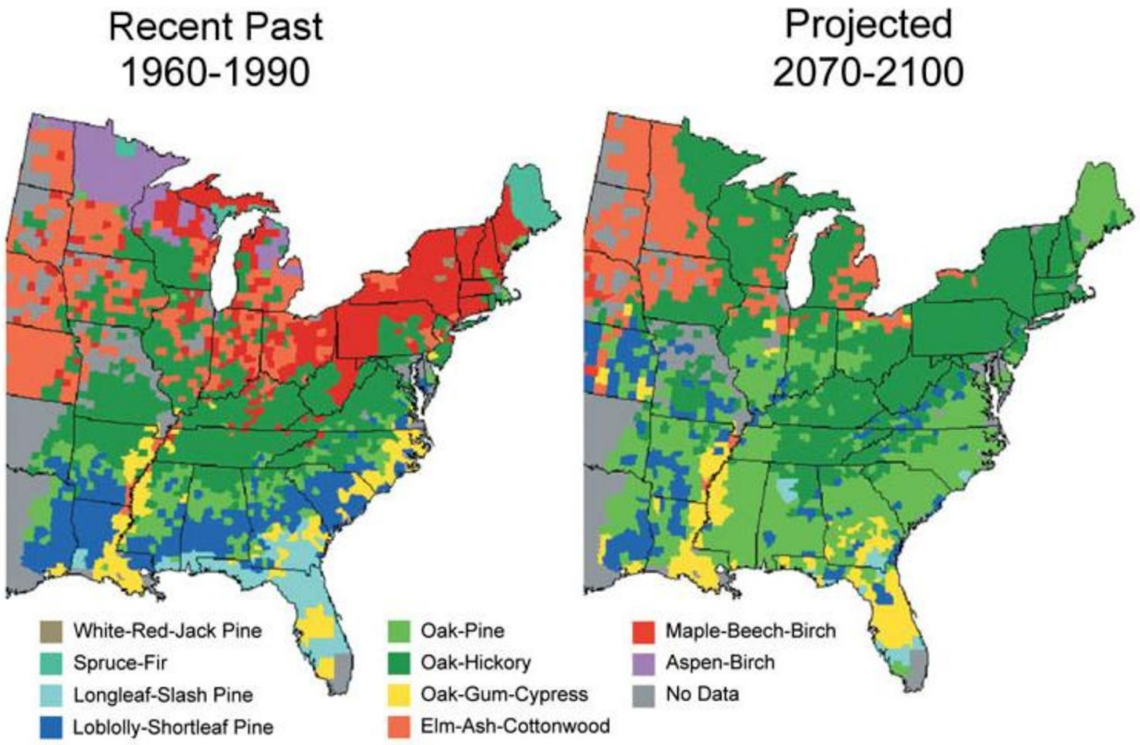
Project Report for Machine Learning

Mesinovic Munib, Iiyoshi Ken, Tauseef Mahrukh

May 6th, 2019

I. Introduction

With rising sea levels, uncontrollable wildfires, and melting ice caps and higher temperatures, we are regularly reminded of climate change's adverse effects on our environment and lives. One of the foundations of our environment are the forest ecosystems that provide us with the source of oxygen, drain of carbon-dioxide, sanctuary for life and water, and guardians against land erosion and drought. It is important to appreciate these cornerstones of our survival on this planet and to develop tools to make it easier to spot when they are in danger. This project attempts to do just that by using satellite spectral imaging to identify different forest ecosystems and thereby note down their change over time. This becomes increasingly important when we take into account that more arid environments have a higher proportions of cactus-like ecosystems, and more humid have more tropic forestry. Identifying the progression of each forest ecosystem in a specific region helps us understand what environmental factors are more prevalent and therefore how the environment in the region changed. A great example of this method is in the figure below:



Projected shifts in forest types. The maps show recent and projected forest types. Major changes are projected for many regions. For example, in the Northeast, under a lower emissions scenario, the currently dominant maple-beech-birch forest type (red shading) is projected to be displaced by the oak-hickory forest type in a warmer future. Source: USGCRP (2009)

Figure 1. Changing forest ecosystems with a more extreme environment in the American Northeast (https://19january2017snapshot.epa.gov/climate-impacts/climate-impacts-forests_.html).

Just some of the factors certain forest changes over time can tell us more about are: CO2 emissions, drought, temperature, erosion, and pests. Identifying troubled regions helps us understand climate change better and focus our resources and attention on high-risk ecosystems.

As for the source of the data, as mentioned previously, it is the spectral data of an image taken over forested area from a satellite. The data was obtained from UCI's database: https://archive.ics.uci.edu/ml/datasets/Forest+type+mapping.

As one can tell, it consists of 526 examples of a forest imaged spectral data, where the number of features is 27 (corresponding to the different spectral densities and frequencies).

Infrared spectral images obtained through remote sensing can be classified into 4 unique forests: Sugi ('s'), Hinoki ('h'), Mixed deciduous ('d'), and Other ('o'). This is vital in the preservation of forests and dense ecosystems in order to mitigate consequences of climate change on the living world.

The data file is in .csv format and is already divided into 198 training and 325 testing data, each with 27 attributes. Attributes b1 to b9 are ASTER image bands of green, red, and near IR wavelengths from three different recording dates. The remaining two sets of 9 attributes are the predicted spectral values minus actual spectral values for the 's' and 'h' classes respectively. The predicted values were generated from spatial interpolation.

We have applied classification machine learning algorithms on this sensory data in the hopes of finding the best way to utilize these powerful algorithms and to detect ecosystems in need.

II. Methods

A. Logistic Regression

Logistic Regression was chosen to perform a multiclass classification on the data. The sklearn.linear_model module was used to create Logistic Regression models. The module has many solvers of which the following were selected for this project: (saga, lbfgs and newton-cg). Saga was chosen because it is the only solver that allows Lasso Regularization in the case of multiclass classification. Lbfgs and Newton-cg operate well on small sized data and allow Ridge Regularization. A polynomial feature scaling of degree 2 was also performed with solvers lbfgs (penalty l2) and saga (penalty l1 and l2).

Since regularization is being performed, it expects preprocessed zero-centered data. The best way to pick the correct preprocessing method for the data is to compare them with one another and choose the one that gives the highest F score when each model is created. Three preprocessing methods were chosen from the sklearn.preprocessing module: scale, StandardScaler and normalize. Normalize turns the data into unit norm (vector length). Both scale and StandardScaler turn the data into zero-centered with unit variance, the scale function does not change the shape of the data whereas the StandardScaler class turns the output it into a Gaussian[1].

Two different types of data splits were used to run the model: the original split and customized split. The original split is the form in which the training and test data were provided by the source[2]. The train: test ratio of the data was 38:62. We split the data to get train:test ratio 70:30 and run the models for both the splits.

B. SVM - Support Vector Machines

It is difficult to discern if the data is linearly separable, let alone visualize such possibility. Although PCA and K-Means Clustering can be used to eliminate some of the features, as noted in the data's source paper, all 27 features contribute significantly to predictive models[3]. This leads to testing methods for both linearly separable and

[1] https://towardsdatascience.com/scale-standardize-or-normalize-with-scikit-learn-6ccc7d176a02

[2] http://archive.ics.uci.edu/ml/datasets/Forest+type+mapping#

[3] Johnson, B., Tateishi, R., Xie, Z., 2012. Using geographically-weighted variables for image classification. Remote Sensing Letters, 3 (6), 491-499.

nonseparable assumptions. Linear kernel is used under the separable assumption. Both the radial basis function (RBF) and polynomial transformation method will be implemented to identify the better-performing non-separable method.

All three methods will be implemented using sklearn.svm.SVC() function, which uses the "one-against-one" approach[4]. Here, $\binom{K}{2}$ SVMs are trained for each pair of class, testing samples assigned to a dominating side of the pair. This approach, compared to "one-against-all", gives best results but is usually slow[5]. However, for this project, computation only took at most few seconds for each method. It should be noted that relative to those using logistic regression or neural network, the computation time for model-fitting using SVM is faster. This is due to the its use of closed form solution unlike the use of gradient descent algorithms for the other two classification methods.

Different values of C, the inverse of penalty for the weight function, will be tested through 5-fold cross validation within the sklearn.svm.SVC() function for each method. Within each method, the best performing value will be identified, along with a corresponding F1 Score. Just as with logistic regression, F1 score is used instead of accuracy or precision since it is essentially an average of the two metrics and the given data does not yield a balanced class. In cases where the performance saturates with higher C values, the first C that reaches saturated value will be picked.

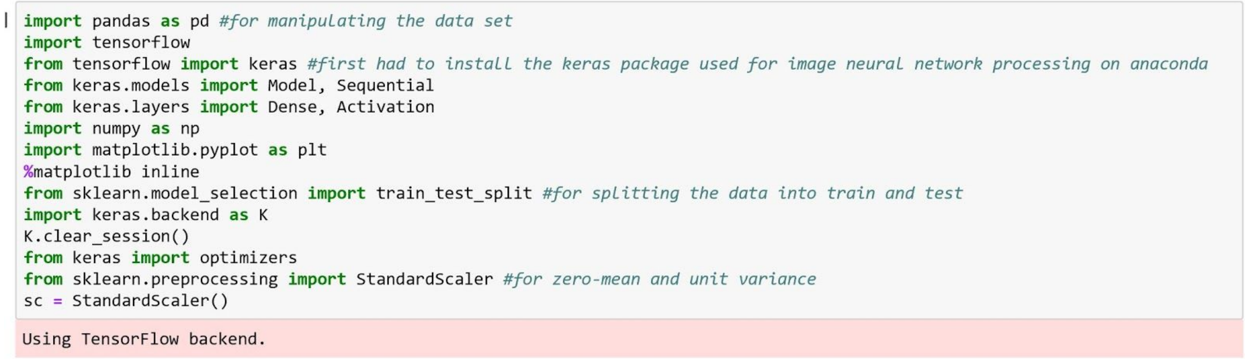
Both the default and custom data split will be applied to the models, similar to what has been done for logistic regression.

[4] https://link.springer.com/chapter/10.1007/978-3-642-76153-9_5
[5] Sellie, Linda. Lecture 7 Support Vector Machines. Power Point. 6 May 2019.

C. Neural Network

We will be using a normal neural network to try to classify the spectral data into the 4 forest classes. Since this is, therefore, a multi-class output of a neural network, we will have to resort to some libraries and functions we learned independently from lecture. A lot, of course, still remains in common.

The libraries used were:



```
import pandas as pd #for manipulating the data set
import tensorflow
from tensorflow import keras #first had to install the keras package used for image neural network processing on anaconda
from keras.models import Model, Sequential
from keras.layers import Dense, Activation
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn.model_selection import train_test_split #for splitting the data into train and test
import keras.backend as K
K.clear_session()
from keras import optimizers
from sklearn.preprocessing import StandardScaler #for zero-mean and unit variance
sc = StandardScaler()
```

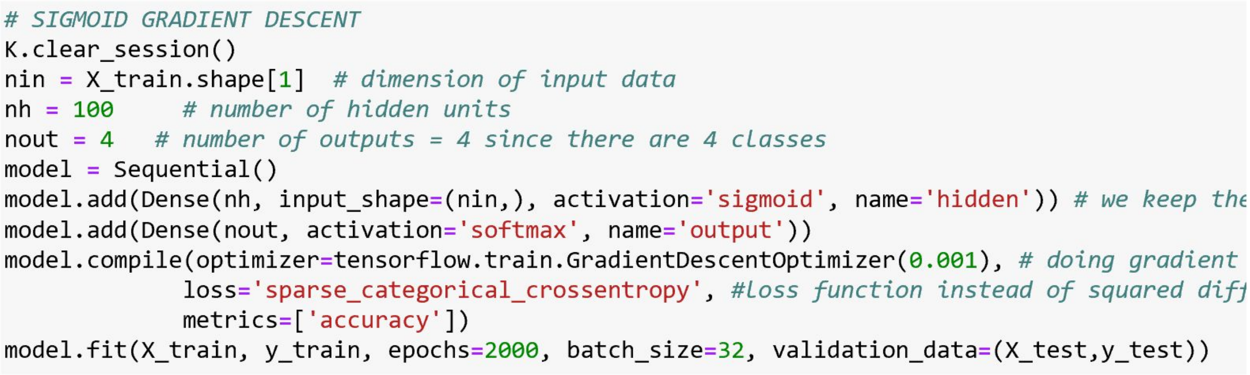
```
Using TensorFlow backend.
```

Figure . Libraries and modules needed for simple neural processing of the multi-class

The pandas, numpy, matplotlib, and sklearn modules we used before so they do not necessitate any more explanation and description. Tensorflow is the foremost used library for neural network processing, significantly due to a module called keras which contains the functions, metrics, and attributes needed for not just forward and backward propagation but a lot more, even with convolutional neural networks. Model inside keras is what it stands for, it contains the model being used, as described by its metrics, activation functions, hidden and output layers, optimization algorithm etc. The layers module contains the LeakyRelu function that will be used later as the activation function.

The clear.session() makes sure that when we try out different activation functions or optimization algorithms, they do not computationally interfere with the next one. The scaling of data here was similar to the two above methods, i.e. we utilized the Standard Scaler which zero-centers and unit-varies the data.

The first step was to use the normal neural network learned in class, which had gradient descent as its optimization algorithm, and sigmoid for its activation function.



```
# SIGMOID GRADIENT DESCENT
K.clear_session()
nin = X_train.shape[1]  # dimension of input data
nh = 100     # number of hidden units
nout = 4   # number of outputs = 4 since there are 4 classes
model = Sequential()
model.add(Dense(nh, input_shape=(nin,), activation='sigmoid', name='hidden')) # we keep the
model.add(Dense(nout, activation='softmax', name='output'))
model.compile(optimizer=tensorflow.train.GradientDescentOptimizer(0.001), # doing gradient
             loss='sparse_categorical_crossentropy', #loss function instead of squared diff
             metrics=['accuracy'])
model.fit(X_train, y_train, epochs=2000, batch_size=32, validation_data=(X_test,y_test))
```

Figure . Neural Network code for gradient descent sigmoid activation and multi-class classification

Here we used 100 hidden neurons, and we had to keep in mind that using a small number of hidden neurons would decrease the accuracy of our model while increasing the number of hidden neurons would increase the computational cost. We made an educated guess about using this number after trying out some values. The number of output neurons is 4 because there are 4 classes to classify the data in. The sequential() just sets up the module to run the forward and the backward propagation, i.e. it sets up the linear stack of layers. The activation function for the hidden layer neurons is the sigmoid, but for the pre-output layer neurons it is softmax. Now, softmax had to be used because we will be assigning probabilities to the data belonging to each of the classes, and therefore the output is now a vector of probabilities (normalized with respect to their sum) instead of just one continuous value predicted value as before[6]. The optimizer here used is the Gradient Descent which just means that the way we update the parameters is with a constant learning rate of 0.001 (not too big - skip minima or too small - long to compute) and using the loss-function derivative evaluated at the updated weight and data values. In the code, we tried out another optimizer, Adam, commonly used in multi-class neural network processing, but which uses a varying learning rate, useful for when the gradient starts to plateau[7]. For the loss function, instead of using our mean-squared-difference, we resorted to sparse categorical cross entropy which is simply used for multi-class problems such as this. As far as we could understand, we cannot use our original loss function as well because now the output will be mutually exclusive classes and not just one number to subtract from the true value. Therefore, encoding has to be done to compare the different ways the output could be classified to the actual true class it belongs. Now, sparse skips one-hot encoding and keeps the integer output for comparison[8]. The metric we use for evaluating the models here will be accuracy, as compared to the F1 score above. This is due to the removal of the F1 metric from keras about 2 years ago, and upon attempts to try to implement our own metric with custom F1 function, we could not obtain a working algorithm. Accuracy, as the ratio of correct classifications over total examples, is an intuitive and trustworthy evaluation metric when we have no reason to believe the data are unbalanced between precision and recall. The number of epochs tells us how many times to we want to run the update of the weights, the variable we usually call num_iter, and we made it 2000. Doing more we observed had a significant toll on the computational

[6] https://developers.google.com/machine-learning/crash-course/multi-class-neural-networks/softmax
[7] https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/
[8] https://www.dlology.com/blog/how-to-use-keras-sparse_categorical_crossentropy/

speed. The batch_size is used because we are using batch gradient descent instead of stochastic gradient descent, basically meaning that we will use 32 data samples to run each update instead of 1 like with stochastic. We also tried halving the batch size to see if the accuracy improves because there would be more updates, stay tuned to see how it turned out! As for validation data, we used the test data split in the beginning according firstly to the provided ratio in the data set which was 32:68 (train vs. test), and then using our own custom ratio 70:30 (train vs. test).

A comment on the activation functions used is needed. We used the sigmoid, Relu, and LeakyRelu to see how, if at all, the accuracy and behavior of the neural network changes. We also tried using a different optimizer, Adam, with the sigmoid to notice interesting behavior.

III. Results

When the default data split was used (train:test ratio 32:68), the maximum F score for test set was obtained for logistic regression with saga solver and l1 regularization with polynomial feature training of degree 2 (see Table 1 below). When the custom data split was used (train:test 70:30), the maximum F1 score for test set was higher than that obtained with default data split. It was almost the same as the maximum F1 score obtained using linear SVM Kernel.

Table 1. Model Performance of Logistic Regression

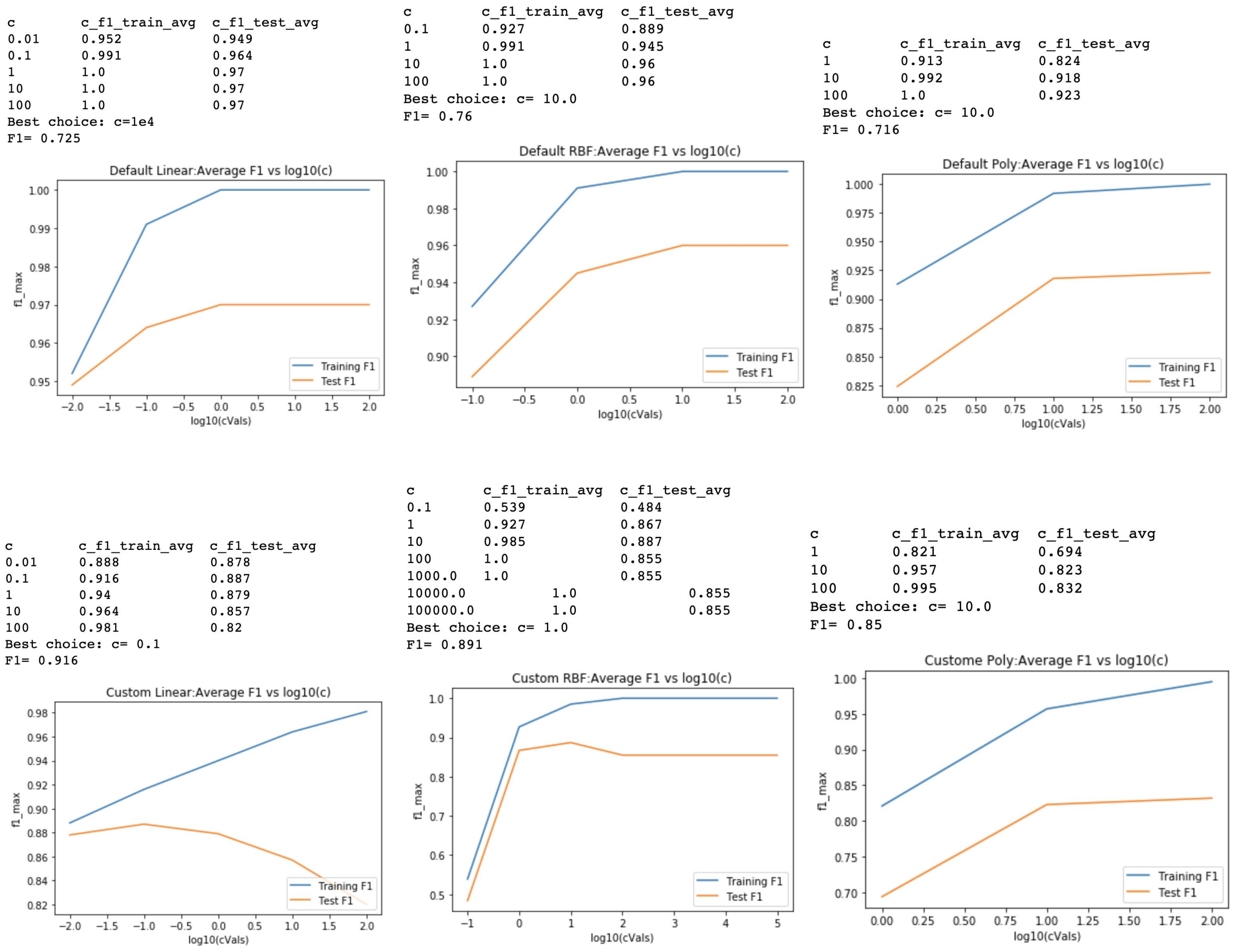
Data Split used	Solver/Penalty	Preprocessing	Max F1 Score (test data)	F1 Score (train data)
Default	saga / l1	Standard Scalar + Polynomial Features (degree 2)	0.850283	1.0
Custom	saga /l1	Scale + Polynomial Features (degree 2)	0.911363	0.951253

However, Logistic regression is not an efficient method for this data size since the most accurate results were obtained using the solver saga with l1 norm.

The processing time for this model was very high. This solver might have worked more efficiently for a bigger set of data[9].

As seen in the figure below, the best F1 score was obtained by fitting the custom split data through linear SVM kernel.

[9] https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html



```
c         c_f1_train_avg  c_f1_test_avg
0.01      0.952           0.949
0.1       0.991           0.964
1         1.0             0.97
10        1.0             0.97
100       1.0             0.97
Best choice: c=1e4
F1= 0.725
```

```
c          c_f1_train_avg  c_f1_test_avg
0.1        0.927           0.889
1          0.991           0.945
10         1.0             0.96
100        1.0             0.96
Best choice: c= 10.0
F1= 0.76
```

```
c          c_f1_train_avg  c_f1_test_avg
1          0.913           0.824
10         0.992           0.918
100        1.0             0.923
Best choice: c= 10.0
F1= 0.716
```

```
c         c_f1_train_avg  c_f1_test_avg
0.01      0.888           0.878
0.1       0.916           0.887
1         0.94            0.879
10        0.964           0.857
100       0.981           0.82
Best choice: c= 0.1
F1= 0.916
```

```
c          c_f1_train_avg  c_f1_test_avg
0.1        0.539           0.484
1          0.927           0.867
10         0.985           0.887
100        1.0             0.855
1000.0     1.0             0.855
10000.0            1.0             0.855
100000.0           1.0             0.855
Best choice: c= 1.0
F1= 0.891
```

```
c          c_f1_train_avg  c_f1_test_avg
1          0.821           0.694
10         0.957           0.823
100        0.995           0.832
Best choice: c= 10.0
F1= 0.85
```

Default Linear:Average F1 vs log10(c)
Default RBF:Average F1 vs log10(c)
Default Poly:Average F1 vs log10(c)
Custom Linear:Average F1 vs log10(c)
Custom RBF:Average F1 vs log10(c)
Custome Poly:Average F1 vs log10(c)
f1_max
log10(cVals)
Training F1
Test F1

Figure 2 . Average F1 Score of various Cs for Linear, RBF, and Poly Kernels for Default & Custom Split

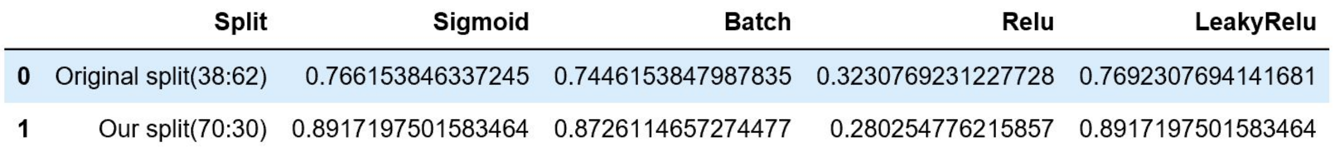
Table 2 . Maximum F1 Score for Linear, RBF, and Poly Kernels

Split	Linear	RBF	Poly
Original (38:62)	0.725	0.76	0.716
Custom (70:30)	0.916	0.891	0.85

The F1 score for all three kernel types were higher for custom data split, in which linear type yielded the highest score of 0.916. This is expected since the model is trained under larger data set. It is interesting note however, that RBF performed the best for original data split. As future extension, additional kernel types such as 'sigmoid' and 'precomputed' can be tested as well as varying the gamma parameter for 'rbf', 'poly', and 'sigmoid' mode.

As for neural networks, the test accuracy results can be seen in the figure below:

Table 3 . Model prediction test accuracy where the 32 and 16 indicate batch_sizes

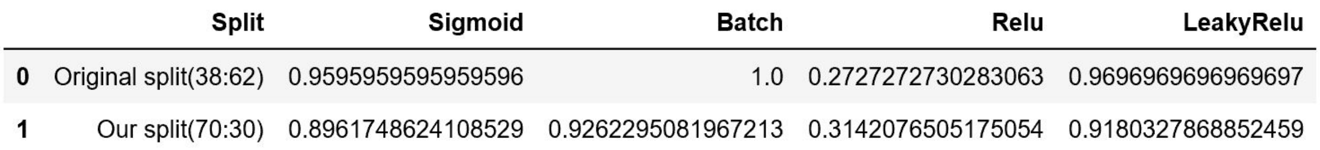


	Split	Sigmoid	Batch	Relu	LeakyRelu
0	Original split(38:62)	0.766153846337245	0.7446153847987835	0.3230769231227728	0.7692307694141681
1	Our split(70:30)	0.8917197501583464	0.8726114657274477	0.280254776215857	0.8917197501583464

32 16

Let us see the training accuracy as well:

Table 4. Model prediction training accuracy



	Split	Sigmoid	Batch	Relu	LeakyRelu
0	Original split(38:62)	0.9595959595959596	1.0	0.2727272730283063	0.9696969696969697
1	Our split(70:30)	0.8961748624108529	0.9262295081967213	0.3142076505175054	0.9180327868852459

As we can see, the split we inserted into the data performed much better than the one the researchers used. This is probably due to the neural network not having enough training data to learn which usually leads to shortages in neural network reliability due to lack of generalization[10]. Secondly, halving the batch size from 32 to 16 slightly decreased the accuracy thereby probably not playing a significant part in this model. If the accuracy fell by a larger number we could say that a smaller batch size would lead to a lower generalization error due to introduction of noise but also means the approximation of the gradient is worse because we are using less points to approximate it. When we tried using the Relu activation function, we noticed a sharp drop in accuracy, probably due to neurons dying. The cause of it is the nature of the Relu function to be flat for all negative values at 0, and so if the learning rate sometimes happens to be a bit too large, the updated weights will become negative and so the activation will become and stay 0 for those neurons. This drags the accuracy down because those neurons remain at 0 and do not get

[10] https://beamandrew.github.io/deeplearning/2017/06/04/deep_learning_works.html

updated. We tried playing around with the learning rate and saw simple accuracy change but still staying at low values. To deal with the Relu problem, we also used LEakyRelu as an activation function which as compared to Relu introduces a positive slope for values less than 0 to prevent neuron death. We can see a significant impact with a very high LeakyRelu accuracy, and comparable to the sigmoid accuracy. The highest accuracy occured with the sigmoid activation function and gradient descent optimization algorithm.

IV. Conclusion

What we can conclude is that neural networks, especially for sigmoid and LeakyRelu definitely improve upon the accuracy used in the 2011 paper with SVMs. We hope that in the future, this technique and other deep learning algorithms can be utilized to help deal with climate change and contribute to the survival of us all.