adaboost_project.py

```python
import numpy as np
from typing import TypeVar, Iterable, Tuple, List, Dict
from typing import Tuple
import math


"""
Description:
    We are asked to create a decision stump. From the text:
    Let x denote a a one-dimensional attribute and y denote
    the class label. Suppose we use only one-level binary
    decision trees, with a test condition x <= k, where k
    is a split position chosen to minimize the entropy of
    the leaf nodes.

    Based on this specification, we will not compute
    information gain. Instead, we just compute the entropy
    of the children.

Assumptions:
    (1) The data we will test on is continuous. As such, we
        choose to split the data using entropy. This is
        described in the algorithm, below.
    (2) Assume binary target.
    (3) Integer targets in range [-1, 1].
    (4) When finding the best split, if there are two splits
        resulting in equal information gain, the second is
        chosen. This is arbitrary, and would need adjustment.

Decision Stump Algorithm:
    (1) Sort the targets by their inputs.
    (2) Find the indices where the target changes.
    (3) For each target change index:
    (4)     Compute the midpoint of the index, and its predecessor.
    (5)     Compute the entropy of that split.
"""

Predictable = TypeVar('Predictable', float, Iterable, np.ndarray)


def mid_point(val_one, val_two):
    """
    :param val_one: lower bound
    :param val_two: upper bound
    :return: the mid point of two bounds
    """
    return (val_one*1.0 + val_two*1.0) / 2.0


def tree_log(val):
    """
    Customized log for building decision trees.
    :param val: The value to take the log of.
    :return: If val is 0, 0 is returned. Else, log2(val).
    """
    if val == 0:
        return 0
    else:
        return math.log2(val)


class HomogeneousClassError(Exception):
    """
    Error raised if a dataset has only one class.
    """
    pass


def sort_data(predictors: np.ndarray, targets: np.ndarray) -> Tuple[np.ndarray, np.ndarray]:
    """
    Sort the data - predictors and targets.
    :param predictors:
    :param targets:
    :return: Sorted tuple of (predictors, targets).
    """
    assert predictors.shape[0] == targets.shape[0]
    sorted_indices = np.argsort(predictors)
    return predictors[sorted_indices], targets[sorted_indices]
```

```python
def find_delta_indices(targets: np.ndarray) -> List[int]:
    """
    Find the indices where the values change. For example:
    >>> data = np.array([1, 1, -1, 1, -1])
    >>> print(find_delta_indices(data))
    [2, 4]
    :return: The indices where values change from the previous position.
    """
    indices = []
    for i in range(1, targets.shape[0]):
        if targets[i] != targets[i - 1]:
            indices.append(i)
    return indices


def test_split(data: np.ndarray, index: int) -> Tuple[np.ndarray, np.ndarray]:
    """
    :param data: To split.
    :param index: To split at.
    :return: A tuple of ndarrays split at `index`.
    """
    return (
        data[0:index], data[index:len(data)]
    )


def class_counts(data: np.ndarray) -> Dict:
    """
    :param data: To count.
    :return: A dictionary with counts (values) of array elements (keys).
    """
    counts = {}
    keys, values = np.unique(data, return_counts=True)
    for key, value in zip(keys, values):
        counts[key] = value
    return counts


def majority_class(data: np.ndarray) -> int:
    """
    :param data:
    :return: Majority value in the array.
    """
    classes, counts = np.unique(data, return_counts=True)
    max_index = np.argmax(counts)
    return classes[max_index]


class StumpClassifier:
    _target_range = [-1, 1]

    def __init__(self):
        self._decision_boundary = None
        self._predictors, self._targets = [None] * 2
        self._left_prediction, self._right_prediction = [None] * 2
        self._information = 1.0

    @property
    def decision_boundary(self) -> float:
        return self._decision_boundary

    @property
    def information(self) -> float:
        return self._information

    def fit(self, predictors: np.ndarray, targets: np.ndarray) -> None:
        self._predictors = np.copy(predictors)
        self._targets = np.copy(targets)
        # Data musty be sorted for call to find_delta_indices
        self._predictors, self._targets = sort_data(self._predictors, self._targets)
        # Max gain
        self._find_best_split(self._predictors, self._targets)

    def predict(self, predictors: Predictable) -> np.ndarray:
        # Allow the caller to pass in a single value - results in a one elem array.
        try:
            _ = iter(predictors)
        except TypeError:
            predictors = [predictors]

        return np.array(
```

```
            [self._predict_single(predictor) for predictor in predictors]
        )

    def _predict_single(self, predictor: float) -> int:
        prediction = None
        if predictor <= self.decision_boundary:
            prediction = self._left_prediction
        else:
            prediction = self._right_prediction
        return prediction

    def _find_best_split(self, predictors: np.ndarray, targets: np.ndarray) -> None:
        """
        Find the split that maximizes information gain. This is a trivial linear search.
        See assumptions in header.
        :param predictors:
        :param targets:
        """
        delta_indices = find_delta_indices(targets)
        if len(delta_indices) == 0:
            raise HomogeneousClassError()
        best_index, best_info = -1, -1
        for index in delta_indices:
            left_data, right_data = test_split(targets, index)
            info = self._info(left_data, right_data)
            if info >= best_info:
                best_index = index
                best_info = info
        self._set_model_params(best_index, best_info)

    def _info(self, left_data: np.ndarray, right_data: np.ndarray) -> float:
        """
        :return: Information gain at the parent level of left_data, right_data.
        """
        total = len(self._targets)
        left_len, right_len = len(left_data), len(right_data)
        left_p, right_p = left_len / total, right_len / total
        parent_entropy = self._entropy(self._targets)
        return (
                parent_entropy - (left_p * self._entropy(left_data) + right_p * self._entropy(right_data))
        )

    def _entropy(self, data: np.ndarray) -> float:
        """
        Entropy at a given 'node'.
        """
        sigma = 0
        total = len(data)
        for target, target_count in class_counts(data).items():
            p = target_count / total
            sigma += -(p * tree_log(p))
        return sigma

    def _set_model_params(self, index: int, info_gain: float) -> None:
        self._decision_boundary = mid_point(self._predictors[index - 1], self._predictors[index])
        self._information = info_gain
        left_data, right_data = test_split(self._targets, index)
        self._left_prediction = majority_class(left_data)
        self._right_prediction = majority_class(right_data)

    def __repr__(self):
        def stringify_array(a):
            return [str(x) for x in a]

        return 'decision_boundary: {}\nPred: {}\nTarg: {}'.format(
            self.decision_boundary,
            '|'.join(stringify_array(self._predictors)),
            '|'.join(stringify_array(self._targets))
        )


class AdaBoost:
    def __init__(self, boosting_rounds=10):
        self._predictors, self._targets, self._sample_indices = [None] * 3
        self.boosting_rounds = boosting_rounds
        self.ensemble = []
        self.alphas = []

    @staticmethod
    def uniform_probability_list(n_samples):
        """
        Utility method to return a list of uniform probabilities.
```

```python
        """
        sample_weight = 1 / n_samples
        return np.array([sample_weight] * n_samples)

    def fit(self, predictors: np.ndarray, targets: np.ndarray, verbose = True) -> None:
        # Save data into the object.
        self._initialize_data(predictors, targets)
        # Get uniform weights.
        sample_weights = AdaBoost.uniform_probability_list(len(self._targets))
        boosting_round = 0
        # Iterate over the boosting rounds.
        while boosting_round < self.boosting_rounds:
            # Sample the data, using the weights.
            sample_predictors, sample_targets = self._get_sample(sample_weights)
            stump = StumpClassifier()
            # Train the classifier. Iterate back without moving on to the next round if the
            # targets chosen are Homogeneous.
            try:
                stump.fit(sample_predictors, sample_targets)
            except HomogeneousClassError:
                sample_weights = AdaBoost.uniform_probability_list(len(self._targets))
                continue
            # Compute the error.
            predictions = stump.predict(self._predictors)
            misclassed = self._misclassed_predictions(predictions)
            weighted_error = self._weighted_error(misclassed, sample_weights)
            # If the error exceeds tolerance, iterate back without moving on to the next round.
            if weighted_error >= .5:
                sample_weights = self.uniform_probability_list(len(self._targets))
                continue
            # Else, we add this to the ensemble.
            else:
                boosting_round += 1
                alpha = .5 * math.log((1 - weighted_error) / weighted_error)
                self._add_model(stump, alpha)
                if verbose:
                    def print_div():
                        print('----------------------------------------')

                    def stringify_array(a):
                        return [str(x) for x in a]
                    print_div()
                    print(
                        'Alpha: {}\nError: {}'.format(
                            alpha, weighted_error
                        )
                    )
                    print('Weights:')
                    print(
                        '|'.join(stringify_array(sample_weights))
                    )
                    print(stump)
                    print_div()
                sample_weights = self._update_weights(sample_weights, misclassed, alpha)

    def predict(self, values):
        try:
            _ = iter(values)
        except TypeError:
            values = [values]
        predictions = []
        for value in values:
            predictions.append(self._majority_vote(value))
        return np.array(predictions)

    def _majority_vote(self, value):
        sigma = 0
        for alpha, model in zip(self.alphas, self.ensemble):
            sigma += (alpha * model.predict(value)[0])
        if sigma < 0:
            return -1
        else:
            return 1

    def _initialize_data(self, predictors: np.ndarray, targets: np.ndarray) -> None:
        self._predictors = np.copy(predictors)
        self._targets = np.copy(targets)
        self._sample_indices = list(range(len(targets)))

    def _get_sample(self, probabilities: np.ndarray) -> Tuple[np.ndarray, np.ndarray]:
        random_indices = np.random.choice(self._sample_indices, size=len(self._targets), replace=True, p=probabilities)
        return self._predictors[random_indices], self._targets[random_indices]
```

```
    def _misclassed_predictions(self, predictions):
        return self._targets != predictions

    @staticmethod
    def _weighted_error(misclassed_selectors: np.ndarray, sample_weights: np.ndarray):
        misclassed_bitmap = misclassed_selectors.astype(np.int)
        return np.dot(sample_weights, misclassed_bitmap)

    def _add_model(self, model, alpha):
        self.ensemble.append(model)
        self.alphas.append(alpha)

    def _update_weights(self, weights, misclassed, alpha):
        # This can (and likely should) be done with a dot product.
        # However, that introduces various annoyances.
        new_weights = []
        for is_misclassed, weight in zip(misclassed, weights):
            if is_misclassed:
                a_exp = -alpha
            else:
                a_exp = alpha
            new_weights.append(weight * math.exp(a_exp))
        new_weights = np.array(new_weights)
        new_weights /= new_weights.sum()
        return new_weights


if __name__ == '__main__':
    predictors = np.array([.5, 3.0, 4.5, 4.6, 4.9, 5.2, 5.3, 5.5, 7.0, 9.5])
    targets = np.array([-1, -1, 1, 1, 1, -1, -1, 1, -1, -1])
    classifier = AdaBoost(10)
    classifier.fit(predictors, targets, verbose=True)
    # print(classifier.predict(predictors))
    test = np.arange(1, 11) * 1.0
    print('Test Data:')
    print(test)
    print('Test Predictions:')
    print(classifier.predict(test))
```