Classification on the UCI Letter Recognition Dataset

DATS_6202_10
Project report
Group# 17
Jacob Maibach and Mrigank Tiwari

Introduction

Letter recognition is often considered a toy problem to show off the power of machine learning classification, primarily because it is an elegant example where simple machine learning models do well. However, this is largely because very clean and homogeneous data is used. However, using more complex data, this can become a quite difficult problem.

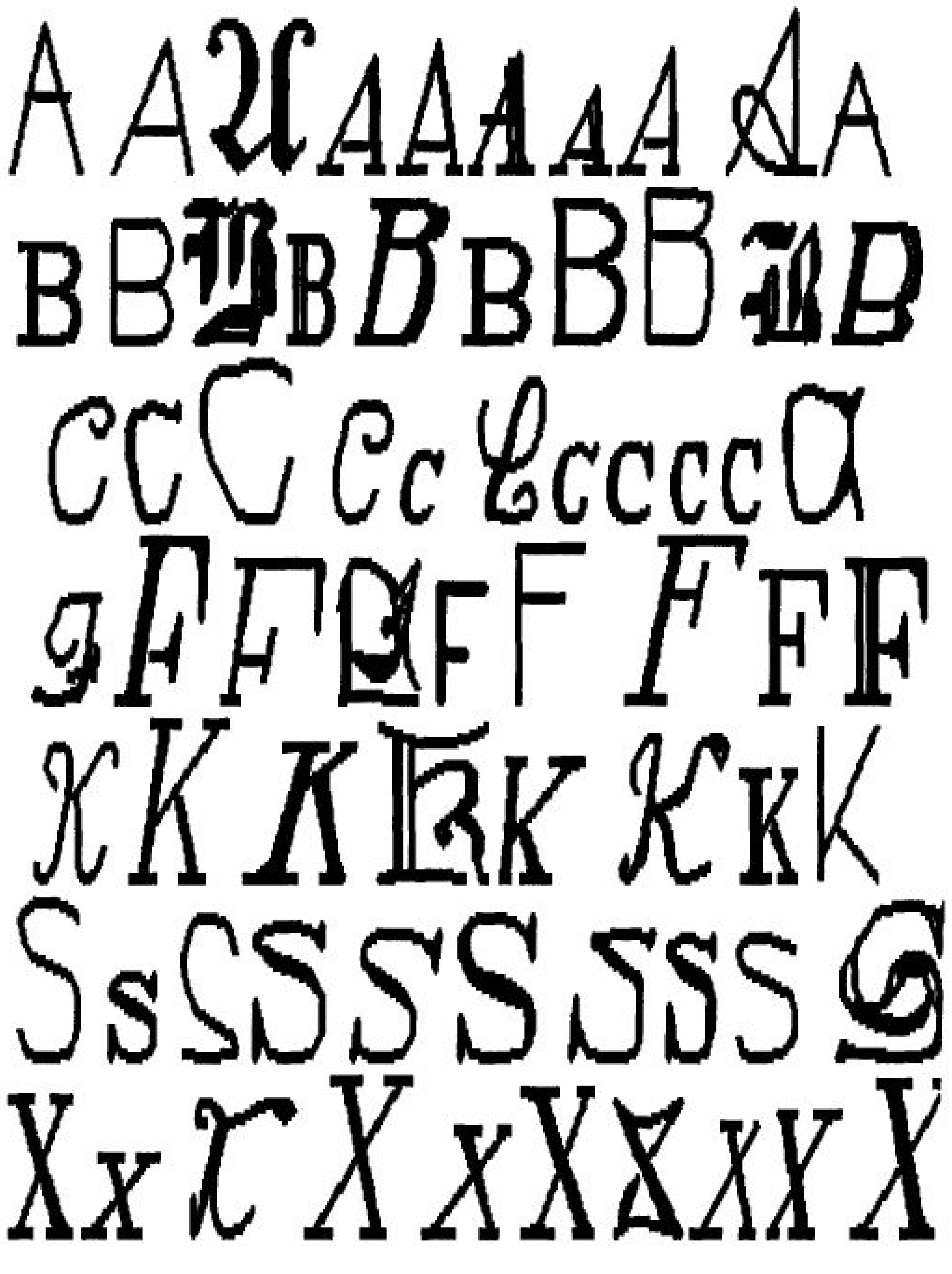
In this report, we look an example of exactly that. The “letters recognition data” from the UCI machine learning repository contains summary statistics (from image analysis) of randomly-distorted english letters in 20 different fonts. This data was originally created to train Holland-style classifiers in a 1991 paper by Frey and Slate.

Dataset description

“The character images were based on 20 different fonts and each letter within these 20 fonts was randomly distorted to produce a file of 20,000 unique stimuli. Each stimulus was converted into 16 primitive numerical attributes (statistical moments and edge counts) which were then scaled to fit into a range of integer values from 0 through 15”. The data is already mined and statistically scaled so we can directly start with classification. There are total 17 attributes which are described as below.

1.	letter	capital letter	(26 values from A to Z)
2.	x-box	horizontal position of box	(integer)
3.	y-box	vertical position of box	(integer)
4.	width	width of box	(integer)
5.	high	height of box	(integer)
6.	onpix	total # on pixels	(integer)
7.	x-bar	mean x of on pixels in box	(integer)
8.	y-bar	mean y of on pixels in box	(integer)
9.	x2bar	mean x variance	(integer)
10.	y2bar	mean y variance	(integer)
11 .	xybar	mean x y correlation	(integer)
12.	x2ybr	mean of x * x * y	(integer)
13.	xy2br	mean of x * y * y	(integer)
14.	x-ege	mean edge count left to right	(integer)
15.	xegvy	correlation of x-ege with y	(integer)
16.	y-ege	mean edge count bottom to top	(integer)
17.	yegvx	correlation of y-ege with x	(integer)

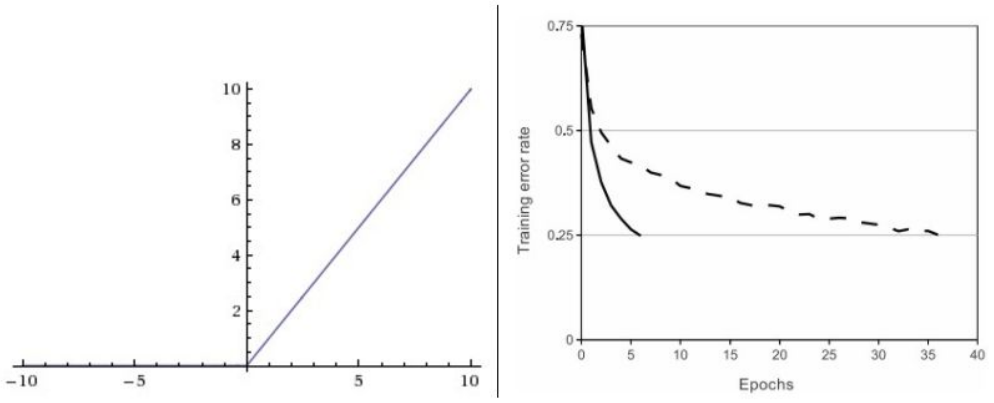
Examples of character images are shown in figure below.



Machine Learning algorithms used

To fulfil our purpose of classifying the letter images based on the data from distorted images, we tried to implement three algorithms. Neural networks, logistic regression and an ensemble of logistic regression with clustering.

Neural network methodology has evolved over time since its origin decades ago. And with the developments came very efficient activation functions, the great backpropagation algorithm for efficient classification. Here we used one of the most suitable activations known as 'ReLU'. ReLU has several advances over older and traditional activations like 'sigmoid' and 'tanh'.
It does not saturate in positive region minimizing the chances of vanishing gradient problem, it is computationally efficient and does converge relatively faster. The graph below shows the ReLU function.



Left: Rectified Linear Unit (ReLU) activation function, which is zero when x &lt 0 and then linear with slope 1 when x &gt 0. Right: A plot from Krizhevsky et al. (pdf) paper indicating the 6x improvement in convergence with the ReLU unit compared to the tanh unit.

Below equations show how the SoftMax output layer works to calculate probabilities of each class and the cross-entropy loss.

$$p_k = \frac{e^{f_k}}{\sum_j e^{f_j}} \qquad\qquad L_i = -\log\left(p_{y_i}\right)$$

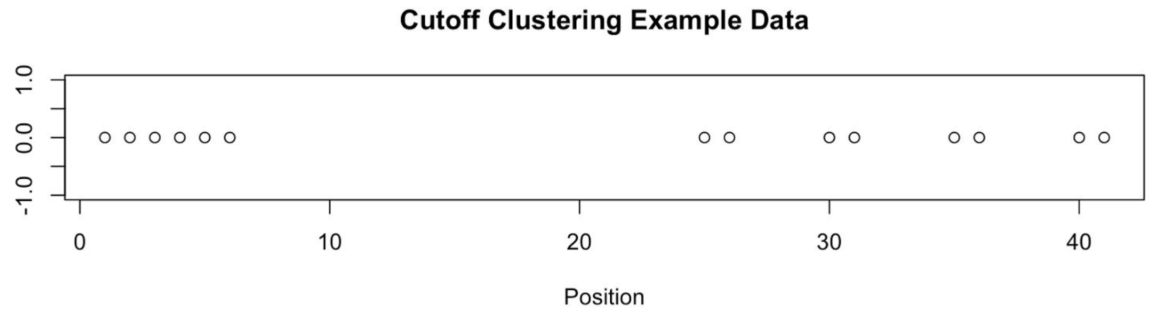
K-means:

Given a hyperparameter k, k-means tries to fit the data into k clusters. It does so by determining k centroid vectors, where each point is clustered by placing it in the cluster of the nearest centroid. The training method iteratively improves the centroids by first clustering according to the current centroid vector, and then replacing each centroid by the center (mean) of its corresponding cluster. Starting with randomly initialized centroids, this method can converge to a final result in only a few iterations. It finishes when the centroids do not change between iterations (they stabilize).

Cutoff clustering:

The premise of cutoff clustering is that we give a hyperparameter, called the cutoff distance, which determines the density-scale of clusters we are looking for. In particular, points are clustered if they are closer together than the cutoff distance. Thus for small cutoff distances, we get many, very dense clusters, and for large cutoff distances, we get a few, low density clusters. We can think of cutoff clustering as giving us the most clusters such that the distance between them is bigger than the cutoff distance. Notably, since cutoff clustering only cares about density, we can get arbitrarily wide clusters (large distance), whereas k-means only cares about width. Additionally, cutoff clustering can create clusters of any shape (they can even go around other clusters), while k-means creates clusters that are contained in circles.

You can see an example 1d data set for cutoff clustering below (ignore the y axis). For a cutoff distance of 2, the result is five clusters, with each of the pairs on the right being a cluster. However, for a cutoff distance for 6, the four pairs on the right become one cluster, as the distance between them is only 4. The group on the left is one cluster for each.



Experimental setup

The first attribute (letter) is our target representing letters in capital, but to make our models learn we had to encode these to a different format. In case of neural networks, as we used a softmax output layer, we had to convert the target to 26 numbers between 0-25. For example 'A' is coded into '0', 'Z' is coded into '25' and similarly others. These 26 numbers represent the 26 classes that our softmax output layer will make classifications among.

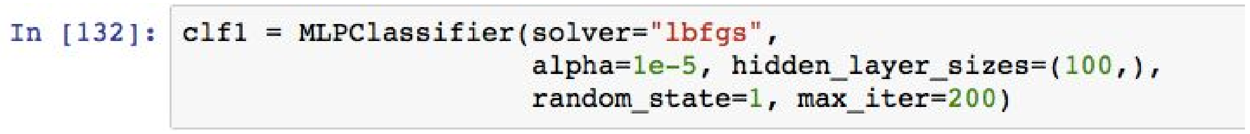
In case of logistic regression, we used a different encoding. Each alphabet is an array of length 26, where all the elements are 0 except the the the element representing the number of that alphabet sequentially. For example, ‘C’ is represented by below vector.

[0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]

To start with, we implemented both logistic regression and neural networks using sigmoid activations. Our initial NN setup consisted of one hidden layer and an output layer, where the hidden layer had 32 neurons and the output with 26 neurons, all of these were sigmoids. The configuration of logistic regression network also comprised of 26 sigmoid outputs. We visioned to get the probabilities of an observation being each of 26 classes. The background research we did before start our work showed that although there are better activations than sigmoid and sigmoid might run into some issues like “vanishing gradient” during back-propagation, still sigmoid has the reputation of getting the job done with fair enough accuracies. So our attempt of implementing these algorithms, although was successfully training the networks, was really poor with predictions in training set itself.

After multiple attempts, we decided of working on the problem using the machine learning library “scikit-learn”, which we must admit is on the epitome of optimization. There are a very few hyperparameters we need to worry about to start with and we can try different combinations as we go along. Now let's take one at a time.

The neural_network.MLPClassifier is specifically written to perform classifications with some default hyperparameters that user need not worry about. Only parameters we are required to pass are like ‘hidden_layer_size’, ‘max_iter’.

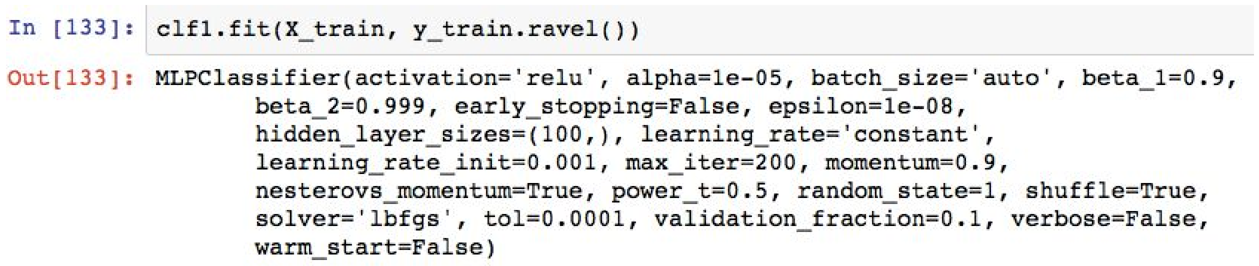


```
In [132]: clf1 = MLPClassifier(solver="lbfgs",
                               alpha=1e-5, hidden_layer_sizes=(100,),
                               random_state=1, max_iter=200)
```

Actually, there is no need to pass any parameters while building your model, we started with our choice of hidden_layer_size and max_iter. The process where we train our network, might give some convergence warning in case maximum iterations are not enough for the learning to converge. Below are the by default set parameters in case one doesn’t pass anything while building the network.

__init__(hidden_layer_sizes=(100,), activation='relu', solver='adam', alpha=0.0001, batch_size='auto', learning_rate='constant', learning_rate_init=0.001, power_t=0.5, max_iter=200, shuffle=True, random_state=None, tol=0.0001, verbose=False, warm_start=False, momentum=0.9, nesterovs_momentum=True, early_stopping=False, validation_fraction=0.1, beta_1=0.9, beta_2=0.999, epsilon=1e-08)

Training the network, happens simply enough by calling fit function on our built model and parameters to be passed are training X and y data, as below. The output shown here is the one without warning of less max_iter. We had several such warning instances and it was just solved with increase in number of iterations.



```
In [133]: clf1.fit(X_train, y_train.ravel())

Out[133]: MLPClassifier(activation='relu', alpha=1e-05, batch_size='auto', beta_1=0.9,
                 beta_2=0.999, early_stopping=False, epsilon=1e-08,
                 hidden_layer_sizes=(100,), learning_rate='constant',
                 learning_rate_init=0.001, max_iter=200, momentum=0.9,
                 nesterovs_momentum=True, power_t=0.5, random_state=1, shuffle=True,
                 solver='lbfgs', tol=0.0001, validation_fraction=0.1, verbose=False,
                 warm_start=False)
```

And that's it, our neural network is ready to make predictions using the predict function of this MLPClassifier class. It does so with good prediction accuracies.

Clustering:

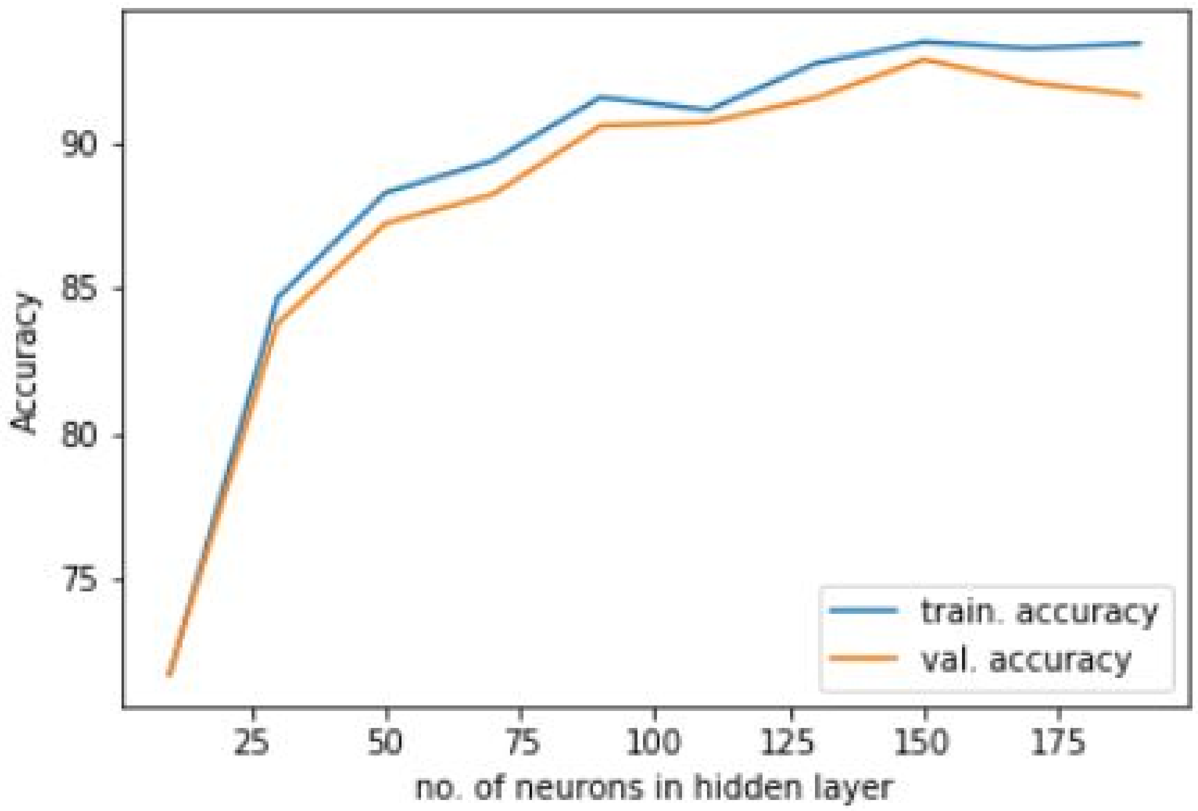
We decided to apply two clustering methods to the data, and to use the resulting labels as predictors in a logistic regression model. We chose the two clustering methods to hopefully produce substantially different results, so they would be useful in an ensemble model. The first clustering method we used was k-means clustering. The second is called cutoff clustering.

Finally, we created an ensemble model by using k-means predictions and cutoff-clustering predictions as input to a logistic regression model. Using k = 26 for the k-means, and a cutoff distance of d_c = 3.5, we fit a model to the resulting 70 predictors. It should be noted that we chose the cutoff based on the performance on a small subset of the data. We additionally at first selected k based on performance, and used k = 260 and got good results, but it we did not use it in the final model due to the computational time it took to make predictions. Thus we used k = 26, since it did decently and seemed natural.

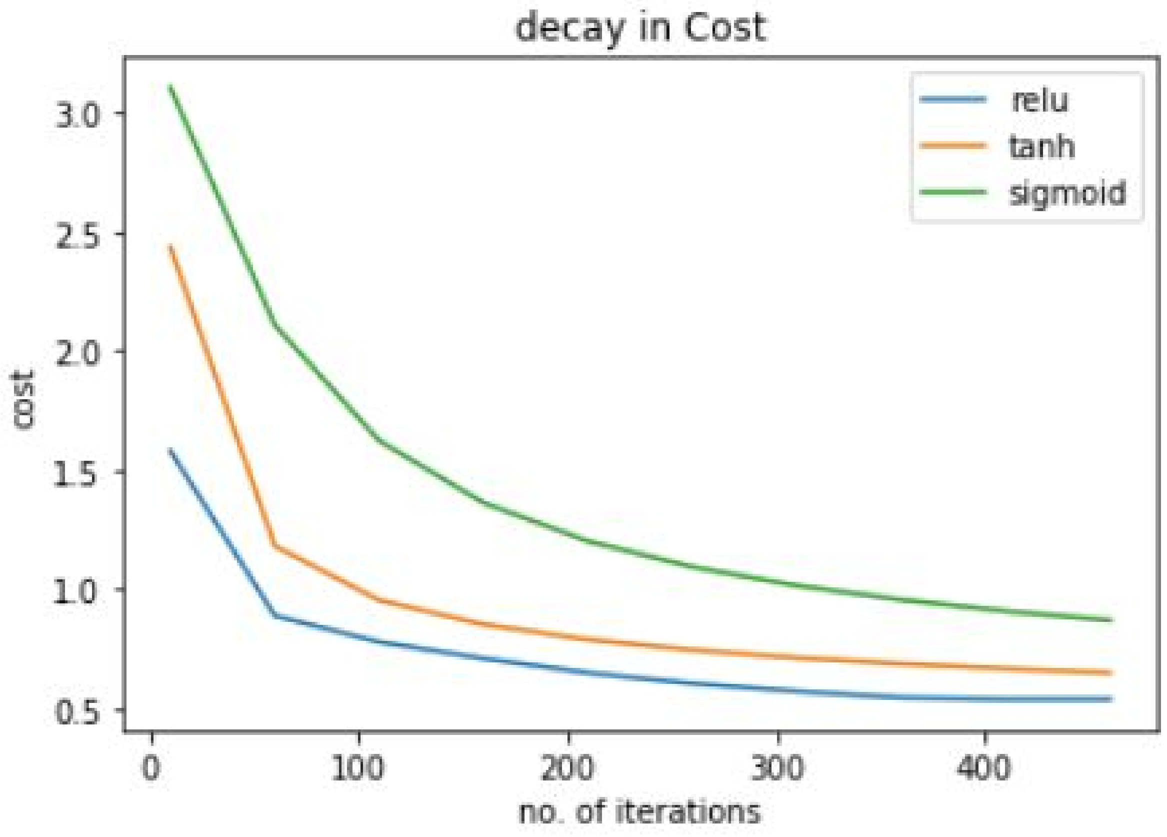
Results

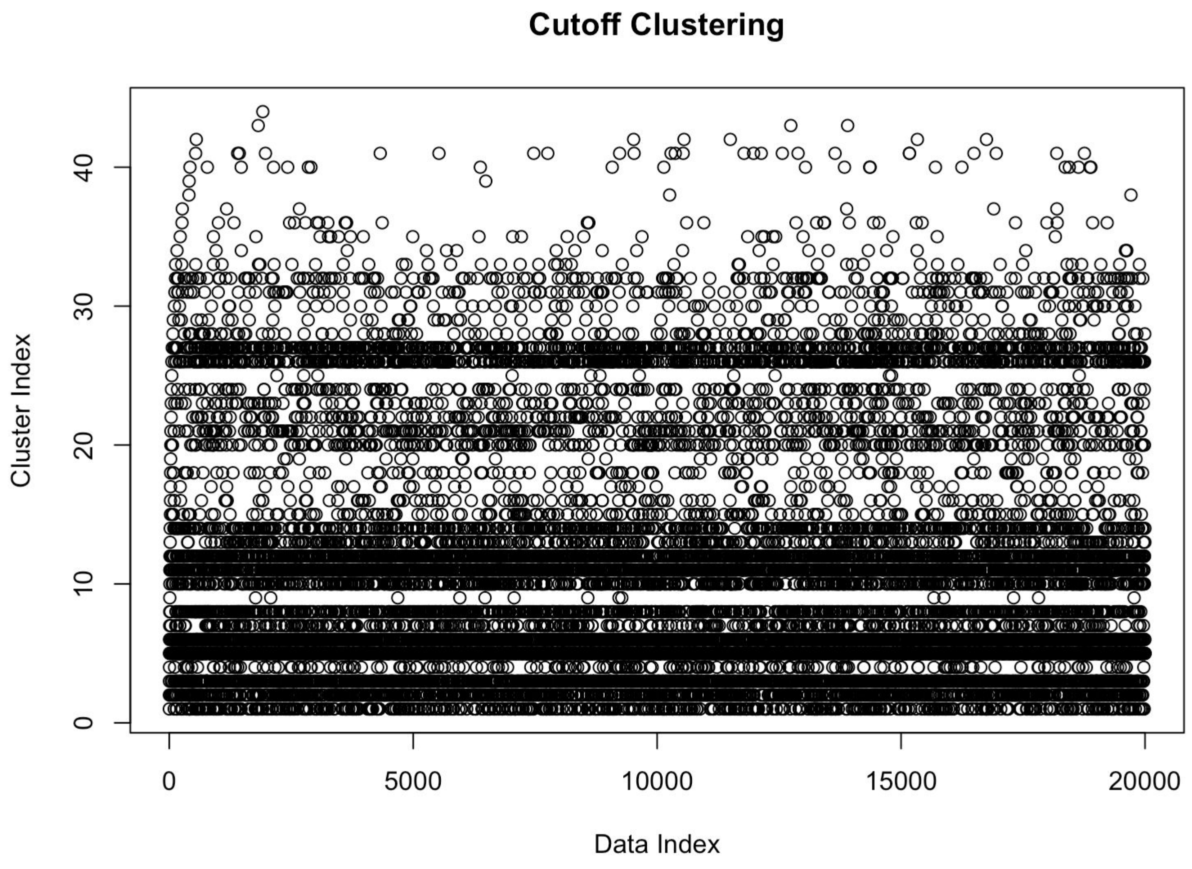
Plots:

Below is a comparative plot of prediction accuracies versus the number of neurons in hidden layer. So we see that the both training and validation accuracies are on rise till 160 neurons where the thin gap between the two plots represent low overfitting. But post 160 hidden neurons the validation accuracy starts to go down and we can presume that any further increase in the neurons is only going to increase the over-fitting.



Below plot shows a comparison in cost decay with number of iterations for ‘relu’, ‘tanh’ and ‘sigmoid’ activations.





The above plot shows the clusters in the cutoff clustering. Each horizontal black line is a very large cluster, and the white space for cluster index 9 is a very small cluster. The relative symmetry across data index indicates that the data is in no particular order. Note that we have roughly 20 large clusters, and number of relatively large clusters. This is a good sign, since these all are likely meaningful clusters.

Metrics:

The three metrics we use are accuracy, maximal confusion rate, and average confusion rate. The accuracy is simply the proportion of correct predictions. The confusion rates both are determined by the confusion matrix. We normalize the rows of the confusion matrix (i.e. across actual predictions), and then let the confusion rate between two letters be the determinant of the corresponding submatrix. Symbolically, for letters x, y we have a confusion rate

$$C(x,\, y) \;=\; P(x\,|\,x) * P(y\,|\,y) \;-\; P(x\,|\,y) * P(y\,|\,x)$$

where P(y | x) is the probability of predicting y given that the actual is x.

The rate is positive one when the prediction is completely correct, zero when it is equivalent to random, and negative one when it is completely wrong. Confusingly, these confusion rates actually increase as the model does better. Naming-wise, it might be appropriate to refer to the negative of this. Nevertheless, we can now define maximal confusion rate (MCR) and average confusion rate (ACR). The maximal confusion rate is

$$MCR = min_{x,y}\ C(x,\ y)$$

which is the confusion rate when the model does worst (it is maximally confused). The average confusion rate is

$$ACR = avg_{x,y}\ C(x,\ y)$$

which is quite literally a measure of average performance. It should be noted that these both measure systematic misprediction, versus random misprediction. Therefore, they reflect aspects of the model not reflected by the accuracy. Additionally, it should be noted that confusion rates are nonlinear, and are generally skewed towards having larger magnitudes.

Model Evaluation:

	Logistic Regression	Multilayer Perceptron	Ensemble
Accuracy	0.73	0.95	0.37
Max. Confusion Rate	0.65	0.88	0.01
Avg. Confusion Rate	0.97	0.99	0.88

As you can see in the table, the average confusion rate of each model was relatively low. This means that each of the models was able to distinguish well between most of the letters. In particular, the perceptron did nearly perfectly in distinguishing letters. From the maximal confusion rate, you can see that the ensemble model had difficulty distinguishing between two letters (S and Z), and could do so only slightly better than random. The perceptron had some difficulty distinguishing G and C, and the logistic regression had some difficulty distinguishing O and H (which is quite surprising). Lastly, we see from accuracy that logistic regression only predicted decently well despite having high average confusion rate. This suggests that its prediction problems were due to random variation and not systematic variation (i.e. it merely failed to pick up on some patterns in the data).

Technical Note:

We initially attempted to implement both logistic regression and the multi-layer perceptron in python using numpy (using gradient descent for training). We spent several weeks trying to fix our code, and implemented momentum gradient descent to improve the logistic regression training, and nesterov gradient descent to improve the perceptron training. However, we were not very successful in improving these and ended up using R and scikit learn to run the logistic regression and perceptron, respectively.

References:

P. W. Frey and D. J. Slate. "Letter Recognition Using Holland-style Adaptive Classifiers". (Machine Learning Vol 6 #2 March 91)

J. Maibach, Undergraduate Thesis, The George Washington University (2016). “Theoretical Foundations for Cutoff-Clustering in Analysis of Transcription Factor Distributions”

Scikit-learn package documentation

Ruder, Sebastian. “An overview of gradient descent optimization algorithms”, 2016 link

Karpathy, Andrej. “Convolutional Neural Networks for Visual Recognition”, , link