Group D Members and Contributions

Alberto Valle: Generated graphs for the different evaluation metrics and worked primarily with the code training of the CatBoost algorithm and the preprocessing of the emotions dataset.

Mathew Simon: Researched libraries, choices of evaluation metrics, objective functions, how metrics are calculated. Extensive work on both CatBoost and XGB code. Ran grid searches on desktop PC. Created data for PCA emotions, feature importance for regression, and CatBoost graphs.

Adrian Luciano: Researched documentation and implementations on XGBoost and CatBoost on MNIST dataset as well as fine-tuning hyperparameters to establish the best model. Primarily focused on the implementation of XGBoost on datasets. Generated points for graphing.

Abbas Siddiqui: Researched the documentation for Catboost and XGBoost models. Experimented with initial parameters tuning on both models. Assisted in running tests for the Catboost/XGBoost models. Assisted in getting data points for the graphs.

All members contributed to planning/formatting for the presentation and report.

Objective

This project explores the use of XGBoost and CatBoost for classification and regression tasks: MNIST, Emotions, and Superconductivity. Also, this project describes the different techniques for preprocessing the datasets as well as the tuning of hyperparameters during training. Overall, this report should work as a guide on how to approach a classification and regression problem using the different and well-known techniques in machine learning.

CatBoost

This is a gradient-boosted trees algorithm that ranks among the best in prediction quality and learning speed. It is known for working really well with default parameters with great categorical feature support. Also, it allows the use of single/multi-GPU and provides a well-documented library with visualization tools for multiple evaluation metrics during training.

MNIST Classification

Dataset Summary
Features: image 28x28 (784) pixels
Classes: 10 (base 10 digits)
N: 70000
train: 60000
test: 10000

Objectives and metrics:
XGBoost
Evaluation metric: merror = #(wrong cases)/#(all cases)
The objective function we used for XGBoost was ‘multi:softmax’.

CatBoost
Evaluation metric: #(wrong cases)/#(all cases)
We used CatBoost's 'MultiClass' objective function as shown below:

MultiClass

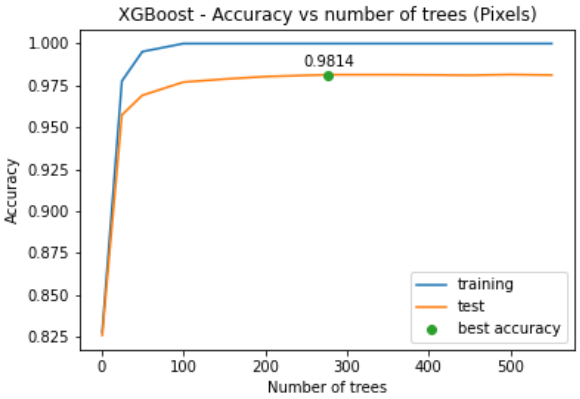
$$\frac{\sum_{i=1}^{N} w_i \log\left(\frac{e^{a_{it_i}}}{\sum_{j=0}^{M-1} e^{a_{ij}}}\right)}{\sum_{i=1}^{N} w_i},$$

$$t \in \{0, ..., M-1\}$$

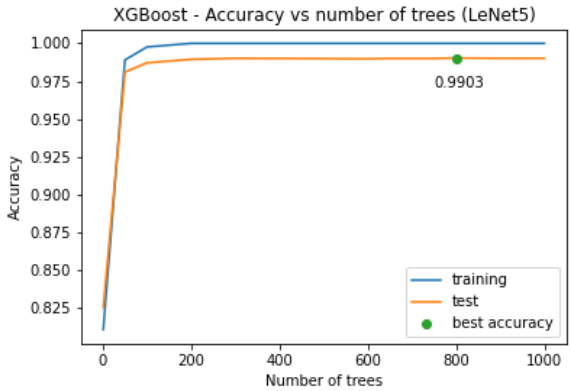
For convenience in comparing models between groups, we have provided data on the test set rather than validation.

Best model params:

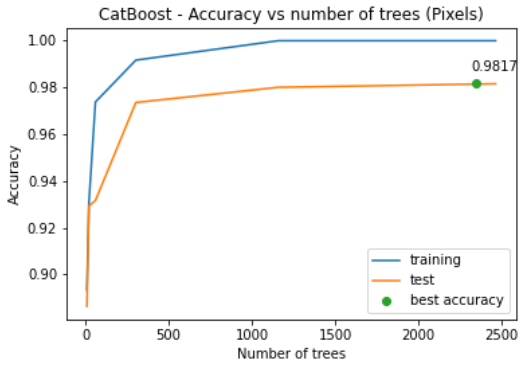
XGBoost (Pixels): {
'num_class': 10,
'n_estimators': 276,
'max_depth': 5,
'eta': .3
}



XGBoost (LeNet5): {
'num_class': 10,
'n_estimators': 800,
'max_depth': 2,
'eta': .4
}

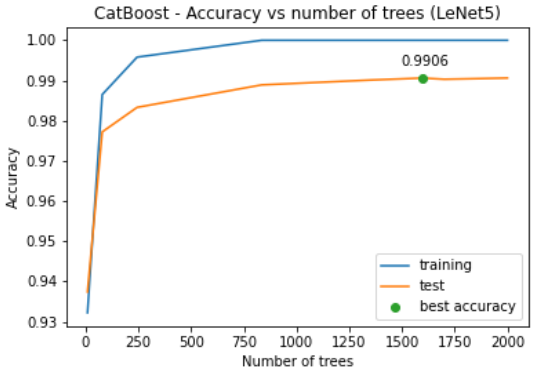


CatBoost (Pixels): {
'Iterations: 2345,
'learning_rate': 15,
'depth: 8',



'l2_leaf_reg: 3'
}

CatBoost (LeNet5): {
'Iterations: 1597,
'learning_rate': .15,
'depth: 8',
'l2_leaf_reg: 3'
}



Emotions Classification
Dataset Summary
Source: https://www.kaggle.com/deadskull7/fer2013

Classes: 7 categories

Features: image 48x48 (2304) pixels

Size: 34034 images

Train shape: (21531, 2304) - 60%
Validation shape: (7178, 2304) - 20%
Test shape: (7178, 2304) - 20%

Note that we define accuracy as the number of correct classifications over the total number of predictions.

Dataset Analysis

- Inconsistencies were found in the jpeg format of the dataset such as fully black or white, and repeated images. After doing more research the original version of the dataset was found in Kaggle in csv format.
- The disgust class is considerably unbalanced (547 compared to ~5000 in other classes). We tried dropping the class but found no impact on the accuracy in the other classes. Therefore, we decided to leave the class in the dataset. Also, we tried using different weights for each class (angry: 1, disgust: 9.7, fear: 1, happy: 0.6, neutral: 0.9, sad: 1.2, surprise: 0.85) but saw a decrease of ~0.3 in the overall accuracy.
- We tried to reduce the image size to 20x20 from the original 48x48 but found a decrease in accuracy of ~2-4 percent in some of the classes. However, we did utilize the smaller images for the coarse search of hyperparameters since the training speed was significantly reduced.

Preprocessing Steps

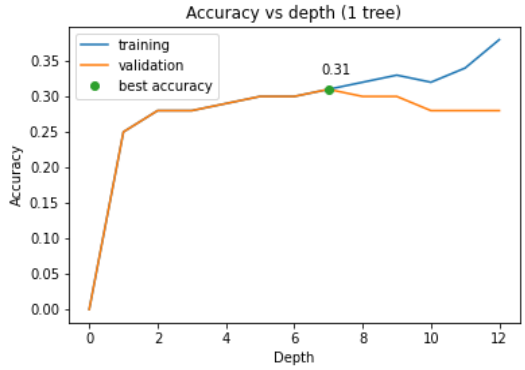
1. Load dataset into a pandas data frame.
2. Create a loop and iterate over the images. Append the images and labels to corresponding train_x/train_y and test_x/test_y NumPy arrays
3. Create a validation set using sklearn train_test_split
    a. Set test_size=0.25 and random_state=42
4. Finally, create Pools (CatBoost dataset preprocessing data structure) for train, validation,

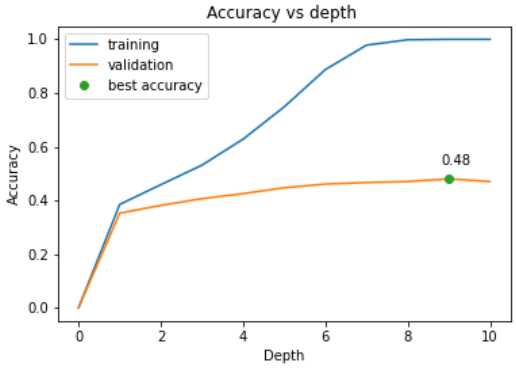
We chose not to apply data standardization or normalization because our tree-based models would choose the same decision splits. Gradient boosting would not be affected by any monotonic transformation of the variables and thus does not require normalization.

Tuning Hyperparameters

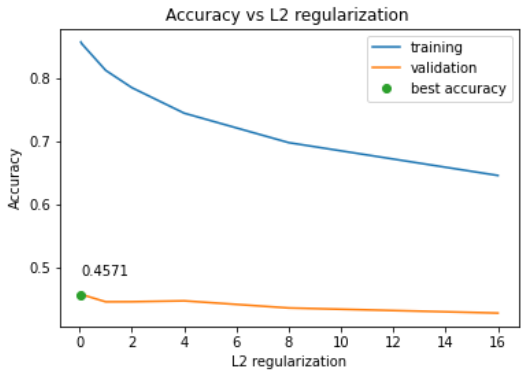
For the tuning of the hyperparameters, we decided to try different values for learning rate, iterations, depth, and l2_leaf_regularization as recommended in the CatBoost documentation.

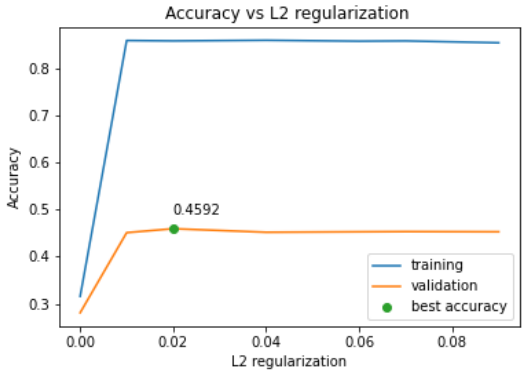
First, we tried different learning rates from 0.03 to 0.12 and found 0.05 with 5000 trees to be the best in training time and accuracy.



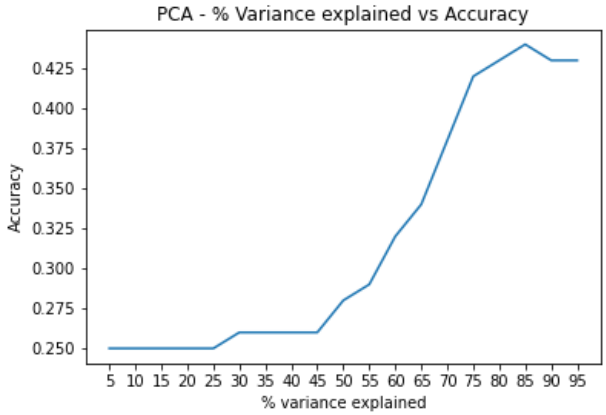


Next, we tried using depth values from 1 to 12 using 1 tree(left image) and then using 1000 trees(right image) from 1 to 10 and found 6 to be the best value with the accuracy/training time tradeoff since one depth increase resulted in approximately doubling the training time using a GPU. Interestingly, we were not able to run more than 10 depths using 1000 trees due to the 32 GB RAM | GTX 1070 GPU running out of memory.

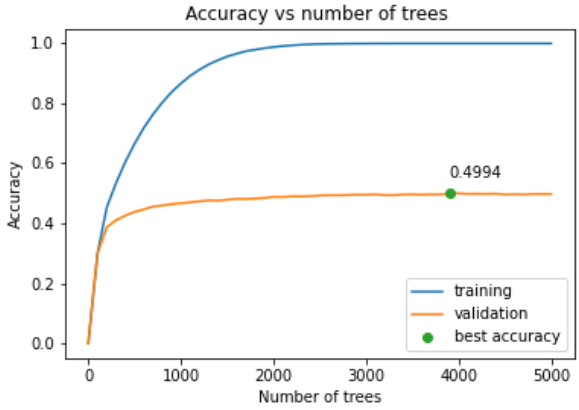




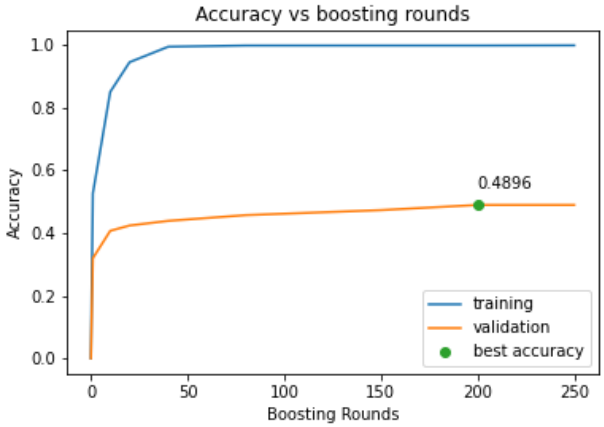
Also, we tested L2 leaf regularization using 1000 iterations for large values where we noticed a steady decrease but a positive impact on small values. Therefore, as we can see in the right image we tested smaller intervals and found that values between 0.02 and 0.09 regularization are the best



Last, we tried to use PCA in order to reduce the number of dimensions of the data to see if it would increase the accuracy of our model. However, as we can see in the graph above dimensionality reduction did not help improve accuracy.



Next, we increased the number of trees while trying the most optimal values of l2 leaf regularization and depths and found 0.4994 to be the best accuracy with: {iterations=3900, learning_rate=0.05, depth=6, l2_leaf_reg=0.09}.

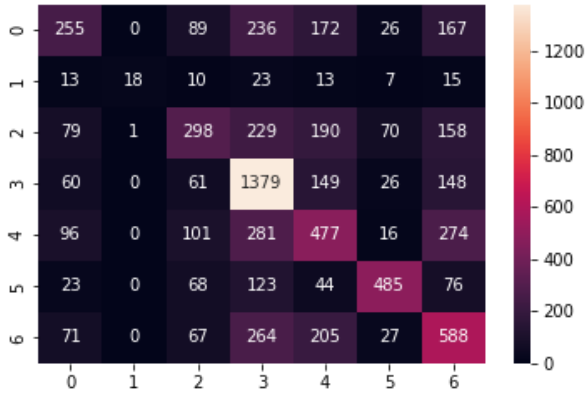


Finally, we ran a similar model using XGBoost where we found the best hyperparameters to be:
{boosting_rounds=200, max_depth=8, eta=0.3}

Final Model Analysis

CatBoost Test Accuracy: 0.49205

XGBoost Test Accuracy: 0.4905



Labels [0: angry, 1: disgust, 2: fear, 3: happy, 4: neutral, 5: sad, 6: surprise]

In the confusion matrix, we found that one of the possible reasons for low accuracy is the high similarity between face gestures in different emotions. For example fear and happiness (2, 3), anger and happiness (0, 3), happiness and surprise (6-3). This suggests that a relabeling of the images (left image) may result in a model with better accuracy.

Superconductor Regression

Dataset Summary

Source: https://archive.ics.uci.edu/ml/datasets/Superconductivty+Data

Features: 81 numerical values
Target: Temperature (K)
N: 21263 materials
training: (17010, 81) - 80%
test: (4253, 81) - 20%
For regression, RMSE and R2 are defined as shown below:

RMSE

$$\sqrt{\frac{\sum_{i=1}^{N}(a_i - t_i)^2 w_i}{\sum_{i=1}^{N} w_i}}$$

R2

$$1 - \frac{\sum_{i=1}^{N} w_i (a_i - t_i)^2}{\sum_{i=1}^{N} w_i (\bar{t} - t_i)^2}$$

$\bar{t}$ is the average label value:

$$\bar{t} = \frac{1}{N}\sum_{i=1}^{N} t_i$$

The objective function was squared error

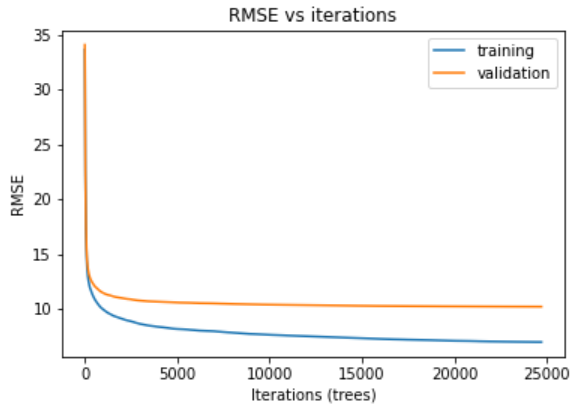
Dataset Analysis
The 81 features consisted of information about the materials such as

Preprocessing Steps

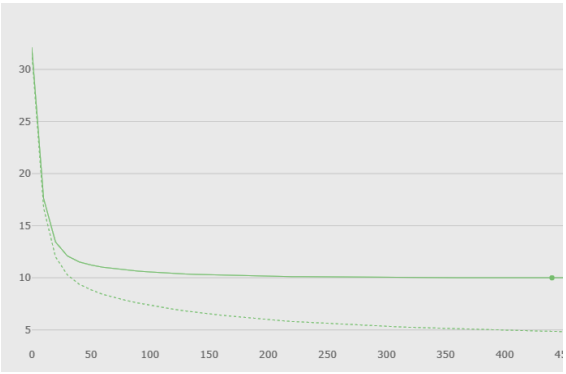
1. Load dataset into a numpy array from csv with np.genfromtxt()
2. Remove first row and column (labels), split last column into target np array
3. Split data into train and validation sets using SKlearn train_test_split()
4. Create Pools / DMatrix of train and validation sets

Again, we chose not to apply data standardization or normalization because gradient boosting would not be affected by any monotonic transformation of the variables, and thus does not require these transformations.

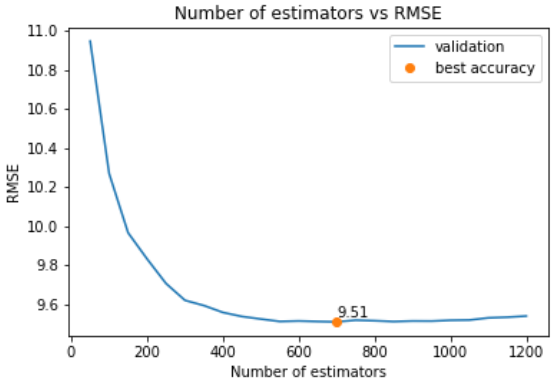
Tuning Hyperparameters
We first tried a coarse tuning of the hyperparameters; we allowed CatBoost to auto-set the learning rate for a 100000 iteration model without regularization and with CatBoost's overfit detector. Later, we found much less complex models that gave similar or slightly better RMSE scores, but this was an interesting application of CatBoost's auto-set learning rate.



CatBoost best model:
Iterations vs RMSE



XGBoost best model:



For XGBoost, we tested boosting rounds (up to 5000), depth (1 to 12), learning rate (0.05 to 0.3), and L2 Leaf Reg (.1 - 5). After identifying promising ranges, we ran a grid search over boosting rounds (400, 425, 450), depth (8, 10, 12), learning rate (.05, .1, .15), and L2 Leaf Reg (3).

For CatBoost we applied GridSearchCV via CatBoostRegressor(). This was a 5-fold cross validation. For XGBoost we applied xgb.cv() which was also 5-fold cross validation.

XGBoost final model

XGBoost: {
'n_estimators=700',
'max_depth = 5',
'learning_rate=0.19'
}

Cross - validated R^2: 0.9200

RMSE: 9.520

Test R^2: 0.8721

CatBoost final model

CatBoost: {
'iterations: 442',
'depth': 12,
'learning_rate': .1,
'l2_leaf_reg': 3

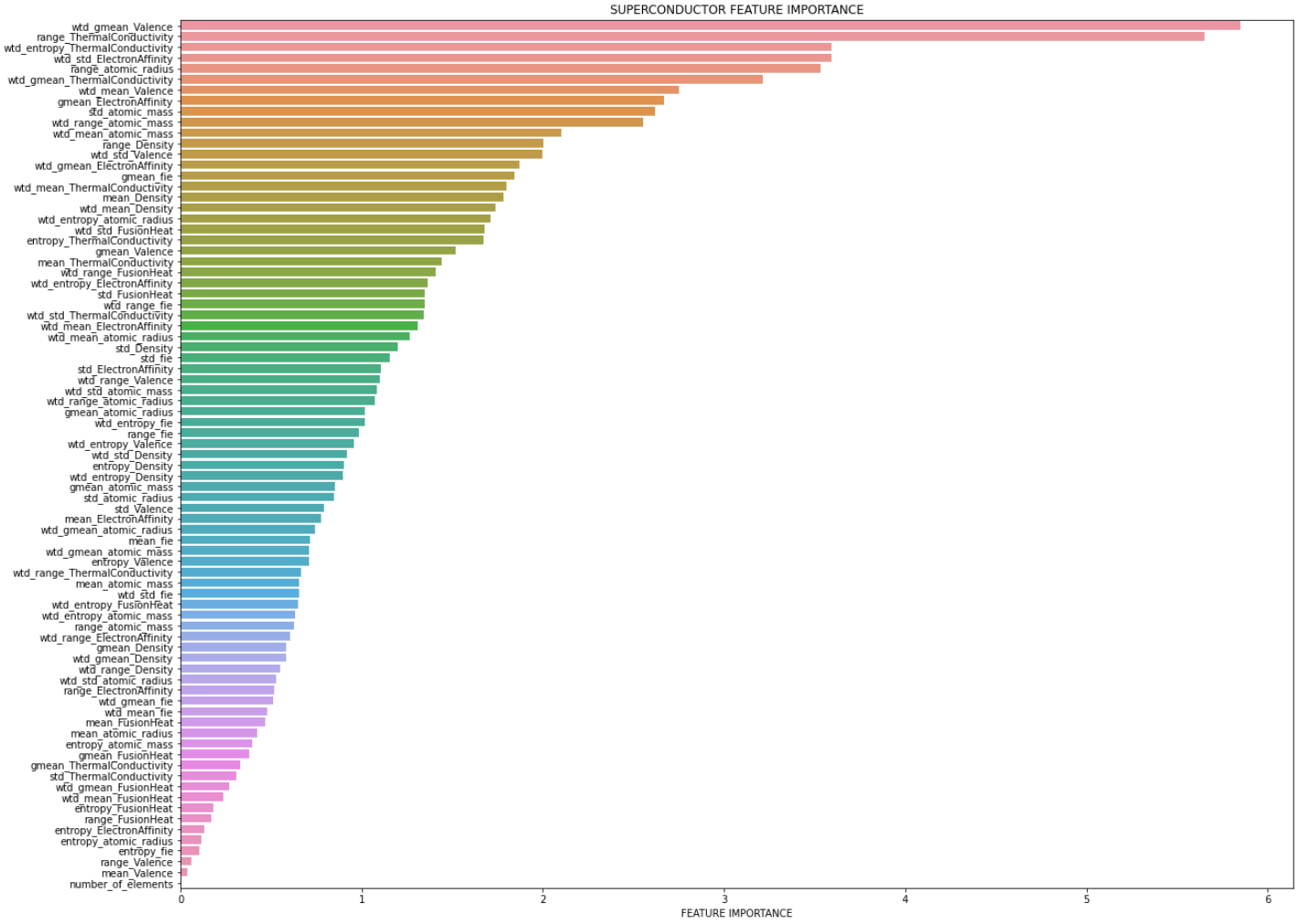
}

Cross-validated R^2: 0.9203

test R^2: 0.9197

test RMSE: 9.99

Feature Importances



We ran CatBoost's feature importance calculation; this calculation is made by fully training a model, then for each feature, removing that feature from the model and estimating the change in the loss function (RMSE). Below is the resulting feature importance array:

[0.00643883 0.65509504 2.10202658 0.85307253 0.71191134 0.39804051
 0.63099884 0.62479374 2.55158346 2.61872861 1.08326704 0.71292587
 0.48041616 1.84414959 0.511986   0.10420205 1.01964177 0.98496449
 1.34664632 1.1532307 0.65425243 0.42194647 1.26463227 1.01992879
 0.74407   0.11571312 1.70927582 3.53006683 1.07400647 0.84937187
 0.52981731 1.78132978 1.73874473 0.58483072 0.58091767 0.90183711
 0.89670274 2.0034773 0.54767501 1.19912228 0.91957726 0.77403731
 1.31207942 2.67095603 1.8717508 0.13467437 1.36264821 0.51657194
 0.6073015 1.10763484 3.59173214 0.46537547 0.23900207 0.38018251
 0.26955221 0.18401211 0.64890858 0.17245337 1.40614931 1.35007816
 1.67589265 1.44307117 1.79725373 0.32739988 3.21202711 1.67168057
 3.59246437 5.65184469 0.66686671 0.30936599 1.34462877 0.03893541
 2.75223143 1.52028509 5.84833779 0.71017883 0.95864923 0.06191112
 1.09811928 0.79460338 1.99773701].

The most important feature to the resulting RMSE is wtd_gmean_Valence at an estimated change of 5.85.

Conclusion

Both CatBoost and XGBoost are gradient-tree boosted models that support multi-classification and regression. We found that CatBoost was easier to work with and had excellent documentation compared to XGBoost. Also, CatBoost allowed everything except number of iterations to be either default or automatically heuristically estimated to give close to ideal results, which was very useful when starting out with a new dataset. Furthermore, the CatBoost library's visualization tools were easier to work with. Maybe due to a difference in boosting algorithm, CatBoost seemed to need more iterations to perform similarly to XGBoost, but had a much faster calculation time per iteration.

Sources

https://catboost.ai/en/docs/concepts/python-reference_catboost
https://catboost.ai/en/docs/concepts/parameter-tuning
https://www.kaggle.com/deadskull7/fer2013
https://github.com/microsoft/FERPlus
https://catboost.ai/en/docs/concepts/python-reference_catboostclassifier
https://catboost.ai/en/docs/concepts/python-reference_catboostregressor
https://xgboost.readthedocs.io/en/stable/python/python_api.html#xgboost.Booster.predict