Artificial Neural Network to Predict the Edibility of Mushrooms

Griffin Salyer

Objective

Hobbyists, mushroom enthusiasts, and even chefs around the world work for years before understanding which mushrooms are from which species and class – and what attributes can best identify a mushroom. One common issue among those individuals who also eat the good mushrooms, is having enough experience to understand and identify which mushrooms, that can be found in a common forest, are edible! In this project, I seek to find an answer quickly and easily with a simple neural network structure – and identify those attributes that are most important in determining poisonous versus edible mushrooms.

The Primary objective for this project is to use a simple feedforward ANN classifier to predict whether a given mushroom, based on its physical features, is poisonous or edible. A secondary objective is to detail which features provide the most explanatory power in determining whether a mushroom is poisonous or edible – as this would provide some tremendous value in understanding what evidence would suggest poisonous mushrooms in the wild. In addition, while selecting the features that are most important, the number of features needed to provide an accurate model result may be decreased, and therefore, the model complexity can be decreased. Finally, a tertiary objective is to create a streamlined architecture that can generalize to other similar – but different – data and models.

Final Model

Overview

```
def final_model():
    # create model
    model = Sequential()
    model.add(Dense(8, input_dim = x.shape[1],  activation = 'relu'))
    model.add(Dense(8, activation = 'relu'))
    model.add(Dense(2, activation = 'softmax'))
    # Compile model
    model.compile(loss = 'categorical_crossentropy', optimizer = 'adam' , metrics = ['accuracy'])
    print(model.summary())

    return model
```

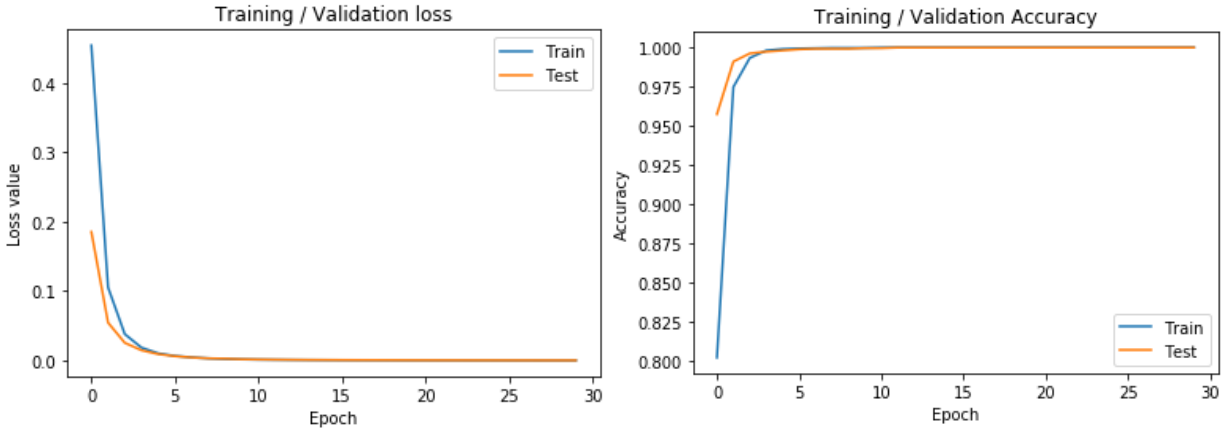
The final model consists of 3 layers, 1 input, 1 hidden, 1 output with 8 nodes, 8 nodes, and 2 output nodes respectively. The input and hidden layer activation is ReLu, while the output layer is multiclass activation softmax. The loss is categorical crossentropy, and the metric is accuracy. The optimizer is adam. Something to note is that this model did not actually need a hidden layer to achieve 100% accuracy, however for consistency and best practice, I decided to include a hidden layer as it did not make the model all that much more complicated and, for this dataset, the execution time was negligible.

Train | Test | Validate

```
def final_model_compile(model_def):
    # create final model with train/test split
    final = KerasClassifier(build_fn = model_def, epochs = 30, batch_size = 50, verbose = 1)
    finalfit = final.fit(x_train,y_train, validation_data = (x_test, y_test))
    #finalpredict = final.predict(x_test)

    plot_eval(finalfit)

    print('The mean accuracy is : %.2f%% ' % round(np.mean(finalfit.history['accuracy'])*100.00,2))
    print('The final loss is:' , np.mean(finalfit.history['loss']))

    return finalfit
```

The model was trained over 30 epochs on a train test split with 66% train, 33% test split. There were 30 epochs, and a batch size of 50. This model performed well, and its output was excellent.

Results




```
The mean accuracy is : 99.22%
The final loss is: 0.021710703327304966
Time required for training: 0:00:19.920806
```

```
Layer (type)                 Output Shape              Param #
=================================================================
dense_258 (Dense)            (None, 8)                 944
_________________________________________________________________
dense_259 (Dense)            (None, 8)                 72
_________________________________________________________________
dense_260 (Dense)            (None, 2)                 18
=================================================================
Total params: 1,034
Trainable params: 1,034
Non-trainable params: 0
_________________________________________________________________
```

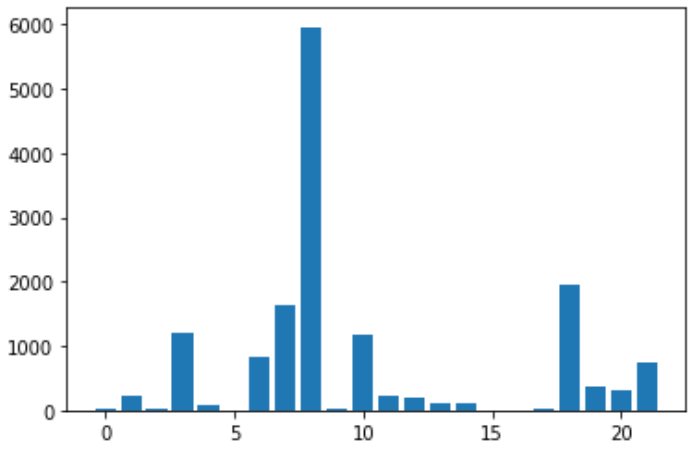
The results were excellent with a reasonably parsimonious model, and an accuracy that crossed between 99% and 100%. The final loss was .022 at 30 epochs – which is great. The time required was only 19 seconds.

Supplemental Models

Feature Selection

A feature selection supplemental model can be beneficial as a first step in the full model architecture because of the context it provides the modeler in terms of data and its facets. In this case, this supplemental model was somewhat difficult to implement because of the difficulty in categorical data. Specifically, the feature selection model essentially became a variable importance model, this model used a statistical test to map the importance of each feature and provided a somewhat different answer from the variable importance model outlined below. Overall, this model was somewhat ambiguous because it did not properly define the optimal number of features explicitly. Another approach to this may have been to use the one-hot encoded feature structure, but this would come as a loss of some interpretability. I wanted to keep the classes of features intact, rather than discarding some of the features in a class because of their relative unimportance – where unbalanced features would certainly be discarded immediately.

Additionally, any way to reduce the complexity of the model without significant changes in accuracy or loss can be particularly impactful in model performance for all problems. Discarding some features would allow the NN to focus on only those mappings that are most important, and therefore, deliver a more succinct conclusion and classification. We always want more parsimonious models, especially when we dive into datasets that have exponentially more instances than this dataset.

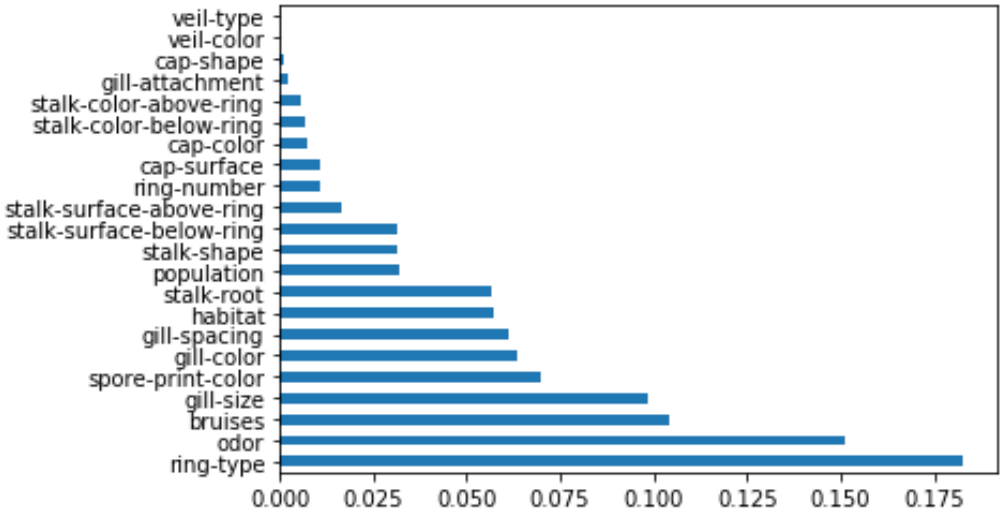


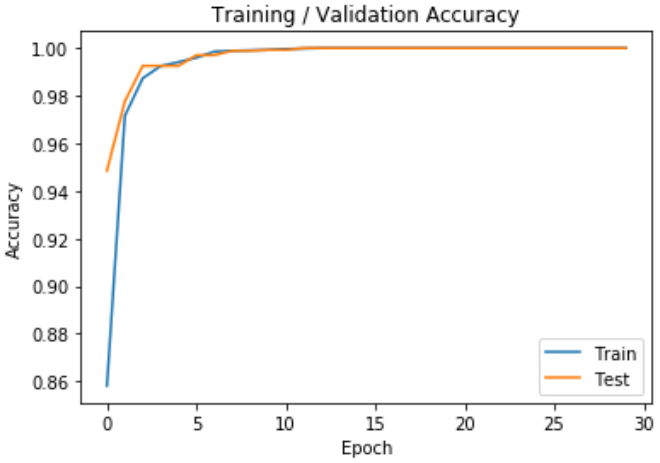

```
Feature 0,  cap-shape: 17.508364
Feature 1,  cap-surface: 214.068544
Feature 2,  cap-color: 11.511382
Feature 3,  bruises: 1194.277352
Feature 4,  odor: 75.910163
Feature 5,  gill-attachment: 3.505447
Feature 6,  gill-spacing: 826.795274
Feature 7,  gill-size: 1636.606833
Feature 8,  gill-color: 5957.764469
Feature 9,  stalk-shape: 36.594105
Feature 10,  stalk-root: 1186.029221
Feature 11,  stalk-surface-above-ring: 222.982400
Feature 12,  stalk-surface-below-ring: 206.648180
Feature 13,  stalk-color-above-ring: 119.792216
Feature 14,  stalk-color-below-ring: 109.789410
Feature 15,  veil-type: nan
Feature 16,  veil-color: 5.126826
Feature 17,  ring-number: 25.646335
Feature 18,  ring-type: 1950.610146
Feature 19,  spore-print-color: 379.132729
Feature 20,  population: 311.766736
Feature 21,  habitat: 751.309489
```

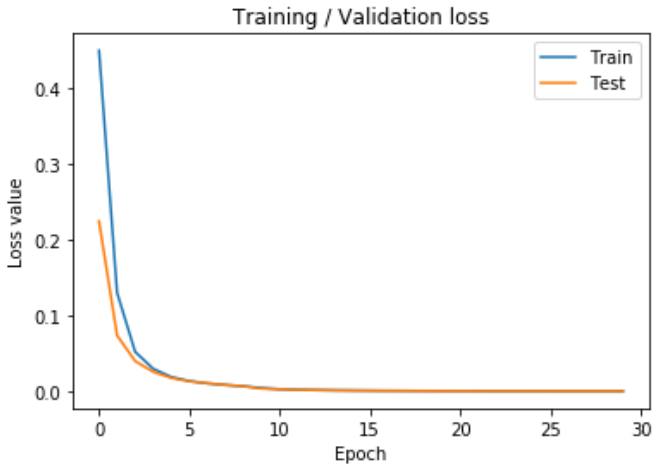
Variable Importance

A variable importance model was beneficial in the architecture of this project because its relevance in understanding the problem scope and why the neural network may classify in one way or another. By uncovering the importance of the features, insight can also be derived about the individual nature of some feature and its relationship with edibility in mushrooms. Below is where the variable importance model is implemented. The ten highest ranking variables were selected, and the results were essentially identical. The benefit comes from making the model much simpler, as we can see from the number of parameters that come from the same network structure as the other models.

```
def feature_select_model(x = x_fsd):
    # create model
    model = Sequential()
    model.add(Dense(8, input_dim = x.shape[1],  activation = 'relu'))
    model.add(Dense(8, activation = 'relu'))
    model.add(Dense(2, activation = 'softmax'))
    # Compile model
    model.compile(loss = 'categorical_crossentropy', optimizer = 'adam' , metrics = ['accuracy'])
    print(model.summary())

    return model
```








```
Layer (type)                 Output Shape              Param #
=================================================================
dense_276 (Dense)            (None, 8)                 480
_________________________________________________________________
dense_277 (Dense)            (None, 8)                 72
_________________________________________________________________
dense_278 (Dense)            (None, 2)                 18
=================================================================
Total params: 570
Trainable params: 570
Non-trainable params: 0
_________________________________________________________________
```

Experimental Plan and Process

Overview

```
def base_model():
    # create model
    model = Sequential()
    model.add(Dense(8, input_dim = x.shape[1],  activation = 'relu'))
    model.add(Dense(8, activation = 'relu'))
    model.add(Dense(2, activation = 'softmax'))
    # Compile model
    model.compile(loss = 'categorical_crossentropy', optimizer = 'adam' , metrics = ['accuracy'])
    print(model.summary())

    return model
```

The experimental process for this architecture was relatively straightforward. No grid search or hyperparameter optimization was needed at the scale of these variables, so there was no tuning in this many employed. Most of the experimental process came from playing around with the number of layers, and ultimately feature selection. The most important experimental process was implementing a baseline model that showed immediately how well the model already trained – it did not require much additional tuning. It is important to note, however, that tuning was only unnecessary in this case because the scale of the data is relatively small. If the dataset had millions of datapoints, the process would be much more thorough and would consider tuning every parameter available.

Train | Test | Validate

```
def experimental_compile(model_def):
    #create model for experimenting
    model = KerasClassifier(build_fn = model_def, epochs = 30, batch_size = 50, verbose = 2)
    kfold = KFold(n_splits = 5, shuffle = True)
    results = cross_val_score(model, x, y, cv = kfold)
    print_eval = print("Baseline: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()*100))

    return print_eval
```

The model is nearly identical to the final model, however in order to really test all the data, a kfold CV was used. This ensures that I catch everything when experimenting with the data and the model. 5 folds were used, and the model performed well.

Results

```
Baseline: 100.00% (0.00%)
Time required for training: 0:01:34.990430
```

The results were 100% accuracy overall between training and validation. This is excellent, and bar any code that causes issues, this means the model is mapping extremely well and classifying perfectly. Even with 5 fold the model runs quickly in a minute and 30 seconds.

Data

The data consists of 8124 instances across 232 rows of the physical features of mushroom – detailed thoroughly below. The response variable is one column consisting of ‘p’ or ‘e’ – whether a mushroom is poisonous or edible. All the samples of mushrooms are hypothetical, but mirror real attributes and mushrooms that can be found in the wild. There are several preprocessing steps required in order for the data to be understood by the neural network, and some steps to be taken to better understand what the data is saying itself. Finally, the data processing steps were optimized in a function to be fed a file and some basic parameters for easy procedure.

Preprocessing Data

All of the data required some restructuring and adjustment that, although not difficult, increases the complexity of the model structure nearly 5-fold. The first step that was required was to label encode the response variable. This label encoded response variable was then split into two columns with one-hot encoding. This step is especially important because not only does this change the output shape from a single output to two outputs, but it requires a multiclass classification procedure instead of a simple binary classification. I decided this was the best option because it allows for more mappings to occur given our data and allows the NN to home in more quickly and succinctly on the correct associations between a mushroom given its features and edibility.

The features also required a label encode and one-hot encode to be fed into the network properly. At the end of this processing the features consisted of 116 columns encoded with 1 or 0 within the category code it was given. Again, this provides the network with more mappings to consider, and better accuracy overall than if each feature had one column and categorical or binary codes – the model could learn the associations in a proliferated manner rather than a truncated feature indication process.

```
def load_and_treat(file):
    mushroom = pd.read_csv(file)
```

```
#make an array of the values
mushroom_a = np.array(mushroom)
#mushroom = mushroom.values #also works

#distinguish our features from response
x = mushroom_a[:, 1:]
y = mushroom_a[:, 0]

#binary encode the response variable
label_encoder = LabelEncoder()
y_a = label_encoder.fit_transform(y)

#now convert integers to a one-hot encode using keras integer encoder to_categorical
y = to_categorical(y_a)

#give back encoded values as letters for y
#inverted_y =label_encoder.inverse_transform([argmax(y[0])])
#print(inverted_y)

#make x features into one-hot encoded features - no need for labelencoder here
ohe = OneHotEncoder(sparse = False) #sparse means output an array not a sparse matrix
x = ohe.fit_transform(x)

#give back encoded values as letters for x
#inverted_x = ohe.inverse_transform(x)
#print(inverted_x)
```

Preprocessing for Final Model

The processing for the final model was the same as described above but then applied into a train/test split function in the sk-learn package pipeline – this allowed for the final model to use succinct validation steps instead of a full CV method or KFold method (which was utilized with the experimental model).

```
#create train test split for use in final model
x_train, x_test, y_train, y_test = train_test_split(x,y, test_size = .33, random_state = 1)
```

Preprocessing for Supplemental Modeling

The supplemental models, like feature selection and importance, required that the data be in a different shape than the neural network. As such pandas was used to shape the data into categorical data as a part of a pandas dataframe. The first step taken was to change the columns into the type 'category' and then apply a category encoder through pandas. This was a little easier than using Keras or sk-learn imports because there were 23 rows, and it could be done in essentially one step rather than a few. Then the categorical data was split into a response dataframe and a feature dataframe that consisted of columns with numbers 1 – 5(depending on column category counts, which differed). This maintained the structure of 22 feature columns and one response column that was easily fed into feature selection and variable importance models.

```
#create a preprocessing function for use in the feature selection models - uses only specified rows
def feature_select_data(mushroom):
    x_fsd = mushroom[['ring-type','odor','bruises','gill-size','spore-print-color','gill-color','gill-spacing','habitat','stalk-root','population']]
    ohe = OneHotEncoder(sparse = False)
    x_fsd = ohe.fit_transform(x_fsd)
    xfsd_train, xfsd_test, yfsd_train, yfsd_test = train_test_split(x_fsd,y, test_size = .33, random_state = 1)


    return x_fsd, xfsd_train, xfsd_test, yfsd_train, yfsd_test
```

Metrics

The metrics used to evaluate all NN models are Categorical Crossentropy as the loss and Accuracy. These will provide the models with appropriate improvement generation, and accuracy reporting to determine the efficacy of each model in both the experimental process and the final process.

Validation Method

The validation method for the experimental process and model will be Kfold Cross-Validation and Cross Validation score measurement. This ensures that the experimental model hops through all the appropriate hoops – including the extra kfold hoop. Although KFold or CV is unnecessary in this model, it is good practice and included for high quality experimental reporting.

The Validation method for the final model is a simple Train/Test(validation) split with a split 66% Train and 33% Test as recommended in standard practice procedure guidelines (Keras, sk-learn).

Results and Conclusions

Overall, the models all performed well from the start. This dataset was easy to work with after preprocessing and its size was such that we would not have to worry about underfitting, but could still process all the datapoints immediately without cross validation. The models achieved between 99% and 100% accuracy in correctly classifying whether a given set of features for a mushroom would predict edibility. Suffice to say, these models were straightforward to implement, and maintain parsimony throughout the process.

Discussion of Methods

Selection of optimum number of units

The optimum number of units was mostly ambiguous insofar as some constraints were met. Those constraints were that the model layer and node size did not facilitate overfitting, and that the model unit size facilitated a rapid minimization of loss and maximization of accuracy. My method was simple because this network did not need any complex layers or unit selection optimization. As long as a reasonable number of parameters were trained, and the parameters did not exceed the instances of data, then this model was well off. Additionally, because the model approached a 100% accuracy so quickly, there was generally no real need for an optimization in this parameter.

Type of network

The network is a straightforward feedforward network with simple parameters and relatively small layer sizes. The scope of this project was easily sustained with this type of ANN and did not require a recurrent, convolutional, or backpropagated network. The dataset is fairly simple, and the model required was fairly simple.

Type of training

The training of this network was also very straightforward and did not necessarily need any optimization procedures like grid-search or cross validation. Although I did use Kfold and CV, it was unnecessary other than a precaution for possible unbalanced data. For the final model the training was done using a train/validation method and a final prediction after training.

Proportion of train test split

As mentioned previously, the train test split was 66% to 33% respectively. I justify this in that it is standard procedure and good practice with a reasonable proportion of the data being trained with, and a validation of that training with a reasonable set of the data proportionately.

Number of input/output

The number of inputs became 116 after encoding the features. This provided ample room for the network to appropriately map and process. The final output was a multiclass output with 2 columns – one column indicating if the mushroom was edible, the other indicating if the mushroom was poisonous.

Number/size of layers

As previously stated, the number and size of layers was kept simple and consistent, as the model did not need a complex set of layers or nodes. I am justified in this decision by the consistency in the model's accuracy of 99 – 100%.

Number of epochs

The number of epochs finally settled on about 30 for caution. The model trained by about 8 epochs, however, and did not require any beyond that, but for the sake of completeness and best practice, the model ended up with 30 epochs. This minimized the loss incredibly and provided a consistent 100% accuracy every time.

Activations/optimization function choice

The activation was another mostly ambiguous choice, as a part of standard procedure. The activation function used for the input and hidden layers was ReLu, while the output layer was a multiclass function softmax. These choices came from best practices and because of the model performance, did not need to be optimized. The optimization function was also a best practice optimizer – adam – and it performs well without the requirement for adjustments.

Size of dataset

The size of the dataset was 8124 instances or datapoints – which provides ample data to not produce concern for over or underfitting in the modeling process.

Learning rate

There was no need for a learning rate optimization in this modeling procedure. Again, this is justified by the inherently good accuracy and loss minimization by a generally baseline model.

Momentum

Momentum was also not a consideration with this model for simplicity sake and because it was not required.

Improvements and Future Work

More Data/ Real Data

One way to improve this project would be to use real data rather than hypothetical (but accurate) data. This would allow for an interesting application in a real-world environment that may help real people. In addition to this real data – data at scale would be an interesting problem to tackle, especially as the scope of the model for this problem is rather simple. How do apply this model structure to several million datapoints, and how then can we optimize the parameters for run times and parsimonious solutions?

Convolutional Data

Something that I would love to tackle is a Convolutional Neural Network for not only identifying types and classes/species of mushrooms, but then also whether they are edible, but just from images. This would be a fascinating process to implement, especially since its not run of the mill animal or vehicles. This could also develop into a real application in the field for hikers, enthusiasts and chefs alike, as this model might be able to be made into a phone app and used out in the real world – Just take a picture of the mushroom to see if you can cook it with your camping meal!

Supplemental Model Additions

In making a more thorough project architecture, it is evident that having supplemental information about the models and data can increase the value and insight provided by the project. I think that adding in models that compute feature selection and importance differently the than the current implementations in this project may be interesting and serve as a comparative guideline for the right features to select. Additionally, proliferating different types of machine learning models that may increase the confidence in the Neural Network model may be pertinent to providing a more concise and complete project reporting structure.

Comparative Analysis

A comparative analysis between additional machine learning models added to the architecture could be insightful and interesting. Testing how other models perform, and why they perform that way, may be an interesting research topic as well, and provide valuable content to a complete project. If I had the time and capacity, I would model logistic regressions, tree methods, and other subset selection methods with this data.

EDA

An interesting provisional step in detailing the data and its facets would have been to include Exploratory Data Analysis in the form of visualizing the data and diving into its intricacies. This step of any architecture can add tremendous value, and aid in developing a rigorously and valuably structured analysis. A thorough analysis should always include some EDA and provide an audience with a thorough explanation of what the data is and how the data looks.

Implementing Class Object Architectures

A final improvement to this project architecture would be creating class object to perform a series of functions, like compiling all the different models. The benefits of this architecture are that everything becomes defined as a function, and all the inputs are initialized up front, and therefore become changeable with relative ease. This architecture allows for more of a pipeline and self-organizes so that the code is easy to read, and easy to implement.

Reference and Research

https://machinelearningmastery.com/multi-class-classification-tutorial-keras-deep-learning-library/

https://www.kaggle.com/gulsahdemiryurek/mushroom-classification-with-ann

data:

https://www.kaggle.com/uciml/mushroom-classification