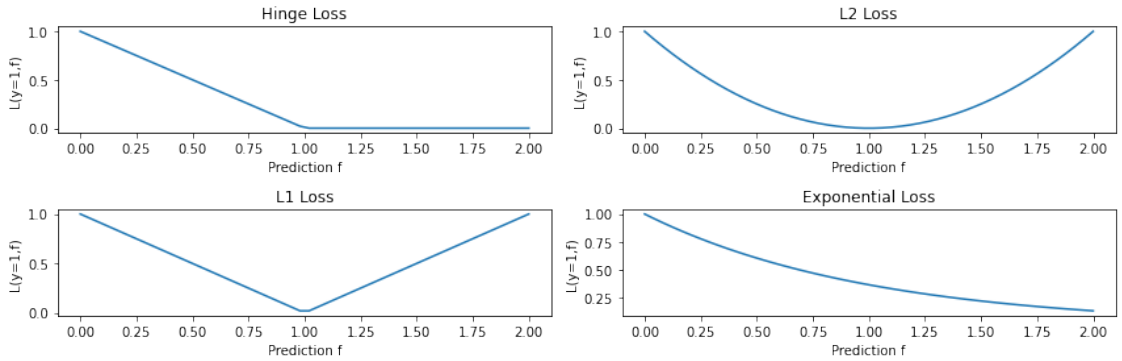
$$L(y, f) = \exp(-y \cdot f).$$

- When $y = 1$, L is small when $f \to \infty$.
- When $y = -1$, L is small when $f \to -\infty$.

Let's visualize the exponential loss and compare it to other losses.

[23]:
```
from matplotlib import pyplot as plt
import numpy as np
plt.rcParams['figure.figsize'] = [12, 4]

# define the losses for a target of y=1
losses = {
    'Hinge' : lambda f: np.maximum(1 - f, 0),
    'L2': lambda f: (1-f)**2,
```

```
    'L1': lambda f: np.abs(f-1),
    'Exponential': lambda f: np.exp(-f)
}

# plot them
f = np.linspace(0, 2)
fig, axes = plt.subplots(2,2)
for ax, (name, loss) in zip(axes.flatten(), losses.items()):
    ax.plot(f, loss(f))
    ax.set_title('%s Loss' % name)
    ax.set_xlabel('Prediction f')
    ax.set_ylabel('L(y=1,f)')
plt.tight_layout()
```



24 Special Case: Adaboost

Adaboost is an instance of forward stagewise additive modeling with the expoential loss.

At each step t we minimize

$$L_t = \sum_{i=1}^{n} e^{-y^{(i)}(f_{t-1}(x^{(i)})+\alpha g(x^{(i)};\phi))} = \sum_{i=1}^{n} w^{(i)} \exp\left(-y^{(i)}\alpha g(x^{(i)};\phi)\right)$$

with $w^{(i)} = \exp(-y^{(i)} f_{t-1}(x^{(i)}))$.

We can derive the Adaboost update rules from this equation.

Suppose that $g(y;\phi) \in \{-1, 1\}$. With a bit of algebraic manipulations, we get that:

$$\begin{aligned} L_t &= e^{\alpha} \sum_{y^{(i)} \neq g(x^{(i)})} w^{(i)} + e^{-\alpha} \sum_{y^{(i)} = g(x^{(i)})} w^{(i)} \\ &= (e^{\alpha} - e^{-\alpha}) \sum_{i=1}^{n} w^{(i)} \mathbb{I}\{y^{(i)} \neq g(x^{(i)})\} + e^{-\alpha} \sum_{i=1}^{n} w^{(i)}. \end{aligned}$$

where $\mathbb{I}\{\cdot\}$ is the indicator function.

From there, we get that:

$$\phi_t = \arg\min_{\phi} \sum_{i=1}^{n} w^{(i)} \mathbb{I}\{y^{(i)} \neq g(x^{(i)}; \phi)\}$$
$$\alpha_t = \log[(1 - e_t)/e_t]$$

where $e_t = \frac{\sum_{i=1}^{n} w^{(i)} \mathbb{I}\{y^{(i)} \neq f(x^{(i)})\}}{\sum_{i=1}^{n} w^{(i)}\}}$.

These are update rules for Adaboost, and it's not hard to show that the update rule for $w^{(i)}$ is the same as well.

25 Squared Loss

Another popular choice of loss is the squared loss.

$$L(y, f) = (y - f)^2.$$

The resulting algorithm is often called L2Boost. At step t we minimize

$$\sum_{i=1}^{n} (r_t^{(i)} - g(x^{(i)}; \phi))^2,$$

where $r_t^{(i)} = y^{(i)} - f(x^{(i)})_{t-1}$ is the residual from the model at time $t - 1$.

26 Logistic Loss

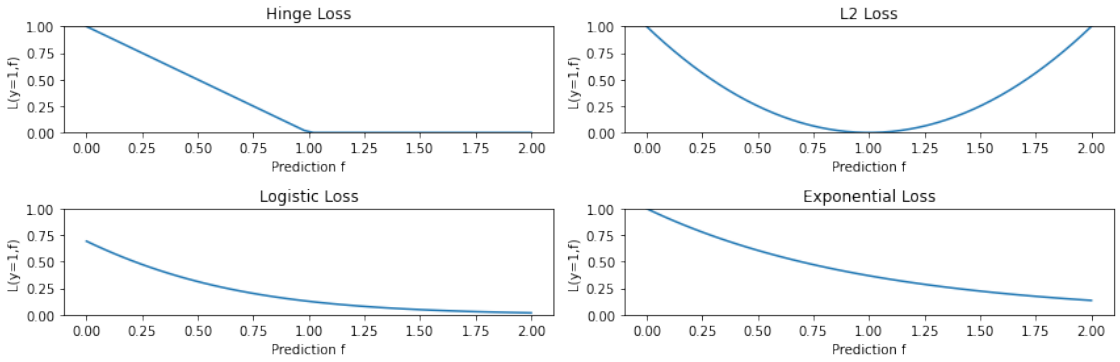
Another common loss is the log-loss. When $\mathcal{Y} = \{-1, 1\}$ it is defined as:

$$L(y, f) = \log(1 + \exp(-2 \cdot y \cdot f)).$$

This looks like the log of the exponential loss; it is less sensitive to outliers since it doesn't penalize large errors as much.

[25]:
```
from matplotlib import pyplot as plt
import numpy as np
plt.rcParams['figure.figsize'] = [12, 4]

# define the losses for a target of y=1
losses = {
    'Hinge' : lambda f: np.maximum(1 - f, 0),
    'L2': lambda f: (1-f)**2,
    'Logistic': lambda f: np.log(1+np.exp(-2*f)),
    'Exponential': lambda f: np.exp(-f)
}
```

```
# plot them
f = np.linspace(0, 2)
fig, axes = plt.subplots(2,2)
for ax, (name, loss) in zip(axes.flatten(), losses.items()):
    ax.plot(f, loss(f))
    ax.set_title('%s Loss' % name)
    ax.set_xlabel('Prediction f')
    ax.set_ylabel('L(y=1,f)')
    ax.set_ylim([0,1])
plt.tight_layout()
```



In the context of boosting, we minimize

$$J(\alpha, \phi) = \sum_{i=1}^{n} \log\left(1 + \exp\left(-2y^{(i)}(f_{t-1}(x^{(i)}) + \alpha g(x^{(i)}; \phi)\right)\right).$$

This give a different weight update compared to Adabost. This algorithm is called LogitBoost.

27 Pros and Cons of Boosting

The boosting algorithms we have seen so far improve over Adaboost. * They optimize a wide range of objectives. * Thus, they are more robust to outliers and extend beyond classification.

Cons: * Computational time is still an issue. * Optimizing greedily over each ϕ_t can take time. * Each loss requires specialized derivations.

28 Summary

- Additive models have the form
$$f(x) = \sum_{t=1}^{T} \alpha_t g(x; \phi_t).$$
- These models can be fit using the forward stagewise additive approach.
- This reproduces Adaboost and can be used to derive new boosting-type algorithms.

Part 3: Gradient Boosting

We are now going to see another way of deriving boosting algorithms that is inspired by gradient descent.

29 Review: Boosting

The idea of *boosting* is to reduce *underfitting* by combining models that correct each others' errors.

- As in bagging, we combine many models g_i into one *ensemble* f.
- Unlike bagging, the g_i are small and tend to underfit.
- Each g_i fits the points where the previous models made errors.

30 Review: Additive Models

Boosting can be seen as a way of fitting an *additive model*:

$$f(x) = \sum_{t=1}^{T} \alpha_t g(x; \phi_t).$$

- The main model $f(x)$ consists of T smaller models g with weights α_t and paramaters ϕ_t.
- The parameters are the α_t plus the parameters ϕ_t of each g.

This is not a linear model, because g can be non-linear in ϕ_t (therefore so is f).

31 Review: Forward Stagewise Additive Modeling

A general way to fit additive models is the forward stagewise approach.

- Suppose we have a loss $L : \mathcal{Y} \times \mathcal{Y} \to [0, \infty)$.
- Start with $f_0 = \arg\min_\phi \sum_{i=1}^{n} L(y^{(i)}, g(x^{(i)}; \phi))$.
- At each iteration t we fit the best addition to the current model.

$$\alpha_t, \phi_t = \arg\min_{\alpha,\phi} \sum_{i=1}^{n} L(y^{(i)}, f_{t-1}(x^{(i)}) + \alpha g(x^{(i)}; \phi))$$

32 Limitations of Forward Stagewise Additive Modeling

Forward stagewise additive modeling is not without limitations. * There may exist other losses for which it is complex to derive boosting-type weight update rules. * At each step, we may need to solve a costly optimization problem over ϕ_t. * Optimizing each ϕ_t greedily may cause us to overfit.

33 What Do Weak Learners Learn?

Consider, for example, L2Boost, which optimizes the L2 loss

$$L(y, f) = \frac{1}{2}(y - f)^2.$$

At step t we minimize

$$\sum_{i=1}^{n}(r_t^{(i)} - g(x^{(i)}; \phi))^2,$$

where $r_t^{(i)} = y^{(i)} - f_{t-1}(x^{(i)})$ is the residual from the model at time $t-1$.

Recall that the residual is

$$r_t^{(i)} = y^{(i)} - f_{t-1}(x^{(i)})$$

Observe that the residual is also the derivative of the $L2$ loss

$$\frac{1}{2}(y^{(i)} - f_{t-1}(x^{(i)}))^2$$

with respect to f at $f_{t-1}(x^{(i)})$:

$$r_t^{(i)} = \frac{\partial L(y^{(i)}, \mathrm{f})}{\partial \mathrm{f}}\bigg|_{\mathrm{f}=f_{t-1}(x)}$$

Thus, at step t we minimize

$$\sum_{i=1}^{n}\left(\underbrace{\left(y^{(i)} - f_{t-1}(x^{(i)})\right)}_{\text{derivative of } L \text{ at } f_{t-1}(x^{(i)})} - g(x^{(i)}; \phi)\right)^2.$$

Why does L2Boost fit the derivatives of the L2 loss?

34 Recall: Parametric Optimization

A machine learning model is a function

$$f_\theta : \mathcal{X} \to \mathcal{Y}$$

that maps inputs $x \in \mathcal{X}$ to targets $y \in \mathcal{Y}$.

The model has a d-dimensional set of parameters θ:

$$\theta = (\theta_1, \theta_2, ..., \theta_d).$$

Intuitively, f_θ should perform well in expectation on new data $\dot{x}, \dot{y}$ sampled from the data distribution $\mathbb{P}$:

$$J(\theta) = \mathbb{E}_{(\dot{x},\dot{y})\sim\mathbb{P}}\left[L\left(\dot{y}, f_\theta(\dot{x})\right)\right] \text{ is "good".}$$

Here, $L : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}$ is a performance metric and we take its expectation or average over all the possible samples $\dot{x}, \dot{y}$ from $\mathbb{P}$.

Recall that formally, an expectation $\mathbb{E}_{x)\sim P} f(x)$ is $\sum_{x \in \mathcal{X}} f(x)P(x)$ if x is discrete and $\int_{x \in \mathcal{X}} f(x)P(x)dx$ if x is continuous.

Intuitively,

$$\mathbb{E}_{(\dot{x},\dot{y})\sim\mathbb{P}} \left[L\left(\dot{y}, f_\theta(\dot{x})\right)\right] = \sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} L\left(y, f_\theta(x)\right) \mathbb{P}(x, y)$$

is the performance on an *infinite-sized* holdout set, where we have sampled every possible point.

In practice, we cannot measure

$$\mathbb{E}_{(\dot{x},\dot{y})\sim\mathbb{P}} \left[L\left(\dot{y}, f_\theta(\dot{x})\right)\right]$$

on infinite data.

We approximate its performance with a sample $\dot{\mathcal{D}}$ from $\mathbb{P}$ and we measure

$$\frac{1}{m} \sum_{i=1}^{m} L\left(\dot{y}^{(i)}, f_\theta(\dot{x}^{(i)})\right).$$

If the number of IID samples m is large, this approximation holds (we call this a Monte Carlo approximation).

35 Recall: The Gradient

The gradient $\nabla J(\theta)$ is the d-dimensional vector of partial derivatives:

$$\nabla J(\theta) = \begin{bmatrix} \frac{\partial J(\theta)}{\partial \theta_1} \\ \frac{\partial J(\theta)}{\partial \theta_2} \\ \vdots \\ \frac{\partial J(\theta)}{\partial \theta_d} \end{bmatrix}.$$

The j-th entry of the vector $\nabla J(\theta)$ is the partial derivative $\frac{\partial J(\theta)}{\partial \theta_j}$ of J with respect to the j-th component of θ.

36 Recall: Gradient Descent

We can optimize our objective using gradient descent via the usual update rule:

$$\theta_t \leftarrow \theta_{t-1} - \alpha_t \nabla J(\theta_{t-1}).$$

37 Functional Optimization

Instead of finite-dimensional paramters, we can try optimizing directly over infinite-dimensionals functions

$$f : \mathcal{X} \to \mathcal{Y}$$

from inputs $x \in \mathcal{X}$ to targets $y \in \mathcal{Y}$.

We want to optimize over the space of f directly.

- Each function is an infinite-dimensional *vector* indexed by $x \in \mathcal{X}$:

$$f = \begin{bmatrix} \vdots \\ f(x) \\ \vdots \end{bmatrix}.$$

  The x-th component of the vector f is $f(x)$.

- It's as if we choose infinite parameters $\theta = (..., f(x), ...)$ that specify function values, and we optimize over that.

Intuitively, a model f is successful if it performs well in expectation on new data $\dot{x}, \dot{y}$ sampled from the data distribution $\mathbb{P}$:

$$J(\theta) = \mathbb{E}_{(\dot{x},\dot{y})\sim\mathbb{P}}\left[L\left(\dot{y}, f(\dot{x})\right)\right] \text{ is "good".}$$

Here, $L : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}$ is a performance metric and we take its expectation or average over all the possible samples $\dot{x}, \dot{y}$ from $\mathbb{P}$.

38 Functional Gradients

Consider solving the optimization problem using gradient descent:

$$\min_f J(f) = \min_f \mathbb{E}_{(\dot{x},\dot{y})\sim\mathbb{P}}\left[L\left(\dot{y}, f(\dot{x})\right)\right].$$

We may define the functional gradient of this loss at f as a function $\nabla J(f) : \mathcal{X} \to \mathbb{R}$

$$\nabla J(f) = \begin{bmatrix} \vdots \\ \frac{\partial J(f)}{\partial f(x)} \\ \vdots \end{bmatrix}.$$

Let's make a few observations about the functional gradient.

- It's an object indexed by $x \in \mathcal{X}$.
- At each $x \in \mathcal{X}$, $\nabla J(f_0)(x)$ tells us how to modify $f_0(x)$ to make $L(y, f_0(x))$ smaller.
- This is consistent with the fact that we are optimizing over a "vector" f, also indexed by $x \in \mathcal{X}$.

This is best understood via a picture.

The functional gradient is a function that tells us how much we "move" $f(x)$ at each point x.

Given a good step size, the resulting new function will be closer to minimizing L.

The x-th entry of the vector $\nabla J(f)$ is the partial derivative $\frac{\partial J(f)}{\partial f(x)}$ of J with respect to $f(x)$, the x-th component of f.

$$\frac{\partial J(f)}{\partial f(x)} = \frac{\partial}{\partial f(x)}\left(\mathbb{E}_{(\dot{x},\dot{y})\sim\mathbb{P}}\left[L\left(\dot{y}, f(\dot{x})\right)\right]\right) \approx \left.\frac{\partial L(y,f)}{\partial f}\right|_{f=f(x)}$$

So the functional gradient is approximately

$$\nabla J(f) = \begin{bmatrix} \vdots \\ \left.\frac{\partial L(y,f)}{\partial f}\right|_{f=f(x)} \\ \vdots \end{bmatrix}.$$

This is an infinite-dimensional indexed by x.

39 Functional Gradient Descent

We can optimize our objective using gradient descent in functional space via the usual update rule:

$$f_t \leftarrow f_{t-1} - \alpha_t \nabla J(f_{t-1}).$$

After T steps, we will learn a model of the form

$$f_T = f_0 - \sum_{t=0}^{T-1} \alpha_t \nabla J(f_t)$$

* Recall that each $\nabla J(f_t)$ is a function of x * Therefore f_T is a function of x as well * Because it's the result of gradient descent, f_T will minimize J.

As defined, this is not a practical algorithm: * We cannot represent $\nabla J(f)$ because it's a general function * We cannot measure $\nabla J(f)$ at each x (only at n training points). * Even if we could, the problem would be too unconstrained

40 Modeling Functional Gradients

We will address this problem by learning a *model* of gradients.

- In supervised learning, we only have access to n data points that describe the true $\mathcal{X} \to \mathcal{Y}$ mapping (call it f^*).
- We learn a *model* $f_\theta : \mathcal{X} \to \mathcal{Y}$ within $\mathcal{M}$ to approximate f^*.
- The model extrapolates beyond the training set. Given enough datapoints, f_θ learns a true mapping.

We will apply the same idea to gradients. * We assume a model $g_\theta : \mathcal{X} \to R$ of the functional gradient $\nabla J(f)$ within a class $\mathcal{M}$.

$$g \in \mathcal{M} \qquad\qquad g \approx \nabla J(f)$$

* The model extrapolates beyond the training set. Given enough datapoints, g_θ learns $\nabla J(f)$.

Functional descent then has the form:

$$\underbrace{f(x)}_{\text{new function}} \leftarrow \underbrace{f(x) - \alpha g(x)}_{\text{old function - gradient step}}.$$

If g generalizes, this approximates $f \leftarrow f - \alpha \nabla J(f)$.

41 Fitting Functional Gradients

What does it mean to approximate a functional gradient $g \approx \nabla J(f)$ in practice? We can use standard supervised learning.

Suppose we have a fixed function f and we want to estimate the functional gradient of L

$$\left.\frac{\partial L(\mathrm{y},\mathrm{f})}{\partial \mathrm{f}}\right|_{\mathrm{f}=f(x)}.$$

at any $x \in \mathcal{X}$

1. We define a loss L_g (e.g., L2 loss) measure how well $g \approx \nabla J(f)$.
2. We compute $\nabla J(f)$ on the training dataset:

$$\mathcal{D}_g = \left\{ \left(x^{(i)}, \underbrace{\left.\frac{\partial L(y^{(i)},\mathrm{f})}{\partial \mathrm{f}}\right|_{\mathrm{f}=f(x^{(i)})}}_{\text{functional derivative } \nabla_{\mathbf{f}} J(\mathbf{f})_i \text{ at } f(x^{(i)})} \right), i = 1, 2, \ldots, n \right\}$$

3. We train a model $g : \mathcal{X} \to \mathbb{R}$ on $\mathcal{D}_g$ to predict functional gradients at any x:

$$g(x) \approx \left.\frac{\partial L(\mathrm{y},\mathrm{f})}{\partial \mathrm{f}}\right|_{\mathrm{f}=f_0(x)}.$$

42 Gradient Boosting

Gradient boosting is a procedure that performs functional gradient descent with approximate gradients.

Start with $f(x) = 0$. Then, at each step $t > 1$:

1. Create a training dataset $\mathcal{D}_g$ and fit $g_t(x^{(i)})$ using loss L_g:

$$g_t(x) \approx \left.\frac{\partial L(\mathrm{y},\mathrm{f})}{\partial \mathrm{f}}\right|_{\mathrm{f}=f_0(x)}.$$

2. Take a step of grad descent using approximate grads with step α_t:

$$f_t(x) = f_{t-1}(x) - \alpha_t \cdot g_t(x).$$

43 Interpreting Gradient Boosting

Notice how after T steps we get an additive model of the form

$$f(x) = \sum_{t=1}^{T} \alpha_t g_t(x).$$

This looks like the output of a boosting algorithm!

- This works for any differentiable loss L.
- It does not require any mathematical derivations for new L.

44 Boosting vs. Gradient Boosting

Consider, for example, L2Boost, which optimizes the L2 loss

$$L(y, f) = \frac{1}{2}(y - f)^2.$$

At step t we minimize

$$\sum_{i=1}^{n}(r_t^{(i)} - g(x^{(i)}; \phi))^2,$$

where $r_t^{(i)} = y^{(i)} - f_{t-1}(x^{(i)})$ is the residual from the model at time $t-1$.

Observe that the residual

$$r_t^{(i)} = y^{(i)} - f(x^{(i)})_{t-1}$$

is also the gradient of the $L2$ loss with respect to f as $f(x^{(i)})$

$$r_t^{(i)} = \left.\frac{\partial L(y^{(i)}, \mathrm{f})}{\partial \mathrm{f}}\right|_{\mathrm{f}=f_0(x)}$$

Many boosting methods are special cases of gradient boosting in this way.

45 Losses for Additive Models

We have seen several losses that can be used with the forward stagewise additive approach. * The exponential loss $L(y, f) = \exp(-yf)$ gives us Adaboost. * The log-loss $L(y, f) = \log(1+\exp(-2yf))$ is more robust to outliers. * The squared loss $L(y, f) = (y - f)^2$ can be used for regression.

46 Losses for Gradient Boosting

Gradient boosting can optimize a wide range of losses.

1. Regression losses:
   - L2, L1, and Huber (L1/L2 interpolation) losses.
   - Quantile loss: estimates quantiles of distribution of $p(y|x)$.
2. Classification losses:
   - Log-loss, softmax loss, exponential loss, negative binomial likelihood, etc.

47 Practical Considerations

When using gradient boosting these additional facts are useful: * We most often use small decision trees as the learner g_t. Thus, input pre-processing is minimal. * We can regularize by controlling tree size, step size α, and using *early stopping*. * We can scale-up gradient boosting to big data by subsampling data at each iteration (a form of *stochastic* gradient descent).

48 Algorithm: Gradient Boosting

- **Type**: Supervised learning (classification and regression).
- **Model family**: Ensembles of weak learners (often decision trees).
- **Objective function**: Any differentiable loss function.
- **Optimizer**: Gradient descent in functional space. Weak learner uses its own optimizer.
- **Probabilistic interpretation**: None in general; certain losses may have one.

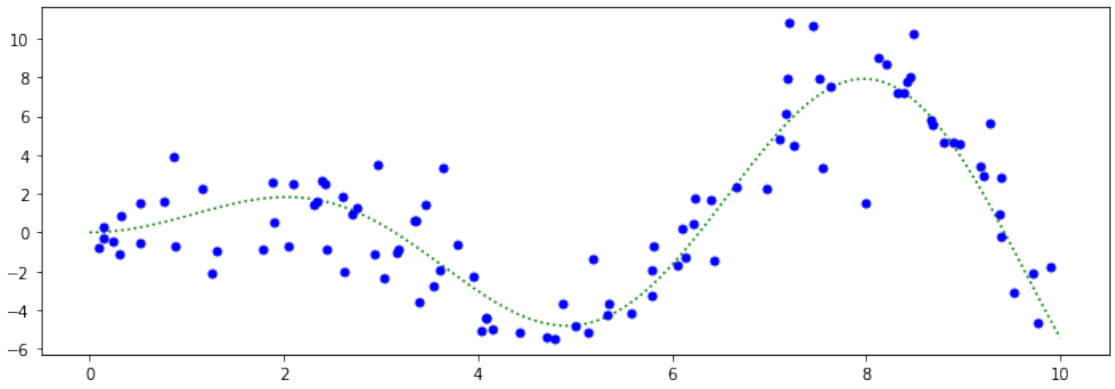
49 Gradient Boosting: An Example

Let's now try running Gradient Boosted Decision Trees on a small regression dataset.

First we create the dataset.

[21]:
```
# https://scikit-learn.org/stable/auto_examples/ensemble/
 ↪plot_gradient_boosting_quantile.html
X = np.atleast_2d(np.random.uniform(0, 10.0, size=100)).T
X = X.astype(np.float32)

# Create dataset
f = lambda x: x * np.sin(x)
y = f(X).ravel()
dy = 1.5 + 1.0 * np.random.random(y.shape)
noise = np.random.normal(0, dy)
y += noise

# Visualize it
xx = np.atleast_2d(np.linspace(0, 10, 1000)).T
plt.plot(xx, f(xx), 'g:', label=r'$f(x) = x\,\sin(x)$')
plt.plot(X, y, 'b.', markersize=10, label=u'Observations')
```

[21]:
```
[<matplotlib.lines.Line2D at 0x12ed61898>]
```



Next, we train a GBDT regressor.

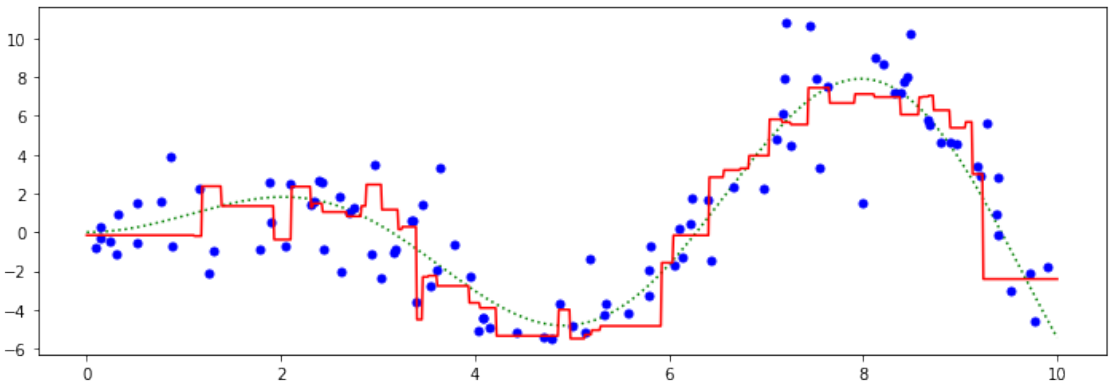
[19]:
```
from sklearn.ensemble import GradientBoostingRegressor

alpha = 0.95
clf = GradientBoostingRegressor(loss='ls', alpha=alpha,
                                n_estimators=250, max_depth=3,
                                learning_rate=.1, min_samples_leaf=9,
                                min_samples_split=9)
clf.fit(X, y)
```

[19]:
```
GradientBoostingRegressor(alpha=0.95, min_samples_leaf=9, min_samples_split=9,
                          n_estimators=250)
```

We may now visualize its predictions

[22]:
```
y_pred = clf.predict(xx)
plt.plot(xx, f(xx), 'g:', label=r'$f(x) = x\,\sin(x)$')
plt.plot(X, y, 'b.', markersize=10, label=u'Observations')
plt.plot(xx, y_pred, 'r-', label=u'Prediction')
```

[22]:
```
[<matplotlib.lines.Line2D at 0x12c98e438>]
```



50 Pros and Cons of Gradient Boosting

Gradient boosted decision trees (GBTs) are one of the best off-the-shelf ML algorithms that exist, often on par with deep learning. * Attain state-of-the-art performance. GBTs have won the most Kaggle competitions. * Require little data pre-processing and tuning. * Work with any objective, including probabilistic ones.

Their main limitations are: * GBTs don't work with unstructured data like images, audio. * Implementations not as flexible as modern neural net libraries.