Pokemon Party Designer

Aditya Mehrotra

April 2021

Brief problem description and project question

This project is based around the famous video game series *Pokemon* (2). These games revolve around a certain feature called "Pokemon battles". These battles are where two "trainers" engage in turn-based combat with a maximum of 6 Pokemon each, where a Pokemon "faints" when their HP (short for hit/health points) becomes 0. The battle ends when one trainer has no more Pokemon left to fight with, which means that the respective trainer loses. Damage is done through certain moves that a trainer can make their Pokemon perform. There are stats which influence the damage dealt by the Pokemon and determine whether they move before the opposing Pokemon in a given turn (Attack, Special attack and Speed respectively). Additionally, when a Pokemon is hit by an attack, the damage dealt is affected by defensive stats (Defense, Special defense, health points or HP for short). Apart from this system, there is also certain matchups which cause Pokemon to take or deal more damage. For example: water type moves will do more damage to fire type Pokemon, but fire type moves do less damage to water type Pokemon (7).

Given the lengthy explanation above, we can see that there's a lot of factors which affect how a Pokemon performs in a battle. This complexity leads to a lot of deep strategical gameplay around team and Pokemon composition. This leads to certain "archetypes" of Pokemon, for example a "tank" would be a Pokemon with high defensive stats which can take a lot of hits. However, given the state of modern Pokemon, composing a party is hard as there exists nearly 800 different pokemon to choose from! This can be very overwhelming, especially for new players. This leads us to our research question:

Can we simplify the process of creating a Pokemon team using a query-based Pokemon recommendation system?

Most Pokemon team creation sites cater to seasoned players who know exactly what they're looking for. Most interfaces simply have a search bar for Pokemon names and nothing more. However, I want to build something a bit more beginner friendly. Rather than typing the name of a Pokemon, you can simply enter a query such as "Fast, Strong and Fire type" and it'll give you a list of options to choose from. This simplifies the process of team construction

heavily as rather than presenting you a huge amount of Pokemon to choose from, some of which you may not even recognize, it streamlines it down to a bearable amount.

Dataset Description

This project uses "The Complete Pokemon Dataset" from kaggle.

https://www.kaggle.com/rounakbanik/pokemon

This dataset is a csv file where each row represents a Pokemon and each column corresponds to a certain piece of information about the Pokemon.

For the purposes of this project, the columns I am using are:

1. attack, defense, speed, hp, sp_attack, sp_defense (for recommending Pokemon in response to user queries and building my trees)
2. pokedex_number (for displaying Pokemon icons in the GUI)
3. name (for displaying the recommended Pokemon on the GUI)
4. type1, type2 (for partitioning Pokemon into their types)

Here is a sample value of the dataset (I have omitted all columns except for the ones that I am using, which I listed above):

Name	speed	attack	sp_attack	hp	sp_defense	defense	type1	type2	pokedex_number
Pikachu	90	55	50	35	50	40	electric		25

Notice that the value in the *type*2 column is blank. This is fine, as sometimes Pokemon don't have a second type, like pikachu.

Additionally, for the purposes of rendering pokemon icons in the GUI, I used all the icons in the generation viii folder of this repository (4):

`https://github.com/PokeAPI/sprites/tree/master/sprites/pokemon`

These icons are png files that consist of pokemon sprites, where the name of the file represents the unique pokedex number of the pokemon that is shown in the png file.

Computational overview

Functionality overview

Before jumping in directly to the explicit details of my project, I will first go over its general functionality.

One of the key functions of this project is a search system that can take in a broad query such as "find fire high speed" and return the user a list of

Pokemon which fit that criteria. This means that the output will consist of fire type Pokemon which have a high speed stat.

These queries for Pokemon must be structured in the following way:

```
find <type> <degree> <stat>
```

Note: please check the appendix for a list of all Pokemon types and stats

Where degree refers to how high or low the stat must be (ex: "find fire high attack" or "find water low speed")

However, upon further inspection, we see something peculiar. Notice that some elements in the query can be thought of as "narrowing down the search space".

Lets say our full search space is the set of ALL Pokemon. For example, take the query "find fire high attack":

> The word "fire" narrows down our search space to solely fire type Pokemon. Which is one of 18 types.
>
> Recall that each Pokemon has 6 stats. However, the word "attack" narrows down the things we need to search through to only the attack of the fire type Pokemon, which is only one of the 6 stats.
>
> "high" doesn't narrow down the search space like "fire" or "attack", but it lets us filter through our current (narrowed) search space for ONLY fire Pokemon with "high" attack.

Therefore, as illustrated above, the desired type and stat from the query serve to "narrow down" the things we need to consider, while the degree ("high", "medium" ...) serves as a a filter which excludes certain Pokemon and includes others. But this is a bit ambiguous, because in order to determine whether a Pokemon's attack is high, medium or low, we need some sort of inequality.

Quantifying degrees of stats

There are other ways to do it, but the approach in this project to quantify what constitutes as a "high attack" or a "low defense" through percentiles. For the purposes of this project, I will only consider 3 possible degrees - "high", "medium" and "low".

This is where we can use statistics. Since we have 6 different stats across our 800+ Pokemon, each stat will have a different distribution. We can quantify what

```
"low <stat>", "medium <stat>", "high <stat>"
```

mean using the 25, 50, and 75 percentiles for the stat across ALL Pokemon respectively. We are using all Pokemon in our percentile calculations because this allows us to have a statistic which generalizes across all Pokemon types.

Here is my approach, given a query with a certain type, a stat and a degree, we filter the set using the following approach:

1. If the degree is "low", return all Pokemon of the type with stat values less than or equal to the 25th percentile of the stat (inclusive)
2. If the degree is "medium", return all Pokemon of the type with stat values in between the 25th and 75th percentiles of the stat (inclusive)
3. If the degree is "high", return all Pokemon of the type with stat values greater than or equal to the 75th percentile of the stat.

For example, given the query "find dragon medium hp", this query should return all dragon type pokemon that have an hp stat that is between the 25th percentile and 75th percentile (inclusive) of the hp stat across ALL pokemon.

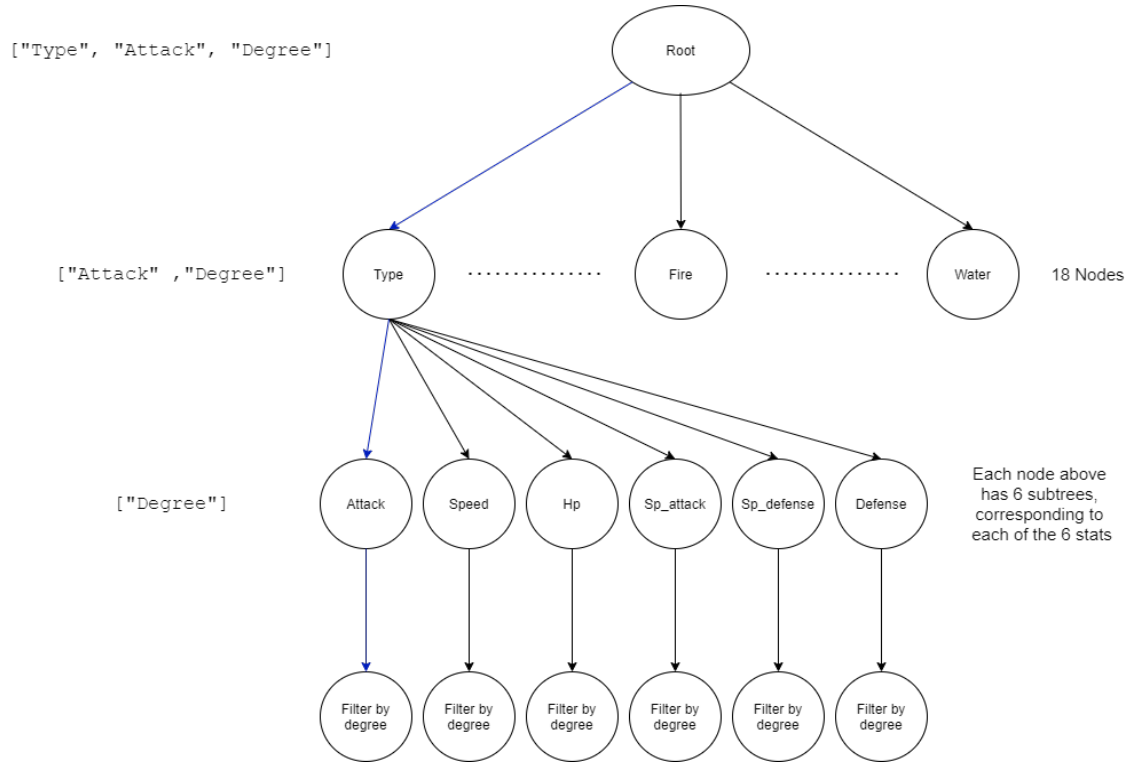
Therefore, for each of the 6 stats, each degree ("high", "medium", "low") introduces a specific numerical restriction for each stat. This effectively quantifies what it means for a Pokemon to have a "high", "low" or "medium" value of a stat.

Tree-like query search structure

Therefore, now that we have a way of quantifying the degrees of stats, we can complete the description of the query processing model. Here is the step by step description:

```
Given a query in the form: "find <type> <degree> <stat>" and a
search space of 800+ Pokemon with 18 possible typings
and 6 stats per Pokemon:

1. Narrow down the search space to all Pokemon that are <type>

2. Narrow down the search space to only the <stat>
of the Pokemon and ignore all the other 5 stats.

3. Extract all Pokemon which meet the conditions set out by the
<degree>. These conditions are quantified using the approach
described in the previous section.
```

Taking our 18 different Pokemon types and 6 stats per Pokemon, we can represent this process as a tree. I will illustrate this with an example. The image below shows the above query processing model applied to the query - "find type degree attack" for an arbitrary degree and type.



Similar to the subtrees of the "type" node, each of the other Pokemon type nodes on the second level (with the first level being the root) have the exact same subtrees as the "type" node. This is not shown due to lack of space.

The blue lines indicate the path we take from the root to the leaf of the tree for our specific query. In the case where instead of attack, we had a different stat, then we would just pick a different subtree on the the third level (with the first level being the root). On the other hand, in the case where we had a different "type", we would pick a different subtree on the second level.

Also notice that on the left of each layer, the list represents the information that is relevant to choosing a subtree at the current level and choosing future subtrees. When we start, we need to whole query, but after we "narrow" down our search space by recursing on a specific Pokemon type, we do not need the type anymore and discard it. The same logic applies to after we choose the subtree corresponding to the desired stat in the query.

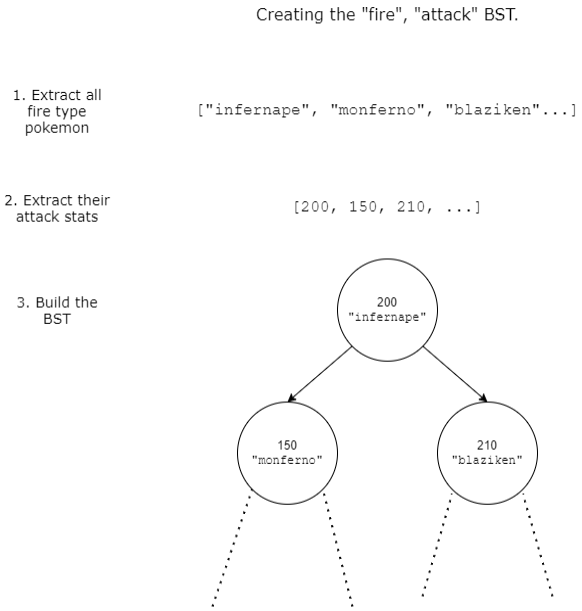
However, step 3 is still ambiguous, how do we extract all Pokemon which meet the restrictions set out by the degree of the query? Do we iterate through each candidate Pokemon and store them in a list?

Using Binary Search Trees

While there are multiple approaches to this, I decided to use something we learned in CSC111. Given a degree and stat, the task is to find all Pokemon of a certain type who have a stat value which meets the inequality constraints set out by the degree. This means there will be Pokemon which aren't even worth considering, which I want to "skip". For example, if I'm searching through Pokemon of a certain type and looking for ones that have a specific stat value that meets the criteria of the "high" degree, there definitely will be quite a few Pokemon I want to skip over, since their stat needs to be in the 75th percentile or above. A datastructure that accomplishes this is called binary search trees (8).

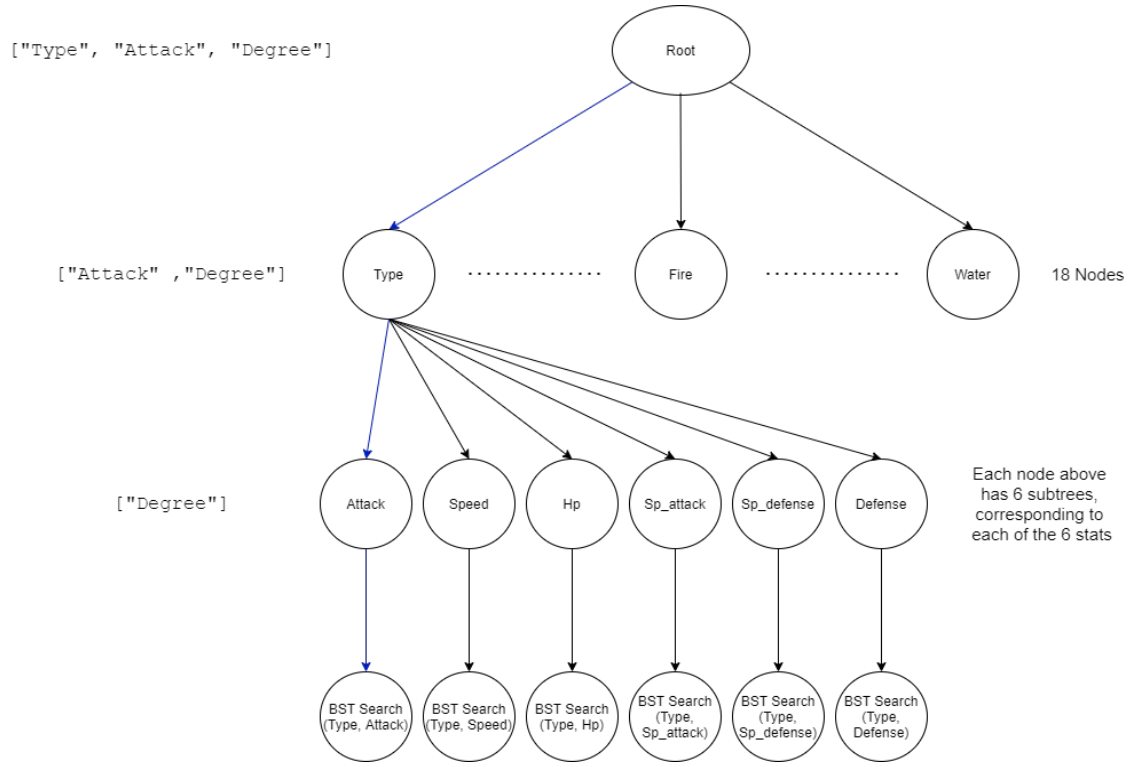
More explicitly, here's how this system would work. Say we had a binary search tree for every typing and stat, this is in total $18 \cdot 6 = 108$ BSTs. To create these trees, given a certain type and a certain stat, we first isolate all Pokemon that have that type. Then, we use this set of Pokemon to create a BST where each node contains a Pokemon's stat value for the specific stat. Additionally, each node also contains the Pokemon's name (as a string). If we made a query with a stat, Pokemon type and degree, then we would first need to look for the BST that corresponds to the typing and stat. Once this tree is found, we can recurse through the BST and find all nodes which contain values which meet the restrictions imposed by the degree. For each node that meets our degree criteria, we can store the Pokemon's name. Hence, in the end, we'll have a list of all Pokemon names of a certain typing where their specific stat from the query is in accordance with the restriction set by the degree.

To be more concrete, here's an example of how we would create the BST for the attack stat of fire type Pokemon, or the "fire", "attack" BST:



Why do this over simply iterating through the list of all Pokemon of the type given in the query and filtering out the ones whose stat values do not meet the degree restriction? This is because of a unique property of BST. As said before, if I'm searching for all Pokemon of certain type that have a specific stat which meets the criteria of the the "high" degree, there definitely will be quite a few Pokemon I want to skip over. Hence, the BST property does this for me. This is because if I run into a node with a stat value lower than the 75th percentile bound (this pokemon's stat is too low to meet the degree restriction), I can simply skip over the left subtree of that node and not recurse on it or any of its children, because I know all nodes in the left subtree are less than or equal to the current node (BST property). This means I am skipping over all Pokemon of the given type with a stat value lower than or equal to the stat value of the Pokemon in the current node. Therefore, this saves a good amount of computation, and is a decent place to use BSTs.

Now, with this new idea, we remove the ambiguity of step 3, and can rewrite our tree diagram from earlier. The image below shows our query processing model (with step 3 now being the new BST search) applied to the query - "find type degree attack" for an arbitrary degree and type.



As before, if we had another stat, the only difference would be in the subtree we recurse onto in the third level (with the root being the first). Instead of "attack" it would be something else.

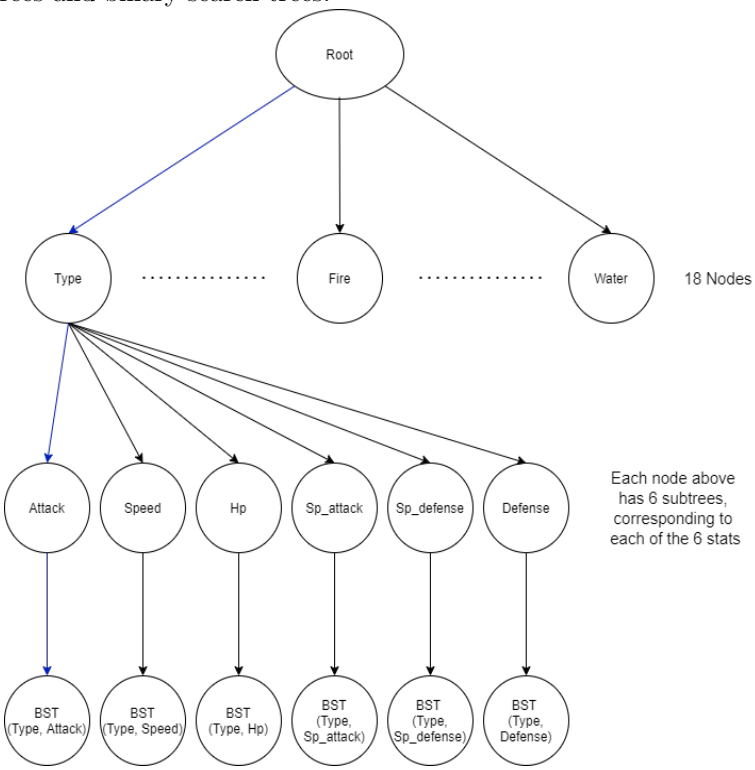
Similar to the subtrees of the "type" node, each of the other Pokemon type nodes on the second level have the exact same subtrees as the "type" node. Except for each type node, the BST children have a different type/stat combination. For example, the "fire" pokemon type node will have a fire/attack BST child, but the "grass" pokemon type node will have a grass/attack BST child.A

Finally, after all of that theoretical groundwork, we have come up with a final query processing model:

```
Given a query in the form: find <type> <degree> <stat> and a
search space of 800+ Pokemon with 18 possible typings
and 6 stats per Pokemon:

1. Narrow down the search space to all Pokemon that are <type>

2. Narrow down the search space to only the <stat>
of the Pokemon and ignore all the other 5 stats.
```

```
3. Convert the <degree> into numerical restrictions
using percentiles.

4. In the BST corresponding to the <type> and <stat>, recurse
through the subtrees using the BST property, storing all Pokemon names
whose <stat> value meet the restrictions set out by the <degree>
```

Notice that the nodes for pokemon typings and stats act as sort of "decision trees", where when evaluated with a query, the subtree to recurse on is decided using a categorical value. The final decision tree node that is recursed on is the one which facilitates the binary tree searching. Hence, given the tree diagram, we can represent a implementable query processing model as a combination of decision trees and binary search trees:



Note that each Decision Tree with a pokemon stat has only one subtree, which is a BST that stores values depending on what the its ancestors are. Given the specific stat and type that its ancestors represent, the binary search tree stores the specific stat values and names of pokemon of that type. When this tree is used to process a query, the decision tree corresponding the stat being searched for will use its BST subtree to find the names of all pokemon that satisfy the constraints set out by the degree. Taking the previous example, given the query "find type degree attack", the "attack" decision tree which is a subtree of the "type" decision tree will use its BST child (the BST for "type",

"attack") to find all pokemon of "type" with attack values which satisfy the constraints set out by the degree.

Implementation details

Before, explaining my main.py, I will go over a class I created. I created a class in decision_tree.py called DecisionTree. This class is similar to the tree class in CSC111, but there's a few changes. The self.subtrees is still the same, but there is an additional an attribute called self.category. Additionally, it also has two attributes called self.is_binary_parent and self.conversion_dictionary, which will be addressed later.

The above query processing model is implemented in the following way in my code, here are the steps in order:

1. I use read_data() from process.py to load in my data and store it as a pandas dataframe (3).

2. Then, I call create_decision_tree from process.py, passing in my dataframe as a parameter. This function has a few parts, so I'll go through them in order.

   (a) I use the find_quantiles function from process.py, which first extracts the columns corresponding to each stat using pandas dataframe methods. Then, I convert them to numpy arrays using the to_numpy() function. The reason I do this is because I can call the function np.percentile, which gives me the percentiles for each numpy array. Using this these numpy arrays and the numpy.percentile (1) function, I create and return a dictionary mapping of degrees to their corresponding numerical restrictions. Their numerical representations are represented as a tuple that looks like $(lowerbound, upperbound)$. For example, "medium speed" would map to a tuple with the 25th percentile of speed as the lower bound and the 75th percentile as the speed bound. But "high attack" would have *None* in the upper bound slot and the 75th percentile of all Pokemon attack stats in the lower bound slot. This is because (as explained previously) there is no upper bound for the "high" degree, so we represent that as *None*. Since we have 6 stats and 3 possible degrees, we have 18 key-value pairs in the returned dictionary.

   (b) Now we create the decision tree (This is simply the code translation of the tree diagram):

      i. First we create a "root tree", which is a decision tree that has a category of *None* and self.is_binary_parent is False since (this will be shown in the next step) this tree is a parent to only DecisionTrees and no BSTs.

ii. Then, we create 18 decision trees (there are 18 types), for each Pokemon type. We represent this by setting the self.category of each of these trees to be equal to the type they belong to. These trees are stored as subtrees to the root tree. Again, self.is_binary_parent is False since (this will be shown in the next step) this tree is a parent to only DecisionTrees and no BSTs.

iii. For each of the trees constructed in the above step, we give them each 6 subtrees corresponding to each of the 6 Pokemon stats. Each subtree's self.category will be equal to the stat that they belong to. Since the single subtree of these subtrees will be a binary search tree (explained below), we set self.is_binary_parent to True for all subtrees constructed duing this step. We also set self.conversion_dictionary to be the dictionary we previously generated (the use of this will be explained later).

iv. Finally, each tree constructed in the above step gets assigned a binary search tree as the sole subtree. Therefore, as explained previously, this is $18 \cdot 6 = 108$ BSTs in total. These BSTs are constructed using the create_bst() function in process.py. The structure of this BST was previously discussed in the functionality overview section. Notice that each BST is the child of a decision tree corresponding to a stat, where the parent of that decision tree corresponds to a Pokemon type. Therefore, each BST is constructed using the values of the stat that its parent represents and the stats are taken from the pokemon of the typing that the parent of the parent represents. For example, the binary search tree which is the child of the "attack" decision tree whose parent is the "fire" decision tree will be built using the attack values of all fire type Pokemon and the names of all fire type Pokemon.

Evaluating a query

Now given the tree constructed in create_decision_tree(), we can use it to compute Pokemon recommendations on queries. Here's an example:

We have query *find fire high defense* and a variable *tree*, which corresponds to the root of our decision tree. We first turn our query into the following list *["fire", "defense", "high defense"]*. Then we call *tree.evaluate(["fire", "defense", "high defense"])*.

1. First, we check if self.binary_parent is True, but we know it isn't true for the root tree. Therefore, since this is true, we take the first element of our input *["fire", "defense", "high defense"]*, this element is *"fire"*. We search the subtrees of the root to find which decision tree corresponds to the *"fire"* subtree. Once we have found it, we recurse onto it by calling *subtree.evaluate(["defense", "high defense"])*. Notice that we discarded

the first item. This is because we don't need it anymore as we've "narrowed down our search space", this will also be seen in preceding recursive calls.

2. First, we check if self.binary_parent is True, but we know it isn't true for the Pokemon type decision trees. Therefore, since this is true, we take the first element of our input *["defense", "high defense"]*, this element is *"defense"*. We search the subtrees of our current tree (Subtrees of the fire type decision tree) to find which decision tree corresponds to the *"defense"* subtree. Once we have found it, we recurse onto it by calling *subtree.evaluate(["high defense"])*.

3. Then, we check if self.binary_parent is True, and in this case it is, since the child of this tree is a BST (the only subtree of this tree is a single BST). Therefore, in this case, we first take the first element of our query and use self.conversion_dictionary, which converts *"high defense"* into a tuple that looks like *(75 percentile of defense stat of fire type Pokemon, None)*. Hence, using this, we call the subtree BST's find_nodes_with_constraints(*(75 percentile of defense stat of fire type Pokemon, None)*) function from bst.py, and this will return a list of names of all fire type Pokemon which have defense stats that meet the constraints of being "high defense". This behavior will only happen if self.is_binary_parent is True, which is the purpose of the variable - allowing our program to know when to use the BST search method.

Visualizing my results

I've prepared a interactive visualization of this computation in the form of a GUI. I used a library called PySimpleGUI to do this (6) (5). This library is based off tkinter and gives the user a lot of widgets to use.

My gui allows users to to search up queries in a search bar. Upon searching, the query processing tree as described above will return a list of Pokemon, which will be displayed in a scrollable field on the GUI. Each Pokemon has its own box with its picture and name on it. The user can click on a Pokemon and click "display stats" which will open a side panel that shows the stats of the Pokemon. Additionally, the user can click "add to party" to add the Pokemon to their party. This is displayed as 6 boxes at the top of the GUI, where all 6 boxes start out blank, but get filled up as the user adds more Pokemon to their party. Additionally, this wasn't covered above, but the user can also make queries which create popups of matplotlib (9) histogram graphs using plt.hist(). These graphs show the histogram distribution of a specific stat for a specific Pokemon type. This way, the user can get an idea of how stats differ between certain types in their distributions. For example: The attack stat distribtion of fighting types is centered higher than the attack stat distribution for ghost types.

Additionally, the use of the "pokedex_number" column in my dataset is to render Pokemon icons on my GUI. Each Pokemon has a unique number, and for

each number, I have an icon in the "icons" folder, which is rendered when that Pokemon needs to appear on the GUI (Each pokemon's icon is named after the pokedex number of the pokemon). The citation to the source where I pulled these icons from is given in (4).

This was possible because of the simplicity of the library, as it gives the user an easy to use event-loop style framework. This means all I needed to do was define functions which trigger every time a certain event happened. The actual design and layout of the GUI was very simple, which is one of the selling points of this library (5).

Use of libraries: Efficiency

One thing to note is that during the BST creation step, we are creating $18 \cdot 6 = 108$ BSTs in total. This is a massive number, and most people would intuitively think this will take a huge amount of time to make (I thought this too during my proposal). The full decision tree is created in a matter of a few seconds. The reason for this can be attributed to the Pandas library. As described in the implementation details section, our first step is loading in the dataset as a pandas dataframe. These dataframes are very powerful, they allow us to efficiently extract columns of data. This means, I don't need to iterate over all rows of the .CSV every time I want to extract a column. Additionally, the library allows me to filter data (eg: extracting ONLY fire types) using boolean operations. For example, I can find all Pokemon of the fire type by simply doing:

```
df[(df["type1"] == "fire") | (df["type2"] == "fire")]
```

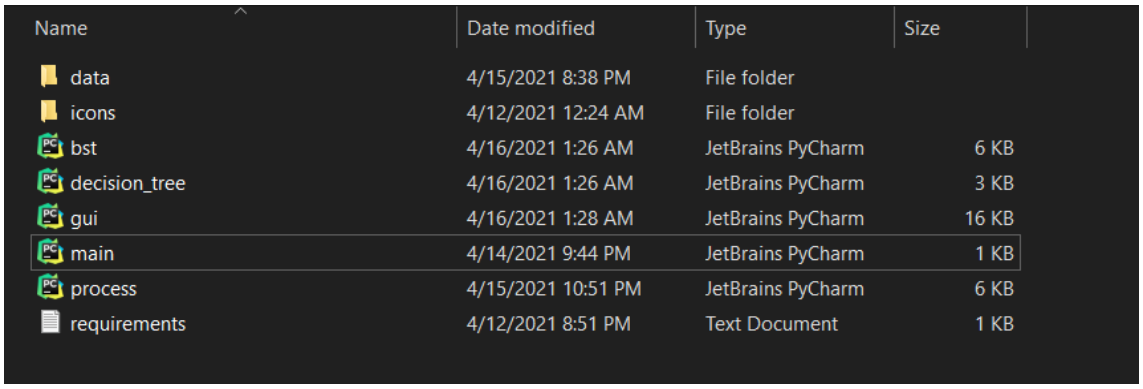
Therefore, all dataset operations (extracting columns and filtering data) that are required for this project can be done with pandas functions (3), which are highly efficient and save a lot of time. Aditionally, since we are creating distinct BSTs for each stat/Pokemon typing combination, each BST only contains a small subset of all Pokemon in the dataset (for example, fire types only make up about 10% of the 800 Pokemon), so they individually don't take too long to create.

Obtaining datasets and running the program

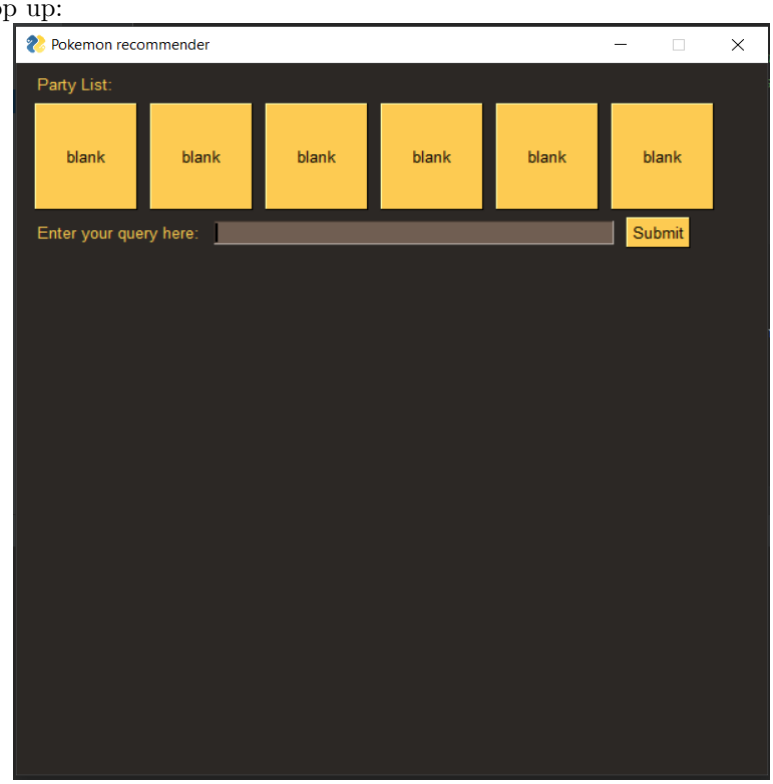
After installing all libraries from the requirements.txt, download this file from google drive.

```
https://drive.google.com/file/d/1bCgAsWunc_wk0klc_tAEBkgvaT6DBAmu/view?usp=sharing
```

Extract the zip file. You will see two files - "data" and "icons". Take both and put them inside the project folder. Here is what your folder should look like:



Run the main.py file. After a few seconds, you should see the following GUI pop up:



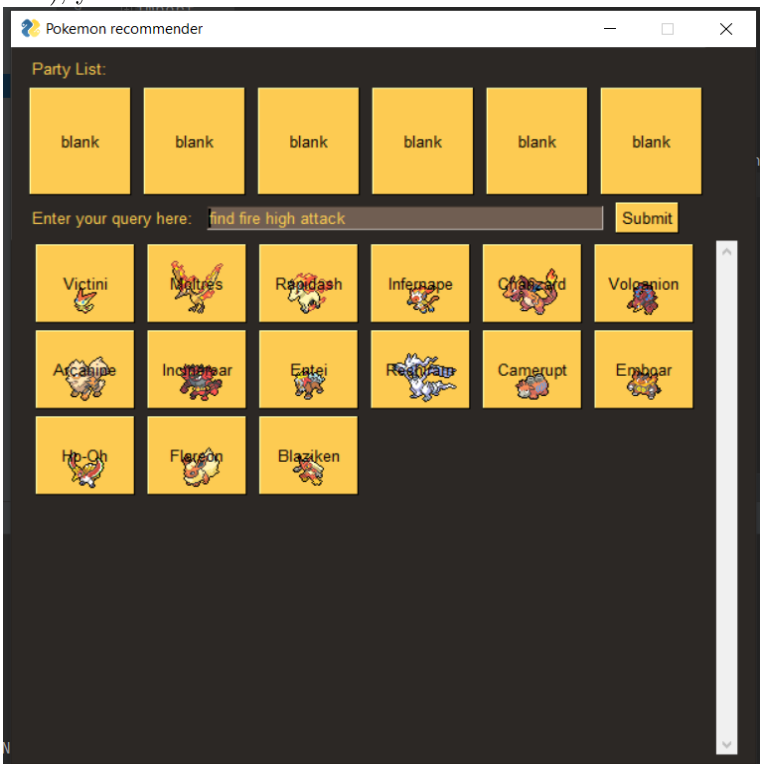
NOTE: all usable degrees, types and stats are in the appendix

Here are the queries you can make. My program does have error handling, but stick to this query format to get results. The queries are case sensitive, so follow the case that all Pokemon types and stats in the appendix follow (all of them are lowercase). Additionally, be mindful of the spaces, there must not be any extra spaces then what is shown in the description, otherwise there will be

a malformed query:

```
To receive Pokemon recommendations, use the following format:
"find <type> <degree> <stat>"

To plot histograms of stats for Pokemon types, use the following format:
"plot <type> <stat>"
```

Upon making a query (in this picture, I make the query "find fire high attack"), you will see a few Pokemon on the scrollable element of the GUI:

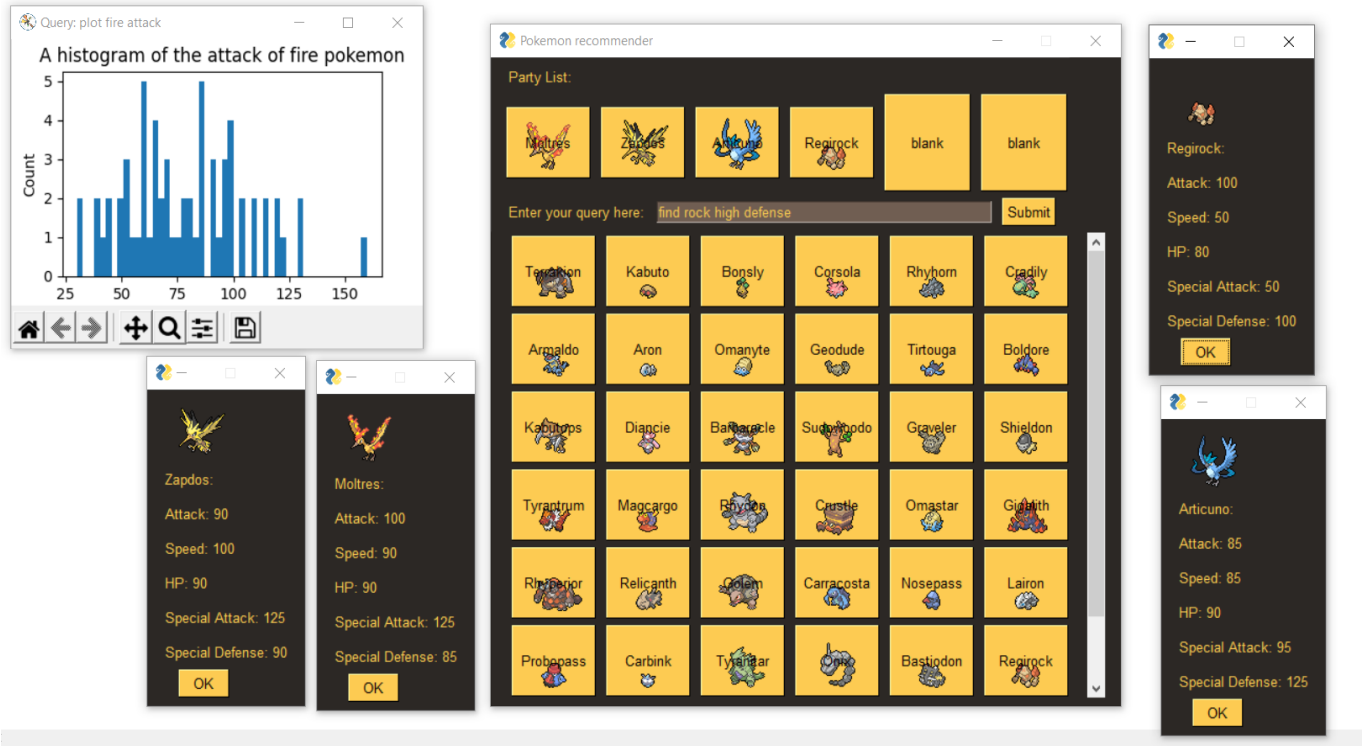


Click on any of the Pokemon in the scrollable field and you will see two options - "add to party" and "display stats". Upon clicking "add to party", you will see the Pokemon appear on your party at the top (Your party is the 6 boxes on the top).

You can also click on Pokemon in your party (if the box has no Pokemon, you are presented with "no action", which means there are no actions to be taken with a blank party slot) and are presented with the options "remove from party" and "display stats". Removing a pokemon simply frees up the party slot, this is useful if you've hit the 6 pokemon party limit and want to free up space.

Clicking "display stats" on a Pokemon in your party or the scrollable query result section will open up a little popup window that shows the stats of the Pokemon.

The same thing happens when you do the "plot" query, a window with a histogram will pop up. All popups don't interrupt the main GUI, so you can open as many as you want and move them onto the side. Here's an example of a few popups open and a few party slots filled:



Note that sometimes the popups open "behind" the main gui, depending on your system settings. If you don't see the popup on your screen immediately, check your taskbar and see if any popups opened.

Lastly, note that your party is saved between queries, so feel free to make multiple queries and add any Pokemon you find interesting between your searches, your party will only change if you add or remove a Pokemon.

Changes between project proposal and final project

Some advice that was given during my proposal was to use decision trees. I didn't really understand how to use this at first. After talking to professor David in office hours, he gave me the idea of using BSTs as subtrees to a decision tree. I had originally never considered the idea of combining two types of trees together into a Decision-BST hybrid tree.

Additionally, the tkinter library was much too hard to use, there were too many things I needed to know to make a functional GUI. After a bit of searching,

I found EasyPythonGui, which had a very well written documentation and easy to grasp structure.

Discussion

Looking back at my research question

I set out to answer the research question:

Can we simplify the process of creating a Pokemon team using a query-based Pokemon recommendation system?

In other words, my initial goal was to use my knowledge of programming and computer science to simplify the process of creating a Pokemon team, to make it more beginner friendly. This is in contrast to most Pokemon teambuilders, as they simply throw you the names of all Pokemon and expect you to know enough to build your own team.

I think my computation + display do a pretty good job at achieving this. My main goal was to prevent new players from being overwhelmed from the various choices of Pokemon they can choose from. Using my decision tree-BST hybrid structure and a query, I "narrow down" the large list of Pokemon that the user sees to something a lot more bearable. Most search results in my GUI give you about 10-25 Pokemon recommendations, which is much better than a scrollable list of 800+ Pokemon. Additionally, I have made the queries broad enough so that the user doesn't really need to memorize Pokemon names, but rather just know the general type of Pokemon they're looking for, which is a lot more palatable. On the other hand, the GUI is generally pretty bare bones, but I think the decision to add the pop-up side windows contributes a lot to making it more beginner friendly. My goal was to abstract away a lot of the of the complexity, namely throwing a bunch of numbers at the user. Rather than displaying the stats directly on the GUI which can be overwhelming, my program only gives the user numerical stat information in the form of popups when they need it, which makes comparing/analyzing Pokemon stats a lot easier.

Additionally, while this was not built into my project, this decision tree-BST hybrid has the ability to scale. As more Pokemon content get released, they bring along with them more Pokemon, with different stats and sometimes even new typings. If I wanted to adapt my tree to these new things, all I would need to do is add new subtrees. For example, if there was a new fire type Pokemon released, integrating it in my tree would simply be adding the Pokemon to the necessary BSTs, which are all BSTs that are descendants of the "fire" decision tree.

Limitations

I did find a key limitation with my dataset, particularly in terms of Pokemon evolutions. To add context, Pokemon can "evolve" into other Pokemon. Ther-

fore, some Pokemon have "evolution chains", which are basically a series of Pokemon which evolve into each other, starting from a base Pokemon. However, Pokemon with no evolutions (Pokemon which cannot evolve any further) are normally much stronger than Pokemon that do have evolutions. By strong, I mean their stats as a whole are higher. I would have liked to give the user an option between choosing to have Pokemon with no evolutions or all Pokemon as the dataset used to create the decision tree. However, I could not find a dataset that paired well with the dataset that I used in the project that contained this information, so I was limited by my dataset in this case.

Another limitation was my SimplePythonGui library. Particularly around "dynamic" GUIs. My GUI operates in a fashion where it changes its structure depending on the user query, whether that be adding a Pokemon to the party or rending query results. This sort of GUI doesn't really play too well with the GUI library I used, and when I tried to increase the complexity by using different widgets, I ended up running into lots of bugs. Therefore, my options on what type of GUI I could design were very limited. Particularly, one feature I would have liked to add was being able to drag Pokemon boxes around within the GUI (Ex: picking a Pokemon up using the mouse and moving it somewhere else within the GUI), but this dynamic GUI feature was not very functional in the library.

Though, to my surprise, I ended up helping the developers of this library out by opening an issue which led to a bugfix to their library:

`https://github.com/PySimpleGUI/PySimpleGUI/issues/4177#issuecomment-818368277`

Additionally, my query-processing system has a key limitation as well. While my decision tree does represent the data in a good way for processing queries in the format outlined in this project report, it doesn't allow queries for similarity. More explicitly, if I want to search for Pokemon similar to a certain Pokemon (Ex: a query like "recommend me Pokemon similar to pikachu"), my tree does not capture this information. The reason I am explicitly mentioning this specific type of query is because this is a common thing that beginners do - they take a Pokemon they're familiar with and look for similar ones. These sort of queries would be more in line with a weighted graph-based algorithm, which stores similar Pokemon as neighbors. I wanted to find a way to simulate this idea in my tree, but I was unable to come up with a method of doing so.

Further steps

This program is far from being a complete application for designing Pokemon parties. While I've managed to create a datastructure that allows user to search for specific Pokemon whose stats meet certain criteria and a user-friendly GUI to go with it, there's a lot that my program does not do. This was briefly introduced in my problem description, but Pokemon has a lot more complexity

to it than just stats and types. Disregarding the specifics, there's also Pokemon items, abilities and moves which all impact Pokemon battles in some way or the other (7). Additionally, there are statistics around certain Pokemon, where some have been shown to perform statistically better than others in professional Pokemon events (during E-sporting events). While it was abstracted away in this project, there's a lot more key information to consider when choosing a Pokemon for your team.

A further step would be to incorporate all of this other data in a beginner friendly manner into my application.

Conclusion

Throughout this project, I've successfully used my knowledge of Trees from CSC111 to create a query processing Pokemon recommendation algorithm and structure. Additionally, I've created a functional GUI which is simple to use for beginners. I've also covered the limitations and further steps of improvement of my program.

Appendix

Here is some information about Pokemon that could be relevant in understanding this report or using the program.

The type of a Pokemon can be any of the 18 types: 'flying', 'ice', 'psychic', 'ghost', 'water', 'ground', 'steel','rock', 'fighting', 'fire', 'electric', 'poison', 'grass', 'bug', 'dark', 'normal', 'fairy', 'dragon'

The stat of a Pokemon can be one of the 6 stats: 'attack', 'defense', 'sp_attack', 'sp_defense', 'hp', 'speed'

This was stated in the computational plan section, but the degrees that can be used in a query are 'low', 'medium' and 'high'

References

(1) NumPy V1.20 MANUAL. (n.d.). Retrieved April 16, 2021, from https://numpy.org/doc/stable/

(2) The official pokémon website: Pokemon.com: Explore the world of pokémon. (n.d.). Retrieved April 16, 2021, from https://www.pokemon.com/us/

(3) Pandas documentation¶. (n.d.). Retrieved April 16, 2021, from https://pandas.pydata.org/docs/

(4) PokeAPI. (n.d.). Pokeapi/sprites. Retrieved April 16, 2021, from https://github.com/PokeAPI/sprites

(5) The pysimplegui cookbook. (n.d.). Retrieved April 16, 2021, from https://pysimplegui.readthedocs.io/en/latest/cookbook/

(6) PySimpleGUI user's manual. (n.d.). Retrieved April 16, 2021, from https://pysimplegui.readthedocs.io/en/latest/

(7) Introduction to competitive pokemon. (n.d.). Retrieved April 16, 2021, from https://www.smogon.com/dp/articles/intro_comp_pokemon

(8) Badr, D. (n.d.). CSC110/111 course notes. Retrieved April 16, 2021, from https://www.teach.cs.toronto.edu/ csc110y/fall/notes/

(9) Overview. (n.d.). Retrieved April 16, 2021, from https://matplotlib.org/stable/contents.html