Kaggle Data Challenge 1

Name: Kaushik Moudgalya
Student Number: 20176760
Kaggle Username: diraizel
Kaggle Display Name: Banana_Leopard
Kaggle Account Link: https://www.kaggle.com/diraizel

Introduction – Problem Description

The objective of this challenge was to classify events as either standard background conditions / tropical cyclones or atmospheric rivers based on set of climate features for a given location at a given time. This dataset is a subset of the ClimateNet dataset.

Introduction – Summary of approach and results

Our approach tried a plethora of learning algorithms, some feature engineering and data sampling methods and tricks. The final submissions were based on:

1) Removing the duplicates from the data, relabeling features which had 2 different labels using K-Nearest Neighbors, splitting the year column into different features for year, month and date, under-sampling using RandomUnderSampler and over-sampling the resulting data using SMOTE (Synthetic Minority Oversampling Technique), and making predictions using a Random Forest model selected using HalvingRandomSearchCV.
2) The second approach is the same as the above, but instead of using SMOTE to over-resample after under-sampling, we use SMOTE Tomek. This approach does not handle the duplicates in the data or the data that has two different labels, since it is expected that SMOTE Tomek will take care of those samples.

Feature Design

The features tried along with the models that were tried with are mentioned below:

- Duplicates in the data were dropped for SMOTE and retained for SMOTE Tomek. Duplicates won't necessarily help the models learn any better, following the adage of "garbage in, garbage out".
- There are certain samples in the data which have the same feature values but different labels. These were addressed by using a K-Nearest Neighbors model on the conflicting samples. The objective was to avoid confusing the model, since the learning due to these conflicting samples would cancel each other out.
- Tried a few models where the **lat** and **lon** columns were converted to x, y, z values instead of being in degrees. The idea was to try having the distance values in addition to the degree values. The **lat** and **lon** columns have comparatively large values in them and converting them to x, y, z involves taking the product of their sin and cos values thereby rendering them more manageable.

- Tried some models where the **lat** and **lon** models were converted to radians. The rationale was to convert the co-ordinates to follow the International System (S.I.) units instead of degrees, since almost all the other features followed the same.
- Tried few models where we used Agglomerative Clustering to cluster the **lat** and **lon** columns. We clustered the **lat** column in 5 clusters and the **lon** column into 7. We tried label encoding these clusters and also one-hot encoding them. The one hot encoding was to prevent the model from assigning arbitrarily high importance to the larger cluster numbers. The label encoding was inspired by the fact that geographically closer regions tend to have the same climate.
- Split the **time** column into year, month and date columns. Since the merged time column would result in a huge number, even when scaled this could result in biases in the trained model.
- Drop the **S.No** column. We already have an index column in pandas, we could use this as the index instead. Numpy arrays won't have an explicit index column anyway.
- We also tried Outlier Removal, retaining data as follows:

$$\text{Relevant Dataset} = [Q1 - 1.5 * IQR, Q3 + 1.5 * IQR] \text{ (Dataset without outliers)}$$
$$\text{where } Q1 = \text{first quartile}, Q3 = \text{third quartile}, IQR = [Q1, Q3] \text{ is the interquartile range.}$$

Equation 1 - IQR Range equation used to remove outliers

Algorithms

We tried the Logistic Regression algorithm and then threw the entire kitchen sink of algorithms at the problem. The results including code, hyperparameters and metrics for each model can be found on this GitHub page. The algorithms tried were: XGB (eXtreme Gradient Boosting) Classifier, Support Vector Classifiers (SVC), Gradient Boosting Classifiers, Random Forest Classifiers, Multilayer Perceptron Classifiers, Light Gradient Boosting Machines (LGBM), K-Nearest Neighbors Classifiers, Decision Trees Classifiers, Gaussian Naïve Bayes and Multinomial Naive Bayes and also stacking aforementioned classifiers.

- We cannot know for sure which model will work well in advance, though for the given problem, based on pre-existing intuition it seemed like XGB or Random Forests would do the trick.
- We had a lot of hope in Bayesian methods but these weren't explored in depth due to unpromising preliminary results. We did try them because theoretically, Bayesian classifiers can have the highest accuracy possible on a given dataset, so much so that the irreducible error is also called the Bayes error.
- Since the original ClimateNet models use Neural Networks for segmentation, there is precedent for Neural Networks working well on this kind of dataset, which led to the MLP Classifier trials.
- The other classifiers were trained mostly to be used within the Stacking Classifier. The intuition behind stacking is that different classifiers might do well on different parts of the dataset, and averaging the probabilities and training a final classifier on their predictions might do better than any individual classifier.

Methodology

Data strategies

The classes were very imbalanced. We tried a number of techniques to handle this imbalance:

- Firstly, remove all the duplicates from the dataset.
- Second, remove the values from the dataset where the same features have different labels.

- Use K-NN on the dataset without any duplicates and predict the label on the part of the dataset with conflicting values.
- Third, use the Random Under Sampler from the imb learn library to under-sample the majority class. According to the imb learn documentation [3], under-sampling combined with oversampling works better than over-sampling alone. The over-sampling was done in such a way that the number of examples aren't exactly equal to avoid overfitting. A limited number of examples will only be able to produce a limited number of high-quality new examples.
- We also tried SMOTE, ADASYN, Smote Tomek and Smote ENN without the under-sampling but the performance doesn't match up to that of the approach mentioned above.
- Using pandas we found out the features that were the most correlated to the labels, giving us more of an idea about which features are truly relevant and which features are redundant.

Train / test split strategies

- Since the data is unbalanced, just splitting it randomly into train and test sets will not give us good results. To retain the same proportion of examples in the test set as in the train set, we shuffle the dataframe and then extract around 15% of the samples of each label for the test set, in the same proportion that they are present in the train set.

Standardization strategies

- There is no guarantee that all the features in the data are normally distributed. We considered some approaches where we would additively add features to a model, standardizing or normalizing (min-max) each feature as required, based on the criteria of higher balanced accuracy or ROC AUC (Receiver Operating Characteristic – Area Under the Curve).
- For simplicity, we decided to standardize all the features. However, for Gaussian NB and Multinomial NB, we used min-max normalization.

Optimization strategies

Logistic regression:

- We add the bias to the weight matrix itself, using numpy hstack and adding a column of 1s.
- We use one-hot encoding on the labels.
- We have a different weight matrix for each feature (one vs. all), and we compute the final prediction by comparing the value of the prediction function for each class and predicting the class with the highest predicted value.

$$f(W^T . X + b), \text{ where } f \text{ is the sigmoid function.}$$

Equation 2 - Logistic Regression prediction function

Other algorithms:

- We trained all the other algorithms on two different kinds of data, one with the synthetic data generated using the methods outlined above, and one without synthetic data. We also tried training our models on data without the aforementioned duplicates being removed and after the duplicates were removed.

Choice of hyperparameters

- For logistic regression, we manually tried different hyperparameters which are shown in the appendix below.
- For all the other algorithms, we used HalvingRandomSearchCV from the sklearn library. This is a relatively new release and it still is an experimental feature. It is essentially a Grid Search technique which is faster than both GridSearchCV and RandomizedSearchCV.

- Wherever possible we try to use hyperparameters that account for imbalanced classes.
- We also played with the prediction class probability cutoff threshold in addition to the learning rate and number of iterations to obtain the best model.
- We experimented with both learning rate decay after a certain number of iterations and also the factor by which the learning rate decays.

Results

We'll compare the results across various hyperparameters for our logistic regression classifier:

Table 1 - Comparison across differently processed datasets:

Algorithm	Dataset	Learning Rate	Iterations	Threshold	Accuracy
LogReg	**D1**	**0.001**	**40000**	**0.5**	**81.09%**
LogReg	D2	0.001	40000	0.5	78.25%
LogReg	D3	0.001	40000	0.5	79.77%

D1 = Standardized data with year handling.

D2 = Normalized data with year handling.

D3 = Combination of normalized and standardized features with year handling.

The curves associated with these are discussed in the appendix.

Table 2 - Comparison across different hyperparameters:

Algorithm	Dataset	Learning Rate	Iterations	Threshold	Accuracy
LogReg	D1	0.001	100000	0.5	81.81%
LogReg	**D1**	**0.001**	**150000**	**0.5**	**82.18%**
LogReg	D1	0.001	150000	0.4	**82.76%**

From the table above, increasing the number of iterations does seem to increase the accuracy. However, the time for more iterations to complete seems to increase non-linearly. We decided to explore other avenues of improvement, such as tweaking the threshold (see third row above). Increasing the number of iterations does not seem to provide as significant an increase in accuracy as changing the threshold.

Table 3 - Comparison across different thresholds:

Algorithm	Dataset	Learning Rate	Iterations	Threshold	Accuracy
LogReg	D1	0.001	40000	0.2	69.06%
LogReg	**D1**	**0.001**	**40000**	**0.4**	**81.67%**
LogReg	D1	0.001	40000	0.6	79.97%
LogReg	D1	0.001	40000	0.8	78.57%

Based on the observations above, we decided to use a threshold of 0.4.

Other Models

We also tried a bunch of other models as mentioned in the Algorithms section. Check out the GitHub page for details. Following is a sample:

	precision	recall	f1-score	support
0.0	0.98	0.93	0.95	2827
1.0	0.42	0.78	0.54	109
2.0	0.84	0.94	0.89	458
accuracy			0.92	3394
macro avg	0.74	0.88	0.79	3394
weighted avg	0.94	0.92	0.93	3394

Result 1 - Classification Report - Random Forest Classifier 1

	precision	recall	f1-score	support
0.0	0.99	1.00	1.00	2827
1.0	1.00	0.91	0.95	109
2.0	1.00	0.98	0.99	458
accuracy			0.99	3394
macro avg	1.00	0.96	0.98	3394
weighted avg	0.99	0.99	0.99	3394

Result 2 - Classification Report - Random Forest Classifier 2

A bunch of algorithms were trained using the same principle as the example shown above with Random Forest Classifier (RFC1 and RFC2). RFC1 was trained on synthetically generated data using ADASYN to equalize the number of examples in each class which was then standardized and run through a HalvingRandomSearchCV object to find the best parameters. RFC2 is much the same but it does not use synthetic data.

Table 4 - Model summarization table

Algorithm	Dataset	Accuracy
DTs	ADASYN	79%
DTs	D1	97%
MLP	ADASYN	85%
MLP	D1	99%
RF	ADAYN	92%
RF	D1	99%

Algorithm	Dataset	Accuracy
XGB	ADASYN	79%
GaussianNB	D1	57.51%
KNN	ADASYN	98%
KNN	D1	100%
LGBM	ADASYN	42%
LGBM	D1	89%

DT = Decision Trees, MLP = Multilayer Perceptron Classifier, RF = Random Forests, XGB= eXtreme Gradient Boosting, KNN = K-Nearest Neighbors, LGBM = Light Gradient Boosting Machines. All the algorithms used came from the sklearn package. The best hyperparameters for each model are mentioned in the relevant notebooks.

Final submitted models

For the final submissions, we used the Random Forests model on a data processed in a couple of different ways as described in the introduction. We can call these models Model_with_dups and Model_without_dups per the data processing techniques used. The only difference between them is the data processing technique used. Both of them use under-sampling followed by over-sampling, followed by a HalvingRandomSearchCV to search for the best model. Both models were evaluated on a bunch of different metrics as shown below:

```
Accuracy is: 0.8694755450795522
Balanced Accuracy is: 0.8628456595567
F1 score is: [0.91686996 0.69291339 0.69709544]
F1 sample score is: [0.91686996 0.69291339 0.69709544]
Precision is: [0.97641886 0.60689655 0.562249  ]
Recall is: [0.86416696 0.80733945 0.91703057]
ROC AUC Score is: 0.9728533106491081
```

Result 3- Metrics for Model_without_dups

```
Accuracy is: 0.8594674556213018
Balanced Accuracy is: 0.8737531559567747
F1 score is: [0.90653266 0.75144509 0.72272411]
F1 sample score is: [0.90653266 0.75144509 0.72272411]
Precision is: [0.96781116 0.65326633 0.61104583]
Recall is: [0.85255198 0.88435374 0.88435374]
ROC AUC Score is: 0.9604769615620347
```

Result 4 - Metrics for Model_with_dups

Discussion

Pros:

- Our methodology tries a lot of combinations of the data and addresses the imbalances in the data by sometimes generating new data, sometimes retaining them and by using class balanced model parameters.
- Our methodology is great for quick prototyping since HalvingRandomSearchCV can be several times faster than both GridSearchCV and RandomizedSearchCV.
- Our methodology proposes that we exhaustively try models and features until we find the best model and the best features, as outlined above.

Cons:

- Doing normal GridSearchCV would have taken more time but would probably have given us better results. HalvingRandomSearchCV would find locally optimum hyperparameters with a chance of finding the global optimum but with GridSearchCV we can be sure that we found the global optimum.
- Some more experimentation with under and over sampling ratios would have helped us concretize this approach.
- The experiments are all over the place and experiment tracking using trains or Comet ML might have helped steer things faster in the right direction probably with better results stemming from better organization. Furthermore, running the same experiment a number of times to get an idea of the variance in the range of metrics obtained would be even better.
- We could have experimented with even more metrics for imbalanced classes or come up with a metric of our own, since vanilla accuracy is terrible for imbalanced classes.
- Using a training, validation and test set might have been better instead of just using a training and a testing set. Additionally, printing training curves for each model would have given us a better picture of the training process within each model.

Statement of Contributions

I hereby state that all the work presented in this report is that of the author.

References

1. All the references used while coding are mentioned in the code itself.
2. The IFT6390 Kaggle Competition Guide document.
3. https://imbalanced-learn.org/stable/auto_examples/combine/plot_comparison_combine.html#sphx-glr-auto-examples-combine-plot-comparison-combine-py

Appendix

1. GitHub repository with all the code and notebooks: Link.

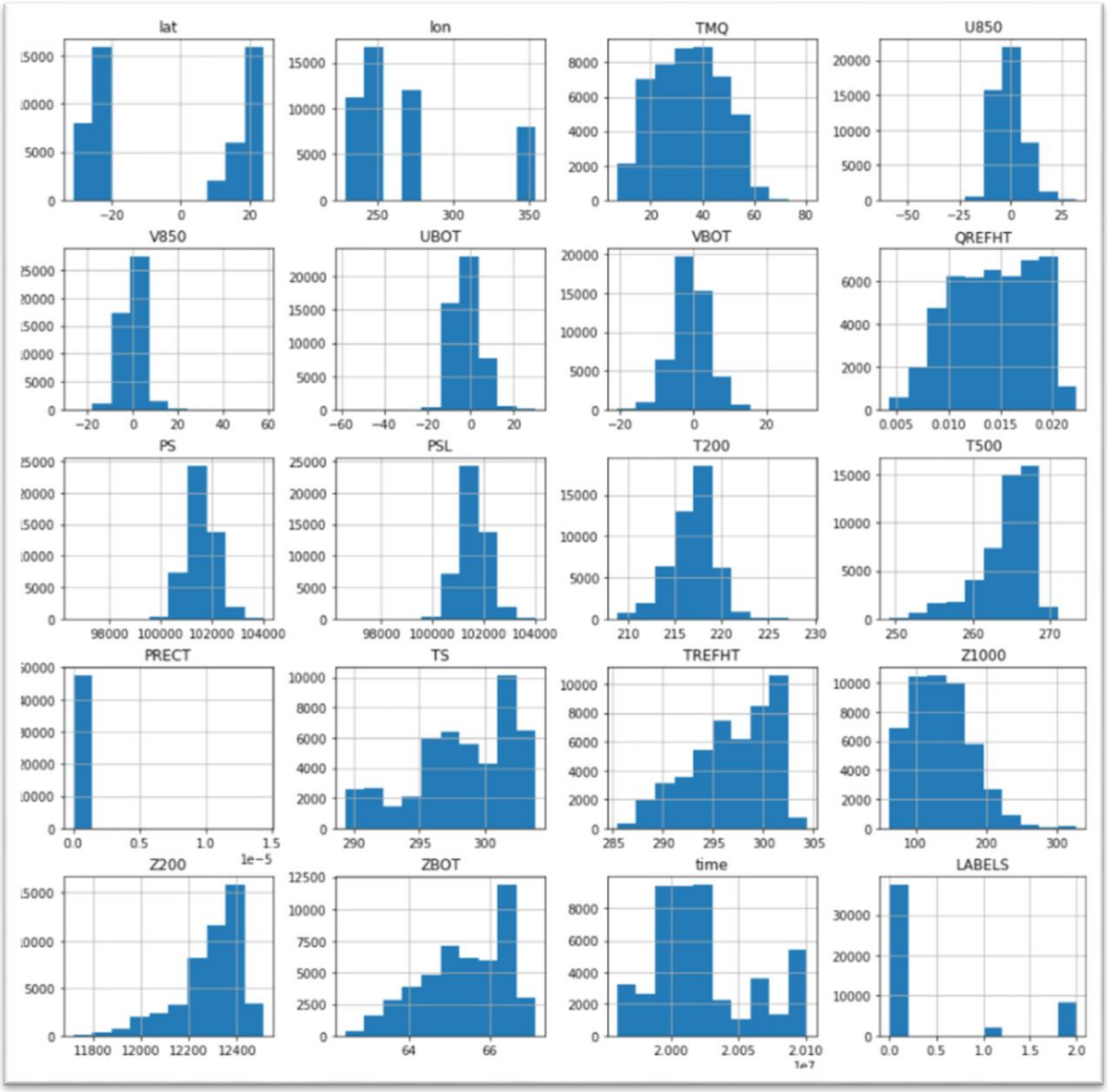

Figure 1 - Distribution histogram for features in the dataset

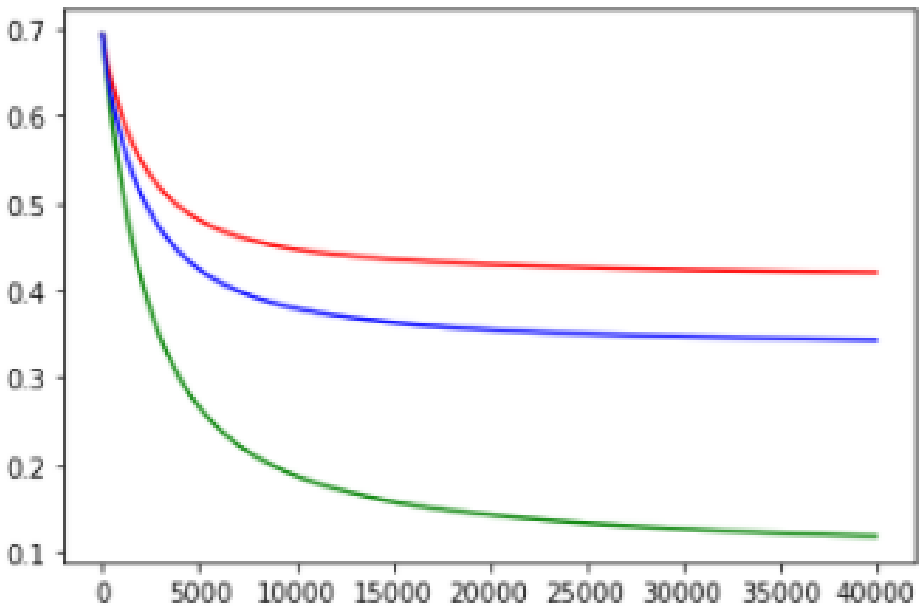

Figure 2 - Training cost curve for each class of the logistic regression classifier on standardized data

(Red = Class 0, Green = Class 1 and Blue = Class 2)

[X axis represents the number of iterations; the Y axis represents the averaged Cross Entropy]

We notice that the training curve is pretty steep before flattening out asymptotically in the above figure for standardized data. The accuracy on our test set is around 81%.

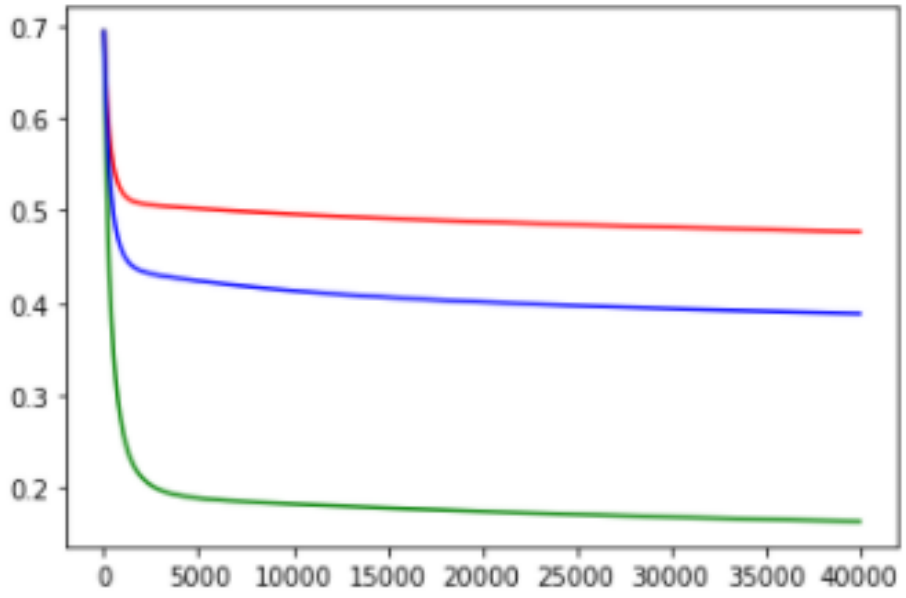

Figure 3 - Training cost curve for each class of the logistic regression classifier on normalized data

(Red = Class 0, Green = Class 1 and Blue = Class 2)

[X axis represents the number of iterations; the Y axis represents the averaged Cross Entropy]

We notice that on normalized data the training curve is even steeper than the normalized training curve. The accuracy on our test set is around 78%, so despite converging faster we obtain perhaps slightly lesser accuracy when we normalize the data.

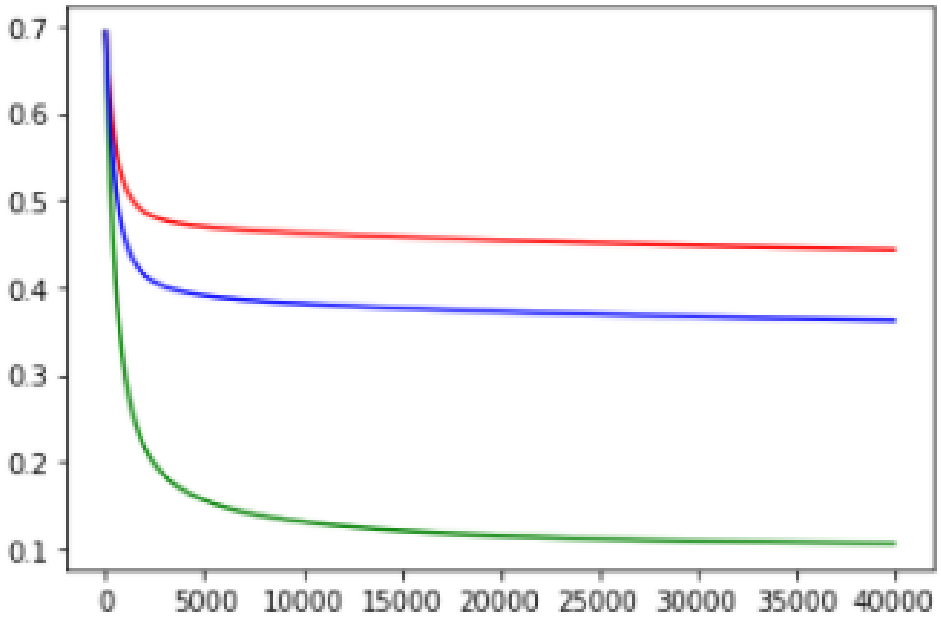

Figure 4 - Training cost curve for each class of the logistic regression classifier with a mix of standardized and normalized features (Red = Class 0, Green = Class 1 and Blue = Class 2).

[X axis represents the number of iterations; the Y axis represents the averaged Cross Entropy]

With some features normalized and others standardized, the training curve looks like a combination of the two curves above which makes intuitive sense. The accuracy on our test set is around 79%.

2. Comparison of different learning decay frequencies and factors on Logistic Regression:

All the models the D1 Dataset mentioned before with a threshold of 0.4. Decay freq refers to decay frequency, i.e. the number of iterations after which the learning rate is changed. Decay factor refers to the number the original learning rate is divided by.

Algorithm	Learning Rate	Iterations	Decay freq	Decay factor	Accuracy
LogReg	0.001	60000	15000	2	81.30%
LogReg	0.001	60000	20000	2	81.55%
LogReg	0.001	60000	25000	2	81.67%
LogReg	0.001	100000	15000	2	81.43%
LogReg	0.001	100000	20000	2	81.64%
LogReg	0.001	100000	25000	2	81.71%

3. Comparison of several different hyperparameters on Logistic Regression:

All models trained on Dataset D1.

Algorithm	Learning Rate	Iterations	Decay freq	Decay factor	Accuracy
LogReg	0.001	150000	50000	2	82.10%
LogReg	0.01	10000	-	-	82.30%
LogReg	**0.1**	**10000**	**-**	**-**	**83.04%**
LogReg	0.2	10000	-	-	83.00%
LogReg	0.1	100000	50000	2	82.92%
LogReg	0.001	200000	-	-	82.76%
LogReg	0.1	100000	50000	4	82.97%
LogReg	0.1	100000	50000	2	82.90%
LogReg	0.1	100000	-	-	82.94%
LogReg	0.1	100000	50000	10	82.94%