COMP0186 Coursework

December 2, 2024

1 COMP186: Foundations of Artificial Intelligence Individual Coursework

1.0.1 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.

1.1 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.

1.2 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.

1.3 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.)

1.4 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.

[81]:
```python
import pandas as pd
import numpy as np
```

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

[82]:
```python
lectures.head(10)
```

[82]:
```
   auxiliary_rate  conjugate_rate  normalization_rate  tobe_verb_rate  \
0        0.013323        0.033309            0.034049        0.035159
1        0.014363        0.030668            0.018763        0.036749
2        0.019028        0.033242            0.030720        0.037827
3        0.023416        0.042700            0.016873        0.046832
4        0.021173        0.041531            0.023412        0.038884
5        0.017616        0.036921            0.023649        0.043195
6        0.011080        0.039036            0.018423        0.042257
7        0.026247        0.038064            0.008956        0.038313
8        0.021587        0.033706            0.018557        0.041091
9        0.023666        0.052065            0.018933        0.027539

   preposition_rate  pronoun_rate  document_entropy   easiness  \
0          0.121392      0.089563          7.753995  75.583936
1          0.095885      0.103002          8.305269  86.870523
2          0.118294      0.124255          7.965583  81.915968
3          0.122590      0.104339          8.142877  80.148937
4          0.130700      0.102606          8.161250  76.907549
5          0.137307      0.098938          8.182952  76.684133
6          0.111698      0.112342          8.101635  85.303173
7          0.098644      0.163951          7.733064  97.572190
8          0.099792      0.123840          8.219794  87.008975
9          0.131239      0.108434          7.714182  88.650478

   fraction_stopword_coverage  fraction_stopword_presence  …  \
0                    0.428135                    0.553664  …
1                    0.602446                    0.584498  …
2                    0.525994                    0.605685  …
3                    0.504587                    0.593664  …
4                    0.559633                    0.581637  …
5                    0.522936                    0.575290  …
6                    0.596330                    0.600232  …
7                    0.584098                    0.687275  …
8                    0.541284                    0.600454  …
9                    0.437309                    0.617900  …

  title_word_count  word_count  \
0                9        2668
1                6        7512
2                3        4264
3                9        2869
```

```
4                  9        4840
5                 10        4108
6                 10        7523
7                  9        7790
8                  7        5112
9                 10        2299

                                 most_covered_topic  topic_coverage duration  \
0  http://en.wikipedia.org/wiki/Kernel_density_es…        0.414578      890
1         http://en.wikipedia.org/wiki/Interest_rate        0.292437     2850
2   http://en.wikipedia.org/wiki/Normal_distribution        0.271424     1680
3  http://en.wikipedia.org/wiki/Matrix_(mathematics)        0.308092     1270
4             http://en.wikipedia.org/wiki/Transport        0.414219     2000
5                  http://en.wikipedia.org/wiki/Time        0.338298     1830
6         http://en.wikipedia.org/wiki/Phase_diagram        0.438675     3060
7  http://en.wikipedia.org/wiki/Rank_(linear_alge…        0.212774     3910
8      http://en.wikipedia.org/wiki/Machine_learning        0.298585     2980
9                http://en.wikipedia.org/wiki/Photon        0.300573     1040

  lecture_type  has_parts  speaker_speed silent_period_rate  median_engagement
0           vl      False       2.997753           0.000000           0.502923
1           vl      False       2.635789           0.000000           0.011989
2          vit      False       2.538095           0.000000           0.041627
3           vl      False       2.259055           0.000000           0.064989
4          vkn      False       2.420000           0.000000           0.052154
5           vl      False       2.244809           0.000000           0.256300
6           vl      False       2.458497           0.196126           0.032233
7           vl      False       1.992327           0.289208           0.015063
8           vl      False       1.715436           0.000000           0.025882
9           vl      False       2.210577           0.000000           0.031795

[10 rows x 22 columns]
```

[83]:
```python
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.

1.4.1 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

2 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.

2.1 *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	Discrete Despite conceptually it may be continuous if it is measuring time, the data shows integers only, so assumed discrete
title_word_count	Discrete
word_count	Discrete
most_covered_topic	Categorical
topic_coverage	Continuous
duration	Discrete Despite conceptually it may be continuous if it is measuring time, the data shows integers only, so assumed discrete
lecture_type	Categorical
has_parts	Categorical
speaker_speed	Continuous
silent_period_rate	Continuous
median_engagement	Continuous

2.2 *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. Numerical Features Statistics and Distribution: understand the central tendency and spread as well as visualise the distribution for any possible outliers
2. Categorical Features Frequency Distribution: visualise distribution for imbalances and sparsity
3. Target Variable Distribution: visualise the distribution of target variable to inform adequate modelling approach
4. Correlation Analysis: understand possible relationships between variables to identify collinearity and potential target predictor
5. Missing Values Analysis: identify and decide appropriate imputation techniques

[84]:
```python
# All the imports
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Random seed for reproducibility
RANDOM_SEED = 200
np.random.seed(RANDOM_SEED)
```

Continuous Features Statistics and Distribution

[85]:
```python
target_column = "median_engagement"
features_columns = lectures.columns.drop(target_column).tolist()
numerical_features_columns = lectures[features_columns].
↪select_dtypes(include=[np.number]).columns.tolist()
categorical_features_columns = [col for col in features_columns if col not in␣
↪numerical_features_columns]

print(f"""
Numerical features: {len(numerical_features_columns)}
```

```
    {numerical_features_columns}

Categorical features: {len(categorical_features_columns)}
    {categorical_features_columns}

Target column: {target_column}


""")

target = lectures[target_column]
features = lectures[features_columns]
numerical_features = lectures[numerical_features_columns]
categorical_features = lectures[categorical_features_columns]
```

```
Numerical features: 17
    ['auxiliary_rate', 'conjugate_rate', 'normalization_rate', 'tobe_verb_rate',
'preposition_rate', 'pronoun_rate', 'document_entropy', 'easiness',
'fraction_stopword_coverage', 'fraction_stopword_presence', 'freshness',
'title_word_count', 'word_count', 'topic_coverage', 'duration', 'speaker_speed',
'silent_period_rate']

Categorical features: 4
    ['subject_domain', 'most_covered_topic', 'lecture_type', 'has_parts']

Target column: median_engagement
```

[86]:
```
# Summary statistics for numerical features
numerical_features.describe().T
```

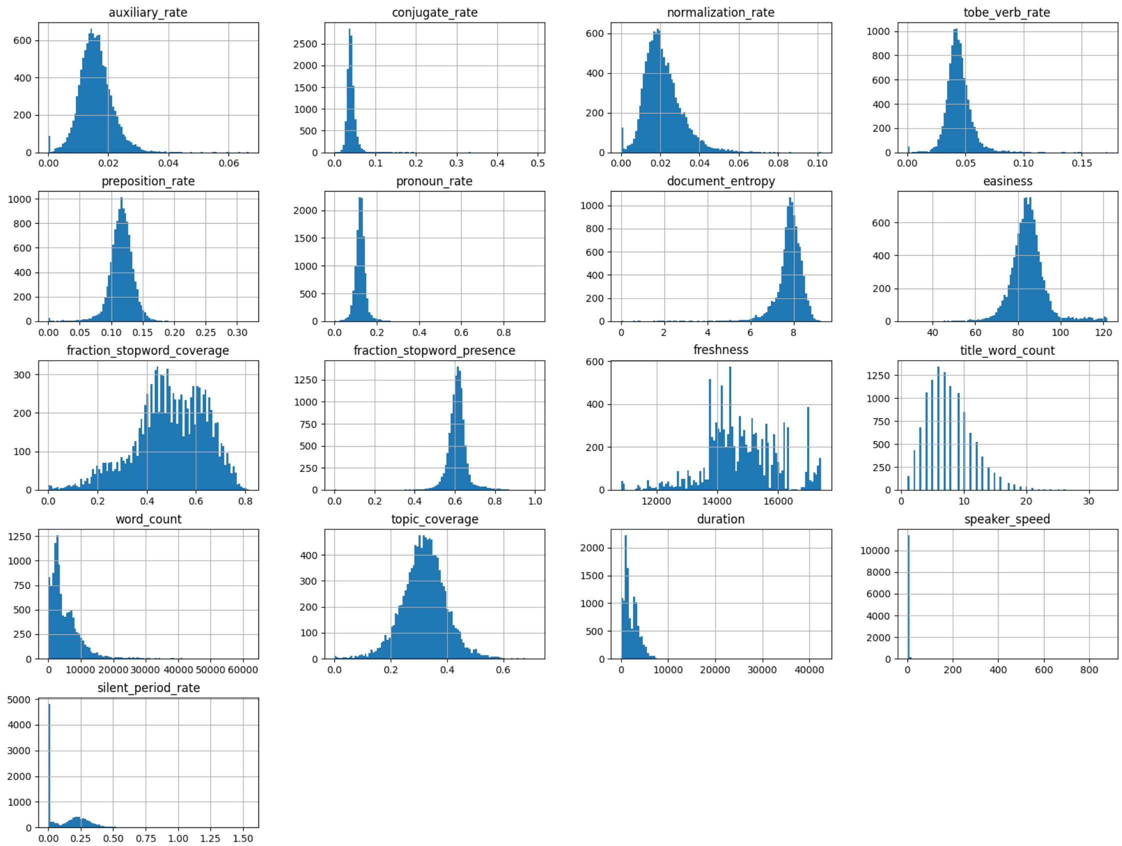
[86]:

	count	mean	std	min \
auxiliary_rate	11548.0	0.015811	0.005465	0.000000
conjugate_rate	11548.0	0.040899	0.013378	0.000000
normalization_rate	11548.0	0.021373	0.009630	0.000000
tobe_verb_rate	11548.0	0.044119	0.010820	0.000000
preposition_rate	11548.0	0.116088	0.018797	0.000000
pronoun_rate	11548.0	0.123087	0.029803	0.000000
document_entropy	11548.0	7.791438	0.696269	0.000000
easiness	11548.0	84.730652	8.330281	28.210966
fraction_stopword_coverage	11548.0	0.494730	0.144478	0.000000
fraction_stopword_presence	11548.0	0.612392	0.051470	0.000000
freshness	11548.0	14819.083824	1204.580175	10830.000000
title_word_count	11548.0	7.705230	3.775191	1.000000
word_count	11548.0	5347.890890	5413.868119	1.000000

```
topic_coverage             11548.0     0.318766     0.079595     0.000000
duration                   11548.0  2137.064427  1540.859597    20.000000
speaker_speed              11548.0     2.495543     8.337300     0.000302
silent_period_rate         11548.0     0.146467     0.172548     0.000000

                                  25%           50%           75%  \
auxiliary_rate               0.012369      0.015441      0.018846
conjugate_rate               0.034486      0.039472      0.044937
normalization_rate           0.014932      0.019856      0.026235
tobe_verb_rate               0.038092      0.043179      0.049180
preposition_rate             0.106400      0.116836      0.127139
pronoun_rate                 0.109159      0.122261      0.135148
document_entropy             7.590194      7.876202      8.164841
easiness                    80.345858     84.429664     88.382702
fraction_stopword_coverage   0.409786      0.498471      0.608563
fraction_stopword_presence   0.589435      0.613258      0.634699
freshness                14070.000000  14750.000000  15600.000000
title_word_count             5.000000      7.000000     10.000000
word_count                2102.000000   3642.500000   7188.000000
topic_coverage               0.271512      0.320175      0.367327
duration                  1020.000000   1630.000000   3050.000000
speaker_speed                1.972993      2.265999      2.540229
silent_period_rate           0.000000      0.101739      0.250981

                                  max
auxiliary_rate               0.066667
conjugate_rate               0.492754
normalization_rate           0.101990
tobe_verb_rate               0.172414
preposition_rate             0.318182
pronoun_rate                 0.947967
document_entropy             9.278573
easiness                   122.032000
fraction_stopword_coverage   0.813456
fraction_stopword_presence   1.000000
freshness                17430.000000
title_word_count            33.000000
word_count               61653.000000
topic_coverage               0.712735
duration                 42520.000000
speaker_speed              881.000000
silent_period_rate           1.542962
```

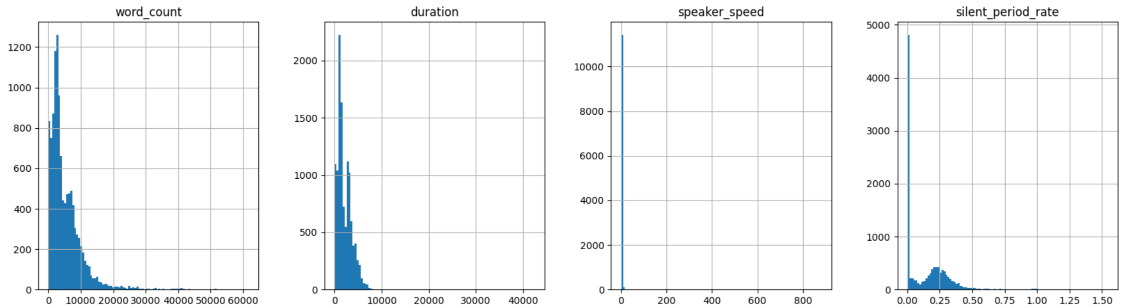
[87]:

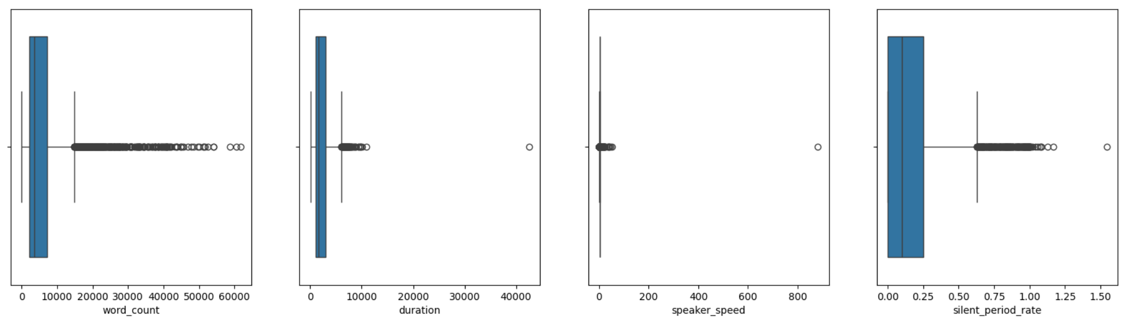
```
numerical_features.hist(bins=100, figsize=(20, 15))
plt.show()
```

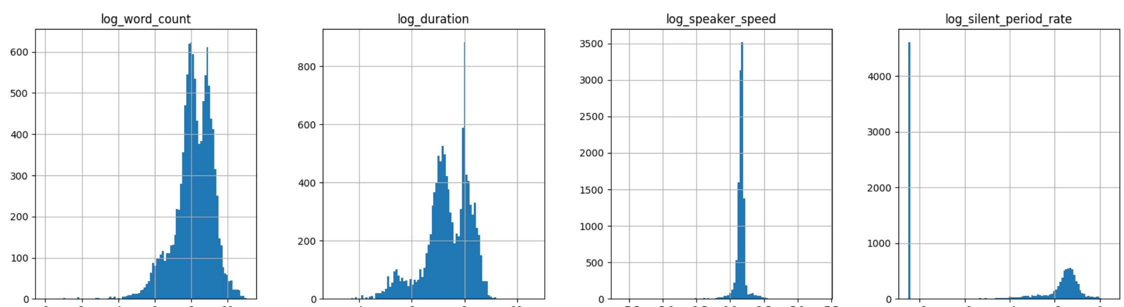


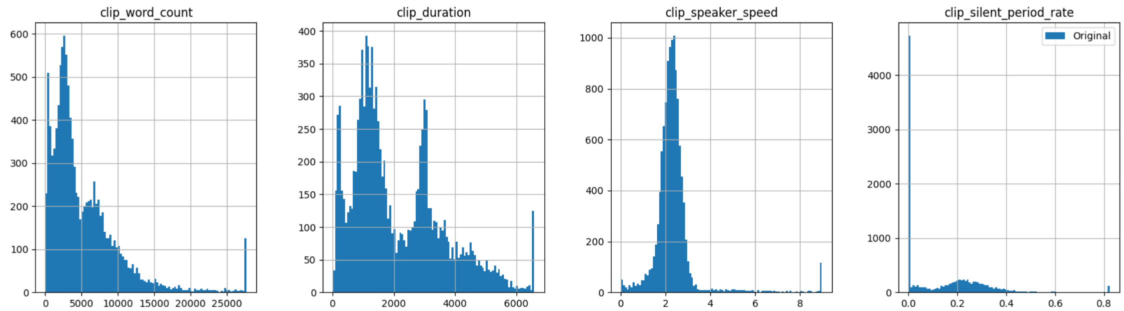
[88]:
```python
revised_columns = ["word_count", "duration", "speaker_speed",␣
↪"silent_period_rate"]

def safe_log(x):
    # When x is a pandas series and the log of 0 takes log of smallest number␣
↪in the series
    minimum = x[x > 0].min()
    return np.log(x.clip(lower=minimum))
def clip_at_99(x):
    return x.clip(upper=x.quantile(0.99))


# Plotting the original, box plot, and transformed features
lectures[revised_columns].hist(layout=(1, 4), bins=100, figsize=(20, 5))
fig, axes = plt.subplots(1, 4, figsize=(20, 5))
for i, col in enumerate(revised_columns):
    sns.boxplot(x=col, data=lectures, ax=axes[i])
lectures[revised_columns].apply(safe_log).rename(columns={col: "log_"+col for␣
↪col in revised_columns}).hist(layout=(1, 4), bins=100, figsize=(20, 5))
```

```
lectures[revised_columns].apply(clip_at_99).rename(columns={col: "clip_"+col for col in revised_columns}).hist(layout=(1, 4), bins=100, figsize=(20, 5))
plt.legend(["Original", "Log"])
plt.show()
```



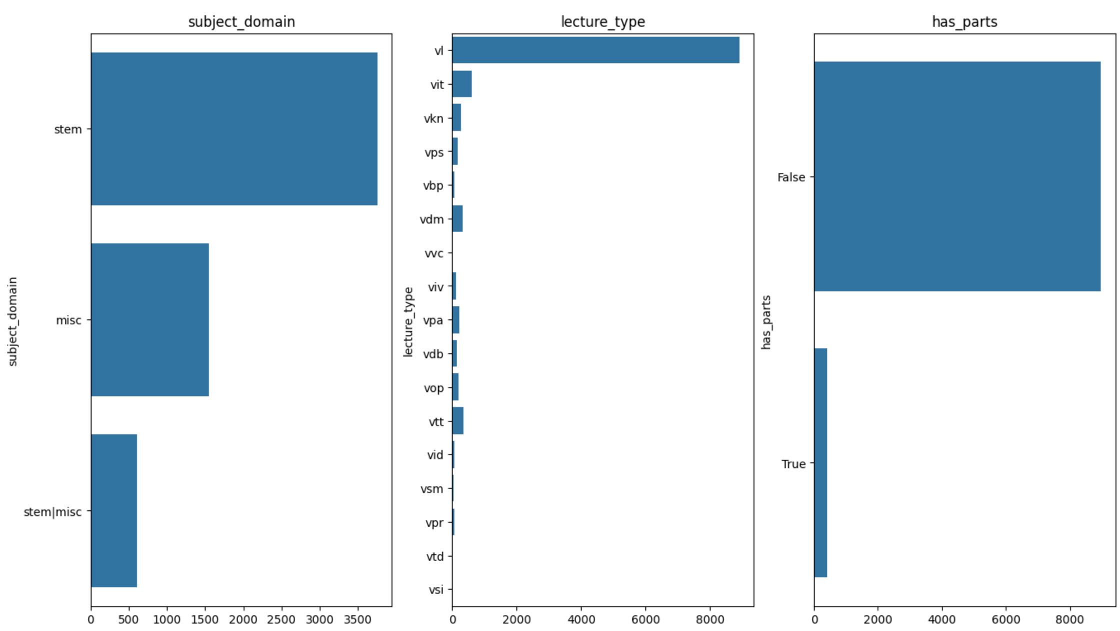






Categorical Features Frequency Distribution

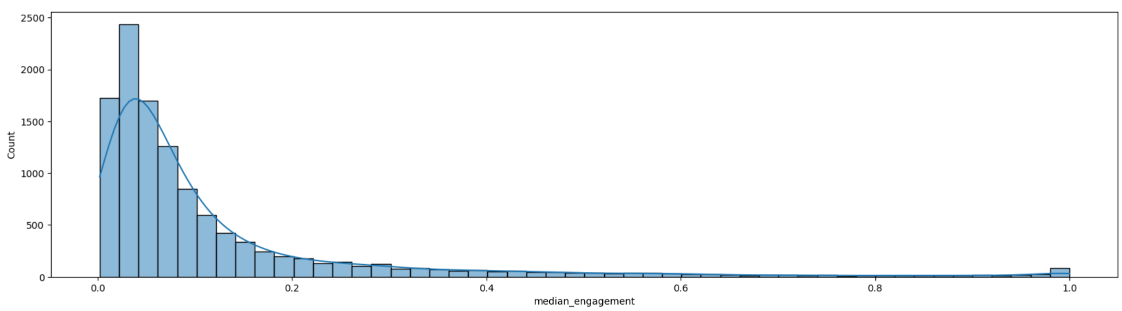
[89]:
```python
# Category distribution
topic_column = "most_covered_topic"
plottable = [col for col in categorical_features_columns if col != topic_column]
plt.figure(figsize=(16, 9))
for i, col in enumerate(plottable):
    plt.subplot(1, 3, i + 1)
    sns.countplot(y=categorical_features[col])
    plt.title(col)
plt.show()

print(f"Most covered topic: {categorical_features[topic_column].value_counts()}")
```



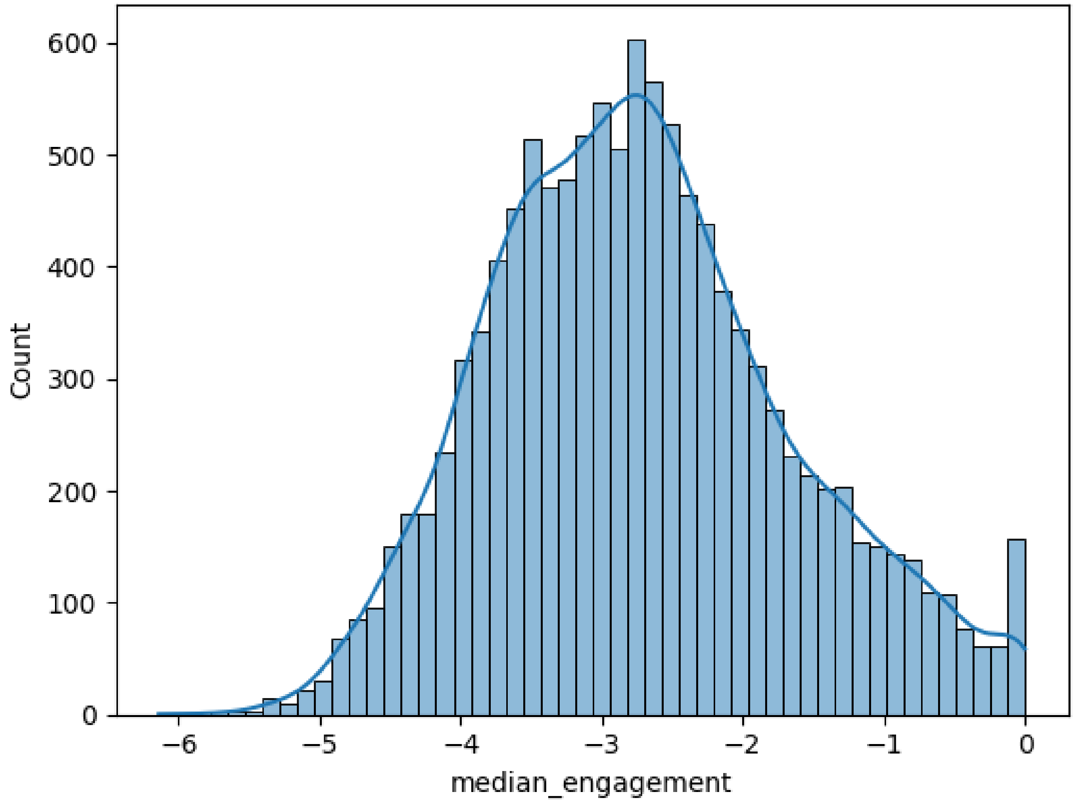
```
Most covered topic: most_covered_topic
http://en.wikipedia.org/wiki/Time                     1006
http://en.wikipedia.org/wiki/Scientific_method         612
http://en.wikipedia.org/wiki/Science                   439
http://en.wikipedia.org/wiki/Algorithm                 306
http://en.wikipedia.org/wiki/Technology                280
                                                      …
http://en.wikipedia.org/wiki/Diffusion_map               1
http://en.wikipedia.org/wiki/Heat                        1
http://en.wikipedia.org/wiki/Breast_cancer               1
http://en.wikipedia.org/wiki/Political_party             1
http://en.wikipedia.org/wiki/Linear_map                  1
Name: count, Length: 2096, dtype: int64
```

Target Variable Distribution

[90]:
```python
# Target distribution
plt.figure(figsize=(20, 5))
sns.histplot(target, bins=50, kde=True)
plt.show()
```

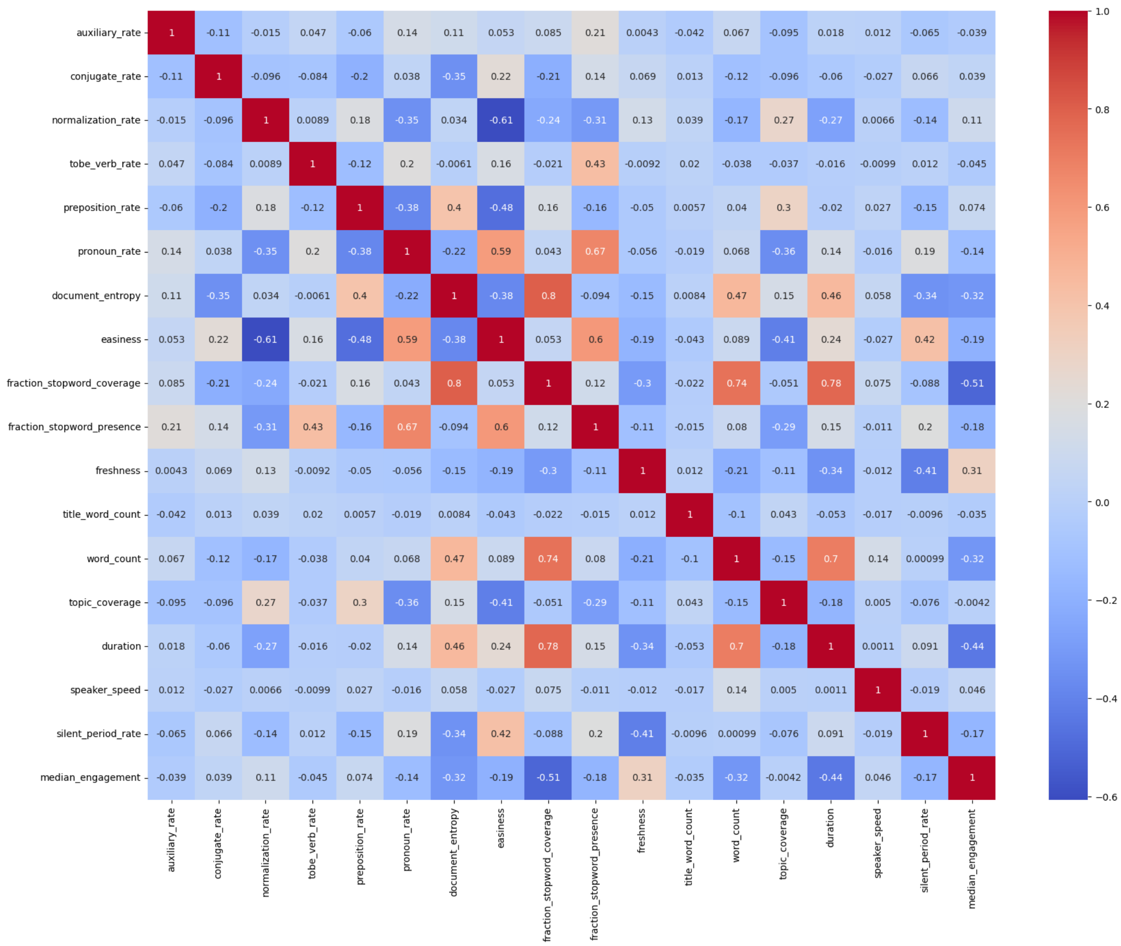


[91]:
```python
# Log transformation of the target
sns.histplot(lectures[target_column].apply("log"), bins=50, kde=True)
plt.show()
```



Correlation Analysis

[92]:
```
# Correlation Matrix
correlation_matrix = pd.concat([numerical_features, target], axis=1).corr()
plt.figure(figsize=(20, 15))
sns.heatmap(correlation_matrix, annot=True, cmap="coolwarm")
plt.show()
```



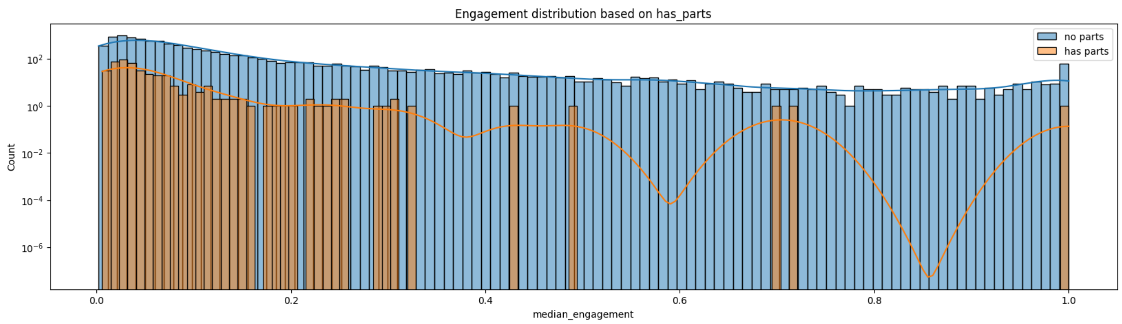
Missing Values Analysis

[93]:
```
lectures.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 11548 entries, 0 to 11547
Data columns (total 22 columns):
 #   Column                      Non-Null Count  Dtype
---  ------                      --------------  -----
 0   auxiliary_rate              11548 non-null  float64
 1   conjugate_rate              11548 non-null  float64
 2   normalization_rate          11548 non-null  float64
 3   tobe_verb_rate              11548 non-null  float64
 4   preposition_rate            11548 non-null  float64
 5   pronoun_rate                11548 non-null  float64
 6   document_entropy            11548 non-null  float64
 7   easiness                    11548 non-null  float64
 8   fraction_stopword_coverage  11548 non-null  float64
 9   fraction_stopword_presence  11548 non-null  float64
```

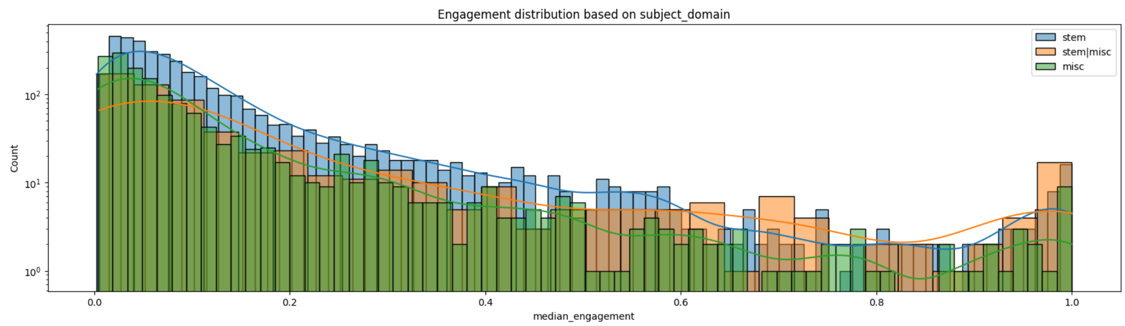
```
 10   subject_domain               5913 non-null   object
 11   freshness                    11548 non-null  int64
 12   title_word_count             11548 non-null  int64
 13   word_count                   11548 non-null  int64
 14   most_covered_topic           11548 non-null  object
 15   topic_coverage               11548 non-null  float64
 16   duration                     11548 non-null  int64
 17   lecture_type                 11548 non-null  object
 18   has_parts                    9396 non-null   object
 19   speaker_speed                11548 non-null  float64
 20   silent_period_rate           11548 non-null  float64
 21   median_engagement            11548 non-null  float64
dtypes: float64(14), int64(4), object(4)
memory usage: 1.9+ MB
```

[94]:
```
plt.figure(figsize=(20, 5))
df = lectures[["has_parts", target_column]].dropna()
sns.histplot(df[df["has_parts"] == False][target_column], kde=True, label="no␣
↪parts")
sns.histplot(df[df["has_parts"] == True][target_column], kde=True, label="has␣
↪parts")
plt.yscale("log")
plt.legend()
plt.title("Engagement distribution based on has_parts")
plt.show()
```



[95]:
```
plt.figure(figsize=(20, 5))
df = lectures[["subject_domain", target_column]].dropna()
sns.histplot(df[df["subject_domain"] == "stem"][target_column], kde=True,␣
↪label="stem")
sns.histplot(df[df["subject_domain"] == "stem|misc"][target_column], kde=True,␣
↪label="stem|misc")
sns.histplot(df[df["subject_domain"] == "misc"][target_column], kde=True,␣
↪label="misc")
plt.yscale("log")
```

```
plt.legend()
plt.title("Engagement distribution based on subject_domain")
plt.show()
```



Question Summarise the key findings of your analyses.

Numerical Features:

- They mostly look like near-normal distributions, which is good because models like linear regression would assume normal distribution.
- Presence of outliers, especially marked in `word_count`, `duration`, `speaker_speed`, and `silent_period_rate`. In order to handle, log transform and clipping are explored visually. On the other hand, high skewness in rate features is acceptable since they are normalised.
- We also observe 0 values in some of the rates which could possibly be abnormal. But we don't have a convenient way to impute them as data generation process is unknown, so they are left as it is for now.

Categorical Features:

- We will need to encode them after, either label-encoding, one-hot-encoding, or target-encoding.
- Compared to other three categorical features, `most_covered_topic` is notably more sparse in values, up to 2k topics.
- Non-split video lecture dominates among `lecture_type` and `has_parts`, aligning the common form of teaching in online resources by conveying one full video lecture on the matter.

Target Variable:

- This is a naturally right-skewed distribution, with a high portion of `median_engagement` between 0 and 0.2. It decreases until the end, where a small peak appears around 1.0. If applied a log transformation, it approaches normal distribution.
- Probably learners decide early whether the video is right for them. They may quit after realising this is not the content they expect, maybe due to content or difficulty.
- Remaining learners tend to complete the video, because they understand that this is the right video for them. This could explain the small peak at 1.0.

Correlation Analysis:

- At this stage, we haven't encoded categorical features yet, so only the numerical ones are visualised.
- From correlation matrix, there are few interesting correlations between features:
  - Complex Language and Topic Coverage
    1. Suffixes and prepositions associated with complex language.`easiness` negatively correlates with `normalization_rate` and `preposition_rate` (which could be indicative of long sentences). But complex language is also indicative of deeper topic coverage, shown by negative correlation between `topic_coverage` and `easiness`.
    2. Stopping and pronouns, in contrast, increases easiness. Notice `easiness` positively correlates with `pronoun_rate`, `fraction_stopword_presence`, `silent_period_rate`. High stop word presence is probably necessary to support the linkage between phrases. Pronouns carry singleton high-value semantics, which is probably why it is helpful in understanding content.
  - Cohesion and Entropy
    1. Usage of less-informative words and length reduces cohesion in video. According to Bendersky et al., 2011, document entropy is lower when more cohesive and single-topic focused the document is. `document_entropy` is positively correlated with `fraction_stopword_coverage`, `preposition`, `word_count`, and `duration`. Especially usage of variety of stop words which has a strong effect.
    2. `duration` and `word_count` are strongly associated which means effect in one probably has same effect in the other. Documents tend to be less cohesive when longer.
    3. `conjugate_rate` and `silent_period`, in contrast, could increases cohesiveness, maybe explained as conjugate words has an strong effect in the logic of sentences and silent period reduces out-of-topic mentions.
  - Word Type
    1. On word type level, positive correlation between `pronoun_rate` and `fraction_stopword_presence` are suggesting that these two are commonly used together. Additionally, stop words also positively correlates with `tobe_verb_rate`, whereas pronouns negatively correlate with usage of complex suffixes and prepositions, as suggested in point 1.
  - Freshness and Tendency
    1. The tendency suggests trends towards more concise and denser information in videos. `freshness` could be approximated as a measure of how recent the videos are, thus picturing the evolution in time of the videos. It negatively correlates with `fraction_stopword_coverage`, using only the ones necessary.
    2. It also negatively correlates with `duration`, and `silent_period_rate`. Probably suggesting the tendency of making shorter videos with no interruptions in the middle, maybe trying to continuously stimulate learners with dense packs of information. Lack of interruptions could be interpreted as invitation for learners to stop whenever they want, or merely to maintain the continuous stimuli that engages the learners. And in fact, it achieves high engagement by positively correlating with `median_engagement`.
  - Engagement
    1. About **Target Variable**, `median_engagement`, videos tended to be more engaging as they evolve in time (`freshness`) and many of its negatively correlated characteristics match with freshness too. Explicitly we want to com-

ment `fraction_stopword_coverage`, which has highest negative correlation. One may think hypothesise that videos became more concise while being integral, but we see that stop word coverage correlated with less cohesive documents (`document_entropy`). This indicates that engagement somehow is encouraged by less cohesion and more loosely detached videos, indeed it negatively correlates with document entropy.

Missing Values

- We observe a significant portion of missing values in `subject_domain` and moderately in `has_parts`. These missing values are probably too many to be removed, instead we could impute either static values or predicted values in them.
- Effects on engagement of known values in missing features are visualised. `has_parts` doesn't exhibit high difference whereas `subject_domain` does in some degree.

This reflective observation and analysis sets the stage for informed preprocessing and model development strategies afterwards.

2.3 *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 Preprocessing that we want to include:

0. Streamline the url prefix in most covered topics.
1. Outlier Handling: fields with outliers are either treated with log transformation (when variability naturally increases as value increases) or clipped at 99% (if likely because of rare events and these don't carry significant information).
   - Word Count: log transform. This is a feature that skews due to multiplicative nature and exhibits normal distribution after log transform.
   - Duration: log transform. This is a feature that skews due to multiplicative nature and exhibits normal distribution after log transform.
   - Speaker Speed: clip at 99% quantile. This alleviates largely outliers.
   - Silent Period Rate: clip at 99% quantile. This alleviates largely outliers.
2. Missing Value Imputation:
   - Subject Domain: establish a new "unknown" entry. Since earlier visualisations show how subject domain may influence in engagement, and we want to avoid bold assumptions of other features predicting this feature.
   - Has Parts: impute most frequent mode "True". As videos predominantly don't have parts.

What's handled in the next section 1. Encoding: to give appropriate numerical representations to categorical data. One-hot-encoding can be used for categories with manageable size and target encoding for sparse and disproportionate categories. 2. Scaling: standardise feature space for models that benefits from normally distributed data like linear models; min-max scale it if algorithms are sensitive to feature scale e.g. SVM, Neural Networks.

Future possible preprocessing, suitable when preliminary results are obtained, include feature engineering, feature selection, and dimension reductions. Also, more powerful models would benefit from experimenting different strategies in handling data.

[96]:
```
def safe_log(x):
    # When x is a pandas series and the log of 0 takes log of smallest number␣
 ↪in the series
    minimum = x[x > 0].min()
    return np.log(x.clip(lower=minimum))
def clip_at_99(x):
    return x.clip(upper=x.quantile(0.99))
```

[97]:
```
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 coursework

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

    LOG_COLUMNS = ["word_count", "duration"]
    CLIP_COLUMNS = ["speaker_speed", "silent_period_rate"]
    MODE_IMPUTE_COLUMNS = ["has_parts"]
    UNKNOWN_IMPUTE_COLUMNS = ["subject_domain"]

    preprocessed_dataset = dataset.copy()

    # Streamline most_covered_topic
    preprocessed_dataset["most_covered_topic"] =␣
 ↪preprocessed_dataset["most_covered_topic"].apply(lambda x: x.split("/")[-1].
 ↪lower())

    # Log transformation
```

```
    preprocessed_dataset[LOG_COLUMNS] = preprocessed_dataset[LOG_COLUMNS].
↪apply(safe_log)
    preprocessed_dataset = preprocessed_dataset.rename(columns={col: "log_"+col␣
↪for col in LOG_COLUMNS})

    # Clip at 99th percentile
    preprocessed_dataset[CLIP_COLUMNS] = preprocessed_dataset[CLIP_COLUMNS].
↪apply(clip_at_99)

    # Mode imputation
    preprocessed_dataset[MODE_IMPUTE_COLUMNS] =␣
↪preprocessed_dataset[MODE_IMPUTE_COLUMNS].
↪fillna(preprocessed_dataset[MODE_IMPUTE_COLUMNS].mode().iloc[0])

    # Unknown imputation
    preprocessed_dataset[UNKNOWN_IMPUTE_COLUMNS] =␣
↪preprocessed_dataset[UNKNOWN_IMPUTE_COLUMNS].fillna("unknown")

    return preprocessed_dataset
```

[98]:
```
preprocessed_lectures = preprocess_lecture_dataset(lectures)
```

```
C:\Users\lifeng\AppData\Local\Temp\ipykernel_19992\1507714059.py:31:
FutureWarning: Downcasting object dtype arrays on .fillna, .ffill, .bfill is
deprecated and will change in a future version. Call
result.infer_objects(copy=False) instead. To opt-in to the future behavior, set
`pd.set_option('future.no_silent_downcasting', True)`
  preprocessed_dataset[MODE_IMPUTE_COLUMNS] = preprocessed_dataset[MODE_IMPUTE_C
OLUMNS].fillna(preprocessed_dataset[MODE_IMPUTE_COLUMNS].mode().iloc[0])
```

[99]:
```
preprocessed_lectures.head()
```

[99]:
```
   auxiliary_rate  conjugate_rate  normalization_rate  tobe_verb_rate  \
0        0.013323        0.033309            0.034049        0.035159
1        0.014363        0.030668            0.018763        0.036749
2        0.019028        0.033242            0.030720        0.037827
3        0.023416        0.042700            0.016873        0.046832
4        0.021173        0.041531            0.023412        0.038884

   preposition_rate  pronoun_rate  document_entropy   easiness  \
0          0.121392      0.089563          7.753995  75.583936
1          0.095885      0.103002          8.305269  86.870523
2          0.118294      0.124255          7.965583  81.915968
3          0.122590      0.104339          8.142877  80.148937
4          0.130700      0.102606          8.161250  76.907549

   fraction_stopword_coverage  fraction_stopword_presence  …  \
```

```
0                    0.428135                     0.553664  …
1                    0.602446                     0.584498  …
2                    0.525994                     0.605685  …
3                    0.504587                     0.593664  …
4                    0.559633                     0.581637  …

  title_word_count  log_word_count         most_covered_topic  topic_coverage  \
0                9        7.889084  kernel_density_estimation        0.414578
1                6        8.924257              interest_rate        0.292437
2                3        8.357963        normal_distribution        0.271424
3                9        7.961719       matrix_(mathematics)        0.308092
4                9        8.484670                  transport        0.414219

  log_duration  lecture_type  has_parts speaker_speed  silent_period_rate  \
0     6.791221            vl      False      2.997753                 0.0
1     7.955074            vl      False      2.635789                 0.0
2     7.426549           vit      False      2.538095                 0.0
3     7.146772            vl      False      2.259055                 0.0
4     7.600902           vkn      False      2.420000                 0.0

   median_engagement
0           0.502923
1           0.011989
2           0.041627
3           0.064989
4           0.052154

[5 rows x 22 columns]
```

2.4 *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.

[100]:
```python
from sklearn.preprocessing import StandardScaler

def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

[101]:
```python
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
    """

    BINARY_ENCODE_COLUMNS = ["has_parts"]
    ONE_HOT_ENCODE_COLUMNS = ["subject_domain"]
    TARGET_ENCODE_COLUMNS = ["most_covered_topic", "lecture_type"]
    TARGET_COLUMN = "median_engagement"

    X = preprocessed_dataset.drop(columns=[TARGET_COLUMN])
    y = preprocessed_dataset[TARGET_COLUMN]

    # Binary encoding
    X[BINARY_ENCODE_COLUMNS] = X[BINARY_ENCODE_COLUMNS].astype(int)

    # One-hot encoding
    X = pd.get_dummies(X, columns=ONE_HOT_ENCODE_COLUMNS)

    # Target encoding with blending which avoids overfitting by controlling how␣
↪much the encoding should be influenced by the target based on the number of␣
↪samples
    # Enc(i) = (sigmoid(n_i - 1) * mean(y_i) + (1 - sigmoid(n_i - 1)) *␣
↪global_mean) where n_i is the number of samples in the category i and y_i is␣
↪the target mean of category i
    for col in TARGET_ENCODE_COLUMNS:
        target_mean = preprocessed_dataset.groupby(col)[TARGET_COLUMN].mean()
        category_count = preprocessed_dataset[col].value_counts()
        global_mean = preprocessed_dataset[TARGET_COLUMN].mean()
        X[col] = X[col].map(lambda x: sigmoid(category_count[x] - 1) *␣
↪target_mean[x] + (1 - sigmoid(category_count[x] - 1)) * global_mean)

    # Scaling
    scaler = StandardScaler()
    scaler.fit(X)
    X = pd.DataFrame(scaler.transform(X), columns=X.columns)
```

```python
    preprocessed_dataset = pd.concat([X, y], axis=1)

    return preprocessed_dataset, X, y
```

[102]:
```python
final_dataset, full_X, full_y = prepare_final_dataset(preprocessed_lectures)
```

[103]:
```python
full_X
```

[103]:
```
       auxiliary_rate  conjugate_rate  normalization_rate  tobe_verb_rate  \
0           -0.455211       -0.567377            1.316333       -0.828105
1           -0.264918       -0.764801           -0.271093       -0.681119
2            0.588676       -0.572388            0.970620       -0.581560
3            1.391652        0.134649           -0.467333        0.250741
4            0.981136        0.047277            0.211712       -0.483806
…                 …               …                  …              …
11543       -0.212095       -0.129517            1.028419       -0.379653
11544        2.173640       -0.364648           -0.769130        0.383880
11545        0.002506       -1.092838           -0.299858       -0.739620
11546       -1.813709        0.911919            1.150126        0.557175
11547       -0.762916       -0.766685            0.186999        0.398893

       preposition_rate  pronoun_rate  document_entropy  easiness  \
0              0.282186     -1.124909         -0.053779 -1.098056
1             -1.074835     -0.673968          0.738010  0.256890
2              0.117406      0.039175          0.250123 -0.337900
3              0.345922     -0.629113          0.504768 -0.550031
4              0.777440     -0.687263          0.531157 -0.939157
…                   …             …                …         …
11543         -0.224789      0.181179         -0.013823 -0.493452
11544         -0.752919      0.400262          0.013398  1.122408
11545         -0.487536      0.014145          0.638273  1.253604
11546          0.101431     -0.962771         -1.459191 -1.281831
11547          1.145412     -0.389584          0.127578 -0.415118

       fraction_stopword_coverage  fraction_stopword_presence  …  \
0                       -0.460956                   -1.141065  …
1                        0.745592                   -0.541976  …
2                        0.216404                   -0.130312  …
3                        0.068232                   -0.363885  …
4                        0.449247                   -0.597566  …
…                             …                          …  …
11543                    0.025897                   -0.013033  …
11544                    0.068232                    0.822822  …
11545                    1.613461                    0.122273  …
11546                   -2.112023                   -1.008788  …
11547                    0.533917                    0.506100  …
```

```
       topic_coverage  log_duration  lecture_type  has_parts  speaker_speed  \
0            1.203785     -0.646609     -0.263265  -0.194275       0.600978
1           -0.330800      0.675140     -0.263265  -0.194275       0.256250
2           -0.594817      0.074911     -0.712883  -0.194275       0.163208
3           -0.134113     -0.242822     -0.263265  -0.194275      -0.102545
4            1.199283      0.272919     -0.603900  -0.194275       0.050736
…                 …             …            …        …            …
11543        0.417711      0.019502     -0.263265  -0.194275      -0.165336
11544        0.763961      0.548825     -0.263265  -0.194275      -0.749266
11545       -0.918190      0.974630     -0.773312   5.147353       3.030285
11546        3.092730     -2.831277      4.735661  -0.194275       0.214839
11547        0.069226      0.952997     -0.263265  -0.194275       0.090820

       silent_period_rate  subject_domain_misc  subject_domain_stem  \
0               -0.868131            -0.394033             1.439761
1               -0.868131            -0.394033            -0.694560
2               -0.868131            -0.394033             1.439761
3               -0.868131            -0.394033             1.439761
4               -0.868131            -0.394033            -0.694560
…                     …                  …                    …
11543            0.296181            -0.394033             1.439761
11544            1.154448            -0.394033            -0.694560
11545            1.338114            -0.394033             1.439761
11546           -0.837479            -0.394033            -0.694560
11547            0.475802            -0.394033             1.439761

       subject_domain_stem|misc  subject_domain_unknown
0                      -0.23472               -0.976209
1                      -0.23472                1.024370
2                      -0.23472               -0.976209
3                      -0.23472               -0.976209
4                      -0.23472                1.024370
…                           …                       …
11543                  -0.23472               -0.976209
11544                  -0.23472                1.024370
11545                  -0.23472               -0.976209
11546                  -0.23472                1.024370
11547                  -0.23472               -0.976209

[11548 rows x 24 columns]
```

[104]:
```
full_X.shape
```

[104]:
```
(11548, 24)
```

[105]:
```
full_y
```

[105]:
```
0        0.502923
1        0.011989
2        0.041627
3        0.064989
4        0.052154
           …
11543    0.044655
11544    0.038525
11545    0.012572
11546    0.998364
11547    0.032745
Name: median_engagement, Length: 11548, dtype: float64
```

Let us now save the final data

[106]:
```
full_X.to_csv("features_final.csv", index=False)
np.save("labels_final.npy", full_y.to_numpy())
```

3 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.

[107]:
```
full_X = pd.read_csv("features_final.csv")
full_y = np.load("labels_final.npy")

# 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")
```

3.1 *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

[108]:
```
from sklearn.linear_model import Ridge
from sklearn.kernel_ridge import KernelRidge
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import make_scorer
```

[109]:
```
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
        :
    """

    ridge_model = Ridge(**hyperparams)

    ridge_model.fit(X, y)

    return ridge_model
```

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

[110]:
```
hyperparams = {
    "alpha": 1.0,
}
```

[111]:
```
temp_ridge_model = train_ridge_model(full_X, full_y, hyperparams)
```

3.2 *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.

[112]:
```
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
    """

    # Your Code Here
    norm_sq = np.linalg.norm(x - x_dash) ** 2
    similarity = np.exp(-gamma * norm_sq)

    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).

[113]:
```
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
    kernel_ridge_model = KernelRidge(kernel=kernel_function,␣
↪kernel_params=kernel_params, **hyperparams)

    kernel_ridge_model.fit(X, y)

    return kernel_ridge_model
```

[114]:
```
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)
```

[115]:
```
temp_y = temp_kernel_ridge_model.predict(full_X)
print(temp_y)
```

```
[0.58578346 0.10051844 0.0819317  … 0.03986811 0.70381138 0.03097646]
```

3.3 *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: Root Mean Square Error

Root Mean Square Error ($RMSE$) provides a quantification for the error between actual and predicted value. It is a common evaluation metric in the form of loss function (Jadon et al., 2022; Li et al., 2024). It doesn't penalise as sever as MSE and gives error in the same units of variable of interest. Moreover, it gives high weights to large errors, particularly informative in this case where outliers can significantly impact results.

Its formula is given by: $\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$

[116]:
```
def rmse(y_actual, y_predicted):
    """
    returns the root mean squared error between the actual and predicted labels

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

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

    # Your Code Here
```

```
    metric = np.sqrt(np.mean((y_actual - y_predicted) ** 2))

    return metric
```

Metric 2: Adjusted R^2

R-squared (R^2) is another family of evaluation metrics that, unlike error metrics, indicates the proportion of variance in the target variable that a model is able to explain from the features. It is sometimes regarded as "goodness of fit" and takes range between (0, 1) where 0 is no explanation and 1 when fully explainable. While also used in the literature (Wu et al., 2018), it favours addition of features, regardless of its contribution (Plevris et al., 2022). Therefore, we use *Adjusted* R^2 instead.

Its formula is given by: Adjusted $R^2 = 1 - \left(\frac{(1-R^2)(n-1)}{n-k-1}\right)$,
where $R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$ and,
k is the number of features.

By using a mixture of two complementary views, we aim to conduct a balanced analysis on both effectiveness and explanatory power of the model.

[117]:
```
def adjusted_r_squared(y_actual, y_predicted):
    """
    returns adjusted r-squared between the actual and predicted labels

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

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

    # Your Code Here
    n = len(y_actual)
    k = full_X.shape[1]
    r_squared = 1 - np.sum((y_actual - y_predicted) ** 2) / np.sum((y_actual -␣
↪np.mean(y_actual)) ** 2)
    metric = 1 - (1 - r_squared) * (n - 1) / (n - k - 1)

    return metric
```

3.4 *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.

[118]:
```
# Prepare a preprocessor class to streamline the preprocessing steps and␣
↪separate fitting with transformation
class LecturePreprocessor:
    """
    A class that combines both preprocessing steps and separates fitting with␣
↪transformation.
    """

    def __init__(self):
        self.scaler = None
        self.category_count = {}
        self.target_means = {}
        self.global_mean = None

    def preprocess(self, X):
        """
        Apply streamline preprocessing steps.

        Params:
            X (pandas.DataFrame): DataFrame object that contains the features

        Returns:
            preprocessed_X (pandas.DataFrame): DataFrame object that contains␣
↪the preprocessed features
        """

        preprocessed_X = X.copy()

        # Streamline most_covered_topic
        preprocessed_X["most_covered_topic"] =␣
↪preprocessed_X["most_covered_topic"].apply(lambda x: x.split("/")[-1].
↪lower())

        return preprocessed_X

    def fit(self, X, y):
        """
```

```
        Fits the preprocessor with the training data

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


        TARGET_ENCODE_COLUMNS = ["most_covered_topic", "lecture_type"]

        # Target encoding
        for col in TARGET_ENCODE_COLUMNS:
            target_mean = y.groupby(X[col]).mean()
            category_count = X[col].value_counts()
            self.global_mean = y.mean()
            self.category_count[col] = category_count
            self.target_means[col] = target_mean


    def transform(self, X, fit_scaler=False):
        """
        Transforms the input data with fitted preprocessor

        Params:
            X (pandas.DataFrame): DataFrame object that contains the features

        Returns:
            transformed_X (pandas.DataFrame): DataFrame object that contains␣
↪the transformed features
        """

        LOG_COLUMNS = ["word_count", "duration"]
        CLIP_COLUMNS = ["speaker_speed", "silent_period_rate"]
        MODE_IMPUTE_COLUMNS = ["has_parts"]
        UNKNOWN_IMPUTE_COLUMNS = ["subject_domain"]
        BINARY_ENCODE_COLUMNS = ["has_parts"]
        ONE_HOT_ENCODE_COLUMNS = ["subject_domain"]
        TARGET_ENCODE_COLUMNS = ["most_covered_topic", "lecture_type"]

        transformed_X = X.copy()


        # Log transformation
        transformed_X[LOG_COLUMNS] = transformed_X[LOG_COLUMNS].apply(safe_log)
        transformed_X = transformed_X.rename(columns={col: "log_"+col for col␣
↪in LOG_COLUMNS})
```

```
        # Clip at 99th percentile
        transformed_X[CLIP_COLUMNS] = transformed_X[CLIP_COLUMNS].
↪apply(clip_at_99)

        # Mode imputation
        transformed_X[MODE_IMPUTE_COLUMNS] = transformed_X[MODE_IMPUTE_COLUMNS].
↪fillna(transformed_X[MODE_IMPUTE_COLUMNS].mode().iloc[0])


        # Unknown imputation
        transformed_X[UNKNOWN_IMPUTE_COLUMNS] =␣
↪transformed_X[UNKNOWN_IMPUTE_COLUMNS].fillna("unknown")

        # Binary encoding
        transformed_X[BINARY_ENCODE_COLUMNS] =␣
↪transformed_X[BINARY_ENCODE_COLUMNS].astype(int)

        # One-hot encoding
        transformed_X = pd.get_dummies(transformed_X,␣
↪columns=ONE_HOT_ENCODE_COLUMNS)

        # Target encoding
        for col in TARGET_ENCODE_COLUMNS:
            target_mean = self.target_means[col]
            category_count = self.category_count[col]
            transformed_X[col] = transformed_X[col].map(lambda x:␣
↪sigmoid(category_count.get(x, 1) - 1) * target_mean.get(x, self.global_mean)␣
↪+ (1 - sigmoid(category_count.get(x, 1) - 1)) * self.global_mean)

        # Scaling
        if self.scaler is not None and not fit_scaler:
            transformed_X = pd.DataFrame(self.scaler.transform(transformed_X),␣
↪columns=transformed_X.columns)
        else:
            self.scaler = StandardScaler()
            self.scaler.fit(transformed_X)
            print("Fitting the scaler")
            transformed_X = pd.DataFrame(self.scaler.transform(transformed_X),␣
↪columns=transformed_X.columns)

        return transformed_X
```

[119]:
```
def evaluate_ridge_model(X,y):
    """
    trains the most viable model using the lecture data for median engagement␣
↪prediction to evaluate it using the proposed metrics.
```

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

    # Your Code Here
    hyperparams = {
        "alpha": [0.0001, 0.001, 0.01, 0.1, 1, 10]
    }

    TEST_SIZE = 0.2

    # Train, validation, test split
    X_train, X_test, y_train, y_test = train_test_split(X, y,␣
↪test_size=TEST_SIZE, random_state=RANDOM_SEED)

    # Preprocess the data
    preprocessor = LecturePreprocessor()
    X_train = preprocessor.preprocess(X_train)
    X_test = preprocessor.preprocess(X_test)
    preprocessor.fit(X_train, y_train)
    X_train = preprocessor.transform(X_train, fit_scaler=True)
    X_test = preprocessor.transform(X_test)

    # Train the model and cross validate with grid search to use the best␣
↪hyperparameters
    model = Ridge()
    scorer = make_scorer(rmse, greater_is_better=False)
    grid_search = GridSearchCV(model, hyperparams, scoring=scorer, cv=5)
    grid_search.fit(X_train, y_train)
    best_model = grid_search.best_estimator_
    best_params = grid_search.best_params_

    print(f"Best hyperparameters: {best_params}")

    # Predict
    y_train_pred = best_model.predict(X_train)
    y_test_pred = best_model.predict(X_test)

    # Evaluate
    train_rmse = rmse(y_train, y_train_pred)
    test_rmse = rmse(y_test, y_test_pred)
    train_adj_r_squared = adjusted_r_squared(y_train, y_train_pred)
    test_adj_r_squared = adjusted_r_squared(y_test, y_test_pred)
```

```
    print(f"""
    Results Report:
    Train RMSE: {train_rmse:.4f}
    Test RMSE: {test_rmse:.4f}
    Train Adjusted R-Squared: {train_adj_r_squared:.4f}
    Test Adjusted R-Squared: {test_adj_r_squared:.4f}


    """)
```

[120]:
```
lectures = pd.read_csv(data_path)

X = lectures.drop(columns=[target_column])
y = lectures[target_column]

evaluate_ridge_model(X, y)
```

```
C:\Users\lifeng\AppData\Local\Temp\ipykernel_19992\3992612705.py:82:
FutureWarning: Downcasting object dtype arrays on .fillna, .ffill, .bfill is
deprecated and will change in a future version. Call
result.infer_objects(copy=False) instead. To opt-in to the future behavior, set
`pd.set_option('future.no_silent_downcasting', True)`
  transformed_X[MODE_IMPUTE_COLUMNS] = transformed_X[MODE_IMPUTE_COLUMNS].fillna
(transformed_X[MODE_IMPUTE_COLUMNS].mode().iloc[0])
C:\Users\lifeng\AppData\Local\Temp\ipykernel_19992\3992612705.py:82:
FutureWarning: Downcasting object dtype arrays on .fillna, .ffill, .bfill is
deprecated and will change in a future version. Call
result.infer_objects(copy=False) instead. To opt-in to the future behavior, set
`pd.set_option('future.no_silent_downcasting', True)`
  transformed_X[MODE_IMPUTE_COLUMNS] = transformed_X[MODE_IMPUTE_COLUMNS].fillna
(transformed_X[MODE_IMPUTE_COLUMNS].mode().iloc[0])
```

```
Fitting the scaler
Best hyperparameters: {'alpha': 1}

    Results Report:
    Train RMSE: 0.1102
    Test RMSE: 0.1227
    Train Adjusted R-Squared: 0.5702
    Test Adjusted R-Squared: 0.4848
```

Question

- Is the model exhibiting overfitting? Justify your answer

In order to perform a pragmatic evaluation on the model, we must perform train-test data split to measure generalisability. The full dataset imported in part 2 was preprocessed and transformed based on the full dataset, which may incur data leakage if then train-test split is done. Instead, we reload the raw dataset in this question and split it before the actual preprocessing. Afterwards, the parameters of preprocessing are fitted based only on the train set, while the transformation is applied for all the sets. This procedure reflects real performance of the model without data leakage.

Train set results 0.1102 in RMSE and 0.5702 in Adjusted R^2, whereas evaluation on test set shows RMSE of 0.1227 and Adjusted R^2 of 0.4848.

Yes, the model is exhibiting some degree of overfitting. While it performs well on the training data (intra-distribution), its performance degrades on unseen test data (inter-distribution). Specifically, the model shows approximately an 11.43% increase in RMSE and a 14.98% decrease in Adjusted R^2 when evaluated on the test dataset. This disparity indicates that the model fits the training data better than the test data, suggesting an overfitting issue.

Despite employing a train-test split, using K-Folds Cross-Validation, and implementing ridge regularisation, some degree of overfitting persists, although it's not as pronounced as one might expect without these robust methodologies. The test dataset would better reflect the model's real-world predictive performance in measuring engagement. Further steps, such as more aggressive and diverse regularisations or feature engineering, might help reduce this overfitting even further.

4 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

4.0.1 Dataset

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

[121]:
```
full_X = pd.read_csv("features_seed.csv")
full_y = np.load("labels_seed.npy")
```

4.1 *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.

[122]:
```
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
    X = X.to_numpy() # (n_samples, n_features)

    # Add intercept column
    X = np.concatenate([np.ones((X.shape[0], 1)), X], axis=1) # (n_samples,␣
 ↪n_features + 1)

    y = y # (n_samples,) if not None


    return X, y
```

[123]:
```
X_, y_ = prepare_data_for_training(full_X, full_y)
```

4.2 *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.

[124]:
```
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
    ridge_lambda = hyperparams["lambda"]
    lambda_identity = ridge_lambda * np.identity(X.shape[1])
    _theta = np.linalg.inv(X.T @ X + lambda_identity) @ X.T @ y


    return _theta
```

[125]:
```
hyperparams = {
    "lambda": 0.001
}

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

[126]:
```
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.

[127]:
```
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
        if not self.fitted:
            raise ValueError("Model is not trained yet")

        X, _ = prepare_data_for_training(X)
        predictions = X @ self.theta

        return predictions
```

[128]:
```
hyperparams = {
    "lambda": 0.001
```

```
}

RR = RidgeRegression(hyperparams)
```

[129]:
```
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.

[130]:
```
RR.fit(full_X, full_y)
```

[131]:
```
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,  1.30120135, -0.26920015,
        0.27774184, -0.53857572])}
```

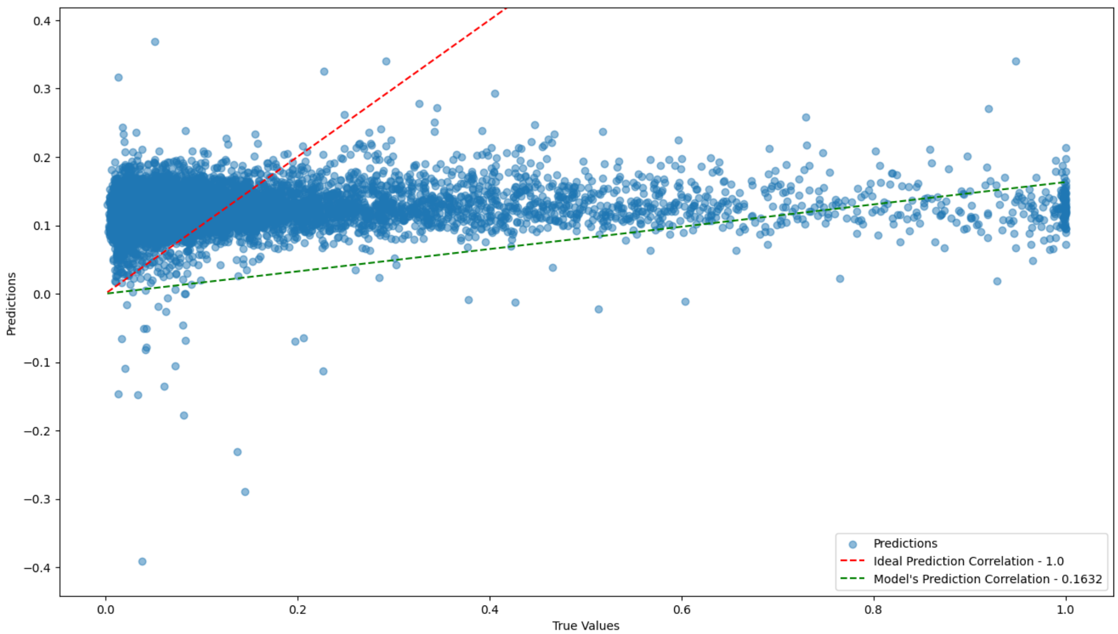
Question:

- 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`

[132]:
```
# Your Code Here
# Make predictions and measure the correlation between the true values and the␣
↪predictions
predictions = RR.predict(full_X)
corr_coef = np.corrcoef(full_y, predictions)[0, 1] # Pearson's Correlation␣
↪Coefficient

print(f"The shape of the predictions: {predictions.shape} \n")
print(f"The Pearson's Correlation Coefficient (linear correlation): {corr_coef:.
↪4f}")

# Plot the predictions against the true values, show the ideal prediction␣
↪correlation and the model's prediction correlation
plt.figure(figsize=(16, 9))
plt.scatter(full_y, predictions, alpha=0.5, label="Predictions")
plt.plot([full_y.min(), full_y.max()], [full_y.min(), full_y.max()], "r--",␣
↪label=f"Ideal Prediction Correlation - 1.0") # Ideal
plt.plot([full_y.min(), full_y.max()], [full_y.min() * corr_coef, full_y.max()␣
↪* corr_coef], "g--", label=f"Model's Prediction Correlation - {corr_coef:.
↪4f}") # Model's
plt.xlabel("True Values")
```

```
plt.ylabel("Predictions")
plt.ylim(predictions.min()-0.05, predictions.max()+0.05) # Limit the y-axis to␣
 ↪show the correlation better, 0.05 for padding
plt.legend()
plt.show()
```

```
The shape of the predictions: (11548,)

The Pearson's Correlation Coefficient (linear correlation): 0.1632
```



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

Initially, the model predicts using the full feature set, and these predictions are compared to actual values to measure their linear correlation. To quantify this relationship, Pearson's Correlation Coefficient was used, a standard method for assessing the degree of linear correlation between two variables.

The scatter plot of predictions against actual values visually demonstrates this correlation. Additionally, by plotting a reference line that represents perfect correlation (a coefficient of 1.0), along with our model's correlation line, we illustrate both the ideal scenario and our model's performance.

The observed linear correlation between our model's predictions and the actual labels is both numerically and visually evident. Specifically, Pearson's Correlation Coefficient is 0.1632, indicating a positive, albeit modest, correlation. Perhaps a non-linear correlation can better explain this. Nonetheless, it still suggests the model generally predicts in the correct direction and is performing better than random chance. Such a correlation serves as a performance metric that can potentially be enhanced with more refined preprocessing steps, thorough validation, and a more advanced model design.

4.3 *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 $\frac{\delta \mathcal{L}}{\delta \theta}$ of the loss function $\mathcal{L}$.

[133]:
```
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
    ridge_lambda = hyperparams["lambda"]
    batch_size = X.shape[0] # For Stochastic Gradient Descent it will be 1, for␣
↪mini-batch it will be the batch size
    assert batch_size == 1, "Should be Stochastic Gradient Descent thus batch␣
↪size should be 1"
    derivative = -2 / batch_size * X.T @ (y - X @ theta) + 2 * ridge_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.

[134]:
```
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 -= learning_rate * derivative

    return _theta
```

4.4 *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 $\mathcal{L}$ 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

[135]:
```
from sklearn.model_selection import train_test_split

def ridge_reg_loss(X, y, theta, hyperparams):
    """
    takes data, parameters and hyperparameters to calculate the 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:
        loss (float): the loss value
    """

    # Your Code Here
    ridge_lambda = hyperparams["lambda"]
    ridge_regularisation = ridge_lambda * np.linalg.norm(theta) ** 2
    loss = np.mean((X @ theta - y) ** 2) + ridge_regularisation

    return loss
```

[136]:
```
def train_entire_model(X_, train_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
    iterations = hyperparams["iterations"]

    np.random.seed(RANDOM_SEED) # For reproducibility

    # Train-test split
    train_X, test_X, train_y, test_y = train_test_split(X_, y_, test_size=0.2,␣
↪random_state=RANDOM_SEED)

    losses = []
    theta = np.random.normal(size=train_X.shape[1]) # Theta of size n_features␣
↪initialised with normal distribution (0, 1)

    for _ in range(iterations):
```

```
        # Randomly shuffle the data for each iteration for better convergence
        indices = np.arange(train_X.shape[0])
        np.random.shuffle(indices)
        X = train_X[indices]
        y = train_y[indices]

        for i in range(X.shape[0]): # Apply stochastic gradient descent, for␣
 ↪each sample (equivalent to batch size 1)
            theta = train_stoch_ridge_reg(X[i:i+1], y[i:i+1], theta,␣
 ↪hyperparams)

        loss = ridge_reg_loss(test_X, test_y, theta, hyperparams)
        losses.append(loss)

    return losses
```

[137]:

```
X_, y_ = X_, y_ # Reusing data structures from before

hyperparameters = {
    # Your Code Here
    "lambda": 0.1,
    "learning_rate": 1e-6,
    "iterations": 2000
}

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

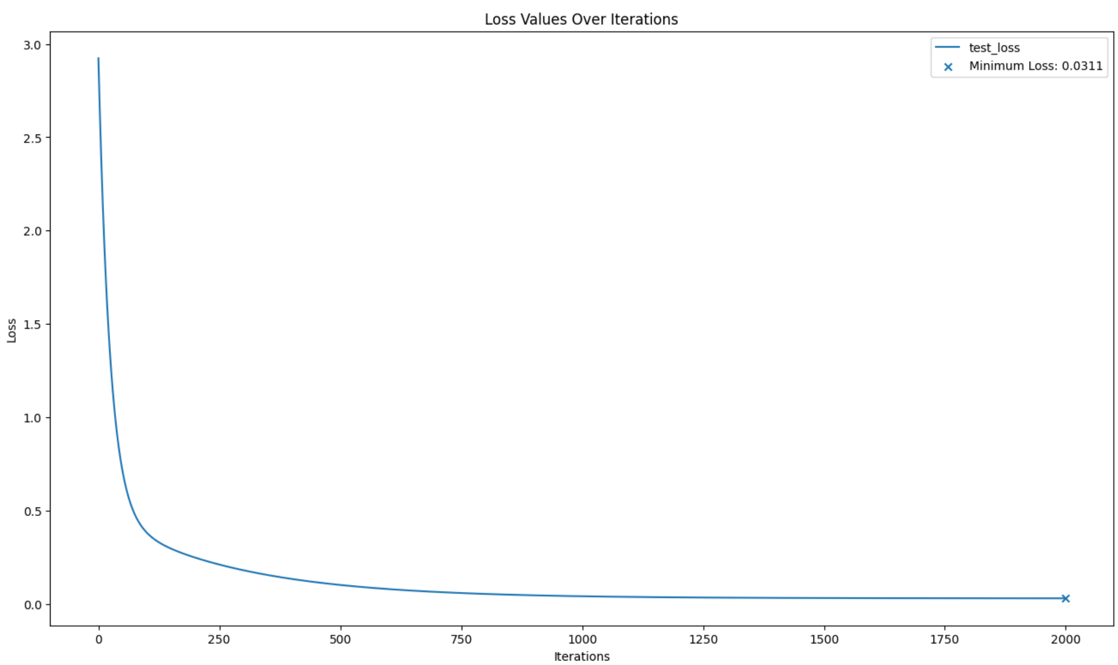
- Implement the `visualise_loss_values` function to use the appropriate visualisations to plot the loss values in a meaningful way.
- The function does not have to return anything. Display the visualisation as a step within the implemented function.

[138]:

```
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

    # Zoom in parameters
    upper_among_losses = 0
```

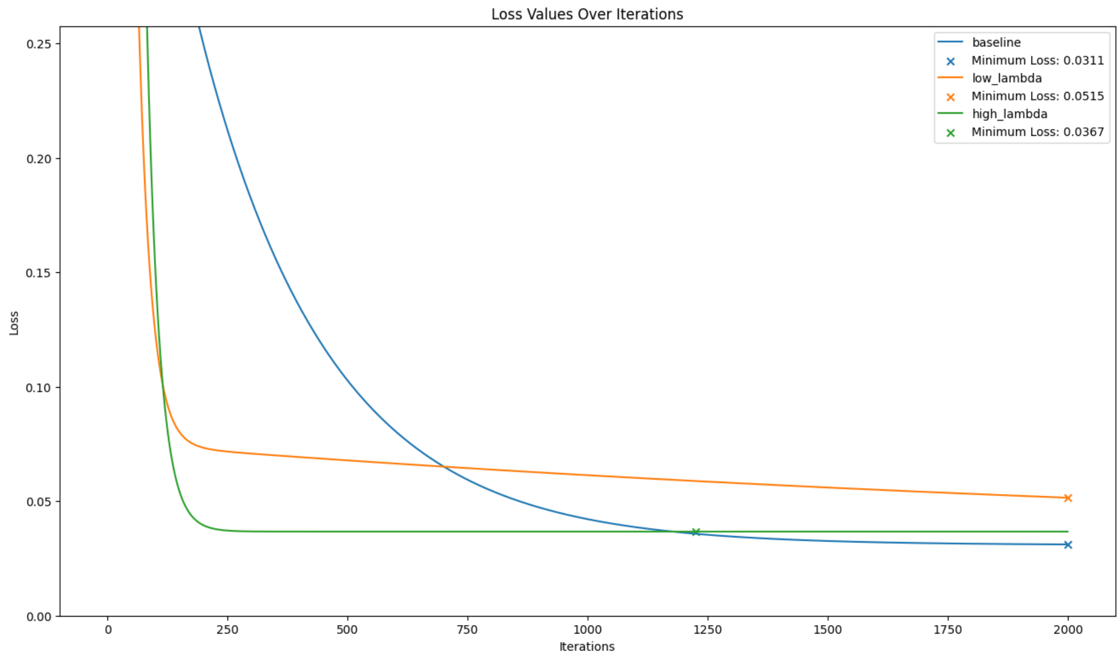
```
    plt.figure(figsize=(16, 9))
    for loss_type, values in loss_values.items():
        plt.plot(values, label=loss_type)
        plt.scatter(np.argmin(values), np.min(values), marker="x",␣
 ↪label=f"Minimum Loss: {np.min(values):.4f}")
        upper_among_losses = max(upper_among_losses, np.min(values))

    # Zoom in if necessary
    if len(loss_values) > 1:
        plt.ylim(0, upper_among_losses * 5)
    plt.xlabel("Iterations")
    plt.ylabel("Loss")
    plt.legend()
    plt.title("Loss Values Over Iterations")
    plt.show()
```

[139]:
```
visualise_loss_values({"test_loss": losses})
```



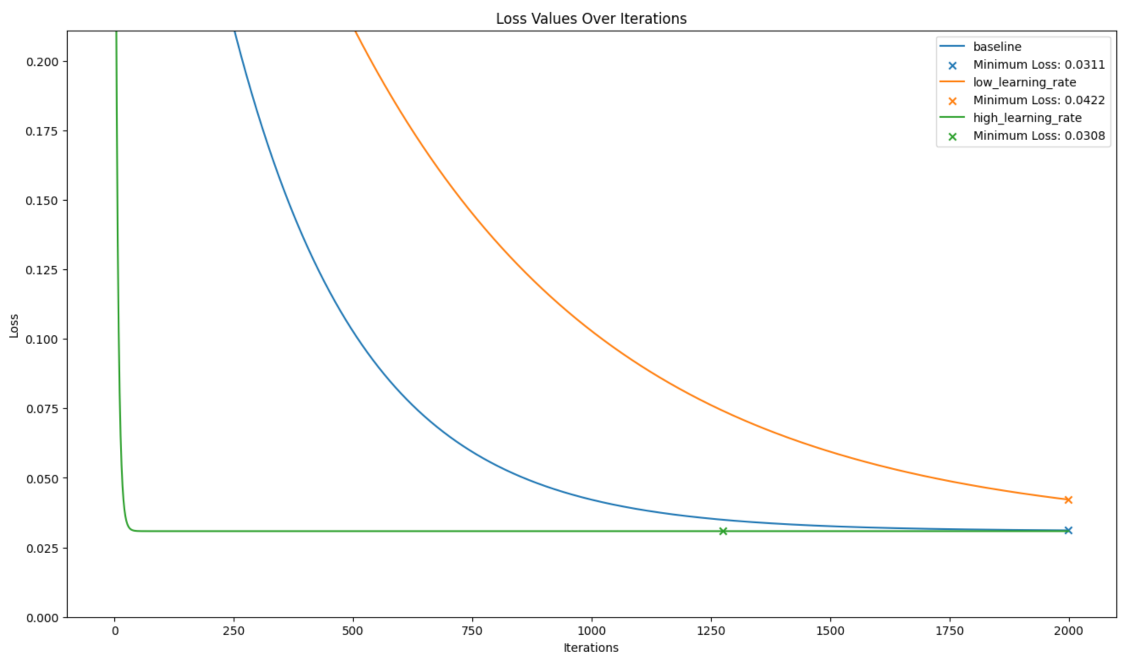
[140]:
```
# Testing with different lambda values
list_of_hyperparameters = {
    "baseline":{
        "lambda": 0.1,
        "learning_rate": 1e-6,
        "iterations": 2000
    },
```

```
    "low_lambda":{
        "lambda": 0.01,
        "learning_rate": 1e-6,
        "iterations": 2000
    },
    "high_lambda":{
        "lambda": 1,
        "learning_rate": 1e-6,
        "iterations": 2000
    },
}

lambda_losses = {loss_type: train_entire_model(X_, y_, hyperparams) for␣
 ↪loss_type, hyperparams in list_of_hyperparameters.items()}
visualise_loss_values(lambda_losses)
```



[ ]:
```
# Testing with different learning rate
list_of_hyperparameters = {
    "baseline":{
        "lambda": 0.1,
        "learning_rate": 1e-6,
        "iterations": 2000
    },
    "low_learning_rate":{
        "lambda": 0.1,
```

```
        "learning_rate": 5e-7,
        "iterations": 2000
    },
    "high_learning_rate":{
        "lambda": 0.1,
        "learning_rate": 5e-5,
        "iterations": 2000
    },
}

learning_rate_losses = {loss_type: train_entire_model(X_, y_, hyperparams) for␣
 ↪loss_type, hyperparams in list_of_hyperparameters.items()}
visualise_loss_values(learning_rate_losses)
```



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 values chosen for lambda and learning rate are 0.1 and 1e-6 respectively. Upon visualising the loss over iterations, we observe that the test loss with the selected hyperparameters follows a downward trajectory, reaching a minimum of approximately 0.0311. Test set is used to instead of training set for a less overfitted loss calculation. This gradual decline in loss indicates that the model is learning from the data with each iteration. It keeps decreasing at the final iterations too, suggesting that the model's capability may not be fully exploited using the proposed set of hyperparameters.

When adjusting hyperparameters individually, they exhibit distinct influences on the loss curve. Altering the penalisation constant, lambda, changes the path of loss curve. High lambda (1) and low lambda (0.01) both show quicker decrease in earlier iterations but also quicker stabilisation, resulting in a lower minimum loss than the baseline. We hypothesise this is due to over-penalisation and under-penalisation that would impede elevating the model to its full potential. Therefore, choosing an appropriate penalisation that allows enough expression but with low bias is essential.

In contrast, changes in the learning rate affect the rate of convergence while maintaining the path of curve. A higher learning rate (5e-5) results in a more rapidly decreasing loss curve, stabilising quicker, whereas a lower learning rate (5e-7) does not converge within the 2000 iterations. This suggests that a higher learning rate propels the model's weights more efficiently towards the optimal solution due to larger step sizes, while too low a rate causes insufficient updates to the model weights before reaching the maximum iteration count. In this case, higher learning rate reached to minimum loss before reaching the maximum iterations. Notice that the baseline within the proposed learning rate range actually achieves a very close performance to the high learning rate setting.

4.5 - End of Coursework -