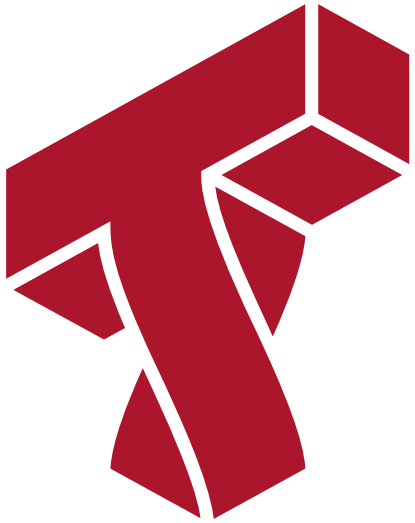


Lecture 21: Boosting

Part 1: Boosting and Ensembling

We will look at ways in which multiple machine learning can be combined.

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

Review: Overfitting

Overfitting is one of the most common failure modes of machine learning.

- An expressive model (e.g., polynomial) fits training data perfectly.
- But it makes incorrect test set predictions, and doesn't generalize.

Review: Bagging

Bagging reduces *overfitting* by averaging many expressive models .

```
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(x_test)
```

Review: Underfitting

Underfitting is another common problem in machine learning.

- The model is too simple to fit the data well.
- Training performance is low, hence test performance is low.

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.

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.

Structure of a Boosting Algorithm

Boosting reduces *underfitting* via models that correct each others' errors.

1. Compute weights $w^{(i)}$ for each i based on t-th model predictions $f_t(x^{(i)})$ and targets $y^{(i)}$. Give more weight to points with errors.

1. Fit new weak learner g_t on $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}$ with weights $w^{(i)}$.

1. Set $f_{t+1} = f_t + \alpha_t g_t$ for some weight α_t. Go to Step 1 and repeat.

Origins of Boosting

Boosting was initially developed in the 90s within theoretical ML.

- Boosting originally addressed a theoretical question: can weak learners (>50% accuracy) can be combined into a strong learner?
- This research led to a practical algorithm called *AdaBoost*.

There are now many types of boosting algorithms.

AdaBoost: An Example

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

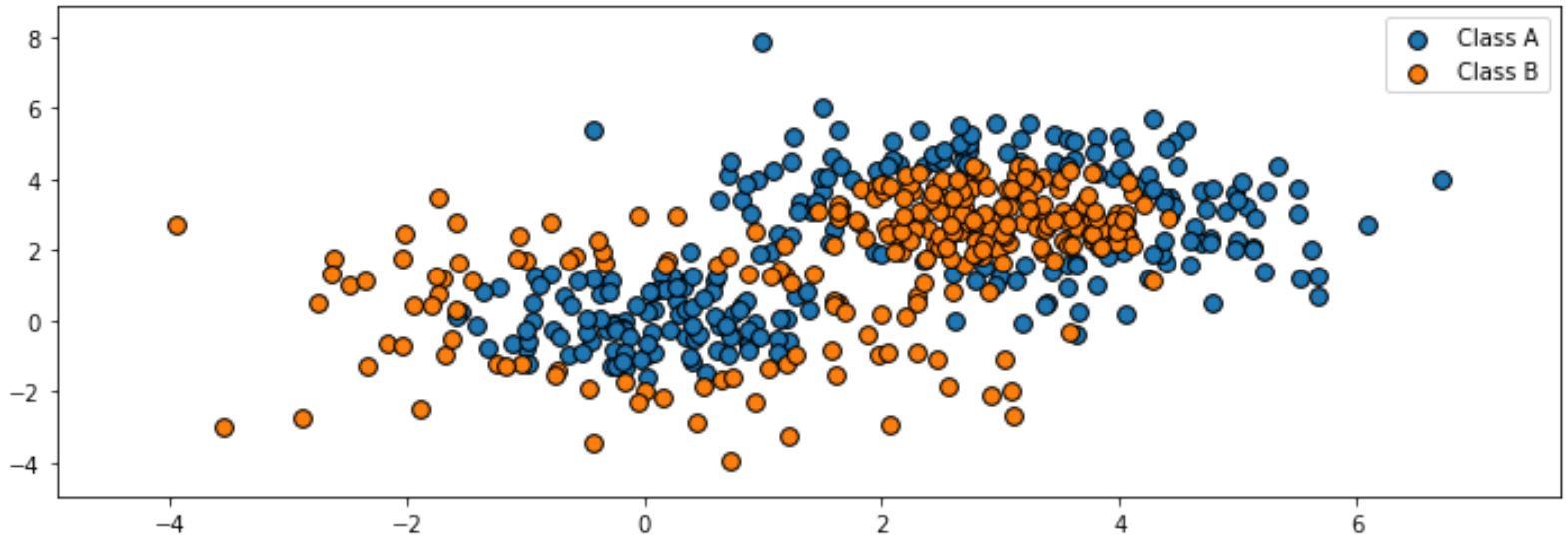
Let's start by creating a classification dataset.

```
# 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`.

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

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)
```

```
<matplotlib.legend.Legend at 0x12afda198>
```



Let's now train AdaBoost on this dataset.

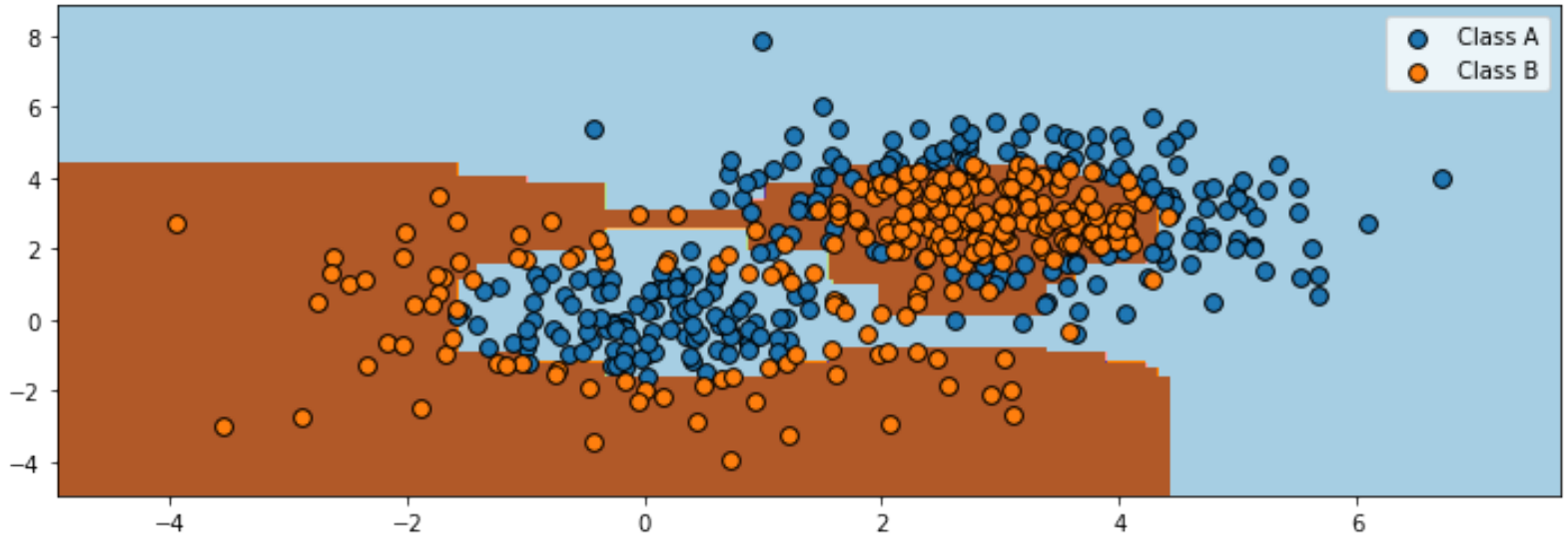
```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier

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

```
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.

```
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)
```

```
<matplotlib.legend.Legend at 0x12b3b8438>
```



Ensembling

Boosting and bagging are special cases of *ensembling*.

The idea of ensembling is to combine many models into one.

- Bagging is a form of ensembling to reduce overfitting
- Boosting is a form of ensembling to reduce underfitting

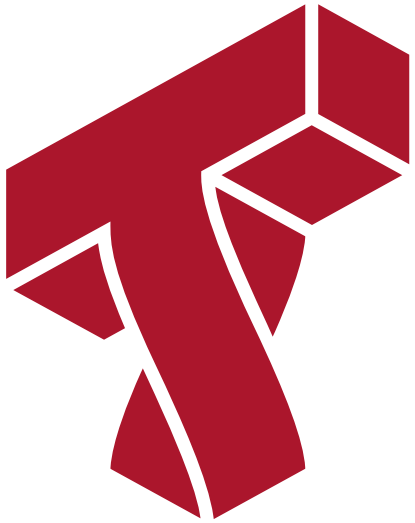
Pros and Cons of Boosting

Boosting algorithms generalize AdaBoost and offer many advantages:

- High accuracy via a highly expressive non-linear model family.
- Low pre-processing requirements if trees are used as weak learners.

Disadvantages include:

- Large ensembles can be expensive to train.
- The interpretability of the weak learners is lost.



Part 2: Additive Models

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

Additive Models

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

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

- $f(x)$ consists of T smaller models g with weights α_t & params ϕ_t.
- The parameters are the α_t plus the parameters ϕ_t of each g.

Note that g can be non-linear in ϕ_t (therefore so is f).

Example: Boosting Algorithms

Boosting is one way of training additive models.

1. Compute weights $w^{(i)}$ for each i based on t-th model predictions $f_t(x^{(i)})$ and targets $y^{(i)}$. Give more weight to points with errors.

1. Fit new weak learner g_t on $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}$ with weights $w^{(i)}$.

1. Set $f_{t+1} = f_t + \alpha_t g_t$ for some weight α_t. Go to Step 1 and repeat.

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 t, freeze f_{t-1} and fit best new addition $\alpha_t g(x, \phi_t)$ to f_{t-1}:

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

Example: Squared Loss

A 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

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

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

Facts About Additive Modeling

Boosting and additive models are closely connected.

- Forward stagewise training (FST) is a form of boosting with g_t
- Classical boosting algorithms (AdaBoost) are special cases of FST.
- Additive modeling is more general, supports many different losses.

Additive modeling is a principled way of doing boosting for any loss L.

Practical Considerations

- Popular choices of g include decision trees or cubic splines.
- We may use a fixed 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.

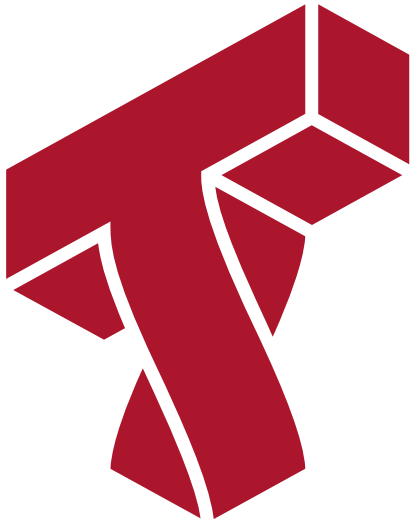
Pros and Cons of Additive Models

The 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.



Part 3: Gradient Boosting

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

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.

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

What Do Weak Learners Learn?

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

At step t we minimize

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

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

The residual error of model f_{t-1} is given by:

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

Note that $r_t^{(i)}$ is the derivative of the L_2 loss with respect to $f_{t-1}(x^{(i)})$:

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

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?

Recall: Supervised Learning

Recall that in regular supervised learning, we minimize an objective $J(\theta)$

$$J(\theta) = \sum_{i=1}^{n} L\left(y^{(i)}, f_\theta(x^{(i)})\right)$$

over a dataset, where L is a loss and f_θ is a model with parameters θ.

Functional Optimization

Instead of optimizing $J(\theta)$, let's directly optimize $J(f)$ over *functions* f!

This requires a few simplifying assumptions (for now):

- The set $\mathcal{X} = \{x_1, \ldots, x_m\}$ of possible x is finite and has size m.
- Thus, each $f(x)$ is a finite dimensional vector of size m.

In other words, we can view $f : \mathcal{X} \to \mathcal{Y}$ over a finite $\mathcal{X}$ as a vector in $\mathbb{R}^m$:

$$f = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_m \end{bmatrix} = \begin{bmatrix} f(x_1) \\ f(x_2) \\ \vdots \\ f(x_m) \end{bmatrix}.$$

f is an m-dimensional vector; its j-th component is the prediction $f(x_j)$.

Supervised Learning Over Functions

Now, supervised learning becomes a functional optimization problem over

$$J(f) = \sum_{i=1}^{n} L\left(y^{(i)}, f(x^{(i)})\right)$$

where $f \in \mathbb{R}^m$ is a vector L is a loss over a dataset $\{x^{(i)}, y^{(i)}\}_{i=1}^{n}$.

We can think of this as setting the parameters θ directly to values of f.

Functional Gradients

How do we optimize $J(f)$ over f?

f is a finite-dimensional vector; thus we use the *functional gradient* of J:

$$\nabla J(f) = \begin{bmatrix} \frac{dJ}{df_1} \\ \frac{dJ}{df_2} \\ \vdots \\ \frac{dJ}{df_m} \end{bmatrix} = \begin{bmatrix} \frac{dJ}{df(x_1)} \\ \frac{dJ}{df(x_2)} \\ \vdots \\ \frac{dJ}{df(x_m)} \end{bmatrix}$$

This is very similar to taking the gradient with respect to parameters θ.

Let's further compare the parametric and the functional gradients.

- $\nabla J(\theta_0)$ tells how to modify θ_0 to decrease J at $J(\theta_0)$.
- $\nabla J(f_0)$ tells how to modify f_0 to decrease J at $J(f_0)$.

Functional Gradient Descent

How do we optimize the functional supervised learning objective?

$$J(f) = \sum_{i=1}^{n} L\left(y^{(i)}, f(x^{(i)})\right)$$

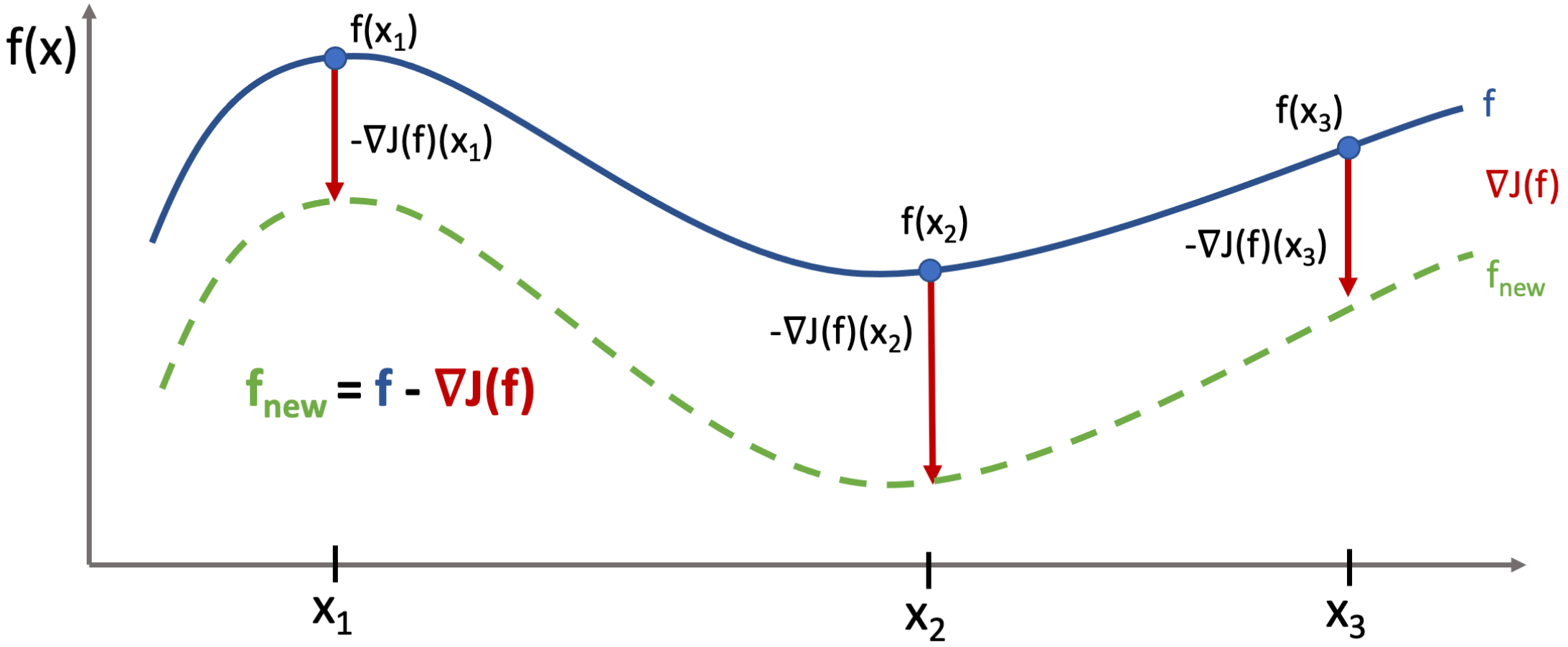
We perform gradient descent over f:

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

This is analogous to regular gradient descent:

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

This is best understood via a picture.



After T steps of $f_t \leftarrow f_{t-1} - \alpha_t \nabla J(f_{t-1})$, we get a model of the form

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

- Each $\nabla J(f_t)$ is a function of x, thus so is f_T
- This also looks like an additive model!

Problems With Functional Supervised Learning

However, in its current form, this approach is not practical.

- Typically $m = |\mathcal{X}|$ is very large (even infinite). Thus, we cannot estimate $\nabla J(f)$ everywhere.
- Even if we could, the model f would be too expressive and overfit.

Approximating Functional Gradients

Our approach will be to estimate $\nabla J(f_t)$ from data using a model g_t.

We will then perform approximate functional gradient descent using

$$f_t \leftarrow f_{t-1} - \alpha_t g_t$$

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

Estimating Functional Gradients From Data

How do we approximate $\nabla J(f)$ using a parametric model g_ϕ with parameters ϕ?

- Recall: $\nabla J(f)$ is a vector. We can evaluate $\nabla J(f)(x_j)$ at any x_j.
- We also have a dataset $\mathcal{D} = \{x^{(i)}, y^{(i)} \mid i = 1, \ldots, n\}$.

Let's use supervised learning to build a model $g_t(x) \approx \nabla J(f)(x)$ on $\mathcal{D}$.

First, we need a formula for $\nabla J(f)(x^{(i)})$. Assume all $x^{(i)}$ are unique.

- Recall that we defined $J(f) = \sum_{i=1}^{n} L\left(y^{(i)}, f(x^{(i)})\right)$.
- Then we have that

$$\nabla J(f)(x^{(i)}) = \frac{d}{df(x^{(i)})} J(f) = \frac{d}{df(x^{(i)})} L(y^{(i)}, f(x^{(i)})).$$

- For example, if L is the L_2 loss, then $\nabla J(f)(x^{(i)}) = y^{(i)} - f(x^{(i)})$

Fitting Functional Gradients

In order to fit functional gradients, we apply supervised learning.

1. We compute $\nabla J(f)$ on the training dataset:

$$\mathcal{D}_g = \left\{ \left(x^{(i)}, \nabla J(f)(x^{(i)}) \right), i = 1, 2, \ldots, n \right\}$$

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

$$g(x) \approx \nabla J(f)(x).$$

Gradient Boosting

Gradient boosting is functional gradient descent with estimated gradients.

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

1. Create a training dataset $\mathcal{D}_g$ and fit g_t on $\mathcal{D}_g$ such that

$$g_t(x) \approx \nabla J(f)(x).$$

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

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

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.

Boosting vs. Gradient Boosting

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

At step t we minimize

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

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

$$r_t^{(i)} = \nabla J(f)(x^{(i)})$$

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

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.

Practical Considerations

When using gradient boosting these additional facts are useful:

- We often use decision trees as g_t for minimal input processing.
- We can regularize via size of g_t, 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).

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.

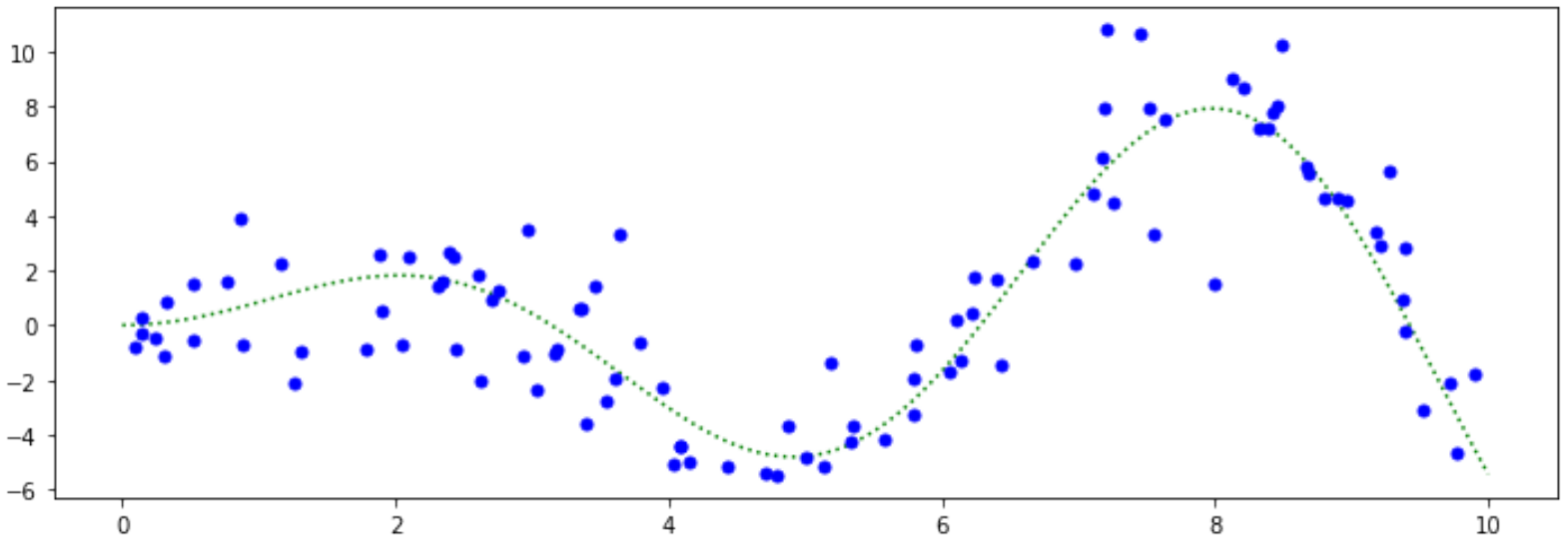
Gradient Boosting: An Example

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

First we create the dataset.

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

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')
```

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



Next, we train a GBDT regressor.

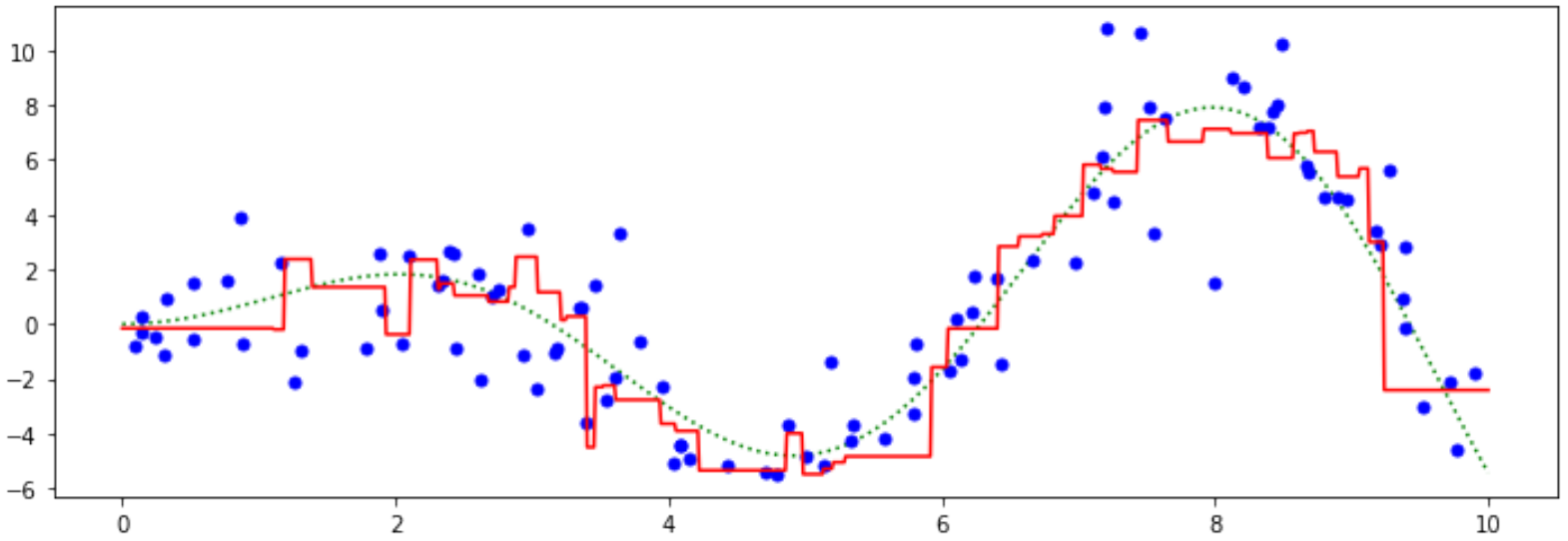
```
from sklearn.ensemble import GradientBoostingRegressor

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)
```

```
GradientBoostingRegressor(alpha=0.95, min_samples_leaf=9, min_samples_split=9,
                          n_estimators=250)
```

We may now visualize its predictions

```
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')
```

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



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 rule on Kaggle.
- 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.