Final Project Report: Pokémon Evolution

CECS 456

Kenny Vu, Raymond Lo, Briant Anaya, Ramon Delgadillo

We chose a Pokémon dataset since it is a popular franchise and fun to work with. This dataset consists of hundreds of unique pokémon's and their names, type, and stats such as HP, Attack, and more. We decided to add a new column to this dataset called “Evolution Stage”, which is our target. The rest of the data, which consists of HP, Attack, Defense, S.Atk, and Sp.Def, is our features. Our primary goal is to determine what stage evolution a pokemon is based on their given stats. For example, a pokemon with very high stats would most likely be considered a pokemon that has evolved into a higher state. We will be applying several different models such as Logistic Regression, Random Forest, and Neural Network and comparing the performance of those algorithms based on their results. Of course we start by processing the data. We open the csv file with our newly appended column called “Evolution Stage”, and drop unnecessary columns such as the pokemon’s type and name as we will not be needing this for determining the evolution stage. After that, we split the data into their train and test sets. The first model we used is Logistic Regression. We import a Logistic Regression classification model from sklearn, which uses multi-class classification by default. We also import and use StandardScalar to standardize the features so that it transforms the data to have a zero-mean and unit variance. Unsurprisingly, we found that without feature scaling or normalization, the data becomes heavily skewed and difficult to use. In order to train our Logistic Regression model, we need to initialize the model and fit it to the training data. Next, we use the trained model to make predictions on the test set. Afterwards, we evaluate the model by using an accuracy_report object and a classifcation_report object. The figures below demonstrate how we set up our logistic regression model and what the results were.

Accuracy: 0.4789915966386555

	precision	recall	f1-score	support
1	0.52	0.77	0.62	123
2	0.36	0.21	0.27	85
3	0.17	0.03	0.06	30
accuracy			0.48	238
macro avg	0.35	0.34	0.32	238
weighted avg	0.42	0.48	0.42	238

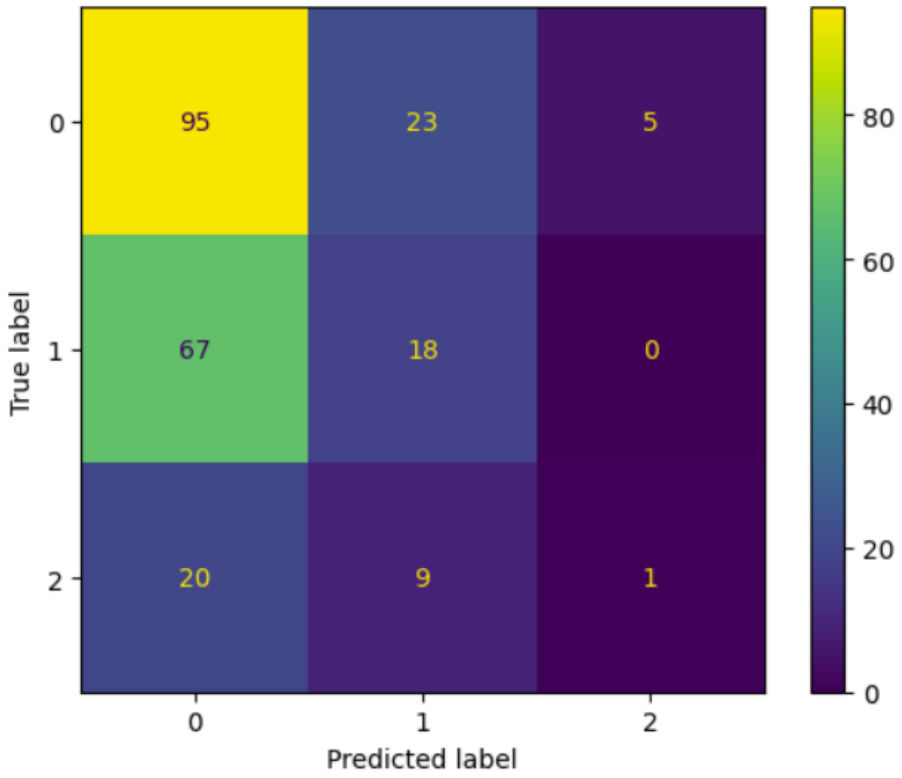
```
lrm = LogisticRegression(multi_class = 'ovr', max_iter = 1000)
lrm.fit(X_train, y_train)

y_pred = lrm.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)

# Print classification report
print(classification_report(y_test, y_pred))
```

Our results show an accuracy of 0.47 to 0.48. This could be due to a lack of features in our dataset. Because of this, it's very difficult to classify our data especially when there are only 3 distinct labels. There is not much correlation among features either to employ feature engineering. Overall, it's difficult to classify data when there are not many features to consider and not much correlation amongst the features to distinguish between classes. Nonetheless, our confusion matrix reported a high accuracy for stage evolution 1 pokemon but falters when classifying stage 2 and stage 3 evolutions. (aka class 2 and class 3).



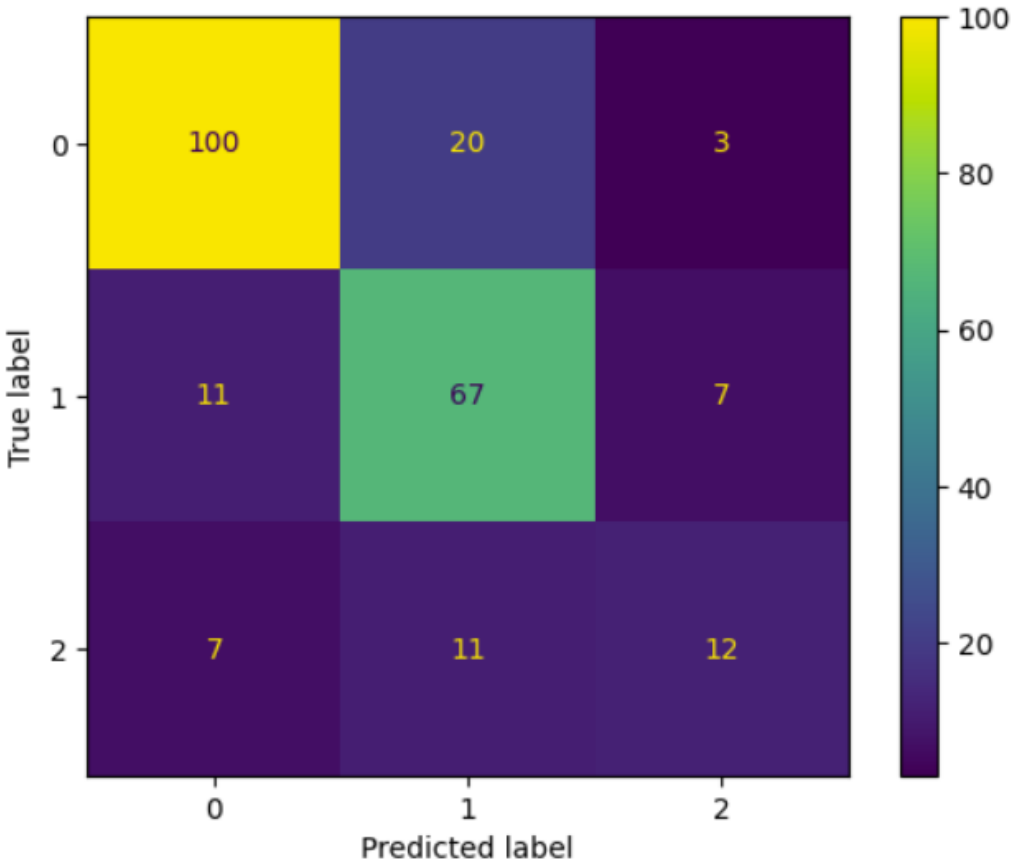
For our Random Forest model, similar to Logistic Regression, we need to initialize the model and fit it into the training data, which we split earlier. We used the trained model to make predictions on the test data and evaluate the model's performance by using the same accuracry_report and classification report objects we used for Logistic Regression. However, our Random Forest model performed much better using the same dataset. Below are the results of the Random Forest Model.

```
Accuracy: 0.7605042016806722
              precision    recall  f1-score   support

           1       0.88      0.81      0.85       123
           2       0.69      0.80      0.74        85
           3       0.50      0.43      0.46        30

    accuracy                           0.76       238
   macro avg       0.69      0.68      0.68       238
weighted avg       0.77      0.76      0.76       238
```

As you can see, the results show an accuracy of 0.76. The Random Forest model classified stage evolution 1 pokemon very accurately. The model also predicted stage evolution 2 pokemon well, especially compared to Logistic Regression.



Finally, for our Neural Network model, we imported the necessary Library TensorFlow in order to use the Keras API to help build and train the model. By having access to Keras, we are able to use the Sequential API to help create the architecture for the Neural Network. Then, we compile the model by using the Adam optimizer and loss function. After that, we used the trained neural network model to make predictions on the training data using the softmax function. We then calculate the confusion matrix using TensorFlow's 'tf.math.confusion_matrix' function with the result converted into a NumPy array. Below are the results from the Neural Network model.

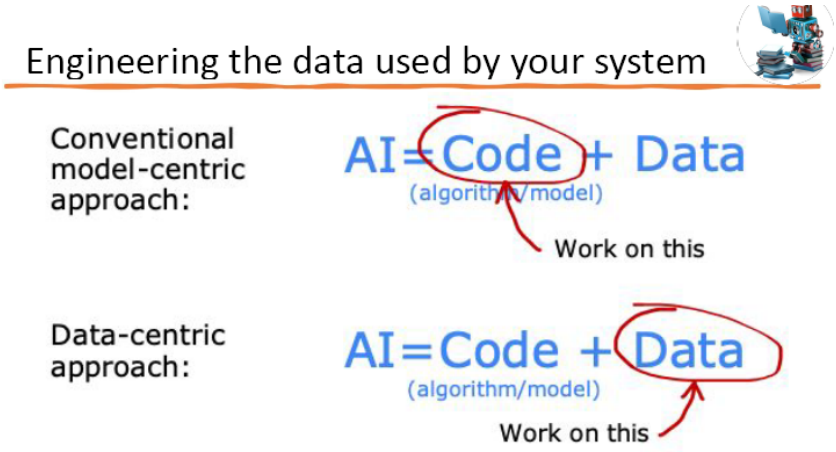
```
30/30 [==============================] - 0s 1ms/step
[[310 194  63]
 [175 114  46]
 [ 23  20   7]]
Accuracy on test data set:  0.4527310924369748
```

- Confusion Matrix for NN.

As you can see, the result shows an accuracy of 0.45. It seems as if the Neural Networks model has performed the worst out of the three models.

Overall, the Random Forest has performed better than Logistic Regression partly because it is better at catching non-linear relationships and uses fewer hyperparameters compared to Logistic Regression. At first, we thought our dataset has a decent linear relationship with each other, since stats growth usually affects each other and growing one means others will grow as well. However, that does not seem like the case. As for the Neural Networks model, it requires much more data for the model to accurately make predictions or classifications, we would have to employ the use of data engineering or similar techniques to increase our dataset for this model to work.

The general consensus, for our case, is that no matter what model we use, we can't seem to improve the accuracy of the models unless we alter the current dataset.



For resampling technique, we tried undersampling, oversampling, and grid search CV for hyperparameter tuning technique. First, we employed undersampling since we found that there are much more stage evolution 1 pokemon as compared to stage evolution 2 and 3 pokemon. To offset this, we lowered the number of stage evolution 1 & 2 pokemon to match more closely to the amount of the other stage evolutions. After undersampling and running our 3 models, we found that the general result turned out to be bad. This is as expected since we didn't start out with a lot of data to begin with, so when we scaled other classes to match label "3", we lost a lot of data and thus made all the models perform worse. On the other hand, oversampling yields good results for most of our models. Though NN only improved by 5% from an average of 45%, random forest and plain logistic regression improved significantly. For random forest the average of 76% went to a stable 89%. The logistic regression didn't really improve unless we apply grid search cross validation to it. By itself, GridSearchCV is ineffective with our models, it is expected since we can only tune so much with the limited data variations.

```
from sklearn.utils import resample
# Identify the subset of data corresponding to stage 3 evolution
stage_3_data = data[data['Evolution Stage'] == 3]

# Undersample the majority class (stage 1 and 2) to match the minority class
undersampled_majority_data = resample(data[data['Evolution Stage'] != 3], replace=False, n_samples=len(stage_3_data), random_state=42)

# Combine the undersampled majority data with the stage 3 data
undersampled_df = pd.concat([undersampled_majority_data, stage_3_data])

# Shuffle the combined dataset to mix the undersampled majority data and stage 3 data
undersampled_df = undersampled_df.sample(frac=1, random_state=42)

print(undersampled_df)

# Load the Iris dataset as an example
X = undersampled_df.iloc[:,: -1] # read input columns
y = undersampled_df.iloc[:,-1]

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size= 0.2, random_state=42)
```

The accuracy came back much worse at .32 or 32%. We surmise that reducing data for our dataset is not particularly helpful when our data is already limited in scope.

```
2/2 [==============================] - 0s 3ms/step
[[ 6  5 11]
 [ 6  2  9]
 [ 5  6  9]]
Accuracy on test data set:  0.288135593220339
```

- Undersampling result

However, after doing the opposite, which is oversampling, we found the results improved somewhat moderately.

```
12/12 [==============================] - 0s 1ms/step
[[ 83  36  42]
 [  2   2   0]
 [ 49  42 109]]
Accuracy on test data set:  0.5315068493150685
```

Results improved with an accuracy of .53 or 53%. Interestingly enough and unsurprising at this stage of our project, the confusion matrix shows that stage evolution 2 and 3 still get confused for other classes while stage evolution 1 gets predicted accurately enough. Overall, oversampling does improve results for all of our models besides the NN. Both undersampling and oversampling increased our accuracy on the Logistic Regression model using GridSearchCV. Oversampling increased the accuracy for Logistic Regression (with grid search CV) by about 20 percent alone, which came out to be ~65%. Oversampling also improved Random Forest with results going from 76% to 89%

The lack of features and feature engineering, however, still holds results back regardless if we oversample due to the very nature of our data. As stated before, the features or stats of pokemon do not have a strong correlation as to what stage evolution a pokemon may be in, so new or combined features would be needed. The only feature that is in some way indicative of a pokemon's evolution stage is the "Total" feature, which is the sum of all the pokemon's stats, but that alone is not enough.