CAP4770 – Intro to Data Science

Ensemble Learning and Random Forests

Prof. Ye Xia

Overview

- By aggregating the predictions of a group of predictors (i.e., classifiers or regressors), you will often get better predictions than with the best individual predictor.
- A group of predictors is called an **ensemble**; thus, this technique is called **Ensemble Learning**.
- Example – **Random Forest**: You can train a group of decision tree classifiers, each on a different random subset of the training set. To make predictions, you obtain the predictions of all the individual trees, then predict the class that gets the most votes.
- Near the end of a project and after having built a few good predictors, people often use ensemble methods to combine them into an even better predictor.

Voting Classifiers

- Suppose you have trained a few classifiers, such as a Logistic Regression classifier, an SVM classifier, a Random Forest classifier, a K-Nearest Neighbors classifier, etc.
- A very simple way to create an even better classifier is to have each classifier make a prediction. The class that gets the most votes is declared to be the final prediction; this is known as **hard voting**.
- Somewhat surprisingly, this voting classifier often achieves a higher accuracy than the best classifier in the ensemble.
- In fact, even if each classifier is a weak learner (meaning it does only slightly better than random guessing), the ensemble can still be a strong learner (achieving high accuracy), provided there is a sufficient number of weak learners and they are sufficiently diverse.

- To see why, consider M binary classifiers with the positive class being 1 and negative class being 0. Suppose each classifier makes a correct prediction 51% of the times (weak!).

  Suppose a positive class is given. Let X_i be the prediction made by the ith classifier. If we view X_i as a random variable, $P(X_i = 1) = 0.51$.

  The number of positive predictions is equal to $\sum_{i=1}^{M} X_i$. If all the X_i are independent from each other, by the law of large numbers,

  $$\frac{1}{M}\sum_{i=1}^{M} X_i \to 0.51 \quad \text{almost surely as } M \to \infty.$$

  Hence, when M is sufficiently large, the majority vote is almost always correct.

- However, the above is only true if all classifiers are perfectly independent, making uncorrelated errors, which is clearly not the case because they are trained on the same data. They are likely to

make the same types of errors, so there will be many votes for the wrong class, reducing the ensemble's accuracy.

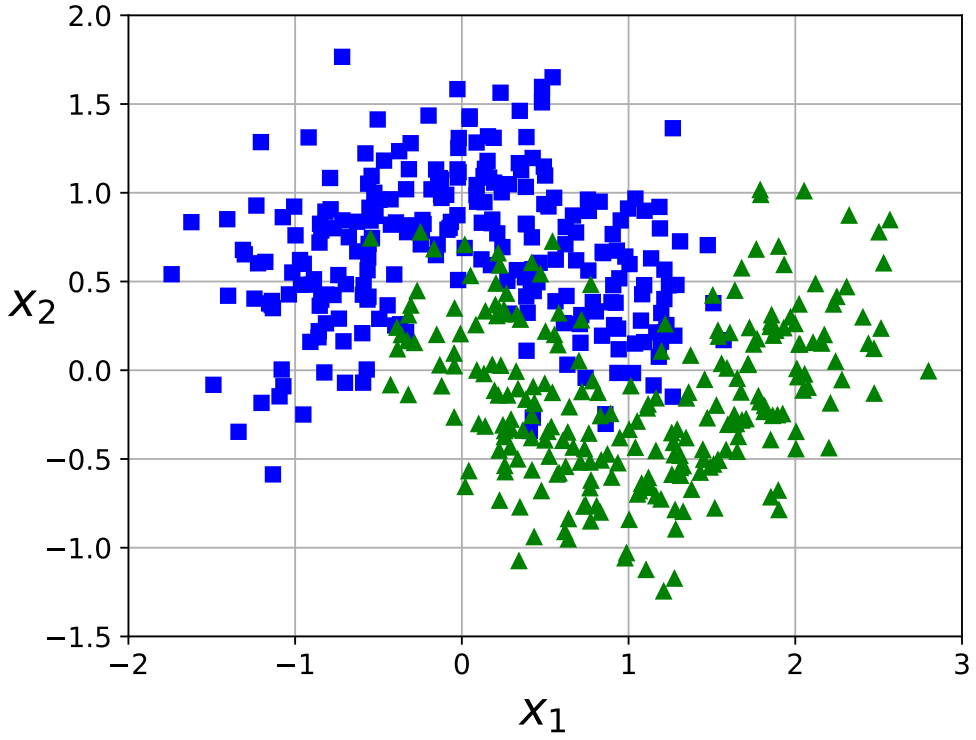
- Ensemble methods work best when the predictors are as independent from one another as possible. One way to get diverse classifiers is to train them using very different algorithms. This increases the chance that they will make very different types of errors, improving the ensemble's accuracy.

Voting Classifiers in Scikit-Learn

- Let's test this on the moons dataset: This is a toy dataset for binary classification in which the data points are shaped as two interleaving half circles. You can generate this dataset using the `make_moons()` function.

```
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=500, noise=0.30, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)
```



- Here is how to use the voting classifier.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC

log_clf = LogisticRegression(solver="lbfgs", random_state=42)
rnd_clf = RandomForestClassifier(n_estimators=100, random_state=42)
svm_clf = SVC(gamma="scale", random_state=42)

voting_clf = VotingClassifier(
    estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
    voting='hard')
```

```
voting_clf.fit(X_train, y_train)
```

By default, `SVC` uses the radial basis function (rbf) as the kernel. `gamma` is one of its parameters.

- Here are the accuracies on the test set.

```python
from sklearn.metrics import accuracy_score

for clf in (log_clf, rnd_clf, svm_clf, voting_clf):
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    print(clf.__class__.__name__, accuracy_score(y_test, y_pred))
```

```
LogisticRegression 0.864
RandomForestClassifier 0.896
SVC 0.896
VotingClassifier 0.912
```

Here, it is fine to use accuracy as the metric because the test set is balanced, with 61 and 64 instances for the two classes.

- Note the hyperparameter `voting='hard'`.

Soft Voting

- Suppose all classifiers are able to estimate class probabilities (i.e., they all have a `predict_proba()`.
- Soft voting computes the average probability for each class, and makes the prediction based on the highest average.
- By default, SVC does not have class probabilities. You need to set its `probability` hyperparameter to `True`. This will add a `predict_proba()` method.
  - For binary classification, the way to compute the class probabilities is based on Platt's method.
    `https://home.cs.colorado.edu/~mozer/Teaching/syllabi/6622/papers/Platt1999.pdf`
  - First, SVC uses cross-validation to generate scores for all the training instances. Then, a logistic regression model is trained

using the SVC scores as the input and the true classes of the training instances as the output.

  - Given a test instance, SVC first produces a score for the instance. The SVC score is used as the input to the logistic regression model, and the output is the probability of being in the positive class.
  - Due to the cross-validation step, the training process may be slow for a large dataset.
  - The method can be generalized to the multiclass case.

- For our current dataset (and random seed), soft voting gives a slightly better performance. The accuracy is 0.92.

VotingRegressor

- Scikit-Learn also has `VotingRegressor`.
- It fits several base regressors, each on the whole dataset. Then, it averages the individual predictions to form a final prediction.
- Optionally, the averaging can be weighted.

Bagging and Pasting

- Instead of using very different training algorithms, bagging and pasting use the same training algorithm but generate different classifiers/regressors by training on different random subsets of the training set.
- **Bagging:** (short for bootstrap aggregating) Sampling is performed with replacement.

  In statistics, resampling with replacement is called **bootstrapping**.
- **Pasting:** Sampling is performed without replacement.
- The aggregation function is typically the statistical mode (i.e., the most frequent prediction, just like a hard voting classifier) for classification, or the average for regression.
- Bagging and pasting are popular methods, partially because they

scale very well. Predictors can all be trained in parallel, via different CPU cores or even different servers.

- Scikit-Learn offers a simple API for both bagging and pasting with the `BaggingClassifier` class (or `BaggingRegressor` for regression).

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

bag_clf = BaggingClassifier(
    DecisionTreeClassifier(), n_estimators=500,
    max_samples=100, bootstrap=True, random_state=42)
bag_clf.fit(X_train, y_train)
y_pred = bag_clf.predict(X_test)
print(accuracy_score(y_test, y_pred))
```

```
0.904
```

  - `n_estimators=500`: The code trains an ensemble of 500 decision tree classifiers.
  - `max_samples=100`: Each tree is trained on 100 training

instances randomly sampled from the training set.

Duplicates are allowed in the 100 training instances.

- `bootstrap=True`: This is an example of bagging. If you want to use pasting instead, just set `bootstrap=False`.
- `n_jobs`: The number of jobs to run in parallel. The default is 1. If it is -1, it tells Scikit-Learn to use all processors.

- The `BaggingClassifier` automatically performs soft voting instead of hard voting if the base classifier can estimate class probabilities (i.e., if it has a `predict_proba()` method), which is the case with decision tree classifiers.

Compare the Decision Boundaries

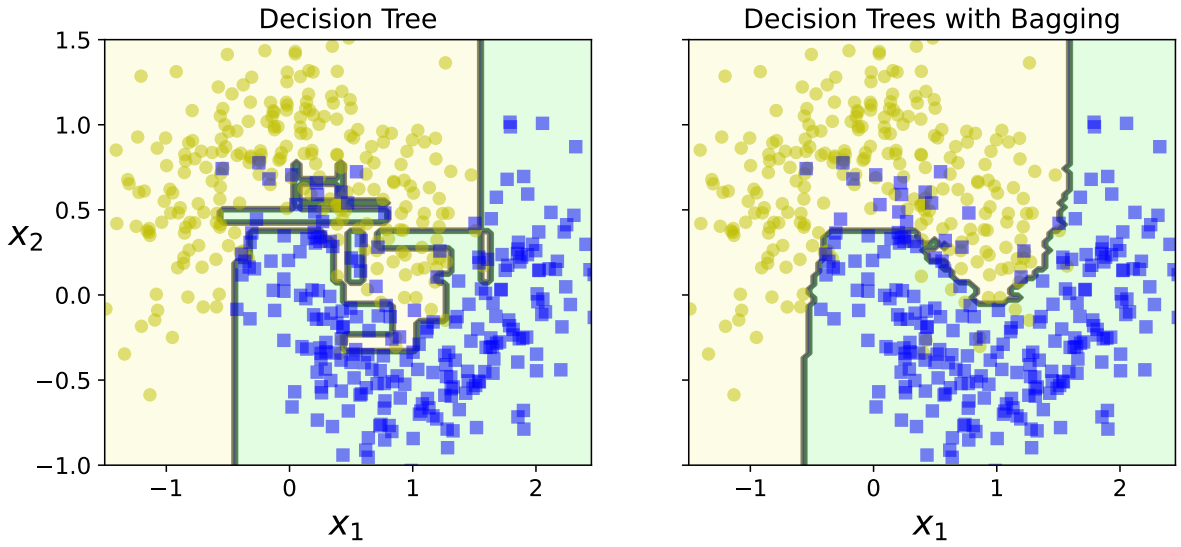


Figure 1: A single decision tree (left) versus a bagging ensemble of 500 trees (right)

- The ensemble's predictions should generalize better than the single decision tree, because the decision boundary for the single tree is

highly adapted to the training set, which suggests the single tree is overfitting.

The accuracy scores on the training set: ensemble (0.939); single tree (1)

- Indeed, by using `accuracy_score()` for the test set, we find that the bagging ensemble has an accuracy of 0.904 and the single decision tree has an accuracy of 0.856.
- Moreover, if a different training set is used, the decision boundary for the ensemble is less likely to change dramatically, and therefore, the ensemble is more likely to give the same prediction for a given test instance. This is especially relevant for the instances near the boundary.
- The stability of the decision boundary can be related to having a lower variance.

  To see why, suppose the two classes are coded by 1 and 0. Consider a given *test* instance with a label 1. The decision boundary being stable means that the probability

of correct classification, denoted by p, is either high or low. Here, the probability arises because the training dataset is random (or equivalently, with different sets of training data, p is the fraction of times when the test instance is correctly classified).

The expected value of classification is equal to $1 \cdot p + 0 \cdot (1 - p) = p$. The variance is equal to $(1 - p)^2 \cdot p + (0 - p)^2 \cdot (1 - p) = p(1 - p)$.

The variance is small when p is near 1 or near 0, and it is maximized when $p = 1/2$. Imagine that the decision boundaries change a lot from training set to training set so that the test instance may get classified as 1 half of the times and 0 the other half of the times; in that case, the variance is high.

Note that the prediction error, as it is usually defined, is equal to $(0 - 1)^2 \cdot (1 - p) + (1 - 1) \cdot p = (1 - p)$. This is exactly the probability of mis-classification for the given positive instance.

For $p = 1/2$, the prediction error is relatively high and the variance is also high.

However, it is not true that a low variance implies a low prediction error. When p is small, the variance is small, but, the prediction error $1 - p$ is large. In this case, the squared bias is equal to $((1 - 1) \cdot p + (1 - 0) \cdot (1 - p))^2 = (1 - p)^2$, which is large.

Here, we also see the decomposition of the prediction error into the sum of the squared bias and the variance: $(1 - p)^2 + p(1 - p) = 1 - p$.

The accuracy score is a different kind of metric from the bias, mis-classification probability or prediction error. It is an average over all the test instances, but under a single set of training data, and hence, random.

Compare Bagging and Pasting

- In bagging, the same instance can show up multiple times in the same training subset. Also, due to statistical fluctuation, how many times each instance is sampled is uneven. Bagging endss up with a slightly higher bias than pasting.
- However, the predictors in bagging are less correlated, so the ensemble's variance is reduced.
- For a small dataset, bagging is preferred.
- Overall, bagging often results in better models, which explains why it is generally preferred.

Bagging: Out-of-Bag Evaluation

- With bagging, some instances may be sampled several times for any given predictor, while others may not be sampled at all. The latter instances are called **out-of-bag (oob)** instances.
- Since a predictor never sees the oob instances during training, it can be evaluated on these instances to estimate the generalization error, without the need for a separate validation set.
- The ensemble itself can be evaluated by averaging out the oob evaluations of each predictor.
- You can set `oob_score=True` when creating a `BaggingClassifier` to request an automatic oob evaluation after training.

```
bag_clf = BaggingClassifier(
    DecisionTreeClassifier(), n_estimators=500,
    bootstrap=True, oob_score=True, random_state=40)
bag_clf.fit(X_train, y_train)
bag_clf.oob_score_
```

```
0.8986666666666666
```

- The above code doesn't specify `max_samples`. The default value is `max_samples=1.0`. When `max_samples` is a `float`, each predictor draws $m \times$ `max_samples`, where m is the total number of training instances. Here, each predictor makes exactly m draws.
- According to the oob evaluation, this `BaggingClassifier` is likely to achieve about 89.9% accuracy on the test set. We next verify this on the test set:

```
from sklearn.metrics import accuracy_score
y_pred = bag_clf.predict(X_test)
accuracy_score(y_test, y_pred)
```

```
0.912
```

- The oob decision function for each training instance is also available through the `oob_decision_function_` variable.

  In the current case, the base estimator has a `predict_proba()` method. The decision function returns the class probabilities for each training instance.

```
bag_clf.oob_decision_function_
```

```
array([[0.32275132, 0.67724868],
       [0.34117647, 0.65882353],
       [1.        , 0.        ],
       [0.        , 1.        ],
       [0.        , 1.        ],
       [0.09497207, 0.90502793],
```

Bagging: How Many OOB Instances

- We will first compute the expected number of distinct instances in m draws.

  Suppose we make m draws with replacement from a training set with m instances. Let Z_m be the number of distinct instances drawn.

  For $i = 1, 2, \ldots, m$, let us define an indicator variable I_i by

$$I_i = \begin{cases} 1 & \text{if instance } i \text{ is drawn} \\ 0 & \text{otherwise.} \end{cases}$$

  For any i, the probability that instance i is not drawn is given by

$$P(I_i = 0) = \left(1 - \frac{1}{m}\right)^m .$$

  Therefore,

$$P(I_i = 1) = 1 - \left(1 - \frac{1}{m}\right)^m .$$

Since

$$Z_m = \sum_{i=1}^{m} I_i,$$

$$E[Z_m] = \sum_{i=1}^{m} E[I_i] = \sum_{i=1}^{m} P(I_i = 1) = m\left(1 - \left(1 - \frac{1}{m}\right)^m\right).$$

Therefore,

$$\frac{E[Z_m]}{m} = \left(1 - \left(1 - \frac{1}{m}\right)^m\right).$$

As $m \to \infty$, $\left(1 - \frac{1}{m}\right)^m \to e^{-1}$, and hence,

$$\frac{E[Z_m]}{m} \to 1 - e^{-1} \approx 0.632.$$

- Conclusion: For large m, the expected number of distinct instances in m draws is approximately 63.2% of the total number of instances. The rest 36.8% are out-of-bag instances, which can be used for out-of-bag evaluation.

Random Patches and Random Subspaces

- The `BaggingClassifier` class supports sampling the features as well. Sampling is controlled by two hyperparameters: `max_features` and `bootstrap_features`.
- This technique is particularly useful when you are dealing with high-dimensional inputs (such as images).
- Sampling both training instances and features is called the **Random Patches** method.
- Keeping all training instances (by setting `bootstrap=False` and `max_samples=1.0`) but sampling features (by setting `bootstrap_features=True` and/or `max_features` to a value smaller than 1.0) is called the **Random Subspaces** method.

Random Forest

- A random forest is an ensemble of decision trees, generally trained via the bagging method (or sometimes pasting), typically with `max_samples` set to the size of the training set.
- Instead of building a `BaggingClassifier` and passing it a `DecisionTreeClassifier`, you can instead use the `RandomForestClassifier` class, which is more convenient and optimized for decision trees.

```
from sklearn.ensemble import RandomForestClassifier

rnd_clf = RandomForestClassifier(n_estimators=500, max_leaf_nodes=16, random_state=42)
rnd_clf.fit(X_train, y_train)

y_pred_rf = rnd_clf.predict(X_test)
```

- With a few exceptions, a `RandomForestClassifier` has all the hyperparameters of a `DecisionTreeClassifier` (to control

how trees are grown), plus all the hyperparameters of a `BaggingClassifier` to control the ensemble itself.

- The random forest algorithm introduces extra randomness when growing a tree; instead of searching for the very best feature when splitting a node, it searches for the best feature among a random subset of features.

  In the code above, the hyperparameter `max_features` takes the default value `"sqrt"`, which means `max_features=sqrt(n_features)`.

  Other choices are `"log2"`, `None`, int or float.

- The algorithm trades a higher bias for a lower variance, generally yielding an overall better model.

- There is also a `RandomForestRegressor` class for regression.

Random Forest: Feature Importance

- Scikit-Learn measures a feature's importance by looking at how much the tree nodes that use that feature reduce impurity on average (across all trees in the forest).

  More precisely, it is a weighted average, where each node's weight is equal to the number of training samples that are associated with it.

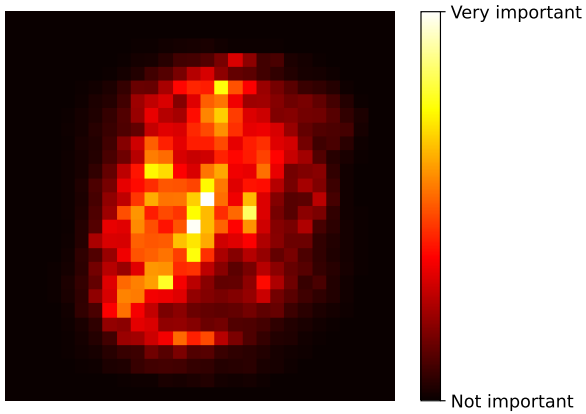
- Scikit-Learn computes this score automatically for each feature after training, then it scales the results so that the sum of all importances is equal to 1.

  Example: `RandomForestClassifier` on the iris dataset

```python
from sklearn.datasets import load_iris
iris = load_iris()
rnd_clf = RandomForestClassifier(n_estimators=500, random_state=42)
rnd_clf.fit(iris["data"], iris["target"])
for name, score in zip(iris["feature_names"], rnd_clf.feature_importances_):
    print(name, score)
```

```
sepal length (cm) 0.11249225099876375
sepal width (cm) 0.02311928828251033
petal length (cm) 0.4410304643639577
petal width (cm) 0.4233579963547682
```

Example: `RandomForestClassifier` on the MNIST dataset - feature importance



Boosting

- *Boosting* refers to any ensemble method that can combine several weak learners into a strong learner.

  Weak Learner: A predictor that works slightly better than chance.

- The general idea of most boosting methods is to train predictors sequentially, each trying to correct its predecessor.
- The most popular boosting methods are AdaBoost (short for Adaptive Boosting) and Gradient Boosting.

AdaBoost

- One way for a new predictor to correct its predecessor is to pay a bit more attention to the training instances that the predecessor underfitted. This results in new predictors focusing more and more on the hard cases.
- A classifier example:
  - The algorithm first trains a base classifier (such as a decision tree) and uses it to make predictions on the training set.
  - The algorithm then increases the relative weight of misclassified training instances.
  - Then it trains a second classifier, using the updated weights; then again makes predictions on the training set and updates the instance weights.
  - The algorithm stops when the desired number of predictors is

reached, or when a perfect predictor is found.

– To make predictions, AdaBoost computes the predictions of all the predictors and weighs them using the predictor weights α_j. The predicted class is the one that receives the majority of weighted votes.

AdaBoost for Binary Classification and Exponential Loss

- Let $(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)$ be the training instances, where each $y_i \in \{-1, 1\}$.
- Let ϕ_k denote the k-th classifier, with $\phi_k(x_i) \in \{-1, 1\}$. Will train K classifiers.
- $(w_{i,k})_{i=1}^m$: the weights of instances in training ϕ_k.
- $\eta > 0$: the learning rate.

Initialize $w_{i,1} = 1/m$ for $i = 1, \ldots, m$.
Repeat the following for $k = 1, \ldots, K$.

1. Training ϕ_k using the training set with weights $(w_{i,k})_{i=1}^m$.

2. Compute weighted error rate of the k-th classifier:

$$\epsilon_k = \frac{\sum_{y_i \neq \phi_k(x_i)} w_{i,k}}{\sum_{i=1}^{m} w_{i,k}},$$

3. Compute the weight of the k-th classifier:

$$\alpha_k = \eta \log \frac{1-\epsilon_k}{\epsilon_k}.$$

4. Update instance weights (if $k < K$): For each i,

$$w_{i,k+1} = \begin{cases} w_{i,k} & \text{if } y_i = \phi_k(x_i) \\ w_{i,k} e^{\alpha_k} & \text{if } y_i \neq \phi_k(x_i). \end{cases}$$

   Then, normalize the weights: For each i,

$$w_{i,k+1} \Leftarrow \frac{w_{i,k+1}}{\sum_{i=1}^{m} w_{i,k+1}}.$$

- Note: $0 \leq \epsilon_k \leq 1$; $\alpha_k > 0$ if $\epsilon_k < 1/2$

- For prediction, use:

$$f(x) = \text{sgn}\left(\sum_{k=1}^{K} \alpha_k \phi_k(x)\right) .$$

Exponential Loss Function

- The problem of classification (or regression) is often cast as finding a function f^* such that

$$f^* \in \operatorname*{argmin}_f \mathrm{E}[L(Y, f(X))],$$

  where $L(y, z)$ is a loss function. The expectation is taken over the joint distribution $P(X, Y)$, which is usually unknown.

- The optimization can be carried out point-wise. That is, for each given input x, we have

$$\mathrm{E}[L(Y, f(X))|X = x] = \mathrm{E}[L(Y, f(x))|X = x].$$

  Suppose $a_x^* \in \operatorname{argmin}_a \mathrm{E}[L(Y, a)|X = x]$. Then f^* is given by $f^*(x) = a_x^*$ for each x. The expectation is over the conditional probability $P(Y|X = x)$, which is still unknown.

- The empirical version is to use the training instances $(x_1, y_1), \ldots, (x_m, y_m)$:

$$\min_f \sum_{i=1}^{m} L(y_i, f(x_i)).$$

- For binary classification, the exponential loss function is often used. Suppose the output y is coded as -1 or 1. The **exponential loss function** is defined as

$$L(y, z) = \exp(-yz),$$

which implies,

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

Here, z and $f(x)$ are understood to be real-valued. The actual predicted output is usually $\text{sgn}(f(x))$.

With the exponential loss function, we can solve the optimization

problem

$$\min_a \mathrm{E}[L(Y,a)|X=x] = \min_a \mathrm{E}[\exp(-Ya)|X=x]$$
$$= \min_a \left(\exp(-a)P(Y=1|X=x) + \exp(a)P(Y=-1|X=x)\right).$$

To find a minimum a, we take derivative of the function with respect to a and set the result to 0. (It is easy to check the second derivative of the objective function is positive, and therefore, the function is convex. The local minimum is the global minimum.)

$$-\exp(-a)P(Y=1|X=x) + \exp(a)P(Y=-1|X=x) = 0.$$

We get

$$\exp(2a) = \frac{P(Y=1|X=x)}{P(Y=-1|X=x)}.$$

Then,

$$a = \frac{1}{2}\log\frac{P(Y=1|X=x)}{P(Y=-1|X=x)} = \frac{1}{2}\log\frac{P(Y=1|X=x)}{1-P(Y=1|X=x)}.$$

Thus, we have found that the optimal prediction function f^* is

$$f^*(x) = \frac{1}{2} \log \frac{P(Y = 1|X = x)}{1 - P(Y = 1|X = x)}.$$

AdaBoost – Derivation

- Suppose we have a set of weak classifiers $\phi_1, \ldots, \phi_K$, where each $\phi_k(x_i) \in \{-1, 1\}$.
- We get a sequence of boosted classifiers from linear combinations of the ϕ_k's:

$$f_k(x) = \sum_{t=1}^{k} \alpha_t \phi_t(x) = f_{k-1}(x) + \alpha_k \phi_k(x).$$

  Here, $f_k(x)$ is not necessarily -1 or 1.

- The final model is f_K and the prediction for an input x is $\text{sgn}(f_K(x))$.
  - We see that using the final boosted model is like performing weighted voting of the binary weak classifiers $\phi_1, \ldots, \phi_K$, where the classifier weights are the α_k's.

- We will also see that although each ϕ_k is trained on the entire training set, more emphasis is given a subset of the training instances by assigning more sample weights to those instances.

- The process of boosting is sequential. At each step t, we have already found $\phi_1, \ldots, \phi_{k-1}$ and $\alpha_1, \ldots, \alpha_{k-1}$, and hence, f_{k-1}; we wish to find ϕ_k and α_k.
- The empirical version of the total error at step k is

$$E_k = \sum_{i=1}^{m} \exp(-y_i f_k(x_i)) = \sum_{i=1}^{m} \exp(-y_i f_{k-1}(x_i)) \exp(-y_i \alpha_k \phi_k(x_i)).$$

- For each instance i, let us define the instance weight $w_{i,1} = 1$, and for $k \geq 2$,

$$w_{i,k} = \exp(-y_i f_{k-1}(x_i)).$$

- Then,

$$E_k = \sum_{i=1}^{m} w_{i,k} \exp(-y_i \alpha_k \phi_k(x_i)).$$

- We next split the sum according to the instances that are correctly or incorrectly classified by ϕ_k.

$$
\begin{aligned}
E_k &= \sum_{y_i = \phi_k(x_i)} w_{i,k} \exp(-y_i \alpha_k \phi_k(x_i)) + \sum_{y_i \neq \phi_k(x_i)} w_{i,k} \exp(-y_i \alpha_k \phi_k(x_i)) \\
&= \sum_{y_i = \phi_k(x_i)} w_{i,k} \exp(-\alpha_k) + \sum_{y_i \neq \phi_k(x_i)} w_{i,k} \exp(\alpha_k) \qquad (1) \\
&= \sum_{i=1}^{m} w_{i,k} \exp(-\alpha_k) + \sum_{y_i \neq \phi_k(x_i)} w_{i,k} (\exp(\alpha_k) - \exp(-\alpha_k)).
\end{aligned}
$$

- Only the second term above has ϕ_k. To truly minimize E_k, we would have set $\phi_k(x_i) = y_i$ for all i so that the second term is equal to 0.

  However, this is unlikely to be possible because ϕ_k is a weak learner from a particular family, such as a decision tree with depth 1. In fact, we don't want to set $\phi_k(x_i) = y_i$ for all i since it is clearly overfitting.

  Note: This suggests that having a family of weak learners is crucial for boosting to work.

- Given the family of weak learners, we will try to do the best, which is to fit the training instances but with the instance weights $w_{i,k}$ for each instance i.
  - The factor $\exp(\alpha_k) - \exp(-\alpha_k)$ does not matter for the fitting because it is the same for all the instances.
  - The training result will be the same if we use the normalized instance weights $\hat{w}_{i,k}$, where for each instance i,

$$\hat{w}_{i,k} = \frac{w_{i,k}}{\sum_{j=1}^{m} w_{j,k}}.$$

  - How the weights are used in the weak learner ϕ_k depends on the type of the learner. See later.

- Once ϕ_k is decided, we choose the α_k to minimize E_k. It can be verified that E_k is a convex function of α_k. From (1), we take derivative of E_k with respect to α_k and set it to 0. This leads to

$$-\exp(-\alpha_k) \sum_{y_i = \phi_k(x_i)} w_{i,k} + \exp(\alpha_k) \sum_{y_i \neq \phi_k(x_i)} w_{i,k} = 0,$$

which has the same solution as

$$-\sum_{y_i=\phi_k(x_i)} w_{i,k} + \exp(2\alpha_k) \sum_{y_i \neq \phi_k(x_i)} w_{i,k} = 0.$$

We get

$$\begin{aligned}\alpha_k &= \frac{1}{2}\log\left(\frac{\sum_{y_i=\phi_k(x_i)} w_{i,k}}{\sum_{y_i\neq\phi_k(x_i)} w_{i,k}}\right) \\ &= \frac{1}{2}\log\left(\frac{\sum_{y_i=\phi_k(x_i)} \hat{w}_{i,k}}{\sum_{y_i\neq\phi_k(x_i)} \hat{w}_{i,k}}\right). \end{aligned} \tag{2}$$

- As before, let us define the weighted error rate ϵ_k:

$$\epsilon_k = \frac{\sum_{y_i\neq\phi_k(x_i)} w_{i,k}}{\sum_{i=1}^m w_{i,k}} = \frac{\sum_{y_i\neq\phi_k(x_i)} \hat{w}_{i,k}}{\sum_{i=1}^m \hat{w}_{i,k}}. \tag{3}$$

We see that

$$\alpha_k = \frac{1}{2}\log\left(\frac{1-\epsilon_k}{\epsilon_k}\right).$$

Note that $\alpha_k > 0$ iff $\epsilon_k < 0.5$. We need the weak learner to be able to bring the weighted error rate below 0.5.

Moreover, the more accurate ϕ_k is (the smaller ϵ_k is), the larger the classifier weight α_k is.

- We also have

$$\begin{aligned} w_{i,k+1} &= \exp(-y_i f_k(x_i)) = \exp(-y_i f_{k-1}(x_i) - y_i \alpha_k \phi_k(x_i)) \\ &= w_{i,k} \exp(-y_i \alpha_k \phi_k(x_i)) \\ &= \begin{cases} w_{i,k} \exp(-\alpha_k) & \text{if } y_i = \phi_k(x_i) \\ w_{i,k} \exp(\alpha_k) & \text{if } y_i \neq \phi_k(x_i). \end{cases} \end{aligned}$$

  As we expect $\alpha_k > 0$, we see that the weight increases for mis-classified instances, and decreases for correctly classified instances.

- In the boosting algorithm, for each k, the instance weights are used in

three places: the calculation α_k as shown in (2), the calculation ϵ_k as shown in (3), and the fitting of ϕ_k to the weighted instances.

In all three places, we can use the **normalized** weights $\hat{w}_{i,k}$.

If we multiply all the instance weights by a common factor, say $\mu_k > 0$, the normalized weights are unchanged.

Since the weight can be written as

$$w_{i,k+1} = \exp(-\alpha_k) \begin{cases} w_{i,k} & \text{if } y_i = \phi_k(x_i) \\ w_{i,k} \exp(2\alpha_k) & \text{if } y_i \neq \phi_k(x_i), \end{cases}$$

we can get rid of the common factor $\exp(-\alpha_k)$. Then, the weight update becomes

$$w_{i,k+1} = \begin{cases} w_{i,k} & \text{if } y_i = \phi_k(x_i) \\ w_{i,k} \exp(2\alpha_k) & \text{if } y_i \neq \phi_k(x_i), \end{cases}$$

which is what's used in the earlier boosting algorithm.

Boosting: How the Instance (Sample) Weights Are Used

- How the instance weights are used in training the weak learner ϕ_k depends on the type of the learner.
- If ϕ_k is a decision tree, the weights are used to compute the fractional numbers $p_{s,c}$ at each tree node s, which is now the *sample-weighted* fraction of instances at node s that belong to class c:

$$p_{s,c} = \frac{\sum_{i \in s, y_i = c} w_{i,k}}{\sum_{i \in s} w_{i,k}}.$$

  The impurity value at s is computed using $p_{s,c}$ for all classes c.

  Recall that in the unweighted version (i.e., all instance weights are the same), $p_{s,c}$ is simply the fraction of instances at node s that belong to class c.

- If ϕ_k is a support vector machine (SVM), the sample weights rescale

the `C` parameter, which means that the classifier puts more emphasis on getting these points right.

SVM solves the following type of problem, where C is a parameter. In the weighted version of SVM, C becomes C_i, which depends on the weight of instance i.

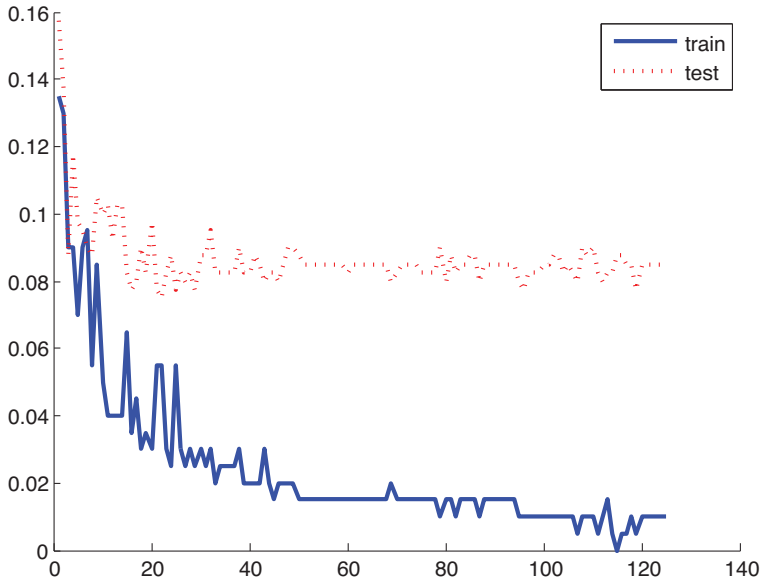
$$\textbf{(soft margin)} \quad \min_{u,b,\xi} \frac{1}{2} u^T u + \sum_{i=1}^{m} C\xi_i$$

$$\text{subject to} \quad t_i(u^T x_i + b) \geq 1 - \xi_i, \text{ and } \xi_i \geq 0, \text{ for } i = 1, \ldots, m.$$

- For other families of learners, ϕ_k may be trained using a resampled set of instances from the original training set. Then, the normalized weights can be used as sampling probabilities.

Boosting: Benefits

- Boosting is often very resistant to overfitting.



- We see that the training set error rapidly goes to near zero. The test set error continues to decline even after the training set error has reached zero (although the test error will eventually go up).

Why Boosting Works So Well

- Boosting can be seen as a form of l_1 regularization, which is known to help prevent overfitting by eliminating less-useful features.
- For a given input's original feature vector x, $\phi(x) = (\phi_1(x), \ldots, \phi_K(x))$ can be viewed as engineered features.
- If the dimension of the engineered feature vector $\phi(x)$ is very large, one usually uses l_1 regularization to select a subset of these.
- When we can use boosting, at each step, a new ϕ_k is created on the fly, and its importance is automatically computed, which is the classifier weight α_k.

AdaBoost in Scikit-Learn

- **Decision Stump:** a decision tree with `max_depth=1`, i.e., a tree composed of a single decision node plus two leaf nodes.
- The following code trains an AdaBoost classifier based on 200 decision stumps using Scikit-Learn's `AdaBoostClassifier` class.

```
from sklearn.ensemble import AdaBoostClassifier

ada_clf = AdaBoostClassifier(
    DecisionTreeClassifier(max_depth=1), n_estimators=200,
    algorithm="SAMME.R", learning_rate=0.5, random_state=42)
ada_clf.fit(X_train, y_train)
```

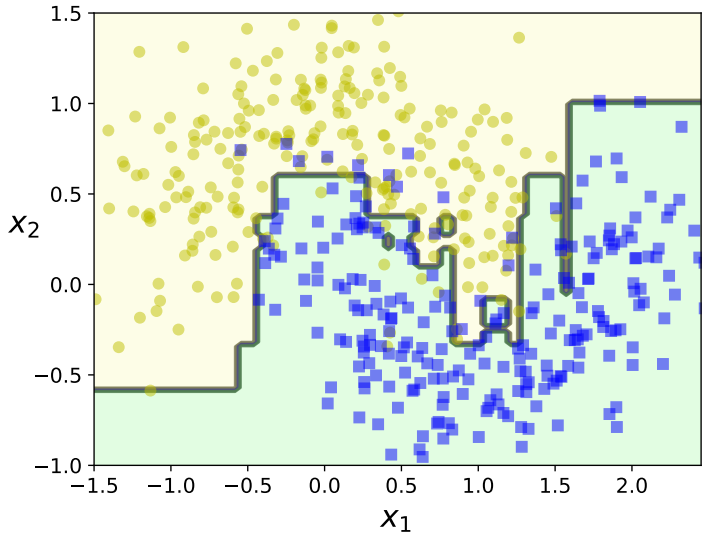


Figure 2: Decision boundary when using SAMME.R. There seems to be some overfitting.

- Scikit-Learn uses a multiclass version of AdaBoost called SAMME (which stands for Stagewise Additive Modeling using a Multiclass Exponential loss function).
- When there are just two classes, SAMME is equivalent to AdaBoost.

- If the predictors can estimate class probabilities (i.e., if they have a `predict_proba()` method), Scikit-Learn can use a variant of SAMME called SAMME.R (the R stands for“Real”), which relies on class probabilities rather than predictions.

  Note: Since version 1.4: SAMME.R is deprecated and will be removed in version 1.6. SAMME will become the default.

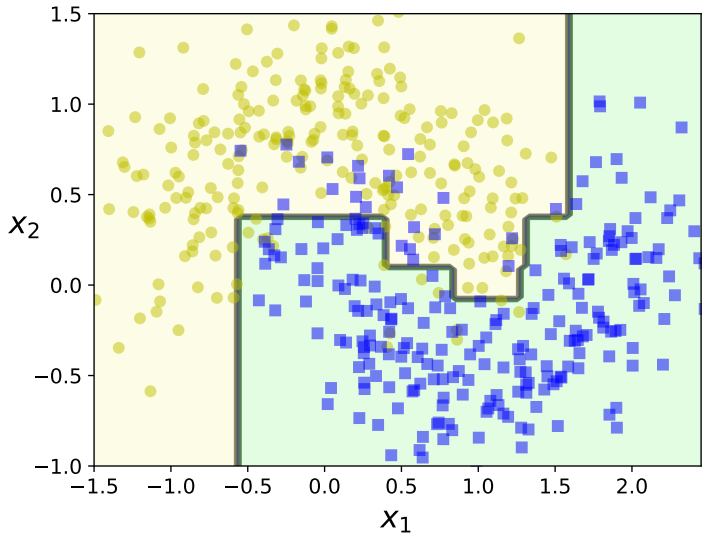


Figure 3: Decision boundary when using SAMME. Less overfitting.

- More generally, if your AdaBoost ensemble is overfitting the training set, you can try reducing the number of estimators or more strongly regularizing the base estimator.