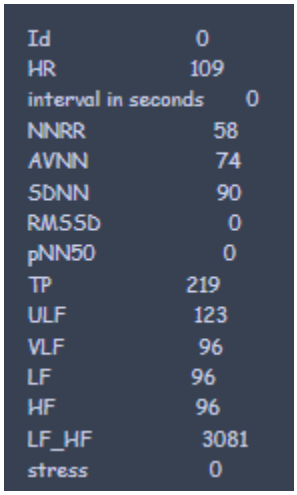
Report
Programming Assignment 2

Blake Downey - 2127448

A 1)

This dataset had lots of missing data and or features that were not informative. Preprocessing that had to be done consisted of removing ‘inf’ values, imputing samples with nan values, removing features that were uninformative, and normalizing. My code first enters into a cleaning function which performs 3 actions: removes columns if nan count is greater than 15% of the samples in the dataset, replaces ‘inf’ values with nan, and then goes through each sample and drops the sample if the nan count for a certain row is greater than 25% of the number of features. This drops the last column ‘LF_HF’ as most of the values are blank. I then manually removed the columns ‘LF’ and ‘HF’ as those columns mainly took on the value 0.

A 2)



Id	0
HR	109
interval in seconds	0
NNRR	58
AVNN	74
SDNN	90
RMSSD	0
pNN50	0
TP	219
ULF	123
VLF	96
LF	96
HF	96
LF_HF	3081
stress	0

Shown on the left is a breakdown (feature-wise) of the number of nan values (missing samples) in the raw, unprocessed dataframe after being read into the Python notebook. As we can see the column ‘LF_HF’ has nearly all nan values. Taking a closer look at the data, the columns ‘LF’ and ‘HF’ have a lot of values equal to 0. The value counts for 0 in each column are both 2985, this is out of 3129 samples. Due to these columns having either mostly nan or 0 values, all three were removed for the assignment cleaning.

As for data imputation methods, I tried 4 different types of imputation, each with their respective f1 scores for validation in the table below. I chose to implement 5 types of missing data handling, 4 imputation methods and one that just removed the samples with nan after data cleaning. The 4 imputation methods were mean, median, and hot deck with each type of distance measure, l1 and l2. I found that each performed around the same as each other, but median performed slightly better with validation as well as the kaggle data. Here I used a Random Forest Classifier to compare the F1 Scores, the hyper parameters remained the same during the varying methods of imputations.

	Mean	Median	Hot Deck - l1	Hot Deck - l2	Remove
F1 Score (Validation Data on RF)	0.606277	0.611947	0.588784	0.593003	0.573776

Prior to generating this table, I was anticipating that Hot Deck imputation, either with l1 or l2, was going to get me the best results. Seeing that median imputation achieves the highest results is actually a bit surprising. I theorize that this happened because it's more aligned with

the distribution of the data instead of being geared towards particular samples in the data like Hot Deck would fall victim to.

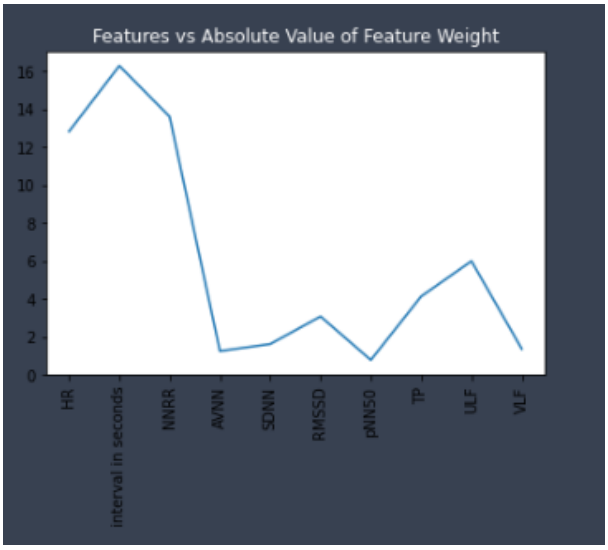
A 3)

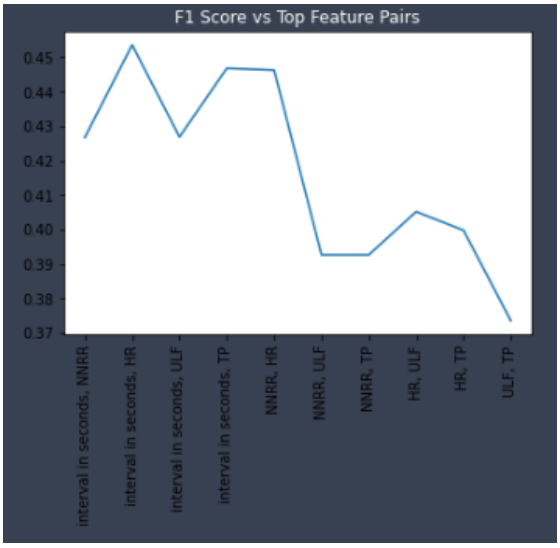
	F1	Accuracy	Precision	Recall
Random Forest	0.6040	0.6196	0.6134	0.6196
K Nearest Neighbors	0.6155	0.6276	0.6211	0.6276
Keras Sequential NN	0.6031	0.6260	0.6216	0.6260

The above results were calculated with a 80/20 train/validation split with median imputation used as the method of missing data imputation. Highlighted in the table is the ML method that gave me the best validation results, K Nearest Neighbors. To achieve optimal results in my KNN model, I ran hyper parameter searches for values p, and n. With this I was able to test thousands of combinations to find the best f1 score on the validation set. I found that when doing this, I was able to achieve comparable results with the kaggle submission.

When testing different data splits, 70/30, 90/10, I was able to fluctuate the results I got for f1 validation score greatly. I was able to achieve my lowest loss training cycle with my sequential NN with a split at 90/10, achieving a validation f1 of ~0.64. This translated to an overfitting to the validation set, and thus didn't perform as well on the kaggle data as effectively as hoped.

A 4)






```
The top 5 features are ['interval in seconds', 'NNRR', 'HR', 'ULF', 'TP']
The top 5 feature paris are: ['interval in seconds, HR', 'interval in seconds, TP', 'NNRR, HR', 'interval in seconds, ULF', 'interval in seconds, NNRR']
```

To calculate feature importance, I utilized my Logistic Regression model using the model.coef_ parameter, which returns the positive and negative weight values of the model. Using the absolute values of these, I found the top k features based on greatest weight, where k=5. Plotted in the above left graph is the feature weight value per feature, graphically showing the order of importance. To the right is the f1 score of the LR model, and on the y is the feature pairs that the individual model was trained on. Below these two graphs I am stating the top 5 features to be interval in seconds, NNRR, HR, ULF, and TP, and the pairs being interval in seconds with HR, interval in seconds with TP, NNRR with HR, interval in seconds with ULF, and interval in seconds with NNRR.

These top features are relevant because they hold the most weight in the selected model, LR. Taking a look at the top features, we see that the feature interval in seconds pops out on top. This is the time interval between two heartbeats in milliseconds. This makes sense to be the top feature because as the mind / body becomes more stressed, it is likely that the interval between heartbeats increases, and likewise as a person becomes relaxed this interval decreases. This is similar to HR, which is why we see that listed in the top 5 features as well.

Looking at the pairs, we can note that the top pair is interval in seconds with HR. Naturally this makes sense based on the reasoning in the paragraph prior, the HR and the interval of heartbeats are similar and carry large weight in the prediction of stress in this dataset. Likewise, the second pair on the top of the list is interval in seconds and TP, or the total spectral power of all NN intervals up to 0.04 Hz. Given that a lot of the features have to do with a given frequency range, the total spectral power makes sense to be in the top of the features. The pairs as a whole list give indication that the most important feature is interval in seconds as that appears most often in the pairs, which lines up as that is the top feature.

A 5)

I used 4 different models for this problem: Logistic Regression, Random Forest, K Nearest Neighbors, and a Sequential NN. In answer 3 I have the metrics for the last 3 models I tried, this is because LR performed the worst on the data. I believe this is because of the simplicity of the LR model. Since this model is a linear model and the data is complex, this model struggles heavily to perform well. Random Forest was a large improvement on validation f1 scores as this model is able to represent non-linear equations easily, and is an ensemble method. This ML algorithm performed about .1 better on f1 across the board then LR did. K Nearest Neighbors, a relatively simple algorithm, performed well with respect to the other models. The main idea behind this clustering type of algorithm is that things that are similar will be grouped together, because they're similar! This seems intuitive enough that the algorithm is able to get a good representation of the data after some hyper-parameter tuning. Lastly, the most adaptable model that I used was the NN because of the nature of them. Since the non-linear activation functions enable these layers to represent non-linear equations, and thus non-linear data, this model performed the best on the kaggle and often the validation data (depending on the split of data, imputation method used). This makes sense that this model would perform the best as the simple NN model is able to store massive amounts of weight values for learning the best feature

representation in a high dimensional space. All these models indicate that this problem is a complex one and the data available is non-linear in nature.

A 6)

The model that ended up getting the best results on Kaggle for me was the small sequential NN that I implemented, with a close second being the K nearest neighbors algorithm. Given the nonlinearity and model complexity that neural networks can represent, it is not surprising that this model achieved the highest results on kaggle. This NN model consisted of 7 dense layers and a dropout layer, which was trained across 150 epochs of the data. I used keras callbacks to track the validation loss and found my best kaggle results at 0.62 f1 when optimizing for validation loss instead of validation accuracy. When I was optimizing for validation accuracy, by tracking it with callbacks, I found that I was overfitting to the validation dataset, as I was getting ~0.65 f1 on validation and anywhere from 0.59-0.61 on kaggle test data.