Eigenmon : Application of a Modified LSA Algorithm to Measure Similarity of Pokémon Battling Capabilities

Benedict Darrel Setiawan - 13524057
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: benedictdarrel572006@gmail.com , 13524057@std.stei.itb.ac.id

***Abstract*—Being the highest-grossing media franchise of all time, Pokémon's battling mechanics have a deeper complexity than they seem on the surface. The competitive scene behind these games has since determined a majority of each Pokemon's roles in a battle and what their replacements could be. This paper aims to construct an algorithms to find out the similarities between a Pokemon's battling capabilities by taking advantage of a modified Latent Semantic Analysis.**

***Keywords* — Pokémon, VGC, Singular Value Decomposition, TF-IDF, Latent Semantic Analysis**

I. Introduction

Fig 1.1. Pokémon Logo English (https://bulbapedia.bulbagarden.net/wiki/Pok%C3%A9mon)

Currently, in this day and age, Pokémon is considered to be the highest-grossing franchise of all time. Behind that success comes a slew of games that feature the usual battle mechanics you would expect in a turn-based RPG. However, despite seeming simple on the surface, Pokémon's battle mechanics are actually more complex than people give them credit for.

At first glance, Pokémon might just be a game of clicking the strongest moves against your opponent. However, behind every move comes a set of unique effects that might be impactful to the battle, even if the move isn't very strong. And behind every Pokémon comes a set of unique abilities that may help in the face of a disadvantage. Not to mention the type chart that defines the interactions, be it offensive or defensive, between each of Pokémon's 18 types.

Fig 1.2. Pokémon Type Chart (https://pokemondb.net/type)

The appeal of Pokémon's complexity can be seen in the main series games' official competitive scene. Recently, Pokémon's VGC (Video Game Championship) tournaments have seen a massive increase in both interest and competitors, with some tournaments even reaching a record number of over a thousand participants. In these tournaments, participants are required to bring a team of 6 Pokémon, but they may only use up to 4 in each battle. This forces players to be very particular in building their Pokémon teams, in determining their strengths, weaknesses, and also what roles they have in a battle.

These tournaments also usually follow a specific format. These formats differ in the number of legal Pokémon that can be brought to a tournament. For example, in Pokémon Scarlet and Violet's Regulation Set C, only Pokémon that are available in that game's current Pokédex are allowed to be brought to tournaments. Since Charizard isn't in said Pokédex yet, it cannot be brought into tournaments.

These formats really limit the player's ability to build an optimal team since they can limit out a Pokémon that should be the best slot for the specific role that the player is looking for. While more experienced players can easily

identify the best replacement for said Pokémon, it is going to be a more difficult task for newer players in the scene due to the sheer number of Pokémon in these games.

This paper aims to construct an algorithm to determine a list of the most similar Pokémon to a certain one. The measurement of the similarity comes purely out of a Pokémon's battling attributes (moves, stats, type, and abilities) and not any other trivial factors. This algorithm will be constructed by modifying a Latent Semantic Analysis (LSA) to instead analyze a relation between a Pokémon and the role that it may have in a battle.

II. Theoretical Framework

A. Singular Value Decomposition (SVD)

Singular Value Decomposition (SVD) is a factorization method in linear algebra that decomposes a matrix into three other matrices, providing a way to represent data in terms of its singular values.

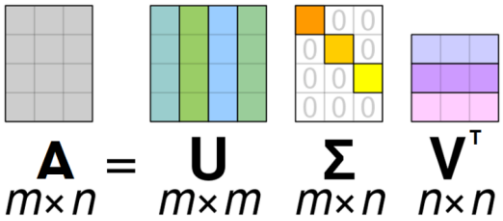


Fig 2.1. Singular Value Decomposition (Source : [1])

SVD splits a m x n matrix into three parts:

- U: Orthogonal m x m matrix
- Σ: An m x n matrix whose main diagonal consists of singular values of the original matrix and 0 for every other element
- V^T: Orthogonal n x n matrix

The steps in performing an SVD are as follows:

1. Compute A^TA
2. Find the Eigenvalues of A^TA
3. Find the Right Singular Vectors (Eigenvectors of A^TA)
4. Compute the Left Singular Vectors (Matrix U) with the formula of $\mathbf{u}_i = A^*\mathbf{v}_i/\sigma_i$

B. Term Frequency-Inverse Document Frequency

TF-IDF (Term Frequency–Inverse Document Frequency) is a statistical method used to weigh a word in a document in relation to a larger collection of documents. TF-IDF combines two components:

1. Term Frequency (TF): Measures how often a word appears in a single document.

2. Inverse Document Frequency (IDF): Measures a term's rarity across the entire collection of documents.

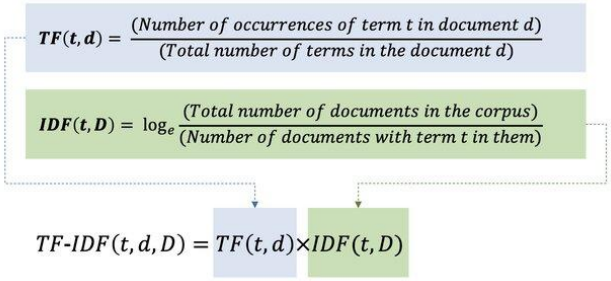


Fig 2.2. TF-IDF Formula (Source: https://www.researchgate.net/publication/376247075_Enhancing_Fake_News_Detection_by_Multi-Feature_Classification/figures)

C. Latent Semantic Analysis

Latent Semantic Analysis (LSA) is a Natural Language Processing (NLP) method that finds hidden relationships (semantics) between words and documents by analyzing word co-occurrence, uncovering underlying topics or concepts. It uses math, specifically Singular Value Decomposition (SVD), to transform a large word-document matrix into a lower-dimensional space, revealing patterns and grouping similar words/documents by meaning, not just exact keywords, making it great for concept searching, document categorization, and information retrieval.

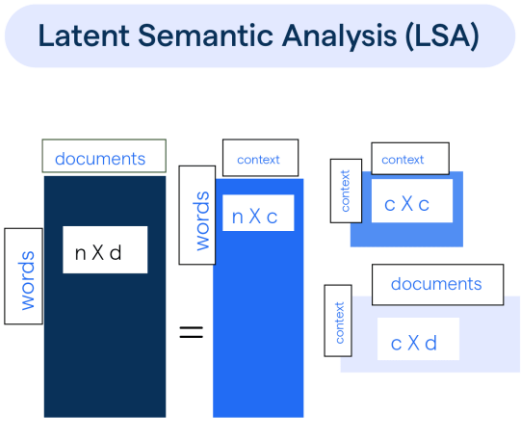


Fig 2.3. Latent Semantic Analysis (Source: [3])

III. Method and Results

A. Preparing the Dataset

Firstly, it is important to prepare the dataset to be used. For this paper, the data is picked from PokéAPI [5], which provides full datasets on every attribute that each Pokémon has.

Since the list of moves and abilities of each Pokémon can differ between games, this paper will limit the analyzed Pokédex to that of the latest main series game,

which is Pokémon Scarlet and Violet. This will also include the Pokédex of both the Teal Mask and Indigo Disk DLC to get the largest consistent dataset as possible.

To further refine the data, the author has also removed inconsequential data, such as forms of a certain Pokémon that are essentially the same as their default form, and also added some missing data regarding the effects of some moves.

B. Finding Similarities in Abilities

The method for finding the similarities in the list of abilities that each Pokémon has is to simply use a standard LSA algorithm. Assuming that every ability listed is unique, the algorithm to create the Ability-Pokémon Matrix are as follows:

```python
def createAbilityPokemonMatrix(pokemon_data) :
    ability_ids = pd.read_csv('data/csv/abilities.csv')['id'].tolist()
    matrix = []
    for i in range(len(ability_ids)) :
        row = []
        for key in pokemon_data :
            if ability_ids[i] in pokemon_data[key].abilities :
                row.append(1)
            else :
                row.append(0)
        matrix.append(row)
    return matrix
```

Fig 3.1. Code for creating Ability-Pokémon Matrix

From there, the next step is to give each of the elements in the matrix a weight. Usually, this is done by using TF-IDF. However, since the frequency of abilities doesn't inherently matter, there is no need to weigh the elements using Term Frequency. So, the only process to do for this matrix before decomposing it is to multiply it by its Inverse Document Frequency, or in this case, the Inverse Ability Frequency.

```python
def inverseFrequency(matrix) :
    ipf = []
    n = len(matrix)
    for row in matrix :
        df = len([i for i in row if i != 0])
        ipf_i = math.log10(n/(1 + df))
        ipf.append(ipf_i)

    return ipf
```

Fig 3.2. Code to find the Inverse Frequency Value

The final step to take in determining the similarities in terms of abilities is to apply the standard LSA algorithm. From there, the values of similarity a Pokémon has to every other Pokémon in terms of abilities can be found by using Cosine Similarity.

```python
def normalizeRows(matrix) :
    normalized = []
    for row in matrix :
        norm = np.linalg.norm(np.array(row))
        if norm == 0 :
            normed_row = [0 for i in row]
        else :
            normed_row = [i/norm for i in row]
        normalized.append(normed_row)

    return normalized

def LRA(matrix, k) :
    _, S, Vt = np.linalg.svd(matrix)
    S_trunc = np.diag(S)[:k, :k]
    Vt_trunc = Vt[:k, :]
    V_trunc = np.transpose(Vt_trunc)
    D = np.matmul(V_trunc, S_trunc)
    D_normed = np.array(normalizeRows(D.tolist()))
    return D_normed

def LRA_to_dict(matrix, k, query, diction) :
    D_normed_type_damage = LRA(np.array(matrix), k)
    sim_val = {}
    for i in range(len(D_normed_type_damage)) :
        if (i != query) :
            sim_i = np.dot(D_normed_type_damage[query, :], D_normed_type_damage[i, :])
            sim_val[list(diction.keys())[i]] = sim_i
    sorted_sim_val = dict(sorted(sim_val.items(), key= lambda items: items[1])[::-1])

    return sorted_sim_val
```

Fig 3.3. Code to find the Cosine Similarity of a certain Pokémon to every other Pokémon in terms of abilities

To make sure the algorithm is correct, it must be ensured that Pokémon that have the same ability as another should be among the top in the sorted list of similarity. The code to test this and the results are as follows.

```python
print("Loading Move Data...")
move_data = loadMovesData()
print("Loading Pokemon Data...")
dex = loadPokemonData()
pokemon_id = '727'
ability_pokemon_matrix = createAbilityPokemonMatrix(dex)
ability_pokemon_matrix = np.matmul(np.diag(inverseFrequency(ability_pokemon_matrix)), ability_pokemon_matrix)
sim_val = cosineSimilarity(ability_pokemon_matrix, len(ability_pokemon_matrix), list(dex.keys()).index(pokemon_id), dex)
for i in range(15) :
    key = list(sim_val.keys())[i]
    print(f"{i + 1}. [{key}] {dex[key].name} : ({sim_val[key]*100}%)")
```

Fig 3.4. Code to print the sorted Cosine Similarity of a certain Pokémon to every other Pokémon in terms of abilities

```
Loading Pokemon Data...
1. [726] Torracat : (100.00000000000003%)
2. [725] Litten : (100.00000000000003%)
3. [10021] Therian Landorus : (65.77772919676737%)
4. [10233] Hisuian Typhlosion : (52.893258476639424%)
5. [499] Pignite : (50.8857195029623%)
6. [498] Tepig : (50.88575195029623%)
7. [156] Quilava : (50.885751950296196%)
8. [155] Cyndaquil : (50.885751950296196%)
9. [157] Typhlosion : (50.88575195029619%)
10. [391] Monferno : (46.829899060009794%)
11. [392] Infernape : (46.82989906000979%)
12. [390] Chimchar : (46.82989906000979%)
13. [500] Emboar : (44.88345725171358%)
14. [911] Skeledirge : (44.15583612061633%)
15. [910] Crocalor : (44.15583612061633%)
```

Fig 3.5. Result of Fig 3.4.

As seen in Fig 3.5., the result of the algorithm shows Pokémon that are similar to Incineroar (pokemon_id: 727) in terms of abilities. Numbers 1 and 2 on the list are Torracat and Litten, which are Incineroar's pre-evolutions that have the exact same abilities as it, which are Blaze and Intimidate [6]. The next Pokémon on the list is Landorus-Therian, whose only ability is Intimidate [7]. And then the rest of the list exclusively contains Pokémon

with the ability Blaze [8]. From this, we can surmise that the algorithm is correct and works properly.

C. Finding Similarities in Base Stats

Similar to finding similarities in abilities, the method to find the similarities between a Pokémon's base stats is to simply put it through a standard LSA algorithm. Firstly, a Stat-Pokemon Matrix needs to be created.

```python
def createStatPokemonMatrix(pokemon_data) :
    matrix = []

    for key in pokemon_data :
        matrix.append(pokemon_data[key].stats)

    return np.transpose(matrix).tolist()
```

Fig 3.1. Code for creating Ability-Pokémon Matrix

Since the Matrix consists of numerical values and not frequencies of a certain term, there is no need to process it through the TF-IDF. Hence, it's already able to be decomposed by using the standard LSA. However, due to the nature of SVD, the cosine similarity will mostly compare the ratio of each of the base stats. To rectify this, another factor contributing to the similarity should be added, that is, the base stat total. To measure the similarity between the base stat total of one Pokémon to every other the following formula is used

$$sim(a,b) = \frac{1}{1 + log(1 + |a - b|)} \quad (1)$$

From there, we can average out the two similarity values to get the true similarity for the Pokémon's stats. To make sure the algorithm is correct, it must be ensured that Pokémon that have the same stat ratio and base stat total as the queried Pokémon need to be near 100% in similarity. The code to test this and the results are as follows.

```python
print("Loading Move Data...")
move_data = loadMovesData()
print("Loading Pokemon Data...")
dex = loadPokemonData()
pokemon_id = "6"
stat_pokemon_matrix = createStatPokemonMatrix(dex)
stat_ratio_sim_val = cosineSimilarity(stat_pokemon_matrix, len(stat_pokemon_matrix), list(dex.keys()).index(pokemon_id), dex)
sim_val = {}
for key in dex :
    if key != pokemon_id :
        bst_sim = 1/(1 + math.log10(1 + abs(np.sum(dex[key].stats) - np.sum(dex[pokemon_id].stats))))
        sim_val[key] = (stat_ratio_sim_val[key] + bst_sim)/2
sim_val = dict(sorted(sim_val.items(), key= lambda item: item[1])[::-1])
for i in range(15) :
    key = list(sim_val.keys())[i]
    print(f"{i + 1}. [{key}] {dex[key].name} : {(sim_val[key]*100)%}")
```

Fig 3.7. Code to print the sorted Similarity of a certain Pokémon to every other Pokémon in terms of base stats

```
1. [157] Typhlosion : (99.99999999999999%)
2. [10233] Hisuian Typhlosion : (99.92364651081974%)
3. [655] Delphox : (99.72860678128993%)
4. [392] Infernape : (99.62324910792158%)
5. [715] Noivern : (87.9502269051536 6%)
6. [474] Porygon-Z : (87.93417860007303%)
7. [884] Duraludon : (86.88357608282209%)
8. [260] Swampert : (86.63442738702261%)
9. [462] Magnezone : (86.60894166941104%)
10. [131] Lapras : (85.9160804979772%)
11. [979] Annihilape : (85.60893850467008%)
12. [464] Rhyperior : (81.67184588318894%)
13. [673] Gogoat : (79.31319099737925%)
14. [254] Sceptile : (79.09685675525004%)
15. [658] Greninja : (78.9222911528317%)
```

Fig 3.8. Result of Fig 3.7.

As seen in Fig 3.8., the algorithm shows Pokémon that are similar to Charizard (pokemon_id: 6) in terms of base stats. Number 1 on the list is Typhlosion, with essentially the similarity of 100%, because it has the same exact base stats as Charizard [9][10]. From this, we can surmise that the algorithm is correct and works properly.

D. Finding Similarity in Defensive Typing

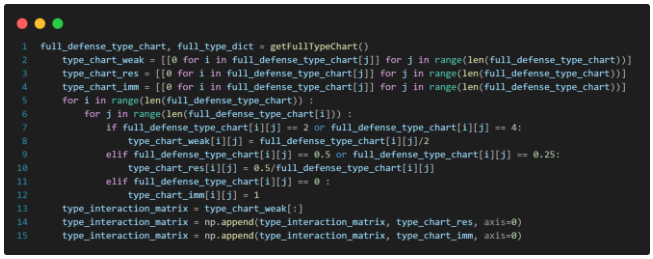
Accurately defining a similarity in a Pokémon's defensive typing has a higher level of complexity than determining the similarities for the abilities and stats. In this paper, the similarity between a typing will be measured based on its interactions with the other types. For example, the defensive typing of Ice/Grass and Bug/Grass will be more similar to each other than that of others, since Ice/Grass and Bug/Grass are both 4 times weak to Fire.

Firstly, a matrix representing the full defensive type chart (including all of the type combinations) needs to be made. This matrix's rows represent the attacking types and its columns represent the defending types. The code snippet to make such matrix is as follows.

```python
def getFullTypeChart() :
    basic_type_chart, basic_type_dict = loadTypeChart()
    defense_type_chart = np.transpose(basic_type_chart).tolist()
    full_defense_type_chart = defense_type_chart
    full_type_dict = dict(basic_type_dict)
    for i in range(len(basic_type_dict.keys())) :
        key_i = list(basic_type_dict.keys())[i]
        name_i = basic_type_dict[key_i]
        for j in range(i + 1, len(basic_type_dict.keys())) :
            key_j = list(basic_type_dict.keys())[j]
            name_j = basic_type_dict[key_j]
            full_type_dict[f"{key_i}/{key_j}"] = f"{name_i}/{name_j}"
            type_combo_row = []
            for a in range(len(basic_type_dict)) :
                type_combo_row.append(defense_type_chart[i][a]*defense_type_chart[j][a])
            full_defense_type_chart.append(type_combo_row)
    full_defense_type_chart = np.transpose(full_defense_type_chart).tolist()
    return full_defense_type_chart, full_type_dict
```

Fig 3.9. Code for creating the Full Type Chart Matrix

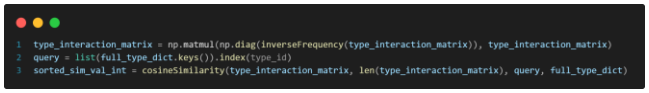
Next, the matrix needs to be expanded so that it contains three different interactions, which are weakness, resistance, and immunity. This is done so that every interaction is treated as its own characteristic and not as a weight on each type. Since in dual types, weaknesses and resistances can be twice as effective, they are weighted as a value of 2 in their respective matrices.



```
full_defense_type_chart, full_type_dict = getFullTypeChart()
    type_chart_weak = [[0 for i in full_defense_type_chart[j]] for j in range(len(full_defense_type_chart))]
    type_chart_res = [[0 for i in full_defense_type_chart[j]] for j in range(len(full_defense_type_chart))]
    type_chart_imm = [[0 for i in full_defense_type_chart[j]] for j in range(len(full_defense_type_chart))]
    for i in range(len(full_defense_type_chart)) :
        for j in range(len(full_defense_type_chart[i])) :
            if full_defense_type_chart[i][j] == 2 or full_defense_type_chart[i][j] == 4:
                type_chart_weak[i][j] = full_defense_type_chart[i][j]/2
            elif full_defense_type_chart[i][j] == 0.5 or full_defense_type_chart[i][j] == 0.25:
                type_chart_res[i][j] = 0.5/full_defense_type_chart[i][j]
            elif full_defense_type_chart[i][j] == 0 :
                type_chart_imm[i][j] = 1
    type_interaction_matrix = type_chart_weak[:]
    type_interaction_matrix = np.append(type_interaction_matrix, type_chart_res, axis=0)
    type_interaction_matrix = np.append(type_interaction_matrix, type_chart_imm, axis=0)
```

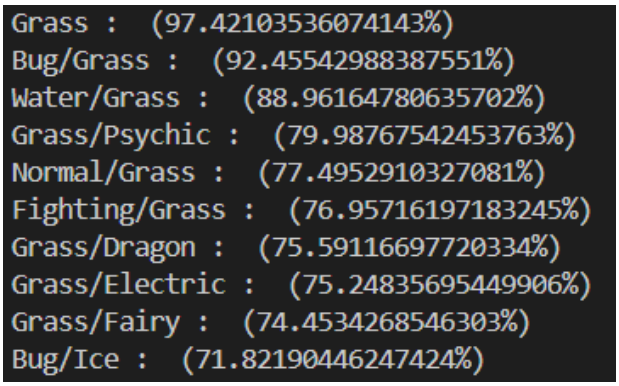
Fig 3.10. Code for expanding the Full Type Chart Matrix

To find the similarities between defensive typings, the expanded Full Type Chart Matrix needs to be processed through the usual Inverse Frequency and LSA algorithm.



```
type_interaction_matrix = np.matmul(np.diag(inverseFrequency(type_interaction_matrix)), type_interaction_matrix)
query = list(full_type_dict.keys()).index(type_id)
sorted_sim_val_int = cosineSimilarity(type_interaction_matrix, len(type_interaction_matrix), query, full_type_dict)
```

Fig 3.11. Continuation of Fig 3.10., Code to find Cosine Similarity of each Defensive typings



```
Grass :  (97.42103536074143%)
Bug/Grass :  (92.45542988387551%)
Water/Grass :  (88.96164780635702%)
Grass/Psychic :  (79.98767542453763%)
Normal/Grass :  (77.4952910327081%)
Fighting/Grass :  (76.95716197183245%)
Grass/Dragon :  (75.59116697720334%)
Grass/Electric :  (75.24835695449906%)
Grass/Fairy :  (74.4534268546303%)
Bug/Ice :  (71.82190446247424%)
```

Fig 3.12. Result of Fig 3.10. and Fig 3.11. with the query of '12/15' (Grass/Ice)

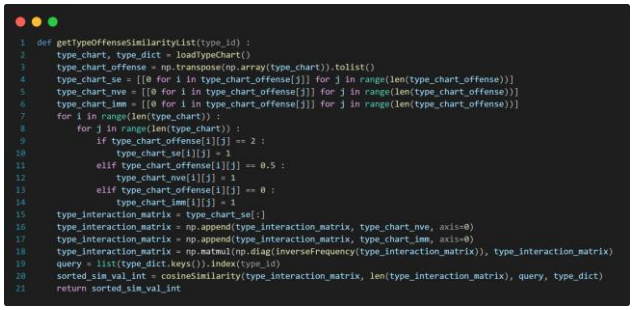
As seen in Fig 3.12. and as explained before, the similarity values for the query Grass/Ice (type_id: '12/15') result in Bug/Grass at number 2 of the list.

To generate similarity between Pokémon based on typing alone, all that needs to be done is to simply take the queried Pokémon's typing as the query for the code in Fig 3.10. and Fig 3.11. and set the similarity values of each Pokémon to the similarity value of their and the queried Pokémon's typing.

E. Finding Similarity Between Move Learnsets

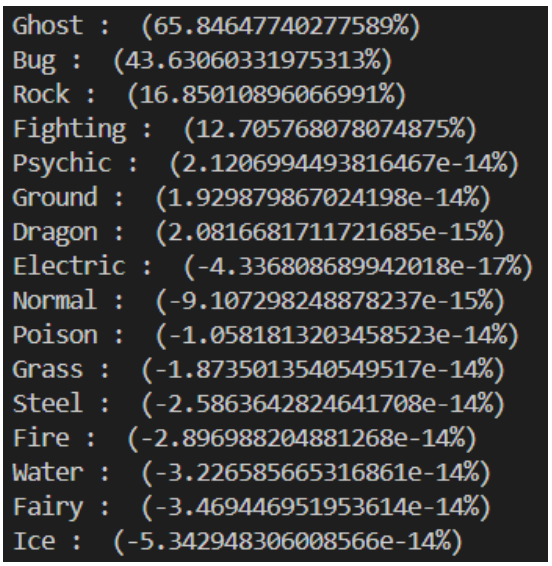
Before finding the similarity values of each Pokémon based on their learnsets, there needs to be an already established similarity value table for each move in the dataset. This is done so that if a move is similar enough to another one, it can act as a replacement in that Pokémon's learnset. Therefore, each attribute of a move should be its own matrix, which can then later be combined into one to find the similarity values.

The first attribute of a move to be defined in a matrix is the move type. Like before, each type needs to be compared with one another, the difference being that this time it is the offensive interactions that are measured. This time, the base matrix's rows will represent the defending types, and the columns will represent the attacking types. Then the matrix will also need to be expanded to contain three different interactions, this time super-effective, not-very-effective, and no-effect. Finally, the matrix will be processed through the usual Inverse Frequency and LSA algorithms to find the queried type's cosine similarity with every other type.



```
def getTypeOffenseSimilarityList(type_id) :
    type_chart, type_dict = loadTypeChart()
    type_chart_offense = np.transpose(np.array(type_chart)).tolist()
    type_chart_se = [[0 for i in type_chart_offense[j]] for j in range(len(type_chart_offense))]
    type_chart_nve = [[0 for i in type_chart_offense[j]] for j in range(len(type_chart_offense))]
    type_chart_imm = [[0 for i in type_chart_offense[j]] for j in range(len(type_chart_offense))]
    for i in range(len(type_chart)) :
        for j in range(len(type_chart)) :
            if type_chart_offense[i][j] == 2 :
                type_chart_se[i][j] = 1
            elif type_chart_offense[i][j] == 0.5 :
                type_chart_nve[i][j] = 1
            elif type_chart_offense[i][j] == 0 :
                type_chart_imm[i][j] = 1
    type_interaction_matrix = type_chart_se[:]
    type_interaction_matrix = np.append(type_interaction_matrix, type_chart_nve, axis=0)
    type_interaction_matrix = np.append(type_interaction_matrix, type_chart_imm, axis=0)
    type_interaction_matrix = np.matmul(np.diag(inverseFrequency(type_interaction_matrix)), type_interaction_matrix)
    query = list(type_dict.keys()).index(type_id)
    sorted_sim_val_int = cosineSimilarity(type_interaction_matrix, len(type_interaction_matrix), query, type_dict)
    return sorted_sim_val_int
```

Fig 3.13. Code to find the Cosine Similarity of each type's offensive interaction



```
Ghost :  (65.84647740277589%)
Bug :  (43.63060331975313%)
Rock :  (16.85010896066991%)
Fighting :  (12.705768078074875%)
Psychic :  (2.1206994493816467e-14%)
Ground :  (1.929879867024198e-14%)
Dragon :  (2.0816681711721685e-15%)
Electric :  (-4.336808689942018e-17%)
Normal :  (-9.107298248878237e-15%)
Poison :  (-1.0581813203458523e-14%)
Grass :  (-1.8735013540549517e-14%)
Steel :  (-2.5863642824641708e-14%)
Fire :  (-2.896988204881268e-14%)
Water :  (-3.226585665316861e-14%)
Fairy :  (-3.469446951953614e-14%)
Ice :  (-5.342948306008566e-14%)
```

Fig 3.14. Result of Fig 3.13., with the query of '17' (Dark)

As seen in Fig 3.14. the most similar type to Dark (type_id: 17) in terms of offensive interactions is Ghost. This is because Dark and Ghost both are super-effective against Psychic and Ghost types, and also are both resisted by Dark types.

From here, the Type-Move Matrix can be made with its rows representing each of the types and in columns representing every move in the dataset. The elements present in the matrix represent a similarity value of one move's type to every other type.

Since the matrix's elements represent similarity, the regular Inverse Function method of weighing each row cannot be used. So, for this specific matrix, the method is modified so that it will only count it as a frequency if the value is exactly 1. This will ensure that the weighing of each move will still be determined by the number of

available moves of that type, while maintaining the similarity values.

```python
def getSimilarityValues(matrix, k, query, diction) :
    itf = np.diag(inverseFrequency(matrix))
    itf2 = np.matmul(itf, matrix)
    D_normed_type_damage = LRA(np.array(itf2), k)
    sim_val = {}
    for i in range(len(D_normed_type_damage)) :
        if (i != query) :
            sim_i = np.dot(D_normed_type_damage[query, :], D_normed_type_damage[i, :])
            sim_val[list(diction.keys())[i]] = sim_i
    sorted_sim_val = dict(sorted(sim_val.items(), key= lambda items: items[1])[::-1])

    return sorted_sim_val
```

Fig 3.15. Code to find the Inverse Similarity Value

```python
type_sim_list = {}
for key in type_dict.keys() :
    type_sim_list[key] = getTypeOffenseSimilarityList(key)
    type_sim_list[key][key] = 1

for key in move_data.keys() :
    move_type_sim_list = []
    for key2 in type_dict.keys() :
        move_type_sim_list.append(type_sim_list[f'{move_data[key].type}'][key2])
    move_type_relation.append(move_type_sim_list)
move_type_relation = np.transpose(move_type_relation)
move_type_relation = np.matmul(np.diag(inverseSimilarity(move_type_relation)), move_type_relation)
```

Fig 3.15. Code to create and weigh the Type-Move Matrix

The next step is to create the rest of the Matrices needed to fully compare each and every move. Since there are some differences in measurement, they will be split into two groups. One that can be counted as frequencies and therefore needs to be weighed through Inverse Frequency or similar means, which includes the Type-Move, Target-Move, Effect-Move, and DamageClass-Move matrices, and one that could be seen as values, such as the Power-Move, Accuracy-Move, EffectChance-Move, and Priority-Move matrices. These two groups' LSA result will then be averaged to get the true similarity values regarding each and every move. The full code snippet regarding this is as follows:

```python
def getMoveSimilarityList(move_id, move_data) :
    move_attribute_matrix = []
    move_type_relation = []
    effect_ids = pd.read_csv('data/csv/move_effects.csv')['id'].tolist()
    target_ids = pd.read_csv('data/csv/move_targets.csv')['id'].tolist()
    _, type_dict = loadTypeChart()


    physical_row = []
    special_row = []
    status_row = []
    power_row = []
    accuracy_row = []
    effect_chance_row = []
    priority_row = []
    type_sim_list = {}
    for key in type_dict.keys() :
        type_sim_list[key] = getTypeOffenseSimilarityList(key)
        type_sim_list[key][key] = 1

    for key in move_data.keys() :
        move_type_sim_list = []
        for key2 in type_dict.keys() :
            move_type_sim_list.append(type_sim_list[f'{move_data[key].type}'][key2])
        move_type_relation.append(move_type_sim_list)
        if (move_data[key].classification == 1) :
            status_row.append(1)
            physical_row.append(0)
            special_row.append(0)
        elif (move_data[key].classification == 2) :
            status_row.append(0)
            physical_row.append(1)
            special_row.append(0)
        elif (move_data[key].classification == 3) :
            status_row.append(0)
            physical_row.append(0)
            special_row.append(1)
        power_row.append(move_data[key].power)
        accuracy_row.append(move_data[key].accuracy)
        effect_chance_row.append(move_data[key].effect_chance)
        priority_row.append(move_data[key].priority + 7)

    effect_matrix = []
    for id in effect_ids :
        effect_row = []
        for key in move_data :
            if move_data[key].effect == id:
                effect_row.append(1)
            else :
                effect_row.append(0)
        effect_matrix.append(effect_row)
    ief = inverseFrequency(effect_matrix)
    effect_matrix = np.matmul(np.diag(ief), np.array(effect_matrix))

    target_matrix = []
    for id in target_ids :
        target_row = []
        for key in move_data :
            if move_data[key].target == id:
                target_row.append(1)
            else :
                target_row.append(0)
        target_matrix.append(target_row)
    move_type_relation = np.transpose(move_type_relation)
    move_type_relation = np.matmul(np.diag(inverseSimilarity(move_type_relation)), move_type_relation)
    move_attribute_matrix = move_type_relation
    move_attribute_matrix = np.append(move_attribute_matrix, effect_matrix, axis=0)
    target_matrix = np.matmul(np.diag(inverseFrequency(target_matrix)), np.array(target_matrix))
    move_attribute_matrix = np.append(move_attribute_matrix, target_matrix, axis=0)
    physical_row = np.matmul(np.diag(inverseFrequency([physical_row])), [physical_row])
    move_attribute_matrix = np.append(move_attribute_matrix, physical_row, axis=0)
    special_row = np.matmul(np.diag(inverseFrequency([special_row])), [special_row])
    move_attribute_matrix = np.append(move_attribute_matrix, special_row, axis=0)
    status_row = np.matmul(np.diag(inverseFrequency([status_row])), [status_row])
    move_attribute_matrix = np.append(move_attribute_matrix, status_row, axis=0)

    move_value_matrix = np.array([power_row])
    move_value_matrix = np.append(move_value_matrix, np.array([accuracy_row]), axis=0)
    move_value_matrix = np.append(move_value_matrix, np.array([effect_chance_row]), axis=0)
    move_value_matrix = np.append(move_value_matrix, np.array([priority_row]), axis=0)
    value_sim_val = cosineSimilarity(move_value_matrix, len(move_value_matrix), list(move_data.keys()).index(move_id), move_data)
    atr_sim_val = cosineSimilarity(move_attribute_matrix, len(move_attribute_matrix), list(move_data.keys()).index(move_id), move_data)
    sim_val = {}
    for key in move_data :
        if key != move_id :
            sim_val[key] = (value_sim_val[key] + atr_sim_val[key])/2
    sorted_sim_val = dict(sorted(sim_val.items(), key= lambda items: items[1])[::-1])
    return sorted_sim_val
```

Fig 3.16. Code to determine a queried move's similarity to every other move

```
Flame Charge : (99.3466032976723%)
Aqua Step : (98.77886885198377%)
Vine Whip : (88.83993023270251%)
Branch Poke : (88.76232995731597%)
Leafage : (88.76232995731597%)
Seed Bomb : (88.10292433460127%)
Rock Throw : (87.94746267970282%)
Wing Attack : (87.80237407503111%)
Peck : (87.66080715593361%)
X-Scissor : (87.48337688212662%)
```

Fig 3.17. Result of Fig 3.16. for the query of '885' (Trailblaze)

As can be seen in 3.17. the most similar move to Trailblaze (move_id: 885) is Flame Charge. This is because Flame Charge and Trailblaze are the same move, with the only factor differing being the type [11][12]. Aqua Step also has the same effect as Trailblaze [13]. However, since it has 20 more power than it, it has a slightly lower similarity value than Flame Charge. From this, it is surmisable that the algorithm is working

properly as intended.

From here, the Move-Pokémon matrix can finally be constructed. Like the Type-Move one, this matrix's elements consist of similarity values, with the value of 1 meaning that it can learn that move, values less than 1 but larger than 0 meaning it can learn a similar move, and the value of 0 meaning it cannot learn that said move or anything similar to it. Due to the sheer number of moves, it is best if only a partial of the most similar moves be inserted into the matrix. For this paper, only the 5 most similar moves are represented in the Move-Pokémon Matrix. Also, due to the sheer number of times the Code from Fig 3.16. will be called, the Move-Pokémon matrix should be stored in a .json file to ensure that future calculations do not take a long time to compute. The code snippet for creating the Move-Pokémon matrix is as follows

```python
def createMovePokemonMatrix(move_data, pokemon_data, k) :
    matrix = []
    saving_to_json = {}
    row_processed = 0
    try:
        with open('data/json/pokemon_move_matrix.json', 'r') as file:
            saving_to_json = json.load(file)
            row_processed = saving_to_json['row_processed']
            matrix = saving_to_json['matrix'][:]
    except FileNotFoundError :
        print("File not found")
        for i in move_data :
            row = []
            for j in pokemon_data :
                if move_data[i].id in pokemon_data[j].moves :
                    row.append(1)
                else :
                    row.append(0)
            first_row = False
            matrix.append(row)

    for i in range(row_processed, len(matrix)) :
        sim_list = getMoveSimilarityList(list(move_data.keys())[i], move_data)
        sim_keys = list(sim_list.keys())[:k]
        for j in range(len(matrix[i])) :
            if matrix[i][j] == 1 :
                for l in sim_keys :
                    if (matrix[list(move_data.keys()).index(l)][j] < sim_list[l]) :
                        matrix[list(move_data.keys()).index(l)][j] = sim_list[l]
        saving_to_json['row_processed'] = i + 1
        saving_to_json['matrix'] = matrix
        try:
            with open('data/json/pokemon_move_matrix.json', 'w') as json_file:
                json.dump(saving_to_json, json_file, indent=4)

            print("Data successfully saved to data.json")

        except IOError as e:
            print(f"Error saving file: {e}")
        print(f"{move_data[list(move_data.keys())[i]].name}")

    return matrix
```

Fig 3.18. Code to create the Move-Pokémon Matrix

The final step in calculating the similarity in learnsets is to weigh the matrix using Inverse Similarity, due to the elements being similarity values, and the LSA algorithm.

```python
print("Loading Move Data...")
move_data = loadMovesData()
print("Loading Pokemon Data...")
dex = loadPokemonData()
pokemon_id = '888'
move_pokemon_matrix = createMovePokemonMatrix(move_data, dex, 5)
move_pokemon_matrix = np.matmul(np.diag(inverseSimilarity(move_pokemon_matrix)), move_pokemon_matrix)
sim_val = cosineSimilarity(move_pokemon_matrix, len(move_pokemon_matrix), list(dex.keys()).index(pokemon_id), dex)
for i in range(15) :
    key = list(sim_val.keys())[i]
    print(f"{i + 1}. [{key}] {dex[key].name} : ({sim_val[key]*100}%)")
```

Fig 3.19. Code to find the Cosine Similarity of Pokémon based on move learnset

```
Loading Pokemon Data...
1. [10188] Crowned Zacian : (99.99999999999997%)
2. [889] Zamazenta : (90.67477121516336%)
3. [10189] Crowned Zamazenta : (89.17102628744851%)
4. [638] Cobalion : (75.45280729921303%)
5. [892] Single Strike Urshifu : (74.15909430423153%)
6. [745] Midday Lycanroc : (73.76374278873516%)
7. [10152] Dusk Lycanroc : (73.27048970637018%)
8. [927] Dachsbun : (72.75914217128872%)
9. [639] Terrakion : (72.71047423625254%)
10. [448] Lucario : (72.66873525725573%)
11. [870] Falinks : (72.00320333792943%)
12. [212] Scizor : (70.7264107268509%)
13. [1014] Okidogi : (70.11387091435164%)
14. [244] Entei : (69.56017659170993%)
15. [1022] Iron Boulder : (69.51328408649499%)
```

Fig 3.20. Result of Fig 3.19. for the query of '888' (Zacian)

As can be seen in Fig 3.20. the most similar Pokémon to Zacian is Crowned Zacian. While being the same Pokemon just different forms, Crowned Zacian has one difference in its learnset, which is that it has access to Behemoth Blade instead of Iron Head [14]. Because Behemoth Blade and Iron Head's differences are only in power and effect, the similarity value is near an exact 100%. Another thing to note is that the number 2 most similar Pokémon is Zamazenta. This is because Zacian and Zamazenta are paired legendary Pokémon, which often have similar move learnsets to one another.

F. Final Similarity Measurement and Results

For the final step in determining the similarity between Pokémon, every similarity value needs to be combined into one singular value. This will be done by averaging out all four of the similarity values measured in this paper. The code that does so and prints out the results is as follows:

```python
def printMostSimilar(sim_dict, dex, k) :
    for i in range(k) :
        key = list(sim_dict.keys())[i]
        sim_val = sim_dict[key]
        if sim_val > 1 :
            sim_val = 1
        print(f"{i + 1}. [{key}] {dex[key].name} ({sim_val*100}%)")

def getPokemonSimilarityList(pokemon_id, dex, move_data) :

    pokemon_move_matrix = np.array(createMovePokemonMatrix(move_data, dex, 5))

    pokemon_ability_matrix = np.array(createAbilityPokemonMatrix(dex), dtype=np.float64)

    pokemon_stat_matrix = np.array(createStatPokemonMatrix(dex))

    ims = np.matmul(np.diag(inverseSimilarity(pokemon_move_matrix)), pokemon_move_matrix)
    iaf = np.matmul(np.diag(inverseFrequency(pokemon_ability_matrix)), pokemon_ability_matrix)
    move_sim_val = cosineSimilarity(ims, 400, list(dex.keys()).index(pokemon_id), dex)
    type_sim_val = getTypeDefenseSimilarityList(dex[pokemon_id].type)
    type_sim_val[dex[pokemon_id].type] = 1
    ability_sim_val = cosineSimilarity(iaf, len(iaf), list(dex.keys()).index(pokemon_id), dex)
    stat_sim_val = cosineSimilarity(pokemon_stat_matrix, len(pokemon_stat_matrix), list(dex.keys()).index(pokemon_id), dex)
    sim_val = {}
    for key in dex :
        if key != pokemon_id :
            bst_sim_val = 1/(1 + math.log10(1 + abs(int(np.sum(dex[pokemon_id].stats)) - int(np.sum(dex[key].stats)))))
            true_stat_sim_val = (move_sim_val[key] + type_sim_val[dex[key].type] + ability_sim_val[key] + (stat_sim_val[key] + bst_sim_val)/2)/4
            sim_val[key] = true_stat_sim_val
    sorted_sim_val = dict(sorted(sim_val.items(), key= lambda items: items[1])[::-1])
    printMostSimilar(sorted_sim_val, dex, 15)

print("Loading Move Data...")
move_data = loadMovesData()
print("Loading Pokemon Data...")
dex = loadPokemonData()
while(True) :
    pokemon_id = input("Input Pokemon ID : ")
    if pokemon_id == '-1' :
        break
    try :
        print(f"Pokemon similar to [{pokemon_id}] {dex[pokemon_id].name}")
        getPokemonSimilarityList(pokemon_id, dex, move_data)
    except ValueError :
        print("Pokemon not found")
```

Fig 3.21. Code to find the most similar Pokémon to the one that is queried

As an example, if we were to query Incineroar (pokemon_id: 727) the results are as follows.

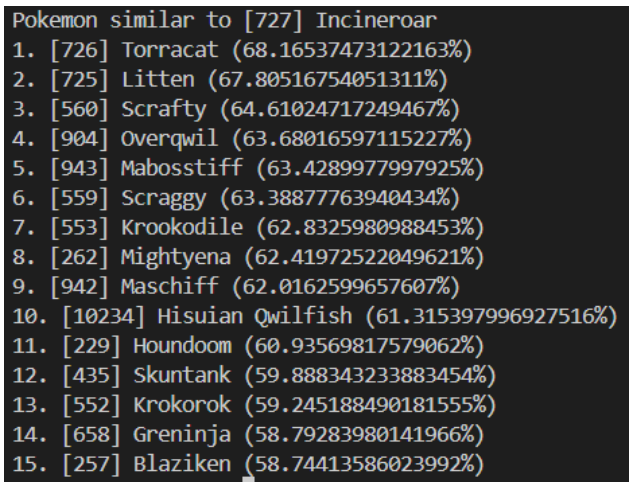


Fig 3.22. Result of Fig 3.21. for the query of '727' (Incineroar)

As seen above, the two most similar Pokémon to Incineroar are its pre-evolutions, Torracat and Litten. This was to be expected since Pokémon in an evolutionary line have overlapping move learsets, typings, and also tend to have the same ability and stat ratios in every single stage. Besides its pre-evolutions, Scrafty seems to be the most similar among every other Pokémon to Incineroar. This is supported by the fact that Scrafty is a Dark type, has the Intimidate ability, has equal Defense and Special Defense, and can learn a lot of Incineroar's moves, such as Fake Out, Parting Shot, Taunt, Throat Chop, Foul Play, and many else [15]. Among the top 15, there are more Dark-type Pokémon with the Intimidate ability, such as Overqwil, Mabosstiff, Krookodile, and Mightyena. There are even some final stages of starter Pokémon like Greninja and Blaziken, which often have nearly the same base stat total as each other.

IV. Conclusion

From this paper, the author has concluded that Pokémon does indeed have latent semantic values behind every move, ability, stat points, and type that it has. Due to this, the author can create an algorithm to search for these values and determine which Pokémon has the most similar role in a battle to another specific Pokémon. This was proven in the case of Incineroar's list of most similar Pokémon, Scrafty, which has been considered as a predecessor to Incineroar before it was released by members of the competitive scene, including a former VGC World Champion [16].

V. Appendix

GitHub repository for the source code made for this research paper:
https://github.com/BenedictD-RH/Eigenmon-Application-of-SVD-in-Measuring-Similarity-in-Pokemon-Battling-Capabilities

VII. Acknowledgment

First of all, the author would like to give his deepest gratitude to the Almighty God for giving him the health and capacity to finish this paper and this semester. The author would also like to thank Mr. Rinaldi Munir for his guidance and teachings on Linear Algebra, which provided the foundations for the concepts used in this paper.

References

[1] R. Munir, "Singular Value Decomposition (SV) (Bagian 1)," Informatika STEI ITB, Bandung, Indonesia. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/AljabarGeometri/2025-2026/Algeo-21-Singular-value-decomposition-Bagian1-2025.pdf. [Accessed: Dec. 23, 2025].

[2] GeeksForGeeks, "Understanding TF-IDF (Term Frequency-Inverse Document Frequency)", GeeksForGeeks [Online]. Available: https://www.geeksforgeeks.org/machine-learning/understanding-tf-idf-term-frequency-inverse-document-frequency/.

[3] BotPenguin, "Latent Semantic Analysis", BotPenguin [Online]. Available: https://botpenguin.com/glossary/latent-semantic-analysis.

[4] GeeksForGeeks, "Latent Semantic Analysis", GeeksForGeeks [Online]. Available: https://www.geeksforgeeks.org/machine-learning/latent-semantic-analysis/.

[5] The PokéAPI project. (2025). "PokéAPI" [Computer Software]. GitHub. https://github.com/PokeAPI

[6] Bulbagarden, "Incineroar", Bulbapedia. [Online]. Available: https://bulbapedia.bulbagarden.net/wiki/Incineroar_(Pok%C3%A9mon). [Accessed: Dec 23, 2025]

[7] Bulbagarden, "Landorus", Bulbapedia. [Online]. Available: https://bulbapedia.bulbagarden.net/wiki/Landorus_(Pok%C3%A9mon). [Accessed: Dec 23, 2025]

[8] Bulbagarden, "Blaze", Bulbapedia. [Online]. Available: https://bulbapedia.bulbagarden.net/wiki/Blaze_(Ability). [Accessed: Dec 23, 2025]

[9] Bulbagarden, "Charizard", Bulbapedia. [Online]. Available: https://bulbapedia.bulbagarden.net/wiki/Charizard_(Pok%C3%A9mon). [Accessed: Dec 23, 2025]

[10] Bulbagarden, "Typhlosion", Bulbapedia. [Online]. Available: https://bulbapedia.bulbagarden.net/wiki/Typhlosion_(Pok%C3%A9mon). [Accessed: Dec 23, 2025]

[11] Bulbagarden, "Trailblaze", Bulbapedia. [Online]. Available: https://bulbapedia.bulbagarden.net/wiki/Trailblaze_(Move). [Accessed: Dec 23, 2025]

[12] Bulbagarden, "Flame Charge", Bulbapedia. [Online]. Available: https://bulbapedia.bulbagarden.net/wiki/Flame_Charge_(Move). [Accessed: Dec 23, 2025]

[13] Bulbagarden, "Aqua Step", Bulbapedia. [Online]. Available: https://bulbapedia.bulbagarden.net/wiki/Aqua_Step_(Move). [Accessed: Dec 23, 2025]

[14] Bulbagarden, "Zacian", Bulbapedia. [Online]. Available: https://bulbapedia.bulbagarden.net/wiki/Zacian_(Pok%C3%A9mon). [Accessed: Dec 24, 2025]

[15] Bulbagarden, "Scrafty", Bulbapedia. [Online]. Available: https://bulbapedia.bulbagarden.net/wiki/Scrafty_(Pok%C3%A9mon). [Accessed: Dec 24, 2025]

[16] Wolfe Glick. What is The Best Dark Type Pokemon? (25 September, 2025). Accessed: Dec 24, 2025. [Online Video]. Available: https://youtu.be/6ldbXM75hfU?si=UIvy_ACWECN6FVd9

STATEMENT

Hereby, I declare that this paper I have written is my own work, not a reproduction or translation of someone else's paper, and not plagiarized.

Bandung, 24 December 2025

Benedict Darrel Setiawan