Performance of Classification Methods in Pokémon Type Prediction

Allen Zheng, Jerry Zhen, Justin Xu, Ryan Wu – University of Waterloo

1 Summary

Predicting a Pokémon's type is an inherently difficult task, with 18 possible categories and no clear pattern. Therefore, there is great appeal in applying new methods in an attempt to achieve great accuracy. In this report, we apply classification methods such as logistic regression, support vector machines (SVMs), random forests, and neural networks to an enhanced version of the Pokémon Database dataset. For each model, we perform hyperparameter tuning using Bayesian Optimization and evaluate the tuning process using stratified nested cross-validation. We find that the enhanced features allow all models to achieve near-perfect accuracy. Thus, we turn our attention to a subset of the features to better compare the models. Further, to deal with sparse type categories, we investigate a "duplication" method for multi-label prediction, where a single-label classifier is trained on two copies of each Pokémon corresponding to each of the Pokémon's types. We discover that some types are easier to classify than others, particularly Dragon. The classifiers overall achieve very similar prediction accuracy for duplication as for primary-type classification.

2 Introduction

Pokémon is one of the most popular games worldwide. In it, players catch, train, and battle using monsters known as Pokémon. In particular, each Pokémon has one or two types such as the elemental type "Water" or the abstract type "Fighting". Each type has advantages and disadvantages against others in battle. As a result, Pokémon players share great interest in using effective types to win battles. Many have attempted to predict a Pokémon's type using other attributes. We would like to compare the methods they used in a standard setting, however their datasets contain different observations and features. Here, we use an improved data set with a more diverse feature space. Using a standard nested cross-validation method, we compare the performance between different models.

3 Dataset

3.1 Data Scraping

Using the Pokédex Dataset as an inspiration, we created our own scrapers with Python libraries `requests`, `beautifulsoup`, and `imageio` to scrape data from the website Pokémon DB. We chose numerical features, or categorical features with a small number of different values. The scraped attributes include biometrics, combat statistics, and training characteristics as well as the `weaknesses` (damage multipliers against other types) of each Pokémon. To augment this data, we scraped images of each Pokémon and computed simple characteristics of the image to serve as additional attributes. A detailed discussion on data processing can be found in Appendix.

3.1.1 Categorical and Binary Features

Feature	Description
status	Whether the Pokémon is normal, legendary, sub-legendary, or mythical
type_1	The primary type of the Pokémon

Feature	Description
type_2	The secondary type of the Pokémon if it exists
has_gender	Whether the Pokémon has a gender

The types include: Bug, Dark, Dragon, Electric, Fairy, Fighting, Fire, Flying, Ghost, Grass, Ground, Ice, Normal, Poison, Psychic, Rock, Steel, Water. See Appendix for Pokemon Type Frequency

3.1.2 Numerical Features

Feature	Description
generation	The chronological divisions of Pokémon, from 1st generation to 8th
type_number	The number of types the Pokémon belong to, either 1 or 2
height_m	The height of the Pokémon in meters
weight_kg	The weight of the Pokémon in kilograms
abilities_number	The number of abilities possessed by the Pokémon
total_points	Total number of base points
hp	The base health points (hp) of the Pokémon
attack	The base attack of the Pokémon
defense	The base defense of the Pokémon
sp_attack	The base special attack of the Pokémon
sp_defense	The base special defense of the Pokémon
speed	The base speed of the Pokémon
catch_rate	Catch rate of the Pokémon
base_friendship	The base friendship of the Pokémon
base_experience	The base experience of a wild Pokémon when caught
maximum_experience	The experience needed for the Pokémon to reach the maximum level
egg_type_number	The number of egg groups the Pokémon egg belongs to
proportion_male	The proportion of Pokémon that is male, 50% if genderless
egg_cycles	The number of cycles required to hatch an egg of the Pokémon
damage_from_**Type**	The damage multiplier when damaged by the move from a **Type**

3.1.3 Image Features (all numerical)

Feature	Description
size	The proportion of pixels occupied by the Pokémon's sprite
perimeter	The number of pixels occupied by the sprite's boundary
perimeter_size_ratio	The ratio of the sprite's perimeter to its actual size
Value_mean	The mean of the **Value** pixel value over the entire sprite
Value_sd	The standard deviation of the **Value** pixel over the entire sprite
vertical overflow	The amount by which the sprite touches the boundaries of the image vertically
horizontal overflow	The amount by which the sprite touches the boundaries of the image horizontally

Value = **Red**, **Green**, **Blue**, **Brightness**

4 Previous Explorations

***Below**: Previous explorations of the dataset, with models (**DT**: Decision Tree, **kNN**: k-Nearest Neighbours, **LR**: Logistic Regression, **NB**: Naive Bayes, **NN**: Neural Network, **RF**: Random Forest, **SVM**: Support Vector Machine) and multilabel classification methods (label powerset, binary relevant, duplication)*

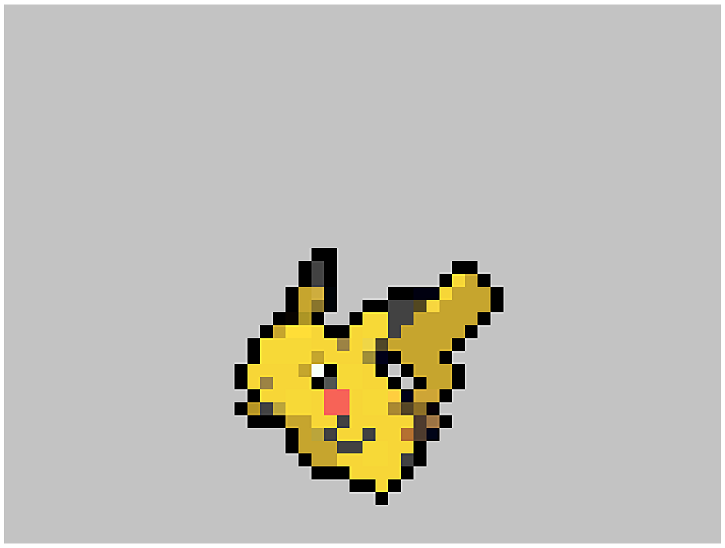


Figure 1: Pikachu. generation: 1, status: Normal, type_number: 1, type_1: Electric, type_2: None, height_m: 0.4, speed: 90, catch_rate: 190, damage_from_water: 1, damage_from_ground: 2, sprite_red_sd: 0.414

Features	**Single-Label Classification**	**Multi-Label Classification**
Pokédex	**Hsiao, 2021**: DT	**Ezeilo, 2018**: kNN; label powerset
	kn-kn, 2017: NB, SVM, RF	
Sprite	**Soares, 2017**: NN	**Shahir, 2022**: NN; binary relevance
		Zahroof, 2021: NN; binary relevance
Text		**Belfer, 2021**: NN
Pokédex/Sprite	**Ours**: LR, SVM, DT, RF, NN	**Ours**: duplication

- (Wei Chen 2021): Simple analysis of the pokemon dataset from generations 1-8. Trying to predict `type_1` using pokemon stats, generation number, as well as mythical/legendary status.
- ((kn-kn). 2017): Trying to predict pokemon `type_1` using naive bayes, random forest, and SVMs. Since flying type is severely underrepresented, they remove it and predict only 17 types. Uses pokemon stats, generation number, and legendary status. Additionally used grid search for hyper-parameter tuning, cross-validation, and leave one out cross-validation.
- (Soares 2017): Trying to predict dual pokemon types (`type_1` and `type_2`) using in game sprites from generation 1-5. Utilizes CNN for classification.
- (Ezeilo 2018): Trying to predict dual pokemon types (`type_1` and `type_2`) using KNN. For pokemon with only one type, they imputed `type_2` as a repeat of `type_1`, however only predicted for one observation.
- (Shahir 2022): Trying to do multi-label classification with NN in TensorFlow using the in game sprite images.
- (Tan 2022): Multi-label classification using MultiLabelBinarizer, basically one-hot encoding the 18 types and thus allowing it to predict more than one class. Another approach used is classifier chaining and label powersets. The data used only includes the stats of each Pokémon.
- (Zahroof 2019): Multi-label classification of both types using CNN and fully-connected network (SNN) with pokemon in game sprite images.
- (Belfer 2021): Multi-label classification of both types using natural language processing (NLP) on the pokemon's pokedex descriptions.

5 Methods and Results

5.1 Nested Cross-Validation

Acknowledging that the average `type_1` category has fewer than 60 Pokémon, we are working with a small amount of data. To make the most of this, we use 5-fold nested cross validation (CV) so that every observation can take part in both the fitting and evaluation of models. The folds are stratified by `type_1` to ensure a similar type distribution in all folds. For each train-test split, we perform hyperparameter tuning on the training set and use the tuned model to make predictions on the remaining fold. Through this process, we obtain one predicted type for each Pokémon. Comparing the predicted types to the actual types, we get an overall accuracy score for the model and plot a confusion matrix.

5.2 Hyperparameter Tuning

For each train-test split of the outer CV, we perform another inner stratified 5-fold CV to select the best hyperparameters for the given model. However, for most models, the hyperparameter space is continuous and makes exhaustive search impossible. For example, regularized logistic regression admits a two-dimensional parameter space: $(0, \infty) \times (0, \infty)$.

In early attempts, we tried grid search where we discretize the continuous search space and perform exhaustive search over the combination of hyperparameters in the search space. We then extracted the hyperparameters that optimize the model in a stratified 5-fold CV and treat it as the optimal hyperparameters. For instance, we can reduce the search space of regularized logistic regression into $\{10^{-4}, 10^{-2}, 1, 10^2, 10^4\} \times \{10^{-4}, 10^{-2}, 1, 10^2, 10^4\}$ which contains $5^2 = 25$ points. However, there are two major drawbacks to this method. First, the hyperparameters in the discretized search space may be far from the true optimal parameters. Second, the number of models needed to be fit increases exponentially with the number of dimensions in the search space. This forces one to reduce the density of values in each dimension in order to keep the tuning process manageable.

To address the issue, we use Bayesian optimization (BayesOpt).(Snoek, Larochelle, and Adams 2012) Instead of a fixed set of values, we sample one point at a time from the parameter space, and evaluate the corresponding model's cross validation accuracy. We then treat the samples as data and fit a Gaussian process regression model to estimate the cross validation accuracy as a probabilistic function of the parameter space. In doing so, BayesOpt chooses the next sample according to a balance of exploration and exploitation. For exploration, BayesOpt targets points in regions with high uncertainty to gather information. For exploitation, BayesOpt targets points in regions with high expected accuracy to make incremental improvements. After some number of samples (50 in our case), BayesOpt reports the sample with the highest accuracy as the optimum. While it is not necessarily deterministic, it solves the problems faced by grid search. Namely, the optimum is not restricted to a predetermined set of points, and the algorithm adapts to any number of dimensions without forced exponential blowup.

To implement BayesOpt, we used `scikit-optimize`, which is only capable of tuning `scikit-learn` models. The neural network model is not implemented with `scikit-learn`, so it was tuned using random search. It is similar to BayesOpt in that it is nondeterministic, but random search chooses its samples entirely at random.

5.3 Duplication

Pokemon type prediction can be treated as either a single-label classification task (`type_1`), or a multi-label classification task (`type_1` and `type_2`). In the multi-label case, a Pokemon's type can be treated as a multiset of cardinality 2. Pure Pokemon have a single type with multiplicity 2, while most Pokemon have two different types with multiplicity 1.

To easily develop models for both tasks, we propose duplication, a technique to convert any single-label classifier that outputs class probabilities, into a fixed-cardinality label multiset classifier. Given a dataset $\mathcal{D} = \{(x_i, \{y_{i1}, y_{i2}\})\}_{i=1}^N$, we generate a new dataset $\mathcal{D}' = \{(x_i, y_{i1}), (x, y_{i2})\}_{i=1}^N$ containing twice as many observations. We tune a single-label classifier on the $\mathcal{D}'$. We then take this classifier and scale its predicted type probabilities by 2. The scaled probabilities represent the expected multiplicity of each label in y_i. Finally, to predict, we apply the following algorithm:

1. $y_1 = \text{argmax}_y p_y$ (predict y_1 as the type with highest probability)
2. $p_{y_1} \leftarrow p_{y_1} - 1$ (subtract 1 from the probability of y_1)
3. $y_2 = \text{argmax}_y p_y$ (predict y_2 as the next type that maximizes the probability)

For example, a Pokemon with the predicted probabilities (Water: 75%, Grass: 20%, Flying: 5%, rest: 0%) would be scaled to (Water: 1.5, Grass: 0.4, Flying: 0.1, rest: 0.0). Then, Water would be the first predicted class, after which the probability vector would be updated to (Water: 0.5, Grass: 0.4, Flying: 0.1, rest: 0.0). Since Water is still the highest class, it is predicted again (multiplicity 2).

To demonstrate the method, we implemented duplication on all models that use `sklearn`: logistic regression, SVM, and random forest. We compare the models' performance at the duplication task alongside the normal prediction of `type_1` only.

To measure the accuracy of a multiset prediction, we take the multiset intersection between the predicted and true labels, and divide that set's cardinality by 2.

5.4 Logistic Regression

We focused on multinomial logistic regression using `scikit-learn` package in Python. For multinomial logistic regression with $K \geq 2$ classes, we fit a single classifier with K outputs (one of them is 1), and take the softmax thereof. For regularization, ElasticNet encompasses both L1 and L2 penalties (and with a weak enough regularization term, no-penalty as well). We thus parameterize the search space of potential hyperparameters by the ratio of the L1 and L2 penalties, and the coefficient `C` representing the inverse of regularization strength. Specifically, we utilize Bayesian Optimization to sample over the entire continuous space with ranges of the ratio of the L1 and L2 penalties and `C` being $[0.0, 1.0] \times [10^{-4}, 10^4]$.

After applying Bayesian Optimization and nested cross-validation with $k = 5$ folds, we observe that the ratio of the L1 and L2 penalties is constant at 1.0 which indicates that the L1 penalty (LASSO regularization) is chosen each time. The coefficient `C` fluctuates between the ranges of $[0, 5]$. Overall, we obtain a model accuracy of 99.147%. The high accuracy suggests that the data is linearly separable.

We also examined the model performance without the features `type_2` and the `damage_from_{Type}` and used the same search space. We observe that the ratio of the L1 and L2 penalties fluctuates in the range $[0.4, 1]$ and the coefficient `C` fluctuates between $[3.5, 10]$, indicating weaker regularization. The model performance decreased significantly where the overall prediction accuracy decreased to 36.903%. As a result, it would suggest that the `type_2` and `damage_from_{Type}` variables are very significant features in predicting Pokémon type.

We observe similar trends at the duplication task. That is, the majority of models choose to use solely the L1 penalty (LASSO regularization). When excluding `type_2` and `damage_from_{Type}`, the level of regularization also decreases.

5.5 Support Vector Machines

There are two formulations of multiclass classification using SVMs: one-versus-rest (OVR) and one-versus-one (OVO). We decide to use OVR because it fits one SVM for each of the 18 classes, whereas OVO will create $18 \times 17 = 306$ models which would increase the model complexity, training time, and the risk of overfitting.

We trained support vector machine (SVM) models using the `scikit-learn` package in Python. The hyperparameters we tested are the regularization parameter `C`, the `kernel` type, the `degree` of the polynomial if the kernel is polynomial, the kernel coefficient `gamma`, an independent term in the kernel function `coef0` (only significant for a polynomial/sigmoid kernel), and whether to use probability output. For tuning, we consider the linear, polynomial, and radial basis function (RBF) kernels. The sigmoid kernel is never guaranteed to be positive semi-definite and is therefore avoided.(Karatzoglou 2023) Moreover, the linear kernel is just a special case of the polynomial kernel (with `degree` $= 1$, `gamma` $= 1$, `coef0` $= 0$), allowing us to specify it together with the polynomial kernel. Thus the hyperparameter search space is presented as `degree`, `C`, `gamma`, and `coef0` ranging between $[2, 5] \times [10^{-4}, 10^4] \times [10^{-3}, 10^3] \times [0.0, 1.0]$ for the polynomial kernel. For the RBF kernel, the search space is `C` and `gamma` ranging between $[10^{-4}, 10^4] \times [10^{-3}, 10^3]$ respectively.

After hyperparameter tuning, we observe that in general, the best kernel was the RBF with the regularization parameter `C` fluctuating from 33 to 10,000 where `C` is inversely proportional to the strength of regularization and `gamma` was between $[0.002, 0.036]$. Overall, we obtain a model accuracy of 96.680%.

Next, we investigate the strength of the model without the features `type_2` and the `damage_from_{Type}` without changing the search space. We observed that in general the best kernel was still the radial basis function, with `C` and `gamma` ranging from $[18, 867] \times [0.15, 0.27]$ respectively. After removing these features, the overall model accuracy falls to 46.583%.

When investigating duplication, we find that the model tuning code does not terminate. The reason for this is discussed in sklearn itself. When two points have exactly the same predictor values, their corresponding rows in the kernel matrix are identical. This makes the matrix singular, preventing optimization. Thus, we are unable to evaluate the duplication task for SVMs.

5.6 Random Forests

For random forests, we utilized the `scikit-learn` package in Python. The search space of hyperparameters we tested include

- `criterion`: The splitting criterion at each node, either Gini index or logistic loss (equivalent to entropy).
- `max_features`: The number of random features to try at each split. Ranges from 1 to the number of features in the dataset.
- `max_depth`: the maximum depth of the tree. Ranges from 1 to the number of observations, but in practice we do not need even close to that much. We generously limit this to the number of features, to make the optimization problem easier.

After hyperparameter tuning, we observe that in general the Gini index criterion was preferred. The optimal `max_depth`s range between 10 and 38. The optimal `max_features` range between 35 and 74. Overall, we obtain a model accuracy of 96.491%.

Next, we also investigate the strength of the model without the features `type_2` and the `damage_from_{Type}`, with the same search space as before. We observe that the `gini` index criterion was still preferred while the `max_depth` and `max_features` fluctuate between $[13, 37] \times [3, 37]$ respectively. Overall, we obtain a model accuracy of 44.971% after removing these features.

5.7 Neural Network

We used PyTorch to implement neural networks. Our network is a multilayer perceptron with `H-layer` hidden layers and `node` number of nodes per hidden layer. The model is trained for `epoch` epochs using batch stochastic gradient descent with an Adam optimizer. `epoch` acts as a regularization parameter; by stopping the training process early it can prevent overfitting.

We using `H_layer` = 2, `node` = 250, and `epoch` = 350 as a rough estimate of the optimal values, selected using a broad grid search. See Appendix Initial Hyperparameter Values for Neural Networks . Then, we select hyperparameters using random search (20 steps) of nearby values. The nested CV result shows an average of 98.957% accuracy with lowest fold having 98.881% accuracy and highest fold having 99.318% accuracy.

After removing `type_2` and `damage_from_{Type}`, we find that the prediction task becomes much more difficult, thus we need more expressive models to converge within a reasonable number of epochs. In this case, the broad grid search selects `H_layer` = 2, `node` = 1300, and `epoch` = 200 as initial values for random search. The nested CV result shows an average of 45.160% accuracy with lowest fold having 43.454% and highest fold having 45.940% accuracy.

6 Analysis and Model Comparision

6.1 Confusion Matrices Intepretation

Overall, every classifier performed extremely well (over 95% accuracy) when `type_2` and `damage_from_{Type}` were included in the feature space. The result is not surprising as `damage_from_{Type}` is strongly related to the type of the Pokemon and `type_2` eliminates a choice of `type_1` (`type_2` and `type_1` must be different). Thus, the performance of the models are expected. See Appendix for Confusion Matrices Confusion Matrix for Logistic Regression (Base) , Confusion Matrix for Support Vector Machine (Base) , Confusion Matrix for Random Forest (Base) , Confusion Matrix for Neural Network (Base)

However, the same does not hold when we train the model without the `type_2` and `damage_from_{Type}` variables. The result from nested CV suggests a drastic decrease in prediction accuracy for all models.

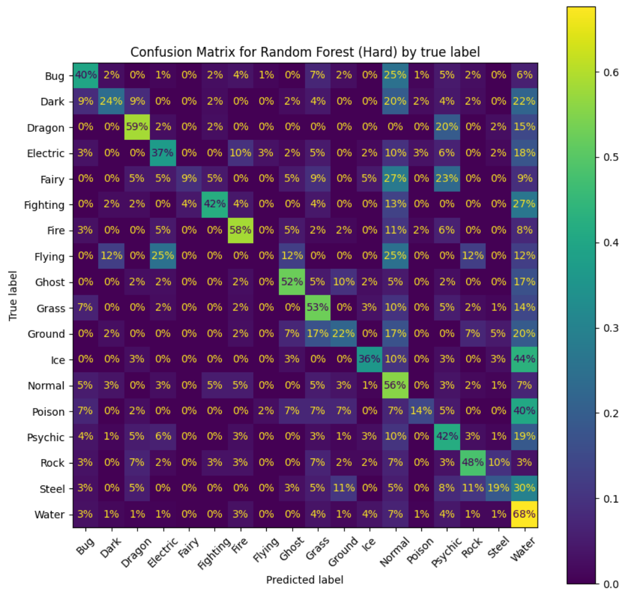


Figure 2: Confusion Matrix on Test Data for Random Forest (SVM) Without type_2 and damage_from_{type} Features

For instance, in the Confusion Matrix for Random Forest (Hard), we observed that the model was not able to predict any `Flying` type correctly and did extremely poorly for `Fairy` type. This is likely due to the small sample size of `Flying` and `Fairy` type Pokemon which has 8 and 19 samples respectively. Another interesting observation is that all models tend to make errors by falsely classifying the Pokemon as `Water` or `Bug` type. This is most obvious in random forest where most errors occurred in the `Water` column and might have to do with the relatively large `Water` type Pokemon population (136 `Water` type) and their generic attributes compared to the population. In contrast, `Dragon` and `Fire` type Pokémon has a relatively higher prediction accuracy across more models. We are certain that the high `Dragon` type prediction accuracy is because some of their particularly noticeable features: an extremely high `egg_cycles` and an extremely low `catch_rate` See Appendix Dragon Pokémon Features . As for the `Fire` type Pokémon, we did not find a particular standout feature that explains its high prediction accuracy. See Appendix for Confusion Matrices Confusion Matrix for Logistic Regression (Hard), Confusion Matrix for Support Vector Machine (Hard), Confusion Matrix for Neural Network (Hard)

6.2 Model Comparison

Model	Performance
Logistic Regression (base)	99.147%
Logistic Regression (hard)	36.903%
Logistic Regression (dup/base)	97.817%
Logistic Regression (dup/hard)	35.626%
Support Vector Machine (base)	96.680%
Support Vector Machine (hard)	46.583%
Support Vector Machine (dup/base)	NA
Support Vector Machine (dup/hard)	NA
Random Forest (base)	96.491%
Random Forest (hard)	44.971%
Random Forest (dup/base)	99.763%
Random Forest (dup/hard)	39.945%
Neural Network (base)	99.318%
Neural Network (hard)	45.160%
Neural Network (dup/base)	NA
Neural Network (dup/hard)	NA

- base: with all features in the dataset
- hard: without type_2 and damage_from_{Type} features
- dup: with duplication method applied

From the table, it is evident that using the base features the models returned by each method performs extremely well (over 95% accuracy). Comparing `base` versus `dup/base`, logistic regression performs slightly worse with duplication applied, while random forests perform slightly better. However, the difference is only a few percent, which is not large compared to the fluctuations we experience in cross-validation accuracy.

Whereas, once hard mode was applied, the accuracy dropped significantly regardless if the duplication method was applied or not, indicating that type_2 and damage_from_{Type} features add valuable information in predicting the Pokémon's type.

We achieve the highest overall accuracy with random forest with the duplication method applied on all the features, and the lowest overall accuracy with logistic regression with the duplication method applied on hard mode.

7 References

Belfer, Ryan. 2021. “Predicting Pokémon Type Using Pokédex Entries.” https://medium.com/analytics-vidhya/predicting-pok%C3%A9mon-type-with-the-pok%C3%A9dex-7038754dc422.

Ezeilo, Chima. 2018. “Predicting Pokemon Types.” https://www.kaggle.com/code/chimae/predicting-pokemon-types/notebook.

Karatzoglou, Alexandros. 2023. “Kernel Functions.” https://search.r-project.org/CRAN/refmans/kernlab/html/dots.html.

(kn-kn)., Kevin. 2017. “Predictin Pokemon Types.” https://github.com/kn-kn/pokemon-type-prediction/blob/master/Pokemon_Jupyter.ipynb.

Shahir, Jamshaid. 2022. “Multi-Label Classification of Pokemon Types with TensorFlow.” https://towardsdatascience.com/multi-label-classification-of-pokemon-types-with-tensorflow-8217a38038a6.

Snoek, Jasper, Hugo Larochelle, and Ryan P. Adams. 2012. “Practical Bayesian Optimization of Machine Learning Algorithms.” https://arxiv.org/abs/1206.2944.

Soares, Henrique M. 2017. “Who Is That Neural Network?” https://jgeekstudies.org/2017/03/12/who-is-that-neural-network/.

Tan, Jonathan. 2022. “Can We Guess a Pokemon’s Type from Its Stats? Machine Learning in Python.” https://medium.com/@jsw.tan1991/can-we-guess-a-pokemons-type-from-its-stats-machine-learning-in-python-c4586bec9f48.

Wei Chen, Hsiao. 2021. “Pokemon Decision Tree.” https://www.kaggle.com/code/sha310139/pokemon-decision-tree.

Zahroof, Tariq. 2019. “What’s That Pokemon?” https://cs230.stanford.edu/projects_spring_2019/reports/18664574.pdf.

8 Appendix

8.1 Initial Hyperparameter Values for Neural Networks

Since neural networks are more expensive to train than the other models explored here, we first conduct a broad grid search to find approximate optimal values of the tuning parameters `H-layer`, `node`, and `epoch`. Instead of BayesOpt, we do a broad grid search with cross-validation for `H-layer` (between 1 and 5) and `node` (between 100 and 300), fixing `epoch` = 400 to a reasonable value. Then, we perform another broad grid search over `epoch` (between 300 and 550) using the optimal `H-layer` and `node`. Due to the variability of batch stochastic gradient descent, we repeat the training process 5 times for each CV fold to reduce variance. With the "optimized" hyperparameters, we conduct a nested CV to evaluate the algorithm performance where we randomly generate hyperparameters that is close to "optimized" hyperparameters in each inner CV in search of a potentially better model. (Note we will include the "optimized" hyperparameters in nested CV).

After removing `type_2` and `damage_from_{Type}`, we find that the prediction task becomes much more difficult, thus we need more expressive models to converge within a reasonable number of epochs. This time, the first broad grid search is over `H-layer` (between 1 and 4) and `node` (between 1000 and 2000), fixing `epoch` = 600. After examining the training curves of the models from the first grid search, we choose the second grid search over `epoch` to be between 100 and 300.

8.2 Pokemon Type Frequency

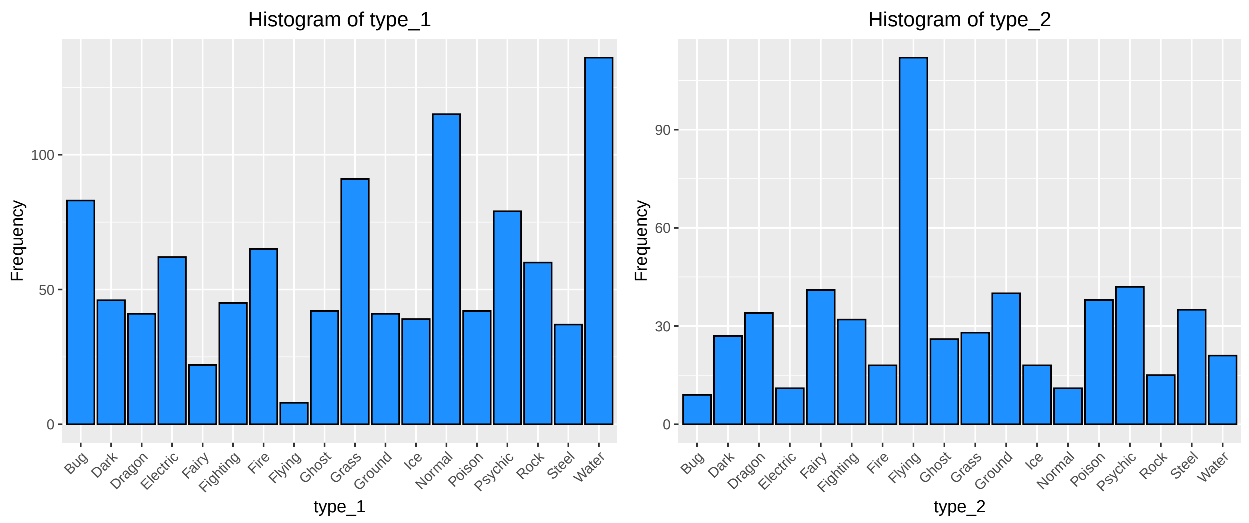


Figure 3: Pokemon Type Frequency for Type 1 and Type 2

8.3 Dragon Pokemon Features

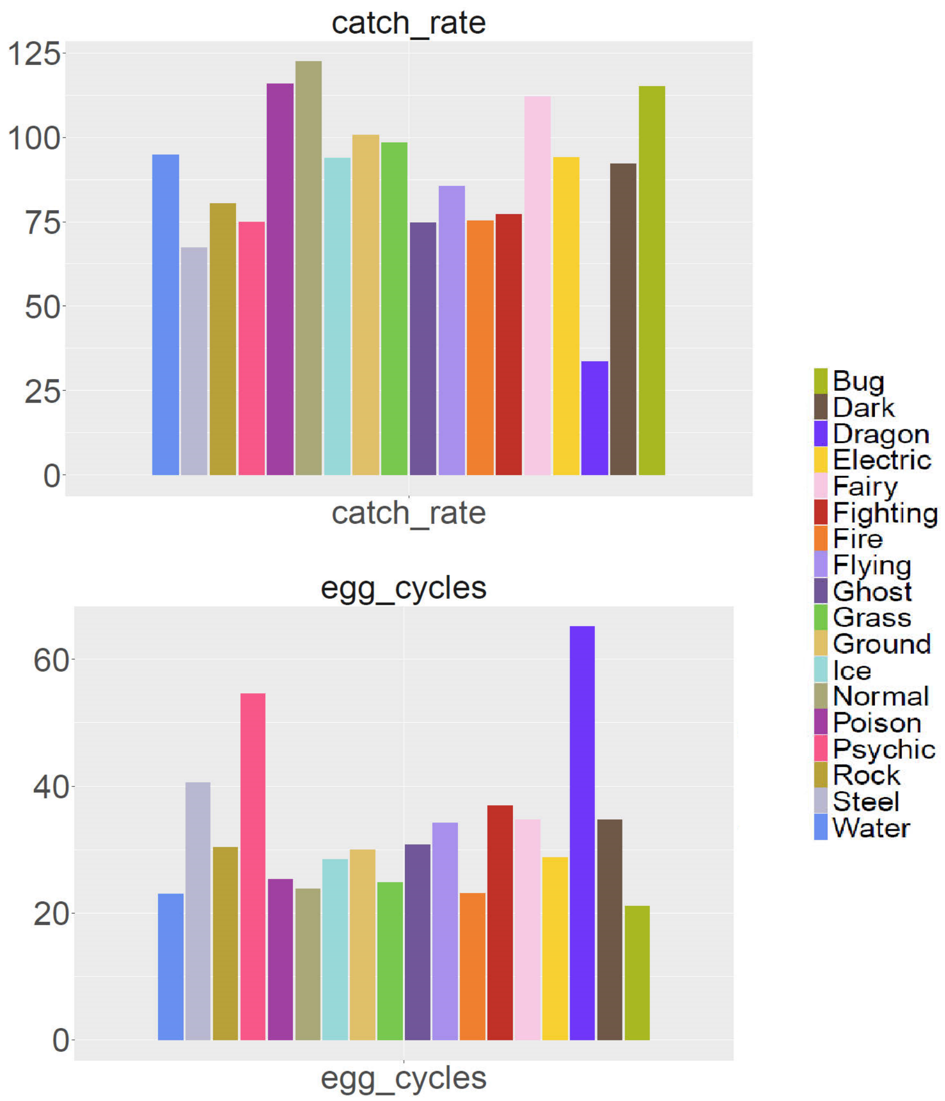


Figure 4: The noticable feature of Dragon type Pokemon (purple in colour)

8.4 Confusion Matrix for Logistic Regression (Base)

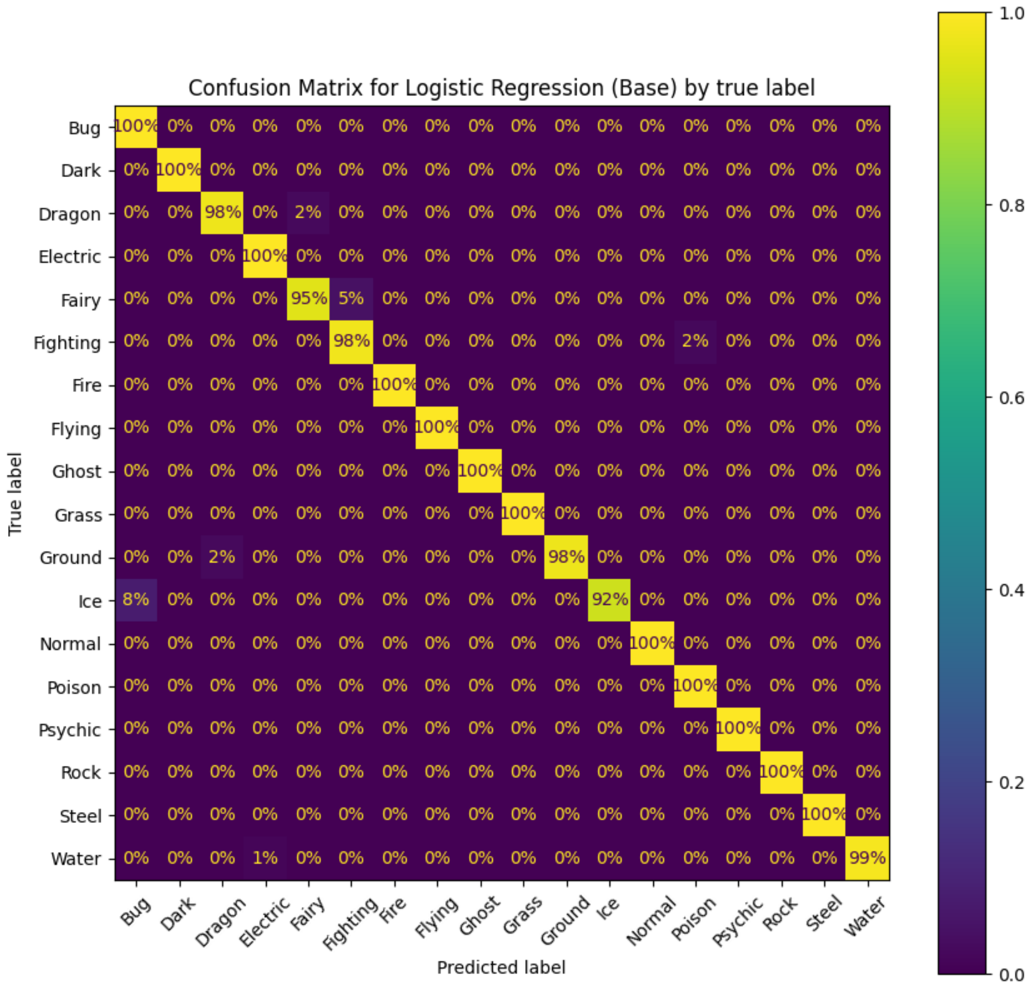


Figure 5: Confusion Matrix on Test Data for Logistic Regression

8.5 Confusion Matrix for Logistic Regression (Hard)

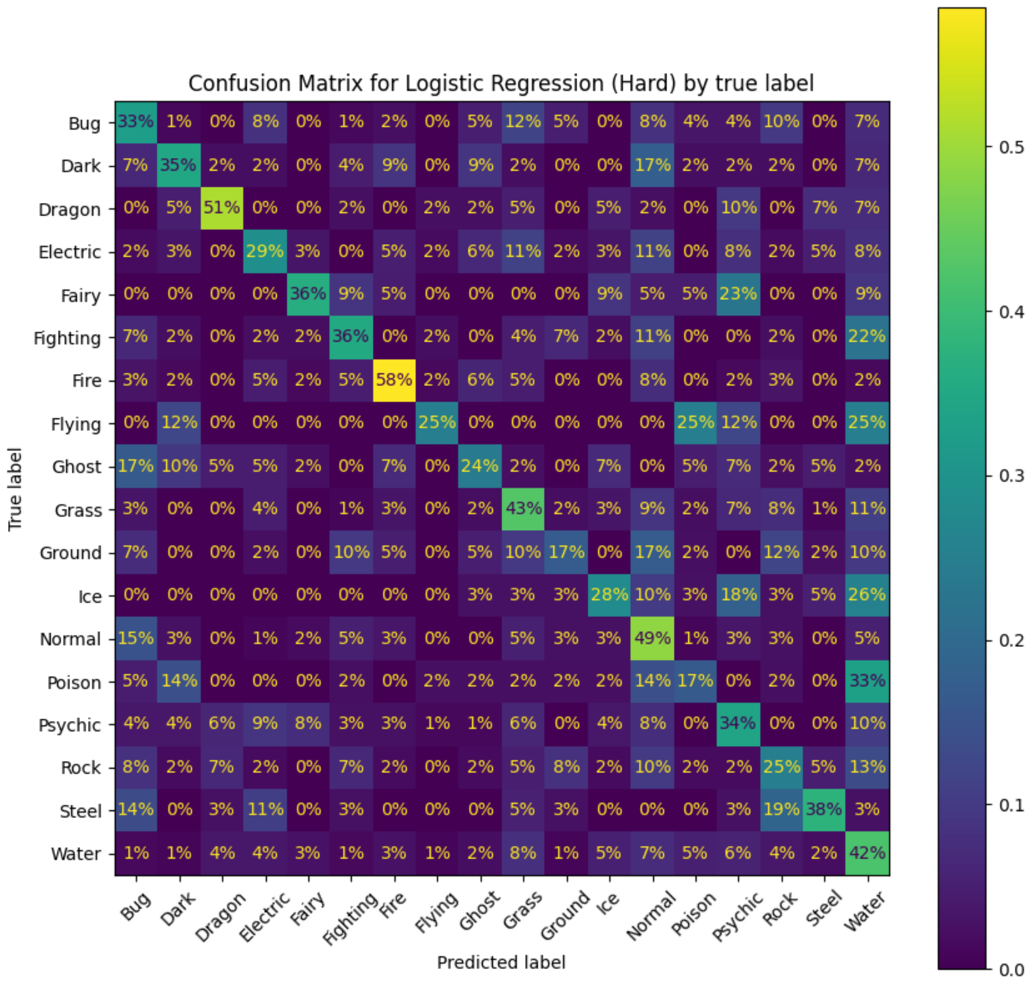


Figure 6: Confusion Matrix on Test Data for Logistic Regression Without type_2 and damage_from_{type} Features

8.6 Confusion Matrix for Support Vector Machine (Base)

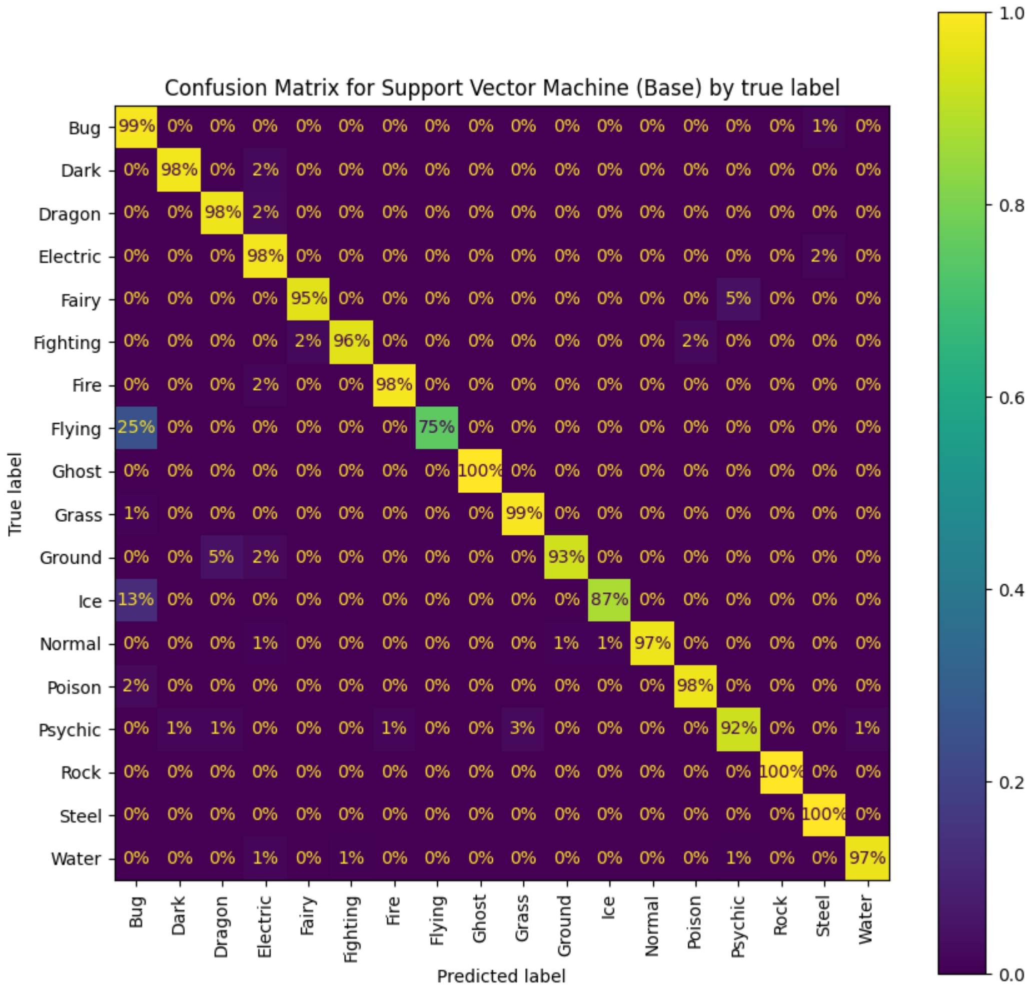
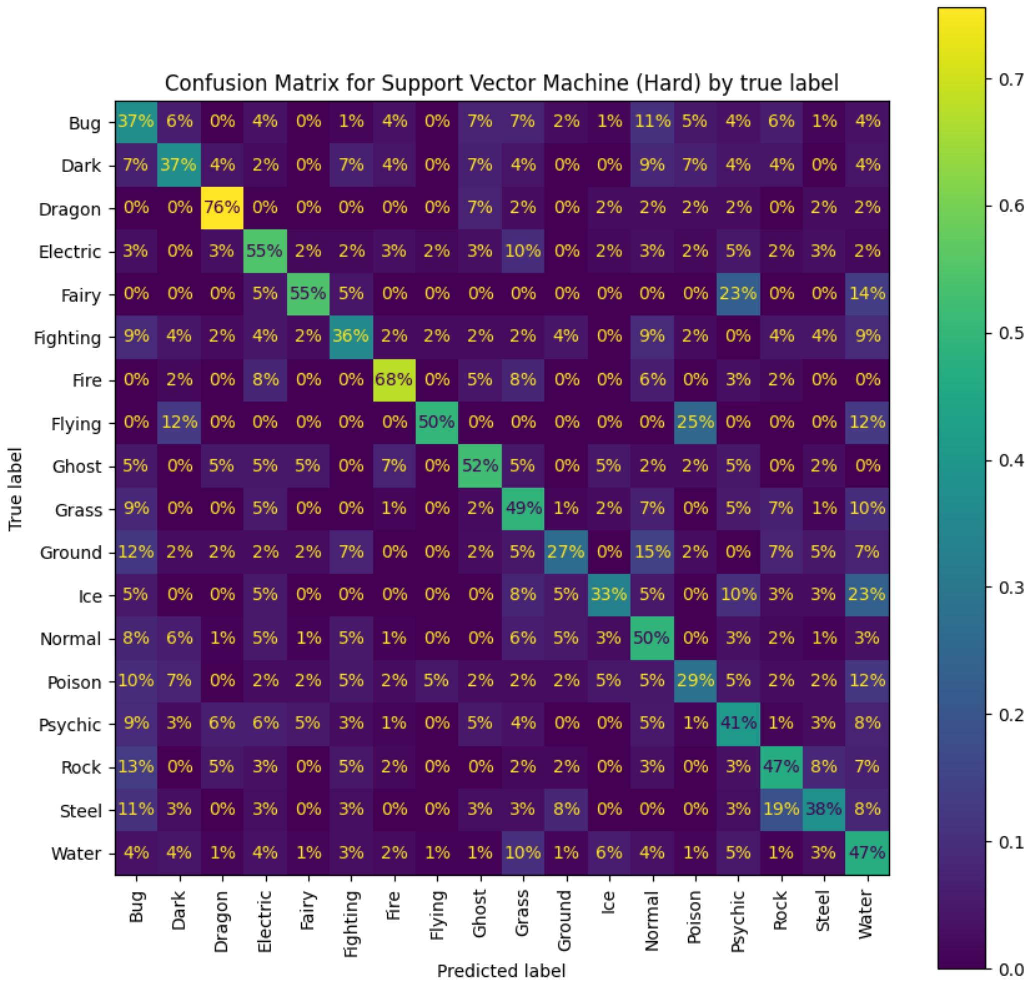


Figure 7: Confusion Matrix on Test Data for Support Vector Machine (SVM) Without type_2 and damage_from_{type} Features

8.7 Confusion Matrix for Support Vector Machine (Hard)



	Bug	Dark	Dragon	Electric	Fairy	Fighting	Fire	Flying	Ghost	Grass	Ground	Ice	Normal	Poison	Psychic	Rock	Steel	Water
Bug	37%	6%	0%	4%	0%	1%	4%	0%	7%	7%	2%	1%	11%	5%	4%	6%	1%	4%
Dark	7%	37%	4%	2%	0%	7%	4%	0%	7%	4%	0%	0%	9%	7%	4%	4%	0%	4%
Dragon	0%	0%	76%	0%	0%	0%	0%	0%	7%	2%	0%	2%	2%	2%	2%	0%	2%	2%
Electric	3%	0%	3%	55%	2%	2%	3%	2%	3%	10%	0%	2%	3%	2%	5%	2%	3%	2%
Fairy	0%	0%	0%	5%	55%	5%	0%	0%	0%	0%	0%	0%	0%	0%	23%	0%	0%	14%
Fighting	9%	4%	2%	4%	2%	36%	2%	2%	2%	2%	4%	0%	9%	2%	0%	4%	4%	9%
Fire	0%	2%	0%	8%	0%	0%	68%	0%	5%	8%	0%	0%	6%	0%	3%	2%	0%	0%
Flying	0%	12%	0%	0%	0%	0%	0%	50%	0%	0%	0%	0%	0%	25%	0%	0%	0%	12%
Ghost	5%	0%	5%	5%	5%	0%	7%	0%	52%	5%	0%	5%	2%	2%	5%	0%	2%	0%
Grass	9%	0%	0%	5%	0%	0%	1%	0%	2%	49%	1%	2%	7%	0%	5%	7%	1%	10%
Ground	12%	2%	2%	2%	2%	7%	0%	0%	2%	5%	27%	0%	15%	2%	0%	7%	5%	7%
Ice	5%	0%	0%	5%	0%	0%	0%	0%	0%	8%	5%	33%	5%	0%	10%	3%	3%	23%
Normal	8%	6%	1%	5%	1%	5%	1%	0%	0%	6%	5%	3%	50%	0%	3%	2%	1%	3%
Poison	10%	7%	0%	2%	2%	5%	2%	5%	2%	2%	2%	5%	5%	29%	5%	2%	2%	12%
Psychic	9%	3%	6%	6%	5%	3%	1%	0%	5%	4%	0%	0%	5%	1%	41%	1%	3%	8%
Rock	13%	0%	5%	3%	0%	5%	2%	0%	0%	2%	2%	0%	3%	0%	3%	47%	8%	7%
Steel	11%	3%	0%	3%	0%	3%	0%	0%	3%	3%	8%	0%	0%	0%	3%	19%	38%	8%
Water	4%	4%	1%	4%	1%	3%	2%	1%	1%	10%	1%	6%	4%	1%	5%	1%	3%	47%

Figure 8: Confusion Matrix on Test Data for Support Vector Machine (SVM) Without type_2 and damage_from_{type} Features

8.8 Confusion Matrix for Random Forest (Base)

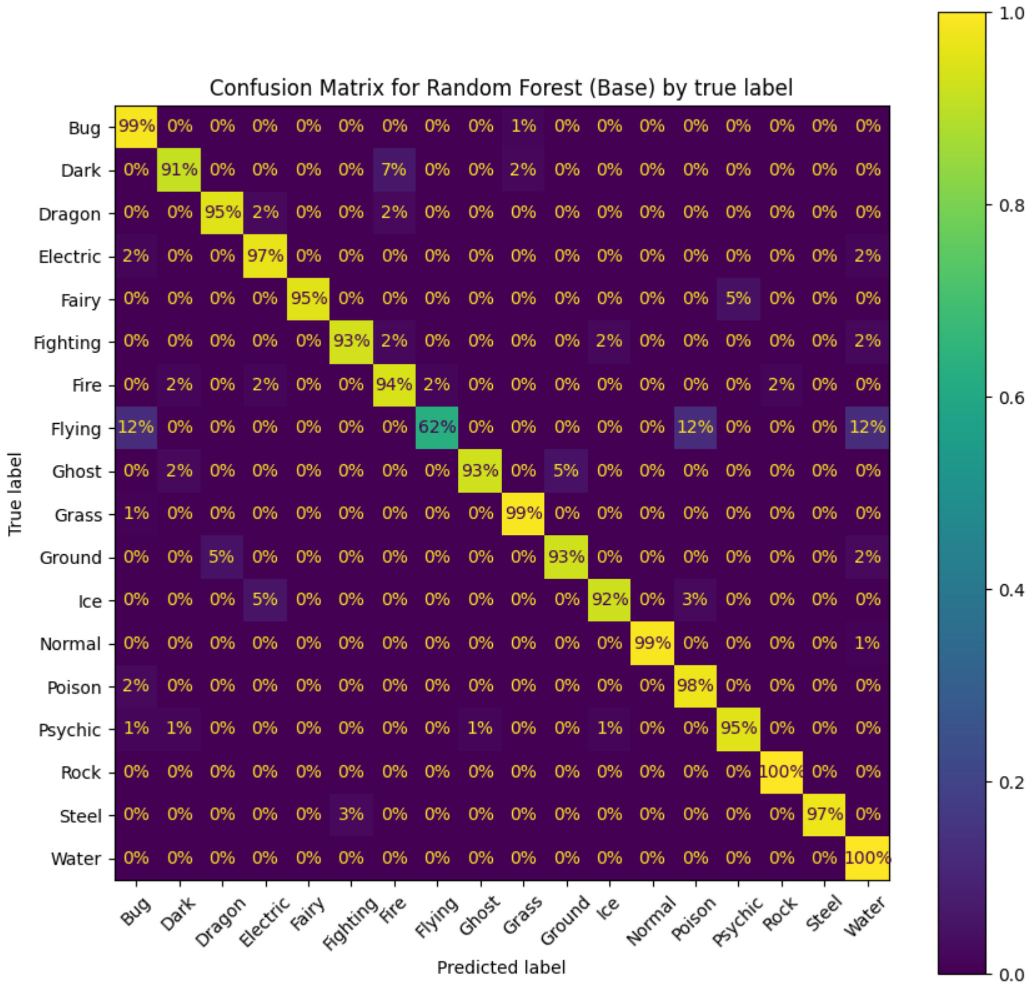


Figure 9: Confusion Matrix on Test Data for Random Forest (SVM)

8.9 Accuracy Heat map for Neural Network (Base)

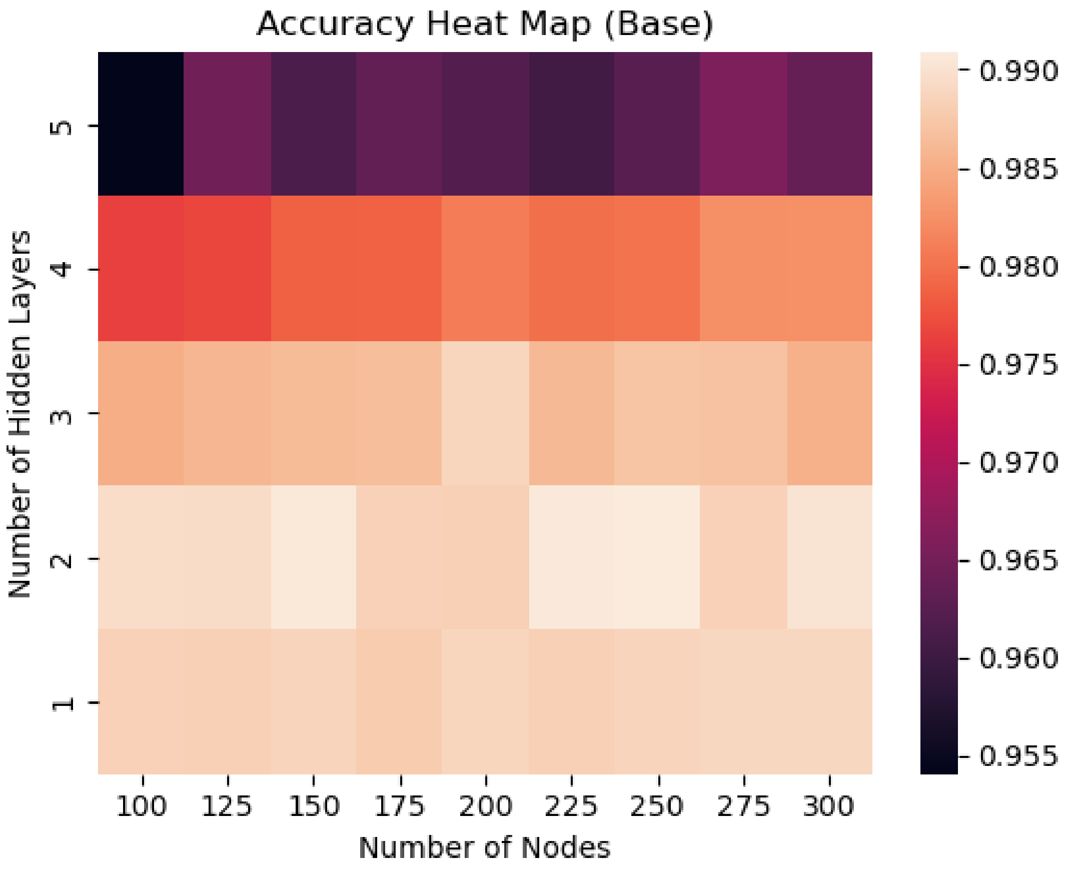


Figure 10: The accuracy heat map for Neural Network with different number of hidden layers and nodes per layer. 2 hidden layers with 250 nodes seem to work the best.

8.10 Confusion Matrix for Neural Network (Base)

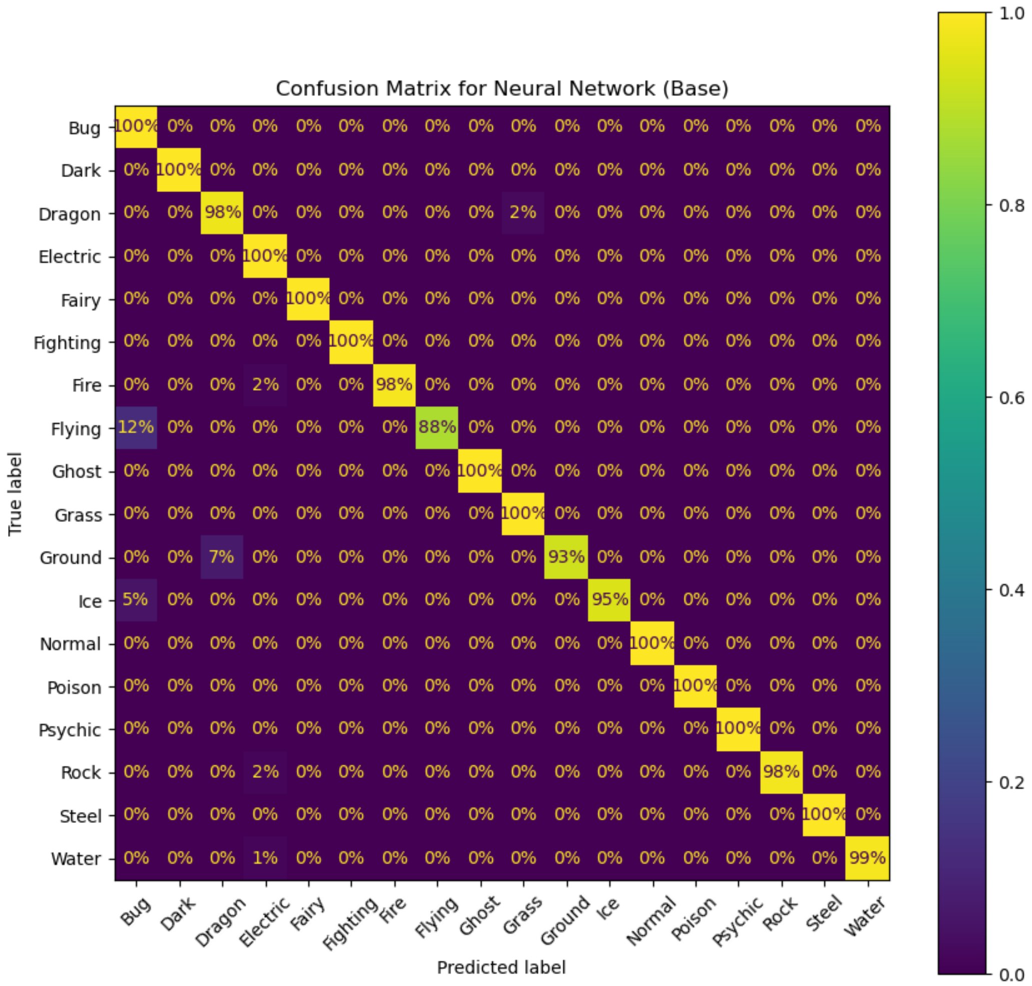


Figure 11: The confusion matrix for Neural Network

8.11 Accuracy Heat map for Neural Network (Hard)

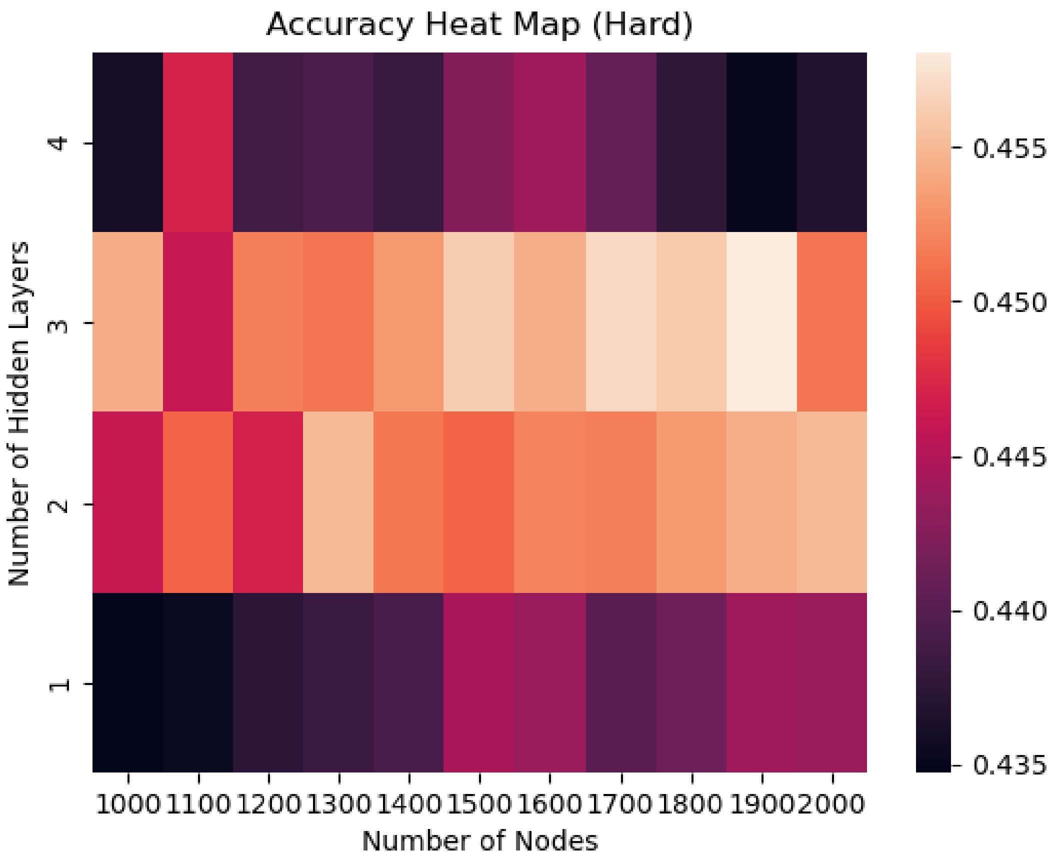
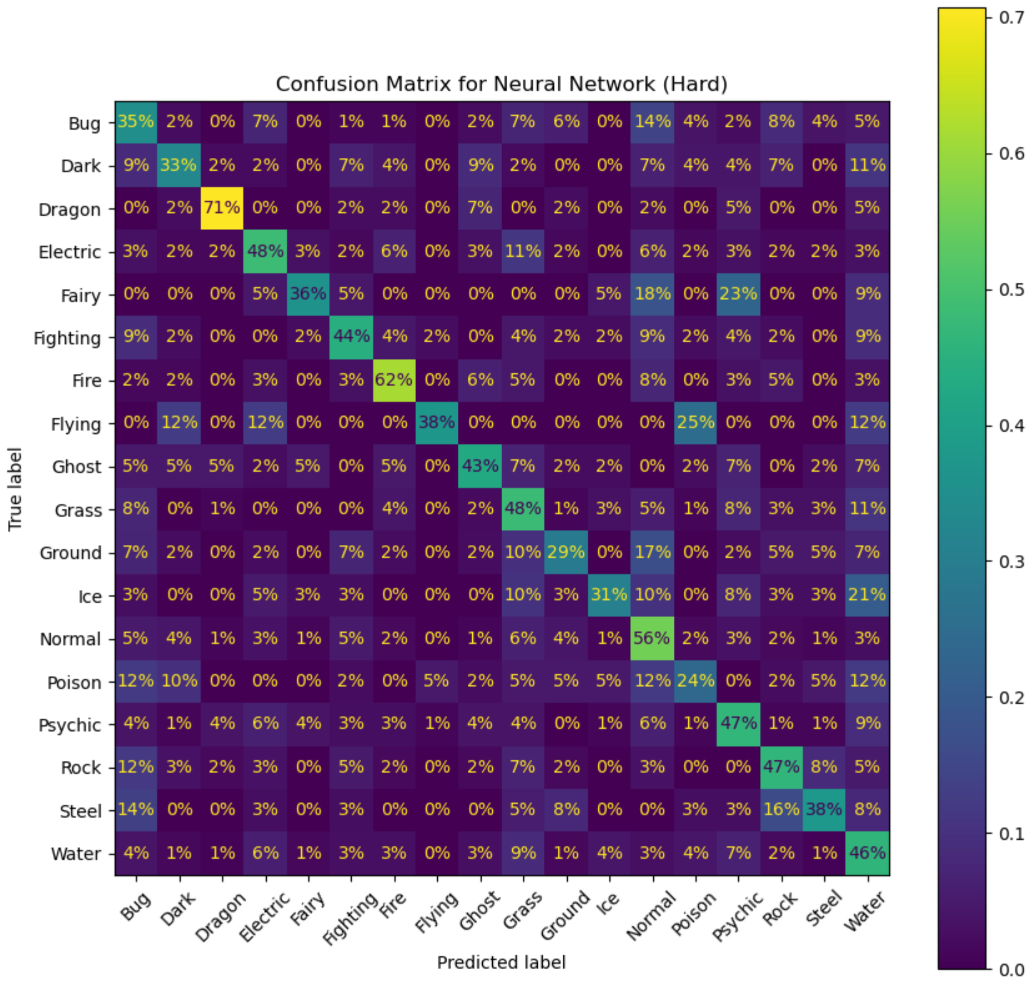


Figure 12: The accuracy heat map for Neural Network with different number of hidden layers and nodes per layer Without type_2 and damage_from_{type} Features. 3 hidden layers with 1900 nodes seem to work the best

8.12 Confusion Matrix for Neural Network (Hard)



	Bug	Dark	Dragon	Electric	Fairy	Fighting	Fire	Flying	Ghost	Grass	Ground	Ice	Normal	Poison	Psychic	Rock	Steel	Water
Bug	35%	2%	0%	7%	0%	1%	1%	0%	2%	7%	6%	0%	14%	4%	2%	8%	4%	5%
Dark	9%	33%	2%	2%	0%	7%	4%	0%	9%	2%	0%	0%	7%	4%	4%	7%	0%	11%
Dragon	0%	2%	71%	0%	0%	2%	2%	0%	7%	0%	2%	0%	2%	0%	5%	0%	0%	5%
Electric	3%	2%	2%	48%	3%	2%	6%	0%	3%	11%	2%	0%	6%	2%	3%	2%	2%	3%
Fairy	0%	0%	0%	5%	36%	5%	0%	0%	0%	0%	0%	5%	18%	0%	23%	0%	0%	9%
Fighting	9%	2%	0%	0%	2%	44%	4%	2%	0%	4%	2%	2%	9%	2%	4%	2%	0%	9%
Fire	2%	2%	0%	3%	0%	3%	62%	0%	6%	5%	0%	0%	8%	0%	3%	5%	0%	3%
Flying	0%	12%	0%	12%	0%	0%	0%	38%	0%	0%	0%	0%	0%	25%	0%	0%	0%	12%
Ghost	5%	5%	5%	2%	5%	0%	5%	0%	43%	7%	2%	2%	0%	2%	7%	0%	2%	7%
Grass	8%	0%	1%	0%	0%	0%	4%	0%	2%	48%	1%	3%	5%	1%	8%	3%	3%	11%
Ground	7%	2%	0%	2%	0%	7%	2%	0%	2%	10%	29%	0%	17%	0%	2%	5%	5%	7%
Ice	3%	0%	0%	5%	3%	3%	0%	0%	0%	10%	3%	31%	10%	0%	8%	3%	3%	21%
Normal	5%	4%	1%	3%	1%	5%	2%	0%	1%	6%	4%	1%	56%	2%	3%	2%	1%	3%
Poison	12%	10%	0%	0%	0%	2%	0%	5%	2%	5%	5%	5%	12%	24%	0%	2%	5%	12%
Psychic	4%	1%	4%	6%	4%	3%	3%	1%	4%	4%	0%	1%	6%	1%	47%	1%	1%	9%
Rock	12%	3%	2%	3%	0%	5%	2%	0%	2%	7%	2%	0%	3%	0%	0%	47%	8%	5%
Steel	14%	0%	0%	3%	0%	3%	0%	0%	0%	5%	8%	0%	0%	3%	3%	16%	38%	8%
Water	4%	1%	1%	6%	1%	3%	3%	0%	3%	9%	1%	4%	3%	4%	7%	2%	1%	46%

Figure 13: The accuracy heat map for Neural Network without type_2 and damage_from_{type} Features

9 Code

9.1 Logistic Regression

We would like to perform hyperparameter selection over the set of all logistic regression models.

Logistic regression with $K > 2$ classes is actually broken into two "flavours": - **one-versus-all**: Fit a separate binary classifier for each class against the rest, and classify according to the highest score. - **multinomial**: Fit a single classifier with K outputs (one of them is 1), and take the softmax thereof. This is the flavour we learn in STAT 441, and is the default used by `sklearn`.

We choose to focus on multinomial logistic regression here.

9.1.1 Setup

```
from model import SKLogisticRegression, DuplicateSKLogisticRegression
from tune import (
    Constant,
    outer_cv,
    Real,
    SKBayesTuner,
)
```

9.1.2 Search Space

For regularization, ElasticNet encompasses both L1 and L2 penalties (and with a weak enough regularization term, no-penalty as well). We thus parameterize the search space by the ratio of the L1 and L2 penalties, and the coefficient C representing the extent of regularization.

Unfortunately, the only solver in `sklearn` that works with ElasticNet is the `saga` solver, and that solver is quite sluggish.

```
space = dict(
    # constants
    multi_class=Constant("multinomial"),
    penalty=Constant("elasticnet"),
    solver=Constant("saga"),
    max_iter=Constant(9001),  # practically unlimited
    random_state=Constant(441),
    warm_start=Constant(True),  # try to speed up optimization

    # variables
    C=Real(low=1e-4, high=1e4, prior="log-uniform"),
    l1_ratio=Real(low=0.0, high=1.0, prior="uniform"),
)
```

9.1.3 Base

```
outer_cv(
    tuner=SKBayesTuner(SKLogisticRegression()),
    search=space,
    name="Logistic Regression",
    duplicate=False,
    hard_mode=False,
)
```

9.1.4 Hard

```
outer_cv(
    tuner=SKBayesTuner(SKLogisticRegression()),
    search=space,
    name="Logistic Regression",
    duplicate=False,
    hard_mode=True,
)
```

9.1.5 Duplication

```
outer_cv(
    tuner=SKBayesTuner(DuplicateSKLogisticRegression()),
    search=space,
    name="Logistic Regression",
    duplicate=True,
    hard_mode=False,
)
```

9.1.6 Duplication, Hard

```
outer_cv(
    tuner=SKBayesTuner(DuplicateSKLogisticRegression()),
    search=space,
    name="Logistic Regression",
    duplicate=True,
    hard_mode=True,
)
```

9.2 Support Vector Machine

Support vector machines with $K > 2$ classes are broken into two "flavours": - **one-versus-rest**: Fit a separate SVM for each class against the rest, and classify according to the highest score. - **one-versus-one**: Fit a separate SVM for each pair of classes, and classify according to the class that wins the most.

We choose to focus on one-versus-rest, because it fits fewer models (thus fewer parameters).

9.2.1 Setup

```
from model import SKSupportVectorMachine, DuplicateSKSupportVectorMachine
from tune import (
    Categorical,
    Constant,
    Integer,
    outer_cv,
    Real,
    SKBayesTuner,
)
```

9.2.2 Search Space

For tuning, `sklearn` provides us 4 kernels, each slightly differently parameterized. Of these, we consider the linear, polynomial, and radial basis function kernels. The sigmoid kernel is never guaranteed to be positive semi-definite so we avoid using it. Moreover, the linear kernel is just a special case of the polynomial kernel (`degree = 1`, `gamma = 1`, `coef0 = 0`) so we can specify it together with the polynomial kernel.

```
space = [
    dict(
        # constants
        probability=Constant(True),
        random_state=Constant(441),
        decision_function_shape=Constant("ovr"),

        kernel=Constant("poly"),
        # variables
        degree=Integer(low=2, high=5, prior="log-uniform"),
        C=Real(low=1e-4, high=1e4, prior="log-uniform"),
        gamma=Real(low=1e-3, high=1e3, prior="log-uniform"),
        coef0=Categorical([0.0, 1.0]),  # effect of coef0 scales with gamma
    ),
    dict(
        # constants
        probability=Constant(True),
        random_state=Constant(441),
        decision_function_shape=Constant("ovr"),

        kernel=Constant("rbf"),
        # variables
        C=Real(low=1e-4, high=1e4, prior="log-uniform"),
        gamma=Real(low=1e-3, high=1e3, prior="log-uniform"),
    ),
]
```

9.2.3 Base

```
outer_cv(
    tuner=SKBayesTuner(SKSupportVectorMachine()),
    search=space,
    name="SVM",
    duplicate=False,
    hard_mode=False,
)
```

9.2.4 Hard

```
outer_cv(
    tuner=SKBayesTuner(SKSupportVectorMachine()),
    search=space,
    name="SVM",
    duplicate=False,
    hard_mode=True,
)
```

9.2.5 Duplication

```
outer_cv(
    tuner=SKBayesTuner(DuplicateSKSupportVectorMachine()),
    search=space,
    name="SVM",
    duplicate=True,
    hard_mode=False,
)
```

9.2.6 Duplication, Hard

```
outer_cv(
    tuner=SKBayesTuner(DuplicateSKSupportVectorMachine()),
    search=space,
    name="SVM",
    duplicate=True,
    hard_mode=True,
)
```

9.3 Random Forest

9.3.1 Setup

```
from model import SKRandomForest, DuplicateSKRandomForest
from tune import (
    Categorical,
    Constant,
    Integer,
    outer_cv,
    SKBayesTuner,
)
from util import load_data
```

9.3.2 Search Space

To tune the random forest we use maximum depth and maximum features.

The maximum features depends on the number of features in the data. Meanwhile, the maximum theoretical depth is as much as the number of observations, but in practice we do not need even close to that much. We also limit it to the number of features to make the optimization problem easier.

Furthermore, when applying duplication, we set `min_samples_split` to 3 to allow terminal nodes with both duplicates of an observation.

```
def space(duplicate: bool, hard_mode: bool):
    X, _ = load_data(duplicate=duplicate, hard_mode=hard_mode)
    _, n_feat = X.shape
    print(f"    Features: {n_feat}")

    return dict(
        # constants
        n_estimators=Constant(441),
        random_state=Constant(441),

        # variables
        criterion=Categorical(["gini", "log_loss"]),
        max_features=Integer(low=1, high=n_feat, prior="log-uniform"),
        max_depth=Integer(low=1, high=n_feat, prior="log-uniform"),
        min_samples_split=Constant(3 if duplicate else 2),
    )
```

9.3.3 Base

```
outer_cv(
    tuner=SKBayesTuner(SKRandomForest()),
    search=space(duplicate=False, hard_mode=False),
    name="Random Forest",
    duplicate=False,
    hard_mode=False,
)
```

9.3.4 Hard

```
outer_cv(
    tuner=SKBayesTuner(SKRandomForest()),
    search=space(duplicate=False, hard_mode=True),
    name="Random Forest",
    duplicate=False,
    hard_mode=True,
)
```

9.3.5 Duplication

```
outer_cv(
    tuner=SKBayesTuner(DuplicateSKRandomForest()),
    search=space(duplicate=True, hard_mode=False),
    name="Random Forest",
    duplicate=True,
    hard_mode=False,
)
```

9.3.6 Duplication, Hard

```
outer_cv(
    tuner=SKBayesTuner(DuplicateSKRandomForest()),
    search=space(duplicate=True, hard_mode=True),
    name="Random Forest",
    duplicate=True,
    hard_mode=True,
)
```

9.4 Neural Network (Base)

```
# Import libraries and functions
import random
import itertools
import math
import torch
import time
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import torch.nn as nn
import torch.optim as optim
from tqdm.notebook import tqdm
from torch.utils.data import TensorDataset, DataLoader
from sklearn.preprocessing import StandardScaler, OneHotEncoder, MinMaxScaler
from sklearn.model_selection import train_test_split, StratifiedKFold
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
# General Setup
torch.manual_seed(0)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
# load and modify the data
pokemon = pd.read_csv('../data/scraped.csv')

# Define columns
to_drop = []
categorical = ["type_2",'status']
weakness = ['damage_from_normal', 'damage_from_fire', 'damage_from_water', 'damage_from_electric',
            'damage_from_grass', 'damage_from_ice', 'damage_from_fighting', 'damage_from_poison',
            'damage_from_ground', 'damage_from_flying', 'damage_from_psychic', 'damage_from_bug','damag
            'damage_from_ghost', 'damage_from_dragon', 'damage_from_dark','damage_from_steel', 'damage_

# Drop unwanted variables
pokemon = pokemon.drop(columns=to_drop)

# Define class and features
y = pokemon['type_1']; # Class
X = pokemon.drop(['type_1'],axis=1) # Feature

# Store the label names
y_label = pokemon['type_1'] # uncoded features

# Encode categorical variables
y = pd.get_dummies(y)
X = pd.get_dummies(X,columns=categorical)

# Scale numerical variables
scaler = MinMaxScaler()
ending = sum(X.dtypes == ["float64"]*X.shape[1])
X.iloc[:,:ending] = scaler.fit_transform(X.iloc[:,:ending])

# Convert training and testing data into PyTorch tensor
y = torch.tensor(y.values).float()
X = torch.tensor(X.values).float()

# Set up number classes and features
n_class = 18
n_feature = X.shape[1]
```

```
# Define model
class NN(nn.Module):

    # Set up model
    def __init__(self, node_list,drop):
        super(NN, self).__init__()
        self.layers = nn.ModuleList()
        for i in range(1, len(node_list)):
            self.layers.append(nn.Linear(node_list[i-1], node_list[i]))

        self.activation = nn.ReLU() # activation function
        self.dropout = nn.Dropout(p=drop) # drop rate

    # Set up forward function
    def forward(self, x):
        out = x
        for i in range(len(self.layers)-1):
            out = self.layers[i](out)
            out = self.activation(out)
        out = self.layers[len(self.layers)-1](out)
        return out
```

```
# Prediction function
def predict(model, X, y):

    # Set up loader
    test_dataset = TensorDataset(X,y)
    loader = DataLoader(test_dataset, batch_size = 9999, shuffle = False)

    # Set model to evaluation
    model.eval()

    # Evaluate model
    num_correct = 0
    num_samples = 0

    with torch.no_grad():
        for x, y in loader:
            x = x.to(device=device)
            y = y.to(device=device)
            x = x.reshape(x.shape[0], -1)
            model_result = model(x)
            _, prediction = model_result.max(1)
            _, actual = y.max(1)
            num_correct += (prediction == actual).sum()
            num_samples += prediction.size(0)

    # Set model to training
    model.train()

    # Compute accuracy
    accuracy = num_correct/num_samples

    return prediction.cpu().numpy(), actual.cpu().numpy(), accuracy.cpu().item()
```

```python
# Training function
def train_model(num_epochs, batch_size, node_list, drop, X, y):

    # Create the model
    model = NN([n_feature]+node_list+[n_class], drop).to(device)

    # Set up loss function and optimizer
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters())

    # Set up data loader
    train_dataset = TensorDataset(X,y)
    train_loader = DataLoader(train_dataset, batch_size = batch_size, shuffle = True)

    # Initialize loss
    loss = []

    # Train model
    for epoch in range(num_epochs):
        for batch_idx, (data, targets) in enumerate(train_loader):
            # Get data to cuda if possible
            data = data.to(device = device)
            targets = targets.to(device = device)

            # Reshape data
            data = data.reshape(data.shape[0], -1)

            # Forward propagation
            model_result = model(data)
            current_loss = criterion(model_result, targets)

            # Zero previous gradients
            optimizer.zero_grad()

            # back-propagation
            current_loss.backward()

            # optimize
            optimizer.step()

        # Record loss of the current epoch
        loss.append(float(current_loss.item()))

    return model, loss
```

```python
def model_examination(num_epochs, batch_size, node_list, drop, X_train, y_train, X_test, y_test):
    # Train model
    model, loss = train_model(num_epochs, batch_size, node_list, drop, X_train, y_train)
    # Estimate model accuracy
    pred, actual, accuracy = predict(model, X_test, y_test)

    return model, loss, pred, actual, accuracy
```

```
def model_examination_repeated_cv(num_epochs, batch_size, node_list, drop, X, y, n_split, repeat, y_labe

    # Set up and initialize
    skf = StratifiedKFold(n_splits=n_split,random_state=441, shuffle=True)
    acc = []

    # Perform k-fold cv n times, record result for every model trained
    for ii in tqdm(range(repeat),leave=False):
        acc_temp = []
        for idx, (train_index, test_index) in enumerate(skf.split(X,y_label)):
            # Set up test and training variables
            X_test = X[test_index]; y_test = y[test_index];
            X_train = X[train_index]; y_train = y[train_index]
            # train the model and evaluate the accuracy
            model, _ = train_model(num_epochs,batch_size,node_list,drop,X_train,y_train)
            _,_, accuracy = predict(model, X_test, y_test)
            accuracy = accuracy
            acc_temp.append(accuracy)
        acc.append(acc_temp)

    return acc
```

```
def merge_list(lst):
    lst_final = list(itertools.chain.from_iterable(lst))
    return lst_final
```

9.4.1 Check if function works properly

```
num_epochs,batch_size,drop = 300,200,0.2
node_list = [300]
X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.2,random_state=random.seed(time.tim
model, loss, pred, actual, accuracy = model_examination(num_epochs, batch_size, node_list, drop,
                                                         X_train, y_train, X_test, y_test)
print(model)
print(accuracy)
plt.plot(loss)

_,_,pred,actual,accuracy = model_examination(num_epochs,batch_size,node_list,drop,X_train,y_train,X_tes
print(pred)
print(actual)
# acc_test = model_examination_repeated_cv(num_epochs,batch_size,node_list,drop,X,y,5,5,y_label)
# print(acc_test)
# acc_test_merged = merge_list(acc_test)
# print(sum(acc_test_merged)/len(acc_test_merged))
```

9.4.2 Examine how number of layers and nodes affect the performance

```
# Set up initial parameters
max_hidden_layer = 5
node_max_multiplier = 9
num_epochs,batch_size,drop = 400,200,0.2

acc_1 = []
```

```
# Fix number of hidden layer
for i in tqdm(range(max_hidden_layer)):
    # Number of nodes per layer
    for j in tqdm(range(node_max_multiplier),leave=False):
        node_list = [100+25*j]*(i+1)
        acc_list = model_examination_repeated_cv(num_epochs,batch_size,node_list,drop,X,y,5,5,y_label)
        # Record result for each set up
        acc_1.append(acc_list)
```

```
# Create heat map
acc_1_temp = acc_1 # Save the result into a temporary variable that will be modified
print(acc_1_temp)
# transform acc_1_temp into heat map list
hmlst1 = [[],[],[],[],[]]
counter = 0
for i in range(max_hidden_layer):
    for j in range(node_max_multiplier):
        merged_list = merge_list(acc_1_temp[counter])
        hmlst1[i].append(sum(merged_list)/len(merged_list))
        counter += 1

# Create Heat map
fix, ax = plt.subplots(figsize=(6,4.5))
hmlst1 = hmlst1[::-1]
hm1 = sns.heatmap(hmlst1)
hm1.set_xlabel('Number of Nodes')
hm1.set_ylabel('Number of Hidden Layers')
hm1.set_xticklabels(list(range(100,325,25)))
hm1.set_yticklabels(list(range(5,0,-1)))
plt.title('Accuracy Heat Map')
plt.show()
plt.savefig('Stat 441 NN Accuracy Heat map')
```

```
# Create scatterplot for each layer
scatter = [[],[],[],[],[]]
colors = ['#5b60cc', '#ff7941', '#6dd66d', '#c75450', '#88c9ff', '#ffc735', '#949494', '#c98ed0', '#f3a
counter = 0
# Extract the data
for i in range(max_hidden_layer):
    for j in range(node_max_multiplier):
        scatter[i].append(merge_list(acc_1_temp[counter]))
        counter += 1
# Plot for different layers
for ii in range(5):
    fig, ax = plt.subplots()
    for i, lst in enumerate(scatter[ii]):
        x = [i*25+100]*len(lst)
        color_index = i%len(colors)
        ax.scatter(x,lst,color=colors[color_index])
        ax.scatter(i*25+100,sum(lst)/len(lst),marker="_",color='black')
    ax.set_xlabel('Number of nodes')
    ax.set_ylabel('Accuracy')
    ax.set_title('Number of hidden layers = '+str(ii+1))
    plt.show()
    plt.savefig('Stat 441 NN Optimization ' + str(ii+1))
```

```
# Optimize drop rate, batch size and learning rate using 2 layers with 250 nodes per layer
epoch_lst = [300,350,400,450,500]
batch_lst = [100,125,150,175,200,225,250]
drop_lst = [0.2,0.225,0.25,0.275,0.3]

# Optimal node_list
node_list = [250,250]
acc_2 = []
for i in tqdm(range(len(epoch_lst))):
    acc_batch = []
    for j in tqdm(range(len(batch_lst)),leave=False):
        acc_drop = []
        for k in tqdm(range(len(drop_lst)),leave=False):
            drop = drop_lst[k]/100;
            batch_size = batch_lst[j]
            num_epochs = epoch_lst[i]
            acc_list = merge_list(model_examination_repeated_cv(num_epochs,batch_size,node_list,drop,X,
            acc_drop.append(sum(acc_list)/len(acc_list))
        acc_batch.append(acc_drop)
    acc_2.append(acc_batch)
```

```
acc_2_temp = acc_2
print(acc_2_temp)
```

```
# check how epcoh affects the accuracy
epoch_result = []
for i in range(len(epoch_lst)):
    lst_temp = []
    for j in range(len(batch_lst)):
        for k in range(len(drop_lst)):
            lst_temp.append(acc_2_temp[i][j][k])
    epoch_result.append(sum(lst_temp)/len(lst_temp))

batch_result = []
for j in range(len(batch_lst)):
    lst_temp = []
    for i in range(len(epoch_lst)):
        for k in range(len(drop_lst)):
            lst_temp.append(acc_2_temp[i][j][k])
    batch_result.append(sum(lst_temp)/len(lst_temp))

drop_result = []
for k in range(len(drop_lst)):
    lst_temp = []
    for i in range(len(epoch_lst)):
        for j in range(len(batch_lst)):
            lst_temp.append(acc_2_temp[i][j][k])
    drop_result.append(sum(lst_temp)/len(lst_temp))

print(epoch_result)
print(batch_result)
print(drop_result)
```

```
max_index = [0,0,0]
current_max = 0
for i in range(len(epoch_lst)):
    for j in range(len(batch_lst)):
        for k in range(len(drop_lst)):
            val_temp = acc_2_temp[i][j][k]
            if val_temp > current_max:
                max_index = [i,j,k]
                current_max = val_temp
print(max_index)
print(acc_2_temp[max_index[0]][max_index[1]][max_index[2]])

min_index = [0,0,0]
current_min = 1
for i in range(len(epoch_lst)):
    for j in range(len(batch_lst)):
        for k in range(len(drop_lst)):
            val_temp = acc_2_temp[i][j][k]
            if val_temp < current_min:
                min_index = [i,j,k]
                current_min = val_temp
print(min_index)
print(acc_2_temp[min_index[0]][min_index[1]][min_index[2]])
```

The optimal result reached an accuracy of 0.993175 with epoch = 350, batch = 125, drop rate = 0.2 and the worst result has an accuracy of 0.9888 where epoch = 300, batch = 250, drop = 0.225. The difference between the two prediction accuracy is very small and its likely caused by the stochasticity during fitting. This implies optimization over epoch, batch, drop rate is rather unnecessary as optimzing them do not improve accuracy drastically. A potential reason is that type_2 and damage_from variable plays very important role determining the types of pokemons. Therefore, there is a clear path for the trainer to optimize the weights and biases and lead to similar results everytime. In other wrod, the examined epoch, batch, drop could not affect the training process effectively, not even in the negative manner.

```
# Do a random nested CV to examine the NN performence,
skf = StratifiedKFold(n_splits = 5, random_state = 441, shuffle = True)
acc = []
pred_lst = []
actual_lst = []
for idx, (train_index, test_index) in enumerate(skf.split(X,y_label)):
    print('Split: '+ str(idx+1))
    # Split the data
    X_test = X[test_index]; y_test = y[test_index]; y_label_test = y_label[test_index];
    X_train = X[train_index]; y_train = y[train_index]; y_label_train = y_label[train_index];
    # Set up the parameters
    epoch_final = 300
    batch_final = 100
    node_final = [250,250]
    drop_final = 0.2
    # Examine the best parameter optimized before
    acc_inner_cv = merge_list(model_examination_repeated_cv(epoch_final,batch_final,node_final,drop_fina
                                                            X_train,y_train,5,4,y_label_train))
    acc_current = sum(acc_inner_cv)/len(acc_inner_cv)
    # Randomly generate 19 hyperparameters and see if they beat the optimized hyperparameters, if so re
    for i in tqdm(range(19),leave=False):
        num_epochs = random.randint(275,325)
        batch_size = random.randint(75,125)
        node_list = [random.randint(225,275),random.randint(225,275)]
        drop = random.randint(17,23)/100
        acc_inner_cv = merge_list(model_examination_repeated_cv(num_epochs,batch_size,node_list,drop,
                                                            X_train,y_train,5,4,y_label_train))
        acc_temp = sum(acc_inner_cv)/len(acc_inner_cv)
        if acc_temp > acc_current:
            epoch_final = num_epochs
            batch_final = batch_size
            node_final = node_list
            drop_final = drop
            acc_current = acc_temp

    # With the best hyper parameters found, we examine its performance
    _,_,pred,actual, accuracy_final = model_examination(epoch_final, batch_final, node_final, drop, X_t
    acc.append(accuracy_final)
    pred_lst.append(pred)
    actual_lst.append(actual)
```

```
# Store the variables
acc_temp = acc
pred_temp = pred_lst
actual_temp = actual_lst
print(acc_temp)
print(sum(acc_temp)/len(acc_temp))
```

```
# Compute confusion matrix
# Set up label names
labels = y_label.unique().tolist()
labels.sort()

# Compute final prediction and final actual
pred_final = merge_list(pred_temp)
actual_final = merge_list(actual_temp)

# Create confusion matrix and normalize
cm = confusion_matrix(actual_final, pred_final, normalize='true')

# Set up plot
fix, ax = plt.subplots(figsize=(10,10))
ax.set_xticks(range(len(labels)))
ax.set_xticklabels(labels)
ax.set_title('Confusion Matrix for Neural Network')
# display
disp = ConfusionMatrixDisplay(confusion_matrix=cm,display_labels=labels)
disp.plot(xticks_rotation=45, values_format='.0%',ax=ax)
plt.show()
plt.savefig('Stat 441 NN CM')
```

9.5 Neural Network (Hard)

```
# Import libraries and functions
import random
import itertools
import math
import torch
import time
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import torch.nn as nn
import torch.optim as optim
from tqdm import tqdm
from torch.utils.data import TensorDataset, DataLoader
from sklearn.preprocessing import StandardScaler, OneHotEncoder, MinMaxScaler
from sklearn.model_selection import train_test_split, StratifiedKFold
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
# General Setup
torch.manual_seed(0)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
# load and modify the data
pokemon = pd.read_csv('../data/scraped.csv')
# Store the labels for  later use
type_1_label = pokemon['type_1'].tolist()
type_2_label = pokemon['type_2'].tolist()

# Define columns
to_drop = ["type_2"]

categorical = ['status']

weakness = ['damage_from_normal', 'damage_from_fire', 'damage_from_water', 'damage_from_electric',
            'damage_from_grass', 'damage_from_ice', 'damage_from_fighting', 'damage_from_poison',
            'damage_from_ground', 'damage_from_flying', 'damage_from_psychic', 'damage_from_bug','damage
            'damage_from_ghost', 'damage_from_dragon', 'damage_from_dark','damage_from_steel', 'damage_

# Drop unwanted variables
pokemon = pokemon.drop(columns=to_drop+weakness)

# Define class and features
y = pokemon['type_1']; # Class
X = pokemon.drop(['type_1'],axis=1) # Feature

# Store the label names
y_label = pokemon['type_1']

# Encode categorical variables
y = pd.get_dummies(y)
X = pd.get_dummies(X,columns=categorical)

# Scale numerical variables
scaler = MinMaxScaler()
```

```python
ending = sum(X.dtypes == ["float64"]*X.shape[1])
X.iloc[:,:ending] = scaler.fit_transform(X.iloc[:,:ending])

# Convert training and testing data into PyTorch tensor
y = torch.tensor(y.values).float()
X = torch.tensor(X.values).float()

# Set up number classes and features
n_class = 18
n_feature = X.shape[1]
```

```python
# Define model
class NN(nn.Module):

    def __init__(self, node_list,drop):
        super(NN, self).__init__()

        self.layers = nn.ModuleList()
        for i in range(1, len(node_list)):
            self.layers.append(nn.Linear(node_list[i-1], node_list[i]))

        self.activation = nn.ReLU()
        self.dropout = nn.Dropout(p=drop)

    def forward(self, x):
        out = x
        for i in range(len(self.layers)-1):
            out = self.layers[i](out)
            out = self.activation(out)
        out = self.layers[len(self.layers)-1](out)
        return out
```

```python
# Prediction function
def predict(model, X, y):

    # Set up loader
    test_dataset = TensorDataset(X,y)
    loader = DataLoader(test_dataset, batch_size = 9999, shuffle = False)

    # Set model to evaluation
    model.eval()

    # Evaluate model
    num_correct = 0
    num_samples = 0

    with torch.no_grad():
        for x, y in loader:
            x = x.to(device=device)
            y = y.to(device=device)
            x = x.reshape(x.shape[0], -1)
            model_result = model(x)
            _, prediction = model_result.max(1)
            _, actual = y.max(1)
            num_correct += (prediction == actual).sum()
            num_samples += prediction.size(0)

    # Set model to training
    model.train()
    accuracy = num_correct/num_samples

    return prediction.cpu().numpy(), actual.cpu().numpy(), accuracy.cpu().item()
```

```
# Training function
# Input: Model hyper parameters, training and testing data
# Output: Model, Loss
def train_model(num_epochs, batch_size, node_list, drop, X, y):

    # Create the model
    model = NN([n_feature]+node_list+[n_class], drop).to(device)

    # Set up loss function and optimizer
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters())

    # Set up data loader
    train_dataset = TensorDataset(X,y)
    train_loader = DataLoader(train_dataset, batch_size = batch_size, shuffle = True)

    # Initialize loss
    loss = []

    # Train model
    for epoch in range(num_epochs):
        for batch_idx, (data, targets) in enumerate(train_loader):
            # Get data to cuda if possible
            data = data.to(device = device)
            targets = targets.to(device = device)

            # Reshape data
            data = data.reshape(data.shape[0], -1)

            # Forward propagation
            model_result = model(data)
            current_loss = criterion(model_result, targets)

            # Zero previous gradients
            optimizer.zero_grad()

            # back-propagation
            current_loss.backward()

            # optimize
            optimizer.step()

        # Record loss of the current epoch
        loss.append(float(current_loss.item()))
    return model, loss
```

```
def model_examination(num_epochs, batch_size, node_list, drop, X_train, y_train, X_test, y_test):
    # Train model
    model, loss = train_model(num_epochs, batch_size, node_list, drop, X_train, y_train)
    # Estimate model accuracy
    pred, actual, accuracy = predict(model, X_test, y_test)

    return model, loss, pred, actual, accuracy
```

```
def model_examination_repeated_cv(num_epochs, batch_size, node_list, drop, X, y, n_split, repeat, y_labe

    # Set up and initialize
    skf = StratifiedKFold(n_splits=n_split,random_state=441, shuffle=True)
    acc = []

    # Perform k-fold cv n times, record result for every model trained
    for ii in tqdm(range(repeat),leave=False):
        acc_temp = []
        for idx, (train_index, test_index) in enumerate(skf.split(X,y_label)):
            X_test = X[test_index]
            y_test = y[test_index]
            X_train = X[train_index]
            y_train = y[train_index]
            model, _ = train_model(num_epochs,batch_size,node_list,drop,X_train,y_train)
            _,_, accuracy = predict(model, X_test, y_test)
            accuracy = accuracy
            acc_temp.append(accuracy)
        acc.append(acc_temp)

    return acc
```

```
def merge_list(lst):
    lst_final = list(itertools.chain.from_iterable(lst))
    return lst_final
```

9.5.1 Check if function works properly

```
num_epochs,batch_size,drop = 300,250,0.3
node_list = [1300,1300]
X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.2,random_state=random.seed(time.tim
model, loss, pred, actual, accuracy = model_examination(num_epochs, batch_size, node_list, drop,
                                                        X_train, y_train, X_test, y_test)
print(model)
print(accuracy)
plt.plot(loss)
```

```
acc_test = model_examination_repeated_cv(num_epochs,batch_size,node_list,drop,X,y,5,5,y_label)
```

```
print(acc_test)
acc_test_merged = merge_list(acc_test)
print(sum(acc_test_merged)/len(acc_test_merged))
```

9.5.2 Examine how number of layers and nodes affect the performance

```
max_hidden_layer = 4
node_max_multiplier = 11
num_epochs,batch_size,drop = 600,200,0.2

acc_1 = []

# Fix number of hidden layer
for i in tqdm(range(max_hidden_layer)):
    # Number of nodes per layer
    for j in tqdm(range(node_max_multiplier),leave=False):
        node_list = [1000+100*j]*(i+1)
        acc_list =model_examination_repeated_cv(num_epochs,batch_size,node_list,drop,X,y,5,5,y_label)
        # Record result for each set up
        acc_1.append(acc_list)
```

```
# Create heat map
acc_1_temp = acc_1 # Save the result into a temporary variable that will be modified

# transform acc_1_temp into heat map list
hmlst1 = [[],[],[],[]]
counter = 0
for i in range(max_hidden_layer):
    for j in range(node_max_multiplier):
        merged_list = merge_list(acc_1_temp[counter])
        hmlst1[i].append(sum(merged_list)/len(merged_list))
        counter += 1
hmlst1 = hmlst1[::-1]
hm1 = sns.heatmap(hmlst1)
hm1.set_xlabel('Number of Nodes')
hm1.set_ylabel('Number of Hidden Layers')
hm1.set_xticklabels(list(range(1000,2100,100)))
hm1.set_yticklabels(list(range(4,0,-1)))
plt.title('Accuracy Heat Map')
plt.show()
plt.savefig('Stat 441 NN Accuracy Heat Map without')
```

```
# Create scatterplot for each layer
scatter = [[],[],[],[]]
colors = ['#5b60cc', '#ff7941', '#6dd66d', '#c75450', '#88c9ff', '#ffc735', '#949494', '#c98ed0', '#f3a
counter = 0
# Extract the data
for i in range(max_hidden_layer):
    for j in range(node_max_multiplier):
        scatter[i].append(merge_list(acc_1_temp[counter]))
        counter += 1

# Plot for different layers
for ii in range(4):
    fig, ax = plt.subplots()
    for i, lst in enumerate(scatter[ii]):
        x = [i*100+1000]*len(lst)
        color_index = i%len(colors)
        ax.scatter(x,lst,color=colors[color_index])
        ax.scatter(i*100+1000,sum(lst)/len(lst),marker="_",color='black')
    ax.set_xlabel('Number of nodes')
    ax.set_ylabel('Accuracy')
    ax.set_title('Number of hidden layers = '+str(ii+1))
    plt.show()
    plt.savefig('Stat 441 NN Optimization '+str(ii+1)+' without')
```

It appeared from the heat map that [1900,1900,1900] performed the best. However, if we examine the data in detail, we realize that [1300,1300] is also performing well. It's average accuracy is considerably high and has a better worst computed prediction accuracy comparing to other models. Also, it is easier to fit a [1300,1300] list. Thus we will optimize further using node list [1300,1300]

```
# Optimize drop rate, batch size and learning rate using 2 layers with 250 nodes per layer
epoch_lst = [100,150,200,250,300]
batch_lst = [100,125,150,175,200,225,250]
drop_lst = [0.2,0.225,0.25,0.275,0.3]

# Optimal node_list
node_list = [1300,1300]

acc_2 = []
for i in tqdm(range(len(epoch_lst))):
    acc_batch = []
    for j in tqdm(range(len(batch_lst)),leave=False):
        acc_drop = []
        for k in tqdm(range(len(drop_lst)),leave=False):
            drop = drop_lst[k]/100;
            batch_size = batch_lst[j]
            num_epochs = epoch_lst[i]
            acc_list = merge_list(model_examination_repeated_cv(num_epochs,batch_size,node_list,drop,X,
            acc_drop.append(sum(acc_list)/len(acc_list))
        acc_batch.append(acc_drop)
    acc_2.append(acc_batch)
```

```
acc_2_temp = acc_2
print(acc_2_temp)
```

```
# check how epcoh affects the accuracy
epoch_result = []
for i in range(len(epoch_lst)):
    lst_temp = []
    for j in range(len(batch_lst)):
        for k in range(len(drop_lst)):
            lst_temp.append(acc_2_temp[i][j][k])
    epoch_result.append(sum(lst_temp)/len(lst_temp))

batch_result = []
for j in range(len(batch_lst)):
    lst_temp = []
    for i in range(len(epoch_lst)):
        for k in range(len(drop_lst)):
            lst_temp.append(acc_2_temp[i][j][k])
    batch_result.append(sum(lst_temp)/len(lst_temp))
```

```python
drop_result = []
for k in range(len(drop_lst)):
    lst_temp = []
    for i in range(len(epoch_lst)):
        for j in range(len(batch_lst)):
            lst_temp.append(acc_2_temp[i][j][k])
    drop_result.append(sum(lst_temp)/len(lst_temp))

print(epoch_result)
print(batch_result)
print(drop_result)

max_index = [0,0,0]
current_max = 0
for i in range(len(epoch_lst)):
    for j in range(len(batch_lst)):
        for k in range(len(drop_lst)):
            val_temp = acc_2_temp[i][j][k]
            if val_temp > current_max:
                max_index = [i,j,k]
                current_max = val_temp
print(max_index)
print(acc_2_temp[max_index[0]][max_index[1]][max_index[2]])

min_index = [0,0,0]
current_min = 1
for i in range(len(epoch_lst)):
    for j in range(len(batch_lst)):
        for k in range(len(drop_lst)):
            val_temp = acc_2_temp[i][j][k]
            if val_temp < current_min:
                min_index = [i,j,k]
                current_min = val_temp
print(min_index)
print(acc_2_temp[min_index[0]][min_index[1]][min_index[2]])
```

Unlike the training with type_2 and damage_from variable in which case we realized optimizing for number of epochs, batch size and drop rate are unnecessary, the training without type_2 and damage_from seem to depend on these hyperparameters. In particular, we noticed that the optimal model is trained with epoch = 200, batch = 100 and drop = 0.275 and has a prediction accuracy 0.459. The worst model is trained with epcoh 100, batch 225, drop rate = 0.25 with prediction accuracy 0.43536. We also noticed that increasing batch size tend to lead to a worse performance. We believe this is because the data is noisy and difficult to generalize. Thus, a smaller batch allows the model to analyze the characteristics of data more frequently and in a detailed way.

```python
# Do a random nested CV to examine the NN performence,
skf = StratifiedKFold(n_splits = 5, random_state = 441, shuffle = True)
acc = []
pred_lst = []
actual_lst = []
for idx, (train_index, test_index) in enumerate(skf.split(X,y_label)):
    print('Split: '+ str(idx+1))
    # Split the data
    X_test = X[test_index]; y_test = y[test_index]; y_label_test = y_label[test_index];
    X_train = X[train_index]; y_train = y[train_index]; y_label_train = y_label[train_index];
    # Set up the parameters
    epoch_final = 200
    batch_final = 100
    node_final = [1300,1300]
    drop_final = 0.275
    # Examine the best parameter optimized before
    acc_inner_cv = merge_list(model_examination_repeated_cv(epoch_final,batch_final,node_final,drop_fina
                                                            X_train,y_train,5,4,y_label_train))
    acc_current = sum(acc_inner_cv)/len(acc_inner_cv)
    # Randomly generate 19 hyperparameters and see if they beat the optimized hyperparameters, if so re
    for i in tqdm(range(19),leave=False):
        num_epochs = random.randint(175,225)
        batch_size = random.randint(75,125)
        node_list = [random.randint(1200,1400),random.randint(1200,1400)]
        drop = random.randint(25,30)/100
        acc_inner_cv = merge_list(model_examination_repeated_cv(num_epochs,batch_size,node_list,drop,
                                                                X_train,y_train,5,4,y_label_train))
        acc_temp = sum(acc_inner_cv)/len(acc_inner_cv)
        if acc_temp > acc_current:
            epoch_final = num_epochs
            batch_final = batch_size
            node_final = node_list
            drop_final = drop
            acc_current = acc_temp

    # With the best hyper parameters found, we examine its performance
    _,_,pred,actual, accuracy_final = model_examination(epoch_final, batch_final, node_final, drop, X_tr
    acc.append(accuracy_final)
    pred_lst.append(pred)
    actual_lst.append(actual)
```

```
# Store the variables
acc_temp = acc
pred_temp = pred_lst
actual_temp = actual_lst
print(acc_temp)
print(sum(acc_temp)/len(acc_temp))
```

```
# Compute confusion matrix
# Set up label names
labels = y_label.unique().tolist()
labels.sort()

# Compute final prediction and final actual
pred_final = merge_list(pred_temp)
actual_final = merge_list(actual_temp)

# Create confusion matrix and normalize
cm = confusion_matrix(actual_final, pred_final, normalize='true')

# Set up plot
fix, ax = plt.subplots(figsize=(10,10))
ax.set_xticks(range(len(labels)))
ax.set_xticklabels(labels)
ax.set_title('Confusion Matrix for Neural Network')
# display
disp = ConfusionMatrixDisplay(confusion_matrix=cm,display_labels=labels)
disp.plot(xticks_rotation=45, values_format='.0%',ax=ax)
plt.show()
plt.savefig('Stat 441 NN CM without')
```

```
# Type1 label vs type2 label, y-axis being type1, x-axis being type2
for i in range(len(type_2_label)):
    if type_2_label[i] == "None":
        type_2_label[i] = type_1_label[i]
# Create confusion matrix
cm2 = confusion_matrix(type_1_label,type_2_label)

# Set up plot
fix,ax = plt.subplots(figsize=(10,10))
ax.set_xticks(range(len(labels)))
ax.set_xticklabels(labels)
ax.set_title('First type VS Second type')
# Display
disp2 = ConfusionMatrixDisplay(confusion_matrix=cm2,display_labels=labels)
disp2.plot(xticks_rotation=45,ax=ax)
plt.xlabel('Second type')
plt.ylabel('First type')
plt.show()
plt.savefig('Stat 441 type1 vs type2 CM without')
```

9.6 model/__init__.py

```
from .model import Model, Config
from .sk_model import (
    SKModel,
    SKClassifier,
    SKLogisticRegression,
    SKSupportVectorMachine,
    SKRandomForest,
)
from .duplication import (
    DuplicateSKLogisticRegression,
    DuplicateSKRandomForest,
    DuplicateSKSupportVectorMachine,
)
```

9.7 model/duplication.py

```
import numpy as np

from util import duplicate, multiset_accuracy, predict_multiset_indices
from .sk_model import (
    SKLogisticRegression,
    SKSupportVectorMachine,
    SKRandomForest,
```

```python
)


class DuplicateSKRandomForest(SKRandomForest):
    """A multiset random forest classifier using the duplication method.

    Implements the duplication method by intercepting calls to fit and predict
    on the wrapped classifier. When fitting, it converts a set of multi-label
    training observations into num_labels sets of single-label training
    observations. When predicting, it selects the top num_labels labels (as a
    multiset) instead of predicting a single label.
    """

    @property
    def cardinality(self) -> int:
        """The cardinality of the multisets predicted by the model.

        Override this for different cardinality.

        Returns:
            cardinality (int): The size (counting duplicates) of the multiset of
                labels predicted by the model.
        """
        return 2

    def fit(self, X: np.ndarray, y: np.ndarray) -> None:
        """Fit the model on the given data.

        Each observation is duplicated self.cardinality times, such that

        """
        X, y = duplicate(X, y)
        super().fit(X, y)

    def predict(self, X: np.ndarray) -> np.ndarray:
        """Predict num_labels for each observation.
        """
        p = self.predict_proba(X)
        ids = predict_multiset_indices(p, cardinality=self.cardinality)
        return self.classes_[ids]

    def predict_proba(self, X: np.ndarray) -> np.ndarray:
        """Predict the expected multiplicity of each label.
        """
        return self.cardinality * super().predict_proba(X)

    def score(self, X: np.ndarray, y: np.ndarray) -> float:
        return multiset_accuracy(y_true=y, y_pred=self.predict(X))


class DuplicateSKLogisticRegression(SKLogisticRegression):
    """A multiset logistic regression classifier using the duplication method.
    Implements the duplication method by intercepting calls to fit and predict
    on the wrapped classifier. When fitting, it converts a set of multi-label
    training observations into num_labels sets of single-label training
    observations. When predicting, it selects the top num_labels labels (as a
    multiset) instead of predicting a single label.
    """

    @property
    def cardinality(self) -> int:
        """The cardinality of the multisets predicted by the model.

        Override this for different cardinality.

        Returns:
            cardinality (int): The size (counting duplicates) of the multiset of
                labels predicted by the model.
        """
        return 2

    def fit(self, X: np.ndarray, y: np.ndarray) -> None:
        """Fit the model on the given data.

        Each observation is duplicated self.cardinality times, such that

        """
        X, y = duplicate(X, y)
        super().fit(X, y)

    def predict(self, X: np.ndarray) -> np.ndarray:
        """Predict num_labels for each observation.
        """
        p = self.predict_proba(X)
        ids = predict_multiset_indices(p, cardinality=self.cardinality)
        return self.classes_[ids]

    def predict_proba(self, X: np.ndarray) -> np.ndarray:
        """Predict the expected multiplicity of each label.
        """
        return self.cardinality * super().predict_proba(X)

    def score(self, X: np.ndarray, y: np.ndarray) -> float:
        return multiset_accuracy(y_true=y, y_pred=self.predict(X))


class DuplicateSKSupportVectorMachine(SKSupportVectorMachine):
    """A multiset support vector classifier using the duplication method.

    Implements the duplication method by intercepting calls to fit and predict
    on the wrapped classifier. When fitting, it converts a set of multi-label
    training observations into num_labels sets of single-label training
    observations. When predicting, it selects the top num_labels labels (as a
    multiset) instead of predicting a single label.
    """
    @property
    def cardinality(self) -> int:
        """The cardinality of the multisets predicted by the model.

        Override this for different cardinality.

        Returns:
            cardinality (int): The size (counting duplicates) of the multiset of
                labels predicted by the model.
        """
        return 2

    def fit(self, X: np.ndarray, y: np.ndarray) -> None:
        """Fit the model on the given data.

        Each observation is duplicated self.cardinality times, such that

        """
        X, y = duplicate(X, y)
        super().fit(X, y)

    def predict(self, X: np.ndarray) -> np.ndarray:
        """Predict num_labels for each observation.
        """
        p = self.predict_proba(X)
        ids = predict_multiset_indices(p, cardinality=self.cardinality)
        return self.classes_[ids]

    def predict_proba(self, X: np.ndarray) -> np.ndarray:
        """Predict the expected multiplicity of each label.
        """
        return self.cardinality * super().predict_proba(X)

    def score(self, X: np.ndarray, y: np.ndarray) -> float:
        return multiset_accuracy(y_true=y, y_pred=self.predict(X))
```

9.8 model/model.py

```
from abc import ABC, abstractmethod
import joblib
from typing import Any, Type, TypeVar

import numpy as np


Self = TypeVar("Self", bound="Model")
Config = dict[str, Any]


class Model(ABC):
    """An abstract K-class M-label multiset classification model.
    Predicts on P-dimensional inputs. That is, R^P -> {1, ..., K}^M.
    When M=1, this is regular single-label classification.

    Args:
        config (Config): The hyperparameters of the model.
    """

    def __init__(self, config: Config):
        assert isinstance(config, dict)
        for k in config.keys():
            assert isinstance(k, str)

        self.config = config

    def predict(self, X: np.ndarray) -> np.ndarray:
        """Predict class labels.

        Args:
            X (ndarray): [N x P] predictor values.

        Returns:
            y (ndarray): [N] (if M=1) or [N x M] predicted class labels.
        """
        p = self.predict_probabilities(X)
        return self.labels[p.argmax(axis=-1)]

    @abstractmethod
    def predict_probabilities(self, X: np.ndarray) -> np.ndarray:
        """Predict class probabilities.

        When M>=2, it is more accurate to think of the probabilities as the
        "expected multiplicity" of each class, so it is possible for a value
        larger than 1 to reflect an element that occurs multiple times. This is
        consistent with single-label classification when M=1.

        The total probabilities must sum to M.

        Args:
            X (ndarray): [N x P] predictor values.

        Returns:
            p (ndarray): [N x K] predicted "probabilities", where p[i, k] is the
                expected multiplicity of class k in y[i].
        """
        pass

    @property
    @abstractmethod
    def labels(self) -> np.ndarray:
        """The label names associated with the model.

        Returns:
            labels (ndarray): [K] label names of the data, such that labels[k]
```

```
                is the name of the kth label.
        """
        pass

    def save(self, path: str) -> None:
        """Save the model to the specified path.

        Args:
            path (str): The path where the model will be stored.
        """
        joblib.dump(self, path)

    @classmethod
    def load(cls: Type[Self], path: str) -> Self:
        """Load model from the specified path.

        Args:
            path (str): The path to the model file.

        Returns:
            model (Model): A model of this class's type.
        """
        model = joblib.load(path)
        assert isinstance(model, cls)
        return model
```

9.9 model/sk_model.py

```
from typing import Generic, TypeVar

import numpy as np
from sklearn.base import BaseEstimator
from sklearn.ensemble import RandomForestClassifier as SKRandomForest
from sklearn.linear_model import LogisticRegression as SKLogisticRegression
from sklearn.svm import SVC as SKSupportVectorMachine

from .model import Model, Config

SKClassifier = TypeVar("SKClassifier", bound=BaseEstimator)

class SKModel(Model, Generic[SKClassifier]):
    """Wrapper for an sklearn model.

    Args:
        model (SKClassifier): An sklearn classification model. Must support
            predict() and predict_proba() methods. e.g.: LogisticRegression
        config (Config): The hyperparameters of the model.
    """

    def __init__(self, model: SKClassifier, config: Config):
        assert isinstance(model, BaseEstimator)

        super().__init__(config)
        self._model = model

    def predict(self, X: np.ndarray) -> np.ndarray:
        """Predict class labels.

        Args:
            X (ndarray): [N x P] Predictor values.

        Returns:
            y (ndarray): [N] Predicted class labels.
        """
        return self._model.predict(X)

    def predict_probabilities(self, X: np.ndarray) -> np.ndarray:
        """Predict class probabilities.

        Args:
            X (ndarray): [N x P] Predictor values.

        Returns:
            p (ndarray): [N x K] Predicted probabilities, where p[i, k] is the
                probability that X[i] is class k.
        """
        return self._model.predict_proba(X)

    @property
    def labels(self) -> np.ndarray:
        """The class names associated with the model.

        Returns:
            labels (ndarray): [K] Label names of the data, such that labels[k]
                is the name of the kth label.
        """
        return self._model.classes_
```

9.10 util/dict.py

```
from typing import Generic, TypeVar
from difflib import get_close_matches


V = TypeVar('V')


class FuzzyDict(Generic[V]):
    """A string-keyed dictionary of values V with fuzzy matching for keys.

    Args:
        items (dict[str, V]): A normal string-keyed dictionary with values V.
    """
    def __init__(self, items: dict[str, V]):
        assert len(items) > 0
        self._items = items
        self._n_items = len(self._items)
        self._keys = list(self._items.keys())

    def get(self, key: str) -> V:
        """Retrieve a value stored at or near the given key.

        Args:
            key (str): The key.

        Returns:
            value (V): The value.
        """
        if (key in self._items):
            return self._items[key]
        # hack to get around Darmanitan mapping to Galarian
        elif key.startswith("Darmanitan "):
            return self.get(key[len("Darmanitan "):])
        closest, = get_close_matches(
            word=key,
            possibilities=self._keys,
            n=1,
            cutoff=0.0,
        )
        print(f"'{key}' not found, falling back to '{closest}'")
        return self._items[closest]
```

9.11 util/soup.py

```
from os import path

from bs4 import BeautifulSoup
import requests


def load_soup(load_path: str) -> BeautifulSoup:
    """Load a web element from a local file.

    Args:
        load_path (str): The path to the local file.

    Returns:
        soup (BeautifulSoup): The web element.
    """
    with open(load_path, "r") as f:
        soup = BeautifulSoup(f.read(), 'html.parser')
    return soup


def save_soup(soup: BeautifulSoup, save_path: str) -> None:
    """Save a web element to a local file.

    Args:
        soup (BeautifulSoup): The web element.
        save_path (str): The path to the local file.
    """
    with open(save_path, "w") as f:
        f.write(str(soup))


def fetch_soup(url: str, cache_path: str) -> BeautifulSoup:
    """Retrieve a webpage.

    Args:
        url (str): The URL of the webpage.
        cache_path (str): The path to a local file where the page is cached.

    Returns:
        soup (BeautifulSoup): The webpage.
    """
    if path.exists(cache_path):
        # print(f"Resource {url} found in cache at {cache_path}")
        soup = load_soup(cache_path)
    else:
        # print(f"Resource {url} not in cache, fetching...")
        req = requests.get(url)
        soup = BeautifulSoup(req.text, 'html.parser')
        # print(f"Saving resource {url} to cache at {cache_path}")
        save_soup(soup, cache_path)
    return soup


def parse_str(soup: BeautifulSoup) -> str:
    """Parse a webpage element that has a string.

    Args:
        soup (BeautifulSoup): The webpage element.

    Returns:
        value (str): The string, with whitespace trimmed.
    """

    return soup.text.strip()


def parse_int(soup: BeautifulSoup) -> int:
    """Parse a webpage element that has an integer.

    For example, an element containing 441 (and possibly some text after) is
    parsed as 441.

    Args:
        soup (BeautifulSoup): The webpage element.

    Returns:
        value (integer): The value of the integer.
    """

    return int(parse_str(soup).split()[0])


def parse_float(soup: BeautifulSoup) -> int:
    """Parse a webpage element that has a decimal number.

    For example, an element containing 44.1 (and possibly some text after) is
    parsed as 44.1.

    Args:
        soup (BeautifulSoup): The webpage element.

    Returns:
        value (float): The value of the decimal number.
    """
    return float(parse_str(soup).split()[0])


def parse_percent(soup: BeautifulSoup) -> float:
    """Parse a webpage element that has a percentage.

    For example, an element containing 44.1% (and possibly some text after) is
    parsed as 44.1.

    Args:
        soup (BeautifulSoup): The webpage element.

    Returns:
        value (float): The value of the percentage.
    """
    raw = parse_str(soup).split()[0]
    assert raw.endswith("%")
    return float(raw[:-1])
```

9.12 util/sprite.py

```
from functools import cached_property
from os import path

import imageio.v3 as iio
import requests
import numpy as np


class Sprite:
    """A low-resolution image of a Pokemon.

    Args:
        img (ndarray): [H x W x C] A 3-D array of RGBA values.
    """
    def __init__(self, img: np.ndarray):
```

```python
        height, width, channels = img.shape
        assert height == 42
        assert width in (52, 56)
        assert channels == 4

        self.img = img / 255.  # normalize the image
        self.height = height
        self.width = width

    @staticmethod
    def fetch(url: str, cache_path: str) -> "Sprite":
        """Retrieves a sprite image from the given URL.

        Args:
            url (str): The URL of the image.
            cache_path (str): The path to a local file to cache the image at.

        Returns:
            sprite (Sprite): The sprite.
        """
        if not path.exists(cache_path):
            req = requests.get(url)
            with open(cache_path, "wb") as f:
                f.write(req.content)
        img = iio.imread(cache_path, extension=".png", mode="RGBA")
        return Sprite(img)

    @property
    def red(self) -> np.ndarray:
        """The red channel of the sprite.

        Returns:
            red (ndarray): [H x W] The red value of each pixel, from 0 to 1.
        """
        return self.img[:, :, 0]

    @property
    def green(self) -> np.ndarray:
        """The green channel of the sprite.

        Returns:
            green (ndarray): [H x W] The green value of each pixel, from 0 to 1.
        """
        return self.img[:, :, 1]

    @property
    def blue(self) -> np.ndarray:
        """The blue channel of the sprite.

        Returns:
            blue (ndarray): [H x W] The blue value of each pixel, from 0 to 1.
        """
        return self.img[:, :, 2]
    @cached_property
    def brightness(self) -> np.ndarray:
        """The brightness channel of the sprite.

        Returns:
            brightness (ndarray): [H x W] The brightness of each pixel, from 0
                to 1.
        """
        return (self.red + self.green + self.blue) / 3.

    @property
    def alpha(self) -> np.ndarray:
        """The opacity channel of the sprite.

        Returns:
            alpha (ndarray): [H x W] The opacity of each pixel, from 0 to 1.
        """
        return self.img[:, :, 3]

    @cached_property
    def perimeter(self) -> np.ndarray:
        """The perimeter channel of the sprite.

        A perimeter pixel is any opaque pixel that is either on the border of
        the sprite, or orthogonally adjacent to a transparent pixel.

        Returns:
            perimeter (ndarray): [H x W] 1 if the pixel is on the perimeter or
                0 otherwise.
        """
        is_perimeter = np.zeros_like(self.alpha)
        for i in range(self.height):
            for j in range(self.width):
                if self.alpha[i, j] > 0.:
                    if (
                        i == 0 or i == self.height - 1
                        or j == 0 or j == self.width - 1
                    ):
                        is_perimeter[i, j] = 1.
                    elif (
                        self.alpha[i-1, j] == 0.
                        or self.alpha[i+1, j] == 0.
                        or self.alpha[i, j-1] == 0.
                        or self.alpha[i, j+1] == 0.
                    ):
                        is_perimeter[i, j] = 1.
        return is_perimeter
```

9.13 scrape/__init__.py

```python
from .variant import Variant
from .scrape import all_variants
```

9.14 scrape/scrape.py

```python
from os import path, mkdir

from bs4 import BeautifulSoup
from typing import Iterator

from .util.soup import fetch_soup, load_soup, parse_str, save_soup
from .util.sprite import Sprite
from .variant import Variant


POKEDEX_URL = "https://pokemondb.net/pokedex/"
ALL_URL = path.join(POKEDEX_URL, "all")

SCRIPT_PATH = path.realpath(__file__)

BASE_CACHE_DIR = path.join(path.dirname(SCRIPT_PATH), ".cache")
if not path.exists(BASE_CACHE_DIR):
    mkdir(BASE_CACHE_DIR)

POKEMON_CACHE_DIR = path.join(BASE_CACHE_DIR, "pokemon")
if not path.exists(POKEMON_CACHE_DIR):
    mkdir(POKEMON_CACHE_DIR)

VARIANT_CACHE_DIR = path.join(BASE_CACHE_DIR, "variant")
if not path.exists(VARIANT_CACHE_DIR):
    mkdir(VARIANT_CACHE_DIR)

SPRITE_CACHE_DIR = path.join(BASE_CACHE_DIR, "sprite")
if not path.exists(SPRITE_CACHE_DIR):
    mkdir(SPRITE_CACHE_DIR)


def safe_name(pokemon_name: str, variant_name: str | None = None) -> str:
    """Get a normalized name for the Pokemon.

    Spaces, punctuation, and special symbols are removed.

    Args:
        pokemon_name (str): The Pokemon's name.
        variant_name (str): The Pokemon's variant's name.

    Return:
        safe_name (str): The normalized name.
    """
    name = (
        pokemon_name if variant_name is None
        else f'{pokemon_name}+{variant_name}'
    )
    name = name.replace(" ", "_")  # flatten all spaces
    name = name.replace(".", "").replace("'", "")  # remove punctuation
    name = name.replace(u"\u2640", "-f").replace(u"\u2642", "-m")  # unroll genders
    name = name.lower()  # lowercase
    return name


def fetch_pokemon_soup(dex_path: str) -> BeautifulSoup:
    """Fetch the Pokemon's data.

    Args:
        dex_path (str): The path to the Pokemon's data in PokemonDB.

    Return:
        soup (BeautifulSoup): The HTML web data of the Pokemon.
    """
    url = path.join(POKEDEX_URL, dex_path)
    cache_path = path.join(POKEMON_CACHE_DIR, f"{dex_path}.html")
    return fetch_soup(url=url, cache_path=cache_path)


def fetch_variant_sprite(
    url: str,
    pokemon_name: str,
    variant_name: str | None,
) -> Sprite:
    """Fetch the Pokemon's variant's sprite.

    Args:
        url (str): The website URL.
        pokemon_name (str): The Pokemon's name
        variant_name (str): The Pokemon's variant's name

    Return:
        sprite (Sprite): The Pokemon's variant's small picture representation.
    """
    cache_path = path.join(
        SPRITE_CACHE_DIR,
        f"{safe_name(pokemon_name, variant_name)}.png"
    )
    return Sprite.fetch(url, cache_path)


def get_variant_soup(
    pokemon_soup: BeautifulSoup,
    pokemon_name: str,
    variant_name: str,
) -> BeautifulSoup:
    """Extract a variant from a Pokemon's web page.

    Args:
        pokemon_soup (BeautifulSoup): The Pokemon's data.
        pokemon_name (str): The Pokemon's name.
        variant_name (str): The Pokemon's variant's name.

    Return:
        soup (BeautifulSoup): A web element containing the Pokemon's variant's
```

```
            data.
    """
    cache_path = path.join(
        VARIANT_CACHE_DIR,
        f"{safe_name(pokemon_name, variant_name)}.html"
    )
    if path.exists(cache_path):
        variant_soup = load_soup(cache_path)
    else:
        tabs = pokemon_soup.find("div", "sv-tabs-tab-list").find_all("a")
        variant_to_id = {
            parse_str(tab): tab["href"].lstrip("#")
            for tab in tabs
        }
        closest_id = variant_to_id[variant_name]
        variant_soup = pokemon_soup.find("div", id=closest_id)
        save_soup(variant_soup, cache_path)
    return variant_soup


def all_variants() -> Iterator[Variant]:
    """Return all Pokemons' variants' data.

    Return:
        data (Iterator[Variant]): All Pokemons' variant's data.
    """
    page = fetch_soup(
        url=path.join(POKEDEX_URL, "all"),
        cache_path=path.join(BASE_CACHE_DIR, "all.html"),
    )
    rows = page.find_all("tr")

    for i, row in enumerate(rows[1:]):
        name_td = row.find("td", "cell-name")
        pokemon_name_a = name_td.find("a", "ent-name")
        pokemon_name = parse_str(pokemon_name_a)
        variant_name_small = name_td.find("small", "text-muted")
        if variant_name_small is None:
            variant_name = None
        else:
            variant_name = parse_str(variant_name_small)

        dex_path = path.basename(pokemon_name_a["href"])
        pokemon_soup = fetch_pokemon_soup(dex_path)
        variant_soup = get_variant_soup(
            pokemon_soup=pokemon_soup,
            pokemon_name=pokemon_name,
            variant_name=pokemon_name if variant_name is None else variant_name,
        )

        sprite_img = row.find("img", "img-fixed icon-pkmn")
        sprite_url: str = sprite_img["src"]
        variant_sprite = fetch_variant_sprite(
            url=sprite_url,
            pokemon_name=pokemon_name,
            variant_name=variant_name,
        )

        if variant_name is None:
            print(f"{i+1:04d}\t{pokemon_name}")
        else:
            print(f"{i+1:04d}\t{pokemon_name}: {variant_name}")

        yield Variant(
            pokemon_name=pokemon_name,
            variant_name=variant_name,
            soup=variant_soup,
            sprite=variant_sprite,
        )
```

9.15 scrape/variant.py

```
from functools import cached_property
from math import sqrt

from bs4 import BeautifulSoup
from .util.sprite import Sprite

from .util.soup import parse_float, parse_int, parse_percent, parse_str


REGIONAL_PREFIXES = ["Galarian", "Alolan"]


class Variant:
    """The attributes of a single Pokemon variant.

    Args:
        pokemon_name (str): The name of the Pokemon.
        variant_name (str | None): The name of the Pokemon's variation, or None
            if the Pokemon only has one variation.
        soup (BeautifulSoup): The web element containing all information about
            the Pokemon variant.
        sprite (Sprite): An image of the Pokemon variant.
    """

    PROPERTIES = [
        "type_number",
        "type_1",
        "type_2",
        "height_m",
        "weight_kg",
        "abilities_number",
        "total_points",
        "hp",
        "attack",
        "defense",
        "sp_attack",
        "sp_defense",
        "speed",
        "catch_rate",
        "base_friendship",
        "base_experience",
        "maximum_experience",
        "egg_type_number",
        "has_gender",
        "proportion_male",
        "egg_cycles",
        "damage_from_normal",
        "damage_from_fire",
        "damage_from_water",
        "damage_from_electric",
        "damage_from_grass",
        "damage_from_ice",
        "damage_from_fighting",
        "damage_from_poison",
        "damage_from_ground",
        "damage_from_flying",
        "damage_from_psychic",
        "damage_from_bug",
        "damage_from_rock",
        "damage_from_ghost",
        "damage_from_dragon",
        "damage_from_dark",
        "damage_from_steel",
        "damage_from_fairy",
        "sprite_size",
        "sprite_perimeter",
        "sprite_perimeter_to_size_ratio",
        "sprite_red_mean",
        "sprite_green_mean",
        "sprite_blue_mean",
        "sprite_brightness_mean",
        "sprite_red_sd",
        "sprite_green_sd",
        "sprite_blue_sd",
        "sprite_brightness_sd",
        "sprite_overflow_vertical",
        "sprite_overflow_horizontal",
    ]

    def __init__(
        self,
        pokemon_name: str,
        variant_name: str | None,
        soup: BeautifulSoup,
        sprite: Sprite
    ):
        self.pokemon_name = pokemon_name
        self.variant_name = variant_name
        self._soup = soup
        self._sprite = sprite

    @cached_property
    def full_name(self) -> str:
        """The full name of the Pokemon."""
        if self.variant_name is None:
            return self.pokemon_name
        if self.pokemon_name in self.variant_name:
            return self.variant_name
        for prefix in REGIONAL_PREFIXES:
            if self.variant_name.startswith(prefix):
                variant_name_rest = self.variant_name[len(prefix):].strip()
                return f'{prefix} {self.pokemon_name} {variant_name_rest}'
        return f'{self.pokemon_name} {self.variant_name}'

    def _get_cell(self, label: str) -> BeautifulSoup:
        return self._soup.find("th", string=label).find_next_sibling("td")

    @cached_property
    def pokedex_number(self) -> int:
        """The index number of the Pokemon in the national Pokedex."""
        return parse_int(self._get_cell("National №"))

    @cached_property
    def type_number(self) -> int:
        """The number of types the Pokemon has (1 or 2)."""
        return len(self._get_cell("Type").find_all("a"))

    @cached_property
    def type_1(self) -> str:
        """Type 1 of the Pokemon."""
        return parse_str(self._get_cell("Type").find_all("a")[0])

    @cached_property
    def type_2(self) -> str:
        """Type 2 of the Pokemon."""
        if self.type_number >= 2:
            return parse_str(self._get_cell("Type").find_all("a")[1])
        return "None"

    @cached_property
    def height_m(self) -> float:
        """The height of the Pokemon in metres."""
        return parse_float(self._get_cell("Height"))

    @cached_property
    def weight_kg(self) -> float | None:
        """The weight of the Pokemon in kilograms."""
        cell = self._get_cell("Weight")
        if parse_str(cell) == "-":
            return None
        return parse_float(cell)

    @cached_property
    def abilities_number(self) -> int:
        """The number of abilities the Pokemon has (0 to 3)."""
        return len(self._get_cell("Abilities").find_all("br"))

    @cached_property
    def total_points(self) -> int:
        """The total number of combat points the Pokemon has.

        Points include hp, attack, defence, sp_attack, sp_defence, and speed.
        """
        total = parse_int(self._get_cell("Total"))
        assert total == (
            self.hp
            + self.attack
            + self.defense
            + self.sp_attack
            + self.sp_defense
            + self.speed
        )
        return total

    @cached_property
    def hp(self) -> int:
        """The number of hit points the Pokemon has."""
        return parse_int(self._get_cell("HP"))

    @cached_property
    def attack(self) -> int:
        """The attack value of the Pokemon."""
        return parse_int(self._get_cell("Attack"))

    @cached_property
    def defense(self) -> int:
        """The defense value of the Pokemon."""
        return parse_int(self._get_cell("Defense"))

    @cached_property
    def sp_attack(self) -> int:
        """The special attack value of the Pokemon."""
        return parse_int(self._get_cell("Sp. Atk"))

    @cached_property
    def sp_defense(self) -> int:
        """The special defense value of the Pokemon."""
        return parse_int(self._get_cell("Sp. Def"))

    @cached_property
    def speed(self) -> int:
        """The speed value of the Pokemon."""
        return parse_int(self._get_cell("Speed"))
    @cached_property
    def catch_rate(self) -> int | None:
        """The catch rate of the Pokemon."""
        cell = self._get_cell("Catch rate")
        if parse_str(cell) == "-":
            return None
        return parse_int(self._get_cell("Catch rate"))

    @cached_property
    def base_friendship(self) -> int:
        """The base friendship value of the Pokemon."""
        cell = self._soup \
            .find("a", href="/glossary#def-friendship") \
            .find_parent("th") \
            .find_next_sibling("td")
        return parse_int(cell)

    @cached_property
    def base_experience(self) -> int:
        """The base experience value of the Pokemon."""
        return parse_int(self._get_cell("Base Exp."))

    @cached_property
    def _growth_rate(self) -> str:
        """The growth rate type of the Pokemon.

        Either Erratic, Fast, Medium Fast, Medium Slow, Slow, or Fluctuating.
        """
        return parse_str(self._get_cell("Growth Rate"))

    _MAX_EXP = {
        "Erratic": 600_000,
        "Fast": 800_000,
        "Medium Fast": 1_000_000,
        "Medium Slow": 1_059_860,
        "Slow": 1_250_000,
        "Fluctuating": 1_640_000,
    }

    @cached_property
    def maximum_experience(self) -> int:
        """The experience required by the Pokemon to achieve maximum level."""
        return self._MAX_EXP[self._growth_rate]

    @cached_property
    def egg_type_number(self) -> int:
        """The number of egg types of the Pokemon."""
        return len(self._get_cell("Egg Groups").find_all("a"))

    @cached_property
    def has_gender(self) -> bool:
        """Whether the Pokemon has a gender."""
        cell = self._get_cell("Gender")
        s = parse_str(cell)
        return s not in ("Genderless", "-")

    @cached_property
    def proportion_male(self) -> float:
        """The proportion of the Pokemon that are male (percentage, 0-100)."""
        cell = self._get_cell("Gender")
        s = parse_str(cell)
        return parse_percent(cell) / 100. if self.has_gender else 0.5

    @cached_property
    def egg_cycles(self) -> int:
        """The number of step cycles required for the Pokemon's egg to hatch."""
        return parse_int(self._get_cell("Egg cycles"))

    def _get_damage_from(self, r: int, c: int) -> float:
        """The multiplier applied to damage of X type against the Pokemon.

        Return the damage received coefficient of the Pokemon,
        when fighting against an enemy Pokemon of a specific type.

        Args:
            r (int): The row of type X in the PokemonDB damage chart.
            c (int): The column of type X in the PokemonDB damage chart.

        Return:
            damage_from (float): The multiplier on damage of type X.
        """
        section = self._soup \
            .find("h2", string="Type defenses") \
            .find_parent("div")
        cell = section \
            .find_all("table")[r] \
            .find_all("tr")[-1] \
            .find_all("td")[c]
        s = parse_str(cell)
        return {
            "4": 4.,
            "3": 3.,
            "2": 2.,
            u"1\u00BD": 1.5,
            "1.25": 1.25,
            "": 1.,
            u"\u00BD": 1./2.,
            u"\u00BC": 1./4.,
            u"\u215B": 1./8.,
            "0": 0.,
        }[s]

    @cached_property
    def damage_from_normal(self) -> int:
        return self._get_damage_from(r=0, c=0)
    @cached_property
    def damage_from_fire(self) -> int:
        return self._get_damage_from(r=0, c=1)

    @cached_property
    def damage_from_water(self) -> int:
        return self._get_damage_from(r=0, c=2)

    @cached_property
    def damage_from_electric(self) -> int:
        return self._get_damage_from(r=0, c=3)

    @cached_property
    def damage_from_grass(self) -> int:
        return self._get_damage_from(r=0, c=4)

    @cached_property
    def damage_from_ice(self) -> int:
        return self._get_damage_from(r=0, c=5)

    @cached_property
    def damage_from_fighting(self) -> int:
        return self._get_damage_from(r=0, c=6)

    @cached_property
    def damage_from_poison(self) -> int:
        return self._get_damage_from(r=0, c=7)

    @cached_property
    def damage_from_ground(self) -> int:
        return self._get_damage_from(r=0, c=8)

    @cached_property
    def damage_from_flying(self) -> int:
        return self._get_damage_from(r=1, c=0)

    @cached_property
    def damage_from_psychic(self) -> int:
        return self._get_damage_from(r=1, c=1)

    @cached_property
    def damage_from_bug(self) -> int:
        return self._get_damage_from(r=1, c=2)

    @cached_property
    def damage_from_rock(self) -> int:
        return self._get_damage_from(r=1, c=3)

    @cached_property
    def damage_from_ghost(self) -> int:
        return self._get_damage_from(r=1, c=4)

    @cached_property
    def damage_from_dragon(self) -> int:
        return self._get_damage_from(r=1, c=5)

    @cached_property
    def damage_from_dark(self) -> int:
        return self._get_damage_from(r=1, c=6)

    @cached_property
    def damage_from_steel(self) -> int:
        return self._get_damage_from(r=1, c=7)

    @cached_property
    def damage_from_fairy(self) -> int:
        return self._get_damage_from(r=1, c=8)

    @cached_property
    def sprite_size(self) -> float:
        """The size of the Pokemon's sprite.

        The size is defined as the proportion of non-transparent pixels in the
        sprite.
        """
        return (self._sprite.alpha != 0).sum()

    @cached_property
    def sprite_perimeter(self) -> float:
        """The perimeter of the Pokemon's sprite.

        The number of pixels in the sprite's perimeter.
        """
        return (self._sprite.perimeter).sum()

    @cached_property
    def sprite_perimeter_to_size_ratio(self) -> float:
        """The ratio of the Pokemon's sprite's perimeter to its size.

        Roughly increases with the sprite's "spikiness".
        """
        return self.sprite_perimeter / self.sprite_size

    @cached_property
    def sprite_red_mean(self) -> float:
        """The mean red color value of the Pokemon's sprite."""
        return (self._sprite.alpha * self._sprite.red).sum() / self.sprite_size

    @cached_property
    def sprite_green_mean(self) -> float:
        """The mean green color value of the Pokemon's sprite."""
        return (self._sprite.alpha * self._sprite.green).sum() / self.sprite_size

    @cached_property
    def sprite_blue_mean(self) -> float:
        """The mean blue color value of the Pokemon's sprite."""
```

```python
        return (self._sprite.alpha * self._sprite.blue).sum() / self.sprite_size

    @cached_property
    def sprite_brightness_mean(self) -> float:
        """The mean red brightness value of the Pokemon's sprite."""
        return (self._sprite.alpha * self._sprite.brightness).sum() / self.sprite_size

    @cached_property
    def sprite_red_sd(self) -> float:
        """The standard deviation in red color value of the Pokemon's sprite."""
        return sqrt(
            (self._sprite.alpha * (self._sprite.red - self.sprite_red_mean)**2).sum()
            / self.sprite_size
        )

    @cached_property
    def sprite_green_sd(self) -> float:
        """The standard deviation in green color value of the Pokemon's sprite.
        """
        return sqrt(
            (self._sprite.alpha * (self._sprite.green - self.sprite_green_mean)**2).sum()
            / self.sprite_size
        )

    @cached_property
    def sprite_blue_sd(self) -> float:
        """The standard deviation in blue color value of the Pokemon's sprite.
        """
        return sqrt(
            (self._sprite.alpha * (self._sprite.blue - self.sprite_blue_mean)**2).sum()
            / self.sprite_size
        )

    @cached_property
    def sprite_brightness_sd(self) -> float:
        """The standard deviation in brightness of the Pokemon's sprite."""
        return sqrt(
            (self._sprite.alpha * (self._sprite.brightness -
             self.sprite_brightness_mean)**2).sum()
            / self.sprite_size
        )

    @cached_property
    def sprite_overflow_vertical(self) -> float:
        """The amount of the Pokemon touching the top/bottom edges of its
        sprite.
        """
        return self._sprite.alpha[[0, -1], :].mean()

    @cached_property
    def sprite_overflow_horizontal(self) -> float:
        """The amount of the Pokemon touching the left/right edges of its
        sprite.
        """
        return self._sprite.alpha[:, [0, -1]].mean()

    def as_dict(self) -> dict[str, int | str | None]:
        """The attributes of the variant as a dictionary."""
        return {
            attr: getattr(self, attr)
            for attr in Variant.PROPERTIES
        }
```

9.16 tune/__init__.py

```
from .dimension import Dimension, Categorical, Constant, Integer, Real
from .tuner import Tuner, SearchSpace, Splitter
from .sk_bayes import SKBayesTuner
from .outer_cv import outer_cv
```

9.17 tune/dimension.py

```
from typing import Any

from skopt.space import Dimension, Categorical, Integer, Real

class Constant(Categorical):
    """A search space dimension that has only one value.

    Args:
        value: The single value of the search space dimension.
    """
    def __init__(self, value: Any):
        super().__init__(categories=[value])
        self.value = value
```

9.18 tune/outer_cv.py

```
import os
from time import time
from typing import Any, TypeVar

import numpy as np
import pandas as pd
from sklearn.model_selection import KFold, StratifiedKFold

from model import Model
from util import load_data, multiset_accuracy, plot_confusion
from .tuner import Tuner, SearchSpace


M = TypeVar("M", bound=Model)

BASE_MODEL_DIR = "../models/"
```

```python
def outer_cv(
    tuner: Tuner[M],
    search: SearchSpace,
    name: str,
    duplicate: bool,
    hard_mode: bool,
    n_folds_outer: int = 5,
    n_folds_inner: int = 5,
) -> float:
    """Find the nested cross validation accuracy of a model tuning process.

    Args:
        tuner (Tuner[M]): A tuner that produces a model given a set of training
            data via cross-validation.
        search (SearchSpace): The search space for model hyperparameters used by
            the tuner. The final model produced by the tuner is chosen from
            within this space.
        name (str): The name of the model.
        duplicate (bool): Whether to apply the duplication technique to predict
            both type_1 and type_2 together.
        hard_mode (bool): Whether to remove the damage_from and type_2 features
            from the predictors.
        n_folds_outer (int): The number of outer cross-validation folds.
        n_folds_inner (int): The number of inner cross-validation folds.

    Returns:
        accuracy (float): The average out-of-sample prediction accuracy across
            all outer folds.
    """
    X, y = load_data(hard_mode=hard_mode, duplicate=duplicate)
    results: list[dict[str, Any]] = []

    if duplicate:
        if hard_mode:
            name = f"{name} (Duplication, Hard)"
        else:
            name = f"{name} (Duplication)"
    else:
        if hard_mode:
            name = f"{name} (Hard)"
        else:
            name = f"{name} (Base)"

    model_dir = os.path.join(
        BASE_MODEL_DIR,
        name.replace(" ", "_") \
            .replace("(", "").replace(")", "") \
            .replace(",", "") \
            .lower(),
    )
    if not os.path.exists(model_dir):
        os.makedirs(model_dir)
    outer_splitter = (KFold if duplicate else StratifiedKFold)(
        n_splits=n_folds_outer,  # number of folds
        shuffle=True,  # protects against data being ordered, e.g., all successes first
        random_state=441,
    )

    if duplicate:
        y_merged = np.array(["/".join(sorted(row)) for row in y])
        outer_splits = list(outer_splitter.split(X, y_merged))
    else:
        outer_splits = list(outer_splitter.split(X, y))

    predictions = y.copy()
    for i, (train_ids, test_ids) in enumerate(outer_splits):
        id = i+1
        model_path = os.path.join(model_dir, f"cv-{id}.mdl")

        if os.path.exists(model_path):
            print(f"{id}  Cached result loaded from {model_path}")
            model: M = Model.load(model_path)
        else:
            X_train: np.ndarray = X[train_ids, :]
            y_train: np.ndarray = y[train_ids]

            split = (KFold if duplicate else StratifiedKFold)(
                n_splits=n_folds_inner,
                shuffle=True,
                random_state=441,
            )

            start_s = time()
            model = tuner.tune(
                X_train=X_train,
                y_train=y_train,
                search=search,
                split=split,
            )
            end_s = time()
            elapsed_s = end_s - start_s
            print(f"{id}  Tuned model in {elapsed_s} seconds")

            model.save(model_path)
            print(f"{id}  Saved to cache at {model_path}")

        print(f"{id}  Best configuration: {model.config}")

        X_test: np.ndarray = X[test_ids, :]
        y_test: np.ndarray = y[test_ids]
        y_pred = model.predict(X_test)
        accuracy = multiset_accuracy(y_test, y_pred)
        print(f"{id}  Accuracy: {accuracy}")

        results.append(dict(**model.config, accuracy=accuracy))
        predictions[test_ids] = y_pred

    labels = model.labels

    plot_confusion(
        y_true=y,
        y_pred=predictions,
        labels=labels,
        normalize="true",
        name=f"{name} by true label",
        path=os.path.join(model_dir, "confusion_matrix_by_true.png"),
    )

    plot_confusion(
        y_true=y,
        y_pred=predictions,
        labels=labels,
        normalize="pred",
        name=f"{name} by predicted label",
        path=os.path.join(model_dir, "confusion_matrix_by_pred.png"),
    )

    results_df = pd.DataFrame.from_records(results)
    results_df.to_csv(os.path.join(model_dir, "result.csv"))
    return np.average(results_df['accuracy'])
```

9.19 tune/sk_bayes.py

```
from typing import Any, Generic

import numpy as np
from sklearn.base import BaseEstimator
from skopt import BayesSearchCV

from model import SKClassifier, SKModel
from util import PARALLELISM
from .tuner import Tuner, SearchSpace, Splitter


class SKBayesTuner(Tuner[SKModel], Generic[SKClassifier]):
    """Optimizes an sklearn classifier through Bayesian optimization.

    Args:
        Classifier: An sklearn classifier model class (e.g. LogisticRegression).
            Must support the fit, predict, and predict_proba methods.
        duplicate (bool): Whether to wrap the classifier as a multiset
            classifier via the duplication method.
        num_labels (int): If duplicate=True, the number of labels that the
            multiset classifier should predict.
    """
    def __init__(self, estimator: SKClassifier):
        self._estimator = estimator
    def tune(
        self,
        X_train: np.ndarray,
        y_train: np.ndarray,
        search: SearchSpace,
        split: Splitter,
    ) -> SKModel:
        """Tune hyperparameters of a model using cross-validation.

        Performs Bayesian optimization to efficiently sample the search space.

        Args:
            X_train (ndarray): X-values of the training data.
            y_train (ndarray): y-values of the training data.
            search (SearchSpace): The search space for the hyperparameters.
            split (Splitter): A cross validation generator.

        Returns:
            model: A fitted model with the optimal hyperparameters.
        """
        opt = BayesSearchCV(
            estimator=self._estimator,
            search_spaces=search,
            cv=split,
            n_jobs=PARALLELISM,
        )
        opt.fit(X_train, y_train)

        model: BaseEstimator = opt.best_estimator_  # type: ignore
        best_config: dict[str, Any] = opt.best_params_  # type: ignore
        return SKModel(model, config=best_config)
```

9.20 tune/tuner.py

```
from abc import ABC, abstractmethod
from typing import Any, Generic, TypeAlias, TypeVar

import numpy as np
from sklearn.model_selection import BaseCrossValidator
from skopt.space import Dimension

from model import Model

SearchSpaceDict: TypeAlias = dict[str, Dimension]
SearchSpaceList: TypeAlias = list[SearchSpaceDict]
SearchSpaceWeightedList: TypeAlias = list[tuple[SearchSpaceDict, int]]
SearchSpace: TypeAlias = SearchSpaceDict | SearchSpaceList | SearchSpaceWeightedList

Splitter: TypeAlias = BaseCrossValidator

M = TypeVar("M", bound=Model)
```

```python
class Tuner(Generic[M], ABC):
    """
    A thin wrapper around a function that selects the best model given some
    cross validation splits.
    """

    @abstractmethod
    def tune(
        self,
        X_train: np.ndarray,
        y_train: np.ndarray,
        search: SearchSpace,
        split: Splitter,
    ) -> M:
        """Tune hyperparameters of a model using cross-validation.

        Args:
            X_train (ndarray): X-values of the training data.
            y_train (ndarray): y-values of the training data.
            search (SearchSpace): The search space for the hyperparameters.
            split (Splitter): A cross validation generator.

        Returns:
            model: A fitted model with the optimal hyperparameters.
        """
        pass
```

9.21 util/__init__.py

```python
from .accuracy import multiset_accuracy
from .confusion import confusion, plot_confusion
from .data import load_data
from .duplication import duplicate, predict_multiset_indices
from .parallel import PARALLELISM
```

9.22 util/accuracy.py

```python
from multiset import Multiset
import numpy as np
from sklearn.metrics import accuracy_score

def multiset_accuracy(y_true: np.ndarray, y_pred: np.ndarray) -> float:
    """Evaluate the accuracy of predictions versus test data.

    For multilabel predictions, the labels are treated as multisets, and the
    accuracy is measured as the multiset intersection between the true and
    predicted labels.

    Args:
        y_true (ndarray): [N] or [N x M] true class labels.
        y_pred (ndarray): [N] or [N x M] predicted class labels.

    Returns:
        accuracy (float): Accuracy (in the range [0,1]) of the predictions.
    """
    assert y_true.shape == y_pred.shape

    if len(y_true.shape) == 1:
        return float(accuracy_score(y_true=y_true, y_pred=y_pred))

    else:
        N, M = y_true.shape
        # the easiest way to do this is actually just a loop
        correct = np.array([
            len(Multiset(y_true[i,:]) & Multiset(y_pred[i,:])) / M
            for i in range(N)
        ])
        return correct.mean()
```

9.23 util/confusion.py

```
from typing import Literal

import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

def multiset_confusion(
    y_true: np.ndarray,
    y_pred: np.ndarray,
    labels: list[str],
    normalize: Literal["true", "pred", "all"] | None
) -> np.ndarray:
    """Generate a confusion matrix for M-label multiset classification.

    In single-label prediction, the (raw) confusion matrix can be interpreted as
    a sum of outer products between one-hot vectors encoding the true and
    predicted classes.

    Here we treat the multi-label equivalent of the confusion matrix as also a
    sum of outer products, but the encodings are no longer one-hot.

    Args:
        y_true (ndarray): [N x M] true class labels.
        y_pred (ndarray): [N x M] predicted class labels.
        labels (list[str]): [K] All possible labels.
        normalize ("true" | "pred" | "all" | None): Normalizes the confusion
            matrix over the true class (rows), predicted class (columns), or
            the population. If None, the confusion matrix will not be
            normalized. "Normalizing" in this context means scaling such that
            the resulting sum is M (not 1). The reason for preferring M is so
            that the normalized entries may be interpreted as mean cardinality.

    Returns:
        cm (ndarray): [K x K] matrix where cm[i,j] is the count/proportion
```

```
            associated with true class i and predicted class j.
    """
    assert y_true.shape == y_pred.shape
    N, M = y_true.shape
    K = len(labels)

    label_to_index = {label: i for i, label in enumerate(labels)}

    cm = np.zeros((K, K), dtype=float)
    for i in range(N):
        ind_true = [label_to_index[label] for label in y_true[i,:]]
        ind_pred = [label_to_index[label] for label in y_pred[i,:]]
        for i in ind_true:
            for j in ind_pred:
                cm[i, j] += 1.0

    # copied right out of sklearn's confusion matrix code
    if normalize == "true":
        cm = M * cm / cm.sum(axis=1, keepdims=True)
    elif normalize == "pred":
        cm = M * cm / cm.sum(axis=0, keepdims=True)
    elif normalize == "all":
        cm = M * cm / cm.sum()
    cm = np.nan_to_num(cm)

    return cm


def confusion(
    y_true: np.ndarray,
    y_pred: np.ndarray,
    labels: list[str],
    normalize: Literal["true", "pred", "all"] | None
) -> np.ndarray:
    """Generate a confusion matrix for M-label multiset classification.

    For details, see sklearn's documentation of confusion_matrix, and
    multiset_confusion above.

    Args:
        y_true (ndarray): [N x M] true class labels.
        y_pred (ndarray): [N x M] predicted class labels.
        labels (list[str]): [K] All possible labels.
        normalize ("true" | "pred" | "all" | None): Normalizes the confusion
            matrix over the true class (rows), predicted class (columns), or
            the population. If None, the confusion matrix will not be
            normalized.

    Returns:
        cm (ndarray): [K x K] matrix where cm[i,j] is the count/proportion
            associated with true class i and predicted class j.
    """
    if len(y_true.shape) == 1:
```

```python
        return confusion_matrix(
            y_true=y_true,
            y_pred=y_pred,
            labels=labels,
            normalize=normalize,
        )
    else:
        assert len(y_true.shape) == 2
        return multiset_confusion(
            y_true=y_true,
            y_pred=y_pred,
            labels=labels,
            normalize=normalize,
        )


def plot_confusion(
    y_true: np.ndarray,
    y_pred: np.ndarray,
    labels: list[str],
    normalize: Literal["true", "pred", "all"] | None,
    name: str,
    path: str,
) -> None:
    """Generate, plot, display, and save a confusion matrix.

    Args:
        y_true (ndarray): [N x M] true class labels.
        y_pred (ndarray): [N x M] predicted class labels.
        labels (list[str]): [K] All possible labels.
        normalize ("true" | "pred" | "all" | None): Normalizes the confusion
            matrix over the true class (rows), predicted class (columns), or
            the population. If None, the confusion matrix will not be
            normalized.
        name (str): The name of the model that the confusion matrix is about.
        path (str): The path to a local file to save an image of the plot.

    Returns:
        cm (ndarray): [K x K] matrix where cm[i,j] is the count/proportion
            associated with true class i and predicted class j.
    """
    cm = confusion(
        y_true=y_true,
        y_pred=y_pred,
        labels=labels,
        normalize=normalize,
    )
    display = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=labels)
    plt.rcParams['figure.figsize'] = [10, 10]
    plt.figure(figsize=(10,10))
    display.plot(xticks_rotation=45, values_format=".0%")
    plt.title(f"Confusion Matrix for {name}")
    plt.savefig(path)
    plt.show()
    return cm
```

9.24 util/duplication.py

```
import numpy as np


def duplicate(X: np.ndarray, y: np.ndarray):
    """Duplicates the given data.

    Specifically, observation i is duplicated M times, one for each of its
    labels, ending up at indices [i*M, (i+1)*M).

    Args:
        X (ndarray): [N x P] predictor values.
        y (ndarray): [N x M] labels.

    Returns:
        X (ndarray): [N*M x P] predictor values.
        y (ndarray): [N*M] labels.
    """
    N, P = X.shape
    assert y.shape[0] == N
    _, M = y.shape

    X = X.repeat(repeats=M, axis=0)
    assert X.shape == (N * M, P)

    y = y.flatten()
    assert y.shape == (N * M,)

    return X, y


def predict_multiset_indices(p: np.ndarray, cardinality: int):
    """Convert a list of probabilities to indices of a multiset prediction.

    Repeatedly predicts the highest label and subtracts 1 from its probability.

    Args:
        p (ndarray): [N x K] "probabilities", where p[i,k] is the expected
            multiplicity of label k in observation i.
        cardinality (int): The cardinality M of the predicted multiset (sum of
            element multiplicities).

    Returns:
        indices (ndarray): [N x M] label indices, where indices[i] contains the
            M labels (counting possible duplicates) predicted for observation i.
            Labels are ordered according to the algorithm (repeatedly predicting
            one label and subtracting from its probability).
    """
    N, K = p.shape
    all_ids = []
    for _ in range(cardinality):
        ids = p.argmax(axis=-1)
        for i, k in enumerate(ids):
            p[i,k] -= 1
        all_ids.append(ids)

    all_ids = np.stack(all_ids, axis=-1)
    assert all_ids.shape == (N, cardinality)

    return all_ids
```

9.25 util/parallel.py

```
import multiprocessing
import os

def _parallelism() -> int:
    """The level of parallism that this computer can handle.

    Returns:
        parallelism (int): The number of parallel workers that this computer can
            handle.
    """
    try:
        # NOTE: only available on some Unix platforms
        # https://stackoverflow.com/q/74048135
        cores = len(os.sched_getaffinity(0))
    except AttributeError:
        cores = multiprocessing.cpu_count()

    return max(cores - 1, 1)

PARALLELISM = _parallelism()
```