Name: HUANG Shaohang, PENG Tianze

EIDs: shuang293, tianzpeng2

Kaggle Competition: Linking Writing Processes to Writing Quality

Kaggle Team Name: shaohanghuang1999

CS5489 - Course Project (2023A)

Due date: See canvas site.

Possible Projects

For the course project, you may select **one** of the following competitions on Kaggle **or** define your own course project:

LLM - Detect AI Generated Text: Identify which essay was written by a large language model

> In recent years, large language models (LLMs) have become increasingly sophisticated, capable of generating text that is difficult to distinguish from human-written text. In this competition, we hope to foster open research and transparency on AI detection techniques applicable in the real world.
>
> This competition challenges participants to develop a machine learning model that can accurately detect whether an essay was written by a student or an LLM. The competition dataset comprises a mix of student-written essays and essays generated by a variety of LLMs.

Optiver - Trading at the Close: Predict US stocks closing movements

> In this competition, you are challenged to develop a model capable of predicting the closing price movements for hundreds of Nasdaq listed stocks using data from the order book and the closing auction of the stock. Information from the auction can be used to adjust prices, assess supply and demand dynamics, and identify trading opportunities.

Linking Writing Processes to Writing Quality: Use typing behavior to predict essay quality identify which essay was written by a large language model

> The goal of this competition is to predict overall writing quality. Does typing behavior affect the outcome of an essay? You will develop a model trained on a large dataset of keystroke logs that have captured writing process features.
>
> Your work will help explore the relationship between learners' writing behaviors and writing performance, which could provide valuable insights for writing instruction, the development of automated writing evaluation techniques, and intelligent tutoring systems.

Child Mind Institute - Detect Sleep States: Detect sleep onset and wake from wrist-worn accelerometer data

> Your work will improve researchers' ability to analyze accelerometer data for sleep monitoring and enable them to conduct large-scale studies of sleep. Ultimately, the work of this competition could improve awareness and guidance surrounding the importance of sleep. The valuable insights into how environmental factors impact sleep, mood, and behavior can inform the development of personalized interventions and support systems tailored to the unique needs of each child.

Enefit - Predict Energy Behavior of Prosumers: Predict Prosumer Energy Patterns and Minimize Imbalance Costs.

> The goal of the competition is to create an energy prediction model of prosumers to reduce energy imbalance costs.
>
> This competition aims to tackle the issue of energy imbalance, a situation where the energy expected to be used doesn't line up with the actual energy used or produced. Prosumers, who both consume and generate energy, contribute a large part of the energy imbalance. Despite being only a small part of all consumers, their unpredictable energy use causes logistical and financial problems for the energy companies.

Student-defined Course Project

The goal of the student-defined project is to get some hands-on experience using the course material on your own research problems. Keep in mind that there will only be about 4 weeks to do the project, so the scope should not be too large. Following the major themes of the course, here are some general topics for the project:

- *regression* (supervised learning) - use regression methods (e.g. ridge regression, Gaussian processes) to model data or predict from data.
- *classification* (supervised learning) - use classification methods (e.g., SVM, BDR, Logistic Regression, NNs) to learn to distinguish between multiple classes given a feature vector.
- *clustering* (unsupervised learning) - use clustering methods (e.g., K-means, EM, Mean-Shift) to discover the natural groups in data.

- *visualization* (unsupervised learning) - use dimensionality reduction methods (e.g., PCA, kernel-PCA, non-linear embedding) to visualize the structure of high-dimensional data.

You can pick any one of these topics and apply them to your own problem/data.

- *Can my project be my recently submitted or soon-to-be submitted paper?* If you plan to just turn in the results from your paper, then the answer is no. The project cannot be be work that you have already done. However, your course project can be based on extending your work. For example, you can try some models introduced in the course on your data/problem.

Before actually doing the project, you need to write a **project proposal** so that we can make sure the project is doable within the 3-4 weeks. I can also give you some pointers to relevant methods, if necessary.

- The project proposal should be at most one page with the following contents: 1) an introduction that briefy states the problem; 2) a precise description of what you plan to do - e.g., What types of features do you plan to use? What algorithms do you plan to use? What dataset will you use? How will you evaluate your results? How do you define a good outcome for the project?
- The goal of the proposal is to work out, in your head, what your project will be. Once the proposal is done, it is just a matter of implementation!
- *You need to submit the project proposal to Canvas 1 week after the Course project is released.*

Groups

Group projects should contain 2 students. To sign up for a group, go to Canvas and under "People", join an existing **"Project Group X"**, where X is a number. *For group projects, the project report must state the percentage contribution from each project member. You must also submit the contribution percentages to the "Project Group Contribution" assignment on Canvas.*

Methodology

You are free to choose the methodology to solve the task. In machine learning, it is important to use domain knowledge to help solve the problem. Hence, instead of blindly applying the algorithms to the data you need to think about how to represent the data in a way that makes sense for the algorithm to solve the task.

Kaggle: Kaggle Notebooks

The Kaggle competitions have Kaggle Notebooks enabled, which provide free GPU/TPU computing resources (up to a limit). You can develop your model in the Kaggle Notebook, CS5489 JupyterHub (Dive), or on your own computers.

Kaggle: Evaluation on Kaggle

For Kaggle projects, the final evaluation will be performed on Kaggle. Note that for these competitions you need to submit your code via the Kaggle Notebook, which will then generate the submission file for processing.

Project Presentation

Each project group needs to give a presentation at the end of the semester. You will record your presentation and upload it to FlipGrid. The presentation is limited to 5 minutes. You *must* give a presentation. See the details in the "Project Presentations" Canvas assignment.

What to hand in

You need to turn in the following things.

The following files should be uploaded to "Course Project" on Canvas:

1. This ipynb file `CourseProject-2023A.ipynb` with your source code and documentation. **You should write about all the various attempts that you make to find a good solution.** You may also submit .py files, but your documentation should be in the ipynb file.
2. A **PDF** version of your ipynb file.
3. Presentation slides.
4. (Kaggle projects) Your final submission file to Kaggle. Note that most competitions require you to submit the code, and Kaggle will run it on the hidden test set.
5. (Kaggle projects) A downloaded copy of your Kaggle Notebook that is submitted to Kaggle. This file should contain the code that generates the final submission file on Kaggle. This code will be used to verify that your Kaggle submission is reproducible.

Other things that need to be turned in:

- Upload your Project presentation to FlipGrid and the submit the URL to the "Project Presentations" assignment on Canvas. See the detailed instructions in the assignment.
- Enter the percentage contribution for each project member using the "Project Group Contribution" assignment on Canvas.
- (Student-defined projects only) submit your project proposal to the "Project Proposal" assignment on Canvas. The project proposal is due 1 week after the course project is released. Kaggle projects do not need to submit a proposal.

Grading

The marks of the assignment are distributed as follows:

- 40% - Results using various feature representations, dimensionality reduction methods, classifiers, etc.
- 25% - Trying out feature representations (e.g. adding additional features, combining features from different sources) or methods not used in the tutorials.
- 15% - Quality of the written report. More points for insightful observations and analysis.
- 15% - Project presentation
- 5% - For Kaggle projects, final ranking on the Kaggle leaderboard; For student-defined projects, the project proposal.

Late Penalty: 25 marks will be subtracted for each day late.

Group contribution: marks for a group member with less than equal contribution will be deducted according to the following formula:

- Let A% and B% be the percentage contributions for group members Alice and Bob. A%+B%=100%
- Let x be the group project marks.
- If A>B, then Bob's marks will be reduced to be: x*B/A

YOUR METHODS HERE

BASIC EXPLORATION

The first step is doing some basic exploration of the data. After a simple feature extraction, we will try different models and see which one is the best.

Data preparation

import packages

In [ ]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import sklearn
from numpy import *
from sklearn import *
import tensorflow as tf
import tensorflow.keras as keras
import xgboost as xgb
```

read data

In [ ]:
```
train_logs = pd.read_csv('train_logs.csv')
test_logs = pd.read_csv('test_logs.csv')
```

```
train_scores = pd.read_csv('train_scores.csv')
test_logs.head()
```

Out[ ]:

	id	event_id	down_time	up_time	action_time	activity	down_event	up_eve
0	0000aaaa	1	338433	338518	85	Input	Space	Spa
1	0000aaaa	2	760073	760160	87	Input	Space	Spa
2	2222bbbb	1	711956	712023	67	Input	q	
3	2222bbbb	2	290502	290548	46	Input	q	
4	4444cccc	1	635547	635641	94	Input	Space	Spa

We can see each data record presents an operation of a session, and the letters are anonymized. The data is already sorted by session id and timestamp.

Considering the letters are anonymized, We don't think we should restore the article first.

We can count the number of sessions and the number of operations in the data.

In [ ]:

```
train_data_representation = pd.DataFrame()
test_data_representation = pd.DataFrame()

train_data_representation['op_cnt'] = train_logs.groupby('id')['event_id'].count
train_data_representation['op_time_avg'] = train_logs.groupby('id')['action_time
train_data_representation['input_cnt'] = train_logs[train_logs['activity'] == 'I
train_data_representation['q_cnt'] = train_logs[train_logs['down_event'] == 'q']

# add word_cnt column
train_data_representation['word_cnt'] = 0
i = -1
for index, row in train_data_representation.iterrows():
    start_time = train_logs.loc[int(i+1)]['down_time']
    i += row['op_cnt']
    end_time = train_logs.loc[int(i)]['up_time']
    train_data_representation.loc[index, 'word_cnt'] = train_logs.loc[int(i)]['w
    train_data_representation.loc[index, 'total_time'] = (end_time - start_time)

test_data_representation['op_cnt'] = test_logs.groupby('id')['event_id'].count()
test_data_representation['op_time_avg'] = test_logs.groupby('id')['action_time']
test_data_representation['input_cnt'] = test_logs[test_logs['activity'] == 'Inpu
test_data_representation['q_cnt'] = test_logs[test_logs['down_event'] == 'q'].gr
# add word_cnt column
test_data_representation['word_cnt'] = 0
i = -1
for index, row in test_data_representation.iterrows():
    start_time = test_logs.loc[int(i+1)]['down_time']
    i += row['op_cnt']
    end_time = test_logs.loc[int(i)]['up_time']
    test_data_representation.loc[index, 'word_cnt'] = test_logs.loc[int(i)]['wor
    test_data_representation.loc[index, 'total_time'] = (end_time - start_time)


train_data_representation.head()
```

Out[ ]:

	op_cnt	op_time_avg	input_cnt	q_cnt	word_cnt	total_time
id						
001519c8	2557	116.246774	2010	1619	255	1797443.0
0022f953	2454	112.221271	1938	1490	320	1758346.0
0042269b	4136	101.837766	3515	2904	404	1767228.0
0059420b	1556	121.848329	1304	1038	206	1363074.0
0075873a	2531	123.943896	1942	1541	252	1584002.0

From the records, we have extracted some features.

In [ ]:
```
train_labels = train_scores['score']
```

Next, we normalize the data. As we tested in Kaggle, the normalization can improve the performance a lot.

In [ ]:
```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
train_data_normalized = scaler.fit_transform(train_data_representation)
test_data_normalized = scaler.transform(test_data_representation)
train_data_normalized = np.nan_to_num(train_data_normalized)
test_data_normalized = np.nan_to_num(test_data_normalized)
```

In [ ]:
```
from sklearn.model_selection import train_test_split
(trainX, testX, trainY, testY) = train_test_split(train_data_normalized, train_l
```

In [ ]:
```
print(trainX.shape)
print(trainY.shape)
```

```
(1976, 6)
(1976,)
```

MODELING

Linear Regression

The results of linear models may not be good, but we can use them as a baseline.

Ridge Regression

In [ ]:
```
# ridge regression
alphas = logspace(-6, -1, 100)
rr = linear_model.RidgeCV(alphas=alphas, cv=5)
rr.fit(trainX, trainY)

# print the RMSE for the best alpha value
print("Best alpha value for Ridge Regression: ", rr.alpha_)
print("Train RMSE", sqrt(metrics.mean_squared_error(trainY, rr.predict(trainX)))
print("Test RMSE", sqrt(metrics.mean_squared_error(testY, rr.predict(testX))))
```

```
Best alpha value for Ridge Regression:  0.08902150854450393
Train RMSE 0.777888382179744
Test RMSE 0.6869368968560372
```

OLS Regression

In [ ]:

```
# ols regression
ols = linear_model.LinearRegression()
ols.fit(trainX, trainY)

print("Train RMSE", sqrt(metrics.mean_squared_error(trainY, ols.predict(trainX))
print("Test RMSE", sqrt(metrics.mean_squared_error(testY, ols.predict(testX))))
```

```
Train RMSE 0.7770165691114228
Test RMSE 0.6942635843727947
```

Results shows that the linear models' training scores are even better than the validation scores, which means the models can't model the data well.

Nonlinear Regression

Kernel RR Regression

In [ ]:

```
paramgrid = {'alpha': logspace(-6, 0, 10),
             'gamma': logspace(-6, 0, 10)}

krrcv = model_selection.GridSearchCV(estimator=kernel_ridge.KernelRidge(kernel='
                                     param_grid=paramgrid, cv=5)
krrcv.fit(trainX, trainY)

print("Best alpha value for Kernel Ridge Regression: ", krrcv.best_params_['alph
print("Best gamma value for Kernel Ridge Regression: ", krrcv.best_params_['gamm
print("Train RMSE", sqrt(metrics.mean_squared_error(trainY, krrcv.predict(trainX
print("Test RMSE", sqrt(metrics.mean_squared_error(testY, krrcv.predict(testX)))
```

```
Best alpha value for Kernel Ridge Regression:  0.00046415888336127773
Best gamma value for Kernel Ridge Regression:  0.21544346900318823
Train RMSE 0.6992387228262781
Test RMSE 0.6181728139634615
```

SVR Regression

In [ ]:

```
paramgrid = {'C': logspace(-3, 3, 7),
             'gamma': logspace(-3, 3, 7)}

svrcv = model_selection.GridSearchCV(estimator=svm.SVR(kernel='rbf'),
                                     param_grid=paramgrid, cv=5)
svrcv.fit(trainX, trainY)

print("Best C value for SVM Regression: ", svrcv.best_params_['C'])
print("Best gamma value for SVM Regression: ", svrcv.best_params_['gamma'])
print("Train RMSE", sqrt(metrics.mean_squared_error(trainY, svrcv.predict(trainX
print("Test RMSE", sqrt(metrics.mean_squared_error(testY, svrcv.predict(testX)))
```

```
Best C value for SVM Regression:  1.0
Best gamma value for SVM Regression:  10.0
Train RMSE 0.6944586963959372
Test RMSE 0.6159155181275916
```

Random Forest Regression

In [ ]:
```
# random forest regression
paramgrid = {'n_estimators': [300, 400, 500],
             'max_depth': [10, 20, 30]}
rfr = ensemble.RandomForestRegressor()
rfrcv = model_selection.GridSearchCV(estimator=rfr,
                                     param_grid=paramgrid, cv=5)
rfrcv.fit(trainX, trainY)

print("Best n_estimators value for Random Forest Regression: ", rfrcv.best_param
print("Best max_depth value for Random Forest Regression: ", rfrcv.best_params_[
print("Train RMSE", sqrt(metrics.mean_squared_error(trainY, rfrcv.predict(trainX
print("Test RMSE", sqrt(metrics.mean_squared_error(testY, rfrcv.predict(testX)))
```

```
Best n_estimators value for Random Forest Regression:  400
Best max_depth value for Random Forest Regression:  10
Train RMSE 0.4552165610037448
Test RMSE 0.6366798782797369
```

XGBoost Regression

In [ ]:
```
# xgboost with grid search
paramgrid = {'max_depth': [3, 5, 7, 9, 11],
             'n_estimators': [50, 100, 200, 300, 400, 500]}
xgbcv = model_selection.GridSearchCV(estimator=xgb.XGBRegressor(),
                                     param_grid=paramgrid, cv=5)
xgbcv.fit(trainX, trainY)

print("Best max_depth value for XGBoost: ", xgbcv.best_params_['max_depth'])
print("Best n_estimators value for XGBoost: ", xgbcv.best_params_['n_estimators'
print("Train RMSE", sqrt(metrics.mean_squared_error(trainY, xgbcv.predict(trainX
print("Test RMSE", sqrt(metrics.mean_squared_error(testY, xgbcv.predict(testX)))
```

```
Best max_depth value for XGBoost:  3
Best n_estimators value for XGBoost:  50
Train RMSE 0.5938718005393737
Test RMSE 0.6513365176862582
```

Kernel RR Regression and SVR Regression show the same problem as linear models - testing scores are better than validation scores. Maybe because of the outliers.

Random Forest Regression and XGBoost Regression have better performance.

Overall, the nonlinear models have better performance than linear models.

NEW METHODS

Classification

We notice that the scores are all 0.5 times a integer. So we can try classifiers.

MLP

In [ ]:

```
# turn the scores into labels
tags = train_scores['score'].unique()

label_nums = {tag: num for num, tag in enumerate(tags)}
print(label_nums)

trainY_labels_cl = train_scores['score'].map(label_nums)
print(trainY_labels_cl)
```

```
{3.5: 0, 6.0: 1, 2.0: 2, 4.0: 3, 4.5: 4, 2.5: 5, 5.0: 6, 3.0: 7, 1.5: 8, 5.5: 9,
1.0: 10, 0.5: 11}
0       0
1       0
2       1
3       2
4       3
       ..
2466    0
2467    3
2468    8
2469    6
2470    3
Name: score, Length: 2471, dtype: int64
```

In [ ]:

```
K = keras.backend
# MLP
K.clear_session()
model = keras.models.Sequential()
model.add(keras.layers.Dense(256, activation='sigmoid', input_shape=(train_data_
model.add(keras.layers.Dense(128, activation='sigmoid'))
model.add(keras.layers.Dense(64, activation='sigmoid'))
model.add(keras.layers.Dense(32, activation='sigmoid'))
model.add(keras.layers.Dense(len(tags), activation='softmax'))
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=
model.summary()

history = model.fit(train_data_normalized, trainY_labels_cl, epochs=100, batch_s

predMLP = model.predict(train_data_normalized)
predMLP = np.argmax(predMLP, axis=1)
predMLP = [tags[i] for i in predMLP]
print("RMSE for MLP: ", sqrt(metrics.mean_squared_error(train_labels, predMLP)))
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 256)               1792

 dense_1 (Dense)             (None, 128)               32896

 dense_2 (Dense)             (None, 64)                8256

 dense_3 (Dense)             (None, 32)                2080

 dense_4 (Dense)             (None, 12)                396

=================================================================
Total params: 45,420
Trainable params: 45,420
Non-trainable params: 0
_________________________________________________________________
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 256)               1792

 dense_1 (Dense)             (None, 128)               32896

 dense_2 (Dense)             (None, 64)                8256

 dense_3 (Dense)             (None, 32)                2080

 dense_4 (Dense)             (None, 12)                396

=================================================================
Total params: 45,420
Trainable params: 45,420
Non-trainable params: 0
_________________________________________________________________
Epoch 1/100
62/62 [==============================] - 1s 7ms/step - loss: 2.2476 - accuracy:
0.1665 - val_loss: 2.1459 - val_accuracy: 0.2081
Epoch 2/100
62/62 [==============================] - 0s 3ms/step - loss: 2.1372 - accuracy:
0.1913 - val_loss: 2.1196 - val_accuracy: 0.2081
Epoch 3/100
62/62 [==============================] - 0s 3ms/step - loss: 2.1259 - accuracy:
0.1883 - val_loss: 2.1154 - val_accuracy: 0.2081
Epoch 4/100
62/62 [==============================] - 0s 3ms/step - loss: 2.1257 - accuracy:
0.1959 - val_loss: 2.1154 - val_accuracy: 0.2081
Epoch 5/100
62/62 [==============================] - 0s 3ms/step - loss: 2.1227 - accuracy:
0.1913 - val_loss: 2.1196 - val_accuracy: 0.2081
Epoch 6/100
62/62 [==============================] - 0s 3ms/step - loss: 2.1208 - accuracy:
0.1933 - val_loss: 2.1192 - val_accuracy: 0.2343
Epoch 7/100
62/62 [==============================] - 0s 3ms/step - loss: 2.1241 - accuracy:
0.1913 - val_loss: 2.1134 - val_accuracy: 0.2343
Epoch 8/100
62/62 [==============================] - 0s 3ms/step - loss: 2.1207 - accuracy:
```

```
0.1908 - val_loss: 2.1166 - val_accuracy: 0.2081
Epoch 9/100
62/62 [==============================] - 0s 3ms/step - loss: 2.1207 - accuracy:
0.1969 - val_loss: 2.1123 - val_accuracy: 0.2343
Epoch 10/100
62/62 [==============================] - 0s 3ms/step - loss: 2.1222 - accuracy:
0.1776 - val_loss: 2.1104 - val_accuracy: 0.2343
Epoch 11/100
62/62 [==============================] - 0s 3ms/step - loss: 2.1193 - accuracy:
0.1964 - val_loss: 2.1100 - val_accuracy: 0.2081
Epoch 12/100
62/62 [==============================] - 0s 3ms/step - loss: 2.1107 - accuracy:
0.2065 - val_loss: 2.1011 - val_accuracy: 0.2303
Epoch 13/100
62/62 [==============================] - 0s 3ms/step - loss: 2.0569 - accuracy:
0.2439 - val_loss: 2.0228 - val_accuracy: 0.2182
Epoch 14/100
62/62 [==============================] - 0s 3ms/step - loss: 1.9407 - accuracy:
0.2819 - val_loss: 1.9453 - val_accuracy: 0.2242
Epoch 15/100
62/62 [==============================] - 0s 3ms/step - loss: 1.8698 - accuracy:
0.2966 - val_loss: 1.8946 - val_accuracy: 0.2646
Epoch 16/100
62/62 [==============================] - 0s 3ms/step - loss: 1.8438 - accuracy:
0.2981 - val_loss: 1.8737 - val_accuracy: 0.2626
Epoch 17/100
62/62 [==============================] - 0s 4ms/step - loss: 1.8129 - accuracy:
0.3021 - val_loss: 1.8411 - val_accuracy: 0.2909
Epoch 18/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7893 - accuracy:
0.3107 - val_loss: 1.8264 - val_accuracy: 0.2646
Epoch 19/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7793 - accuracy:
0.3102 - val_loss: 1.8337 - val_accuracy: 0.2727
Epoch 20/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7741 - accuracy:
0.3158 - val_loss: 1.8111 - val_accuracy: 0.2949
Epoch 21/100
62/62 [==============================] - 0s 5ms/step - loss: 1.7630 - accuracy:
0.3097 - val_loss: 1.8104 - val_accuracy: 0.2828
Epoch 22/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7620 - accuracy:
0.3148 - val_loss: 1.8366 - val_accuracy: 0.2970
Epoch 23/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7603 - accuracy:
0.3057 - val_loss: 1.8115 - val_accuracy: 0.3030
Epoch 24/100
62/62 [==============================] - 0s 4ms/step - loss: 1.7533 - accuracy:
0.3077 - val_loss: 1.8016 - val_accuracy: 0.2768
Epoch 25/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7534 - accuracy:
0.3158 - val_loss: 1.8064 - val_accuracy: 0.2586
Epoch 26/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7470 - accuracy:
0.3122 - val_loss: 1.8000 - val_accuracy: 0.2768
Epoch 27/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7421 - accuracy:
0.3117 - val_loss: 1.8023 - val_accuracy: 0.2747
Epoch 28/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7392 - accuracy:
```

```
0.3148 - val_loss: 1.8048 - val_accuracy: 0.2808
Epoch 29/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7409 - accuracy:
0.3254 - val_loss: 1.8079 - val_accuracy: 0.2768
Epoch 30/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7441 - accuracy:
0.3209 - val_loss: 1.7951 - val_accuracy: 0.2707
Epoch 31/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7353 - accuracy:
0.3148 - val_loss: 1.7971 - val_accuracy: 0.2828
Epoch 32/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7339 - accuracy:
0.3249 - val_loss: 1.7978 - val_accuracy: 0.2687
Epoch 33/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7332 - accuracy:
0.3325 - val_loss: 1.7967 - val_accuracy: 0.2808
Epoch 34/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7324 - accuracy:
0.3203 - val_loss: 1.8152 - val_accuracy: 0.2990
Epoch 35/100
62/62 [==============================] - 0s 2ms/step - loss: 1.7348 - accuracy:
0.3254 - val_loss: 1.7976 - val_accuracy: 0.2768
Epoch 36/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7312 - accuracy:
0.3310 - val_loss: 1.8081 - val_accuracy: 0.2970
Epoch 37/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7337 - accuracy:
0.3254 - val_loss: 1.7918 - val_accuracy: 0.2768
Epoch 38/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7398 - accuracy:
0.3138 - val_loss: 1.8052 - val_accuracy: 0.2788
Epoch 39/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7348 - accuracy:
0.3168 - val_loss: 1.7959 - val_accuracy: 0.3131
Epoch 40/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7243 - accuracy:
0.3264 - val_loss: 1.7908 - val_accuracy: 0.2929
Epoch 41/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7263 - accuracy:
0.3310 - val_loss: 1.7905 - val_accuracy: 0.2747
Epoch 42/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7294 - accuracy:
0.3224 - val_loss: 1.7936 - val_accuracy: 0.2970
Epoch 43/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7275 - accuracy:
0.3320 - val_loss: 1.7888 - val_accuracy: 0.2848
Epoch 44/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7280 - accuracy:
0.3300 - val_loss: 1.7903 - val_accuracy: 0.2889
Epoch 45/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7254 - accuracy:
0.3224 - val_loss: 1.7864 - val_accuracy: 0.2667
Epoch 46/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7254 - accuracy:
0.3148 - val_loss: 1.8018 - val_accuracy: 0.2687
Epoch 47/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7285 - accuracy:
0.3310 - val_loss: 1.7905 - val_accuracy: 0.2929
Epoch 48/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7296 - accuracy:
```

```
0.3163 - val_loss: 1.8075 - val_accuracy: 0.2970
Epoch 49/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7298 - accuracy:
0.3229 - val_loss: 1.7925 - val_accuracy: 0.2768
Epoch 50/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7230 - accuracy:
0.3198 - val_loss: 1.7943 - val_accuracy: 0.2606
Epoch 51/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7220 - accuracy:
0.3254 - val_loss: 1.7881 - val_accuracy: 0.2788
Epoch 52/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7221 - accuracy:
0.3295 - val_loss: 1.7849 - val_accuracy: 0.3010
Epoch 53/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7233 - accuracy:
0.3300 - val_loss: 1.7917 - val_accuracy: 0.3091
Epoch 54/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7233 - accuracy:
0.3209 - val_loss: 1.7908 - val_accuracy: 0.2768
Epoch 55/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7262 - accuracy:
0.3198 - val_loss: 1.7894 - val_accuracy: 0.2909
Epoch 56/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7243 - accuracy:
0.3295 - val_loss: 1.8080 - val_accuracy: 0.3091
Epoch 57/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7233 - accuracy:
0.3269 - val_loss: 1.8085 - val_accuracy: 0.3111
Epoch 58/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7276 - accuracy:
0.3214 - val_loss: 1.7819 - val_accuracy: 0.2828
Epoch 59/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7201 - accuracy:
0.3284 - val_loss: 1.7846 - val_accuracy: 0.2970
Epoch 60/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7233 - accuracy:
0.3249 - val_loss: 1.7832 - val_accuracy: 0.2949
Epoch 61/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7172 - accuracy:
0.3219 - val_loss: 1.7872 - val_accuracy: 0.2747
Epoch 62/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7193 - accuracy:
0.3289 - val_loss: 1.7900 - val_accuracy: 0.3091
Epoch 63/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7189 - accuracy:
0.3264 - val_loss: 1.7823 - val_accuracy: 0.2768
Epoch 64/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7166 - accuracy:
0.3320 - val_loss: 1.7843 - val_accuracy: 0.2848
Epoch 65/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7166 - accuracy:
0.3254 - val_loss: 1.7828 - val_accuracy: 0.2949
Epoch 66/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7193 - accuracy:
0.3274 - val_loss: 1.7748 - val_accuracy: 0.3051
Epoch 67/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7178 - accuracy:
0.3330 - val_loss: 1.7905 - val_accuracy: 0.2949
Epoch 68/100
62/62 [==============================] - 0s 4ms/step - loss: 1.7142 - accuracy:
```

```
0.3310 - val_loss: 1.7825 - val_accuracy: 0.2808
Epoch 69/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7143 - accuracy:
0.3259 - val_loss: 1.7814 - val_accuracy: 0.3010
Epoch 70/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7192 - accuracy:
0.3259 - val_loss: 1.7962 - val_accuracy: 0.2747
Epoch 71/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7177 - accuracy:
0.3295 - val_loss: 1.7935 - val_accuracy: 0.3131
Epoch 72/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7153 - accuracy:
0.3320 - val_loss: 1.7786 - val_accuracy: 0.3010
Epoch 73/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7158 - accuracy:
0.3295 - val_loss: 1.7820 - val_accuracy: 0.2707
Epoch 74/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7166 - accuracy:
0.3209 - val_loss: 1.7954 - val_accuracy: 0.3232
Epoch 75/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7172 - accuracy:
0.3244 - val_loss: 1.7981 - val_accuracy: 0.3172
Epoch 76/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7145 - accuracy:
0.3244 - val_loss: 1.7831 - val_accuracy: 0.3010
Epoch 77/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7181 - accuracy:
0.3279 - val_loss: 1.7875 - val_accuracy: 0.3232
Epoch 78/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7190 - accuracy:
0.3203 - val_loss: 1.7871 - val_accuracy: 0.3051
Epoch 79/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7149 - accuracy:
0.3259 - val_loss: 1.7789 - val_accuracy: 0.2889
Epoch 80/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7116 - accuracy:
0.3325 - val_loss: 1.7785 - val_accuracy: 0.3010
Epoch 81/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7129 - accuracy:
0.3320 - val_loss: 1.7830 - val_accuracy: 0.2949
Epoch 82/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7113 - accuracy:
0.3320 - val_loss: 1.7783 - val_accuracy: 0.2970
Epoch 83/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7113 - accuracy:
0.3269 - val_loss: 1.7817 - val_accuracy: 0.2788
Epoch 84/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7187 - accuracy:
0.3264 - val_loss: 1.7760 - val_accuracy: 0.2929
Epoch 85/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7129 - accuracy:
0.3259 - val_loss: 1.7859 - val_accuracy: 0.3071
Epoch 86/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7137 - accuracy:
0.3295 - val_loss: 1.7779 - val_accuracy: 0.2889
Epoch 87/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7109 - accuracy:
0.3355 - val_loss: 1.7834 - val_accuracy: 0.2949
Epoch 88/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7178 - accuracy:
```

```
0.3284 - val_loss: 1.7768 - val_accuracy: 0.3212
Epoch 89/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7090 - accuracy:
0.3360 - val_loss: 1.7917 - val_accuracy: 0.2929
Epoch 90/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7159 - accuracy:
0.3305 - val_loss: 1.7835 - val_accuracy: 0.2949
Epoch 91/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7105 - accuracy:
0.3396 - val_loss: 1.7767 - val_accuracy: 0.2909
Epoch 92/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7207 - accuracy:
0.3203 - val_loss: 1.7927 - val_accuracy: 0.2970
Epoch 93/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7229 - accuracy:
0.3224 - val_loss: 1.7785 - val_accuracy: 0.2909
Epoch 94/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7127 - accuracy:
0.3315 - val_loss: 1.7815 - val_accuracy: 0.3010
Epoch 95/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7111 - accuracy:
0.3289 - val_loss: 1.7768 - val_accuracy: 0.2909
Epoch 96/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7095 - accuracy:
0.3229 - val_loss: 1.7794 - val_accuracy: 0.3091
Epoch 97/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7076 - accuracy:
0.3335 - val_loss: 1.7786 - val_accuracy: 0.2869
Epoch 98/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7087 - accuracy:
0.3335 - val_loss: 1.7784 - val_accuracy: 0.2990
Epoch 99/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7094 - accuracy:
0.3340 - val_loss: 1.7852 - val_accuracy: 0.2808
Epoch 100/100
62/62 [==============================] - 0s 3ms/step - loss: 1.7112 - accuracy:
0.3249 - val_loss: 1.7732 - val_accuracy: 0.2828
78/78 [==============================] - 0s 1ms/step
RMSE for MLP:  0.753314542860838
```

In [ ]:

```python
# SVM with rbf kernel, Cross Validation
paramgrid = {'C': logspace(-3, 3, 7),
             'gamma': logspace(-3, 3, 7)}

svmcv = model_selection.GridSearchCV(estimator=svm.SVC(kernel='rbf'),
                                     param_grid=paramgrid, cv=5)
svmcv.fit(train_data_normalized, trainY_labels_cl)

predSVM = svmcv.predict(train_data_normalized)
predSVM = [tags[i] for i in predSVM]
print("Best C value for SVM: ", svmcv.best_params_)
print("RMSE for SVM: ", sqrt(metrics.mean_squared_error(train_labels, predSVM)))
```

```
Best C value for SVM:  {'C': 100.0, 'gamma': 1.0}
RMSE for SVM:  0.753583104203723
```

Classification shows bad results.

It may be because that if we use classification, we will get at least a 0.5 gap from the real score. This makes the RMSE large. The punishment is too large.

LightGBM

We noticed that LightGBM is a good model for this Kaggle competition. So we tried it. Hope it can improve the performance.

In [ ]:
```
# LGBM
from lightgbm import LGBMRegressor
lgbm = LGBMRegressor()
lgbm.fit(trainX, trainY)

predLGBM = lgbm.predict(trainX)
print("RMSE for LGBM: ", sqrt(metrics.mean_squared_error(trainY, predLGBM)))
predLGBM = lgbm.predict(testX)
print("Test RMSE for LGBM: ", sqrt(metrics.mean_squared_error(testY, predLGBM)))
```

```
RMSE for LGBM:  0.4674441903949796
Test RMSE for LGBM:  0.6473908685281795
```

This result is good. But when we submit the result to Kaggle, the score is not good. Even worse than the kernel RR regression. This may be because of the overfitting.

So we can try to tune the parameters.

LightGBM with parameter tuning

In [ ]:
```
param = {'n_estimators': 512,
         'learning_rate': 0.01,
         'metric': 'rmse',
         'random_state': 42,
         'force_col_wise': True,
         'verbosity': 0,}
lgbm = LGBMRegressor(**param)
lgbm.fit(trainX, trainY)
predLGBM = lgbm.predict(trainX)
print("RMSE for LGBM: ", sqrt(metrics.mean_squared_error(trainY, predLGBM)))
predLGBM = lgbm.predict(testX)
print("Test RMSE for LGBM: ", sqrt(metrics.mean_squared_error(testY, predLGBM)))
```

```
RMSE for LGBM:  0.548045646189349
Test RMSE for LGBM:  0.6277352675911901
```

The training score is a little worse, but the test score in Kaggle is as good as the LightGBM model without parameter tuning. It was a good try.

NEW FEATURE REPRESENTATIONS

After trying different models, we can try to improve the performance by adding new features.

Operation Count Representation

The first feature representation only represents one of the operations in a session. So we can try to represent all the operations in a session.

In [ ]:
```
events = ['q', 'Space', 'Backspace', 'Shift', 'ArrowRight', 'Leftclick', 'ArrowL


train_data_representation = pd.DataFrame()
test_data_representation = pd.DataFrame()

train_data_representation['op_cnt'] = train_logs.groupby('id')['event_id'].count
train_data_representation['op_time_avg'] = train_logs.groupby('id')['action_time
train_data_representation['input_cnt'] = train_logs[train_logs['activity'] == 'I

for event in events:
    train_data_representation[event+'_cnt'] = train_logs[train_logs['down_event'

# add word_cnt column
train_data_representation['word_cnt'] = 0
i = -1
for index, row in train_data_representation.iterrows():
    start_time = train_logs.loc[int(i+1)]['down_time']
    i += row['op_cnt']
    end_time = train_logs.loc[int(i)]['up_time']
    train_data_representation.loc[index, 'word_cnt'] = train_logs.loc[int(i)]['w
    train_data_representation.loc[index, 'total_time'] = (end_time - start_time)

test_data_representation['op_cnt'] = test_logs.groupby('id')['event_id'].count()
test_data_representation['op_time_avg'] = test_logs.groupby('id')['action_time']
test_data_representation['input_cnt'] = test_logs[test_logs['activity'] == 'Inpu

for event in events:
    test_data_representation[event+'_cnt'] = test_logs[test_logs['down_event'] =
# add word_cnt column
test_data_representation['word_cnt'] = 0
i = -1
for index, row in test_data_representation.iterrows():
    start_time = test_logs.loc[int(i+1)]['down_time']
    i += row['op_cnt']
    end_time = test_logs.loc[int(i)]['up_time']
    test_data_representation.loc[index, 'word_cnt'] = test_logs.loc[int(i)]['wor
    test_data_representation.loc[index, 'total_time'] = (end_time - start_time)


train_data_normalized = scaler.fit_transform(train_data_representation)
test_data_normalized = scaler.transform(test_data_representation)
train_data_normalized = np.nan_to_num(train_data_normalized)
test_data_normalized = np.nan_to_num(test_data_normalized)
train_data_representation.head()
```

Out[ ]:

id	op_cnt	op_time_avg	input_cnt	q_cnt	Space_cnt	Backspace_cnt	Shift_cnt
001519c8	2557	116.246774	2010	1619	357	417.0	27.0
0022f953	2454	112.221271	1938	1490	391	260.0	97.0
0042269b	4136	101.837766	3515	2904	552	439.0	39.0
0059420b	1556	121.848329	1304	1038	243	152.0	68.0
0075873a	2531	123.943896	1942	1541	324	517.0	39.0

5 rows × 21 columns

In [ ]:

```
(trainX, testX, trainY, testY) = train_test_split(train_data_normalized, train_l
print(trainX.shape)
```

```
(1976, 21)
```

Representation done!

Try some models with the new representation and see if the results are better.

In [ ]:

```
# ridge regression
alphas = logspace(-6, -1, 100)
rr = linear_model.RidgeCV(alphas=alphas, cv=5)
rr.fit(trainX, trainY)

# print the RMSE for the best alpha value
print("Best alpha value for Ridge Regression: ", rr.alpha_)
print("Train RMSE", sqrt(metrics.mean_squared_error(trainY, rr.predict(trainX)))
print("Test RMSE", sqrt(metrics.mean_squared_error(testY, rr.predict(testX))))
```

```
Best alpha value for Ridge Regression:  0.1
Train RMSE 0.7544704130448612
Test RMSE 0.6838390552793188
```

In [ ]:

```
# support vector regression
paramgrid = {'C': logspace(-3, 3, 7),
             'gamma': logspace(-3, 3, 7)}

svrcv = model_selection.GridSearchCV(estimator=svm.SVR(kernel='rbf'),
                                     param_grid=paramgrid, cv=5)
svrcv.fit(trainX, trainY)

print("Best C value for SVM Regression: ", svrcv.best_params_['C'])
print("Best gamma value for SVM Regression: ", svrcv.best_params_['gamma'])
print("Train RMSE", sqrt(metrics.mean_squared_error(trainY, svrcv.predict(trainX
print("Test RMSE", sqrt(metrics.mean_squared_error(testY, svrcv.predict(testX)))
```

```
Best C value for SVM Regression:  10.0
Best gamma value for SVM Regression:  1.0
Train RMSE 0.6475334592590074
Test RMSE 0.6162358702497944
```

In [ ]:

```
# random forest regression
paramgrid = {'n_estimators': [100, 200, 300, 400, 500],
```

```
             'max_depth': [10]}

rfr = ensemble.RandomForestRegressor()
rfrcv = model_selection.GridSearchCV(estimator=rfr,
                                     param_grid=paramgrid, cv=5)
rfrcv.fit(trainX, trainY)

print("Best n_estimators value for Random Forest Regression: ", rfrcv.best_param
print("Best max_depth value for Random Forest Regression: ", rfrcv.best_params_[
print("Train RMSE", sqrt(metrics.mean_squared_error(trainY, rfrcv.predict(trainX
print("Test RMSE", sqrt(metrics.mean_squared_error(testY, rfrcv.predict(testX)))
```

```
Best n_estimators value for Random Forest Regression:  400
Best max_depth value for Random Forest Regression:  10
Train RMSE 0.38766529897599067
Test RMSE 0.5878015269973187
```

In [ ]:
```
# xgboost with grid search
paramgrid = {'max_depth': [3, 5, 7, 9, 11],
             'n_estimators': [50, 100, 200, 300, 400, 500]}
xgbcv = model_selection.GridSearchCV(estimator=xgb.XGBRegressor(),
                                     param_grid=paramgrid, cv=5)
xgbcv.fit(trainX, trainY)

print("Best max_depth value for XGBoost: ", xgbcv.best_params_['max_depth'])
print("Best n_estimators value for XGBoost: ", xgbcv.best_params_['n_estimators'
print("Train RMSE", sqrt(metrics.mean_squared_error(trainY, xgbcv.predict(trainX
print("Test RMSE", sqrt(metrics.mean_squared_error(testY, xgbcv.predict(testX)))
```

```
Best max_depth value for XGBoost:  3
Best n_estimators value for XGBoost:  50
Train RMSE 0.5218485904326169
Test RMSE 0.603791474043163
```

In [ ]:
```
# lightgbm
lgbm = LGBMRegressor()
lgbm.fit(trainX, trainY)

predLGBM = lgbm.predict(trainX)
print("RMSE for LGBM: ", sqrt(metrics.mean_squared_error(trainY, predLGBM)))
predLGBM = lgbm.predict(testX)
print("Test RMSE for LGBM: ", sqrt(metrics.mean_squared_error(testY, predLGBM)))
```

```
RMSE for LGBM:  0.32269849084967533
Test RMSE for LGBM:  0.6059521627221561
```

Representation Exploration and Dimensionality Reduction

Due to the notebook length and the running time, the exploration is done in another notebook named "Representation_Exploration.ipynb".

Here we only show the dimensionality reduction and clustering results.

In [ ]:
```
train_data_fe = pd.read_csv('traindata_v1.csv')
train_data_fe_ids = train_data_fe['id']
train_data_fe = train_data_fe.drop(['id'], axis=1)
train_data_fe = scaler.fit_transform(train_data_fe)
train_data_fe = np.nan_to_num(train_data_fe)
```

```python
(trainXfe, testXfe, trainYfe, testYfe) = train_test_split(train_data_fe, train_l
```

In [ ]:
```python
print(trainXfe.shape)
```

```
(1976, 56)
```

PCA

In [ ]:
```python
# Do PCA
pca = decomposition.PCA(n_components=0.95)
pca.fit(trainXfe)
trainXfe = pca.transform(trainXfe)
testXfe = pca.transform(testXfe)
print(trainXfe.shape)
```

```
(1976, 21)
```

In [ ]:
```python
# krr
paramgrid = {'alpha': logspace(-6, 0, 10),
             'gamma': logspace(-6, 0, 10)}

krrcv = model_selection.GridSearchCV(estimator=kernel_ridge.KernelRidge(kernel='
                                     param_grid=paramgrid, cv=5)
krrcv.fit(trainXfe, trainYfe)

print("Best alpha value for Kernel Ridge Regression: ", krrcv.best_params_['alph
print("Best gamma value for Kernel Ridge Regression: ", krrcv.best_params_['gamm
print("Train RMSE", sqrt(metrics.mean_squared_error(trainYfe, krrcv.predict(trai
print("Test RMSE", sqrt(metrics.mean_squared_error(testYfe, krrcv.predict(testXf
```

```
Best alpha value for Kernel Ridge Regression:  0.01
Best gamma value for Kernel Ridge Regression:  0.046415888336127725
Train RMSE 0.6985249557766922
Test RMSE 0.6226452710185205
```

In [ ]:
```python
# xgboost with grid search
paramgrid = {'max_depth': [3, 5, 7, 9, 11],
             'n_estimators': [50, 100, 200, 300, 400, 500]}
xgbcv = model_selection.GridSearchCV(estimator=xgb.XGBRegressor(),
                                     param_grid=paramgrid, cv=5)
xgbcv.fit(trainXfe, trainYfe)

print("Best max_depth value for XGBoost: ", xgbcv.best_params_['max_depth'])
print("Best n_estimators value for XGBoost: ", xgbcv.best_params_['n_estimators'
print("Train RMSE", sqrt(metrics.mean_squared_error(trainYfe, xgbcv.predict(trai
print("Test RMSE", sqrt(metrics.mean_squared_error(testYfe, xgbcv.predict(testXf
```

```
Best max_depth value for XGBoost:  3
Best n_estimators value for XGBoost:  50
Train RMSE 0.5408028774241864
Test RMSE 0.6671041310364816
```

In [ ]:
```python
# lightgbm
from lightgbm import LGBMRegressor
param = {'n_estimators': 512,
         'learning_rate': 0.01,
         'metric': 'rmse',
         'random_state': 42,
         'force_col_wise': True,
```

```
              'verbosity': 0,}
lgbm = LGBMRegressor(**param)
lgbm.fit(trainXfe, trainYfe)
predLGBM = lgbm.predict(trainXfe)
print("RMSE for LGBM: ", sqrt(metrics.mean_squared_error(trainYfe, predLGBM)))
predLGBM = lgbm.predict(testXfe)
print("Test RMSE for LGBM: ", sqrt(metrics.mean_squared_error(testYfe, predLGBM)
```

```
RMSE for LGBM:  0.432775893909 01744
Test RMSE for LGBM:  0.6504263137361438
```

SUMMARY

Evaluation of the models:

- **Ridge Regression:** bad results, but very fast
- **OLS Regression:** bad results, but very fast
- **Kernel RR Regression:** ok results, medium speed
- **SVR Regression:** ok results, medium speed
- **Random Forest Regression:** good results, very slow
- **XGBoost Regression:** good results, fast
- **MLP Classification:** bad results, fast
- **SVM Classification:** bad results, slow
- **LightGBM:** good results, very fast

Evaluation of the feature representations:

- **Operation Count Representation:** a little better than the basic representation, gets the best results in the Kaggle competition. But it makes the model a little slower.
- **Dimensionality Reduction after Representation Exploration:** similar results as the basic representation, but it takes a lot of time to run the preprocessing.

Overall, the best model is LightGBM with the basic representation. The best model in Kaggle is LightGBM with parameter tuning.