Spring 2021

Clustering Data to Classify Hearthstone Decks

Tim Inzitari
tsi3@zips.uakron.edu

Clustering Data to Classify Hearthstone Decks

Tim Inzitari

The University of Akron – Williams Honors College

April 2, 2021

ABSTRACT

The esports game of "Hearthstone" is a collectible card game with a competitive format that has every team submit 4 decks of 30 cards each. Using K-Means clustering an adaptable way to group data for classifying can be made that works well in every update of the game. This system will take in a list of decks and cluster them to easily classify large amounts of information in a timely fashion. This system will be able to be used by the Universities esports department for years to come to aid the preparation of "Hearthstone" matches. This model uses qualities about the cards in the deck to generate a vector usable by a K-Means and Random Forest classifiers. This model has shown extensive results and consistently reaches seventy percent accuracy for entire datasets.

Table of Contents

Table of Figures

CHAPTER I.

INTRODUCTION

This project aims to create a set of tools in the Python language that will allow users to easily and quickly classify large amounts of decks from *Activision-Blizzard*'s electronic card game *Hearthstone*. The program will give the user freedom in what each deck gets classified and allow for the storage and callback of past classifications to help classify smaller datasets easier. The program will also provide tools to generate data required to train the model in an efficient and timely manner. These tools will be used mainly for tournament preparation, and by being able to quickly and accurately classify large amounts of decks these tools will greatly aid any competitor.

CHAPTER II.

BACKGROUND

Hearthstone is unlike any other esport on the market, unlike popular first person-shooters such as Valve's *Counter-Strike: Global Offensive* or Blizzard's *Overwatch* you do not need any sort of aiming abilities. Additionally, you do not need quick reaction speeds like in faster paced games such as Riot Games' *League of Legends* or Psyonix's *Rocket League*. *Hearthstone* is a slow, turn-based card game that has more similarities to chess or poker than other titles mentioned.

In 2014 Blizzard Entertainment released their collectible card game titled *Hearthstone* and it was almost instantly a commercial success. The online game was simple to play and learn, while still having a surprising amount of depth to it. Very quickly players began gathering and forming tournaments in the game, which led to a fully supported esports program from *Blizzard*. I have been playing *Hearthstone* competitively for nearly 7 years now, 3 of them being for the University of Akron as an initial member of their esports program.

Overtime various ways to run these tournaments existed, but as time passed one became the accepted standard: "Open-list Conquest with Ban" (or simply Conquest). While formally defined in the Player Handbook, the format commonly comes in two variants: best-of-five and best-of-three. In each variant a player selects a predetermined number of decks from the games 10 unique classes, in best-of-five you bring 4 decks, and in best-of-three you bring 3. Before a match starts each side will see a full list of the cards in their opponents' decks and select one of the decks to "ban", these ban decks are not eligible to play. You then play games of *Hearthstone* with the decks until each non-banned deck wins for a player.

Conquest has been the dominant format since 2015 and as time has passed competitors have developed various theories and practices on how to best approach tournaments such as these. With most tournaments not allowing you to change deck lists between rounds, a key part of every tournament comes before any card is played. This preparation phase has developed and changed throughout the years, but one part of it has stayed the same: you must have a firm grasp on the current "metagame". The metagame is a term used to describe the best and most popular decks that will appear in the tournament or while playing *Hearthstone*, this metagame typically changes constantly especially around the game's patches.

One of the best ways to analyze these metagames in preparation for competitive tournaments or matches is to look at previous tournaments to see what was popular, what worked and what did not. This often led to hours of tedious work of pulling up deck lists, classifying them and organizing them into a spreadsheet by hand, like in Example 1.

Player	**Deck 1**	**Deck 2**	**Deck 3**
John	Control Warrior	Aggro Hunter	Tempo Mage
Amy	Ramp Druid	Combo Demon Hunter	Freeze Mage
Riley	Zoo Warlock	Totem Shaman	Highlander Mage

Table 1 Example Spreadsheet

Notice how all three of these players have brought a Mage deck, but each of them was classified differently; John has "tempo", Amy has "freeze" and Riley has "highlander". These different classifications are examples of different "archetypes" within a class and where the root of this problem occurs. While familiar players can look at deck lists and almost instantly classify these decks, tools for computers to do this are far more difficult and complex.

This problem is what I set out to solve with my honor's project, I wanted to develop a tool to help a computer classify these tournament decks. A quick and accurate tool of this caliber could save hours in preparation time for every tournament and every match you compete in.

To accomplish this project, I knew I would need some form of computer intelligence and would be forced to develop a machine learning model on the decks. I realized that using a python library from HearthSim I could convert the base 64 strings that game transcodes decks as into useful numbers that could be applied into a model. This model would then be able to use a K-Means clustering method to classify the decks given the correct vectorization of those decklists. Two other problem arose; if I am using a K-Means algorithm I will need a large quantity of decklists, more than any person could reasonably create on their own. As well as how to structure clusters, a simple cluster would not work as classifying decks across the classes would be pointless. This means a custom cluster structure would have to be created for this project. A fourth problem was that these archetypes are changing constantly, so the system would have to be adaptable to work in any metagame given data supplied.

CHAPTER III.

DESIGN

In terms of programming language that this project was created with, an instant leader appeared while scouring options: Python. With numerous machine-learning and data science packages such as Pandas, NumPy and Sci-kit Learn as well as the HearthSim's hearthstone-python package the choice was simple. Specifically, the version of Python used for this project was 3.8 given it was the most recent stable release at the time I began.

In terms of hardware requirements for this project, it can be run on any device capable of handling Python, but I designed it with the computer builds at University of Akron Esports in mind, specifically machines with multicore processors, graphics cards and high amounts of RAM. The specific components are: Intel i7 8700 6 core-12 thread processors, Nvidia GTX 1080Ti Graphics cards and 16 GB of RAM. While these hardware recommendations are steep, it is since the main user of this program is intended to be the University of Akron Esports team.

Inputting and Exporting Data

The first major design decision that had to be made was how I wanted import and export data throughout the program. This decision would have major impacts on how datasets would be gathered, and how the entire flow of the program would occur. For both I decided upon comma-separated-value (csv) files due to their ease of use with both libraries and how tournaments typically distribute deck lists. This format of distributing deck lists typically used involved a spreadsheet of the base64 "deck codes" the game uses to help players communicate decks between each other. An example of one of these deck codes is:

"AAECAaIHAsGuA9nRAw60Ae0ClwaIB4YJj5cD+6ID9acDqssDpNED390D590D890DgeQDAA=="

These deck codes could then be easily converted to deck lists that can be applied later in the program.

As for exporting data, one exported data the program required was clear: a list of classifications for any given players decks. This was fantastically formed by the csv files and makes transfer easy. The other exported piece of data from the program will be a list of labeled values from the clustering that can be used later to simply classify a set of decks without clustering.

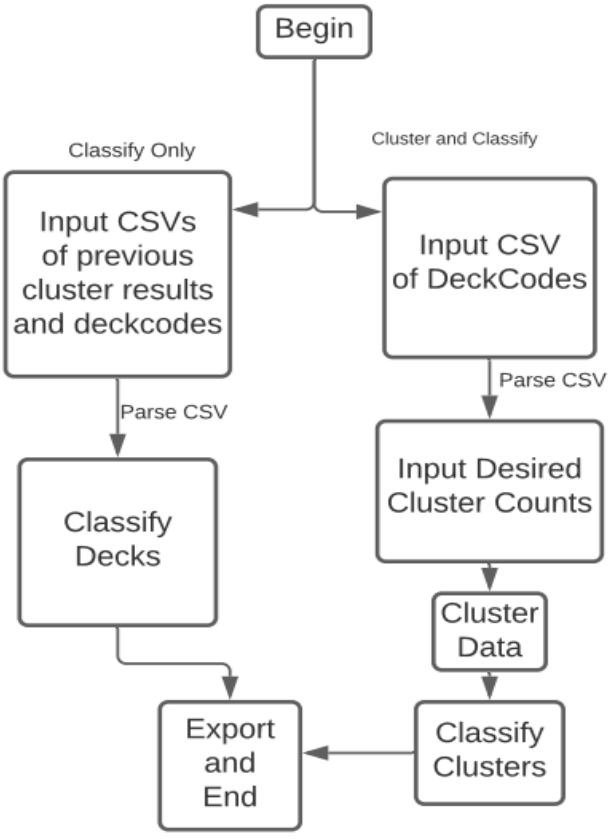

Figure 1 Program Flow Chart

Program Design

For the program to work functionally, a flow structure had to be designed. The result of this design is shown in *Figure 1*. It will begin with giving the user a choice to only classify data using a previously clustered dataset, or cluster and then classify a set of data.

If the user choses to cluster and classify data, it will then begin a program that occurs in five major stages. Stage 1 will be prompting the user to input the csv file containing the deck codes of every team. The input for this can be as simple as taking the Google Spreadsheet and converting it to a CSV file using Google's built-in tools. The program will then parse the CSV into the deck data structure used by the program before moving on to the next stage. The next stage will be another user prompt screen where the user can input desired cluster counts to generate for each in-game class. The program will then proceed to vectorize the decks and cluster them in stage 3. Following this clustering the program will have the user classify each cluster, this will be done by displaying a certain number of decks in each cluster and asking the user to give it a name. A task that for experienced *Hearthstone* players such as those who would compete in tournament will take very little effort and be near instant for each instance. The program will then combine any clusters that were given the same classification and proceed to export the data in stage 5.

If the user choses to only classify using previously clustered data, the user will be prompted for a folder containing the previously generated labels for each class, and an input CSV of deck codes to classify using those labels. The program will then classify the new decks and export the results.

Data Structure Design

Two main data structures had to be designed for this project, the first being a structure that can store information on any given deck. To tackle this data structure an object is created that can implement the HearthSim deck storage while maintaining information on classification, and what team owns the deck. Also included is a list of cards from the deck string, to more quickly access this information without requiring parsing it using HearthSim's system every time. The resulting data structure named *DeckWrapper* is shown in *Figure 2*.

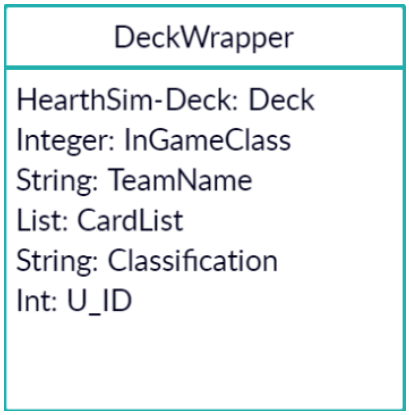

Figure 2 DeckWrapper Diagram

This structure allowed for easy and quick distribution of decks throughout the program while remembering who it belongs to for exporting.

The second main data structure design is in the form of the cluster structure. A tree structure of nested lists was implemented to help transfer and track clusters throughout the program using various types of objects. These trees, called "SuperClusters" would have a height of four and the setup is shown in *Figure 3*.

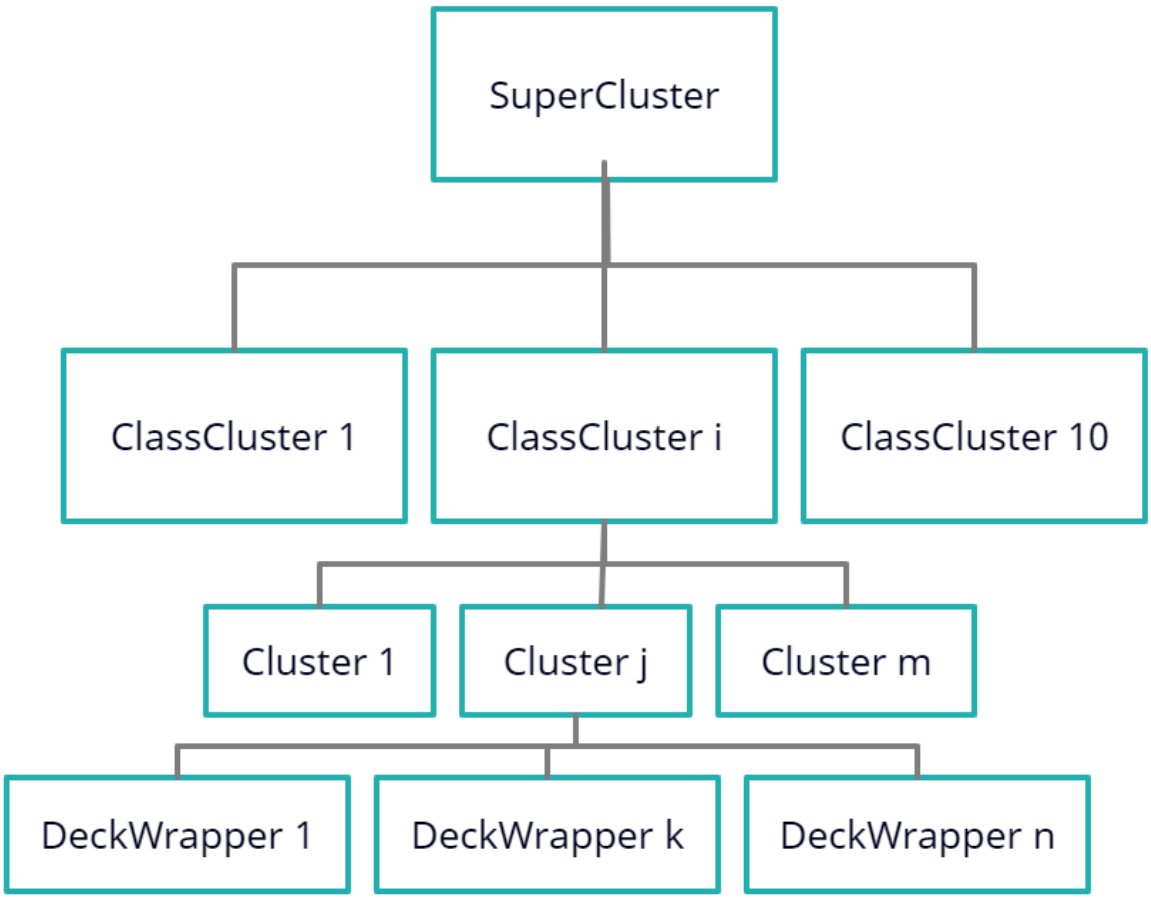

Figure 3 Cluster Structure Design

The SuperCluster will serve as the root of the tree, with the first branch layer consisting of ten ClassCluster objects, one for each class in *Hearthstone*. These ClassClusters will then branch to *m* cluster nodes, for each cluster the user desires for that class. These clusters will contain *n* leaf nodes each, where *n* is the number of decks in each cluster. This structure will allow for any given deck to be found in $O(\log_m n)$, with the base *m* of the logarithm varying for each class and *n* being the number of decks in the class. Each of these classes also contains various functions and values needed to function.

Graphical User Interface and Deck Display

Initially I was only going to have the deck displaying portion of the program have a graphical user interface (GUI) but after some consideration about my target audience I realized that not all will be familiar with a command line prompt, so I moved towards a full GUI system.

The GUI is simple box and text inputs as well as output displays to help users follow the programs progress and input required information. *Figure 4* shows the screen that displays when prompting a user for cluster counts.

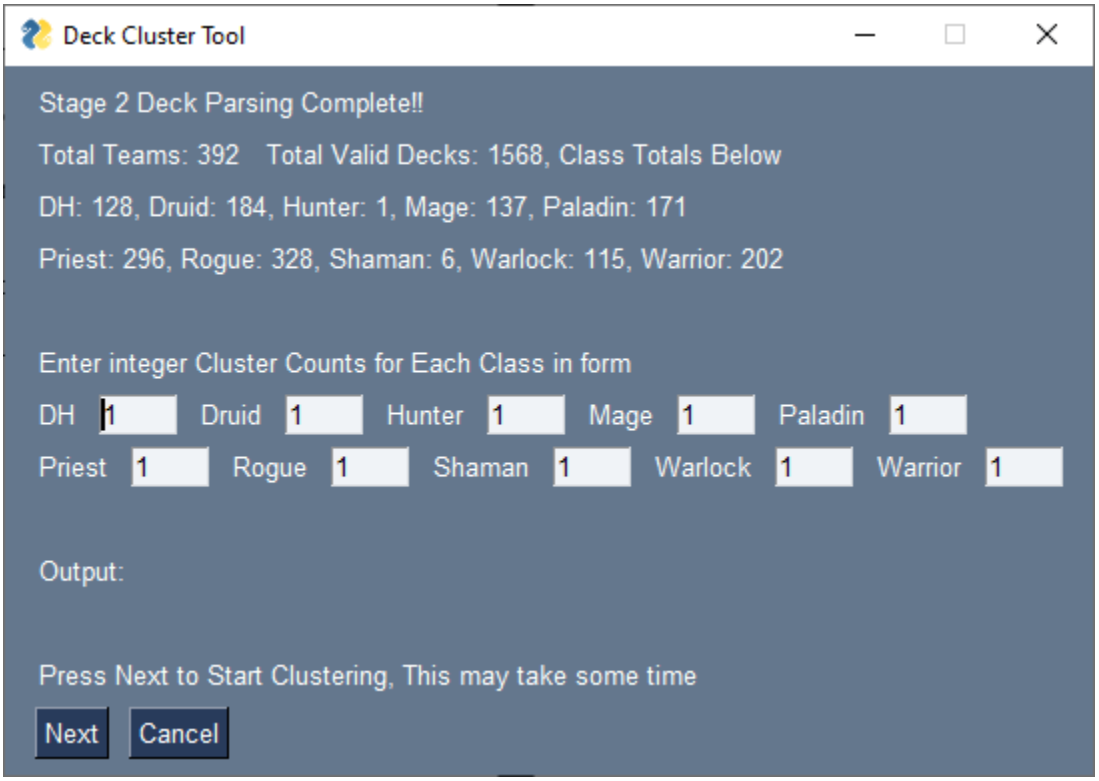

Figure 4 Example of GUI text prompt

The next major GUI challenge was how to display the decks when prompting the user for a classification of an archetype. This was implemented by embedding an image file of up to three random decks from a cluster into the display using a program I previously used called

HSDeckToImage, initially developed by former competitor Riku “Yaytears” Miyao, this program has been modified and improved by myself over the course of several years. *Figure 5* shows an example classification screen.

Figure 5 Deck Display Example

The user is then prompted by a text box to give the deck a name, for the displayed deck I would put “OTK” (or one-turn-kill) to describe it, it would then add the class name (“demon hunter” in this case) to the end of it and store it as the cluster name. If the user does not give the deck any name, leaving the text box blank, it will be classified as “Other”.

CHAPTER IV

IMPLEMENTATION

Data Set Generation

The first major algorithm that had to be implemented and developed was a way to generate large amounts of decks that I could use as a basis for my clustering. For this size of dataset, I turned to the Master's Tour Qualifiers that are run by Hearthstone Esports. This set of qualifiers for larger Masters Tours are series of tournaments that run around 24-30 times nearly every weekend through the Battlefy platform. Each tournament contains an average of 600 players bringing three decks each, which supplies large quantities of decks I can use for this. With help from "Yaytears" on finding the specific I was led to the specifics of the Battlefy API that could parse the decklists with the *CSVs/getCSVs* file. The largest issue with this system however is that every tournament requires hundreds of API calls to gather deck lists, ever since Battlefy restructured their API following an incident with the 2019 Master's Tour Las Vegas. This caused parsing a single weekend of tournaments to take nearly an hour to complete when run in series. This is solved using the multiprocessing library and its pool feature, shown in code in *Figure 6*. With the target machines having the 12-Thread i7 processors the default setting is to use 12 processes to send API requests to their server. This method reduces the run time for a single weekend of 30 decks from 24 minutes 23 seconds in serial to 4 minutes 12 seconds in parallel. A near perfect efficiency.

```
p = Pool(processes = NUM_PROCESSSES)
returnData = p.starmap(parseTournament, zip(tournyIDs, getTournyIDs, range(num_tourneys)))
```

Figure 6 MultiProcessing Pool from CSVs/getCSVs.

The *getCSVs* script prompts the user for 2 command line argument dates, that serve as a start and end date to parse from, and then output a CSVs that stores player names, tournament id and deck codes that is usable in the clustering program.

Vectorization

```
(
  [
    (754, 1),
    (38318, 1),
    (40596, 1),
    (41081, 1),
    (42818, 1),
    (43417, 1),
    (64, 2),
    (95, 2),
    (254, 2),
    (836, 2),
    (1124, 2),
    (40372, 2),
    (40523, 2),
    (40527, 2),
    (40797, 2),
    (41929, 2),
    (42656, 2),
    (42759, 2)
  ],
  [274],
  <FormatType.FT_STANDARD: 2>
)
```

Figure 7 Python-Hearthstone decks decoded, note: 274 here is a hero portrait for the deck

The ability to vectorize any deck from only its base64 string is thanks to the python-hearthstone library that converts them into a list of database ids that they represent, these database ids can then be parsed against a card-db found in the hearthstone-data package and hearthstone-json file (both HearthSim creations) that parse the in-game files to get their values in the databases. An example of this list taken from the python-hearthstone GitHub is shown in *Figure 7*.

It is notable here that these codes show the fact that due to *Hearthstone*'s design everything is a "card" according to the game (even the game boards and user interface features)

the [274] here actually represents a portrait for the deck (Lunara of Druid in this case). The tuples found in the list are the various cards with their database ids followed by the number of the card in the deck. Using these numbers several vectors are created to use in the program. *Figure 8* is a card from the *Hearthstone* game.

Figure 8 Goldshire Footman card from Hearthstone

The first portion of the vector is generated by simply counting any available card that could be put in the deck for that class and whether it is there or not. Then more custom vectors are formed based on various aspects of the cards in the deck, such as mana curve (mana is the blue crystal on the card representing its cost), any keywords present on the cards and how many (Bold face text on cards like “Taunt”. Among several other attributes such as card set releases, class card to neutral card ratios, and several common deck building themes such as if the deck contains duplicate cards or not. These deck trope vectors maybe included more than once depending on their variance (no duplicate is very important currently, so it is weighted 100 times more than other vectors). These leave a vector of various dimension but around 1800 for all classes. That can easily be used in any machine learning algorithm or scaler.

Clustering Methods and Classifying Methods

Using the now vectorized decks, decks are now either clustered or classified, depending on the mode the program is running in. When in cluster mode the system will run a KMean's algorithm on the vectors using the SciKit-Learn implementation. This will run and sort decks into clusters using a sum-squared error algorithm to find minimal distances to centroids of clusters. It will then redefine cluster centers and find distances again, this will repeat until minimums are reached. This is shown in *Equation 1*.

$$\arg\min_{\mathbf{S}} \sum_{i=1}^{k} \sum_{\mathbf{x} \in S_i} \left\| \mathbf{x} - \boldsymbol{\mu}_i \right\|^2.$$

Equation 1 K means Clustering

This method is computationally expensive falling in the NP(Hard) category of computer science problems.

For the classification portion of the program, a random forest classifier is implemented using the Scikit-learn implementation. Another NP(Hard) algorithm, this method generates a multitudes of decision trees at training time and returns the mode of all trees for every classification on the algorithm. This use of multiple trees helps account for a single decision trees overfit tendency. This forest is fitted based on the inputted cluster data, and then used to predict all values from the input csv to be classified, as shown in *testClassify.py* in *Figure 9*.

```
model = RandomForestClassifier(n_estimators=500, random_state=0, bootstrap=True, verbose=0, n_jobs=-1)
model.fit(src_features,srcs_labels)
```

```
Y_classified = model.predict(Y)
```

Figure 9 Portions of testClassify.py

These classifications are then sent back for export in both program modes. With with the cluster mode going through each part of the SuperCluster tree to hit every deck, and the classifying simply going thru a list of input decks and classifying them. Shown in *Figure 10* and *11* respectfully.

```
for hero, dataPoints in zip(CLASSES, data):
for id, dataPointIter in dpsInCluster.items():
    clusters.append(Cluster.create(superCluster.CLUSTER_FACTORY, superCluster, id, dataPointIter))
#print(len(clusters)
#print(type(clusters))
classCluster = ClassCluster.create(superCluster.CLASS_FACTORY, superCluster, int(CardClass[hero]), clusters)
classClusters.append(classCluster)
```

Figure 10 Clustering export method

```
for dp, archetype in zip(dataPoints, Y_classified):
    dp.classification = archetype
```

Figure 11 Classification Export Method

Both algorithms are run 10 times each, once on every class in the game. Forming independent classification methods for all the classes.

CHAPTER V

RESULTS

There are many ways to approach analysis of the results for this project and how efficient it performs. The easiest way would be to simply attach the accuracy of each model and end it there. Those numbers can be seen in *Figure 12* for the combined set of all qualifiers for Masters Tour Ironforge and the first 24 qualifiers for Masters Tour Orgrimmar.

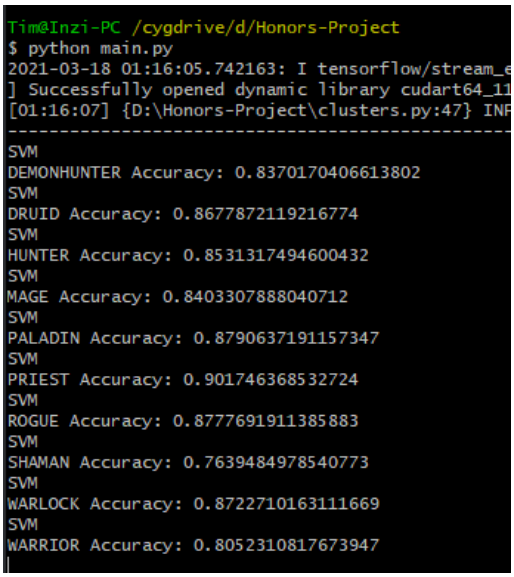

Figure 12 Pure accuracy on dataset from program

However, these numbers, while very good being in the mid-80s on average do not tell the entire story. Take for example the Demon hunter confusion matrix for that dataset, which can be found in *Figure 13*. Notice how the most popular 2 archetypes that account for a large amount of the total decks in the system are accurate. In fact, this becomes even more apparent when you evaluate the results against a manual classification I did on 100 players alphabetically from Masters Tour Ironforge which is shown in *Table 2*. The most popular archetypes are being classified correctly,

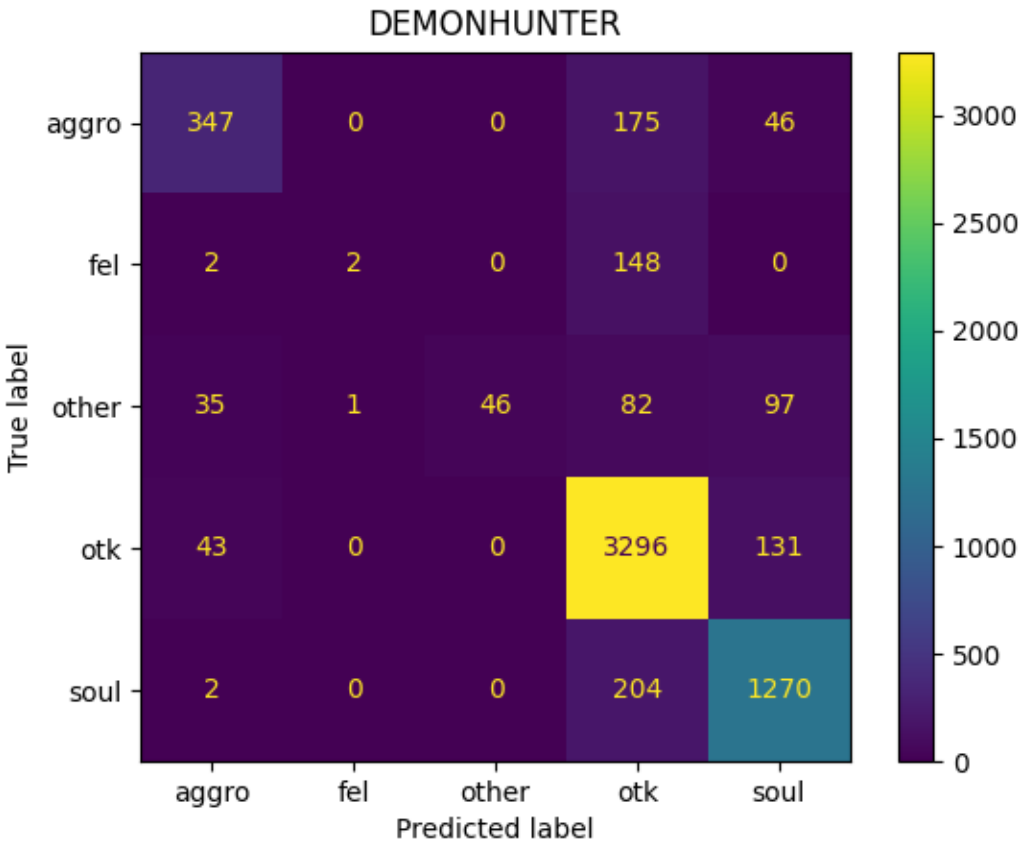

Figure 13 DH Confusion Matrix Generated

True						
	Aggro					
	Fel		1			
	Other				1	2
	Otk				16	
	Soul	2			1	9
		Aggro	Fel	Other	Otk	soul

Predicted

.

Table 2 Manual Confusion Matrix on MT Ironforge

These principles hold true for every other class as well, the most popular archetypes are being identified and a large portion of the inaccuracy is coming from outlier decks. This helps highlight one of the main issues with the K means Algorithm as a whole and that it is very susceptible to outliers.

When applying the algorithm across all classes on a dataset, we find that the total accuracy of the system is around 70% true. Which is far above expected values for this and proves the method to be very successful.

CHAPTER VI

CONCLUSION

This project has provided an extremely interesting way to blend my major in Computer Science with my hobby in the realm of *Hearthstone*. The results I got were better than I expected them to be, and this tool will save countless hours when looking into the future. A setup guide and download are available at the GitHub repository for this project at https://github.com/tim-inzitari/Honors-Project. I hope this tool can help Akron's esports team for as long as the lifespan of *Hearthstone* as an esport. The experience I gained in applying machine learning techniques in this project will also greatly aid my graduate level pursuit of Artificial Intelligence in the future, learning how to vectorize data will especially be useful in this aspect.

CHAPTER VII

FUTURE WORK

The model I developed for this has many areas it can be improved upon including but not limited to the way the decks are vectorized, the clustering mechanics and classification methods. The deck vectorization may benefit greatly from a deeper analysis of what cards are more important than others in certain lists. The difficulty I found in attempting this with this project was crafting it in a way that was scalable across patches. However, I know it is possible if the right algorithm is applied to the dataset. Another improvement could come from the clustering algorithm, the program may work better with a k-centroids variant of k means that can auto merge and reduce cluster counts, again this would be greatly helped with weights being added to a vectorization. Another way that this could be improved is implementing it in TensorFlow on the GPU for a faster run time. This is something I attempted but found the physical limitations on the single 1080TI found in the esports machines could not reach the memory limits where an implementation would surpass the sci-kit learn implementation. I was using an implementation that was modified slightly from a Packt book on TensorFlow 2, this implementation could be improved upon as well to help the TensorFlow version. The last main area I can see for future work on this topic is to better explore more complex neural network systems when applying classification from a label set, while attempting simple neural network systems I could not find a way that would work well enough on every class. A more complex system such as a convolutional neural network may have more success.

Bibliography

BattleFy. (2021, 03 18). *Hearthstone Esports*. Retrieved from Battlefy: https://battlefy.com/hsesports/

Blizzard Entertainment. (2021, 03 18). *Hearthstone*. Retrieved from PlayHearthstone: playhearthstone.com

Cormen, T. (2009). *Introduction to Algorithms.* MIT Press.

Gulli, Kappor, & Pal. (2019). *Deep Learning with Tensorflow 2 and Keras.* Packt.

HearthSim. (2020, September 25). *Python-Hearthstone*. Retrieved from GitHub: https://github.com/HearthSim/python-hearthstone

HearthSim. (2021, 03 18). *Hs Data*. Retrieved from GitHub: https://github.com/HearthSim/hsdata

Hearthstone Esports. (2020, September 25). *Hearthstone Tournament Handbook v2.4 Section 5.6.* Retrieved from Hearthstone Esports: https://bnetcmsus-a.akamaihd.net/cms/page_media/o8/O8QBLEZJUIQA1600999972337.pdf

Miyao, R. (2021, 03 18). *HSDecktoImage*. Retrieved from GitHub: https://github.com/rikumiyao/HS-Deck-to-Image

PySimpleGUI. (2021, 03 18). *PysimpleGUI*. Retrieved from PySimpleGUI: https://pysimplegui.readthedocs.io/en/latest/

University of Akron. (2021, 03 18). *ESPORTS FACILITIES AT UA*. Retrieved from University of Akron: https://uakron.edu/esports/facilities

Weiss, M. (2014). *Data Structures and Algorithm Analysis in C++*. Pearson.

wikimedia. (n.d.). *KMeans Equation.* Retrieved from https://wikimedia.org/api/rest_v1/media/math/render/svg/26f6cae7ba4d68b88acc3dac37c595da607e91f6

Wilmott, P. (2019). *Machine Learning, an Appplied Mathematics Introduction.* Panda Ohana.