Plant Leaf Classification with Artificial Neural Networks using a Probabilistic Framework of Shape, Texture, and Margin Features

Jennylyn Sy

October 13, 2016

Abstract

K nearest neighbor has been the known classifier for leaf species especially for this dataset. This paper will use a tuned K nearest neighbor as the benchmark and use different neural network architectures to improve the multi-class logarithmic loss in classifying the 99 different species of leaves.

I. Introduction

i. Overview

Given how plant life is an essential part of our ecology, classification for these species is very important as this could have impacts to 1. population tracking and preservation, 2. medicinal research, and 3. crop and food supply management. But with almost half a million of plant species all over the globe, classification does not come as an easy task, especially with dozens of possible features a leaf can have like: eccentricity, elongation, solidity, lobedness and smoothness, among others.

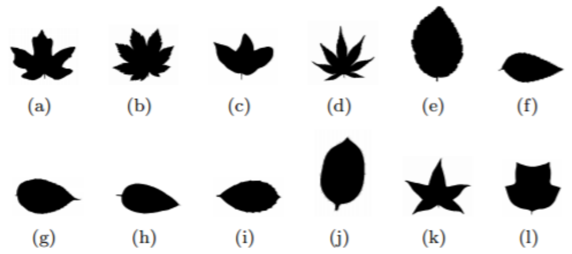


Figure 1: A small variety of leaf species part of the classification task

For this project, I will be classifying leaves based on 3 distinct features each represented by a 64-element vector:

1. Shape – a Centroid Contour Distance Curve shape signature
2. Texture – an Interior Texture Feature Histogram
3. Margin – a Fine-Scale Margin Feature Histogram

Binary images of the data in .jpg format are also included as part of the dataset. These files are from Kaggle and are publicly available via the link below:

https://www.kaggle.com/c/leaf-classification

ii. Problem Statement

For this problem, I'm planning to use artificial neural networks to predict the leaf classification. Below is an outline of how I will tackle the problem:

1. Preprocess the Data
    a. Label Encoding the Output
    b. Standardizing the Features
    c. Apply Dimensionality Reduction
2. Separate 20% for Holdout Validation
3. Compute the Benchmark with KNN
4. Run Artificial Neural Network Trials
5. Do Hyper Parameter Optimization
6. Evaluate the Results

iii. Metric

The goal of this problem is to correctly classify each leaf sample to the right specie. There is a total of 99 species/classes with a train data of 990 leaf samples. With the multi-class nature of this problem, the metric of logarithmic loss would be a more informative indicator for this high class, low sample dataset.

The logarithmic loss can handle multi-classification and will expect an input of the probability for each class instead of just the most likely class. With the huge number of classes, having a measurement that looks into the probabilities for all the 99 classes would be more useful than the precision and recall.

$$logloss = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij}\log(p_{ij})$$

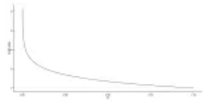


Figure 2: Logarithmic Loss Equation and a Sample Graph

The plot above shows the logloss of a positive instance where the predicted probability goes from 0 to1. As it goes closer to 1, the logloss is nearing 0 meaning the lower the logloss, the better the classifier. Logloss basically quantifies the accuracy of a classifier by penalizing misclassifications (when the classifier is both wrong and confident) heavily. Logloss measures how uncertain (cross entropy) the classifier is compared to the true labels.

II. Analysis

i. Data Exploration

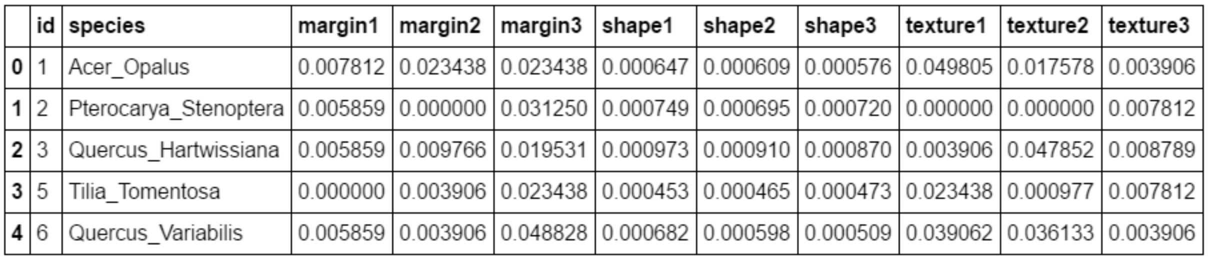
The train data has a total of 990 rows and 194 columns and has no missing values. The 194 columns are composed of:

Column Name	Variable Type	Data Type
Leaf ID	Unique Identification	Int64

Species	Response Variable	Object
Margin (64-element vector)	Feature Variables	Float64
Texture (64-element vector)	Feature Variables	Float64
Shape (64-element vector)	Feature Variables	Float64

Table 1: Data and Variable Types

The 990-row dataset is a total of 99 unique leaf species with 10 samples each. Some of these species include: Quercus_Infectoria_sub, Alnus_Sieboldiana, Acer_Palmatum. The table below shows the top 5 rows of the train set with the first 3 elements of the 3 major features:



	id	species	margin1	margin2	margin3	shape1	shape2	shape3	texture1	texture2	texture3
0	1	Acer_Opalus	0.007812	0.023438	0.023438	0.000647	0.000609	0.000576	0.049805	0.017578	0.003906
1	2	Pterocarya_Stenoptera	0.005859	0.000000	0.031250	0.000749	0.000695	0.000720	0.000000	0.000000	0.007812
2	3	Quercus_Hartwissiana	0.005859	0.009766	0.019531	0.000973	0.000910	0.000870	0.003906	0.047852	0.008789
3	5	Tilia_Tomentosa	0.000000	0.003906	0.023438	0.000453	0.000465	0.000473	0.023438	0.000977	0.007812
4	6	Quercus_Variabilis	0.005859	0.003906	0.048828	0.000682	0.000598	0.000509	0.039062	0.036133	0.003906

Table 2: Top 5 Rows of the Train Dataset

Since the response variable is categorical and is currently in text, these will need to be label encoded so these responses can be fed to the classifiers without any issues. With the 99 classes, the labels would range from 0 to 98.

The test data has a total of 594 rows and 193 columns. The columns in test are exactly the same as in train less the response variable – species. The 594 rows represent 6 samples for each of the 99 species identified in train.

ii. Exploratory Visualization

Feature 1 – Margin

The margin feature refers to the leaf margin or the edges of a leaf. This is of two basic types : 1. Entire - having a smooth edge with no teeth or lobes and 2. Toothed - having a saw-like margin. There are also several variations of toothed as depicted in Figure 3.

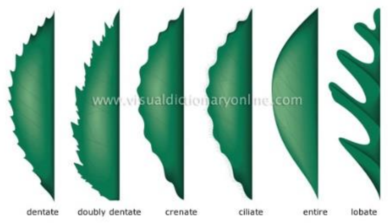


Figure 3: Different Types of Leaf Margins

The leaf margin feature is a quantitative description where evenly spaced points comprising the entire outline of a leaf are gathered and are used to compute magnitude, gradient, and curvature. These feature vectors are then quantized to 64 predetermined vectors where the histogram is built from.

The maximum value for this feature in the train dataset is 0.389 and the minimum value is 0. Taking random samples from the dataset shows the following:

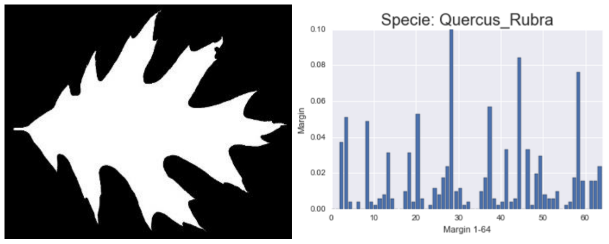


Figure 4: Leaf ID 11 Quercus Rubra and a Graph of its Leaf Margin Feature

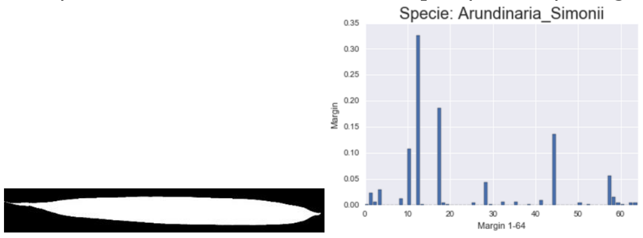


Figure 5: Leaf ID 37 Arundinaria Simonii and a Graph of its Leaf Margin Feature

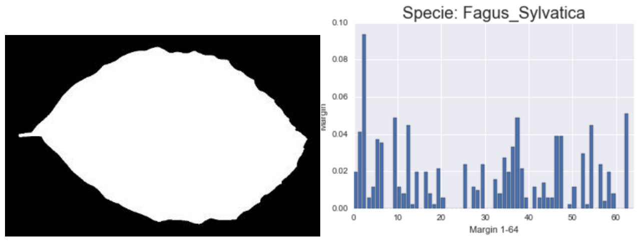


Figure 6: Leaf ID 45 Fagus Sylvatica and a Graph of its Leaf Margin Feature

The first sample shows several peaks in the graph representing the leaf's pointed edges. The second sample, on the other hand, shows a simpler elongated design with a very extreme peak of 0.33. And the last one shows a relatively more balanced measurement with its round shape.

Feature 2 – Shape

The 64-element vector for shape is the Centroid Contour Distance Curve measured for every leaf sample. A set of perimeter points are identified on the contour of leaf that could range from 100 to 3000 elements depending on the leaf's size and complexity. Through an ordering function, a chain of connectivity is created with these pixels in a contiguous fashion. To ensure that every leaf has the same number of elements, the vector is scaled down to 64 elements through linear interpolation.

The maximum value for shape is 0.003 while the minimum value is 2.17*^-5. Below are the bar graphs for the 64-element vector of shape:

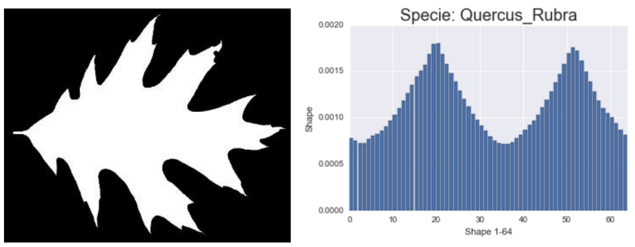


Figure 7: Leaf ID 11 Quercus Rubra and a Graph of its Shape Feature

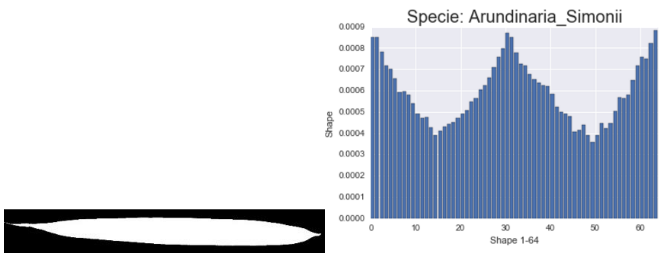


Figure 8: Leaf ID 37 Arundinaria Simonii and a Graph of its Shape Feature

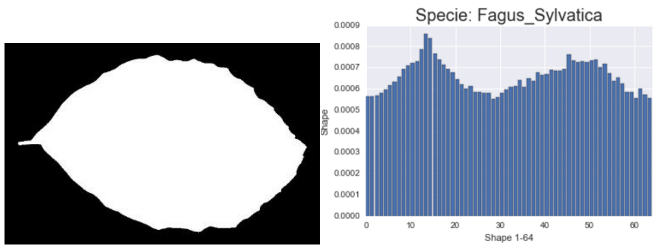


Figure 9: Leaf ID 45 Fagus Sylvatica and a Graph of its Shape Feature

There are very obvious patterns that can be seen for the attribute shape for the leaves. Taking for example, Quercus Rubra (first sample), below are two bar graphs for the shape of the same specie for ID's 7 and 209.

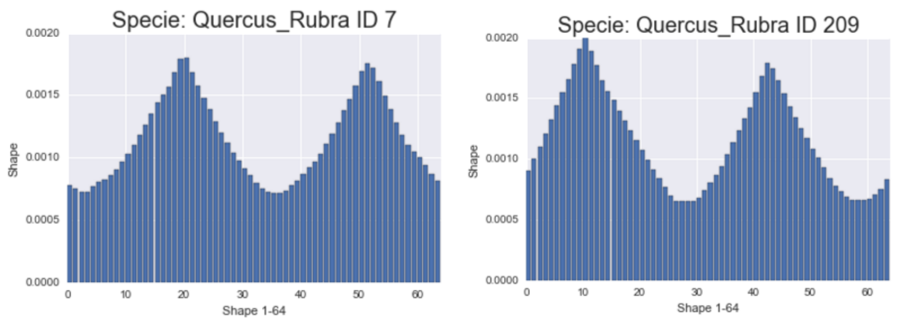


Figure 10: Leaf ID's 7 and 209 Quercus Rubra and Graphs of the Shape Features

Though the two samples above are the same species, there seems to be a misalignment on how the peaks are formed with the left one forming its first peak at around shape20 and the second one at shape10. This is one of the limitation for the shape feature as it does not seem to be rotation invariant. Nevertheless, the shape feature still provides valuable information that can aid in classifying different samples of leaves.

Feature 3 – Texture

Texture from a leaf is mostly coming from its venation with other sources including the hair and the glands. The texture feature of this dataset was obtained using image processing on areas of the leaf that only includes the vein fabric. The 64-element, texture feature histogram is a result of joint distributions from the images, using different scales of the Gabor filter.

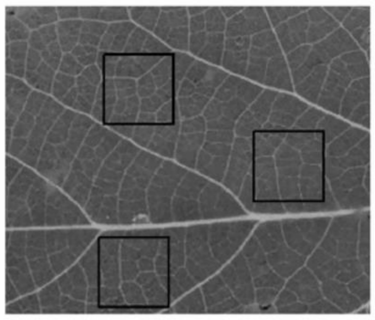


Figure 11: Sample Location Spots where Images for Leaf Texture was Captured

The maximum value for this feature is 0.854 and the minimum value is 0.0. The following graph shows the mean for each of the 64 elements of the texture feature. Overall mean for this (64*990) dataset is 0.0156.

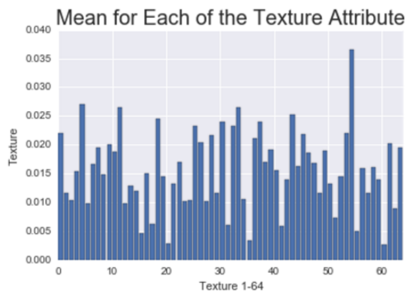


Figure 12: Bar Graph for Texture Mean

The following graph shows the count of 0.0 values for each vector. There are a handful of vectors that mostly seem to have 0.0's in their field. On average, 24/64 or 38% of the texture attribute is 0.0 for every leaf sample.

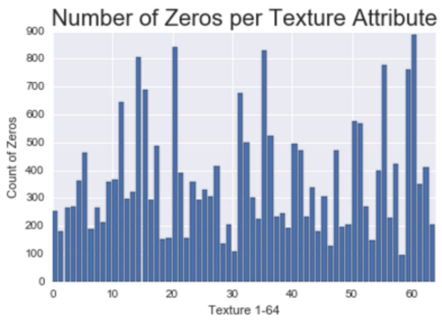


Figure 13: Count of Zeros for per Vector for Texture

Outliers and Scaling

Since each of the 64 elements for all the three features is part of a quantitative chain representation of the feature itself, there will be no removal of outliers in that sense. Also, given that the train dataset has a wide set of classes with low samples (10), none of the original data will be removed when building the model.

Since the classifiers that I will be using for model-building are not tree-based, scaling the data to 0 mean and unit standard deviation is necessary to ensure that all features contribute equally and that none of the weights will update much faster than the others especially where gradient descent is used as the optimizer.

iii. Algorithms and Techniques

Two very glaring characteristics of this dataset are the wide class size and the small sample set. With this in mind, I plan to use three algorithms appropriate for this kind of dataset.

1. Principal Component Analysis - For Dimensionality Reduction
2. K Nearest Neighbors – For Benchmarking
3. Neural Networks – For Model Building

Principal Component Analysis

PCA will be done as part of data preprocessing. For the KNN classifier for example, PCA will be useful as it will reduce the feature dimensions, making the points in the feature space less complex per sample than if you had a lot more features. As for Neural Network, having less of the features (but at the same time retain most of the information) will make the network converge faster. The parameter here would be the number of components which would refer to the new count of the features transformed capturing as much variability of the data possible.

When PCA is applied to the dataset (without the response variable), PCA "finds" a new coordinate system for the dimensions that captures the most variation and transforms the data accordingly with the least amount of information lost as per the number of principal components specified.

K Nearest Neighbors

KNN is a simple algorithm that basically uses the distance between k closest points to compute the probability that the test sample will belong to a particular class. With KNN clustering, patterns emerge and the classification reasoning is easily understandable. Number of neighbors (k) and weights will be tuned using gridsearch while the rest of the paramters will be the default set.

This algorithm will plot each train data into a feature space. The data to predict is then plotted in this feature space and the k nearest neighbors are identified. If the weights parameter is set to 'uniform' then a majority vote is used to identify which class the test data belongs to but if this is set to distance then the closer train points will have a more significant impact in determining the test point's class.

Neural Network

There are three basic steps by which a neural network works: 1. Takes input data 2. Transforms input data by calculating a weighted sum over the inputs and 3. Uses a non-linear function for this transformation to produce an intermediate state. This constitutes a single layer of the neural network. The intermediate state is then fed to the next layer as the input data. When these steps are repeated, the neural network learns from multiple non-linear intermediate states to finally create a prediction. The learning is coming from an error signal that it uses to see the difference between predicted value and actual value from where it then changes the weights to make the predictions more accurate. This error adjustment of the parameters is called backpropagation of errors. Backpropagation is a method for finding the gradient of the error with respect to the weights of the network. The gradient is how the error of the network changes with changes to the network's weights. This is used to perform gradient descent to find a set of weights to minimize the network error.

A simple three-layer diagram of a neural network is shown below with x1 to x3 as the input or the features from the dataset and x0 as the bias unit. These are initialized with some weight as set in one of the parameters (initialization). These weighted inputs are then passed to the activation function towards the hidden layer where it passes a unit step function to produce the predicted output of y.

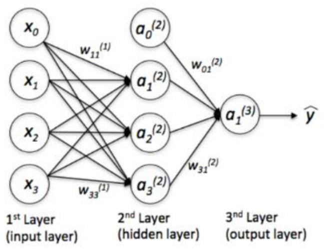


Figure 14: Diagram of a Neural Network

For this project, a sequential model or a linear stack of layers will be used to build the network with the following parameters:

i. Neurons - the constitutive unit in a neural network that receives one or more input, sums it up to produce an output that will then be passed to an activation function
ii. Initialization – refers to how the weights will be assigned in the first layer of the model, the most common ones include a normal or a uniform distribution of weights.
iii. Activation Function – this aims to produce a linear/non-linear (depending on the function chosen) decision boundary via combinations of the weighted inputs essentially transforming input signals into an output signal.

An example for this would be the Rectified Linear Unit (relu) which is the simplest non-linear function. It is 0 for negative inputs while positive values remain untouched (f(x) = max(0,x)). With a gradient/derivative of 1 for positive values and 0 for negative values, this means that during backpropagation negative gradients will not be used to update weights of the outgoing rectified linear unit since these are now 0. With the large gradients of positive values used, this ensures a faster training speed.

Softmax is a way to turn the scores/logits produced by the neural network to probabilities. This is usually used in the last layer of a classification problem trained under a log loss.

Other most common activation functions are the following:

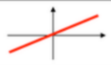
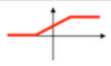
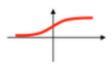
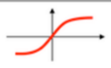
Activation function	Equation	Example	1D Graph
Unit step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Linear	$\phi(z) = z$	Adaline, linear regression	
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multi-layer NN	
Hyperbolic tangent	$\phi(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$	Multi-layer NN	

Figure 15: Some Common Activation Functions in Neural Networks

iv. Dropout – adding a dropout is one way to control overfitting through regularization in the neural network. From the term itself, it randomly drops out units by cutting connections before proceeding to the next layer to ensure that the network generalizes better.
v. Layers – the layer is considered to be the highest level of building block in the network. This serves as a container to accept weighted inputs, transforms these inputs and passes them on to the next layer in mostly non-linear functions. The first layer is called the input layer and the last layer is the output layer. Having these two layers would be a very basic network. Everything in between these two layers are called hidden layers.
vi. Optimizer – used to minimize an objective function through iteration. The popular ones include stochastic gradient descent and adaptive gradient descent. One of these optimization algorithms is called RMSprop or Root Mean Square Propagation. It is an adaptive learning rate method that keeps track of the weighted running mean of the squared gradient then divides each calculated gradient by the square root of the weighted running mean. So when there is a plateau in the error surface and the gradient is small, the update takes greater steps and ensures faster learning. But when there is a large gradient or exploding gradient, the update size becomes smaller.
vii. Epoch – an epoch is a single pass across all the data in the training set (both forward and backward pass). So specifying an epoch of 30 for example would mean that the model will pass through the entire training data 30 times updating the weights for each time.
viii. Batch-Size – is the number of training samples in one pass.
ix. Loss – this is the objective function or the error rate to minimize. As discussed in the previous section, log loss will be used and the keras equivalent of a multiclass log loss is the categorical cross entropy

For this dataset, a basic neural network will initially be built with 2 layers. Then a 3-layer network will be applied on the dataset. When the optimum number of layers is identified, parameters will be tuned to ensure the most appropriate parameters are used to give the best possible log loss.

iv. Benchmark

To gauge how good a model scored, a benchmark is needed as a basis for comparison. It is basically saying how good your model is compared to just randomly guessing the result for a very basic benchmark.

Given the metric of logarithmic loss, the most basic benchmark for this dataset would simply be an equal assignment of probability for each of the 99 classes. If each class would be given the probability of (1.0/99) or 0.0101%, the log loss for this uniform probability benchmark is 4.595 using the entire training set to predict the entire test set. This is part of the initial dataset provided by Kaggle with the file titled 'sample_submission'.

Aside from this, I will also use a tuned KNN as the benchmark for the neural network. The process and logarithmic loss will be computed in the next section.

III. Methodology

i. Data Preprocessing

3 Steps:

1. Label Encode the response variable – this will change the 'specie' column to a list of numbers from 0-98 with every number uniquely representing a specie.

2. Stratify Split the data with 80% as training data or 792 samples and 20% as the validation set or 198 samples
3. Standardize the data – using sklearn's Standard Scaler, transform each of the total 192 features to 0 mean unit variance.
4. Run PCA on the transformed data, excluding the response variable, capturing at least 99% of the data's variability. Without initially specifying the number of components, the plot below shows the cumulative sum of the variability captured as the principal component increases. With a principal component of 130, 99.63% of the data's information is captured.

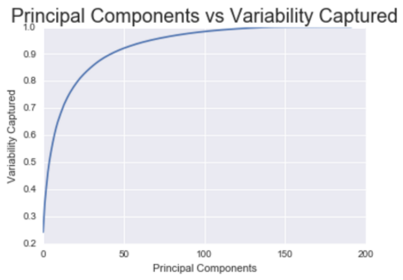


Figure 16: Cumulative Variability Captured as Count of Prinicipal Component Increases

ii. Implementation

KNN

Using an exhaustive grid search on weights and n_neighbors with the following code:

```
parameters = {'weights':('distance','uniform'),'n_neighbors': [2,3,4,5,6,7,8,9]}
knn = KNeighborsClassifier()
logloss = make_scorer(log_loss,greater_is_better=False, needs_proba=True)
grid_obj = GridSearchCV(estimator=knn, param_grid = parameters, scoring = logloss)
```

The optimal parameters are n_neighbors = 4 and weights = 'distance'. This returned a logarithmic loss of 0.05806 on the validation set, which is so much better than the uniform probability benchmark score indicated earlier.

Neural Network

For neural networks, I plan to start the first trial with a simple 2-layer network then continue the experiment depending on the results of each trial run. To ensure that all throughout the test, the same random number sequence is used, I initiate the seed using np.random.seed(0). Since I have 99 classes using a loss function of categorical cross entropy, I need to one-hot encode these classes as this is how multiclass y should be fed into keras.

The keras library is very easy to use given its modularity – which means you add what you want your model to have in terms of layers, activations, dropouts, etc.

```
model = Sequential()
model.add(Dense(1024,input_dim=130,   init='uniform', activation='relu'))
model.add(Dense(99, activation='softmax'))
model.compile(loss='categorical_crossentropy',optimizer='rmsprop', metrics = ["accuracy"])
```

I created the sequential model with model = Sequential() as seen in the first line of the code. I then added a first dense layer to the Sequential model. This dense layer is simply a regular fully connected neural network layer. In this layer, I gave 1024 neurons, an input dimension of 130 (which is equal to the number of features the dataset has), a weight initialization of uniform and an activation function of relu.

I then added the second or the last layer of this network with model.add(). This time, I only gave it 99 units to represent each of the 99 classes and an activation function of softmax to predict a probabilistic distribution of the outcomes.

Now that I have all the necessary layers for the first trial, I compile the model by indicating the loss function which is categorical cross entropy, the optimizer which is rmsprop and a metrics that is by default should be set to accuracy.

```
history = model.fit(X_train, y_trainkeras, batch_size = 128, nb_epoch=30,verbose=1)
```

After the model has been compiled, I can now fit this model to the dataset using model.fit(). The input required for this are the train data with all the features, the one-hot encoded y response variable, the batch size, the number of epochs and verbose to show me the progress of the training.

```
Epoch 1/30
792/792 [==============================] - 156s - loss: 3.6811 - acc: 0.3725
Epoch 2/30
792/792 [==============================] - 155s - loss: 1.7789 - acc: 0.9419
Epoch 3/30
792/792 [==============================] - 155s - loss: 0.9460 - acc: 0.9861
Epoch 4/30
792/792 [==============================] - 153s - loss: 0.5161 - acc: 0.9937
```

The training was done in batches of 128, as indicated, and each epoch run would result to a logloss error rate and the accuracy rate. Once the training was done for 30 epochs, a graph can be plotted to show how the error decreases per epoch run.

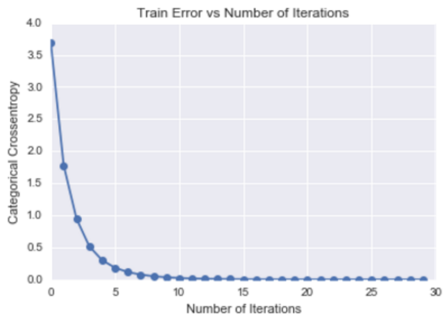


Figure 17: Train Error vs Number of Iterations for NN Trial 1

80% of the train dataset was used to train the model. The remaining 20% will be used to predict the model's log loss. After executing below snippet, the log loss for this model is 0.00693.

```
trial1 = model.predict_proba(X_test)
trial1score = log_loss(y_test, trial1)
```

Though some of keras methods were similar to sklearn like predict_proba and fit, keras being a completely new library for me also has its learning curve. Using their website and the sample codes available helped a lot in reducing complications during the coding process.

iii. Refinement

For the next model, I will add an additional layer to see if this will improve the previous validation score. Now that the neural network has become a bit more complex, there is a higher chance of overfitting. This can be solved by adding regularization techniques like Drop Out in between the layers. The code snipped showing the different parameters for the second trial is as follows:

```
model2 = Sequential()
model2.add(Dense(1024,input_dim=130,  init='uniform', activation='relu'))
model2.add(Dropout(0.2))
model2.add(Dense(512, activation='sigmoid'))
model2.add(Dropout(0.3))
model2.add(Dense(99, activation='softmax'))
model2.compile(loss='categorical_crossentropy',optimizer='rmsprop', metrics = ["accuracy"])
```

Here, an additional dense layer is added with 512 neurons and an activation function of sigmoid. Dropouts of 0.2 and 0.3 have also been added in between layers. The rest of the parameters remain similar to the first trial run. After compilation, fitting and prediction, this model resulted to a logarithmic loss of 0.017167. As per the following error graph, this model took longer to converge which is expected given that there are now more layers than the first network.

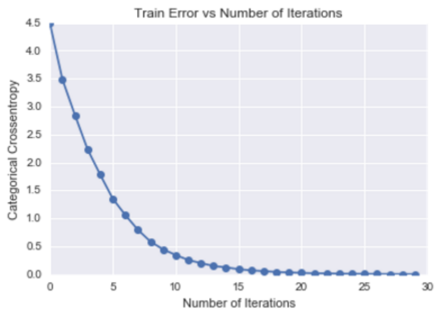


Figure 18: Train Error vs Number of Iterations for NN Trial 2

After the previous findings, a 2-layer network can already produce good results for this particular dataset. To further check if there are rooms for improvement, other parameters are tuned to see if there will be any difference.

First Parameter – Optimizer (rmsprop or adam)

The exact same model was ran with exactly the same architecture except for the change in optimizer to from rmsprop to adam. After training for 30 epochs, this new model produced a log loss of 0.05545 which is not an improvement from the first model of 0.00693.

Second Parameter – Activation Function (relu or sigmoid)

The original activation function used was relu for the first layer. The activation function for the output layer will remain the same since the results still need to be in probabilities. The change though from relu to sigmoid resulted in an increase of log loss to 0.11054. From this, we will stick with relu as the optimization function.

Third Parameter – Initialization (uniform or normal)

All of the previous architectures were using uniform initialization. For this run, an initialization of normal is set to see if this will improve the log loss or not. After 30 epochs, the validation score for normal is 0.012481 which is still not an improvement from the first run.

IV. Results

i. Model Evaluation and Validation

Model evaluation was done using a holdout of 20% separated during the preprocessing stage. It assumes the role of a test data, that's why even during Standardization and PCA feature transformation, the holdout set was not included in the fit function.

After every trial run, the model predicted the classifications of 198 unseen samples. The log loss for these predictions were then recorded and were used as the basis for tuning the parameters, as can be seen in the previous section. On top of that, csv outputs were made to submit to the Kaggle website to get the log loss of the 594-sample test data. The results are in the table below with 'Standard' being the first model ran and having the following parameters: initialization=uniform, activation function = relu, and optimizer = rmsprop.

Trial	Neural Network	20% Validation	Overall Test	Difference	Rank
Trial 1	2 Layers Standard	0.006933	0.03549	0.03	1/2
Trial 2	3 Layers with Dropouts	0.017168	0.05064	0.03	3
Trial 3	2 Layers Std but Adam Optimizer	0.055459	0.09434	0.04	4
Trial 4	2 Layers Std but Sigmoid Act Func	0.110544	0.15392	0.04	5
Trial 5	2 Layers Std but Normal Init	0.012481	0.03188	0.02	1/2

Table 5: Scores for the models for Validation Data and Final Test Data as per Kaggle submissions

It is very noticeable that the score in the validation data is always better than the overall test data by at least 0.02. Besides the fact that the validation set and the overall test have very different sample size, this table shows that the trained model is sensitive to the small fluctuations in the training set causing it to overfit in the test set and thus returning a higher than expected log loss. This can be due to the small 8-sample-per-class train dataset that was used to train the model which apparently is not enough to generalize on a wide base of test set.

Because of this, the model chosen from the validation set is not the best model to predict for the overall test.

ii. Justification

The following table shows the benchmark scores and the chosen final model score predicted on the overall test set:

Model	Scores
KNN	0.13690
Neural Network	0.03549

Table 4: Validation Scores for the Benchmark and the Final Model

For this particular dataset, KNN was deemed to be a good estimator to classify the different leaf samples as per previous scientific papers written on this regard. KNN is definitely appropriate for this dataset. Though neural networks are known to be good classifiers for huge datasets, especially if it has a lot of data to train on, it nevertheless reported good results – even better than KNN, for this particular dataset. Neural network is also expected to perform better when more train data becomes available even if it's just an addition of, for example 10 more samples per class. If more train data do get added, it is very possible that a new optimal network architecture with probably 3 layers could produce better results than the current one. So overall, the current model is ok for this dataset but additional training data may require a more complex architecture than the 2-layer model.

V. Conclusion

i. Free-Form Visualization

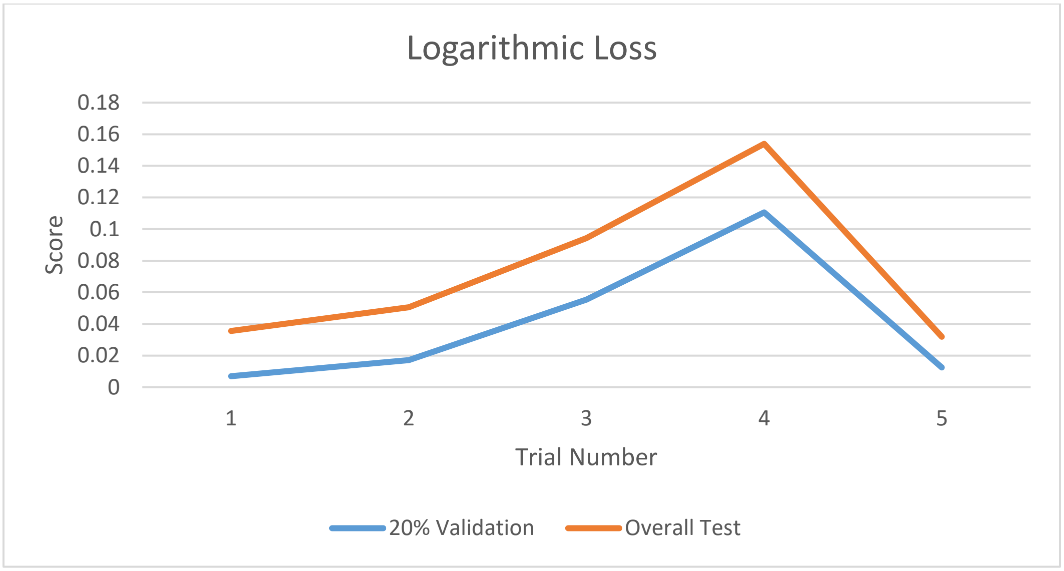


Figure 19: The Logarithmic Loss Score for every Trial

Though both scores show direct proportionality, the logarithmic loss difference shows that the current model will not be able to generalize well and is overfitting the validation data. A more robust validation method should have been in place like cross-validation to ensure a logarithmic loss score that can very well be close to the logarithmic loss score when the model is applied to new unseen data.

ii. Reflection

The dataset was known to be approached using the K Nearest Neighbors algorithm and I wanted to see if Neural Networks can also work well even with the limited dataset.

1. I started the problem by reading through some of the established papers on this dataset and their approach. This greatly helped me in understanding how the feature vectors were obtained which helped me in doing the right kind of data exploration.

2. There was not much preprocessing needed for the features except for label encoding, standardizing, and dimensionality reduction which became obvious steps after doing the data exploration stage.
3. With the data ready, it was simply a matter of fitting this to the models and recording the results. More trials were made to ensure that the optimal network is used for this dataset through hyper parameter tuning.
4. As shown in the results, neural network served to be a good classifier, even better than KNN, for this kind of dataset most especially if this dataset grows to include more classes and more samples.

Despite the dataset's limitation on having 99 classes with only 10 samples to train for each, the margin, shape, and texture features were very well-defined that the classifiers used were able to have good results. I notice though that neural network is data hungry, the current best model can predict the overall test with a log loss of 0.03188 and I'm sure that with more data fed into the network and with more layers and regularization techniques to avoid overfitting, this log loss can still be improved.

Upon exploring the data and researching more about the features, I found how the features generated to be interesting. Through image processing, defined points were gathered on the leaf's contour and compressed into a 64-element vector. I was amazed at the incredible ability to capture so much information and translate this to data especially for a feature like texture.

As for challenges, I found implementing neural networks to be a challenge. Though I understand how the neural network works layer to layer, I believe it is my lack of experience in using this algorithm that makes it currently a challenge. Rules of thumb on the number of neurons or epochs or activation functions are not readily available so there is a lot of knob-tuning to be done to achieve an optimized result. But after having implemented several networks for this project, my comfort level has gone up and I now have a better idea for future implementation.

iii. Improvement

One area for improvement is to explore tuning other hyper parameters like number of neurons and epoch. Though the 2-layer neural network does a good job in reducing log loss for this classification task, a 3-layer network would also work well for this particular dataset. But because of a 3-layer network's increased in complexity, more training data would be better for this model to predict better results. Doing hyperparameter tuning for a 3-layer model would also help improve the score. Developing a better model evaluation method particular for this kind of dataset would also be another area for improvement.

Besides the csv files in the dataset, there is also an images folder that contains 1,584 binary images of the different leaf id's in both the train and test. Though the 3 features in the dataset seem robust enough to produce great scores for leaf classification. This can further be improved by extracting additional features from the images. Using Scipy's imread function to read the .jpg file and binarize it, the images can be in the form of 1's and 0's. First step though is to make sure that these images would all have the same size so when each leaf sample is reshaped to 1 row – similar to how the MNIST data is set up – the dataset can then be processed in a Convolutional Neural Network. Though the current 64-element vectors of shape and margin already capture the leaf's shape, there are still limitations to how these features were computed, that can be offset by using the binary images as an additional feature.

References:

https://keras.io/

https://www.kaggle.com/c/leaf-classification/kernels

https://www.researchgate.net/publication/266632357_Plant_Leaf_Classification_using_Probabilistic_Integration_of_Shape_Texture_and_Margin_Features

http://www.hrpub.org/download/201308/cr.2013.010101.pdf