Arinah Karim, Sam Durm, Sam Wilson
Professor Shih
INFO-I 413
4 May 2023

Final Project Report

DATA COLLECTION

There are two main files we are using for this project. One is *pokemon.csv* and the other is *Trainers.csv*. *pokemon.csv* was a file we found from an existing Kaggle project. This file contains information on pokemon from generation 1-8. *Trainers.csv* was manually created with nearly 4000 instances of trainers. This CSV contains a non-playable character's trainer class and their pokemon team. We used Bulbapedia to fill in the data as it contained the different trainer classes and what teams belonged to the trainer class by generation. We included all battles, including rematches. We excluded trainer classes that were not part of the main games, and limited our collection to be from generation 1-4 instead of 1-8. As far as we know, the *Trainers.csv* is the only file that contains this kind of data in this format; therefore, it would probably be hard to collect all of the data single-handedly, as it took the 3 of us a long time to fill in the data manually.

DATA MANAGEMENT

When pre-processing the *Trainers.csv* file, we had to delete trainer classes that had less than 10 instances. This dropped our trainer classes from summing up to nearly 100 to half of that: 52. Additionally, we had to watch the casing on the pokemon names as the names would act as a key to the *pokemon.csv* data. We had to implement the casing in Python as our casing was incorrect. Additionally, with the help of a function we had created, we could discover where a pokemon name was misspelled in the *Trainers.csv* file. This was expected as there was quite a lot of data. In order to fix the names, we went back into the file and manually changed the names to be the correct name. To ensure that all the classes contained the right spellings, we ran through the entire list of our 52 classes and looked for possible errors. We did not make a copy of the original *Trainers.csv* because it did not make sense to make a copy that contained the errors. To make the data readable, we converted the CSV file to a Pandas DataFrame. We then created a function that would create a dictionary given the trainer class name as the key. It would return a

nested list containing the pokemon teams. This helped greatly with visualizing the elemental types of pokemon found commonly in a trainer class. Because we had created this dataset manually, we did not have to worry about missing values.

When cleaning the *pokemon.csv* file, we dropped columns that we determined would not be useful. Some columns include generation, abilities, classification, base happiness, etc. We did this manually by creating a copy of the *pokemon.csv* file from Kaggle and deleting columns manually. There were a few missing values from some columns. The *Height* column was missing 11 values. The *Weight* column was missing 18 values. Because the three of us were looking at the data together, that was not too big of an amount to just look for the correct heights and weights. Because we had manually coded the *Trainers.csv* file, we also recognized missing heights and weights for pokemon that appeared in our dataset often. We used Bulbapedia to fill in the missing information for all of the pokemon with missing values. While our file contains the corrected data for our usage, there is still an original copy of the csv file from Kaggle that anyone can view at any time.

Additionally, we found some pokemon that had different typing because the Kaggle dataset accounted for generation 8 pokemon, which had pokemon of the same name with different typing. Because there were so few instances of generation 8 pokemon sharing names with generation 1 pokemon, we just double-checked that the typing was correct, and corrected the statistics of the pokemon. We did this manually as we needed to specify which pokemon had which stats, typing, etc. There was also the issue with ensuring that the typing would be represented correctly because a pokemon must have one type but can have an optional second type. Therefore we created a new column ‘typing’ to concatenate the two columns together to visualize the possible combinations of pokemon a trainer class contains.

One pre-processing step we did differently from the midterm is how we standardized the statistics of the pokemon. We intuitively knew that the elemental type of a pokemon would have great influence on what trainer class the pokemon would belong to, but there are some classes that have a greater variety of types than others. Therefore, we wanted to include the statistics (attack, defense, speed, weight, etc.) of the pokemon into our model. However, there was a very wide distribution of data for each of the statistics so we had to use scalers and transformers to figure out how to make the data look more normal or preserve the original distribution and still achieve normality. For the most part, we stuck with power transformations, with a few robust

scalers as well. We did attempt to apply scalers on top of scalers, but that resulted in the loss of data visually, so we only applied one transformation to each statistic. To keep the data frame clean and concise, we only preserved the normalized statistics columns and removed all the other statistic columns that were originally present in the csv or were failed normalization attempts. Something we can take note of is that there is definitely a significance in what transformation is applied to the statistics so after this class ends we want to go back and apply different transformations.

ANALYSIS

To figure out if there are any features that are more important to a certain class than others, stats and typing were plotted against some classes. However, it was quite difficult to check all 52 classes against each other, so instead we decided to do a one-hot encoding of the Pokemon's elemental type and store it as a vector, as well as creating another vector containing the overall statistics of the team. That way, the class can be represented by more than just the elemental types found in the team.

One step we did differently, as mentioned before, was standardizing the data. We first looked into the standard deviations of a given class using the standardized data and took note of the size of the spread of a stat for a given team. By observing the spread, we can determine whether an attribute, such as a team's overall defense stat spread, is within a certain range. We could then use this to figure out which attributes of a pokemon's stats can help identify what class a given pokemon team belongs to. If we could not find a pattern from looking at one class, we can compare it to another class and determine if the difference in standard deviations of a Pokemon team's stats would be statistically significant. Again, because of time, we did not have time to check each class against each other so this would be a future step.

Something else we are doing differently is just solely observing the standard deviations of the teams, rather than the means as we believed the standard deviation to be much more reliable than using the means to represent the entire team. Again, while typing does play a key role in what class a user's team may belong to, we cannot fully rely on typing as a metric as there are classes with great spread in their types they use.

In order to represent this data, we sliced several columns from the *pokemon.csv* file. We then used the pokemon names as data frame headers. We created several data frames that contained only one row of data to make our understanding of it intuitive. For instance, to make a

data frame to analyze the Defense values, we use the entire Defense column from the *pokemon.csv* file and use that as the row value to match with the corresponding pokemon name. We did this for every stat we wanted to observe: attack, special attack, hp, special defense, defense, weight, height, speed, and base total. We then observed the standard deviation of stat values *per team* rather than individually, so we can observe the spread of data for a given team. This would allow us to observe a user's input and look at the team as a whole instead of individually. We also observed the mean stat value of each team to determine if there is an average value for a stat that we can use to determine if a user's input would correspond to a certain class.

We observed the mean Defense stat for each team of the Biker and Fisherman class. We then normalized the Fisherman's and Biker's mean Defense by taking the log of the mean defense lists. We then conducted a t-test to determine if the two were statistically significant. After running the t-test, we found that the p-value was *8.664574051833547e-07* and the t-value was *5.040606963858327* so we can conclude that this isn't due to chance. We did this for several features and attributes to determine how else we can distinguish Trainer classes other than by type. Some concern that we do have is how to normalize all of the data. While we can run t-tests on the data without it being normalized, it can cause issues in reliability of the t-tests. We are currently exploring other methods, such as binning or data augmentation, to normalize data that cannot be normalized as easily as applying a log function on it. We will then determine how to represent the effect of a stat in a one-hot encoding vector in the future.

Something we did want to explore more in the future is grouping our data by elemental types. There is a primary type and a possible secondary type a pokemon can have. We can note that some classes, such as the Fisherman class, have all Water primary types, as well as several different secondary types. While we were not sure how to group the data before this report, this is something we want to investigate more to see if there are more patterns in our data than we are seeing.

When it came to building a model, we tried very hard to implement a MLP-RNN, but we kept running into issues. We kept our implementation of it in the code so we can edit it in the future. Instead, we used a type of RNN called LSTM with Dense layers. First, the training and testing data were split into respective sets and were stratified according to the target variable (i.e., the trainer class) to ensure equal class distribution in both sets. Next, the input variables

(team vectors and team statistics) and target variable (trainer class) were extracted from the training and testing data, respectively, and preprocessed for use in the model. The team vectors were padded using the `pad_sequences` function from TensorFlow. The reason why we did want to use MLP-RNN was to avoid having to pad the Team Vectors, but it was just unavoidable for us. The team statistics were flattened. The target variable was one-hot encoded using the `to_categorical` function from TensorFlow so we could convert it from strings to ints for the model. The model architecture is defined using Keras functional API. The model has two inputs, one for the team vectors and another for the team statistics, and one output for the trainer class. The team vectors input is passed through an LSTM layer with 64 units. The resulting output is concatenated with the team statistics input. The concatenated output is then passed through three dense layers with ReLU activation functions and decreasing numbers of units (128, 64, and 32). The output was passed through a dense layer with 52 units and a softmax activation function to obtain the class probabilities. The model was compiled with Adam optimizer, categorical cross-entropy loss function, and accuracy metric. The model is trained on the training data using the `fit` method, with a batch size of 32, 10 epochs, and validation data consisting of the testing data. The training and validation metrics (loss and accuracy) were printed for each epoch too, which are featured in the Visualization section.

The accuracy of the model was quite poor: 46.54%. However, there were various factors that could have contributed to such a low accuracy. We knew that padding the data would be an issue as there were many teams that had less than 6 pokemon in their team. Additionally, there may not have been enough data for some samples. Something else we could have tried to do is only keep classes with 20 instances or higher. We also had classes that had much more samples than any other class. While we did attempt to stratify the trainer class target variable, it is possible that overfitting was indeed an issue. In the Visualizations section, there are more graphics that make understanding our model output much easier so be sure to look at that section for more information on the data performance.

Overall, there were definitely more steps that we could have taken to make the model improve from 46% accuracy. However, this was originally a 90+ class classification problem turned to 52 class classification. We did want to preserve as much data as we could because we know that some of our audience who played Pokemon games from generation 1 through 4 would have had so much fun seeing how many different NPC trainer classes they could align with.

Maybe we could have taken teams from beyond generation 4 to get more data for some classes. Other classes definitely needed less instances because they had way too many compared to most. There are more preprocessing steps that we can take to improve, but overall we are proud that we were able to get an accuracy not that far from 50%. It definitely is not great, but this project was still enjoyable to do.

INTERVENTION

Our target behaviour of this intervention is to accurately predict the NPC trainer class based on the user's input of their own team of pokemon. While our expected behavior for the model is to analyze the patterns of NPC trainers found in the first four generations of Pokemon games and identify correlations between the NPC trainer classes and the type of Pokemon on a team. Once trained, the model will allow the user to input their own team of Pokemons, and based on the identified correlations, it will predict the NPC trainer class it would belong in. Our form of delivery is by using Django to create a web application where users can input their team of Pokemon and receive a predicted NPC trainer class. The effects of the intervention can be measured by comparing the predicted NPC trainer class with the actual NPC trainer class encountered in the game. This can be done by testing the model on a set known of NPC trainers and comparing the predicted outcome to the actual trainer class. We can also use user feedback to evaluate the accuracy and usability of our application.

DESIGN

For the design of our project, we decided to create a Django application that represents our model through an interface that allows users to interact with our data. Due to our team having an extra member, we felt that this would be an appropriate challenge that best serves our data's purpose. As a team, we consider a few other options including another framework, Flask, as an interface for our project. However, we wanted to challenge ourselves as none of us have worked with Django prior to this project. Using Flask in I211 influenced our design and our approach to creating the Django application, however, we learned quickly that they are a bit different and Django is far more complex. Therefore, we were unable to finish our Django application and its design entirely. We created three functional pages to our web application, the homepage, an about page, and a trainers page. The homepage has no meaningful content, but introduces the project and its purpose through its design. The design is furthered in the

navigation component with a Pokemon logo and the hero banner that displays a pixelated Pokemon image that accompanies our project's intention. We then created an about page that we planned to tell the user more about the team and our project. This was not complete as we found it less important than completing our other tasks, but something our team wants to implement after the semester. Finally, we created a trainers page that allows the user to interact with our data. On this page, we created sections for each trainer class to provide information about each one's functionality that lets users choose a trainer class and receive a Pokemon team that fits that trainer's class given our model. In addition, we display the strengths and weaknesses given the trainer class. Initially, we hoped to create a battle page that allows users to select two trainer classes and battle. The battle would be determined by the strengths and weaknesses of each trainer class and essentially simulate how effective each class is against another. Unfortunately, we were unable to complete this final 'battle' step as time became an issue. Despite this, our team is interested in furthering this project past the semester as we have all found interest in the project and what we have accomplished.

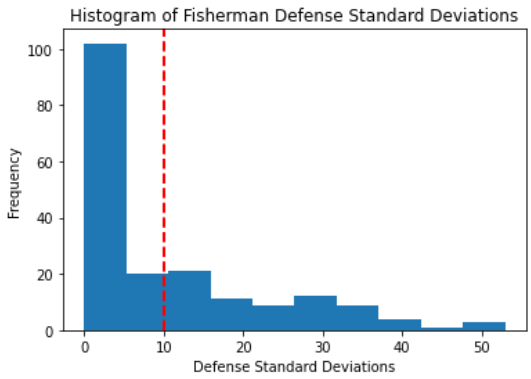
DATA VISUALIZATION

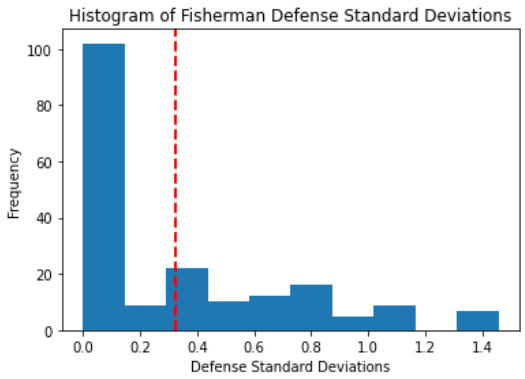
We observed the standard deviations for each team of a given pokemon class. This allowed us to explore the relationship among the different pokemon in a team. Below are the charts for the 'Biker' and 'Fisherman' classes that contain information on the standard deviation of the pokemon teams dependending on which attribute we chose to compare it to. By using these visuals, we can see how pokemon teams within a class vary from each other, as well as see how a class's overall stats are different or similar to another class. In our website, we plan to incorporate a feature that will show the original statistics of each pokemon in a team to show the user more details about the pokemon. We can do this by showing a chart comparing the individual pokemon, or of their pokemon team, to the rest of the distribution of values for statistics of pokemon from generation 1 through 4. We can make these charts interactive by allowing the user to select an individual pokemon from the team and highlight their distributions more than the others.

In order to clearly convey our message, we will be displaying our data through our web application. The website will initially provide details about the course, our team members, and our project on the homepage. In addition, we will have a few pages that allow users to interact

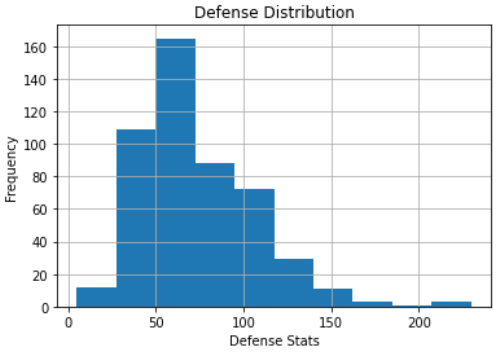
with our data and see its visualizations. Django will connect our python code to HTML and CSS by using queries and python functions that provide specific pages with the necessary data that the user requires. In order to display the data, the HTML tag, canvas, will be used to convert python data into a web readable plot that conveys whatever the user was requesting. This is elaborated more after the charts.

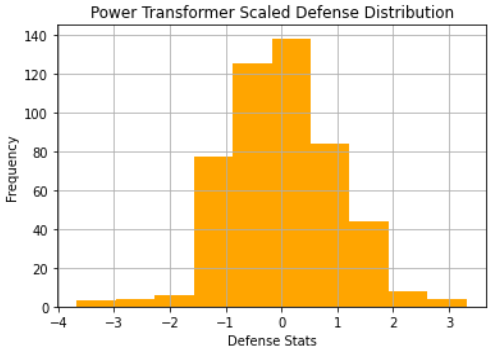
Once again, something we did differently from the previous report was standardize the data. We can observe the difference in range and distribution and see if the visuals help indicate a difference between the biker and fisherman class.





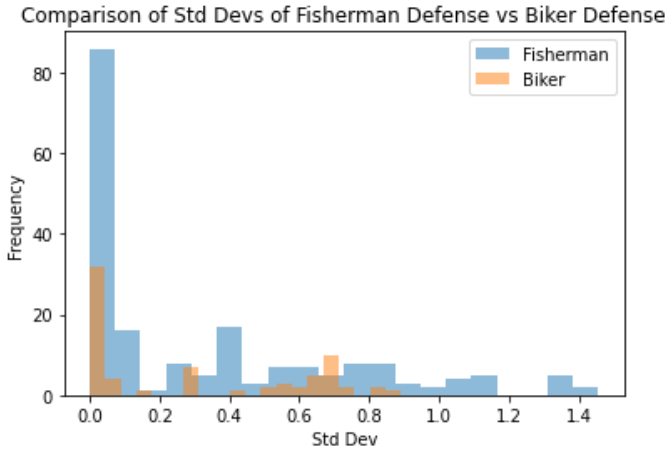
In the above example, we can see a similar distribution of data for the standard deviation of defense for the fisherman trainer class. We can also note that the mean standard deviation, indicated with the red dashed line, is around the same area as before which indicates to us that the distribution of data hasn't changed significantly. We can also see the change in range of values. This will help greatly because of the outliers in the data. Below shows the normalization of the Defense statistic.





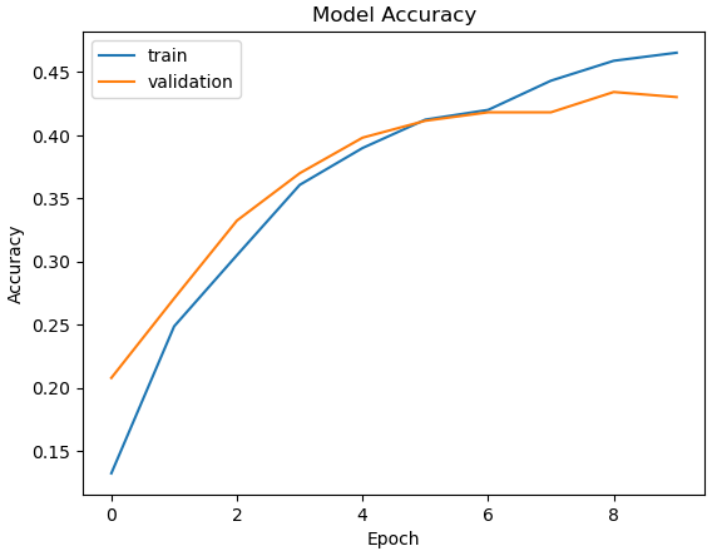
On the left is the original distribution of the Defense stat. We then applied a Power Transformer to it and noted how it became more normalized but it still kept the original distribution somewhat. When going through which standardization method we wanted to use, we graphed the original distribution of the statistic and applied StandardScaler, MinMax, RobustScaler, PowerTransformer, and sometimes Quantile Transformer. To view more transformations, you can view the code and see the various graphs.

Below is a graph that observes the distribution of data for the fisherman and biker classes. We can note the definite skew in the defense data for the fisherman class, while the biker class does not have such a drastic peak. While it would be ideal to check each class, there are 52 classes to check against each other and we unfortunately did not have enough time to do so for this class. However, that's a next step we can do for further preprocessing.



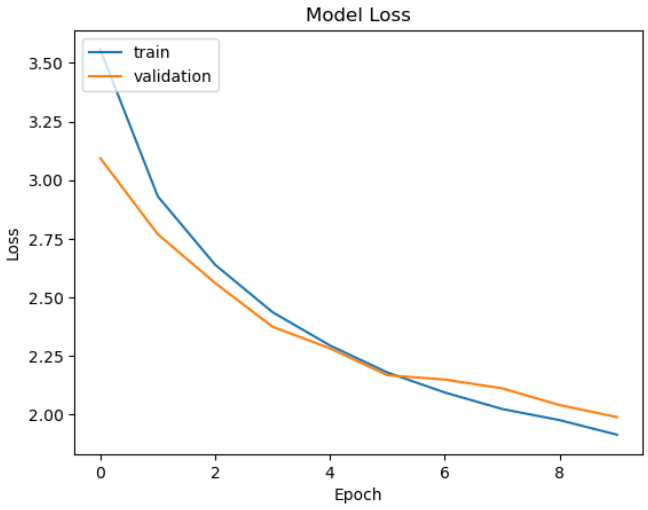
As mentioned from the Analysis section, there are more visualizations here that can help us interpret our model. This code is creating two plots to visualize the training process of the neural network model. The first plot shows the model accuracy over the training epochs, with the blue line representing the accuracy on the training data and the orange line representing the

accuracy on the validation data. The x-axis represents the number of epochs, and the y-axis represents the accuracy score.



From the image, we can see that there could have been overfitting from the first epoch. We can also note that around the 6th epoch, the validation starts to go down so it may have gotten used to the data and could not generalize well to new data.

This next plot shows the model loss over the training epochs, with the blue line representing the loss on the training data and the orange line representing the loss on the validation data. The x-axis represents the number of epochs, and the y-axis represents the loss score.



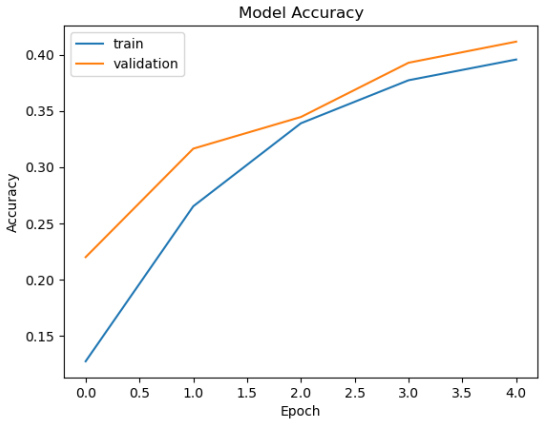
Up until the 5th epoch, we can see that the model was performing better on the validation set than on the training set, which was a good sign. But then the model's performance on the validation set started to degrade compared to its performance on the training set. This may have been caused by the model starting to overfit the data and not generalize well to new data.

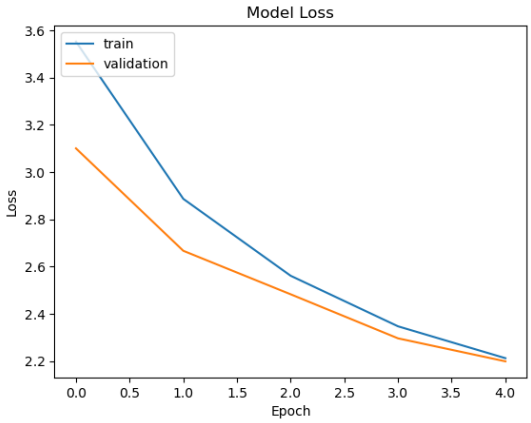
The classification report provides some insight on which classes had good precision, recall, and F-1 scores. We can see that some trainer classes did pretty well overall. For example, trainer class 4 had overall good scores for precision, recall, and F-1. We can also see that there were plenty of classes that had all 0's for these metrics. This could be because there weren't enough instances to train off of. But even some that did have a lot of instances had all 0's as well, such as trainer class 25.

	precision	recall	f1-score	support
0	0.44	0.35	0.39	31
1	0.00	0.00	0.00	10
2	0.00	0.00	0.00	4
3	0.00	0.00	0.00	17
4	0.70	0.92	0.80	36
5	0.00	0.00	0.00	4
6	0.12	0.11	0.12	37
7	0.48	0.63	0.55	62
8	0.00	0.00	0.00	5
9	0.51	1.00	0.67	29
10	0.82	0.47	0.60	19
11	0.75	0.12	0.20	26
12	0.29	0.69	0.41	75
13	0.76	1.00	0.87	26
14	0.57	0.77	0.66	26
15	0.31	0.71	0.43	14
16	0.00	0.00	0.00	12
17	0.00	0.00	0.00	16
18	0.00	0.00	0.00	6
19	0.00	0.00	0.00	8
20	0.65	0.89	0.75	35
21	0.00	0.00	0.00	19
22	0.00	0.00	0.00	2
23	0.40	0.50	0.44	4
24	0.14	0.03	0.06	29
25	0.00	0.00	0.00	2
26	0.45	0.64	0.53	39
27	0.06	0.07	0.06	15
28	0.00	0.00	0.00	2
29	0.75	1.00	0.86	6
30	0.00	0.00	0.00	5
31	0.67	0.57	0.62	7
32	0.00	0.00	0.00	7
33	0.00	0.00	0.00	25
34	0.53	0.44	0.48	18

35	0.07	0.12	0.09	8
36	0.00	0.00	0.00	3
37	0.00	0.00	0.00	4
38	0.12	0.17	0.14	12
39	0.00	0.00	0.00	4
40	0.00	0.00	0.00	4
41	1.00	0.75	0.86	4
42	0.00	0.00	0.00	2
43	0.00	0.00	0.00	2
44	1.00	0.50	0.67	2
45	0.00	0.00	0.00	3
46	0.00	0.00	0.00	4
47	0.00	0.00	0.00	3
48	0.00	0.00	0.00	2
49	0.00	0.00	0.00	5
50	0.00	0.00	0.00	3
51	0.00	0.00	0.00	3

Just to see how reducing the number of epochs would affect the model, we created the same model but with 5 epochs instead.





We can definitely see that there is better performance of the model because there is no overfitting, but the accuracy dropped to 39.56%. We also observed the classification reports and using 5 epochs definitely made the overall metrics much worse than with 10 epochs. Below is the precision, recall, F-1, and support of each model. The first one is for 5 epochs, the second is for 10 epochs

accuracy			0.41	746
macro avg	0.15	0.18	0.15	746
weighted avg	0.33	0.41	0.32	746

accuracy			0.43	746
macro avg	0.22	0.24	0.22	746
weighted avg	0.35	0.43	0.36	746

ETHICS

For the ethics of this project the most relevant ethical values for our design were fairness and integrity. Because of the competition style of the battles between the Pokemon, maintaining fairness in all battles is important. All the Pokemon data is based on the trainer class, and those skills and abilities of each. For Integrity, all data was found from equivalent sites, meaning that all of the same field was found from one site, creating each quality of accuracy of the data. For our program there is no data tracking or gathering of the user, the only intention behind our product is to create an accessible way to test battle Pokemon. Our design portrays these values by the structure of our adjacent website, where users can choose pokemons to battle. This shows our values by the accessibility to all Pokemon and their skills. Due to the topic of our project, the amount of ethical to unethical ways of application are very favored in the ethical ways. Users can learn more about the game by simulating battles and observing how certain trainers battle one another. With the variety of applications and versions of Pokemon, users with access to the internet and a computer will be able to utilize our design and the game. In 2021, Pokemon had over 52 million downloads. Those that are underserved, marginalized, low-resourced, and underrepresented don't have any direct disproportionately affect when it comes to our program, everything is free to use and accessible from any computer with an internet access. Our design has no direct impact on the world's environment, its resources or the climate, all aspects of the projects are digital, including the topic. Both the organization of Pokemon and our group's mission is to create fun through the products and experiences.

DOCUMENTATION

Referenced in the final deliverables folder

code+csvs:

- main.ipynb: contains the source code for the cleaning and implementation, all code can be found in this file

- Trainers.csv: contains all the pokemon npc classes from generation 1-4 and their teams they use; includes duplicate teams because of rematches, all concatenated into one sheet of a csv
- pokemon.csv: a csv from kaggle that contains information about the pokemon, such as statistics (attack, defense, weight, etc.), cleaned for our usage
- Original_csvs
    - og_pokemon_csv: contains all original information from kaggle
    - og_trainers_separated: contains all the trainers, separated by page by member who wrote the team in

Django:

The Django application will be zipped inside of the final_deliverables folder.

- models.py
    - The image below shows the models.py file that is used to create the Django models which is used to create the sqlite database within the project.

```python
from django.db import models
from django.utils.translation import gettext as _

# Create your models here.

# Cite: https://www.youtube.com/watch?v=vs6dXL9Wp7s
class Pokemon (models.Model):
    id = models.BigAutoField(primary_key=True)
    against_bug = models.FloatField(_("Against_bug"))
    against_dark = models.FloatField(_("Against_dark"))
    against_dragon = models.FloatField(_("Against_dragon"))
    against_electric = models.FloatField(_("Against_electric"))
    against_fairy = models.FloatField(_("Against_fairy"))
    against_fight = models.FloatField(_("Against_fight"))
    against_fire = models.FloatField(_("Against_fire"))
    against_flying = models.FloatField(_("Against_flying"))
    against_ghost = models.FloatField(_("Against_ghost"))
    against_grass = models.FloatField(_("Against_grass"))
    against_ground = models.FloatField(_("Against_ground"))
    against_ice = models.FloatField(_("Against_ice"))
    against_normal = models.FloatField(_("Against_normal"))
    against_poison = models.FloatField(_("Against_poison"))
    against_psychic = models.FloatField(_("Against_psychic"))
    against_rock = models.FloatField(_("Against_rock"))
    against_steel = models.FloatField(_("Against_steel"))
    against_water = models.FloatField(_("Against_water"))
    attack = models.IntegerField(_("Attack"))
    base_total = models.IntegerField(_("Base_total"))
    capture_rate = models.IntegerField(_("Capture_rate"))
    defense = models.IntegerField(_("Defense"))
    height_m = models.FloatField(_("Height_m"))
    hp = models.IntegerField(_("Hp"))
    name = models.CharField(_("Name"), max_length=255)
    pokedex_number = models.IntegerField(_("Pokedex_number"))
    sp_attack = models.IntegerField(_("Sp_attack"))
    sp_defense = models.IntegerField(_("Sp_defense"))
    speed = models.IntegerField(_("Speed"))
    type1 = models.CharField(_("Type1"), max_length=255)
    type2 = models.CharField(_("Type2"), max_length=255)
    weight_kg = models.FloatField(_("Weight_kg"))
    is_legendary = models.IntegerField(_("is_legendary"))

class Trainers (models.Model):
    id = models.BigAutoField(primary_key=True)
    trainer_class = models.CharField(_("Class"), max_length=255)
    pokemon1 = models.CharField(_("Pkmn1"), max_length=255)
    pokemon2 = models.CharField(_("Pkmn2"), max_length=255)
    pokemon3 = models.CharField(_("Pkmn3"), max_length=255)
    pokemon4 = models.CharField(_("Pkmn4"), max_length=255)
    pokemon5 = models.CharField(_("Pkmn5"), max_length=255)
    pokemon6 = models.CharField(_("Pkmn6"), max_length=255)
```

    -

- load_pokemon.py
    - The image below shows the load_pokemon.py file that uses the model created above and loads the data into the sqlite database.

```python
from pokemon.models import Pokemon
import csv

# Cite: https://towardsdatascience.com/use-python-scripts-to-insert-csv-data-into-django-databases-72eee7c6a433
def run():
    with open("pokemon.csv") as pokemon_csv:
        reader = csv.reader(pokemon_csv)
        next(reader)

        Pokemon.objects.all().delete()

        for row in reader:
            print(row)

            pokemon = Pokemon(against_bug=row[0],
                              against_dark=row[1],
                              against_dragon=row[2],
                              against_electric=row[3],
                              against_fairy=row[4],
                              against_fight=row[5],
                              against_fire=row[6],
                              against_flying=row[7],
                              against_ghost=row[8],
                              against_grass=row[9],
                              against_ground=row[10],
                              against_ice=row[11],
                              against_normal=row[12],
                              against_poison=row[13],
                              against_psychic=row[14],
                              against_rock=row[15],
                              against_steel=row[16],
                              against_water=row[17],
                              attack=row[18],
                              base_total=row[19],
                              capture_rate=row[20],
                              defense=row[21],
                              height_m=row[22],
                              hp=row[23],
                              name=row[24],
                              pokedex_number=row[25],
                              sp_attack=row[26],
                              sp_defense=row[27],
                              speed=row[28],
                              type1=row[29],
                              type2=row[30],
                              weight_kg=row[31],
                              is_legendary=row[32]
            )
            pokemon.save()
```

    -
- views.py
    - The image below shows the views.py file which creates the url patterns that load the Django templates based on user interaction and send any necessary data to the accompanying templates.

```python
from django.shortcuts import render
from django.urls import reverse
from django.template.loader import render_to_string, get_template
from django.http import Http404, HttpResponseNotFound, HttpResponseRedirect, HttpResponse
from .models import Pokemon, Trainers
import csv

# Create your views here.
def index(request):
    return render(request, "pokemon/index.html")

def about(request):
    return render(request, "pokemon/about.html")

def trainers(request):
    # trainer_classes = Trainers().objects.all()
    trainer_classes = []
    with open("trainers.csv") as trainers_csv:
        trainer_list = csv.reader(trainers_csv)
        for row in trainer_list:
            trainer_classes.append(row)
    return render(request=request, template_name="pokemon/trainers.html", context={'trainer_classes': trainer_classes})

```

- base.html
    - The image below shows the base HTML template that extends to all other pages in the Django application.

```html
{% load static %}
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <title>{% block title %}{% endblock %}</title>
        <link rel="stylesheet" href="https://fonts.googleapis.com/css2?family=Material+Symbols+Outlined:opsz,wght,FILL,GRAD@20..48,100..700,0..1,-50..200" />
        <link rel="stylesheet" href="{% static "pokemon/includes/nav.css" %}">
        <link rel="stylesheet" href="{% static "styles.css" %}">
        {% block css_files %}
        {% endblock %}
    </head>
    <body>
        {% include "pokemon/includes/nav.html" %}
        {% block content %}
        {% endblock %}
    </body>
</html>
```

- nav.html
    - The image below shows the navigation component that exemplifies the use of templates and components within a Django project. In the code, you can also see how to link images, CSS files, and other url patterns within a Django project.

```
{% load static %}
<header>
    <nav>
        <img class="logo" src="{% static 'images/logo.png' %}" alt="Pokeball Logo">
        <ul id="menu">
            <li class="menu-item"><a href="{% url 'index' %}">Home</a></li>
            <li class="menu-item"><a href="{% url 'about' %}">About</a></li>
            <li class="menu-item"><a href="{% url 'trainers' %}">Trainers</a></li>
            <li class="menu-item">Battle</li>
        </ul>
        <span id="menu-btn" class="material-symbols-outlined">
            menu
        </span>
    </nav>
</header>
<script>
    let menu_button = document.querySelector('#menu-btn');
    menu_button.addEventListener({
        if (menu_button.classList.contains('open') {
            menu_button.classList.remove('open');
        } else {
            menu_button.classList.add('open');
        }
    });
</script>
```

-

- nav.css
    - Finally, the image below shows one of the many CSS files used in the application.

```css
header {
    padding: 0;
    margin: 0;
}

nav {
    background-color: black;
    display: flex;
    flex-flow: row nowrap;
    justify-content: space-between;
    align-items: center;
    position: fixed;
    top: 0;
    height: 75px;
    width: 100%;
}

/* Nav Logo */
.logo {
    width: 50px;
    padding-left: 5.0rem;
}

/* Nav Items */
ul {
    display: flex;
    flex-flow: row nowrap;
    justify-content: space-evenly;
    width: 40%;
}

.menu-item {
    list-style: none;
    color: white;
}

.menu-item a {
    color: white;
    text-decoration: none;
}

.material-symbols-outlined {
    display: none;
    height: 48px;
    color: white;
}

.material-symbols-outlined {
    font-size: 40px;
    color: white;
    padding: 20px;
    font-variation-settings:
        'FILL' 0,
        'wght' 0,
        'GRAD' 0,
        'opsz' NaN
}

/* Media Queries */
@media screen and (max-width: 560px) {
    #menu {
        display: none;
    }
    .material-symbols-outlined {
        display: block;
    }
}
```

○