Lab 9: Adaboost

Due Friday by 11:59pm **Points** 5 **Submitting** an external tool
Available Nov 2 at 12:01am - Dec 12 at 11:59pm about 1 month

Lab 9: Adaboost

To complete Lab 9, download: lab9.zip.

Contents

Boosting allows us to combine the strengths of several different, imperfect classifiers

The machine learning classifiers we've been discussing in class are powerful when used correctly, but they are not perfect. Even the best classifiers, under the best of conditions, misclassify some samples. However, due to the nature of each different machine learning algorithm, different classifiers often make different kinds of mistakes. We can take advantage of this variety in errors, using the (reasonable) assumption that *most* classifiers will correctly classify each particular point, to construct an even more powerful **ensemble** or **amalgam** classifier, H, that can correctly classify all or most samples.

Adaboost

The boosting algorithm we teach in 6.034 is called Adaboost, short for *adaptive boosting*. In this lab, you will implement the Adaboost algorithm to perform boosting.

Throughout this lab, we will assume that there are exactly two classifications, meaning that every not-misclassified training point is classified correctly.

A note about floats, roundoff error, and `Fraction`s

Python sometimes rounds floating point numbers, especially when adding them to dictionaries, which can result in two numbers being off by 0.0000000000000001 (10^-16) or more when they should be equal! For example:

```
>>> 1/3.0
0.3333333333333333
>>> 2/3.0
0.6666666666666666
>>> 1 - 2/3.0
0.33333333333333337
>>> 1/3.0 == 1 - 2/3.0
False
```

To avoid dealing with this roundoff error, we recommend using the `Fraction` class. We've already imported it into the lab for you in `utils.py`, and your `lab9.py` file imports it from there. We've also written a function to convert floats or pairs of numbers (numerator and denominator) into fractions:

- `make_fraction(n)`: Returns a `Fraction` approximately equal to `n`, rounding to the nearest fraction with denominator <= 1000.
- `make_fraction(n, d)`: If `n` and `d` are both integers, returns a `Fraction` representing `n/d`. Otherwise, returns a `Fraction` approximately equal to `n/d`, rounding to the nearest fraction with denominator <= 1000.

You can manipulate and compare `Fraction`s just like any other number type. In fact, you can even make new `Fraction`s out of them! Here are a few examples:

```
>>> frac1 = make_fraction(2,6)  # make_fraction can take in two numbers and reduce them to a s
imple Fraction
>>> frac1
1/3
>>> frac2 = make_fraction(1,4)
>>> frac2
1/4
>>> frac2 == 0.25               # Fractions are considered equal to their equivalent floats or
ints
True
>>> frac1 + frac2               # You can add, subtract, multiply, and divide Fractions
```

```
7/12
>>> frac1 * 5                   # You can combine a Fraction and an int to get a new Fraction
5/3
>>> make_fraction(frac1, frac2) # You can make a new Fraction out of any two numbers -- includ
ing Fractions!
4/3
>>> make_fraction(0.9)          # make_fraction can also take in a single number (float, int,
long, or Fraction)
9/10
>>>
```

In this lab, we recommend using `int`s or `Fraction`s, as opposed to floats, for nearly everything numeric. The one exception to this rule is for voting powers, which we recommend representing as floats because they involve logarithms.

With all of that having being said, if you think you've implemented a function correctly but you're not passing all of the tests, make sure you're correctly using `Fraction`s instead of raw floats!

Part 1: Helper functions

We will implement Adaboost piece-by-piece, modularizing the different steps of the algorithm in a similar way to how we taught Adaboost in recitation. As a brief reminder, here is the general control flow of the Adaboost algorithm:

1. Initialize all training points' weights.
2. Compute the error rate of each weak classifier.
3. Pick the "best" weak classifier *h*, by some definition of "best."
4. Use the error rate of *h* to compute the voting power for *h*.
5. Append *h*, along with its voting power, to the ensemble classifier H.
6. Update weights in preparation for the next round.
7. Repeat steps 2-7 until no good classifier remains, we have reached some max number of iterations, or H is "good enough."

Initialize weights

First, implement `initialize_weights` to assign every training point a weight equal to 1/N, where N is the number of training points. This function takes in one argument, `training_points`, which is a list of training points, where each point is represented as a string. The function should return a dictionary mapping points to weights.

```
def initialize_weights(training_points):
```

Calculate error rates

Next, we want to calculate the error rate ε of each classifier. The error rate for a classifier *h* is the sum of the weights of the training points that *h* misclassifies.

`calculate_error_rates` takes as input two dictionaries:

- `point_to_weight`: maps each training point (represented as a string) to its weight (a number).
- `classifier_to_misclassified`: maps each classifier (a string) to a list of the training points (strings) that it misclassifies. For example, this dictionary may contain entries such as `"classifier_0": ["point_A", "point_C"]`, indicating `classifier_0` misclassifies points A and C.

Implement `calculate_error_rates` to return a dictionary mapping each weak classifier (a string) to its error rate (a number):

```
def calculate_error_rates(point_to_weight, classifier_to_misclassified):
```

Pick the best weak classifier

Once we have calculated the error rate of each weak classifier, we need to select the "best" weak classifier. Implement `pick_best_classifier` to return the name of the "best" weak classifier, or raise a `NoGoodClassifiersError` if the "best" weak classifier has an error rate of exactly 1/2. Note that "best" has two possible definitions:

- "smallest error rate" (`use_smallest_error=True`), *or*,
- "error rate furthest from 1/2" (`use_smallest_error=False`)

```
def pick_best_classifier(classifier_to_error_rate, use_smallest_error=True):
```

If two or more weak classifiers have equally good error rates, return the one that comes first alphabetically.

Hint: Depending on your implementation for this function, you may be interested in using `min` or `max` on a dictionary. Recall that `min` and `max` can take an optional `key` argument, which works similarly to the `key` argument for `sorted`.

Calculate voting power

After selecting the best weak classifier, we'll need to compute its voting power. Recall that if ε is the error rate of the weak classifier, then its voting power is:

```
1/2 * ln((1-ε)/ε)
```

Implement `calculate_voting_power` to compute a classifier's voting power, given its error rate $0 \leq \varepsilon \leq 1$. (For your convenience, the constant `INF` and the natural log function `ln` have been defined in `lab9.py`.)

```
def calculate_voting_power(error_rate):
```

Hint: What voting power would you give to a weak classifier that classifies all the training points correctly? What if it misclassifies all the training points?

Is H good enough?

One of the three exit conditions for Adaboost is when the overall classifier H is "good enough" -- that is, when H correctly classifies enough of the training points.

First, implement a helper function to determine which training points are misclassified by H:

```
def get_overall_misclassifications(H, training_points, classifier_to_misclassified):
```

The function `get_overall_misclassifications` takes in three arguments:

- `H`: An overall classifier, represented as a list of `(classifier, voting_power)` tuples. (Recall that each classifier is represented as a string.)
- `training_points`: A list of all training points. (Recall that each training point is represented as a string.)
- `classifier_to_misclassified`: A dictionary mapping each classifier to a list of the training points that it misclassifies.

The function returns a set containing the training points that H misclassifies.

Each training point's classification is determined by a weighted vote of the weak classifiers in H. Although we don't know any point's true classification (+1 or -1), we do know whether each point was classified correctly or incorrectly by each weak classifier, and we know that this is a binary classification problem, so there is sufficient information to determine which points were misclassified.

If the vote among the weak classifiers in H is ever a tie, consider the training point to be misclassified.

Now, we can determine whether H is "good enough". The function `is_good_enough` takes in the three arguments from `get_overall_misclassifications`, as well as fourth argument:

- `mistake_tolerance`: The maximum number of points that H is allowed to misclassify while still being considered "good enough."

`is_good_enough` should return `False` if H is misclassifies too many points, otherwise `True` (because H is "good enough").

```
def is_good_enough(H, training_points, classifier_to_misclassified, mistake_tolerance=0):
```

Update weights

After each round, Adaboost updates weights in preparation for the next round. The updated weight for each point depends on whether the point was classified correctly or incorrectly by the current weak classifier:

- If the point was classified correctly: `new weight = 1/2 * 1/(1-ε) * (old weight)`
- If the point was misclassified: `new weight = 1/2 * 1/ε * (old weight)`

The function `update_weights` takes in 3 arguments:

- `point_to_weight`: A dictionary mapping each point to its current weight. You are allowed to modify this dictionary, but are not required to.
- `misclassified_points`: A list of points misclassified by the current weak classifier.
- `error_rate`: The error rate ε of the current weak classifier.

Implement `update_weights` to return a dictionary mapping each point to its new weight:

```
def update_weights(point_to_weight, misclassified_points, error_rate):
```

Part 2: Adaboost

Using all of the helper functions you've written above, implement the **Adaboost algorithm**:

```
def adaboost(training_points, classifier_to_misclassified,
             use_smallest_error=True, mistake_tolerance=0, max_rounds=INF):
```

The function `adaboost` takes in five arguments:

- `training_points`: A list of all training points.
- `classifier_to_misclassified`: A dictionary mapping each classifier to a list of the training points that it misclassifies.
- `use_smallest_error`: A boolean value indicating which definition of "best" to use: "smallest error rate" (`True`) or "error rate furthest from 1/2" (`False`).
- `mistake_tolerance`: The maximum number of points that H is allowed to misclassify.
- `max_rounds`: The maximum number of rounds of boosting to perform before returning H. (This is equivalent to the maximum number of tuples that can be added to H.)

`adaboost` should return the overall classifier H, represented as a list of `(classifier, voting_power)` tuples.

Keep in mind that Adaboost has three exit conditions:

- If H is "good enough" (see **Is H good enough?**, above)
- If it has completed the maximum number of rounds
- If the best weak classifier is no good (has an error rate ε = 1/2)

Important Notes

Recall that weight updates are based on the points that were misclassified by the *current weak classifier*, not by the overall classifier H.

If you are failing test cases online but believe your implementation to be correct, please ensure that you are breaking ties correctly in your helper functions.

Survey

Please answer these questions at the bottom of your lab file:

- `NAME`: What is your name? (string)
- `COLLABORATORS`: Other than 6.034 staff, whom did you work with on this lab? (string, or empty string if you worked alone)
- `HOW_MANY_HOURS_THIS_LAB_TOOK`: Approximately how many hours did you spend on this lab? (number or string)
- `WHAT_I_FOUND_INTERESTING`: Which parts of this lab, if any, did you find interesting? (string)
- `WHAT_I_FOUND_BORING`: Which parts of this lab, if any, did you find boring or tedious? (string)
- (optional) `SUGGESTIONS`: What specific changes would you recommend, if any, to improve this lab for future years? (string)

(We'd ask which parts you find confusing, but if you're confused you should really ask a TA.)

Submit Programming Assignment

Upload all files for your submission

SUBMISSION METHOD

Upload GitHub Bitbucket

DRAG & DROP

Any file(s) including .zip. Click to browse.