Project 7: AVL Trees

Due: Thursday, March 31st @ 10:00pm ET

This is not a team project. Do not copy someone else's work.

Assignment Overview

AVL trees are a self-balancing binary search tree (BST) optimized to maintain logarithmic-time operations regardless of the order in which data is inserted and deleted. First introduced by Soviet computer scientists Georgy Adelson-Velsky and Evgenii Landis in their 1962 paper "An algorithm for the organization of information," AVL trees have stood the test of time and remain a popular choice when a space-efficient data structure supporting fast insertion/search/deletion is necessary.

To motivate AVL trees, it is worth considering a common problem that arises in traditional BSTs. BSTs are designed to perform logarithmic-time insertion/search/deletion, but may operate at linear-time if data is inserted or deleted according to certain patterns which cause the BST to become *unbalanced*. For example, when data is inserted into a traditional BST in sorted (or reverse-sorted) order, the BST will grow leaves in a single direction and effectively turn into a linked list.

If our dataset is small, this may not be a problem—but when we're working with thousands or millions of records in a database, the difference between logarithmic and linear is astounding!

AVL trees improve upon traditional BSTs by *self-balancing* in order to guarantee logarithmic-time operations. In this project, you will be implementing a traditional BST, then an AVL tree from scratch in Python, using the latter to solve a machine learning-inspired application problem. For more information on BSTs and AVL Trees, check out Week 7-8 Content on D2L.

Assignment Notes

1. In this project, you'll be using Python `Generator` objects to traverse a tree in a space-efficient manner. Unlike a traversal returned in the form `List[Node]` using *O(n)* space, a traversal returning a generator will use *O(1)* space by *yielding* each `Node` in a sequential, on-demand manner. See this link for a nice introduction to `Generator`s in Python! You can also look back at the March 1st Live Class Activity and Week 7-8 Yield materials in D2L.

2. One of the most common errors in this project is forgetting or incorrectly updating the height and rebalancing within functions that change the tree structure (insert and remove). Read the notes we put under the function description in the specs carefully and think about how you can use recursion/call stack to help you rebalance the tree. What is the call stack's relationship to the node which you removed/inserted?

3. AVL Trees are more complicated structures than what you have worked with before but if you boil each function down to the different cases within them, then it begins to look a lot simpler. Try to decompose each function into what checks/cases you need to look for before an operation. Checks like is the node I'm removing/inserting the origin? Is there a right node before I make a call on node.right? Am I updating the correct pointers?

4. The debugger is your friend! Do not be scared to use it, it is worth the extra time to learn its functionality if you haven't yet. Use it to determine if what you think your code is doing, is actually what it's doing! It's the most helpful tool when trying to figure out why your more complex functions aren't working.

5. We have provided visualization and printing functions for when you just want to get a birds eye view of your tree and don't want to have to click through several levels of objects in the debugger. Using the visualization functions is as simple as calling `tree.visualize()` on an object of type BinarySearch, AVLTree or NearestNeighborClassifier. We are, of course, human, and cannot guarantee these visualizations are 100% bug free; if you encounter issues, let us know.

6. Using global variables (with the nonlocal keyword) will result in a 20 point flat-rate deduction.

7. Changing function signatures will result in a 2 point deduction for each violation, up to a maximum of 20 points.

8. Note that the amount of duplicate code this project could be reduced by the use of inheritance. In particular, the code for search and insert within the BinarySearchTree and AVLTree classes will look quite similar in this project, and could be reused if we had made the AVLTree class a subclass of the BinarySearchTree class. We intentionally avoided bringing inheritance into this project to keep the syntax lightweight and to avoid adding a layer of object-oriented complexity, but it is a useful thought exercise to consider how one could implement the AVLTree class as a subclass of the BinarySearchTree class.

9. If you run the solution.py file after implementing all BinarySearchTree and AVLTree functions, you will be greeted by a performance comparison between the two trees!

Assignment Specifications

class Node:

Implements a tree node used in the BinarySearchTree and AVLTree classes.

DO NOT MODIFY the following attributes/functions

- **Attributes**
  - **value: T:** Value held by the `Node`. Note that this may be any type, such as a `str`, `int`, `float`, `dict`, or a more complex object.
  - **parent: Node:** Reference to this `Node`'s parent `Node` (may be `None`).
  - **left: Node:** Reference to this `Node`'s left child `Node` (may be `None`).
  - **right: Node:** Reference to this `Node`'s right child `Node` (may be `None`).
  - **height: int:** Number of levels of `Node`s below (the height of a leaf `Node` is 0).
- **__init__(self, value: T, parent: Node = None, left: Node = None, right: Node = None) -> None**
  - Constructs an AVL Tree node.

- **value: T:** Value held by the `Node`.
- **parent: Node:** Reference to this `Node`'s parent `Node` (may be `None`).
- **left: Node:** Reference to this `Node`'s left child `Node` (may be `None`).
- **right: Node:** Reference to this `Node`'s right child `Node` (may be `None`).
- **Returns:** `None`.

- **__str__(self) -> str** and **__repr__(self) -> str**
  - Represents the `Node` as a string in the form `<value_held_by_node>`. Thus, `<7>` indicates a `Node` object holding an `int` value of 7, whereas `<None>` indicates a `Node` object holding a value of `None`.
  - Note that Python will automatically invoke this function when using printing a `Node` to the console, and PyCharm will automatically invoke this function when displaying a `Node` in the debugger.
  - Call this with `str(node)` (rather than `node.__str__()`).
  - **Returns:** `str`.

class BinarySearchTree:

Implements a traditional BST.

DO NOT MODIFY the following attributes/functions

- **Attributes**
  - **origin: Node:** Root node of the entire `BSTree` (may be `None`). This naming convention helps us disambiguate between when we are referring to the root of the entire `BSTree` and the root of a subtree within the `BSTree`. In fact, any given `Node` object within an `BSTree` can be thought of as being the root of the subtree of all `Node`s below—and `origin` is the uppermost such root in our tree.
  - **size: int:** Number of nodes in the `BSTree`.
- **__init__(self) -> None**
  - Construct an empty `BSTree`. Initialize the `origin` to `None` and set the size to zero.
  - **Returns:** `None`.
- **__str__(self) -> str** and **__repr__(self) -> str**
  - Returns a pretty printed string representation of the binary tree. Each node will be of the form `{value},h={height},⬆{parent.value}`
  - Note that Python will automatically invoke this function when using printing a `Node` to the console, and PyCharm will automatically invoke this function when displaying a `Node` in the debugger.
  - Call this with `str(node)` (rather than `node.__str__()`).
  - **Returns:** `str`.
- **visualize(self, filename="bst_visualization.svg") -> str**
  - Generates an svg image file of the binary tree.
  - filename: str: The filename for the generated svg file. Should end with .svg. Defaults to bst_visualization.svg
  - **Returns:** The svg string.

IMPLEMENT the following functions

- **height(self, root: Node) -> int**

  - Return height of a subtree in the BSTree , properly handling the case of `root = None`. Recall that the height of an empty subtree is -1.
  - *Time / Space: O(1) / O(1)*.
  - **root: Node:** The root `Node` of the subtree being measured.
  - **Returns:** Height of the subtree at `root`, i.e., the number of levels of `Node`s below this `Node`. The height of a leaf `Node` is 0, and the height of a `None`-type is -1.

- **insert(self, root: Node, val: T) -> None**

  - Insert a node with `val` into the subtree rooted at `root`, returning the root node of the balanced subtree after insertion.
  - If `val` already exists in the tree, do nothing.
  - Should update `size` and `origin` attributes of the object if necessary and correctly set parent/child pointers when inserting a new `Node`
  - Easiest to implement recursively.
  - *Time / Space: O(h) / O(1)*, where *h* is the height of the tree.
  - **root: Node:** The root `Node` of the subtree in which to insert `val`.
  - **val: T:** The value to be inserted in the subtree rooted at `root`.
  - **Returns:** None

- **remove(self, root: Node, val: T) -> Node**

  - Remove the node with value `val` from the subtree rooted at `root`, and return the root of the subtree following removal.
  - If `val` does not exist in the BST tree, do nothing.
  - Should update `size` and `origin` attributes of the object and correctly update parent/child pointers of `Node` objects as necessary.
  - Should update the `height` attribute on all `Node` objects affected by the removal (ancestor nodes directly above on path to origin).
  - Recall that there are three distinct cases of BST removal to consider.
  - Easiest to implement recursively.
  - If the node being removed has two children, swap the value of this node with its **predecessor** and recursively remove this predecessor node (which contains the value to be removed after swapping and is guaranteed to be a leaf).
    - Although one technically *could* swap values with the successor node in a two-child removal, our testcases assume you will swap with the predecessor.
  - *Time / Space: O(h) / O(1)*, where *h* is the height of the tree.
  - **root: Node:** The root `Node` of the subtree from which to delete `val`.
  - **val: T:** The value to be deleted from the subtree rooted at `root`.
  - **Returns:** Root of new subtree after removal (could be the original root).

- **search(self, root: Node, val: T) -> Node**

  - Find and return the `Node` with the value `val` in the subtree rooted at `root`.
  - If `val` does not exist in the subtree rooted at `root`, return the `Node` below which `val` would be inserted as a child. For example, on a balanced 1-2-3 tree (with 2 on top and 1, 3 as children), `search(node_2, 0)` would return `node_1` since the value of 0 would be inserted as a left child of `node_1`.

  - Easiest to implement recursively.
  - *Time / Space: O(h) / O(1)*, where *h* is the height of the tree.
  - **root: Node:** The root `Node` of the subtree in which to search for `val`.
  - **val: T:** The value being searched in the subtree rooted at `root`.
  - **Returns:** `Node` object containing `val` if it exists, else the `Node` object below which `val` would be inserted as a child.

class AVLTree:

Implements a self-balancing BST for faster operation.

DO NOT MODIFY the following attributes/functions

- **Attributes**
  - **origin: Node:** Root node of the entire `AVLTree` (may be `None`). This naming convention helps us disambiguate between when we are referring to the root of the entire `AVLTree` and the root of a subtree within the `AVLTree`. In fact, any given `Node` object within an `AVLTree` can be thought of as being the root of the subtree of all `Node`s below—and `origin` is the uppermost such root in our tree.
  - **size: int:** Number of nodes in the `AVLTree`.
- **__init__(self) -> None**
  - Construct an empty `AVLTree`. Initialize the `origin` to `None` and set the size to zero.
  - **Returns:** `None`.
- **__str__(self) -> str** and **__repr__(self) -> str**
  - Returns a pretty printed string representation of the binary tree. Each node will be of the form `{value},h={height},⬆{parent.value}`
  - Note that Python will automatically invoke this function when using printing a `Node` to the console, and PyCharm will automatically invoke this function when displaying a `Node` in the debugger.
  - Call this with `str(node)` (rather than `node.__str__()`).
  - **Returns:** `str`.
- **visualize(self, filename="avl_tree_visualization.svg") -> str**
  - Generates an svg image file of the binary tree.
  - filename: str: The filename for the generated svg file. Should end with .svg. Defaults to avl_tree_visualization.svg
  - **Returns:** The svg string.
- **left_rotate(self, root: Node) -> Node**
  - Perform a left rotation on the subtree rooted at `root`. Return root of new subtree after rotation.
  - This method is already implemented for you.
  - *Time / Space: O(1) / O(1).*
  - **root: Node:** The root `Node` of the subtree being rotated.
  - **Returns:** Root of new subtree after rotation.

- **remove(self, root: Node, val: T) -> Node**
    - This method is already implemented for you.
    - Removes the node with value `val` from the subtree rooted at `root`, and returns the root of the balanced subtree following removal.
    - If `val` does not exist in the AVL tree, do nothing.
    - Updates `size` and `origin` attributes of `AVLTree` and correctly update parent/child pointers of `Node` objects as necessary.
    - Updates the `height` attribute and call `rebalance` on all `Node` objects affected by the removal (ancestor nodes directly above on path to origin).
    - Note that that there are three distinct cases of BST removal to consider.
    - Implemented recursively.
    - If the node being removed has two children, swaps the value of this node with its **predecessor** and recursively removes this predecessor node (which contains the value to be removed after swapping and is guaranteed to be a leaf).
        - Although one technically *could* swap values with the successor node in a two-child removal, we choose to swap with the predecessor.
    - *Time / Space: O(log n) / O(1).*
    - **root: Node:** The root `Node` of the subtree from which to delete `val`.
    - **val: T:** The value to be deleted from the subtree rooted at `root`.
    - **Returns:** Root of new subtree after removal and rebalancing (could be the original root).

IMPLEMENT the following functions

- **height(self, root: Node) -> int**
    - Return height of a subtree in the AVL tree, properly handling the case of `root = None`. Recall that the height of an empty subtree is -1.
    - *Time / Space: O(1) / O(1).*
    - **root: Node:** The root `Node` of the subtree being measured.
    - **Returns:** Height of the subtree at `root`, i.e., the number of levels of `Node`s below this `Node`. The height of a leaf `Node` is 0, and the height of a `None`-type is -1.

- **right_rotate(self, root: Node) -> Node**
    - Perform a right rotation on the subtree rooted at `root`. Return root of new subtree after rotation.
    - This should be nearly identical to `left_rotate`, with only a few lines differing. Team 331 agreed that giving one rotation helps ease the burden of this project on the heels of Exam 2—but writing the other rotation will be a good learning experience!
    - *Time / Space: O(1) / O(1).*
    - **root: Node:** The root `Node` of the subtree being rotated.
    - **Returns:** Root of new subtree after rotation.

- **balance_factor(self, root: Node) -> int**
    - Compute the balance factor of the subtree rooted at `root`.
    - Recall that the balance factor is defined to be `h_L - h_R` where `h_L` is the height of the left subtree beneath this `Node` and `h_R` is the height of the right subtree beneath this `Node`.

- Note that in a properly-balanced AVL tree, the balance factor of all nodes in the tree will be in the set {-1, 0, +1}, as rebalancing will be triggered when a node's balance factor becomes -2 or +2.
- The balance factor of an empty subtree (`None`-type `root`) is 0.
- To stay within time complexity, keep the `height` attribute of each `Node` object updated on all insertions/deletions/rebalances, then use `h_L = left.height` and `h_R = right.height`.
- *Time / Space: O(1) / O(1).*
- **root: Node:** The root `Node` of the subtree on which to compute the balance factor.
- **Returns:** `int` representing the balance factor of `root`.

- **rebalance(self, root: Node) -> Node**
  - Rebalance the subtree rooted at `root` (if necessary) and return the new root of the resulting subtree.
  - Recall that rebalancing is only necessary at this `root` if the balance factor `b` of this `root` satisfies `b >= 2 or b <= -2`.
  - Recall that there are four types of imbalances possible in an AVL tree, and that each requires a different sequence of rotation(s) to be called.
  - *Time / Space: O(1) / O(1).*
  - **root: Node:** The root `Node` of the subtree to be rebalanced.
  - **Returns:** Root of new subtree after rebalancing (could be the original root).

- **insert(self, root: Node, val: T) -> Node**
  - Insert a node with `val` into the subtree rooted at `root`, returning the root node of the balanced subtree after insertion.
  - If `val` already exists in the AVL tree, do nothing.
  - Should update `size` and `origin` attributes of `AVLTree` if necessary and correctly set parent/child pointers when inserting a new `Node`
  - Should update the `height` attribute and call `rebalance` on all `Node` objects affected by the insertion (ancestor nodes directly above on path to origin).
  - Easiest to implement recursively.
  - *Time / Space: O(log n) / O(1).*
  - **root: Node:** The root `Node` of the subtree in which to insert `val`.
  - **val: T:** The value to be inserted in the subtree rooted at `root`.
  - **Returns:** Root of new subtree after insertion and rebalancing (could be the original root).

- **min(self, root: Node) -> Node**
  - Find and return the `Node` with the smallest value in the subtree rooted at `root`.
  - Easiest to implement recursively.
  - *Time / Space: O(log n) / O(1).*
  - **root: Node:** The root `Node` of the subtree in which to search for a minimum.
  - **Returns:** `Node` object containing the smallest value in the subtree rooted at `root`.

- **max(self, root: Node) -> Node**
  - Find and return the `Node` with the largest value in the subtree rooted at `root`.
  - Easiest to implement recursively.

  - *Time / Space: O(log n) / O(1)*.
  - **root: Node:** The root `Node` of the subtree in which to search for a maximum.
  - **Returns:** `Node` object containing the largest value in the subtree rooted at `root`.

- **search(self, root: Node, val: T) -> Node**

  - Find and return the `Node` with the value `val` in the subtree rooted at `root`.
  - If `val` does not exist in the subtree rooted at `root`, return the `Node` below which `val` would be inserted as a child. For example, on a balanced 1-2-3 tree (with 2 on top and 1, 3 as children), `search(node_2, 0)` would return `node_1` since the value of 0 would be inserted as a left child of `node_1`.
  - Easiest to implement recursively.
  - *Time / Space: O(log n) / O(1)*.
  - **root: Node:** The root `Node` of the subtree in which to search for `val`.
  - **val: T:** The value being searched in the subtree rooted at `root`.
  - **Returns:** `Node` object containing `val` if it exists, else the `Node` object below which `val` would be inserted as a child.

- **inorder(self, root: Node) -> Generator[Node, None, None]**

  - Perform an inorder (left, current, right) traversal of the subtree rooted at `root` using a Python generator.
  - Use `yield` to immediately generate an element in your function, and `yield from` to generate an element from a recursive function call.
  - Do not yield (generate) `None`-types.
  - **To pass the test case for this function, you must also make the AVLTree class iterable such that doing `for node in avltree` iterates over the inorder traversal of the tree**
  - *Time / Space: O(n) / O(1)*.
    - Although we will traverse the entire tree and hence incur O(n) time, our use of a generator will keep us at constant space complexity since elements are yielded one at a time! This is a key advantage of returning a generator instead of a list.
  - **root: Node:** The root `Node` of the subtree currently being traversed.
  - **Returns:** `Generator` object which yields `Node` objects only (no `None`-type yields). Once all nodes of the tree have been yielded, a `StopIteration` exception is raised.

- **__iter__(self) -> Generator[Node, None, None]**

  - Implementing this "dunder" method allows you to use an AVL tree class object anywhere you can use an iterable, e.g., inside of a `for node in tree` expression. We want the iteration to use the inorder traversal of the tree so this should be implemented such that it returns the inorder traversal.
  - **This function should only be one line, calling another function you have implemented.**
  - **Returns:** A generator that iterates over the inorder traversal of the tree

- **preorder(self, root: Node) -> Generator[Node, None, None]**

  - Perform a preorder (current, left, right) traversal of the subtree rooted at `root` using a Python generator.

  - Use `yield` to immediately generate an element in your function, and `yield from` to generate an element from a recursive function call.
  - Do not yield (generate) `None`-types.
  - *Time / Space: O(n) / O(1).*
    - Although we will traverse the entire tree and hence incur O(n) time, our use of a generator will keep us at constant space complexity since elements are yielded one at a time! This is a key advantage of returning a generator instead of a list.
  - **root: Node:** The root `Node` of the subtree currently being traversed.
  - **Returns:** `Generator` object which yields `Node` objects only (no `None`-type yields). Once all nodes of the tree have been yielded, a `StopIteration` exception is raised.

- **postorder(self, root: Node) -> Generator[Node, None, None]**
  - Perform a postorder (left, right, current) traversal of the subtree rooted at `root` using a Python generator.
  - Use `yield` to immediately generate an element in your function, and `yield from` to generate an element from a recursive function call.
  - Do not yield (generate) `None`-types.
  - *Time / Space: O(n) / O(1).*
    - Although we will traverse the entire tree and hence incur O(n) time, our use of a generator will keep us at constant space complexity since elements are yielded one at a time! This is a key advantage of returning a generator instead of a list.
  - **root: Node:** The root `Node` of the subtree currently being traversed.
  - **Returns:** `Generator` object which yields `Node` objects only (no `None`-type yields). Once all nodes of the tree have been yielded, a `StopIteration` exception is raised.

- **levelorder(self, root: Node) -> Generator[Node, None, None]**
  - Perform a level-order (breadth-first) traversal of the subtree rooted at `root` using a Python generator.
  - Use the builtin `queue.SimpleQueue` class to maintain your queue of children throughout the course of the traversal—see the official documentation here.
  - Use `yield` to immediately generate an element in your function, and `yield from` to generate an element from a recursive function call.
  - Do not yield (generate) `None`-types.
  - *Time / Space: O(n) / O(n).*
    - We will traverse the entire tree and incur O(n) time. In addition, the queue we must use for an inorder traversal will grow to size n/2 = O(n) just before beginning the final level of leaf nodes in the case of a perfect binary tree.
  - **root: Node:** The root `Node` of the subtree currently being traversed.
  - **Returns:** `Generator` object which yields `Node` objects only (no `None`-type yields). Once all nodes of the tree have been yielded, a `StopIteration` exception is raised.

Application: *k*-Nearest Neighbors

You are a member of the development team for Avida-ED: a truly fascinating program developed at MSU that lets you simulate the process of evolution (natural selection, mutations, genetic drift and the like). You recently came up with the *ground-breaking* idea of integrating *tree*-dwellers - a digital organism that lives in binary trees - into Avida.

So, you get to work and get a prototype up and running. To test your prototype, you decided to go ahead and run your simulation overnight. Unfortunately for you, there was a memory leak in your program, and it crashed your entire computer (That's what you get for using C++ over Rust). Luckily you were able to salvage some of the results and saw that you were indeed able to produce tree dwellers that evolved, but some things were missing - including the timestamps for when each one evolved. You initially try to recover the data manually but see that there is too much. But, there is hope: it seems that the timestamps can be roughly inferred from the properties which were not lost.

So, you decide to make full use of the labels that you generated manually by using them to train a *k*-Nearest Neighbor classifier.

Background Information

Classification refers to the task of categorizing data into one of several buckets. For example, the task of deciding whether an email is "spam" or "not spam" can be viewed as a classification problem. Traditionally, classification problems were solved by using conditional statements to create *rule-based* classifiers—but such rule-based classifiers are difficult to implement, maintain, and tune.

As such, machine learning based classifiers have gained popularity. In contrast to rule-based classifiers, machine learning-based classifiers figure out how to classify new data on their own by learning the patterns embedded in previously-observed, labeled data.

A *k*-Nearest Neighbor classifier is one type of machine learning-based classifier that compares a new data point to its closest *neighbors* with known class labels. In this application problem, you'll be implementing two functions of a custom `NearestNeighborClassifier` class.

Problem Description

Given a *training* dataset of `(x, y)` pairs and *testing* dataset of `x` values, your `NearestNeighborClassifier` will:

1. `fit()` a one-dimensional *k*-Nearest Neighbor classifier to the *training* data, and
2. `predict()` the class labels of *testing* data based on the patterns observed in the *training* data.

More specifically, the training dataset will be a `List[tuple(float, str)]` of `(x: float, y: str)` tuples, where `x` can be thought of as a *feature* and `y` can be thought of as a *target label*. The goal of `fit()` is for the `NearestNeighborClassifier` to *learn* the associations between features `x` and target labels `y`, and the goal of `predict()` is for the `NearestNeighborClassifier` to predict the unknown labels of features `x`.

For example, suppose we have a dataset of `(temperature, season)` pairs in which the `x: float` value is the daily high temperature on a scale of 0 to 1 and the `y: str` value indicates the season in which the temperature was recorded, with possible values `{"spring", "summer", "fall", "winter"}`. Then `fit()` will learn to associate lower temperatures with winter, higher temperatures with summer, and mid-range temperatures with spring and fall Likewise, `predict()` will take in a temperature `x` and output the most likely season `y` in which the temperature was observed.

The specifications below provide further detail.

- **fit(self, data: List[Tuple[float, str]]) -> None**
  - Fits the one-dimensional `NearestNeighborClassifier` to `data` by:
    - Rounding each `x` value to the number of digits specified by `self.resolution` to obtain a `key`,
    - Searching for the `Node` in `self.tree` with this `key`, which is guaranteed to exist by construction of `self.tree` in `self.__init__()`.
    - Accessing the `AVLWrappedDictionary` object stored in `node.value`, and
    - Updating the `dictionary` of this `node.value` by incrementing the number of times class `y` ended up at this node, i.e., the number of times class `y` was associated to `key`.
  - *Time / Space: O(n log n) / O(n)*
    - Searching for a node is *O(log n)* and must be completed *O(n)* times to save *O(n)* `dictionary` entries.
  - **data: List[Tuple[float, str]]:** A list of `(x: float, y: str)` pairs associating *feature* `x` values in the range `[0, 1]` to *target* `y` values. Provides the information necessary for our classifier to *learn*.
  - **Returns:** None.
- **predict(self, x: float, delta: float) -> str**
  - Predicts the class label of a single `x` value by:
    - Rounding `x` to the number of digits specified by `self.resolution` to obtain a `key`,
    - Searching for all `Node` objects in `self.tree` whose key is within `± delta` of this `key`,
    - Accessing the `AVLWrappedDictionary` object stored in all such `node.value`s, and
    - Taking the most common `y` label across all `dictionaries` stored in these `node.value`s.
  - Note that this process effectively predicts the class `y` based on the most common `y` observed in training data close to this `x`
  - If no data in `self.tree` from the training set has a key within `key ± delta`, return `None`.
  - *Time / Space: O(k log n) / O(1)*
    - Here, *k* = *(delta*10**resolution + 1)* is the number of neighbors being searched in `self.tree`, and each search is an *O(log n)* operation.
  - **x: float:** Feature value in range `[0, 1]` with unknown class to be predicted.
  - **delta: float:** Width of interval to search across for neighbors of `x`.
  - **Returns:** `str` of the predicted class label `y`

To implement `fit()` and `predict()`, you'll need to understand how the `NearestNeighborClassifier` is constructed in its provided `__init__()` function. Additionally, you'll need to know how the `AVLWrappedDictionary` class works!

class NearestNeighborClassifier

DO NOT MODIFY the following attributes/functions

- **Attributes**
  - **tree: AVLTree:** Structure storing labeled training data, filled in `fit()` and used in `predict()`.
  - **resolution: int**: Number of decimal places to which feature `x` values are rounded.
- **__init__(self, resolution: int) -> None**
  - Initialize the `NearestNeighborClassifier` with `10**resolution + 1` nodes in `self.tree`, each storing a value of type `AVLWrappedDictionary`.

    - For example, if `resolution == 1`, then 11 nodes will be created with keys `0, 0.1, 0.2 ..., 1.0`, and if `resolution == 2`, then 101 nodes will be created with keys `0, 0.01, 0.02, ..., 1.0`.
  - **resolution: int:** Number of decimal places to which feature `x` values are rounded.
  - **Returns:** None.
- **__str__(self) -> str** and **__repr__(self) -> str**
  - Represents the `NearestNeighborClassifier` as a string.
  - **Returns:** `str`.
- **visualize(self, filename="nnc_visualization.svg") -> None**
  - Generates an svg image file of the nnc tree.
  - filename: str: The filename for the generated svg file. Should end with .svg. Defaults to nnc_visualization.svg
  - **Returns:** The svg string.

IMPLEMENT the following functions

- **fit(self, data: List[Tuple[float, str]]) -> None**
  - See above.
- **predict(self, x: float, delta: float) -> str**
  - See above.

class AVLWrappedDictionary:

DO NOT MODIFY the following attributes/functions

- **Attributes**
  - **key: float:** Floating point key which this `AVLWrappedDictionary` can be searched / looked up by.
  - **dictionary: dict** Price of a stock on a given date.
- **__init__(self, key: float) -> None**
  - Construct a `AVLWrappedDictionary`, a wrapper object which allows one to store dictionaries inside of a `Node` in an `AVLTree`, ordered and searchable by `key`.
  - **key: float:** Floating point key which this `AVLWrappedDictionary` can be searched / looked up by.
  - **Returns:** None.
- **__str__(self) -> str** and **__repr__(self) -> str**
  - Represents the `AVLWrappedDictionary` as a string.
  - **Returns:** `str`.
- **__eq__(self, other: AVLWrappedDictionary) -> bool**
- **__gt__(self, other: AVLWrappedDictionary) -> bool**
- **__lt__(self, other: AVLWrappedDictionary) -> bool**
  - Compare two `AVLWrappedDictionary`s to one another by `key`, returning the appropriate boolean value.
  - **other: AVLWrappedDictionary** Other object to compare to.
  - **Returns:** `bool` indicating whether this object is equal to, greater than, or less than `other`, respectively.

Example

Suppose we have a dataset of images, and wish to classify them by whether they were taken during the day or at night. We may directly measure the brightness `x` of each image on a scale of `[0, 1]`, and it stands to reason that `x` may be a useful predictor of the class label `y` from `{"day", "night"}`, since bright images are more likely to be taken during the day and vice-versa for dark images.

Given the labeled dataset

```
data = [(0.18, "night"), (0.21, "night"), (0.29, "night"),
        (0.49, "night"), (0.51, "day"), (0.53, "day"),
        (0.97, "day"), (0.98, "day"), (0.99, "day")]
```

with a `NearestNeighborClassifier` whose `resolution` is 1, the `tree` of our classifier would look as follows after a call to `fit(data)`:

Now, suppose we are given the following brightness observations from unlabeled images

```
test_images = [0.1, 0.2, 0.5, 0.8, 0.9]
```

and asked to predict the class label `y` from `{"day", "night"}` for each `x in test_images` with `delta = 0.1`. Then:

- `predict(x=0.1, delta=0.1)` would look at the total number of `"day"` and `"night"` instances in the nodes with keys `{0, 0.1, 0.2}`, finding 2 `"night"` instances and 0 `"day"` instances. Thus, `"night"` would be predicted.
- `predict(x=0.2, delta=0.1)` would look at the total number of `"day"` and `"night"` instances in the nodes with keys `{0.1, 0.2, 0.3}`, finding 3 `"night"` instances and 0 `"day"` instances. Thus, `"night"` would be predicted.
- `predict(x=0.5, delta=0.1)` would look at the total number of `"day"` and `"night"` instances in the nodes with keys `{0.4, 0.5, 0.6}`, finding 1 `"night"` instance and 2 `"day"` instances. Thus, `"day"` would be predicted.
- `predict(x=0.8, delta=0.1)` would look at the total number of `"day"` and `"night"` instances in the nodes with keys `{0.7, 0.8, 0.9}`, finding 0 `"night"` instance and 0 `"day"` instances. Thus, `None` would be predicted.
- `predict(x=0.9, delta=0.1)` would look at the total number of `"day"` and `"night"` instances in the nodes with keys `{0.8, 0.9, 1}`, finding 0 `"night"` instance and 3 `"day"` instances. Thus, `"day"` would be predicted.

To summarize, the following code

```
nnc = NearestNeighborClassifier(resolution=1)
nnc.fit(data)
predictions = nnc.predict(test_images)
```

would give

```
predictions = ["night", "night", "day", None, "day"]
```

More cases and precise input/output syntax are provided in the testcases.

Note that the testcases include a `plot` flag variable to enable (optional) visualization of the training and testing data using `numpy` and `matplotlib`. An example plot visualizing the training and testing data for test 1b is shown below. The **visualize function** on the NNC class could also potentially be useful.

Submission

Deliverables

Be sure to upload the following deliverables in a .zip folder to Mimir by 10:00p ET on Thursday, March 31st.

```
Project07.zip
    |— Project07/
        |— feedback.xml     (for project feedback)
        |— __init__.py      (for proper Mimir testcase loading)
        |— solution.py      (contains your solution source code)
```

Grading

- Tests (75)
    - Coding Standard: __/3
    - README.xml Validity Check: __/3
    - `BinarySearchTree`: __/9
        - `insert`: __/3
        - `search`: __/3
        - `remove`: __/3
    - `AVLTree`: __/45
        - `right_rotate`: __/1
        - `balance_factor`: __/1
        - `rebalance`: __/8
        - `insert`: __/8
        - `min`: __/2
        - `max`: __/2
        - `search`: __/8
        - `inorder/__iter__`: __/1
        - `preorder`: __/1
        - `postorder`: __/1
        - `levelorder`: __/1
        - `avl_comprehensive`: __/10
    - `NearestNeighborClassifier`: __/16
        - `nnc_basic`: __/8

        - `nnc_comprehensive`: __/8
- Manual (25)
    - Time and space complexity points are **all-or-nothing** for each function. If you fail to meet time **or** space complexity in a given function, you do not receive manual points for that function.
    - Loss of 1 point per missing docstring (max 3 point loss)
    - Loss of 2 points per changed function signature (max 20 point loss)
    - Loss of 20 points (flat-rate) for use of global variables (with the nonlocal keyword)
        - `BinarySearchTree` time & space: __/6
            - `insert`: __/2
            - `search`: __/2
            - `remove`: __/2
        - `AVLTree` time & space: __/15
            - `right_rotate`: __/1
            - `balance_factor`: __/2
            - `rebalance`: __/2
            - `insert`: __/2
            - `min`: __/1
            - `max`: __/1
            - `search`: __/2
            - `inorder`/`__iter__`: __/1
            - `preorder`: __/1
            - `postorder`: __/1
            - `levelorder`: __/1
        - `NearestNeighborClassifier`: __/4
            - `fit`: __/2
            - `predict`: __/2

Appendix

Authors

Project authored by Bank Premsri and Joseph Pallipadan. Adapted from work by Andrew McDonald, Jacob Caurdy and Lukas Richters.

Memes