COMP186: Foundations of Artificial Intelligence Individual Coursework

Authors: Sahan Bulathwela, Hossein A. (Saeed) Rahmani, Xiao Fu and Joshua Spear

Contact:

NB: Please do **not** discuss the Coursework in the forum or any other public medium. Please ask directly during office hours or any time via an email directed to the TA assigned to the part of the assignment. The tutor and the TAs will respond either via email or via a public announcement to all students.

If you have any questions/clarifications regarding the coursework, please contact the TA responsible for that part of the coursework **via email**.

- Part 1: Wiem Ben Rim (wiem.rim.23@ucl.ac.uk)
- Part 2: Xiao Fu (xiao.fu.20@ucl.ac.uk)
- Part 3: Lynn Kandakji (l.kandakji.22@ucl.ac.uk)
- General Clarifications: Sahan Bulathwela (m.bulathwela@ucl.ac.uk)

This coursework presents a real-world dataset to the learners where they are expected to systematically develop a model that can make good predictions. The coursework attempts to test both the theoretical and practical understanding of the learners regarding training machine learning models.

Coursework Structure

This coursework consists of three parts.

1. Exploratory data analysis and data preparation
2. Model training and evaluation
3. Demonstrating the theoretical understanding of a regression model

Parts 1 and 2 of the coursework involves multiple subtasks of building a machine learning model from data preparation to model evaluation. Part 3 systematically assists the learner to take their mathematical understanding of machine learning and build learning algorithms from scratch.

Guidelines to Providing Solutions

- This is an **INDIVIDUAL** coursework.
- The main questions are marked in red to improve visibility (e.e. Question x. x).
- This coursework consists of 3 parts where Part 1 and 2 carry 30 marks each and part 3 carries 40 marks.
- Each part will be marked **independently**. For example, Part 2 will be marked based solely on the code and answers provided within Part 2; answers from Part 1 or part 3 will not be considered.
- It is expected that learners provide solutions to **ALL** parts of the coursework **in this notebook itself**.
- The learners are expected to provide solutions in this Jupyter notebook itself (Both Code and text answers.).
- The solutions should be provided in the spaces provided. You may add new cells where it is necessary.
- Cells where answers are required in English text is marked with `Your Answer Here`
  - You can use markdown language to add formatting to your text. A cheat sheet is found here
  - Where you feel that mathematical notation is required, you can use latex syntax (e.g. `$x = 2^5$` : $x = 2^5$)
  - Alternatively, you are allowed to attach a image of your mathematical derivations.
- Cells where program code is expected, it is marked with `Your Code Here` .
  - You are expected to provide solutions in **Python** programming language
  - You should implement the code in a way that the function signature is preserved where the function skeleton is already provided (ie, mainly 1) function name 2) input parameters and 3) output parameters).
  - Where external datasets are used, use their **relative path** in the code. This simplifies reproducing results during assessment.
  - Use commenting (`# example comment here`) to describe the crucial steps in your programming code. This will help the examiner to understand your work.

Uploading Solutions

- It is expected that a **single** `.zip` file is uploaded as the solution.
- Zip the **same folder** that was provided as the assignment.

- The zipped directory should have the following files.
  - The completed assignment notebook (With Python code and English Text)
  - A PDF printout of the solutions notebook where all the output cells have been executed and the solution outputs are visible in the notebook. (**THIS IS NOT A SEPARATE PDF REPORT !!!**)
  - The `lectures_dataset.csv` dataset CSV file (in the same relative file location where the file can be loaded to the notebook by executing the relevant cell in the solution notebook.)
  - Any additional data files you generated that become input to your solutions (put the files in the relative file locations that will allow loading the files to the notebook to execute your solution.)

Video Lectures Dataset

This coursework works with a collection of video lectures. Different characteristics identified from the meta data, video data and transcripts of the lectures are included in the `lectures_dataset.csv` dataset.

In [1]:
```
import pandas as pd
import numpy as np



data_path = "lectures_dataset.csv"
lectures = pd.read_csv(data_path)
```

In [2]:
```
display(lectures)
```

	auxiliary_rate	conjugate_rate	normalization_rate	tobe_verb_rate	preposition_rate	pronoun_rate	document_entropy	easines
0	0.013323	0.033309	0.034049	0.035159	0.121392	0.089563	7.753995	75.58393
1	0.014363	0.030668	0.018763	0.036749	0.095885	0.103002	8.305269	86.87052
2	0.019028	0.033242	0.030720	0.037827	0.118294	0.124255	7.965583	81.91596
3	0.023416	0.042700	0.016873	0.046832	0.122590	0.104339	8.142877	80.14893
4	0.021173	0.041531	0.023412	0.038884	0.130700	0.102606	8.161250	76.90754
...	...	...	...	...	...	...	...	.
11543	0.014652	0.039166	0.031276	0.040011	0.111862	0.128487	7.781813	80.62023
11544	0.027689	0.036021	0.013967	0.048272	0.101936	0.135016	7.800766	94.08022
11545	0.015825	0.026280	0.018486	0.036117	0.106924	0.123509	8.235828	95.17307
11546	0.005900	0.053097	0.032448	0.050147	0.117994	0.094395	6.775492	74.05309
11547	0.011642	0.030643	0.023174	0.048435	0.137617	0.111477	7.880263	81.27275

11548 rows × 22 columns

In [3]:
```
print(lectures.columns)
```

```
Index(['auxiliary_rate', 'conjugate_rate', 'normalization_rate',
       'tobe_verb_rate', 'preposition_rate', 'pronoun_rate',
       'document_entropy', 'easiness', 'fraction_stopword_coverage',
       'fraction_stopword_presence', 'subject_domain', 'freshness',
       'title_word_count', 'word_count', 'most_covered_topic',
       'topic_coverage', 'duration', 'lecture_type', 'has_parts',
       'speaker_speed', 'silent_period_rate', 'median_engagement'],
      dtype='object')
```

- The dataset contains 11,548 observations 21 potential features and 1 label column. The label we are aiming to predict is `median_engagement` which can take a value between 0 and 1 where values close to 0 exhibit low engagement and values close to 1 indicate high engagement.

Description of Columns

The following table describes the columns in the dataset.

Variable Name	Type
auxiliary_rate	Fraction of auxiliary verbs in the transcript
conjugate_rate	Fraction of conjugates in the transcript
normalization_rate	Fraction of normalisation suffixes used in the transcript
tobe_verb_rate	Fraction of to-be-verbs in the transcript
preposition_rate	Fraction of prepositions in the transcript

pronoun_rate	Fraction of pronouns words in the transcript
document_entropy	Document entropy computed using word counts (Topic coherence)
easiness	The reading level of the transcript (level of English)
fraction_stopword_coverage	Fraction of unique stopwords used in the transcript
fraction_stopword_presence	Fraction of stopwords in the transcript
subject_domain	If the subject belongs to STEM or not.
freshness	How recently the video published
title_word_count	Number of words in the title
word_count	Number of words in the transcript
most_covered_topic	The Wikipedia URL of the most covered topic
topic_coverage	To what degree is the most covered topic covered
duration	Duration of the video
lecture_type	Type of lecture (e.g. lecture, tutorial, debate, discussion etc.)
has_parts	If the lecture is broken into multiple videos
speaker_speed	The word rate of the speaker (words per minute)
silent_period_rate	Fraction of Silence in the transcript where words are not spoken
median_engagement	Median % of video watched by all the viewers who watched it

Part 1: Exploratory Data Analysis and Feature Extraction (30 Marks)

This section attempts to understand the dataset before we jump into building a machine learning model.

In [ ]:
```
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.metrics import make_scorer
import seaborn as sns
import matplotlib.pyplot as plt
import math
import itertools
```

Question 1.1. What are the different data types each variable in the dataset belong to?

There are different data types different variables fall into. Based on these data types, we may handle these variables differently. In this question, you are expected to identify which data type each variable in the lecture dataset belongs to.

- Replace the `Your Answer Here` with your answer
- Possible values: Continuous, Discrete, Ordinal and Categorical

Variable Name	Type
auxiliary_rate	Continuous
conjugate_rate	Continuous
normalization_rate	Continuous
tobe_verb_rate	Continuous
preposition_rate	Continuous
pronoun_rate	Continuous
document_entropy	Continuous
easiness	Continuous
fraction_stopword_coverage	Continuous
fraction_stopword_presence	Continuous
subject_domain	Categorical
freshness	Discreet
title_word_count	Discreet
word_count	Discreet

most_covered_topic	Categorical
topic_coverage	Continuous
duration	Discreet
lecture_type	Categorical
has_parts	Categorical
speaker_speed	Continuous
silent_period_rate	Continuous
median_engagement	Continuous

In [5]:
```
print(lectures.dtypes)
```

```
auxiliary_rate               float64
conjugate_rate               float64
normalization_rate           float64
tobe_verb_rate               float64
preposition_rate             float64
pronoun_rate                 float64
document_entropy             float64
easiness                     float64
fraction_stopword_coverage   float64
fraction_stopword_presence   float64
subject_domain                object
freshness                      int64
title_word_count               int64
word_count                     int64
most_covered_topic            object
topic_coverage               float64
duration                       int64
lecture_type                  object
has_parts                     object
speaker_speed                float64
silent_period_rate           float64
median_engagement            float64
dtype: object
```

Question 1.2. Analyse the variables to understand them.

This question expects you to carry out `exploratory data analysis` on the dataset to understand the data and the value distributions better. This enables us to carry out specific pre-processing steps.

- List the analyses you would carry out with the features and the labels of the dataset. Justify why you think the proposed analyses are appropriate.
- Carry Out the Analyses you proposed.
  - **You are NOT permitted to use data analysis libraries that automatically run a brute-force set of analyses on the entire dataset. Usage of such libraries will be penalised.**
  - You may use visualisation libraries such as `matplotlib`, `plotly`, `seaborn` etc.
  - You may also use data processing libraries such as `pandas`, `numpy`, `scipy` etc.
  - You are expected to do as many analyses as you feel necessary to understand the data to make informed decisions about preprocessing.
  - You may use as many code cells as you deem necessary here to carry out your analysis. However, do not include analyses that are not meaningful for understanding the dataset (ones that you are unable to justify).
  - Use a markdown cell on top of the code cells to describe the analysis you are carrying out and its justification.

Choice of Analyses to be carried out with justification

1. Distribution of Features

   By plotting the distribution of each numeric feature we can better understand the scales of the data and the shape that they follow, this will help determine how to scale data best if needed at all. It can also help to identify where there may be missing data or outliers. For categorical features, it helps us find class imbalances. These discoveries let us better interpret later analysis and make informed decisions about data preparation.

2. Correlation Matrix

   A correlation matrix gives us an insight as to how numerical features impact each other. This is useful for finding redundant variables. If two variables have extremely high correlation it is likely we only need one to make predictions on the data. Removing these simplifies the dataset allowing for models to be simpler. We can also find features that have almost no correlation with the target. These are candidates for dropping as they can merely introduce noise into the data without actually being useful for prediction.

3. Distribution of Median Engagement in Categorical Features

   By plotting median engagement while grouping by categorical feature values, we can understand how different categorical values impact median engagement if at all. This is useful for determinging which features will actaully be useful indicators of median engagement.

4. Crosstabulation of Categorical Features

   Cross tabulation allows us to better understand the relationships between categorical features. They show how being a member of one categorical feature value impacts the likelihood of being in the values of another categorical feature. This can help find highly connected categorical features which could be redundant. These can then be addressed in the data preparation.

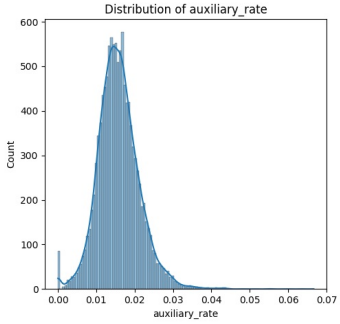
5. Distribution of Numeric Features in Categorical Features

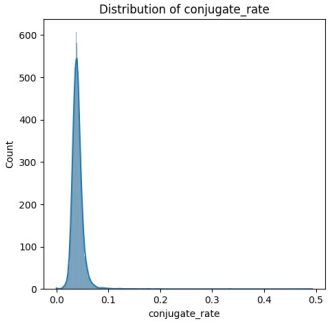
   Helps to understand the relationship between numeric and categorical features. This helps find possible redundancies between which can be addressed later in data preparation. If distributions of numeric features change drastically between values of catergorical features, these features could be adequately represented by a single one of them.

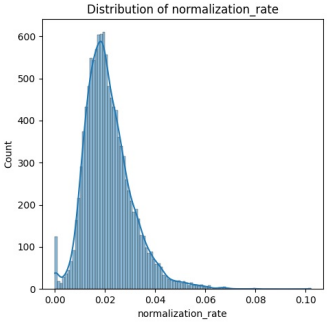
Analysis 1

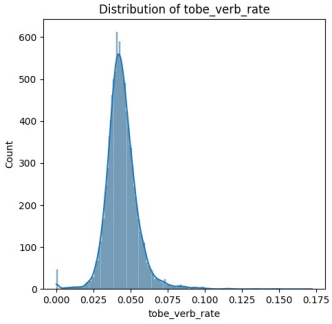
In [6]:
```
# plot distribution of all features
import warnings
warnings.filterwarnings('ignore')

num_cols = len(lectures.columns)
numeric_cols = lectures.select_dtypes(include=['float64', 'int64']).columns
categorical_cols = lectures.select_dtypes(include=['object']).columns
categorical_cols = categorical_cols.drop('most_covered_topic')

graph_cols = 4
graph_rows = int(np.ceil(num_cols / graph_cols))

fig, axs = plt.subplots(graph_rows, graph_cols, figsize=(20, graph_rows * 5))
axs = axs.flatten()

for i, col in enumerate(numeric_cols):
    row = i // graph_cols
    col_idx = i % graph_cols
    sns.histplot(data=lectures, x=col, ax=axs[i], kde=True)
    axs[i].set_title(f'Distribution of {col}')
    # print(f'{lectures[col].describe()}\n')

for i, col in enumerate(categorical_cols):
    row = (i + len(numeric_cols)) // graph_cols
    col_idx = (i + len(numeric_cols)) % graph_cols
    sns.countplot(data=lectures, x=col, ax=axs[i + len(numeric_cols)])
    axs[i + len(numeric_cols)].set_title(f'Distribution of {col}')
    axs[i + len(numeric_cols)].set_xticklabels(axs[i + len(numeric_cols)].get_xticklabels(), rotation=45, ha='r
    # print(f'{lectures[col].value_counts()}\n')

print(f'{lectures["most_covered_topic"].value_counts()}\n')

for ax in axs[len(numeric_cols) + len(categorical_cols):]:
    fig.delaxes(ax)

plt.tight_layout()
plt.show()
```

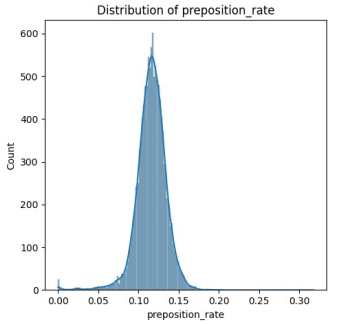
```
most_covered_topic
http://en.wikipedia.org/wiki/Time                1006
http://en.wikipedia.org/wiki/Scientific_method    612
http://en.wikipedia.org/wiki/Science              439
http://en.wikipedia.org/wiki/Algorithm            306
http://en.wikipedia.org/wiki/Technology           280
                                                 ...
http://en.wikipedia.org/wiki/Checkbox               1
http://en.wikipedia.org/wiki/Clique_problem         1
http://en.wikipedia.org/wiki/Confluence             1
http://en.wikipedia.org/wiki/Problem_solving        1
http://sl.wikipedia.org/wiki/Videoigra              1
Name: count, Length: 2096, dtype: int64
```

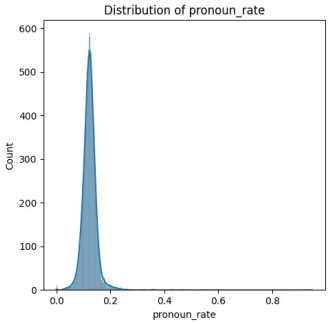


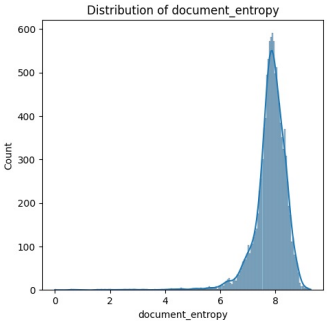


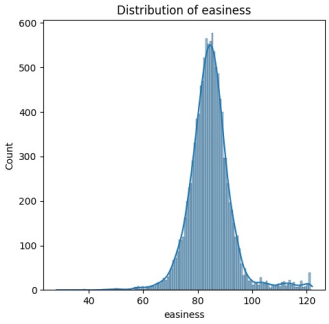


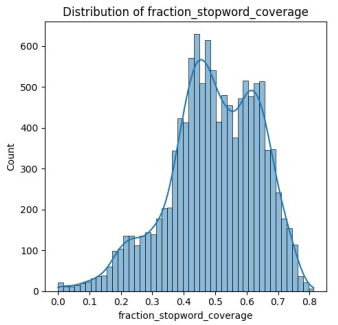


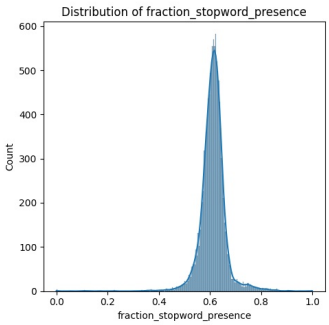


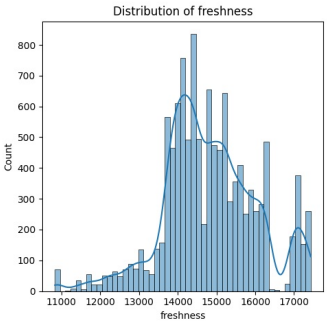


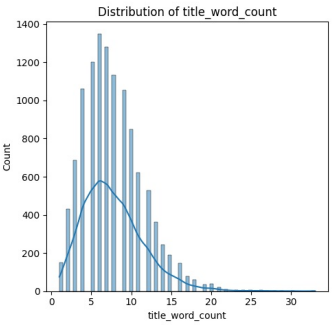


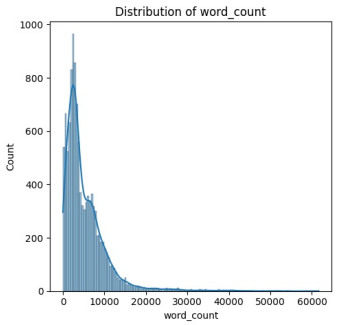


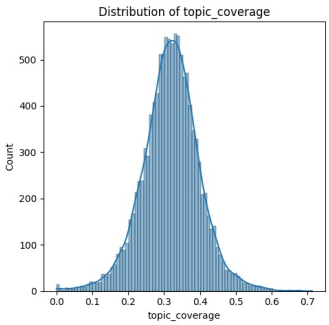


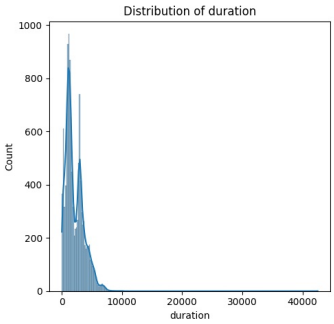


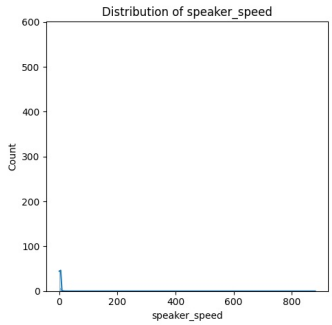


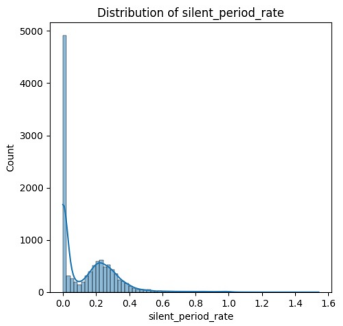


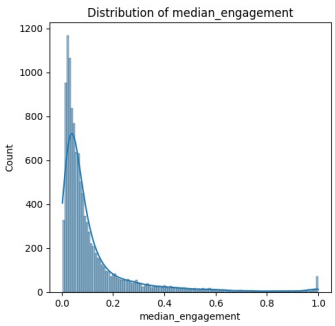


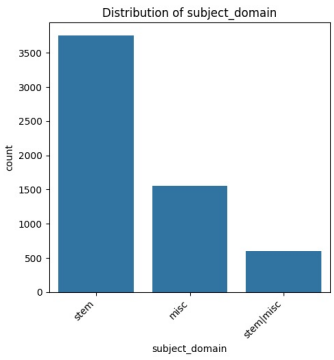


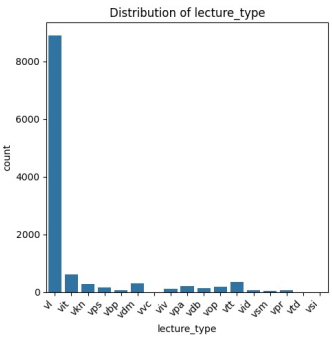


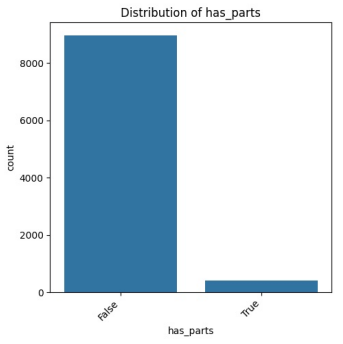








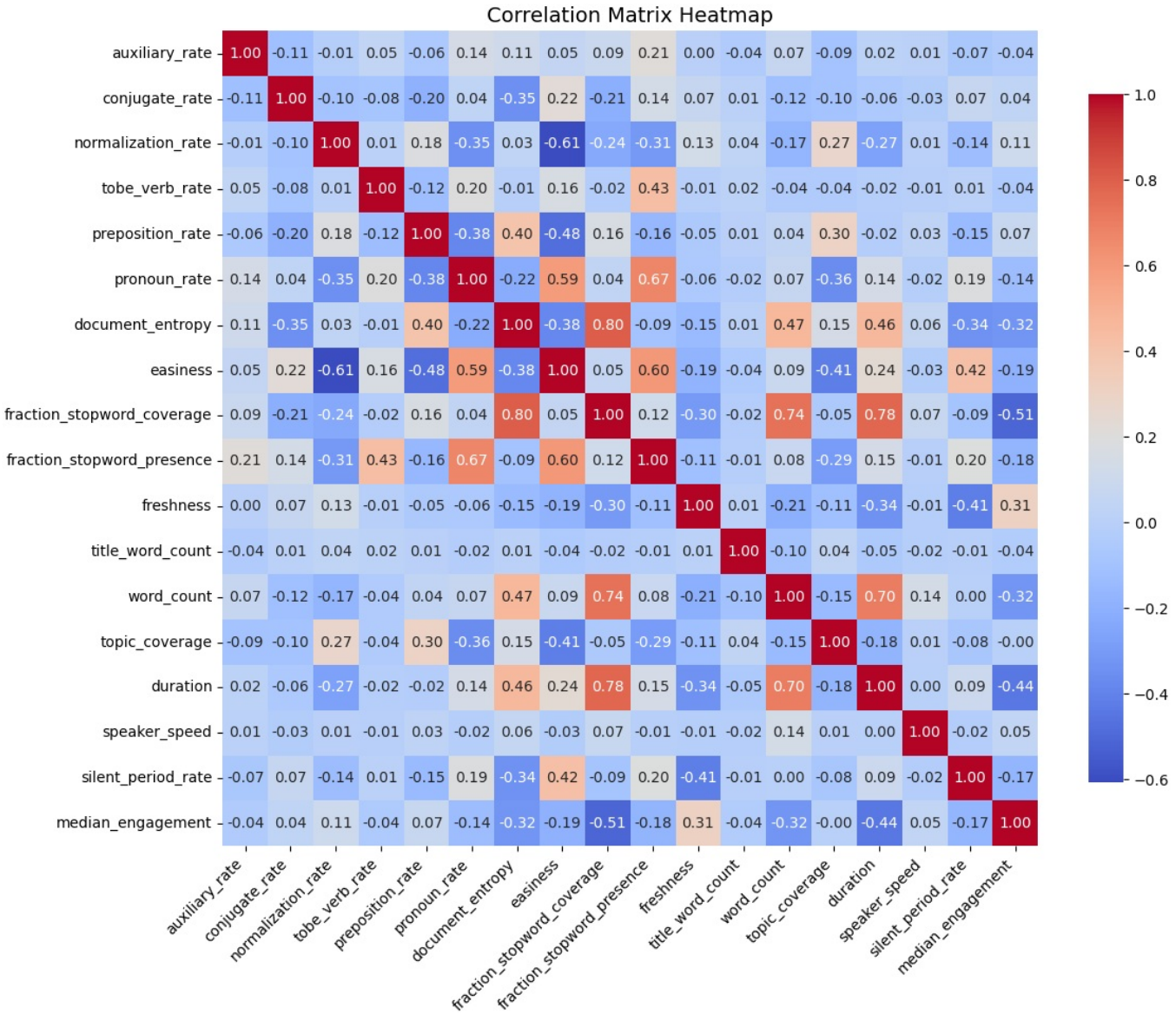




Analysis 2

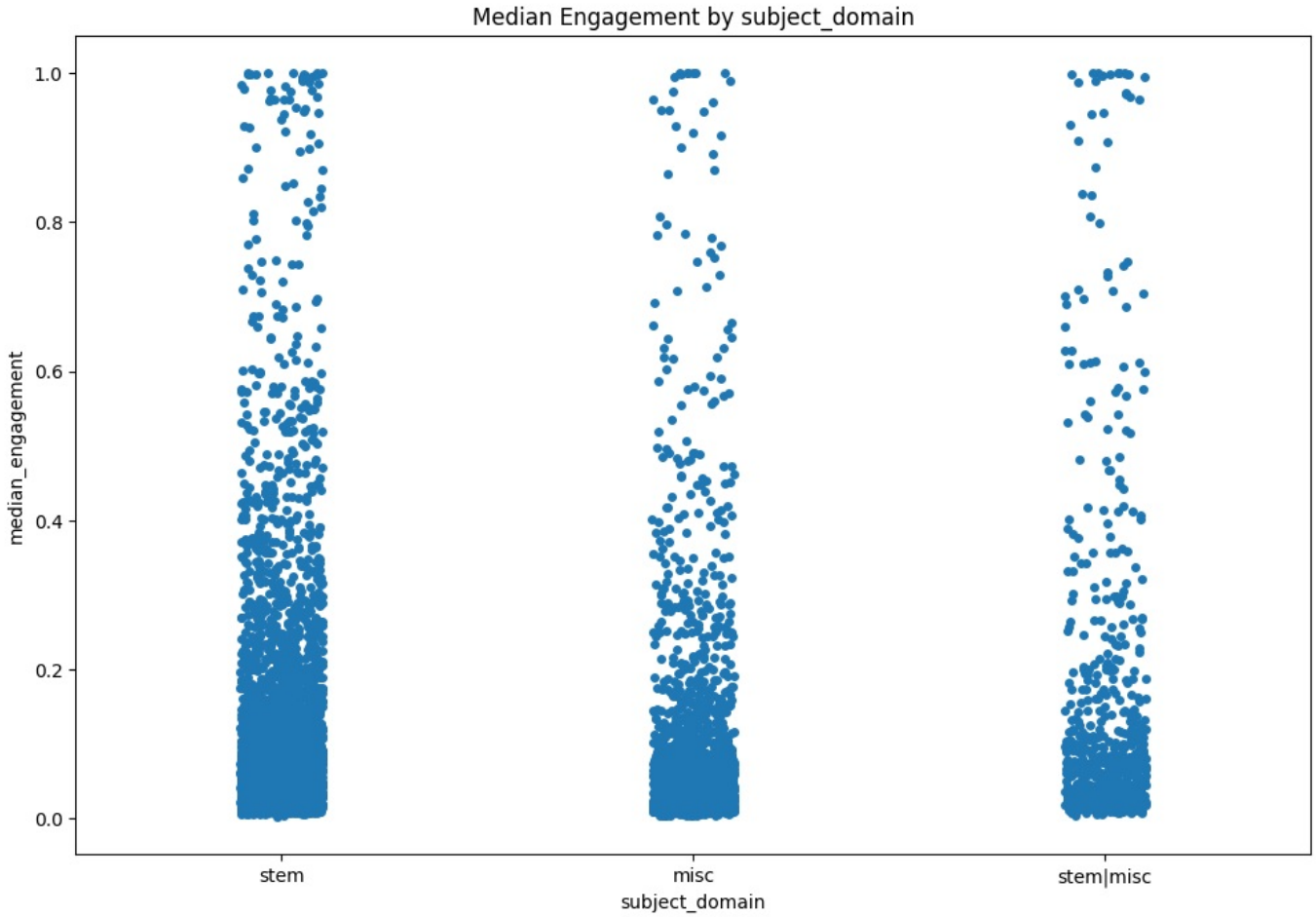
In [7]:
```
# Build a correlation matrix of numeric data
```

```
correlation_matrix = lectures[numeric_cols].corr()

plt.figure(figsize=(12, 10))

# Create heatmap with tweaks
sns.heatmap(
    correlation_matrix,
    annot=True,
    fmt=".2f",
    cmap="coolwarm",
    square=True,
    cbar_kws={"shrink": 0.8}
)

# Rotate axis labels for better readability
plt.xticks(rotation=45, ha="right", fontsize=10)
plt.yticks(fontsize=10)

# Title and show
plt.title("Correlation Matrix Heatmap", fontsize=14)
plt.tight_layout()  # Ensure labels fit
plt.show()
```

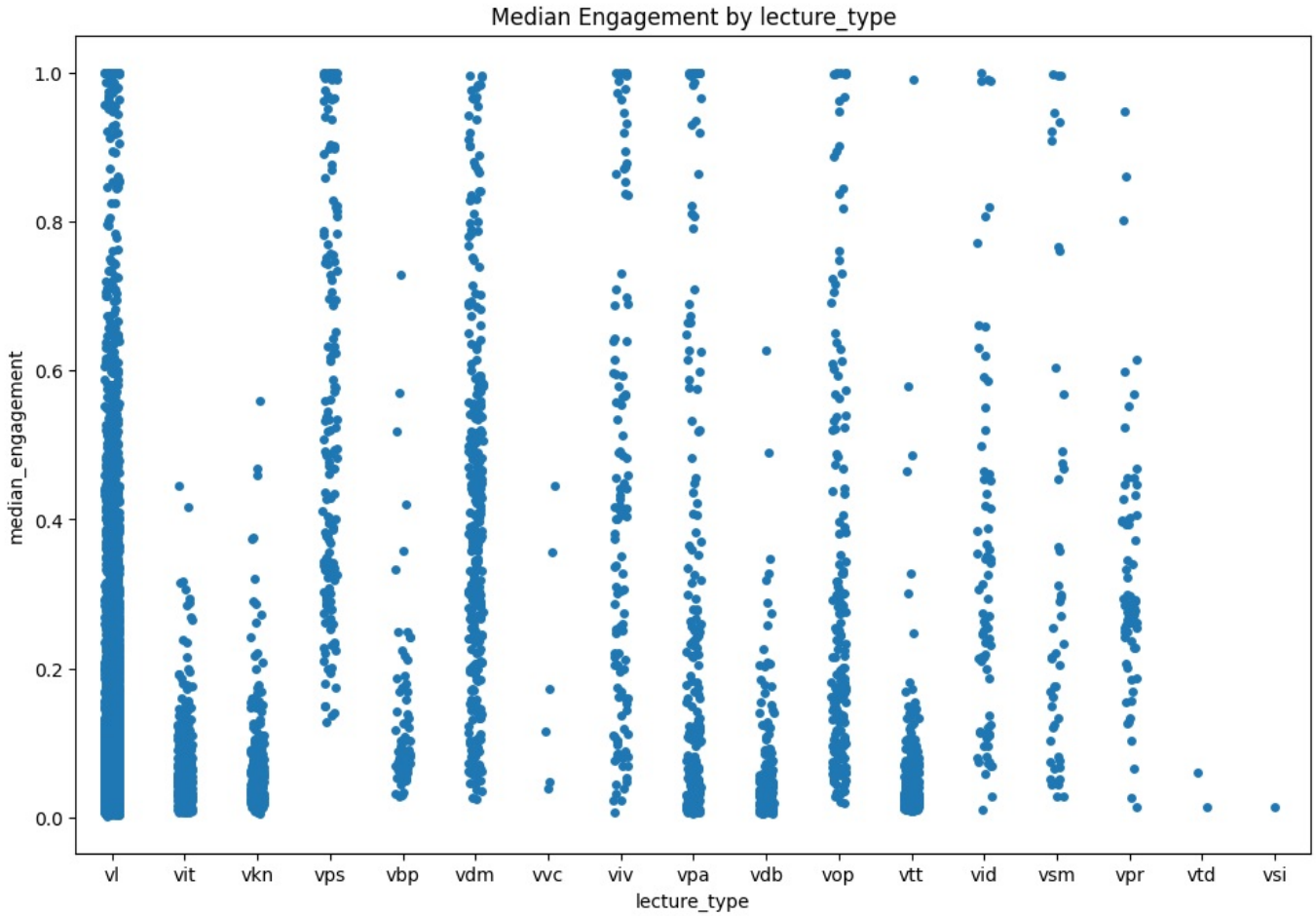


Analysis 3

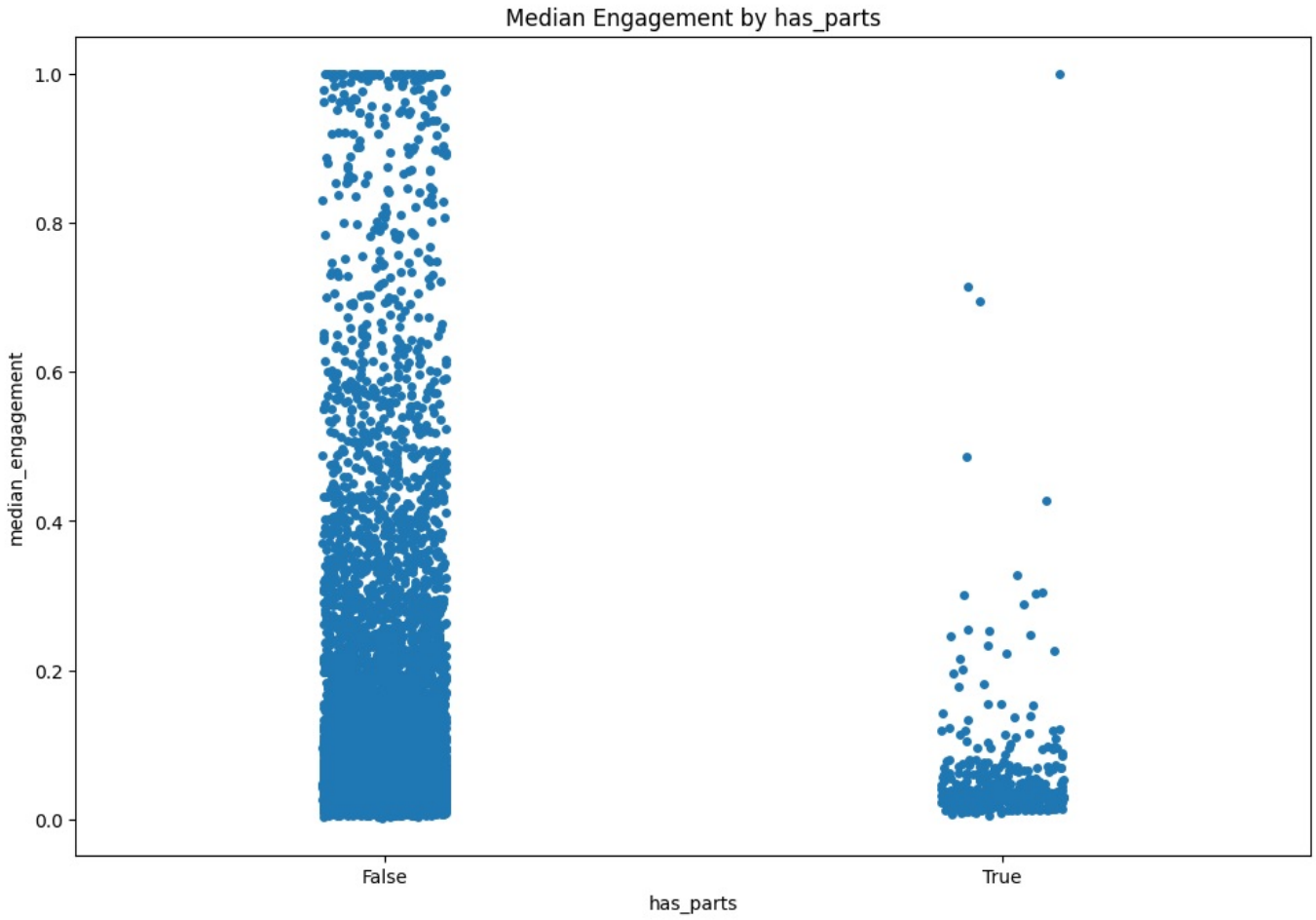
In [8]:
```
for cat_col in categorical_cols:
    plt.figure(figsize=(12, 8))
    if cat_col != 'most_covered_topic':
        sns.stripplot(data=lectures, x=cat_col, y='median_engagement')
        plt.title(f'Median Engagement by {cat_col}')
        plt.show()
    print(lectures.groupby(cat_col)['median_engagement'].mean())
    print()
```




```
subject_domain
misc         0.110278
stem         0.123663
stem|misc    0.170163
Name: median_engagement, dtype: float64
```



```
lecture_type
vbp    0.144168
vdb    0.072453
vdm    0.403549
vid    0.338019
vit    0.055854
viv    0.434905
vkn    0.065823
vl     0.096981
vop    0.296843
vpa    0.254797
vpr    0.325353
vps    0.554231
vsi    0.013199
vsm    0.342475
vtd    0.036951
vtt    0.050327
vvc    0.195983
Name: median_engagement, dtype: float64
```




```
has_parts
False    0.125287
True     0.055935
Name: median_engagement, dtype: float64
```

Analysis 4

In [9]:
```python
for col1, col2 in itertools.combinations(categorical_cols, 2):
    if 'most_covered_topic' not in [col1, col2]:
        # Create a contingency table
        contingency_table = pd.crosstab(lectures[col1], lectures[col2], margins=True)

        print(f"Contingency Table for {col1} and {col2}:")
        display(contingency_table)
        print()
```

```
Contingency Table for subject_domain and lecture_type:
```

lecture_type	vbp	vdb	vdm	vid	vit	viv	vkn	vl	vop	vpa	vpr	vps	vsi	vsm	vtt	vvc	All
subject_domain																	
misc	8	49	26	26	25	27	24	1271	44	27	13	2	0	9	1	0	1552
stem	18	22	120	11	268	11	107	2843	24	53	16	71	1	7	184	2	3758
stem|misc	4	7	23	5	29	25	20	390	42	26	1	13	0	10	7	1	603
All	30	78	169	42	322	63	151	4504	110	106	30	86	1	26	192	3	5913

Contingency Table for subject_domain and has_parts:

has_parts	False	True	All
subject_domain			
misc	1249	15	1264
stem	2905	187	3092
stem|misc	467	11	478
All	4621	213	4834

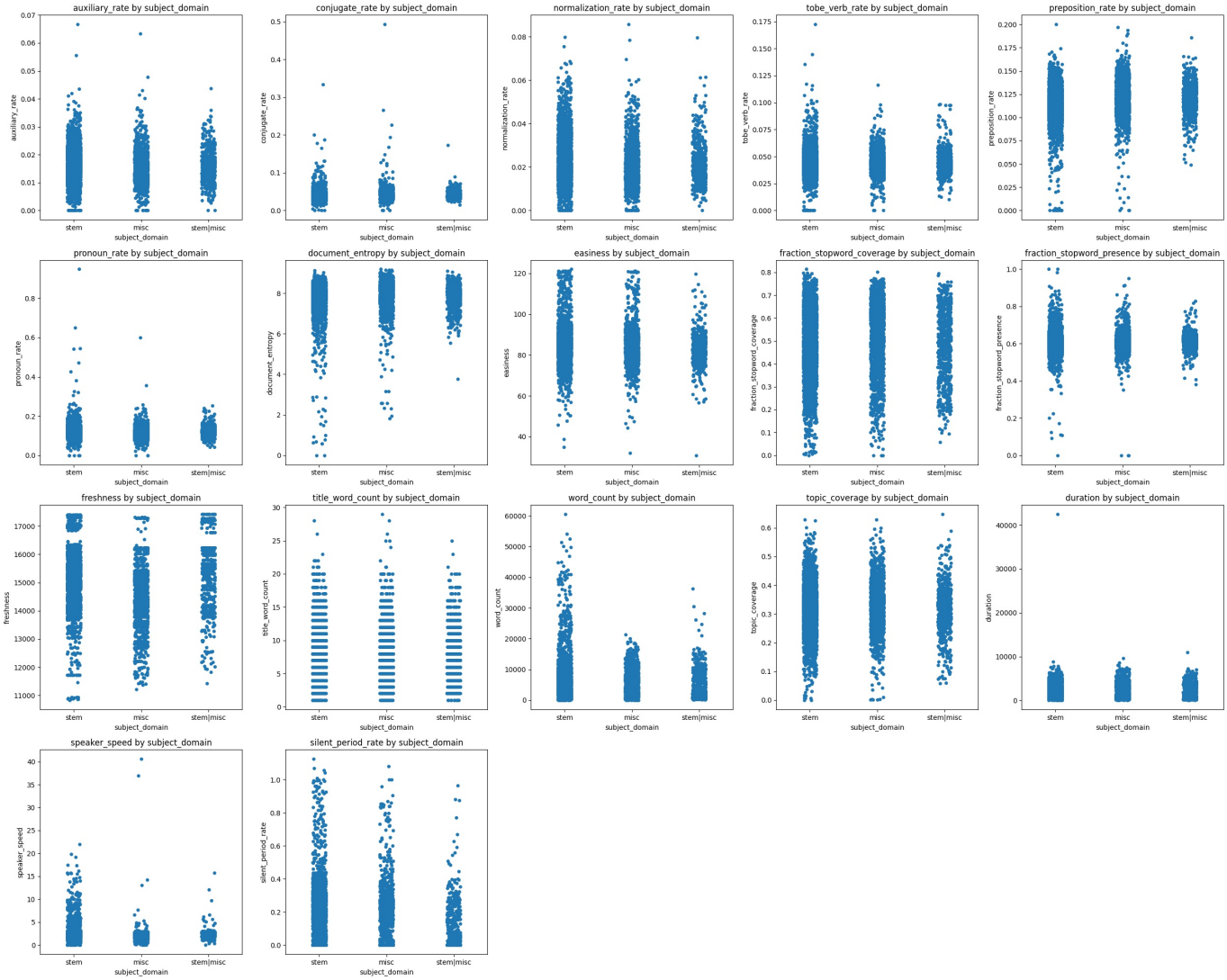
Contingency Table for lecture_type and has_parts:

has_parts	False	True	All
lecture_type			
vbp	57	0	57
vdb	115	1	116
vdm	272	2	274
vid	57	2	59
vit	477	7	484
viv	101	0	101
vkn	211	2	213
vl	7035	211	7246
vop	147	0	147
vpa	165	2	167
vpr	56	0	56
vps	130	0	130
vsi	1	0	1
vsm	40	0	40
vtd	2	0	2
vtt	104	193	297
vvc	6	0	6
All	8976	420	9396

Analysis 5

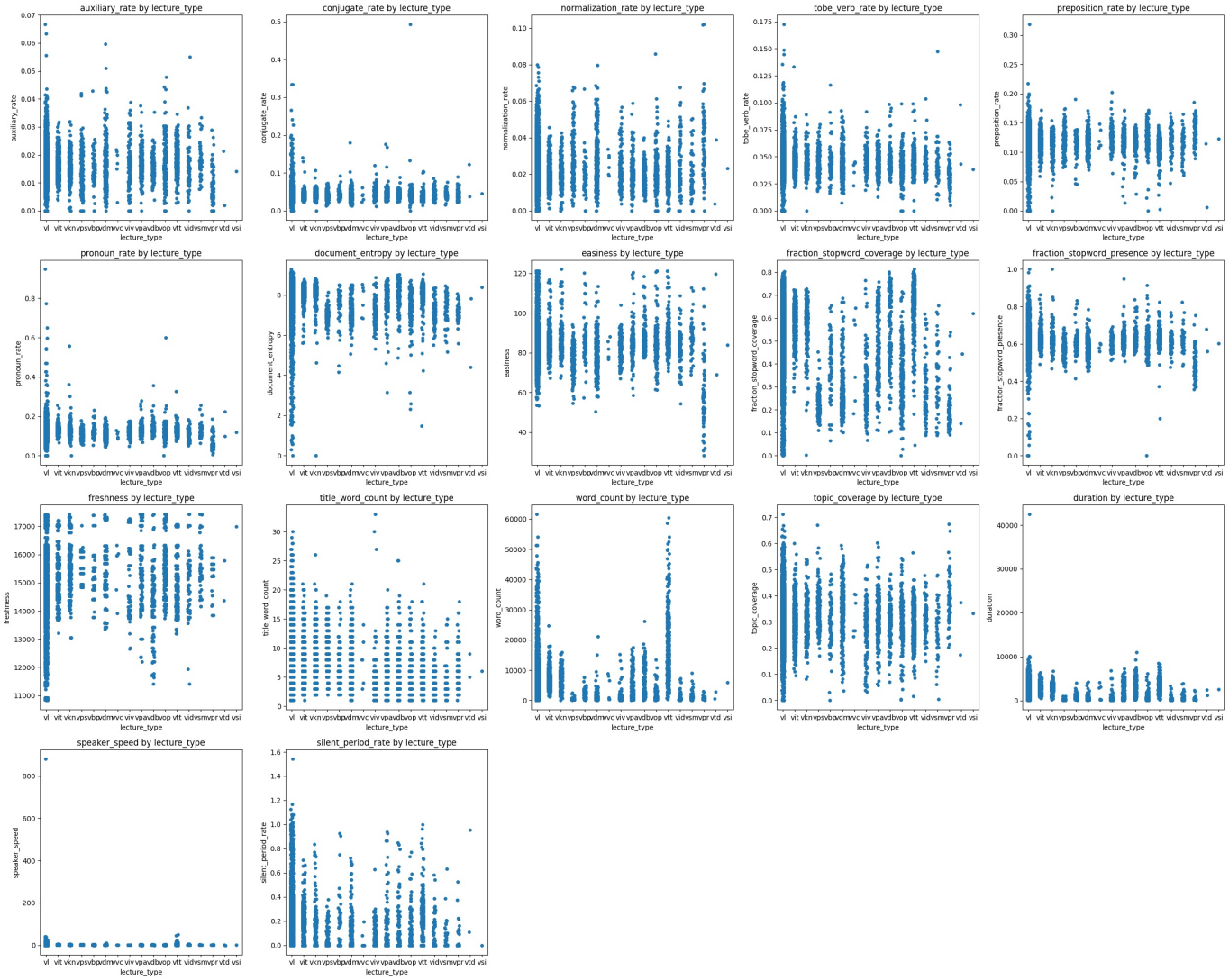
In [10]:
```
for cat_col in categorical_cols:
    if 'most_covered_topic' not in cat_col:
        filtered_numeric_cols = [col for col in numeric_cols if col != 'median_engagement']
        num_plots = len(filtered_numeric_cols)

        # shape of plot
        cols = math.ceil(math.sqrt(num_plots))
        rows = math.ceil(num_plots / cols)

        # Create the subplots
        fig, axes = plt.subplots(rows, cols, figsize=(5 * cols, 5 * rows), sharex=False, sharey=False)
        axes = axes.flatten()

        # Plot each numeric column in a subplot
        for i, num_col in enumerate(filtered_numeric_cols):
            sns.stripplot(data=lectures, x=cat_col, y=num_col, ax=axes[i])
            axes[i].set_title(f'{num_col} by {cat_col}')

        # Hide unused subplots
        for j in range(num_plots, len(axes)):
            fig.delaxes(axes[j])

        # Adjust layout
```

```
        plt.tight_layout()
        plt.show()

        # Print group means for each numeric column
        for num_col in filtered_numeric_cols:
            print(f"Means for {num_col} grouped by {cat_col}:")
            print(lectures.groupby(cat_col)[num_col].mean())
```




```
Means for auxiliary_rate grouped by subject_domain:
subject_domain
misc         0.014973
stem         0.016073
stem|misc    0.015946
Name: auxiliary_rate, dtype: float64
Means for conjugate_rate grouped by subject_domain:
subject_domain
misc         0.042965
stem         0.039979
stem|misc    0.042856
Name: conjugate_rate, dtype: float64
Means for normalization_rate grouped by subject_domain:
subject_domain
misc         0.020134
stem         0.021984
stem|misc    0.021037
Name: normalization_rate, dtype: float64
Means for tobe_verb_rate grouped by subject_domain:
subject_domain
misc         0.044619
stem         0.043664
stem|misc    0.044670
Name: tobe_verb_rate, dtype: float64
Means for preposition_rate grouped by subject_domain:
subject_domain
misc         0.121755
stem         0.113385
stem|misc    0.119637
Name: preposition_rate, dtype: float64
Means for pronoun_rate grouped by subject_domain:
subject_domain
misc         0.120962
stem         0.123190
```

```
stem|misc    0.123816
Name: pronoun_rate, dtype: float64
Means for document_entropy grouped by subject_domain:
subject_domain
misc         7.946639
stem         7.732744
stem|misc    7.798561
Name: document_entropy, dtype: float64
Means for easiness grouped by subject_domain:
subject_domain
misc         85.180473
stem         84.439222
stem|misc    83.059519
Name: easiness, dtype: float64
Means for fraction_stopword_coverage grouped by subject_domain:
subject_domain
misc         0.528181
stem         0.481601
stem|misc    0.475619
Name: fraction_stopword_coverage, dtype: float64
Means for fraction_stopword_presence grouped by subject_domain:
subject_domain
misc         0.612275
stem         0.611320
stem|misc    0.610926
Name: fraction_stopword_presence, dtype: float64
Means for freshness grouped by subject_domain:
subject_domain
misc         14426.256443
stem         15075.973922
stem|misc    15303.631841
Name: freshness, dtype: float64
Means for title_word_count grouped by subject_domain:
subject_domain
misc         7.595361
stem         7.652209
stem|misc    7.945274
Name: title_word_count, dtype: float64
Means for word_count grouped by subject_domain:
subject_domain
misc         5369.750000
stem         5388.445716
stem|misc    4790.766169
Name: word_count, dtype: float64
Means for topic_coverage grouped by subject_domain:
subject_domain
misc         0.327283
stem         0.315445
stem|misc    0.318035
Name: topic_coverage, dtype: float64
Means for duration grouped by subject_domain:
subject_domain
misc         2455.489691
stem         1974.119212
stem|misc    1987.031509
Name: duration, dtype: float64
Means for speaker_speed grouped by subject_domain:
subject_domain
misc         2.187120
stem         2.537748
stem|misc    2.333370
Name: speaker_speed, dtype: float64
Means for silent_period_rate grouped by subject_domain:
subject_domain
misc         0.128036
stem         0.147993
stem|misc    0.084543
Name: silent_period_rate, dtype: float64
```



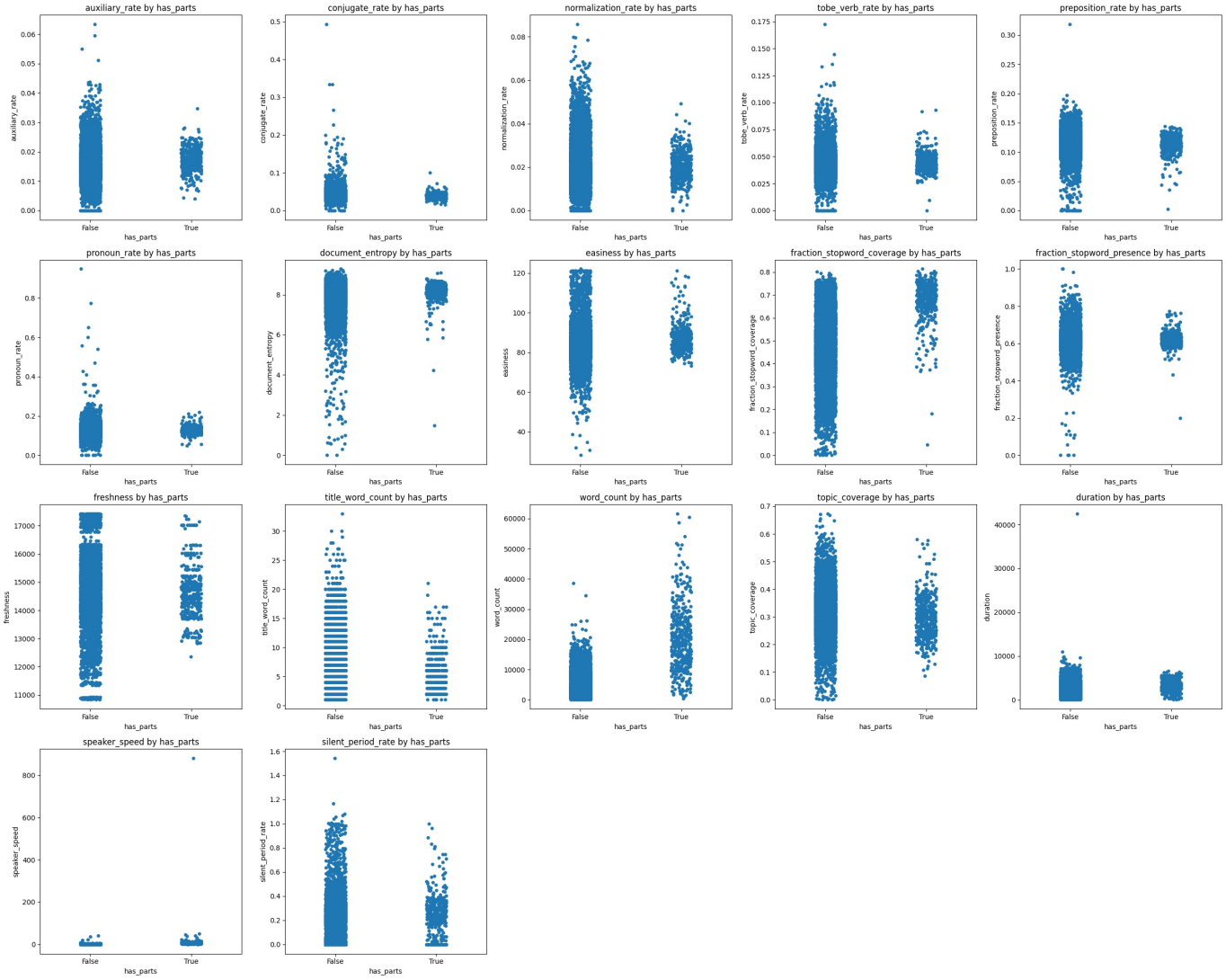

```
Means for auxiliary_rate grouped by lecture_type:
lecture_type
vbp    0.014673
vdb    0.015907
vdm    0.015554
vid    0.015945
vit    0.016097
viv    0.015761
vkn    0.015855
vl     0.015748
vop    0.016641
vpa    0.016693
vpr    0.008674
vps    0.014180
vsi    0.014158
vsm    0.017696
vtd    0.011647
vtt    0.018160
vvc    0.016300
Name: auxiliary_rate, dtype: float64
Means for conjugate_rate grouped by lecture_type:
lecture_type
vbp    0.043463
vdb    0.041822
vdm    0.039825
vid    0.045826
```

```
vit    0.039310
viv    0.047435
vkn    0.041238
vl     0.040741
vop    0.046819
vpa    0.043911
vpr    0.044580
vps    0.039336
vsi    0.045003
vsm    0.043503
vtd    0.080349
vtt    0.038676
vvc    0.040263
Name: conjugate_rate, dtype: float64
Means for normalization_rate grouped by lecture_type:
lecture_type
vbp    0.019083
vdb    0.019298
vdm    0.026499
vid    0.024859
vit    0.020400
viv    0.022496
vkn    0.019154
vl     0.021219
vop    0.021192
vpa    0.019823
vpr    0.037070
vps    0.026972
vsi    0.023091
vsm    0.019947
vtd    0.021255
vtt    0.019722
vvc    0.026022
Name: normalization_rate, dtype: float64
Means for tobe_verb_rate grouped by lecture_type:
lecture_type
vbp    0.045253
vdb    0.046136
vdm    0.043851
vid    0.045475
vit    0.044306
viv    0.046178
vkn    0.043630
vl     0.044215
vop    0.040321
vpa    0.043911
vpr    0.034739
vps    0.044460
vsi    0.038261
vsm    0.044155
vtd    0.070633
vtt    0.043798
vvc    0.039215
Name: tobe_verb_rate, dtype: float64
Means for preposition_rate grouped by lecture_type:
lecture_type
vbp    0.116035
vdb    0.114685
vdm    0.116509
vid    0.123235
vit    0.117478
viv    0.125717
vkn    0.114855
vl     0.115613
vop    0.122455
vpa    0.119075
vpr    0.137985
vps    0.120447
vsi    0.123041
vsm    0.112923
vtd    0.060255
vtt    0.111211
vvc    0.122855
Name: preposition_rate, dtype: float64
Means for pronoun_rate grouped by lecture_type:
lecture_type
vbp    0.125484
vdb    0.136454
vdm    0.109649
vid    0.112294
vit    0.127409
viv    0.123477
vkn    0.129545
```

```
vl     0.123162
vop    0.120595
vpa    0.132459
vpr    0.064881
vps    0.107964
vsi    0.119501
vsm    0.131776
vtd    0.160081
vtt    0.130271
vvc    0.105279
Name: pronoun_rate, dtype: float64
Means for document_entropy grouped by lecture_type:
lecture_type
vbp    7.456302
vdb    8.069509
vdm    7.153170
vid    7.089522
vit    8.090692
viv    7.106597
vkn    8.062995
vl     7.821221
vop    7.358553
vpa    7.738452
vpr    7.125986
vps    6.962049
vsi    8.373353
vsm    7.276435
vtd    6.106268
vtt    8.075058
vvc    7.710770
Name: document_entropy, dtype: float64
Means for easiness grouped by lecture_type:
lecture_type
vbp    86.709036
vdb    87.499740
vdm    79.186450
vid    82.511217
vit    84.695361
viv    82.417320
vkn    85.901004
vl     85.116769
vop    83.995020
vpa    84.821135
vpr    59.321424
vps    77.165216
vsi    83.930060
vsm    87.635273
vtd    94.236175
vtt    87.257484
vvc    81.376023
Name: easiness, dtype: float64
Means for fraction_stopword_coverage grouped by lecture_type:
lecture_type
vbp    0.399242
vdb    0.575558
vdm    0.276220
vid    0.287013
vit    0.587086
viv    0.294816
vkn    0.579611
vl     0.503293
vop    0.341526
vpa    0.468230
vpr    0.200461
vps    0.225922
vsi    0.620795
vsm    0.337827
vtd    0.292049
vtt    0.648497
vvc    0.429154
Name: fraction_stopword_coverage, dtype: float64
Means for fraction_stopword_presence grouped by lecture_type:
lecture_type
vbp    0.618161
vdb    0.628241
vdm    0.580199
vid    0.610227
vit    0.619527
viv    0.611333
vkn    0.617502
vl     0.613723
vop    0.604259
vpa    0.619624
```

```
vpr    0.483569
vps    0.575907
vsi    0.602225
vsm    0.621919
vtd    0.619779
vtt    0.625964
vvc    0.581832
Name: fraction_stopword_presence, dtype: float64
Means for freshness grouped by lecture_type:
lecture_type
vbp    15201.791045
vdb    14306.888889
vdm    15513.587302
vid    15145.733333
vit    15186.250000
viv    15682.975207
vkn    15677.116788
vl     14670.756623
vop    15477.210526
vpa    15640.483092
vpr    15444.202899
vps    15616.543210
vsi    16980.000000
vsm    15696.326531
vtd    15075.000000
vtt    14934.902507
vvc    15491.666667
Name: freshness, dtype: float64
Means for title_word_count grouped by lecture_type:
lecture_type
vbp    7.537313
vdb    6.407407
vdm    7.076190
vid    4.400000
vit    7.501645
viv    4.884298
vkn    7.952555
vl     7.995061
vop    4.315789
vpa    7.415459
vpr    6.014493
vps    9.104938
vsi    6.000000
vsm    4.265306
vtd    7.000000
vtt    5.509749
vvc    8.000000
Name: title_word_count, dtype: float64
Means for word_count grouped by lecture_type:
lecture_type
vbp     2329.641791
vdb     7260.881481
vdm     1041.879365
vid     1085.306667
vit     7387.521382
viv     1256.033058
vkn     7075.441606
vl      5131.124383
vop     1539.810526
vpa     4198.830918
vpr      588.652174
vps      501.561728
vsi     5933.000000
vsm     1581.734694
vtd     1717.500000
vtt    18177.214485
vvc     4190.666667
Name: word_count, dtype: float64
Means for topic_coverage grouped by lecture_type:
lecture_type
vbp    0.296105
vdb    0.299462
vdm    0.336804
vid    0.310318
vit    0.305589
viv    0.268517
vkn    0.303159
vl     0.321964
vop    0.299620
vpa    0.315310
vpr    0.389084
vps    0.348312
vsi    0.333113
```

```
vsm    0.263603
vtd    0.275198
vtt    0.280881
vvc    0.326960
Name: topic_coverage, dtype: float64
Means for duration grouped by lecture_type:
lecture_type
vbp    1098.805970
vdb    3335.111111
vdm     441.714286
vid     463.866667
vit    3048.914474
viv     570.661157
vkn    3043.029197
vl     2165.161652
vop     728.684211
vpa    1887.729469
vpr     303.188406
vps     209.135802
vsi    2600.000000
vsm     819.387755
vtd    1830.000000
vtt    3609.860724
vvc    1805.000000
Name: duration, dtype: float64
Means for speaker_speed grouped by lecture_type:
lecture_type
vbp    2.022295
vdb    2.072711
vdm    2.280880
vid    2.204590
vit    2.433568
viv    2.125763
vkn    2.312615
vl     2.428153
vop    2.073015
vpa    2.304660
vpr    2.115969
vps    2.401467
vsi    2.281923
vsm    1.977285
vtd    1.282448
vtt    5.565561
vvc    2.197683
Name: speaker_speed, dtype: float64
Means for silent_period_rate grouped by lecture_type:
lecture_type
vbp    0.155978
vdb    0.102050
vdm    0.090201
vid    0.085731
vit    0.143616
viv    0.090733
vkn    0.088102
vl     0.155808
vop    0.069228
vpa    0.089819
vpr    0.031996
vps    0.046158
vsi    0.000000
vsm    0.056015
vtd    0.533199
vtt    0.212905
vvc    0.045580
Name: silent_period_rate, dtype: float64
```




```
Means for auxiliary_rate grouped by has_parts:
has_parts
False    0.015771
True     0.017161
Name: auxiliary_rate, dtype: float64
Means for conjugate_rate grouped by has_parts:
has_parts
False    0.041018
True     0.037932
Name: conjugate_rate, dtype: float64
Means for normalization_rate grouped by has_parts:
has_parts
False    0.021362
True     0.020199
Name: normalization_rate, dtype: float64
Means for tobe_verb_rate grouped by has_parts:
has_parts
False    0.044069
True     0.044441
Name: tobe_verb_rate, dtype: float64
Means for preposition_rate grouped by has_parts:
has_parts
False    0.116197
```

```
True     0.113276
Name: preposition_rate, dtype: float64
Means for pronoun_rate grouped by has_parts:
has_parts
False    0.122849
True     0.128147
Name: pronoun_rate, dtype: float64
Means for document_entropy grouped by has_parts:
has_parts
False    7.773954
True     8.146719
Name: document_entropy, dtype: float64
Means for easiness grouped by has_parts:
has_parts
False    84.629371
True     87.321810
Name: easiness, dtype: float64
Means for fraction_stopword_coverage grouped by has_parts:
has_parts
False    0.485859
True     0.674326
Name: fraction_stopword_coverage, dtype: float64
Means for fraction_stopword_presence grouped by has_parts:
has_parts
False    0.611862
True     0.626631
Name: fraction_stopword_presence, dtype: float64
Means for freshness grouped by has_parts:
has_parts
False    14831.411542
True     14668.380952
Name: freshness, dtype: float64
Means for title_word_count grouped by has_parts:
has_parts
False    7.806818
True     5.616667
Name: title_word_count, dtype: float64
Means for word_count grouped by has_parts:
has_parts
False     4620.470254
True     20163.985714
Name: word_count, dtype: float64
Means for topic_coverage grouped by has_parts:
has_parts
False    0.319530
True     0.291172
Name: topic_coverage, dtype: float64
Means for duration grouped by has_parts:
has_parts
False    2080.454545
True     3290.976190
Name: duration, dtype: float64
Means for speaker_speed grouped by has_parts:
has_parts
False    2.208018
True     8.952169
Name: speaker_speed, dtype: float64
Means for silent_period_rate grouped by has_parts:
has_parts
False    0.141998
True     0.226225
Name: silent_period_rate, dtype: float64
```

First, looking at the distributions, we can see some issues. Among numeric features, we see a few normally distributed features, such as topic coverage. We also have some strongly skewed distributions like word counts. These are not issues, but they help indicate which type of scaler we should use. Standard scaling is useful for normal distributions, and Min-Max scaling is good for skewed distributions. We can see issues with potentially missing or poor data. The silent period rate has a massive amount of 0.0, while the speaker speed is so closely distributed it appears useless. The silent period rate may be able to be imputed, but speaker speed is a strong candidate for dropping. The categorical features that can be graphed are extremely imbalanced. The most covered topic was useless to graph because it has too many unique values. Has parts and lecture type are almost entirely one value, meaning that the model will be far better performing in the majority class.

In the correlation matrix, values with $|p| < 0.1$ are considered uncorrelated and may not have any relation to the compared feature. As we are predicting median engagement, we should focus on that section of the table first. Potentially unrelated features include auxiliary rate, conjugate rate, tobe verb rate, preposition rate, title word count, topic coverage, and speaker speed. All of these are candidates for dropping due to how poorly they correlate with median engagement. Some features are highly correlated with each other. Fraction stopword presence had $p > 0.7$ for duration, word count, and document entropy. Some may be redundant as they are so closely related to fraction stopword presence that inclusion isn't necessary.

For the categorical features, it appears that all of them have at least a minor impact on median engagement based on the strip plots. This

means they are probably not redundant and could be useful in predicting median engagement. I cannot draw any conclusions from the most covered topic as it has such high cardinality that any graphical representation now would be useless. In the categorical cross-tabulations, we can see that there are no associations that are too strong to indicate redundancy. The same is true from comparing the categorical and numeric features. The distributions of numeric features do not split completely with any of the categorical features meaning that there is also no redundancy here.

Question 1.3. Derive conclusions from your analyses and implement data preprocessing.

This question expects you to derive conclusions and implement preprocessing steps based on the analyses carried out in the previous question. Use the markdown cell to propose preprocessing steps and the code cell to implement the preprocessing function.

- Based on the results obtained in the previous section, identify noteworthy observations (e.g. missing values, outliers etc.)? Describe what you observed and the implications.
- How are you going to preprocess the dataset based on these observations? Justify your preprocessing steps in relation to the analyses.
- In the subsequent code cell, implement the `preprocess_lecture_dataset` function to take the entire dataset as input and carry out preprocessing
- You may use additional code cells to implement sub-functions.

Question: Justification

The simplest decision is to drop speaker_speed. This feature is completely useless as it is so tightly distributed that the data is meaningless. I have also chosen to drop instances of extremely short lectures or lectures with very few words. These are not representative of a normal lecture and the data gathered within them will be unreliable due to such a small sample size. I also drop instances where silent period rate > 1 as this is nonsensical.

I dummy encode all categorical variables apart from most covered topic. This is because there is no clear ordinality to them and there are few enough unique values that the dataset doesn't massively increase in dimensionality. I have chosen to frequency encode the most covered topics. It is impossible to properly one hot encode this due to the massive cardinality. The frequency of each value could be indicative of some connection to the median engagement and as such is likely better than dropping the feature completely.

I am also imputing the 0 values of some numerical features. For these it is clearly an issue of missing data as in the graphs above there are spikes at 0. I use an iterative imputer which trains on the dataset and predicts what the missing value is based on the present data in the instance. Finally I scale the data. Based on the graphs gathered previously I have chosen a standard scaler which creates a normal distribution centered at 0 with a standard deviation of 1. This is only applied to the already normally distributed data. The non normally distributed data maintain the shape of their distribution by using a min-max scaler. The data needs to be scaled to prevent having disproportionally penalized weights in θ. Weights for large numbers will naturally be far lower than weights for smaller numbers thus penalizing the weights of the smaller variables much more harshly.

In [11]:
```
def preprocess_lecture_dataset(dataset):
    """
    takes the lecture dataset and transforms it with necessary pre-processing steps.

    Params:
        dataset (pandas.DataFrame): DataFrame object that contains the original dataset provided for the courses

    Returns:
        preprocessed_dataset (pandas.DataFrame): DataFrame object that contains the dataset after data
                                                 pre-processing has been carried out
    """

    # Your Code Here
    preprocessed_dataset = dataset.copy()

    # drop speaker speed
    preprocessed_dataset = preprocessed_dataset.drop(columns=['speaker_speed'])

    # drop instances with extremely low duration or word counts or silent period rate > 1
    preprocessed_dataset = preprocessed_dataset.drop(preprocessed_dataset[preprocessed_dataset['duration'] < 10
    preprocessed_dataset = preprocessed_dataset.drop(preprocessed_dataset[preprocessed_dataset['word_count'] < 1
    preprocessed_dataset = preprocessed_dataset.drop(preprocessed_dataset[preprocessed_dataset['silent_period_ra

    # encode categorical variables with dummies (categorical data is encoded here so the imputer can use them fo
    preprocessed_dataset = pd.get_dummies(preprocessed_dataset, columns=['lecture_type', 'has_parts', 'subject_(

    # frequency encode most_covered_topic
    freq_encoding = preprocessed_dataset['most_covered_topic'].value_counts(normalize=True)
    preprocessed_dataset['most_covered_topic'] = preprocessed_dataset['most_covered_topic'].map(freq_encoding)

    # impute missing values

    #replace 0 values with na
```

```
    preprocessed_dataset['silent_period_rate'] = preprocessed_dataset['silent_period_rate'].fillna(0)
    preprocessed_dataset['auxiliary_rate'] = preprocessed_dataset['auxiliary_rate'].fillna(0)
    preprocessed_dataset['normalization_rate'] = preprocessed_dataset['normalization_rate'].fillna(0)
    preprocessed_dataset['tobe_verb_rate'] = preprocessed_dataset['tobe_verb_rate'].fillna(0)
    preprocessed_dataset['preposition_rate'] = preprocessed_dataset['preposition_rate'].fillna(0)
    preprocessed_dataset['conjugate_rate'] = preprocessed_dataset['conjugate_rate'].fillna(0)
    preprocessed_dataset['pronoun_rate'] = preprocessed_dataset['pronoun_rate'].fillna(0)

    #create imputer
    imputer = IterativeImputer(max_iter=10, random_state=0)
    preprocessed_dataset = pd.DataFrame(imputer.fit_transform(preprocessed_dataset), columns=preprocessed_datas

    # scale normally distributed values
    scaler = StandardScaler()

    normal_dists = ['auxiliary_rate', 'conjugate_rate', 'normalization_rate', 'tobe_verb_rate', 'preposition_ra

    preprocessed_dataset[normal_dists] = scaler.fit_transform(preprocessed_dataset[normal_dists])

    # minmax scale others
    scaler = MinMaxScaler()

    min_max_dists = ['duration', 'word_count']
    preprocessed_dataset[min_max_dists] = scaler.fit_transform(preprocessed_dataset[min_max_dists])


    return preprocessed_dataset
```

In [12]:
```
preprocessed_lectures = preprocess_lecture_dataset(lectures)
```

In [13]:
```
preprocessed_lectures.head()
```

Out[13]:

	auxiliary_rate	conjugate_rate	normalization_rate	tobe_verb_rate	preposition_rate	pronoun_rate	document_entropy	easiness
0	-0.475406	-0.626466	1.328709	-0.877254	0.285675	-1.190291	-0.112609	-1.124224
1	-0.279579	-0.850490	-0.281519	-0.723500	-1.163078	-0.713923	0.823715	0.281278
2	0.598840	-0.632152	0.978030	-0.619357	0.109755	0.039433	0.246768	-0.335706
3	1.425170	0.170148	-0.480578	0.251268	0.353719	-0.666538	0.547897	-0.555751
4	1.002714	0.071004	0.208221	-0.517101	0.814407	-0.727968	0.579104	-0.959397

5 rows × 40 columns

In [14]:
```
preprocessed_lectures.describe()
```

Out[14]:

	auxiliary_rate	conjugate_rate	normalization_rate	tobe_verb_rate	preposition_rate	pronoun_rate	document_entropy	
count	1.145500e+04	1.145500e+04	1.145500e+04	1.145500e+04	1.145500e+04	1.145500e+04	1.145500e+04	1.145
mean	-8.001747e-17	-2.549394e-16	4.342033e-16	-9.366386e-17	-5.564006e-16	-3.442612e-17	-4.714208e-16	-7.71
std	1.000044e+00	1.000044e+00	1.000044e+00	1.000044e+00	1.000044e+00	1.000044e+00	1.000044e+00	1.000
min	-2.984421e+00	-3.451932e+00	-2.258019e+00	-4.276421e+00	-6.609296e+00	-4.128750e+00	-1.230625e+01	-7.023
25%	-6.475837e-01	-5.244138e-01	-6.784266e-01	-5.871584e-01	-5.591506e-01	-4.891045e-01	-3.760196e-01	-5.31
50%	-7.321628e-02	-1.045810e-01	-1.644658e-01	-9.891261e-02	2.908059e-02	-2.924045e-02	9.963611e-02	-2.47
75%	5.634509e-01	3.547187e-01	5.056692e-01	4.783993e-01	6.112077e-01	4.256524e-01	5.884941e-01	4.60
max	8.945065e+00	3.834677e+01	8.485678e+00	1.239245e+01	4.582116e+00	2.923779e+01	2.476848e+00	4.556

8 rows × 40 columns

Question 1.4 Numerically encode the dataset for model training.

This question expects you to create the final numerical dataset you will use to carry out model training with ridge regression.

- Implement the `prepare_final_dataset` function to transform different features.
- Features that belong to different data types need to be transformed to an ideal numerical representation
- You may use helper functions in `scikit-learn` machine learning library to implement this function.

In [15]:
```
def prepare_final_dataset(preprocessed_dataset):
    """
    takes the preprocessed lecture dataset and transforms it to the vector representation.

    Params:
```

```
        preprocessed_dataset (pandas.DataFrame): DataFrame object that contains the original
                                                 dataset provided for the coursework

    Returns:
        X (pandas.DataFrame): DataFrame object that contains the features
        y (numpy.array): List of labels
    """

    # Your Code Here
    # categorical variables have already been encoded in the above cells as it was necessary for the imputer
    X = preprocessed_dataset.drop(columns=['median_engagement'])
    y = preprocessed_dataset['median_engagement']

    return preprocessed_dataset, X, y
```

In [16]:
```
final_dataset, full_X, full_y = prepare_final_dataset(preprocessed_lectures)
```

In [17]:
```
full_X
```

Out[17]:

	auxiliary_rate	conjugate_rate	normalization_rate	tobe_verb_rate	preposition_rate	pronoun_rate	document_entropy	easine
0	-0.475406	-0.626466	1.328709	-0.877254	0.285675	-1.190291	-0.112609	-1.1242
1	-0.279579	-0.850490	-0.281519	-0.723500	-1.163078	-0.713923	0.823715	0.2812
2	0.598840	-0.632152	0.978030	-0.619357	0.109755	0.039433	0.246768	-0.3357
3	1.425170	0.170148	-0.480578	0.251268	0.353719	-0.666538	0.547897	-0.5557
4	1.002714	0.071004	0.208221	-0.517101	0.814407	-0.727968	0.579104	-0.9593
...	...	...	...	...	...	...	...	
11450	-0.225220	-0.129610	1.036658	-0.408152	-0.255571	0.189444	-0.065360	-0.4970
11451	2.229901	-0.396422	-0.786709	0.390537	-0.819401	0.420881	-0.033170	1.1790
11452	-0.004378	-1.222725	-0.310697	-0.784694	-0.536079	0.012992	0.705773	1.3151
11453	-1.873415	1.052145	1.160113	0.571812	0.092701	-1.019010	-1.774570	-1.3148
11454	-0.792061	-0.852627	0.183153	0.406242	1.207253	-0.413503	0.101854	-0.4158

11455 rows × 39 columns

In [18]:
```
full_y
```

Out[18]:
```
0        0.502923
1        0.011989
2        0.041627
3        0.064989
4        0.052154
           ...
11450    0.044655
11451    0.038525
11452    0.012572
11453    0.998364
11454    0.032745
Name: median_engagement, Length: 11455, dtype: float64
```

Let us now save the final data

In [19]:
```
full_X.to_csv("features_final.csv", index=False)
np.save("labels_final.npy", full_y.to_numpy())
```

Part 2: Modeling and Evaluation (30 Marks)

In this section, we develop a model using the preprocessed data. We start by loading the data that we saved in the previous part.

In [61]:
```
full_X = pd.read_csv("features_final.csv")
full_y = np.load("labels_final.npy")

print(full_X.shape)
print(full_y.shape)

# If you didn't manage to save the preprocessed data structures from part one.
# You can start the exercise with alternative data. But the performance will be very low.

# full_X = pd.read_csv("features_seed.csv")
# full_y = np.load("labels_seed.npy")
```

```
(11455, 39)
(11455,)
```

Question 2.1 Train Ridge Regression Model.

In this question, you are expected to derive a trained ridge regression model.

- Implement the `train_model` function to output the trained ridge regression model.
- You may use helper functions and models in `scikit-learn` library

In [21]:
```
from sklearn.linear_model import Ridge
from sklearn.kernel_ridge import KernelRidge
```

In [22]:
```
def train_ridge_model(X,y, hyperparams):
    """
    takes the training data with the hyper-parameters to train the ridge model

    Params:
        X (pandas.DataFrame): DataFrame object that contains the features
        y (numpy.array): List of labels
        hyperparams (dict): a dictionary of hyperparameters where the key is the hyperparameter name,
                            and the value is the hyperparameter value

    Returns:
        ridge_model(scikit-learn model): A trained scikit-learn model object
        :
    """

    # Your Code Here
    ridge_model = Ridge(alpha=hyperparams['alpha'], max_iter=hyperparams['max_iter'], solver=hyperparams['solve
    ridge_model.fit(X, y)

    return ridge_model
```

- Define the python dictionary `hyperparams` with the hyperparameters needed for Ridge Regression.

In [23]:
```
hyperparams = {
    # Your Code Here
    "alpha" : 0.1,
    "max_iter" : 10000,
    "solver" : "auto"
}
```

In [24]:
```
temp_ridge_model = train_ridge_model(full_X, full_y, hyperparams)
```

In [25]:
```
ridge_pred = temp_ridge_model.predict(full_X)
```

Question 2.2 Gaussian (RBF) Kernel Regression Model

In this question, you are expected to implement the Gaussian (Radial Basis Function/ RBF) kernel and use it with Ridge Regression to train a Kernel Ridge model that uses the Gaussian Kernel.

- Implement the `gauss_kernel` function to output the similarity between two points (`x` and `x_dash`) using the Gaussian kernel.
- You may use helper functions `numpy` and `scipy` libraries to speed up matrix computations. But the function should be implemented by you.

In [26]:
```
def gauss_kernel(x, x_dash, gamma):
    """
    takes two data points and calculates their similarity using the RBF function.

    params:
        x (numpy.array): point 1 coordinates
        x_dash (numpy.array): point 2 coordinates
        gamma : relevant hyperparameter for the Gaussian Kernel

    returns:
        similarity (float): similarity between the two points
    """

    norm = np.linalg.norm(x - x_dash)
    similarity = np.exp(-gamma * norm ** 2)

    return similarity
```

- Implement the `train_kernel_ridge_model` function to output the trained kernel ridge regression model.

- Use the relevant parameters in the `sklearn.kernel_ridge.KernelRidge` function to pass the `gauss_kernel` function implemented earlier with kernel regression.
- Training this model may take some time (≈ 10 minutes).

In [27]:
```
def train_kernel_ridge_model(X,y, hyperparams, kernel_function, kernel_params):
    """
    takes the training data with the hyper-parameters to train the ridge model

    Params:
        X (pandas.DataFrame): DataFrame object that contains the features
        y (numpy.array): List of labels
        hyperparams (dict): a dictionary of hyperparameters where the key is the hyperparameter name,
                            and the value is the hyperparameter value
        kernel_function (callable): a callable python function which is the kernel function
        kernel_params (dict): a dictionary of kernel parameters where the key is the kernel parameter name,
                            and the value is the parameter value

    Returns:
        kernel_ridge_model(scikit-learn model): A trained scikit-learn model object
    """

    # Your Code Here
    def call_kernel(x, x_dash):
        return kernel_function(x, x_dash, **kernel_params)

    kernel_ridge_model = KernelRidge(alpha=hyperparams['alpha'], kernel=call_kernel, **kernel_params)
    kernel_ridge_model.fit(X, y)

    return kernel_ridge_model
```

In [28]:
```
hyperparams = {
    "alpha" : 0.1
}

kernel_params = {
    "gamma" : 1e-2
}

temp_kernel_ridge_model = train_kernel_ridge_model(full_X, full_y, hyperparams, gauss_kernel, kernel_params)
```

In [29]:
```
temp_y = temp_kernel_ridge_model.predict(full_X)
print(temp_y)
```

```
[0.15273195 0.06538331 0.07644171 ... 0.04495014 0.6095629  0.03808217]
```

Question 2.3 Propose and Implement two evaluation metrics that are suitable for model evaluation in this task.

This question expects you to propose two evaluation metrics that can be used to assess predictive capabilities in this task and implement them.

- Propose two metrics by replacing `Your Answer Here`. You are encourage to propose metrics that go beyond the ones taught in class.
- implement the two metrics while renaming function names from `eval_metric_1` and `eval_metric_2` to the metrics you are proposing.

Metric 1

The first evaluation metric is R^2 ($R^2 = 1 - \sum_{ni=1}(y_i - \hat{y}_i)^2 \sum_{ni=1}(y - \mu_y)^2$). R^2 describes how well the model explains variation in the target variable. It does this by including the variance of data as the denominator in the formula. Low values of R^2 mean that the model is poor, as the model is not predictive of anything with similar performance to just using the mean. High values of R^2 show that the distance between predicted and actual values is far smaller than actual values to their mean. It indicates simply and effectively the quality of a given model at predicting the target variable. It is also standardized which makes it useful for comparing the outputs of the same model on different datasets. The variance denominator accounts for differences in the distribution of two datasets so results can be accurately compared. I use this over adjusted R^2 because it is more complicated and not beneficial for this case. Adjusted R^2 penalizes having more predictors than necessary which can prevent overfitting. The goal of this is to assess two models not two data preparations and as such it is not used.

In [30]:
```
def eval_metric_1(y_actual, y_predicted):
    """
    returns the evaluation metric.

    Params:
        y_actual (numpy.array): List of actual labels
```

```
        y_predicted (numpy.array): List of predicted labels

    Returns:
        metric (float): the evaluation metric
    """

    # R-squared
    rss = np.sum((y_actual - y_predicted) ** 2)
    tss = np.sum((y_actual - np.mean(y_actual)) ** 2)
    r2 = 1 - (rss / tss)
    metric = r2

    return metric
```

Metric 2

Mean squared error(MSE) is used as the second evaluation metric (MSE = $^{1n}\sum_{ni=1}(y_i - \hat{y}_i)^2$). MSE measures the average square of the distance between a prediction and the true value. This is useful as it penalizes models that fit very well to a subset of the data but poorly to the rest of the data more harshly than mean absolute error. It is used in combination with R^2 as it gives an indication as to how the models ability to explain variation actually impacts the predictions. Together they serve to show that models are predicting well and are accurately modelling the data, instead of over or underfitting.

In [31]:
```
def eval_metric_2(y_actual, y_predicted):
    """
    returns the evaluation metric.

    Params:
        y_actual (numpy.array): List of actual labels
        y_predicted (numpy.array): List of predicted labels

    Returns:
        metric (float): the evaluation metric
    """

    # MSE
    mse = np.mean((y_actual - y_predicted) ** 2)
    metric = mse

    return metric
```

Question 2.4 Evaluate the performance of the Ridge Regression model to detect overfitting.

In this question, you are expected to implement a function to evaluate the predictive performance of a trained Ridge Regression model and detect if overfitting is evident.

- Implement the `evaluate_ridge_model` function to take in the lectures data and
  - Handle the data carefully before training the model
  - Design a pipeline that incorporates comprehensive techniques to ensure robust and reliable model training and evaluation.
  - train the model
  - evaluate the model using the proposed metrics and
  - print the relevant information to assess model performance (including overfitting)
- The function does NOT have to return anything. Make sure it prints the relevant metrics instead.
- You are expected to design the training methodology to end up training the most generalisable model from the data provided.

In [32]:
```
def visualize(y_pairs):
    """
    graphs true values and predictions of a list of results from the model

    Params:
        Y_pairs(List): list of lists containing pairs of y_actual and y_predicted to be plotted together
    """
    for pair in y_pairs:
        y_actual = pair[0]
        y_predicted = pair[1]
        plt.scatter(y_actual, y_predicted)
        plt.plot([0, 1], [0,1], color='red', linestyle = '--')
        plt.xlabel('Actual Values')
        plt.ylabel('Predicted Values')
        plt.title('Actual vs. Predicted Values')
    plt.show()

    for pair in y_pairs:
        y_actual = pair[0]
        y_predicted = pair[1]
```

```
        plt.hist(y_predicted - y_actual, bins=30)
        plt.xlabel('Residuals')
        plt.ylabel('Frequency')
        plt.title('Histogram of Residuals')
    plt.show()

    for pair in y_pairs:
        y_actual = pair[0]
        y_predicted = pair[1]
        plt.scatter(y_actual, y_predicted - y_actual)
        plt.xlabel('Actual Values')
        plt.ylabel('Residuals')
        plt.title('Residuals vs. Actual Values')
    plt.show()
```

In [ ]:
```
def evaluate_ridge_model(X,y):
    """
    trains the most viable model using the lecture data for median engagement prediction to evaluate it using tl

    Params:
        X (pandas.DataFrame): features of the dataset
        y (numpy.array): labels
    """

    # custom scorer for grid search
    r2_scorer = make_scorer(eval_metric_1, greater_is_better=True)

    # split the data into train/test sets
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

    # grid search hyperparameters
    param_grid = {
        'alpha': [0.01, 0.1, 1, 10],
        'solver': ['auto', 'svd', 'cholesky', 'lsqr', 'sparse_cg', 'sag', 'lbfgs']
    }

    # grid search with custom r2
    ridge = Ridge(max_iter=10000)
    grid_search = GridSearchCV(ridge, param_grid, cv=5, scoring=r2_scorer)
    grid_search.fit(X_train, y_train)

    # get the best parameters and the best model
    best_params = grid_search.best_params_
    print("Best Parameters:", best_params)

    best_model = grid_search.best_estimator_
    y_train_pred = best_model.predict(X_train)
    y_test_pred = best_model.predict(X_test)

    # calculate metrics
    print("Train R2:", eval_metric_1(y_train, y_train_pred))
    print("Train MSE:", eval_metric_2(y_train, y_train_pred))

    print("Test R2:", eval_metric_1(y_test, y_test_pred))
    print("Test MSE:", eval_metric_2(y_test, y_test_pred))
```

In [62]:
```
evaluate_ridge_model(full_X, full_y)
```

```
Best Parameters: {'alpha': 1, 'solver': 'lsqr'}
Train R2: 0.4333550025201819
Train MSE: 0.014570419458303386
Test R2: 0.4285785229947151
Test MSE: 0.016673705971021163
```

Question

- Is the model exhibiting overfitting? Justify your answer

The model is not exhibiting any overfitting. The use of a cross validated grid search ensures that the model parameters are tuned properly for the strongest performance with minimal overfitting. It does this by performing a 5-fold cross validation on each set of parameters and selecting the ones that optimize R^2 which is the more robust of the two metrics. The model is finally assessed on a completely unseen testing set for the final metrics. The model performs very similarly on both metrics. If the model were exhibiting overfitting we would see large performance differences in either of the metrics. In the plots, the model has a very similar prediction distribution on both the train and test set. These indicate that the model is generalizable.

Part 3: Ridge Regression: From Theory to Implementation (40 Marks)

In this section, we focus on understanding Ridge Regression better. Ridge Regression is the main modelling tool that we use throughout this coursework. It introduces a penalty to the objective of the model if the linear weights become too big.

This part of the coursework expects the learner to gradually implement the ridge regression using matrix operations using python. This is expected to help the learners connect the mathematical derivations to the actual programmatic realisation of the learning algorithms.

Hints:

- All X,y inputs in the proceeding assumes multiple *observations* are being passed

Dataset

We use a pre-created dataset for this part of the exercise. Let us load the dataset.

In [36]:
```
full_X = pd.read_csv("features_seed.csv")
full_y = np.load("labels_seed.npy")

full_X.describe()
```

Out[36]:

	auxiliary_rate	conjugate_rate	normalization_rate	tobe_verb_rate	preposition_rate	pronoun_rate
count	11548.000000	11548.000000	11548.000000	11548.000000	11548.000000	11548.000000
mean	0.015811	0.040899	0.021373	0.044119	0.116088	0.123087
std	0.005465	0.013378	0.009630	0.010820	0.018797	0.029803
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	0.012369	0.034486	0.014932	0.038092	0.106400	0.109159
50%	0.015441	0.039472	0.019856	0.043179	0.116836	0.122261
75%	0.018846	0.044937	0.026235	0.049180	0.127139	0.135148
max	0.066667	0.492754	0.101990	0.172414	0.318182	0.947967

Question 3.1 Transform the data to matrix representations that are suitable for training a Ridge Regression model.

In this question, you are expected to implement a function to prepare the feature and label data that we otherwise input to `scikit-learn` and prepare the matrix/vector representations.

- Implement the `prepare_data_for_training` function to take in the features and labels and return feature matrix/vector and label matrix/vector back.
  - the function should take `pandas.DataFrame` objects as input. These DataFrames should have the data values that are passed to the `fit()` function of the `scikit-learn` model (ie. after all the preprocessing and other transformations)
  - you are expected to determine the suitable dimensionality for the output matrices
- You must NOT use any `scikit-learn` or any other machine learning library's functions within this function. It will be penalised.

In [37]:
```
def prepare_data_for_training(X, y=None):
    """
    returns the matrices that are passed in to the training function of the ridge regression.

    Params:
        X (pandas.DataFrame): Features in the dataset
        y (pandas.DataFrame): Labels in the dataset, Optional

    Returns:
        X (numpy.array): X matrix/vector passed to the Ridge Regression training
        y (numpy.array): y matrix/vector passed to the Ridge Regression training
    """

    # Your Code Here
    # make X numpy
    X = X.to_numpy()
    # add 1s for the constant bias
    X = np.hstack([np.ones((X.shape[0], 1)), X])
    print(X.shape)

    if y is not None and isinstance(y, pd.DataFrame):
        y = y.to_numpy()
        print(y.shape)

    return X, y
```

In [38]:
```
X_, y_ = prepare_data_for_training(full_X, full_y)
```

```
(11548, 7)
```

Question 3.2 Implement the training and prediction functions of the Ridge Regression model (primal form).

This question expects you to implement the training and prediction capabilities of the ridge regression model.

- Implement the `fit_ridge_reg` function to take in the features, labels and the hyper-parameters to return the trained parameters of the model.
- You are expected to use the Primal form when implementing the fitting step.
- You are NOT allowed to use `scikit-learn` functions here. It will be penalised.

In [39]:
```
def fit_ridge_reg(X, y, hyperparams):
    """
     Params:
        X (numpy.array): X matrix/vector passed to the Ridge Regression training
        y (numpy.array): y matrix/vector passed to the Ridge Regression training
        hyperparams (dict): a dictionary where the key is the hyperparameter name
                            and values is the hyperparameter value

    Returns:
        _theta (numpy.array): the trained parameters of the model
    """

    # Your Code Here
    _lambda = hyperparams['lambda']

    # 6 features + 1 constant
    # ((7 x 11548) * (11548 x 7) + lambda x 7x7)^-1 = 7*7 * (7 x 11548) * (11548 x 1) =
    _theta = np.linalg.inv(X.T @ X + _lambda * np.eye(X.shape[1])) @ X.T @ y

    print(_theta.shape)

    return _theta
```

In [40]:
```
hyperparams = {
    "lambda": 0.001
}

theta = fit_ridge_reg(X_, y_, hyperparams)
```

```
(7,)
```

In [41]:
```
print("The shape of theta matrix/vector: {} \n\n The values are: \n {}".format(theta.shape, theta))
```

```
The shape of theta matrix/vector: (7,)

 The values are:
[ 0.11972292 -0.48418088  0.66264088  1.30120135 -0.26920015  0.27774184
-0.53857572]
```

- Implement the relevant parts of the `RidgeRegression` class below.
  - add relevant object attributes including hyperparameters
  - `fit` and `predict` functions need to be implemented as well
- You may reuse the functions you implemented previously in this part of the assignment
- You are NOT allowed to use `scikit-learn` functions here. It will be penalised.

In [42]:
```
class RidgeRegression():
    def __init__(self, hyperparams):
        """
        instantiates the class

        Params:
            hyperparams (dict): a dictionary where the key is the hyperparameter name
                            and values is the hyperparameter value
        """

        self.fitted = False # indicates whether the model is already trained or not

        # Your Code Here
        self.hyperparams = hyperparams
        self.theta = None


    def fit(self, X, y):
        """
        trains the model given the data. Updates  models internal parameters

        Params:
```

```
            X (pandas.DataFrame): Features in the dataset
            y (pandas.DataFrame): Labels in the dataset
        """

        # Your Code Here
        X, y = prepare_data_for_training(X, y)
        self.theta = fit_ridge_reg(X, y, self.hyperparams)
        self.fitted = True


    def predict(self, X):
        """
        makes predictions from given features.
        ! The model should be trained first. Otherwise throws an error.

        Params:
            X (pandas.DataFrame): Features in the dataset
        """

        # Your Code Here
        X,_ = prepare_data_for_training(X)

        if not self.fitted:
            raise Exception("Model not trained yet")
        return X @ self.theta
```

In [43]:
```
hyperparams = {
    "lambda": 0.001
}

RR = RidgeRegression(hyperparams)
```

In [44]:
```
print("Attributes of the RidgeRegression Instance Before Training: \n{}".format(RR.__dict__))
```

```
Attributes of the RidgeRegression Instance Before Training:
{'fitted': False, 'hyperparams': {'lambda': 0.001}, 'theta': None}
```

- Train the model with the appropriate data using the `fit` function of the model instance.

In [45]:
```
RR.fit(full_X, full_y)
```

```
(11548, 7)
(7,)
```

In [46]:
```
print("Attributes of the RidgeRegression Instance After Training: \n{}".format(RR.__dict__))
```

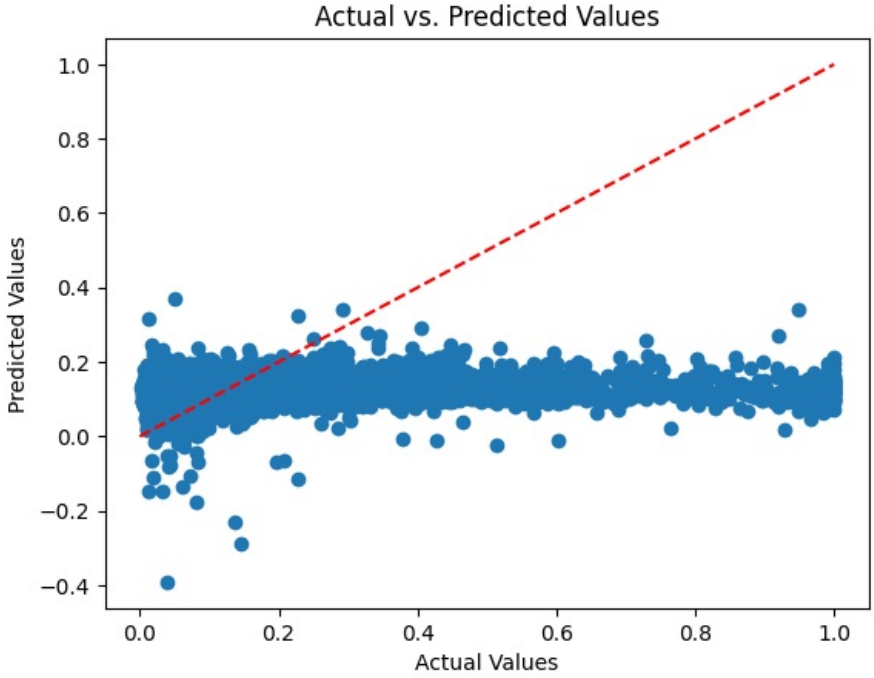
```
Attributes of the RidgeRegression Instance After Training:
{'fitted': True, 'hyperparams': {'lambda': 0.001}, 'theta': array([ 0.11972292, -0.48418088,  0.66264088,  1.301
20135, -0.26920015,
        0.27774184, -0.53857572])}
```

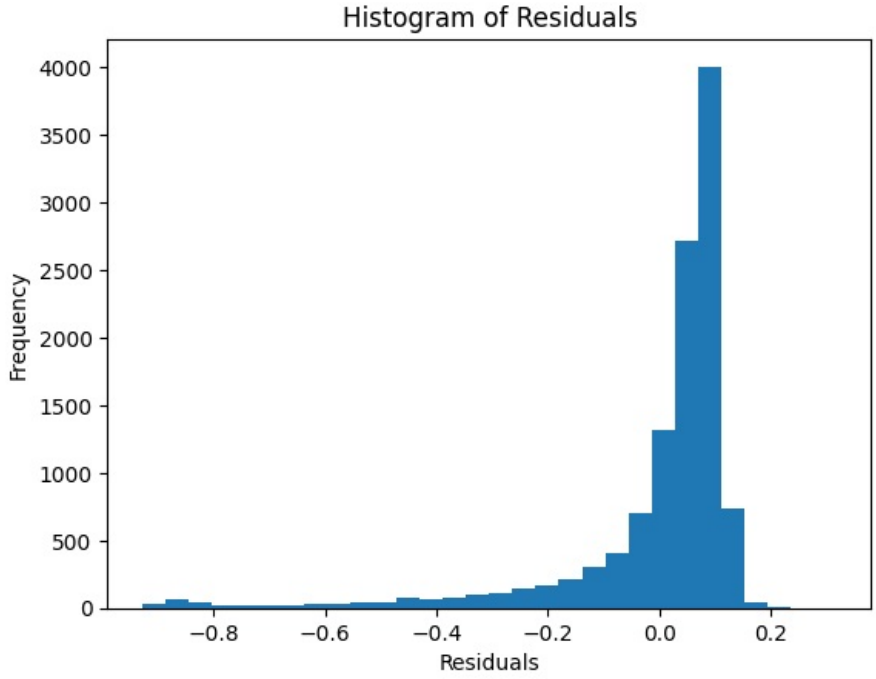
Question:

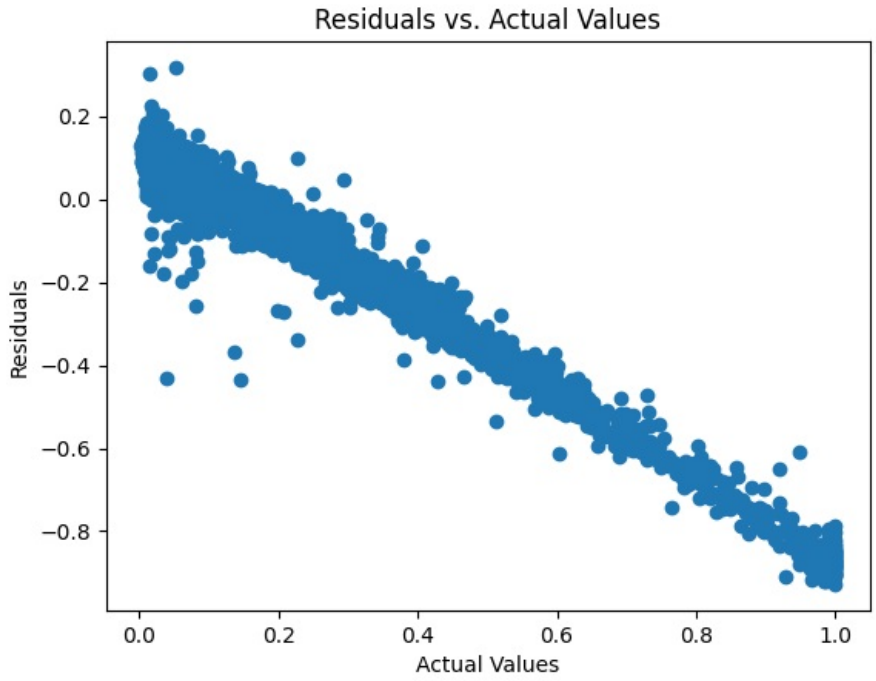
- Get predictions from the trained model and show that the predictions have a linear correlation with the actual labels. For **this question**, you are allowed to use scientific computing packages such as `scikit-learn` or `sciPy`

In [47]:
```
# Your Code Here
from scipy.stats import pearsonr
pred = RR.predict(full_X)

visualize([[full_y, pred]])

# correlation
pearson = pearsonr(full_y, pred)
print(f'Statistic: {pearson.statistic} \nP-Value: {pearson.pvalue}')

# individual predictors linear correlation
df = pd.concat([full_X,pd.Series(full_y, name='median_engagement')],axis=1)
corr_mat = df.corr()
corrs = corr_mat['median_engagement'].to_frame()

sns.heatmap(corrs, annot=True, cmap='coolwarm', fmt='.2f', linewidths=0.5)
plt.show()
```

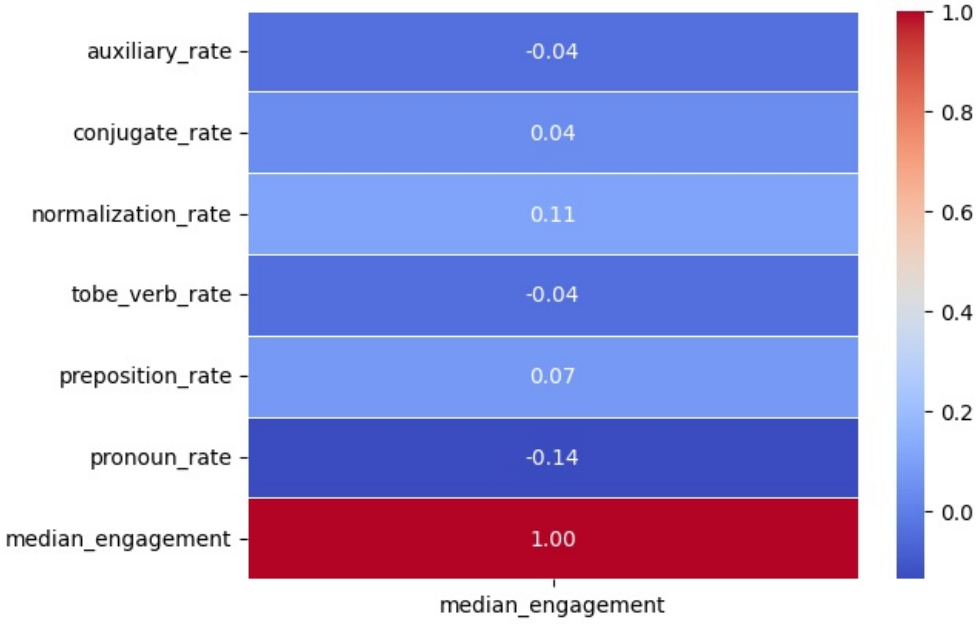
```
(11548, 7)
```


Actual vs. Predicted Values
Predicted Values
Actual Values
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
0.0
0.2
0.4
0.6
0.8
1.0


Histogram of Residuals
Frequency
Residuals
4000
3500
3000
2500
2000
1500
1000
500
0
−0.8
−0.6
−0.4
−0.2
0.0
0.2




```
Statistic: 0.16320990287545165
P-Value: 9.247031518132363e-70
```



Question: Why did you use the above method? Justify your answer

There is clearly very little linear correlation between the predictions and the actual data. The first graph depicts the prediction value against the true value. The predictions do not increase in any obvious way as the true value increases. In the residuals histogram, the data is strongly skewed. If there was a strong linear correlation then we would expect the residuals to be normally distributed. Here it can be seen that the model is much more likely to underpredict than overpredict, this coincides with the first graph. The third graph further

expands upon this. If we had linear correlation, the graph created by the data would be a flat line. The Pearson Correlation statistic indicates linear relationships between two continuous variables. The model predictions have a 0.16 score which indicates a very low positive correlation (values below 0.25 are considered weakly correlated). The p-value of the score is also extremely low indicating that this score is significant and not due to randomness. Finally there is an image of the individual correlations with the target. None of these have strong linear correlations with median engagement, making it unlikely that a model could use them as strong predictors. In this case, ridge regression failed to find a more complex relationship that correlates strongly with median engagement.

I chose these methods as they give a well rounded view of the model results. The graphs give a visual depiction that helps to understand how the model is failing. The Pearson Correlation gives a strong quantitative assessment of the performance of the model and how if at all the data correlates with the target. Finally the individual correlations offer insight as to why the model could be failing to find a linear corelation.

Question 3.3 Ridge Regression in the Online Learning Setting

In this question, we create several building blocks required to learn with Ridge Regression in an online setting using stochastic gradient descent. You are first expected to derive the first derivative of the Ridge Regression loss function.

- Implement the `ridge_reg_loss_derivative` function to take in the features, labels, parameters, and hyperparameters, and return the first derivative $\delta L \delta \theta$ of the loss function L.

In [48]:
```
def ridge_reg_loss_derivative(X, y, theta, hyperparams):
    """
    takes data, parameters and hyperparameters to calculate the first derivative of ridge loss

    Params:
        X (numpy.array): a matrix/vector of features
        y (numpy.array): a matrix/vector of labels
        theta (numpy.array): a matrix/vector of parameters being trained
        hyperparams (dict): a dictionary where the key is the hyperparameter name
                            and values is the hyperparameter value

    Returns:
        derivative (numpy.array): the derivative used for updating the parameters
    """

    # Your Code Here
    _lambda = hyperparams['lambda']
    derivative = -2 * (X.T @ (y - X @ theta)) + 2 * _lambda * theta
    return derivative
```

- Implement the `train_stoch_ridge_reg` function to take data, parameters and hyperparameters and return the updated theta
- You are not allowed to use machine learning libraries such as `scikit-learn` or tensor computation libraries such as `tensorflow`, `keras`, `pytorch` etc. in this section. You will be penalised for using such libraries.

In [49]:
```
def train_stoch_ridge_reg(X, y, _theta, hyperparams):
    """
    takes data, parameters and hyperparameters and returns the updated parameters
    from training with data

    Params:
        X (numpy.array): a matrix/vector of features
        y (numpy.array): a matrix/vector of labels
        _theta (numpy.array): a matrix/vector of parameters being trained
        hyperparams (dict): a dictionary where the key is the hyperparameter name
                            and values is the hyperparameter value

    Returns:
        _theta (numpy.array): a matrix/vector of parameters updated after training
    """

    # Your Code Here
    learning_rate = hyperparams['learning_rate']
    derivative = ridge_reg_loss_derivative(X, y, _theta, hyperparams)
    _theta = _theta - learning_rate * derivative


    return _theta
```

Question 3.4 Train and Monitor the Stochastic Ridge Regression Model

In this question, you are expected to use the previously defined stochastic gradient training function (`train_stoch_ridge_reg`) to train a ridge regression model using the `X_, y_` data structures from before. Record the relevant loss values computed in each iteration to analyse if the loss is diminishing over time.

- Implement `train_entire_model` function to take the dataset and train the model over multiple iterations.
  - Run the model for 2000 iterations to reduce the loss values over time
- Record the loss L values of the model over all the iterations.
- pass the list of losses as output from this function.

Hints:

- Set the initial weights (thetas) to a normal distribution scattered around mean 0.
- As the penalisation constant in the Ridge Regression, 0.1 is a good value to use
- A learning rate between 1e-6 and 1e-10 may be suitable for this task

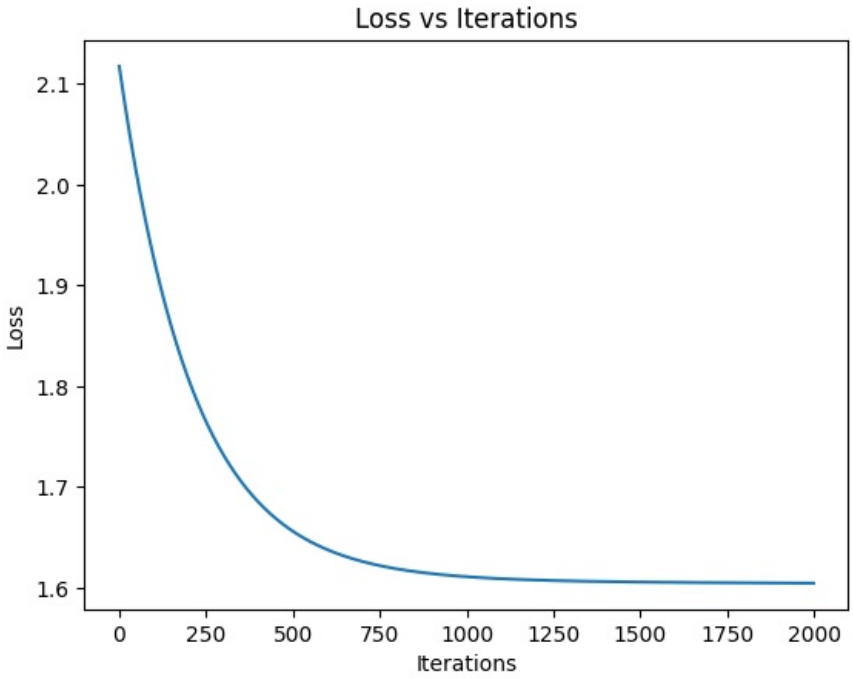
In [50]:
```
def train_entire_model(X_, y_, hyperparams):
    """
    takes data, hyperparameters and returns the list of losses

    Params:
        X_ (numpy.array): a matrix/vector of features
        y_ (numpy.array): a matrix/vector of labels
        hyperparams (dict): a dictionary where the key is the hyperparameter name
                            and values is the hyperparameter value

    Returns:
        losses ([float]): list of loss values for each iteration of learning
    """

    # Your Code Here
    losses = []
    np.random.seed(64)
    _theta = np.random.normal(0, 1, X_.shape[1])
    _lambda = hyperparams["lambda"]


    for i in range(2000):
        _theta = train_stoch_ridge_reg(X_, y_, _theta, hyperparams)
        losses.append(np.mean((y_ - X_ @ _theta) ** 2)+ _lambda * _theta.T @ _theta)

    return losses
```

In [51]:
```
X_, y_ = X_, y_ # Reusing data structures from before

hyperparameters = {
    # Your Code Here
    "lambda": 0.1,
    "learning_rate": 1e-7
}


losses = train_entire_model(X_, y_, hyperparameters)
```

- Implement the `visualise_loss_values` function to use the appropriate visualisations to plot the loss values in a meaningful way.
- The function does not have to return anything. Display the visualisation as a step within the implemented function.

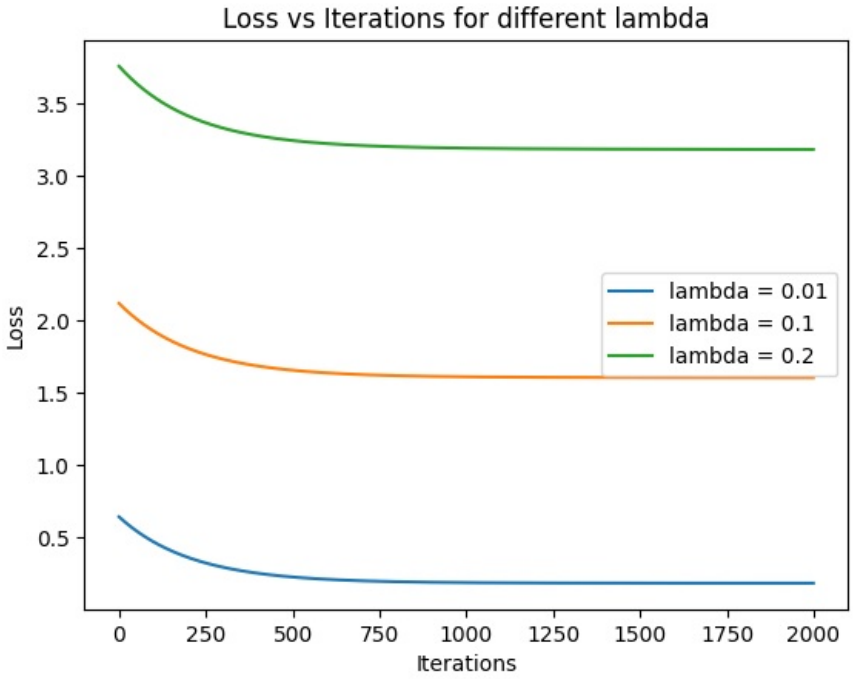
In [52]:
```
def visualise_loss_values(loss_values):
    """
    takes relevant loss values and plots the loss values in the dataset over the iterations (epochs).

    Params:
        loss_values (dict): a dictionary that contains the loss values where key is the loss type
                            and values are the loss values.
    """
    # Your Code Here
    plt.plot(loss_values)
    plt.xlabel('Iterations')
    plt.ylabel('Loss')
    plt.title('Loss vs Iterations')
    plt.show()
```

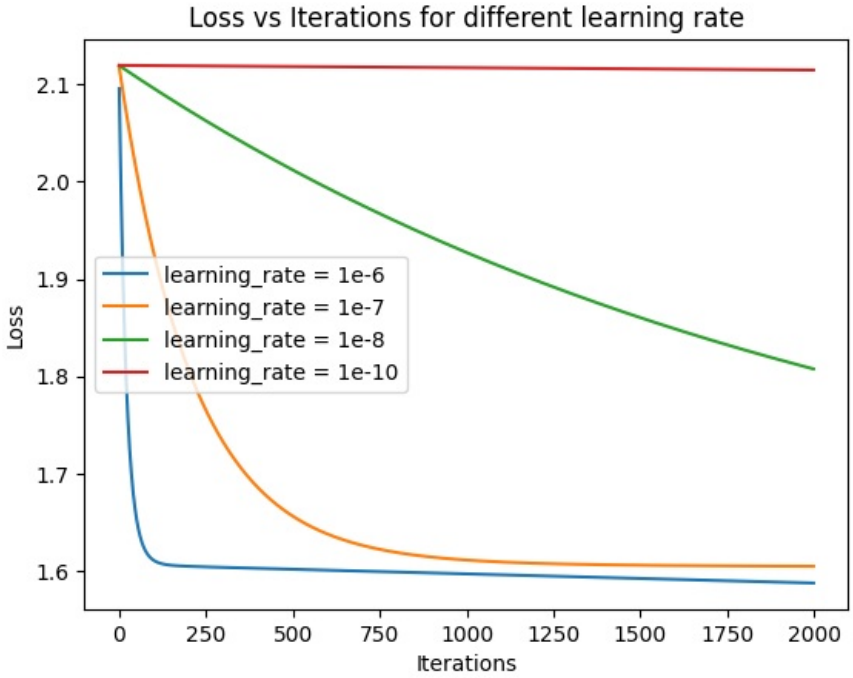
In [53]:
```
visualise_loss_values(losses)
```

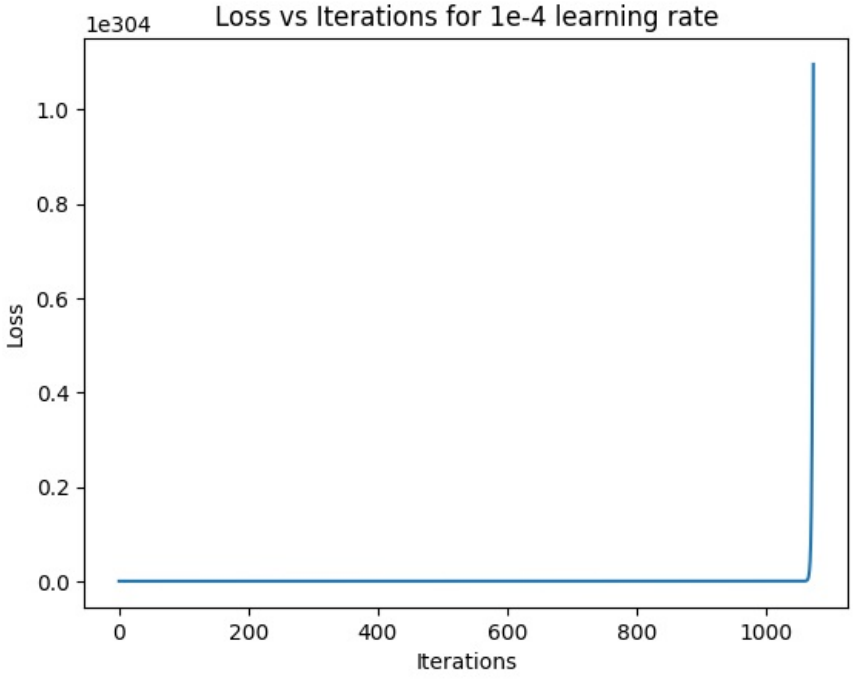


In [54]:

```
# gather losses with different lambda and learning rate
lambda_losses = []
learning_rate_losses = []
for _lambda in [0.01, 0.1, 0.2]:
    hyperparameters = {

        "lambda": _lambda,
        "learning_rate": 1e-7
    }
    losses = train_entire_model(X_, y_, hyperparameters)
    lambda_losses.append(losses)

for learning_rate in [1e-6 , 1e-7, 1e-8, 1e-10, 1e-4]:
    hyperparameters = {
        "lambda": 0.1,
        "learning_rate": learning_rate
    }
    losses = train_entire_model(X_, y_, hyperparameters)
    learning_rate_losses.append(losses)

# plot all lambda losses on the same graph
plt.plot(lambda_losses[0], label='lambda = 0.01')
plt.plot(lambda_losses[1], label='lambda = 0.1')
plt.plot(lambda_losses[2], label='lambda = 0.2')
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.title('Loss vs Iterations for different lambda')
plt.legend()
plt.show()

plt.plot(learning_rate_losses[0], label='learning_rate = 1e-6')
plt.plot(learning_rate_losses[1], label='learning_rate = 1e-7')
plt.plot(learning_rate_losses[2], label='learning_rate = 1e-8')
plt.plot(learning_rate_losses[3], label='learning_rate = 1e-10')
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.title('Loss vs Iterations for different learning rate')
plt.legend()
plt.show()

plt.plot(learning_rate_losses[4])
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.title('Loss vs Iterations for 1e-4 learning rate')

plt.show()
```


Loss vs Iterations for different lambda
Loss
Iterations
lambda = 0.01
lambda = 0.1
lambda = 0.2
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0
250
500
750
1000
1250
1500
1750
2000


Loss vs Iterations for different learning rate
Loss
Iterations
learning_rate = 1e-6
learning_rate = 1e-7
learning_rate = 1e-8
learning_rate = 1e-10
2.1
2.0
1.9
1.8
1.7
1.6
0
250
500
750
1000
1250
1500
1750
2000


Loss vs Iterations for 1e-4 learning rate
1e304
Loss
Iterations
1.0
0.8
0.6
0.4
0.2
0.0
0
200
400
600
800
1000

Question:

- Does the loss get smaller over time? In either case, explain the reason behind it.
- For both the regularisation factor and the learning rate, plot the loss with a sample of larger and smaller values for each hyperparameter. Observe how the loss changes for each hyperparameter *individually* and draw hypotheses justifying these observations.
    - **Note: you do not need to interpret the joint effects of changing the hyperparameter values**

The loss is decreasing over time. This is because the randomly initialized weights are being slowly pushed closer to the optimal weights for the model. The derivative of the loss shows the gradient of the loss function with respect to the model weights. By shifting the weights in the direction the loss is lowest the model is slowly being optimized until it has found some local minimum where it gets caught if the learning rate is low enough. This is because gradients near a minima will be much closer to 0 so smaller adjustments are made to the weights. If the learning rate is too high, it could move out of this local minimum and possibibly never converge. In this case, the loss graph is quite smooth and the loss plateaus near the end of training indicating strong parameterization.

By changing the regularisation factor the only impact is a vertical shift in the loss. This makes sense as the model is very unlikely to overfit on this data. If the model were overfitting we would see different shapes to the loss graphs split by regularisation factor. When changing the learning rate we see a massive difference in the shape of the learning curve. Higher learning rates like 1e-6 have a very steep start and then plateau. This is because large steps are made so minima are reached quicker. Lower learning rates have much shallower curves. This is because the weights are taking much smaller steps, which massively slows down how fast the model improves. For very high learning rates like 1e-4 we can see an explosion in the loss. This is due to the steps being too large so the loss landscape is not properly explored and the model gets completely lost and makes very poor predictions. In this case there is even an overflow that breaks the training function and halts it early due to massive gaps in the predictions and actual values.

- End of Coursework -