Department Of Statistics
University of North Carolina at Chapel Hill
December 2, 2023
STOR 565 Fall 2023

WRITING QUALITY ANALYSIS FROM KEYSTROKES

PRANISH PANTHA
NIMIL PATEL
SANJAY RAVINDRAN
SAMANYU KUNCHANAPALLI
NISHTHA MUKHERJI

Abstract. In this study, we explore the application of neural networks and random forests in assessing writing quality through keystroke analysis. The research focuses on analyzing keystrokes from SAT writing prompts, recorded over 30-minute sessions from 2,471 participants, amounting to over 8.4 million logs. Key metrics such as production rate, pause rate, word count, deletion rate, and others are derived from these logs. Neural networks are employed for their ability to model complex patterns in large datasets, making them ideal for interpreting the nuanced relationships between these metrics and writing quality. Meanwhile, random forests are used for their robustness in handling diverse data types and their efficacy in feature selection, crucial for isolating significant predictors of writing quality among the various metrics. By leveraging these sophisticated machine learning techniques, the study aims to establish a more accurate and comprehensive understanding of the factors that contribute to effective writing, as evidenced by keystroke dynamics.

1. ***AI USE DISCLAIMER***

Chat GPT was used for the exploration of the Python packages used. Prompting was strictly for exploring options such as TensorFlow's "Adam" and "Tuner". It was also used at times for Debugging.

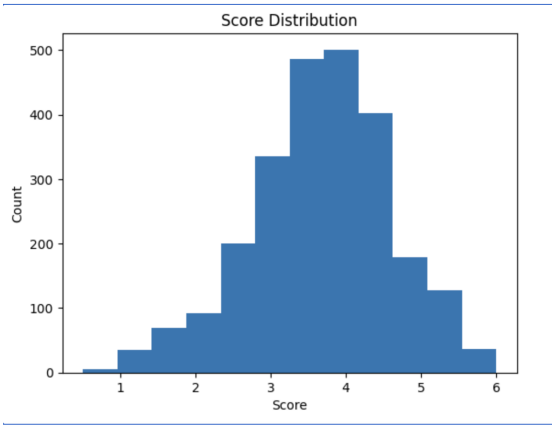
2. Introduction

We chose to compete in the Linking Writing Processes to Writing Quality competition on Kaggle. The goal of this competition is to predict overall writing quality, hoping to establish a relationship between writing style and quality. Participants are given 30 min to respond to an SAT writing prompt, which are then scored out of 6 points. Their keystrokes are recorded and provided as data in the form of logs. This competition focuses on capturing insights on the writing process rather than the final essay contents.

3. Data

The data was provided in comma separated values (csv) form. It contains 8,434,765 logs, spanning 2,471 individuals. Each log was made up of 10 variables:

variable	description
id	identifies the individual essay
event_id	identifies log within essay
down_time	time in ms when a key was pressed down
up_time	time in ms when the key or mouse was release
action_time	how long a specific keystroke took
down_event	time when key is pressed
up_event	time when key was released
activity	category of log (Input, Removal, etc)
text_change	text delta after log
score	score assigned to essay



Based on the distribution of scores above, we can see that the scores are normally distributed (with a mean between 3.5-4). This means that scores in the middle range are more common and easier to predict. This is good, as it means that we will be able to predict the scores of most students with high accuracy. However, it also means that it will be difficult to predict the scores of students who are outliers (very high or very low scores) due to the lack of data points in these regions.

4. Metrics

With the raw data we were provided, each entry would have an average of 10,000+ variables. This is caused by each individual having thousands of logs and each log having 10 variables. It is very difficult to analyze the data in this format. We decided to reduce the dimensionality of our data by computing key metrics which best summarize the data.

```
# Load in the data from csv to pandas dataframe
import pandas as pd
```

```
train_logs_df = pd.read_csv('data/train_logs.csv')
train_scores_df = pd.read_csv('data/train_scores.csv')
test_logs_df = pd.read_csv('data/test_logs.csv')
```

1. Production Rate: this represents the raw rate of words produced per minute.

$$pr(t) = \frac{\Delta words}{\Delta t} \mid pr(0) = 0$$

This is calculated by dividing the total number of words produced by the total time spent writing. This is a good metric for measuring the raw output and speed of a writer, but it does not take into account the quality of the writing. It is also not a good metric for measuring the output of a writer who is not writing continuously, as it will be skewed by the time spent not writing.

```
# Production Rate (words per minute)
production_rates = []
for id, df in train_logs_df.groupby('id'):
    word_count = df['word_count'].iloc[-1]
    time_delta = (df['up_time'].iloc[-1] - df['down_time'].iloc[0
                                        ]) / 60000
    production_rates.append(word_count / time_delta)
train_scores_df['production_rate'] = production_rates
```

2. Pause Rate: this represents the rate of pauses per minute.

$$pa(t) = \frac{\Delta t_{pause}}{\Delta t} \mid pa(0) = 0$$

This is calculated by dividing the total sum of time between logs (pauses) by the total time spent writing. More pauses per minute may indicate a more deliberate writing style, but it is hard to seperate this from a writer who is simply distracted.

```
# Pause Rate (sec paused per minute)
pause_rates = []
for id, df in train_logs_df.groupby('id'):
    i = df.index[0]
    pause_sum = (df['down_time'].loc[i+1:i+len(df['down_time']-1)
                                        ] - df['up_time'].loc[i:i+
                                        len(df['up_time'])-2].values
                                        ).sum() / 1000
    time_delta = (df['up_time'].iloc[-1] - df['down_time'].iloc[0
                                        ]) / 60000
    pause_rates.append(pause_sum / time_delta)
train_scores_df['pause_rate'] = pause_rates
```

3. Word Count: this represents the total number of words written.

$$wc(t) = \sum_{i=0}^{t} words_i$$

This is calculated by summing the total number of words written at each log. This metric differs from production rate in that it is not a rate based metric, but

rather a cumulative metric. This means that it is not affected by the speed of writing, but rather the total amount of writing.

```
# Word Count
word_counts = []
for id, df in train_logs_df.groupby('id'):
    word_counts.append(df['word_count'].iloc[-1])
train_scores_df['word_count'] = word_counts
```

4. Deletion Rate: this represents the rate of deletions per minute.

$$dr(t) = \frac{\Delta words_{deleted}}{\Delta t} \mid dr(0) = 0$$

This is calculated by dividing the total number of words deleted by the total time spent writing. This metric is a good indicator of the quality of writing, as it is a measure of how much a writer is editing their work by deleting prior text and replacing with improved versions.

```
# Deletion Rate (deletions per minute)
deletion_rates = []
for id, df in train_logs_df.groupby('id'):
    # Set Deletion Rate (deletions per minute)
    deletion_count = len(df[df['activity'] == 'Remove/Cut'])
    time_delta = (df['up_time'].iloc[-1] - df['down_time'].iloc[0
                                        ]) / 60000
    deletion_rates.append(deletion_count / time_delta)
train_scores_df['deletion_rate'] = deletion_rates
```

5. Insertion Rate: this represents the rate of insertions per minute.

$$ir(t) = \frac{\Delta words_{inserted}}{\Delta t} \mid ir(0) = 0$$

This is calculated by dividing the total number of words inserted by the total time spent writing. This metric is a good indicator of the quality of writing, as it is a measure of how much a writer is editing their work by adding more complex structure and style to the beginning and middle of their essay.

```
# Insertion Rate (insertions per minute)
insertion_rates = []
for id, df in train_logs_df.groupby('id'):
    time_delta = (df['up_time'].iloc[-1] - df['down_time'].iloc[0
                                        ]) / 60000
    input_count = len(df[(df['activity'] == 'Input') & (df['
                                        activity'] != df['activity']
                                        .shift())])
    insertion_rates.append(input_count / time_delta)
train_scores_df['insertion_rate'] = insertion_rates
```

6. Pause Burst Rate: this represents the rate of pause bursts per minute. Here, a pause burst is defined as a period of continous productivity that is terminated

by a pause.

$$pbr(t) = \frac{\Delta bursts_{pause}}{\Delta t} \mid pbr(0) = 0$$

This is calculated by partitioning the logs into productivity segments that are seperated by 1 or more pauses. Then this value is divided by the total time spent writing. This metric is a good indicator of the quality of writing, as productive bursts are often associated with periods of high creativity and focus. This differs from Revision Burst Rate in that it associates with pauses or periods of thinking, rather than revisions.

```
# Pause Burst Rate (pause bursts per minute)
def count_p_bursts(df, pause_thresh = 2000):
    p__bursts = 0
    last_time = 0

    for _, row in df[df['activity'] == 'Input'].iterrows():

        current_time = row['down_time']

        if (current_time-last_time) > pause_thresh:
            p__bursts += 1

        last_time = row['up_time']


    return(p__bursts)

p_bursts_per_min = []
for id, df in train_logs_df.groupby('id'):
    p_burts_ct = count_p_bursts(df)
    time_delta = (df['up_time'].iloc[-1] - df['down_time'].iloc[0
                                        ]) / 60000
    p_bursts_per_min.append(p_burts_ct/time_delta)
train_scores_df['p_burst_per_min'] = p_bursts_per_min
```

7. Revision Burst Rate: this represents the rate of revision bursts per minute. Here, a revision burst is defined as a period of continous productivity that is terminated by a revision.

$$rbr(t) = \frac{\Delta bursts_{revision}}{\Delta t} \mid rbr(0) = 0$$

This is calculated by partitioning the logs into productivity segments that are seperated by 1 or more revisions. Then this value is divided by the total time spent writing. This metric is a good indicator of the quality of writing, as productive bursts are often associated with periods of high creativity and focus. This differs from Pause Burst Rate in that it associates with revisions or active editing, rather than pauses.

```
# Revision Burst Rate (revision bursts per minute)
def count_r_bursts(df):
    r__bursts = 0
```

```
    last_activity = None

    for _, row in df.iterrows():
        current_activity = row['activity']

        if current_activity == 'Input':
            if last_activity != 'Input' and last_activity != '
                                                Nonproduction':
                r__bursts += 1
        last_activity = current_activity

    return(r__bursts)

r_bursts_per_min = []
for id, df in train_logs_df.groupby('id'):
    r_bursts_ct = count_r_bursts(df)
    time_delta = (df['up_time'].iloc[-1] - df['down_time'].iloc[0
                                        ]) / 60000
    r_bursts_per_min.append(r_bursts_ct/time_delta)
train_scores_df['r_burst_per_min'] = r_bursts_per_min
```

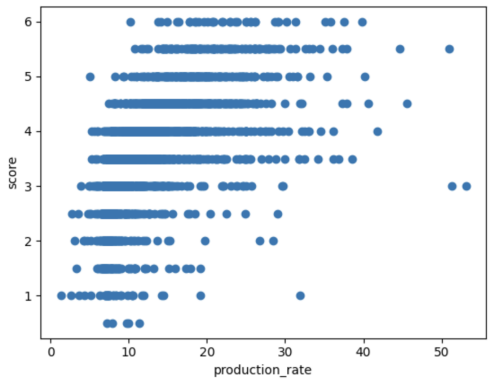
```
# Plot correlation matrix
corr = train_scores_df[['production_rate', 'pause_rate', '
                                word_count', 'deletion_rate', '
                                insertion_rate', '
                                p_burst_per_min', '
                                r_burst_per_min']].corr()
corr.style.background_gradient(cmap='coolwarm')
```

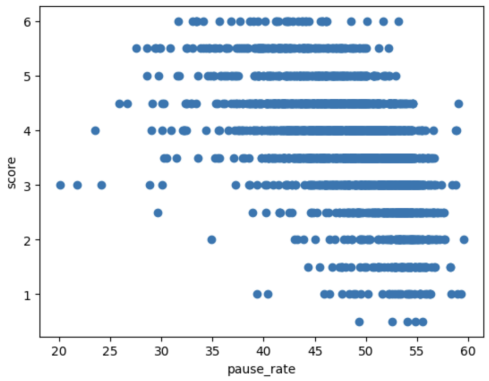
	production_rate	pause_rate	word_count	deletion_rate	insertion_rate	p_burst_per_min	r_burst_per_min
production_rate	1.000000	-0.738637	0.907933	0.281357	0.395419	0.050720	0.437070
pause_rate	-0.738637	1.000000	-0.688394	-0.581277	-0.482491	-0.148273	-0.574374
word_count	0.907933	-0.688394	1.000000	0.300396	0.383424	0.055934	0.430683
deletion_rate	0.281357	-0.581277	0.300396	1.000000	0.485211	0.312909	0.706712
insertion_rate	0.395419	-0.482491	0.383424	0.485211	1.000000	0.195128	0.647883
p_burst_per_min	0.050720	-0.148273	0.055934	0.312909	0.195128	1.000000	0.196956
r_burst_per_min	0.437070	-0.574374	0.430683	0.706712	0.647883	0.196956	1.000000

Based on the correlation matrix above, we can see that most of the metrics are not highly correlated with each other, which is good. This means that each metric is capturing a different aspect of the writing process. Certain metrics are highly correlated with each other, such as Production Rate and Word Count, which makes sense as they are similar measures of the amount of writing produced. Since we believe that each metric is capturing a different aspect of the writing process, we will keep all of them in our analysis.

We plotted scatterplots for metrics with high correlation with score. Both Production Rate and Pause Rate display linear relationships with score, with Production Rate having a positive correlation and Pause Rate having a negative correlation. This makes sense, as a higher production rate is associated with a higher amount of writing, which is associated with a higher score. Similarly, a higher pause rate

is associated with a lower amount of writing, which is associated with a lower score. This is a good sign, as it means that these metrics are capturing the correct information and will be useful in predicting score.





5. Random Forests

First, we decided to train a random forest model in order to predict essay scores. Random forests seemed to be a good candidate for this problem as it can simultaneously prevent overfitting and improve accuracy, due to its aggregation of numerous trees and its easy hyperparameter tuning. Also, random forests are particularly good at dealing with multicollinearity, since they take a sample of covariates across its aggregated trees. One potential issue that could arise with random forests is their complexity, which can be computationally inefficient. Another complication was the approach to take with training models for this problem; specifically whether or not to treat this as a classification or regression problem. There are 11 distinct possible scores, so it may be possible to train a classifier. However, for random forests, we chose not to, as there were very low numbers of observations in certain classes, which caused very low accuracies. In addition, with so many potential classes, it was also difficult for the model to predict scores. To train the random forest model, the training scores data was split into a separate training and testing set(different than the test set used for kaggle submission scoring. Then, using RandomForestRegressor from sklearn.ensemble, we trained a random forest model with default parameters.

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error
model = RandomForestRegressor(bootstrap=True)
model.fit(X_train, y_train)
pred = (np.round(model.predict(X_test) *2)) / 2
mse = mean_squared_error(y_test, pred)
```

The default random forest model did not perform terribly, with a MSE of **0.501**, even without cross validation or hyperparameter tuning. From here we decided to see if we could improve this model.

We used the RandomSearchCV to fit and fine tune the model with cross validation. This is much less computationally demanding than a simple grid search,

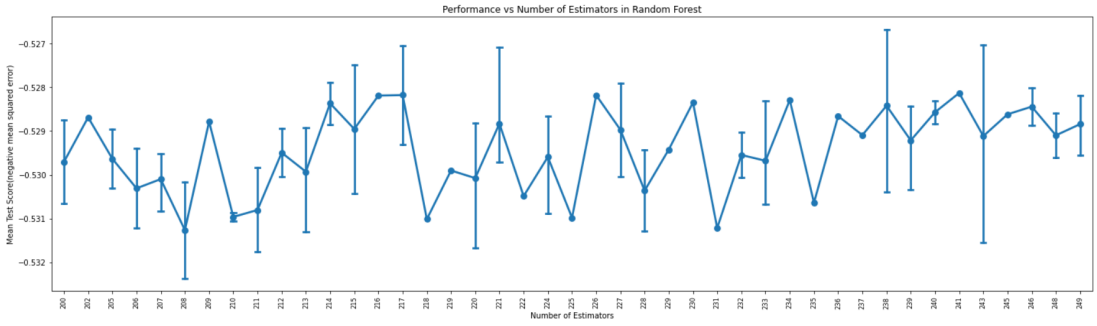
and can also reveal more about the strengths and weaknesses of our model across various combinations of parameters. Some of the possible parameters to tune included n estimators, representing the number of estimators to use in each tree, and max depth, representing the max height of each subtree.

Starting with a wide range for each, we worked to narrow them down by continuous tweaking with the aim of minimizing test MSE. Eventually, we narrowed it down enough and found the optimal hyperparameters. Below is a plot of n estimators against their respective negative mean squared errors. It is important to keep in mind that each of these values also correspond to a combination of other hyperparameters, which causes the random spread.

```
#Random search tuning for Random forest regressor
from scipy.stats import uniform, randint

random_params = {
    'n_estimators': sp_randint(200,250),
    'max_depth': sp_randint(5,8),
    'min_samples_split': sp_randint(2,5),
    'min_samples_leaf' : sp_randint(3,5),
    'max_features': ['sqrt', 'log2'],
    'max_samples':( 0.7,.8,.9)
    }


random_search= RandomizedSearchCV(model, param_distributions=
   random_params,
   n_iter=100, cv=5,
   random_state=20, n_jobs=-1, scoring='neg_mean_squared_error')
random_search.fit(X_train, y_train)
```



It is evident that the best negative error came from the combination of parameters at **238 estimators**. This combination of parameters and cross validation decreased the MSE to around **0.46**.

The plot below shows the feature importance in the final random forest model. Word count and production rate were the most important, while the burst metrics were the least. This matches our earlier correlation scores.

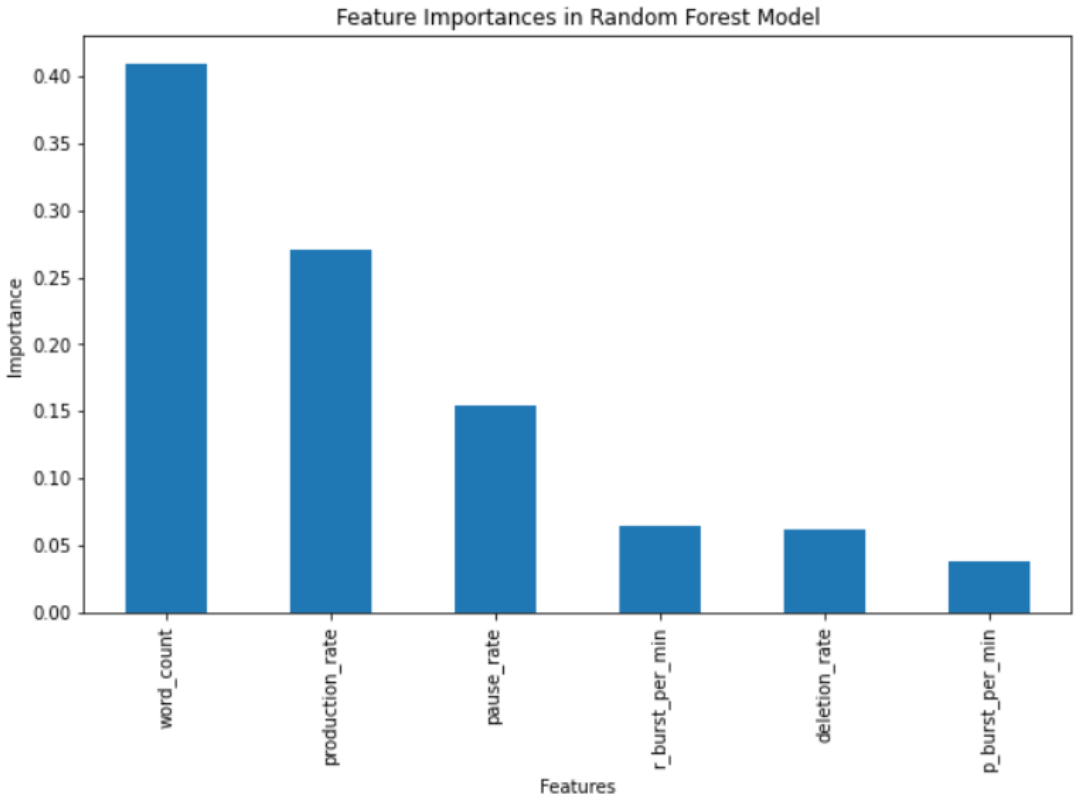


FIGURE 1. Caption for the image.

6. BOOSTING

Next, we wanted to see how a boosted model would compare to the random forest model, as it incudes extra regularization parameters, and can also handle this problem well. We took a similar approach as the training of the random forest model, using RandomSearchCV to tune and train the model using cross validation.

```
#Random Search tuning for XGB Regressor
param_dist_reg = {
    'n_estimators': sp_randint(300, 400),
    'learning_rate': uniform(0.01, .05),
    'max_depth': sp_randint(6, 15),
    'min_child_weight' : sp_randint(1,3),
    'gamma' : uniform(0,.5),
    'subsample': uniform(0.8, .2),
    'colsample_bytree': uniform(0.3, 0.7),
    'reg_lambda': uniform(0 , 1),
    'reg_lambda': uniform(0 , 1)


}
```

```
random_search_xg = RandomizedSearchCV(xgreg, param_distributions=
                                      param_dist_reg, n_iter=100, cv=5
                                      , verbose=2, scoring='
                                      neg_mean_squared_error',
                                      random_state=20, n_jobs=-1)
random_search_xg.fit(X_train, y_train)
```

It includes similar parameters to the random forests, with extra regulairzation parameters such as lambda(similar to ridge regression) and the learning rate. The convergence plot shows the change in negative MSE over the 100 iterations used to fit the model. The best score was found with the optimal combination of hyperparameters at around the 70th iteration.

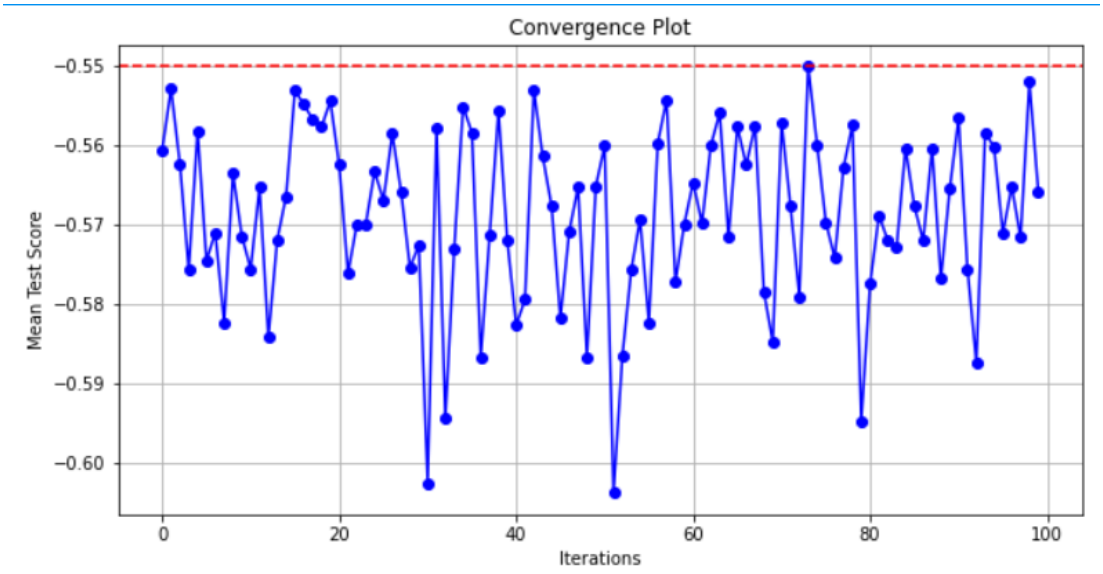


FIGURE 2. Caption describing the content of the image.

Comparing the performance of the boosted model and the random forest model. It seems like they performed similarly in terms of error, but the random forest performed slightly better. Thus we decided to only keep the Random Forest model to submit to the competition.

7. Neural Networks

7.1. **Categorical Neural Network with Tuning.**

```
import pandas as pd

train_scores_df = pd.read_csv('metrics.csv')

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam
```

```
from tensorflow.keras.utils import to_categorical
from keras_tuner.tuners import RandomSearch

# Split the data into features and target
X_train = train_scores_df.drop(['id', 'score'], axis=1)
y_train = train_scores_df['score']

# One-hot encode the labels
y_train = to_categorical(y_train, num_classes=12)

# Define a model-building function
def build_model(hp):
    model = Sequential()
    model.add(Dense(hp.Int('input_units', 32, 256, step=32),
                                    input_dim=X_train.shape[1],
                                    activation='relu'))

    for i in range(hp.Int('n_layers', 1, 4)):
        model.add(Dense(hp.Int(f'dense_{i}_units', 32, 256, step=
                                        32), activation='relu'))

    model.add(Dense(12, activation='softmax'))
    model.compile(optimizer=Adam(hp.Choice('learning_rate', [1e-2
                                    , 1e-3, 1e-4])),
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
    return model

# Create a tuner
tuner = RandomSearch(
    build_model,
    objective='val_accuracy',
    max_trials=5,
    executions_per_trial=3,
    directory='my_dir',
    project_name='hparam_tuning')

# Run the hyperparameter search
tuner.search(X_train, y_train,
             epochs=10,
             validation_split=0.1)

# Get the optimal hyperparameters
best_hps = tuner.get_best_hyperparameters(num_trials=1)[0]

# Build the model with the best hyperparameters and train it on
                                    the data
classifier_model = tuner.hypermodel.build(best_hps)
history = classifier_model.fit(X_train, y_train, epochs=50,
                                    validation_split=0.1)
```

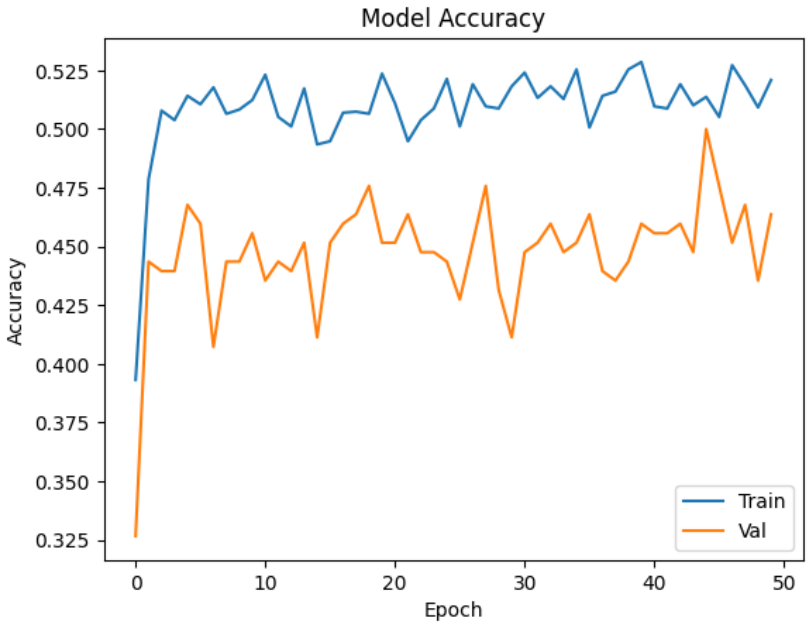


FIGURE 3. Accuracy Over Epochs

This code snippet demonstrates the process of hyperparameter tuning in a neural network using TensorFlow and Keras along with Keras Tuner. The code begins with the necessary imports, setting up a Sequential model from Keras, and preparing the dataset for training. The dataset, represented by 'train scores df', is split into features ('X train') and targets ('y train'). The targets are one-hot encoded to fit the format needed for categorical classification, suitable for a multi-class problem with 12 classes.

The core of this script is the 'build model' function, which dynamically constructs a neural network model. This function is designed to be used with Keras Tuner, a library for automated hyperparameter tuning. Inside this function, various aspects of the model, such as the number of units in each Dense layer and the learning rate of the Adam optimizer, are set to be tuned. The Keras Tuner's 'RandomSearch' method is then utilized to explore different hyperparameter configurations. By specifying metrics like 'val accuracy' and setting a limit on the number of trials and executions per trial, the tuner searches for the best hyperparameters over a defined search space. After the tuning process, the best hyperparameters are extracted, and a final model is built and trained on the data. This approach streamlines the optimization of model parameters, potentially leading to improved model performance on the given task. This neural network is a classification approach where there are 12 final classes and the class with the highest probability is chosen.

7.2. **Regression Neural Network with Tuning.**

```
    # Split the data into features and target
X_train = train_scores_df.drop(['id', 'score'], axis=1)
```

```
y_train = train_scores_df['score']

# Define a model-building function
def build_model(hp):
    model = Sequential()
    model.add(Dense(hp.Int('input_units', 32, 256, step=32),
                                    input_dim=X_train.shape[1],
                                    activation='relu'))

    for i in range(hp.Int('n_layers', 1, 4)):
        model.add(Dense(hp.Int(f'dense_{i}_units', 32, 256, step=
                                        32), activation='relu'))

    model.add(Dense(1, activation='relu'))
    model.compile(optimizer=Adam(hp.Choice('learning_rate', [1e-2
                                    , 1e-3, 1e-4])),
                  loss='mean_squared_error',
                  metrics=['mean_squared_error'])
    return model

# Create a tuner
tuner = RandomSearch(
    build_model,
    objective='val_mean_squared_error',
    max_trials=5,
    executions_per_trial=3,
    directory='my_dir',
    project_name='hparam_tuning')

# Run the hyperparameter search
tuner.search(X_train, y_train,
             epochs=10,
             validation_split=0.1)

# Get the optimal hyperparameters
best_hps = tuner.get_best_hyperparameters(num_trials=1)[0]

# Build the model with the best hyperparameters and train it on
                                    the data
regression_model = tuner.hypermodel.build(best_hps)
history = regression_model.fit(X_train, y_train, epochs=50,
                                    validation_split=0.1)
```

This code snippet is tailored for building and optimizing a regression model using TensorFlow, Keras, and Keras Tuner. Initially, the code prepares the dataset by splitting it into features (‘X train‘) and target (‘y train‘). Unlike the previous example which dealt with a classification problem, here the target ‘y train‘ is not one-hot encoded, indicating that the task is regression, not classification.

The function ‘build model‘ is designed to create a neural network model with tunable hyperparameters. The model is constructed using Keras’ Sequential API, with the number of units in each Dense layer and the learning rate of the Adam

optimizer being adjustable parameters. Notably, the output layer of the model has a single unit with a 'relu' activation function, which is typical for regression models. The model is compiled using the mean squared error (MSE) as both the loss function and the metric, aligning with the regression objective.

In the hyperparameter tuning phase, the 'RandomSearch' method from Keras Tuner is employed. The objective is set to 'val mean squared error', emphasizing the regression nature of the problem. The tuner explores different configurations within the defined limits of maximum trials and executions per trial. Following the search, the best hyperparameters are identified, and a final regression model is built and trained with these parameters. The use of MSE as the optimization metric and the setup of the neural network signify that this approach is specifically tuned for regression tasks. The implication that this model performed better suggests that the hyperparameter tuning was effective in optimizing the model's architecture and parameters for the specific regression task at hand. The **classfication** model had a test mean squared error of **0.78** while the **regression** model was able to achieve **0.72**.

8. Challenges and Limitations

(1) Essay Reconstruction

- In order to protect privacy, the essays could not be reconstructed from the train data. This was done by masking the text change variable.
- Other groups reconstructed from the hidden test data but we did not do this as it was not in the spirit of the competition

(2) Computational Effort

- Hyperparameter tuning and Neural Network training took a lot of time and computation, which slowed analysis work
- We used an NVIDIA GeForce RTX 3070 Ti to accelerate training by relying on built in CUDA libraries for GPU acceleration. We estimate it decreased our training times by 50% compared to CPU only computation

Note: There was still major overhead from tuning and training

9. Conclusion

In conclusion, our best model was Random Forest with hyperparameter tuning. When submitted to the competition, it received a Root Mean Square Error of **0.702**. This means that our predictions were only 1 score (based on 0.5 scoring standard) off on average. Our Regression Neural Network model with the Keras Tuner performed similarly, receiving a score of **0.72**. In the future, we can calculate finer tune metrics to capture more nuances of the writing style variation that the logs contain.

References

[1] Conijn, R., Cook, C., Zaanen, M. van, & Waes, L. V. (2021, August 24). Early prediction of writing quality using keystroke logging - International Journal of Artificial Intelligence in Education. SpringerLink. https://link.springer.com/article/10.1007/s40593-021-00268-w

[2] Chen, L. (2021, December 7). Basic ensemble learning (Random Forest, AdaBoost, Gradient boosting)- step by step explained. Medium. https://towardsdatascience.com/basic-ensemble-learning-random-forest-adaboost-gradient-boosting-step-by-step-explained-95d49d1e2725

[3] Kaggle: your machine learning and data science community. (n.d.). https://kaggle.com/

[4] TensorFlow. (n.d.). TensorFlow. https://www.tensorflow.org/