Comparison Of Machine Learning Models
Introduction to Machine Learning and Pattern Recognition
EECE 5644
Alex Duffy, Alex Mendola, Andrew Popeo, and Jason Booth

Introduction:

With the continual proliferation of new and improved machine learning algorithms, it can be confusing when trying to establish which model best fits a given task. This is an issue in machine learning at large where new advancements create trends in the field and promote a single algorithm as the best for an era. There are pros and cons for each model; this paper will attempt to go into detail on four of them. Beginning with a discussion of each algorithm and its performance in the context of synthetic data, some cases will be highlighted where the models can perform poorly. Each model will then be applied to two real world datasets and their performance will be analyzed. In the end, this paper will elucidate some of the major differences between each model and will aid in developing a deeper understanding of when and where they should be used.

Related Works:

Neural networks have proven to be extremely effective in a wide variety of applications. (Dingankar, 1999) For example, neural networks revolutionized computer vision tasks when a neural network was able to beat out traditional methods in an Imagenet competition. (Alex Krizhevsky) Recurrent neural networks have been able to generate text that mimics inputs, creating new books and nearly functional code. (Karpathy, 2015) This shift has partially been enabled by a rise in computing power. Specifically, development on GPU's has enabled the implementation of Deep Neural networks at a scale that was only previously accessible in theory. (NVIDIA, 2015) As such, there has been an industry trend with the hopes of applying neural networks to a variety of domains, and neural networks have become somewhat of a buzzword. Other models have seen their time in the spotlight as well. SVM was an extremely popular model in the 90's and early 2000 for its unique ability to reach the global optimal solution during training. The discovery of the kernel trick also increased its popularity, because it enabled finding non-linear decision boundaries rather quickly. (Gunn, 1998) The spikes in popularity of algorithms over time can cause confusion regarding which models are best suited for a given application.

There are hundreds of types of machine learning models in practice, and attempts have been made to compare them empirically. (Rich Caruana) Ultimately, general conclusions about their performance are hard to make. The "No Free Lunch Theorem" corroborates this finding, claiming that a general solution to every problem does not exist, and models must be specialized to the specific problem that they are being used for in order to obtain optimal performance. In this paper, general considerations for selecting a machine learning algorithm are explored in the context of four common algorithms.

Methods:

Section 1: Description of Machine Learning Models

KNN, Naive Bayes, SVM, and Neural Networks represent a span of machine learning models that are used across industry. To get a better sense of how each of them work, this project

began by implementing each algorithm using Jupyter Notebooks. These notebooks are included with the supplementary code for this paper, and will be referenced when addressing differences between each algorithm.

KNN, or K Nearest Neighbors, is a rather straightforward algorithm that operates by classifying a test datapoint by features of the K nearest training data points surrounding it. K is the hyperparameter of the model that must be tuned during training. A common choice for K is the square root of the number of instances. (DUDA) The decision function for determining which class the point is assigned to depends on implementation. Some will use a majority vote technique, where the majority class of the nearest points decides the classification. Another method is to weight the voting for each point by the distance it is away from the test data point. When implementing KNN, another consideration is the distance metric used for determining the K-nearest points. Most commonly, Euclidean distance is used, but other metrics such as Manhattan distance or other order Minkowski distances may also be used. An implementation of a weighted Euclidean distance KNN is included in a Jupyter notebook.

One of the implications of KNN is the effect on training time and testing time for the model. KNN uses no internal representation or estimation of parameters for the data. During training time, it will simply store the raw training data, and prepare to use it for predictions. On the other hand, the testing or classification step becomes very expensive. Finding the k nearest neighbors is an *O(N)* operation, requiring comparison to all other data points. This makes KNN very compute intensive at runtime, and also very memory intensive to store all of the training data. This is an important aspect of KNN which will influence use cases where it can be deployed. KNN is not able to be deployed in real time systems, and it is difficult to pass trained representations of the model to others that may want to use it.

There have been some efforts to optimize the runtime of KNN by creating more intelligent data structures internally to reduce the complexity of the calculation of the k nearest neighbors. One approach uses a K-D tree, which partitions the data based on a binary search algorithm created from the features of the data. (Bentley, 1975) Another uses a ball tree, which attempts to create hyperspheres encapsulating regions of the data and calculate nearest neighbors based on the region that the test point falls into. Using these approaches requires the creation of the data structure during training time, and brings the search time complexity down during testing to approximately *O(log(N))*. Sklearn's implementation of KNN for classification allows for the selection of a brute force, K-D tree, or ball-tree approach during the creation of the model.

Naive Bayes is an algorithm which makes some basic and often incorrect assumptions about the data, but can still have impressive performance in practice. Naive Bayes has found major success in applications like text classification and medical diagnosis. (I.Rish) The algorithm works by attempting to calculate the probability a test data point belongs to each class, and classifying the point to the class with the highest probability. The calculation of the probability of a point belonging to a class assumes independence for each feature, and is equal to: $P(C|X) = \prod_{i=1}^{F} P(X_i|C)$ where F is the number of features in each datapoint.

The calculation of $P(X_i|C)$ depends of the type of data input into the model. For discrete categorical data, the prior distribution can be calculated directly by using the counts of each occurrence in the training dataset. For continuous input data, some kind of assumption about the underlying distribution of the input features must be made so that the probability of each feature

can be calculated. Commonly, a normal distribution is used, and the parameters of the normal distribution are estimated for each feature based on the data that belong to each class. A version of a continuous naive bayes with a normal distribution assumption is included with this paper. In addition to using the class conditional probabilities,

Because of the assumptions made by the algorithm, the training time and testing time are fairly fast compared to other algorithms. Also, the memory overhead is very light because only a small amount of information regarding the distribution of each feature for each class needs to be stored to represent the entire model. This is important because the model could be trained, and then easily hosted and deployed in practice.

Naive Bayes will perform poorly in situations where its assumptions start to break down. An example of this is included below in the comparison of models Jupyter notebook. One interesting commentary on when the models perform poorly is from I. Rish at IBM. (Rish) Rish conducted an analysis of Naive Bayes on several different datasets, and concluded that the model tends to perform better when there is lower entropy within each feature distribution. This make sense, because feature distributions with lower entropy means that most of the features for a class are concentrated around a single state, and the assumptions of the model become more accurate. Rish found that as entropy decreases to zero, accuracy will increase to one hundred percent.

Support Vector Machines is a model which is used to find the best fitting hyperplane between two class of data. This hyperplane attempts to create the maximum separation between the two classes. Once the hyperplane has been calculated, it is straightforward to calculate the class of new data points in constant time, using the equation: $class = sign(w \cdot x + b)$

The calculation of the hyperplane is defined as a constrained quadratic programming problem. More details on the math involved in the creation of the constraints, and the dual formulation are included in the SVM Jupyter Notebook. Once the dual is formulated, the complexity for solving for the hyperplane is polynomial in time $O(max(n, d)\ min(n, d)^2)$. (Chapelle, 2006) This is extremely slow compared to previous models, and the long training time should be taken into consideration when choosing a model. However, it should be noted that the mathematical representation of the dual is a convex problem, meaning that the solution reached will be optimal given the training data.

Another interesting thing about SVMs, is the ability to use Kernel functions to find non-linear decision boundaries between classes. A kernel function is a function that corresponds to an inner product in a higher dimensional space. Because the SVM only requires the optimization of a dot product, if a kernel can be used that makes the calculation of this dot product constant time, then the hyperplane in that space can be found and projected down onto the original space of the problem to create a non-linear decision boundary, without added computational complexity. One such kernel is called the radial basis function kernel. This kernel uses the math behind taylor expansions to approximate the infinite dimensional dot product. In this space, a decision boundary for 100 percent accuracy on the data is bound to exist, and overfitting on the training data can be a large problem.

Artificial Neural Networks (ANN) are a classification method which work in a fashion loosely analogous to the way the brain functions, with a network of neurons each performing a relatively simple relaying function. An ANN consists of a layer of input units , a layer of output units, and one or more *hidden layers* in between. The units (or *neurons)* in an ANN each compute the weighted sum of input values from preceding connected units, and then usually apply a nonlinear activation function to the value of the linear combination. The output of this

activation function is then forwarded to the next layer of units. The weight is given by the equation $\sum_{i=1}^{d} x_i w_{ji} + w_{jb}$ where d is the number of preceding units connected to the current unit j, each w_{ji} represents a weight applied to the value sent from unit i to unit j, and w_{jb} is an additional bias weight. The output of the unit then is $f(\sum_{i=1}^{d} x_i w_{ji} + w_{jb})$ where f is the nonlinear activation function. This function is often a sigmoid, hyperbolic tangent, or relu (rectified linear unit) function. When an input sample is fed into the input layer of the ANN, the first layer feeds forward into the hidden layers, which feed forward into the output layer. The output often consists of one unit per class, and the network can assign a classification to the input data by selecting the class corresponding to the output unit with the greatest activation. In order to train a supervised ANN, the true class label for a given input is compared to the output layer activation and the weights throughout the network are then adjusted in order to reduce the error between the desired output and the actual output given during the feed-forward process. The basic training method for neural networks, stochastic gradient descent, works by computing the gradient of each weight in the network with respect to the network's loss function and then making a weight adjustment equal to the negative gradient multiplied by a constant learning rate: $\Delta w_{ij} = -\delta C/\delta w_{ij} * \eta$, where w_{ij} is the weight to be adjusted, C is the cost function, and η is the learning rate.

Because stochastic gradient descent performs weight adjustments for every single input sample which are from different classes, the loss-time curve is not smooth and fluctuates. In batch gradient descent, the backpropagation step is applied after a batch of several input samples are fed forward. This smooths out the loss-time curve during training. There are also more advanced training optimization functions such as the RMSProp algorithm which maintains separate training rates for each unit based on recent gradient magnitudes, and the Adaptive Moment Estimation algorithm which also maintains separate learning rates which depend on a moving average of previous gradient magnitudes as well as the second moments of the gradients. (NG, n.d.) (Diederik P. Kingma, 2017)

An artificial neural network with just one hidden layer of finite length can approximate any continuous function to arbitrary precision, according to the Universal Approximation Theorem. (Domonkos Tikk, 2001) As a result, neural networks are prone to overfitting, especially when data is sparse. There are a number of ways to avoid overfitting. First, early stopping can be used, where training is stopped when the accuracy on a cross-validation test dataset stops increasing. Second, dropout layers can be included in the network which set a randomly selected portion of neuron outputs to zero during training. This in effect prevents the network from over-relying on particular features for classification which could otherwise cause overfitting.

In datasets where information can be extracted from the spatial relationship between input features such as audio or images, an ANN can learn to extract spatial feature information from the inputs before the fully-connected hidden layers by adding one or more convolutional layer. Convolutional layers perform a cross-correlation between the input and a number of feature-extracting filters. The coefficients of the filter become a trainable parameter in such networks. In order to increase speed and generality in a convolutional neural network (CNN), the

outputs of the convolutional layers can be downsampled by max-pooling which reduces a region of the output image to the maximum value in the region.

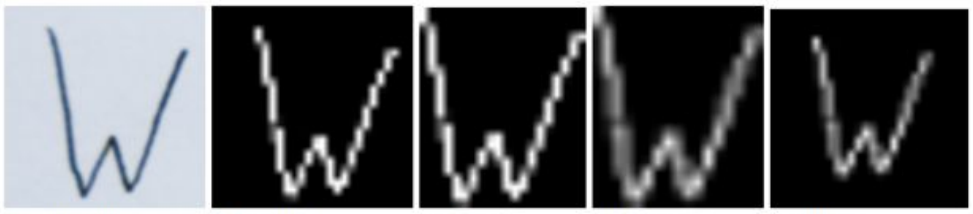
In order to augment the comparison of models in this report, three neural networks are implemented in TensorFlow, in addition to two Sci-Kit default implementations with 2 and 4 hidden layers. The first TensorFlow model is an ANN with one hidden layer, the second model includes a second hidden layer, and the third model adds two convolutional layers each with max-pooling, plus a dropout layer before two hidden layers. All ANN implementations in this report use a relu activation function and Adam optimizer with learning rate of 0.0001. Additionally, the TensorFlow implementations incorporate early stopping based on convergence of accuracy on the test datasets.

Section 2 EMNIST:

In order to compare the models presented above on real data, the SciKit implementations as well as all three TensorFlow neural networks, including the convolutional neural network, were run on the balanced EMNIST balanced dataset. (G. Cohen) The EMNIST dataset is an extension of the MNIST handwritten digit dataset, with the addition of uppercase and lowercase letters. In total, the balanced EMNIST dataset contains 47 classes: 10 digits, plus 26 uppercase letters, plus 9 lowercase letters. Lowercase letters such as 'o', which are written the same way as their uppercase counterpart, are not given separate classifications. The dataset contains 112,800 labeled training images plus 18,800 labeled test images. Both subsets contain uniform class frequencies.

The motivation for applying the models to the EMNIST data were many. Using the EMNIST data would solidify understandings of the similarities and differences of the models by applying them to real life data that could be physically seen. Additionally, the images in the EMNIST dataset provide an opportunity to test the spatial feature-extraction offered by convolutional neural networks. Another factor that led to the choice of EMNIST over similar datasets including MNIST, one of the most used datasets in machine learning, was the initial desire to gather novel data. While that proved to be practically infeasible to do in large scale, with the choice of EMNIST as a dataset, the goal was to convert a team member's handwriting into an identical format allowing the varying methods to be tested on that, completing the development circle from reading physical handwriting into text on the computer screen. This proved easier said than done. Upon investigation into *EMNIST: An extension of MNIST to handwritten letters,* the paper accompanying the release of the dataset, the team behind it explained the conversion process used to convert the NIST dataset.

"The original images are stored as 128x128 pixel binary images… A Gaussian filter with $\sigma = 1$ is applied to the image to soften the edges… As the characters do not fill the entire image, the region around the actual digit is extracted. The digit is then placed and centered into a square image with the aspect ratio preserved. The region of interest is padded with a 2 pixel border when placed into the square image, matching the clear border around all the digits in the MNIST dataset. Finally, the image is down-sampled to 28x28 pixels using bi-cubic interpolation. The range of intensity values are then scaled to [0,255], resulting in the 28x28 pixel gray-scale images"



(a)128x128 (b)28x28 (c)20x24 (d)17x20 (e)28x28

Similarly the team member's handwriting was converted using the following steps: A picture was taken using a high resolution camera of a handwritten sentence “WE LOVE ML 2018” using the RAW image format. After brightening and increasing the contrast of the image in Lightroom it was exported to Photoshop where each character was manually extracted and placed in a
128x128 square. A gaussian blur with $\boldsymbol{\sigma} = 1$ was applied just as in the conversion process described above(a).

These characters were then processed with code written to convert them further, derived from a project implementing one’s own handwritten digits with the MNIST data set. (Kroger, 2016) The code used OPENCV to first convert the image to grayscale using the threshold of 128 out of 255 for brightness, invert the colors, and resize the image to 28x28 pixels since bi-cubic interpolation is used(b), before extracting the region of interest by removing any line that was all black(c). Finally came the process of centering the character, by first using the center of mass measurement of the image with the multi-dimensional image processing pack from the scipy library(d), before padding the character with enough black lines to achieve a 28x28 resolution,at minimum the 2 pixel border required(e).

Section 3 Application to SMS:

The models were also trained on a dataset of 5574 SMS text messages, classified as either spam or ham. In this set, ham is considered as any message that is not spam. It was taken from from the UCI’s machine learning public repository and contains 86.6% ham and 13.4% spam messages. (Almeida) The samples come in the form of a text file, where each message is separated by a new line, and each message’s associated label are the first word of the line. A sample of the a ham and spam message are shown below:

To get this text into a format that can be interpreted by each model, the messages and labels were parsed from each line. Then, a bag of words approach was used to transform the messages into a vector of the unique word frequencies in each message. In a bag of words model, each unique word in a defined dictionary represents one feature. (Shaikh, 2017) A predefined dictionary can be used for this, but to reduce dimensionality, the dictionary can be generated by scanning the words in the specific dataset. To further decrease dimensionality, the dictionary can be created with lemmas instead of whole words. Using this method, words such as “run”, “runs”, “ran”, and “running” would all be considered as the same lemma “run”. Implementing this in python was straight forward, with TextBlob converting the dataset into a vector of lemmas for each message, and sklearn’s built-in CountVectorizer transforming each message into a vector of lemma frequencies. Sklearn’s TF-IDF transformer was also used to normalize this data and reduce the importance of common words such as “this”, “is”, and “an”. Using these steps, the entire dataset was converted into a 1D vector of labels and vector of messages with the shape [numberOfSamples, numberOfWordsInDictionary].

RESULTS:

Section 1:

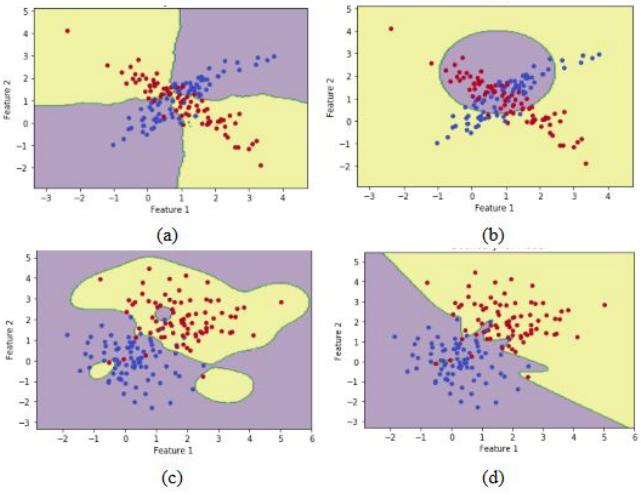
To test the differences between each model, a Jupyter Notebook was created to generate various types of synthetic data and perform classifications on them using the Sci-Kit Learn machine learning package in Python. This package is extremely well supported by the community and is used in industry to perform data exploration and analysis. This library has built-in methods for each of the models mentioned above providing a more well supported and standardized implementation of these algorithms than creating new versions of the models.

The data generated consists of a small dataset with slightly overlapping gaussian distributions, a small dataset with correlated distributions overlapping each other, and a large dataset with little overlap. The models used include: a brute force KNN model, Naive Bayes, a linear as well as RBF kernel SVM, and both a simple and complex neural network. The simple neural network consisted of only two hidden layers with ten neurons each, while the complex network had four hidden layers, each with 100 neurons. This diversity of models and datasets is meant to highlight in which situations a model may perform poorly or optimally compared to other models. The results will be discussed generally, but for a full record of run results, please consult the "ModelComparisons" Jupyter Notebook.

Timing:

The timing generally coincided with what was expected for the complexity of each algorithm. KNN took no time at all to train, and took longer than all other models during prediction. Naive Bayes was comparatively very fast in both training and predicting. SVM took a significantly longer time to train than the previous two models, while Neural networks took the most out of the four with the complex model taking the longest. One interesting result was that the RBF kernel took significantly longer than the linear kernel on the large dataset. This is interesting because the complexity of solving for the two hyperplanes should theoretically be the same. This highlights an important aspect of using these models in practice. The implementation of the libsvm library that this model is built on has an indeterminate complexity between quadratic and cubic depending on the optimization problem it solves. There is also the constant time calculation steps for the kernel itself, which can really add up when running on a lot of data. Although in theory these runtimes should be the same, this experiment shows that in practice it depends on the library, the kernel used, and the dataset.

Fit of each model:



KNN (a), Naive Bayes (b), SVM RBF (c), and Complex Neural Network (d) boundary plots on Synthetic Data

Overall KNN was fairly accurate when classifying the test dataset. It performed between 80 and 99 percent, and its accuracy was changed very little by the distribution of the data. Naive Bayes performed well on all of the datasets except for the overlapping

data, where it did a poor job of classifying the test data points. This is because of the breaking of the assumptions made about the independence of features within a class, which was explicitly not true in this case.

Generally, the SVM and Neural network models were the most accurate. One consideration for both of these models is the potential for overfitting. In the SVM RBF kernel, there is clear evidence of overfitting for some of the datasets. The use of a higher order kernel resulted in a highly non-linear decision boundary that started to overfit the training data. This resulted in a lower accuracy for the RBF kernel than the linear one which shows that a linear division boundary better fits the data. This demonstrates how important the choice of the kernel in SVM is when trying to apply it to real world data.

Just as with the nonlinear SVM RBF model, the complex (four-layer) SciKit neural network model shows overfitting when compared with the simple (two-layer) SciKit neural network, as shown above. The more complex network also took longer to train that the simple network. This goes to emphasize an important consideration when making when using an artificial neural network, tuning hyperparameters such as topology, training epochs, termination conditions, and learning rate in order to maximize learning while avoiding overfitting to the training data. Part of the problem in this case may be the relatively sparse data. In the case of the much larger uncorrelated gaussian dataset, the six-layer ANN does not overfit as drastically as for the sparse dataset. Furthermore, in this case, where the classes are linearly separable, a multi-layer neural network is simply not necessary.

Section 2 EMNIST:

The Naive Bayes classifier performs very poorly on the EMNIST dataset. This may be in part due to the classifier's assumption that features are uncorrelated. This is an invalid assumption in this case because a given pixel is more likely to be dark if its neighbors are dark. In some cases, Naive Bayes can still give good results when the features are not uncorrelated, but did not perform will for this dataset.

EMNIST Balanced Dataset Results

Model	Accuracy	Time to Fit	Time to Predict
Naive Bayes	0.282	2.649	8.456
K-Nearest Neighbor	0.785	60.760	3037.379
SVM (linear)	0.416	14134.4	0.118
SVM (RBF)	0.049	21567.9	2580
Neural Net (1 Layer of 800 neurons)	0.818	1852.1	0.168
Neural Net (2 Layers of 800 neurons)	0.828	2855.5	0.250
Convolutional NN (2 Layers of 80 neurons)	0.885	20171.6	7.381

K-Nearest Neighbor achieves better accuracy, performing as well as 78.5%, but takes an impractical amount of time to classify because it needs to search the data for the k-nearest neighbors at runtime for each new classification. With the sacrifice of some accuracy, classification time could be reduced by downsampling the dataset.

The performance of the SVM models was somewhat confusing at first. It should be noted that the linear SVM model uses a sklearn function specifically for the computation of linear kernels, which uses a different liblinear that is more optimized that the libsvm library used in the RBF kernel. This means that the run times of the two models cannot be compared directly. The linear model resulted in approximately 42% accuracy, and the RBF model performed extremely poorly, around five percent. To further diagnose the cause of this accuracy, a new jupyter

notebook was made to experiment with a subset of the data. There are hyper-parameters of the SVM RBF model that control how the model operates. The parameter C influences how strongly the decision boundary changes for mislabeled points. The parameter gamma influences how far the influence of a single training example reaches. Sklearn provides a great API for tuning the hyper-parameters of a model called GridSearchCV. (Pedregosa F.) This allows the user to input a list of parameters which will be tuned using cross validation. Using this method and sampling values for gamma and C in the SVM RBF model, the accuracy was able to be increased to around 60%. The optimal values for these hyper-parameters were found to be 10 and .01 for C and gamma respectively. Although this model was only run on a small subset of the data, the accuracy results were much lower than other models, and it is likely that SVM just isn't a good fit for this dataset.

Neural Networks:

Unlike the previous classifiers, the neural networks used for the EMNIST dataset were implemented in TensorFlow (mostly for the sake of learning the commonly-used framework). Each neural network was trained with several different neuron topologies and the topologies with the highest accuracy are included in the Results table above. The neural networks have the best accuracy on the EMNIST dataset of the classifiers tested, with the single layer network achieving 81.8% accuracy, the two-layer network achieving 82.8% accuracy, and the convolutional network achieving 88.5% accuracy. It should be noted that the two-layer network uses twice as many hidden-layer neurons as the single layer network, and takes about 50% more time to train, but only achieves a slight increase in accuracy. Furthermore, the convolutional neural network requires over ten times more training time than the single layer and 40 times more classification time. All this extra time results in about a 7% increase in accuracy. These results highlight a central concern when utilizing neural networks. Much of the work involved in using neural networks involves optimizing hyperparameters and finding an appropriate balance between training time and performance.

Section 3 SPAM Analysis:

The naive bayes classifier performed well with this dataset, producing one of the quickest time to fit and time to predict of all of the models being tested and had a 93% accuracy. The fact that it achieved a relatively high accuracy in less than a couple seconds lends credence to why this model is often used in text classification problems. To keep this project as a strict comparison of models, a gaussian naive bayes model was used. However, in most text classification problems, a multivariate bernoulli or multinomial naive bayes model is recommended. On one hand, if the feature set is defined by either 1's or 0's, where a 1 corresponds to a word being in the dictionary, and a 0 corresponds to a word not being in the dictionary, the multivariate bernoulli is

Spam Dataset Results

Model	Accuracy	Time to Fit	Time to Predict
Naive Bayes	0.9309	0.2372	1.4516
K-Nearest Neighbor	0.9424	0.3782	24.834
SVM (linear)	0.9659	6.5115	21.812
SVM (RBF)	0.8661	4.0837	16.254
Neural Net (1 Layer of 800 neurons)	0.9847	191.86	0.1746
Neural Net (2 Layers of 800 neurons)	0.9608	204.46	0.1746

Number of samples: 5574 | Number of words in dictionary: 11012

used. On the other hand, if the features are defined by the TF-IDF of the samples, as is the case with this dataset, a multinomial model is preferred. (Raschka, 2014) Changing the feature selection around and using one of these two models could be helpful in increasing the accuracy.

KNN performed slightly better, at 95% accuracy, but has the cost of a much higher time to predict. In this case, the extra time is trivial (20 seconds vs. 1 second) because of the small size of the dataset. The RBF variant of SVM had the lowest accuracy (87%) of the tested models and a much slower time to fit and time to predict than NB. In contrast, the linear variant of SVM performed with the highest accuracy of the SVM models, correctly classifying 97% of the testing data. However, when comparing this to naive bayes, KNN, and the other SVM models, it had the slowest time to fit and it's time to predict was in the middle of NB and KNN. The SVM linear SVC model improved these times while keeping the accuracy the same, which made this the fastest model and the most accurate non-neural network model that was tested.

The two neural network approaches performed with the best accuracy out of all of the models, but with the highest time to fit for this dataset, by far. As stated before, these networks were developed in tensorflow and the bag of words implementation was fed into the neural networks as input data. Again, the one layer network performed with a faster time to fit, and actually achieved the similar accuracy as the two layer network after both were run for about 50 epoques.

CONCLUSION:

A crucial recurring theme in this paper that is important to keep in mind for the conclusion, is the "No Free Lunch Theorem." There is no best model when solely looking at the results of each of them in a vacuum, the context of what the goal of the user is as well as their restrictions whether it be time, processing power, memory resources, flexibility, or any other number of impeding factors. There are pros and cons for each model and that will continue to be true despite the conception of new models that rival the four detailed in this report. The comparisons above have shed some light on what models are suited for certain applications and will aid in the decision of model for many analysis tasks to come.

Bibliography

Alex Krizhevsky, I. S. (n.d.). ImageNet Classification with Deep Convolutional Neural Networks.

Almeida, T. A. (n.d.). *SMS Spam Collection Data Set.* Retrieved from UCI Machine Learning Repository: https://archive.ics.uci.edu/ml/datasets/SMS+Spam+Collection

Bentley, J. L. (1975). Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 509-517.

Chapelle, O. (2006). *Training a Support Vector Machine in the Primal.*

Diederik P. Kingma, J. B. (2017). *Adam: A Method for Stochastic Optimization.* San Diego.

Dingankar, A. T. (1999). The Unreasonable Effectiveness of Neural Network Approximation. *IEEE Transactions on Automatic Control*, 2043-2044.

Domonkos Tikk, L. T. (2001). Universal approximation and its limits in soft computing techniques. An Overview. *Int. J. of Approx. Reasoning*, 2--0.

G. Cohen, S. A. (n.d.). *EMNIST: an extension of MNIST to handwritten letters.*

Gunn, S. R. (1998). *Support Vector Machines for Classification and Regression.* University of Southampton.

I.Rish. (n.d.). *An Empirical study of the naive Bayes classifier.*

Karpathy, A. (2015, May 21). *The Unreasonable Effectiveness of Recurrent Neural Networks.* Retrieved from Andrej Karpathy Blog: http://karpathy.github.io/2015/05/21/rnn-effectiveness/

Kroger, O. (2016, January 28). *Tensorflow, MNIST and your own handwritten digits.* Retrieved from medium: https://medium.com/@o.kroeger/tensorflow-mnist-and-your-own-handwritten-digits-4d1cd32bbab4

NG, A. (n.d.). *Improving Deep Neural Networks: Hyperparameter tuning, Regularization, and Optimization*. Retrieved from coursera: https://www.coursera.org/learn/deep-neural-network/lecture/BhJlm/rmsprop

NVIDIA. (2015). *GPU-Based Deep Learning Inference: A Performance and Power Analysis.*

Pedregosa F., V. G. (n.d.). *GridSearchCV.* Retrieved from scikit-learn: http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

Raschka, S. (2014, October 4). *Naive Bayes and Text Classification.* Retrieved from sebastion raschka: http://sebastianraschka.com/Articles/2014_naive_bayes_1.html

Rich Caruana, A. N.-M. (n.d.). *An Empirical Comparison of Supervised Learning Algorithms.* Ithica: Cornell University.

Shaikh, J. (2017, July 23). *Machine Learning, NLP: Text Classification using scikit-learn, python and NLTK.* Retrieved from Towards Data Science: https://towardsdatascience.com/machine-learning-nlp-text-classification-using-scikit-learn-python-and-nltk-c52b92a7c73a

University of Wisonsin-Madison. (n.d.). *The Radial Basis Function Kernel.* Retrieved from wisc.edu: http://pages.cs.wisc.edu/~matthewb/pages/notes/pdf/svms/RBFKernel.pdf