Recall: Boosting

AdaBoost Algorithm

An *iterative* algorithm for "ensembling" base learners

- Input: $\{(\mathbf{x}_i, y_i)\}_{i=1}^n, T, \mathscr{F}$, base learner
- Initialize: $\mathbf{w}^1 = (\frac{1}{n}, \ldots, \frac{1}{n})$
- For $t = 1, \ldots, T$
  - $\mathbf{w}^t \to \boxed{\text{base learner finds} \quad \arg\min_{f \in \mathscr{F}} \sum_{i=1}^n w_i^t 1_{\{f(\mathbf{x}_i) \neq y_i\}}} \to f_t$
  - $\alpha_t = \frac{1}{2} \ln \left(\frac{1 - r_t}{r_t} \right)$
    - where $r_t := e_{\mathbf{w}^t}(f_t) = \frac{1}{n} \sum_{i=1}^n w_i \mathbf{1}_{\{f(\mathbf{x}_i) \neq y_i\}}$
  - $w_i^{t+1} = \frac{w_i^t \exp(-\alpha_t y_i f_t(\mathbf{x}_i))}{z_t}$ where z_t normalizes.
- Output: $h_T(\mathbf{x}) = \text{sign} \left(\sum_{t=1}^T \alpha_t f_t(\mathbf{x}) \right)$

Adaboost through Coordinate Descent

It is often said that we can view Adaboost as "Coordinate Descent" on the exponential loss function.

Question: Can you figure out what that means? Why is Adaboost doing coordinate descent?

Hint 1: You need to figure out the objective function being minimized. For simplicity, assume there are a finite number of weak learners in $\mathscr{F}$

Hint 2: Recall that the exponential loss function is $\ell(h; (x, y)) = \exp(-yh(x))$

Hint 3: Let's write down the objective function being minimized. For simplicity, assume there are a finite number of weak learners in $\mathscr{F}$, say indexed by $j = 1, \ldots, m$. Given a weight vector $\vec{\alpha}$, exponential loss over the data for this $\vec{\alpha}$ is:

$$\text{Loss}(\vec{\alpha}) = \sum_{i=1}^n \exp\left(-y_i \left(\sum_{j=1}^m \alpha_j h_j(\vec{x}_i) \right) \right)$$

Coordinate descent chooses the most negative coordiante of $\nabla L(\vec{\alpha})$ and updates *only this coordinate*. Which coordinate is chosen?

Bagging classifiers

Let's explore how bagging (bootstrapped aggregation) works with classifiers to reduce variance, first by evaluating off the shelf tools and then by implementing our own basic bagging classifier.

In both examples we'll be working with the dataset from the forest cover type prediction Kaggle competition (https://www.kaggle.com/c/forest-cover-type-prediction), where the aim is to build a multi-class classifier to predict the forest cover type of a 30x30 meter plot of land based on cartographic features. See their notes about the dataset (https://www.kaggle.com/c/forest-cover-type-prediction/data) for more background.

Exploring bagging

Loading and splitting the dataset

First, let's load the dataset:

In [1]:
```
import pandas as pd
df = pd.read_csv('forest-cover-type.csv')
df.head()
```

Out[1]:

	Id	Elevation	Aspect	Slope	Horizontal_Distance_To_Hydrology	Vertical_Distance_To_
0	1	2596	51	3	258	0
1	2	2590	56	2	212	-6
2	3	2804	139	9	268	65
3	4	2785	155	18	242	118
4	5	2595	45	2	153	-1

5 rows × 56 columns

Now we extract the X/y features and split them into a 60/40 train / test split so that we can see how well the training set performance generalizes to a heldout set.

In [2]:
```
X, y = df.iloc[:, 1:-1].values, df.iloc[:, -1].values
```

In [3]:
```
from sklearn.cross_validation import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.6, random_state=0)
```

Evaluating train/test with and without bagging

Now let's use an off the shelf decision tree classifier and compare its train/test performance with a bagged (http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.BaggingClassifier.html) decision tree.

In [4]:
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import BaggingClassifier
from sklearn.metrics import accuracy_score

models = [
    ('tree', DecisionTreeClassifier(random_state=0)),
    ('bagged tree', BaggingClassifier(
            DecisionTreeClassifier(random_state=0),
            random_state=0,
            n_estimators=10))
]

for label, model in models:
    model.fit(X_train, y_train)
    print("{} training|test accuracy: {:.2f} | {:.2f}".format(
            label,
            accuracy_score(y_train, model.predict(X_train)),
            accuracy_score(y_test, model.predict(X_test))))
```

```
tree training|test accuracy: 1.00 | 0.73
bagged tree training|test accuracy: 0.99 | 0.80
```

Note that both models were able to (nearly) fit the training set perfectly, and that bagging substantially improves test set performance (reduces variance).

Hyperparameters

Let's look at two hyperparametes associated with the bagging classifier:

- **num_estimators** controls how many classifiers make up the ensemble
- **max_samples** controls how many samples each classifier in the ensemble draws

How many classifiers do we need to reduce variance?

The default number of estimators is 10; explore the performance of the bagging classifier with a range values. How many classifiers do we need to reduce variance? What is the point of diminishing returns for this dataset?

In [5]:
```
# your code goes here!
```

How much of the dataset does each classifier need?

By default, max_samples is set to 1.0, which means each classifier gets a number of samples equal to the size of the training set.

How do you suppose bagging manages to reduce variance while still using the same number of samples?

Explore how the performance varies as you range `max_samples` (note, you can use float values between 0.0 and 1.0 to choose a percentage):

In [6]:
```
# your code goes here!
```

Implementing Bagging

We've shown the power of bagging, now let's appreciate its simplicity by implementing our own bagging classifier right here!

In [7]:
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.base import BaseEstimator
import numpy as np

class McBaggingClassifier(BaseEstimator):

    def __init__(self, classifier_factory=DecisionTreeClassifier, num_cl
assifiers=10):
        self.classifier_factory = classifier_factory
        self.num_classifiers = num_classifiers

    def fit(self, X, y):
        # create num_classifier classifiers calling classifier_factory,
 each
        # fitted with a different sample from X
        return self

    def predict(self, X):
        # get the prediction for each classifier, take a majority vote

        return np.ones(X.shape[0])
```

You should be able to achieve similar performance to scikit-learn's implementation:

In [8]:
```
our_models = [
    ('tree', DecisionTreeClassifier(random_state=0)),
    ('our bagged tree', McBaggingClassifier(
            classifier_factory=lambda: DecisionTreeClassifier(random_sta
te=0)
            ))
]

for label, model in our_models:
    model.fit(X_train, y_train)
    print("{} training|test accuracy: {:.2f} | {:.2f}".format(
        label,
        accuracy_score(y_train, model.predict(X_train)),
        accuracy_score(y_test, model.predict(X_test))))
```

```
tree training|test accuracy: 1.00 | 0.73
our bagged tree training|test accuracy: 0.14 | 0.14
```