Zach Siegel (#805435913), Edward Nguyen (#904663048), Boya (#005037574), Banseok Lee (#605351891)
EE219 - *Large-Scale Data Mining: Models and Algorithms*
Prof. Vwani Roychowdhury
Winter 2021 - Project 4

Question 1.

Our report includes three separate data-exploration files, `Bike_Sharing_Report.html`, `Suicide Rates Overview Report.html`, and `Video Transcoding Time Report.html`. The data-exploration files each includes a heatmap of the correlation matrix of features as well as histograms of each of the numerical features in the three datasets.

Note that some of the categorical features were already encoded so they were included in the correlation matrix but we ignore them in the anaylsis.

Bike Dataset

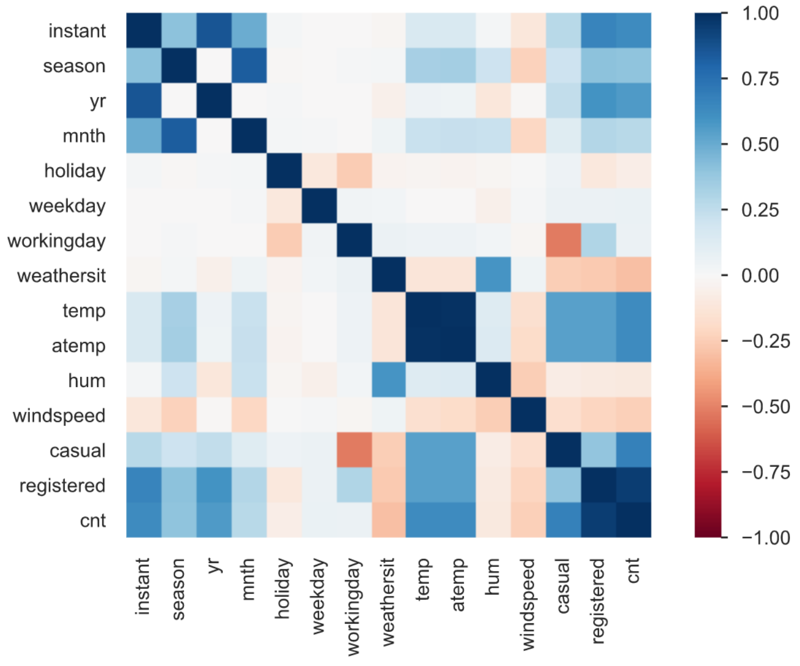


Figure 1: Heatmap of Pearson correlation matrix of dataset columns for the bikes dataset.

For the `casual` target variable (the count of casual users) , we notice there is a strong positive correlation between features `temp` and `atemp` with the `casual` target variable. The positive correlation with the temperature makes sense as warmer temperatures would cause cause bike users to rent a bike. For the `registered` target variable (the count of registered users), we notice that there is a strong positive correlation between features `temp` and `atemp` with the target variable `registered`. The positive correlation with the temperature makes sense as warmer temperatures would cause cause bike users to rent a bike.

For the `count` target variable, the highest absolute correlation comes from the `temp` and `atemp` features where there is a positive correlation. The positive correlation with the temperature makes sense as warmer temperatures would cause cause bike users to rent a bike.

Video Dataset

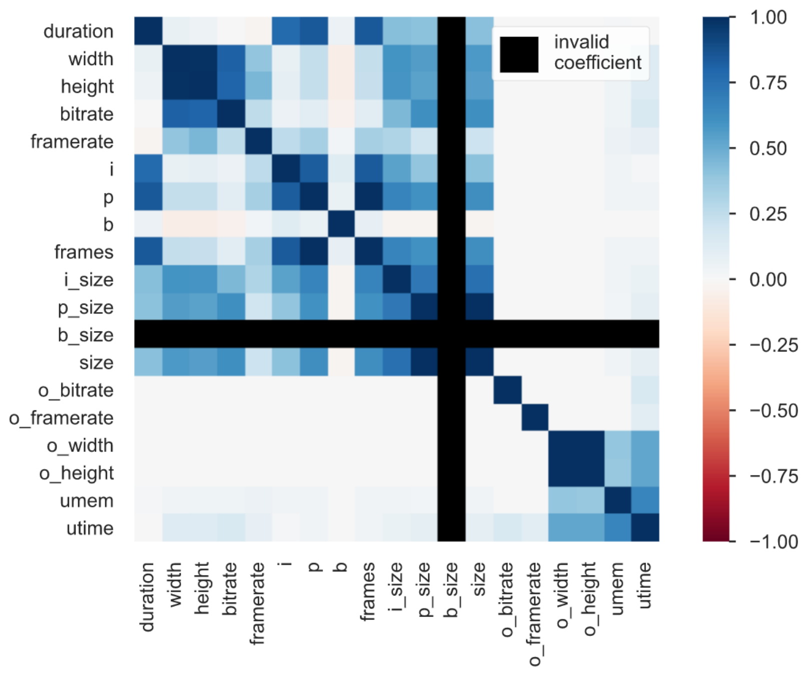


Figure 2: Heatmap of Pearson correlation matrix of dataset columns for the video dataset.

For the `umem` target varaible, we notice there is a strong positive correlation between features `o_width` and `o_height`. This implies the total codec allocated memory for transcoding scales proportionally with a higher output width and output height. For the `utime` target varaible, we notice there is a strong positive correlation between features `o_width` and `o_height`. This implies the total transcoding time for for transcoding scales proportionally with a higher output width and output height. Both of this makes sense logically as a larger output width and height would take longer to transcode and more memory to transcode.

Suicide Dataset

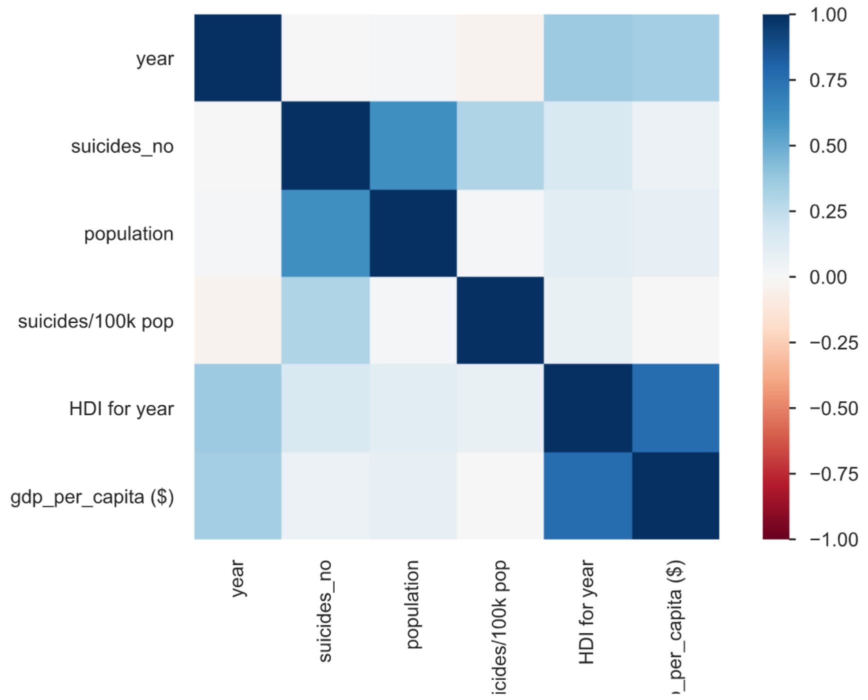
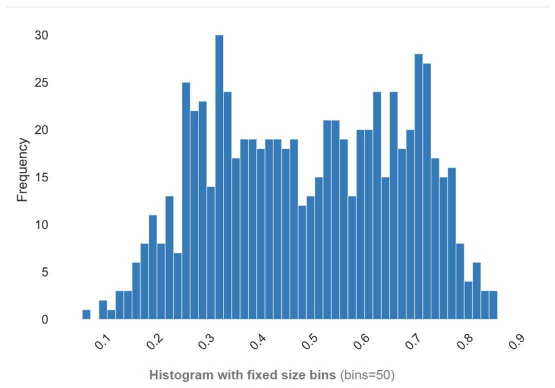


Figure 3: Heatmap of Pearson correlation matrix of dataset columns for the suicide dataset.

For the `suicides_no` target variable, the highest absolute correlation comes from the `population` feature. This makes sense as if there is more people, it is more likely for there to be suicides. For the `suicide/100k pop` target variable, there are no features that have a strong correlation. The variable with the absolute strongest correlation appears to be year with a slight negative correlation. This seems to mean that suicides per 100k in population decreased with time. Overall, we need to look at the other categorical variables.

Question 2.



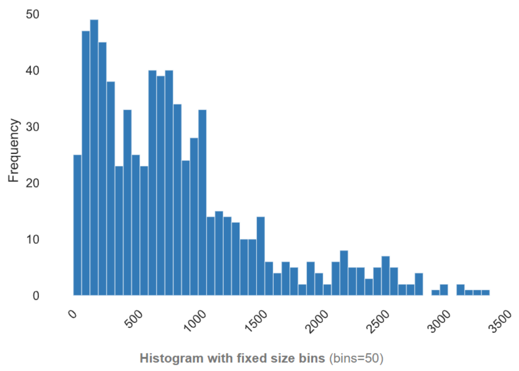
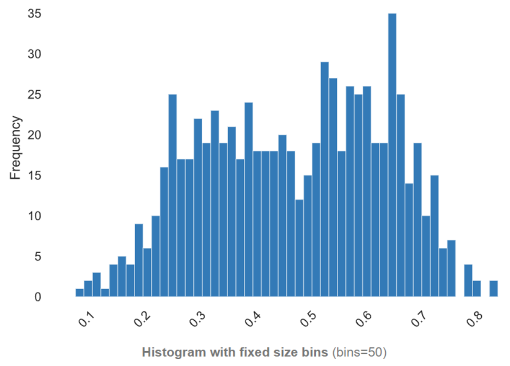


Figure 4: Histogram of temp(left) and casual(right) features of dataset columns for the bike sharing dataset.



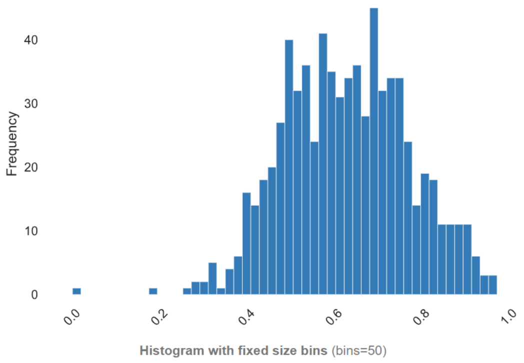
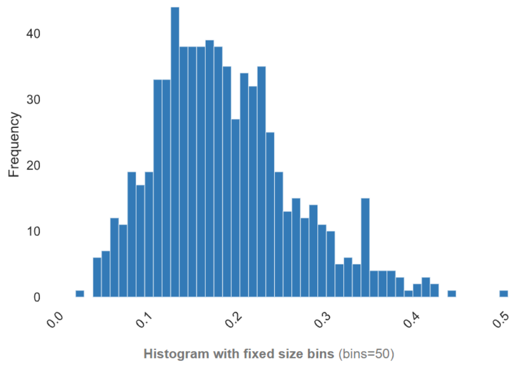


Figure 5: Histogram of atemp(left) and hum(right) features of dataset columns for the bike sharing dataset.



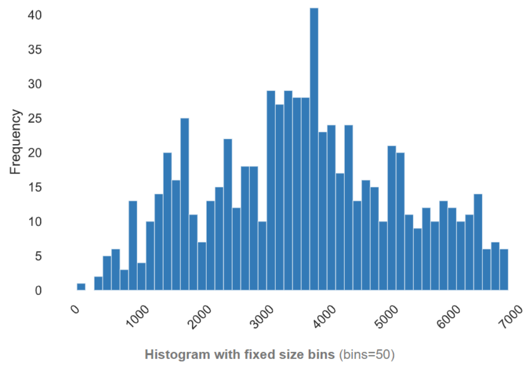


Figure 6: Histogram of windspeed(left) and registered(right) features of dataset columns for the bike sharing dataset.

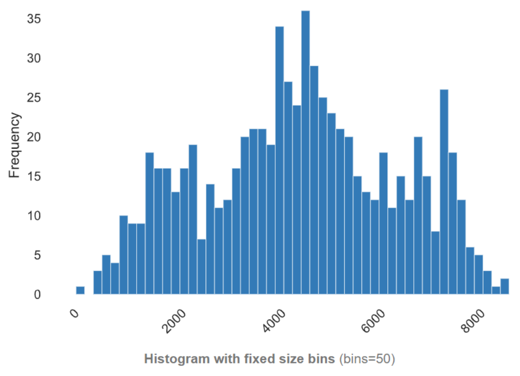
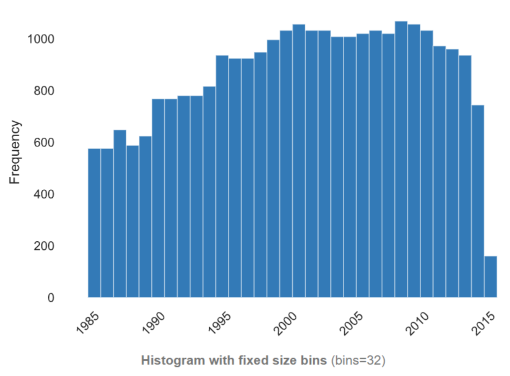


Figure 7: Histogram of cnt(left) feature of dataset columns for the bike sharing dataset.



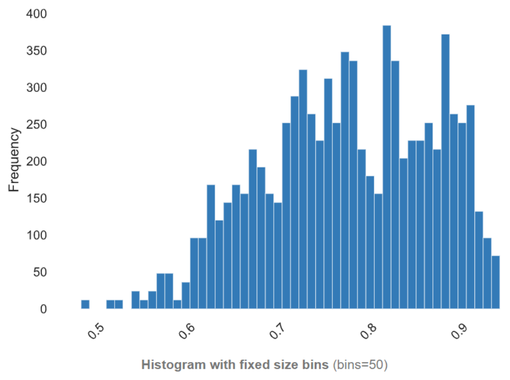
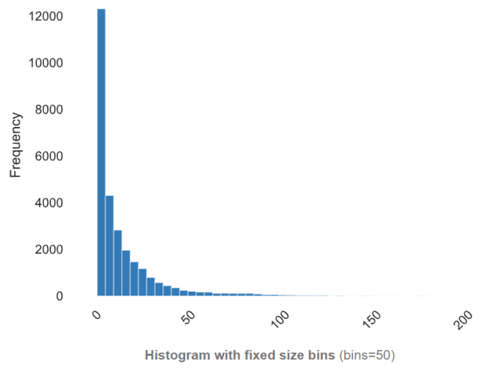


Figure 8: Histogram of year(left) and HDI for year(right) features of dataset columns for the suicide rates dataset.



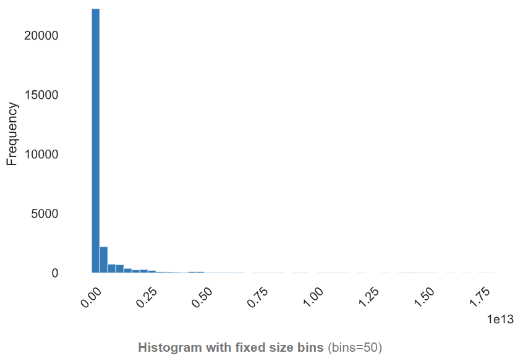
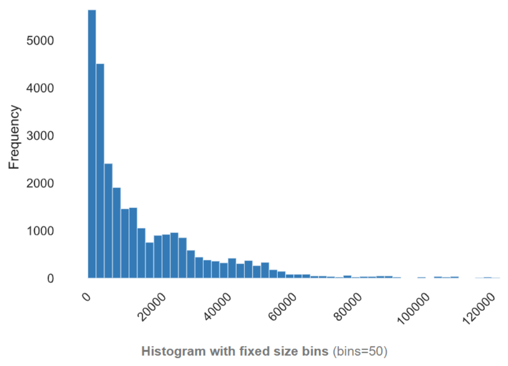


Figure 9: Histogram of suicides/100k pop(left) and gdp for year(right) features of dataset columns for the suicide rates dataset.



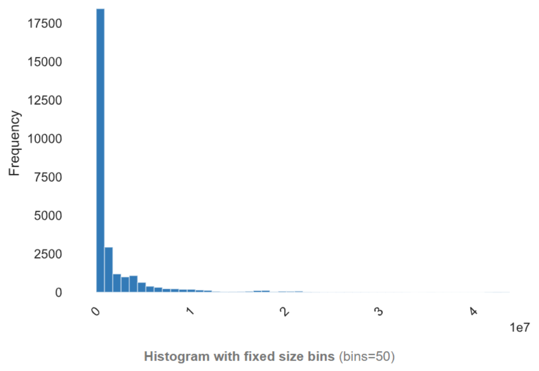


Figure 10: Histogram of gdp per capita(left) and population(right) features of dataset columns for the suicide rates dataset.

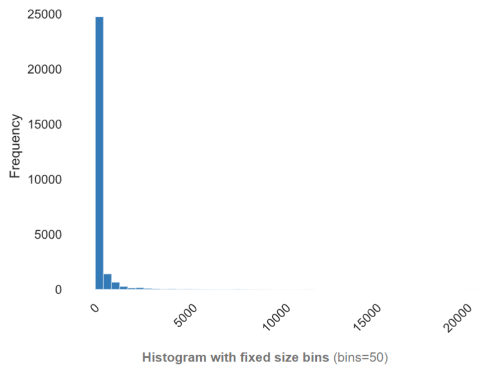
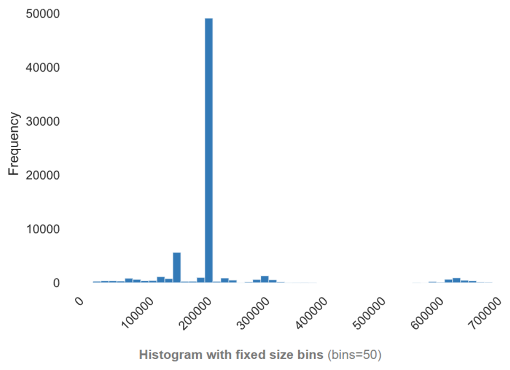


Figure 11: Histogram of suicides no(left) feature of dataset columns for the suicide rates dataset.



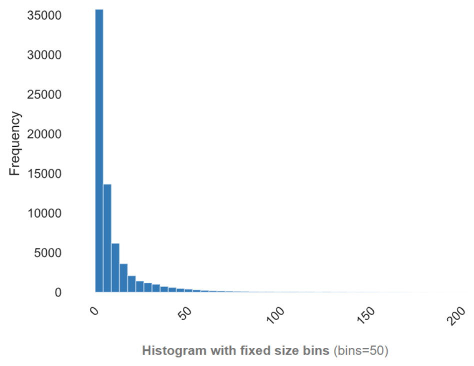
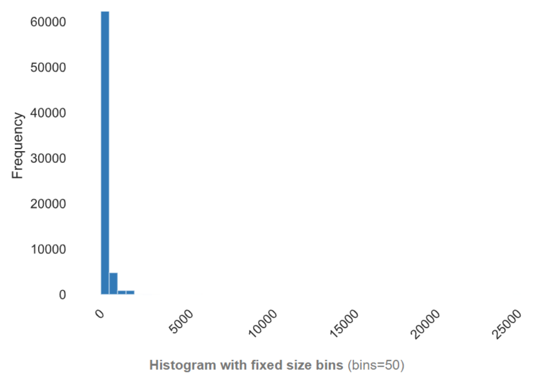


Figure 12: Histogram of umem(left) and utime(right) features of dataset columns for the video transcoding time dataset.



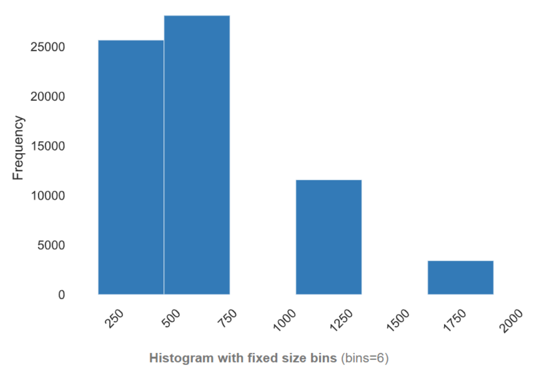
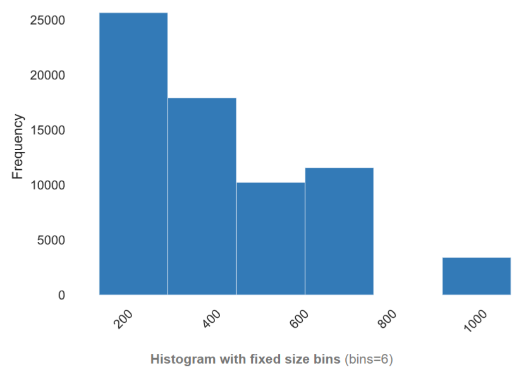


Figure 13: Histogram of duration(left) and width(right) features of dataset columns for the video transcoding time dataset.



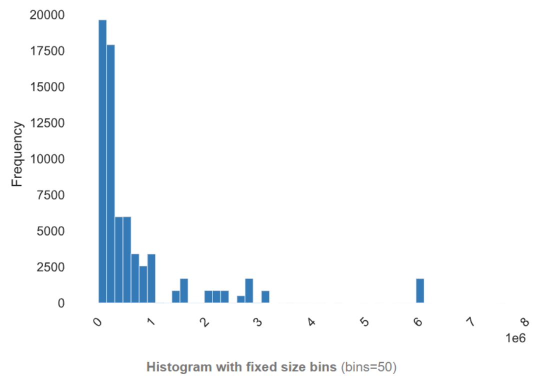
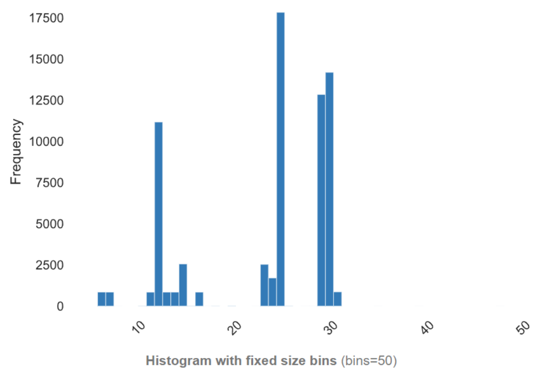


Figure 14: Histogram of height(left) and birthrate(right) features of dataset columns for the video transcoding time dataset.



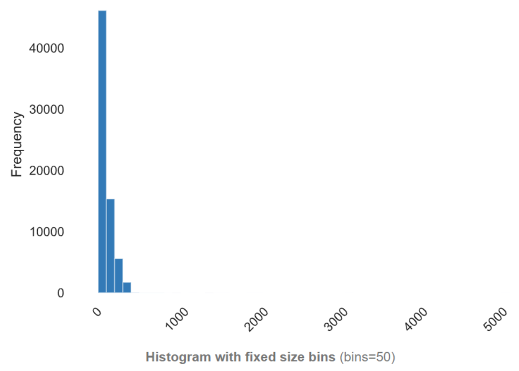
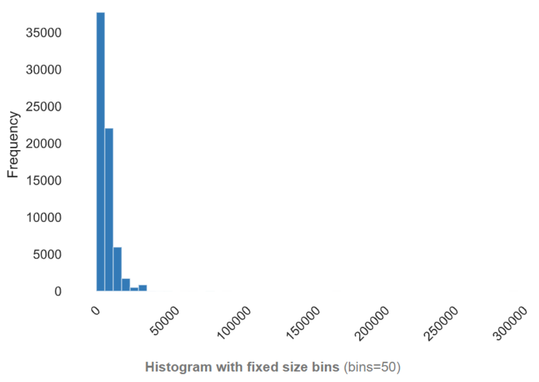


Figure 15: Histogram of framerate(left) and i(right) features of dataset columns for the video transcoding time dataset.



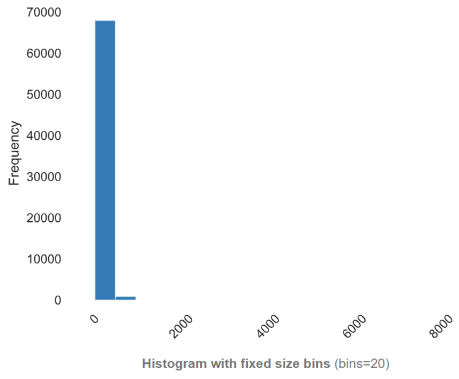
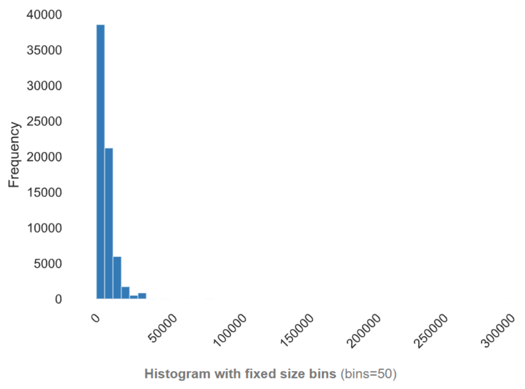


Figure 16: Histogram of p(left) and b(right) features of dataset columns for the video transcoding time dataset.



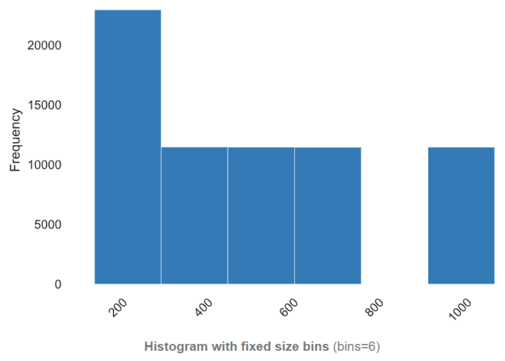
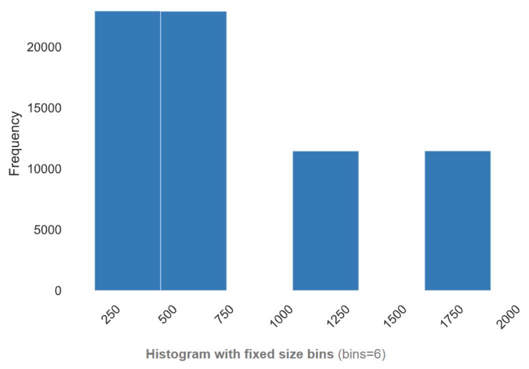


Figure 17: Histogram of frame(left) and o-height(right) features of dataset columns for the video transcoding time dataset.



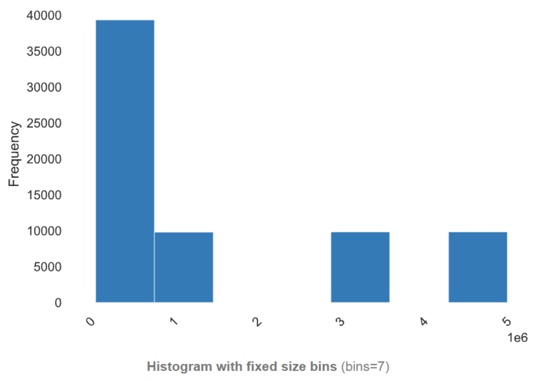
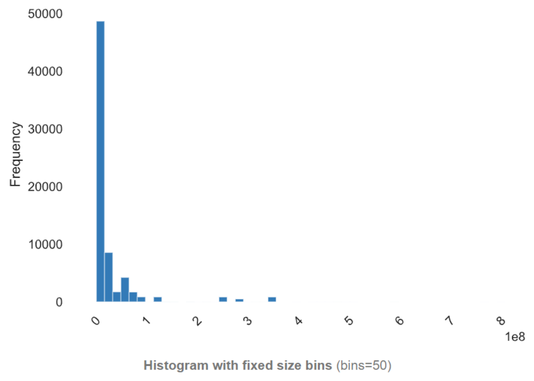


Figure 18: Histogram of o-width(left) and o-bitrate(right) features of dataset columns for the video transcoding time dataset.



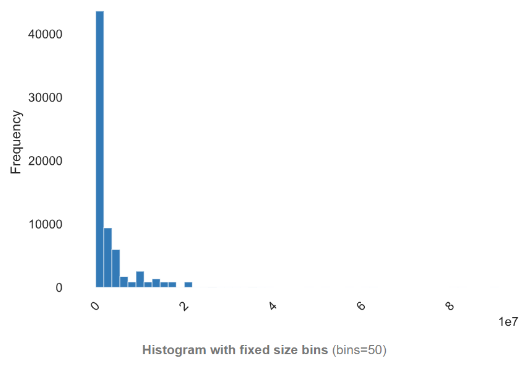


Figure 19: Histogram of size(left) and i-size(right) features of dataset columns for the video transcoding time dataset.

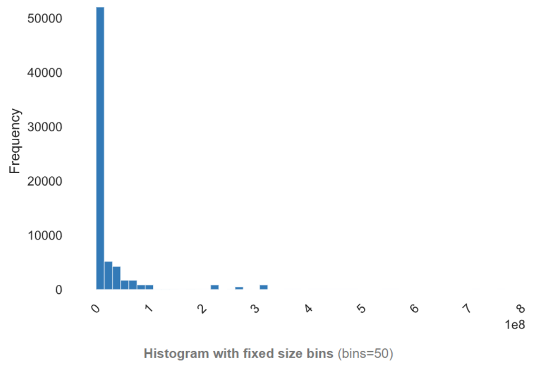


Figure 20: Histogram of p-size(left) feature of dataset columns for the video transcoding time dataset.

Our report includes three separate data-exploration files, `Bike_Sharing_Report.html`, `Suicide Rates Overview Report.html`, and `Video Transcoding Time Report.html`. The data-exploration files each includes a heatmap of the correlation matrix of features as well as histograms of each of the numerical features in the three datasets.

Some of the learning methods in this report are designed with normally distributed data in mind, and perform best when features are all roughly normally distributed, with zero mean, and/or with standard deviation comparable to that of other features. In Question 8., we demonstrate and discuss the effect of data scaling.

Unfortunately, centering and scaling a feature does not eliminate its skewness. The blog post [2] discusses three methods for handling skew:

- logarithm transform,
- square root transform,
- "Box-Cox Transform".

All three of these transformations require strictly positive data and reduce rightward skew by reducing large values.

These transformations may be justified for results-critical applications, but we did not elect to include any of these in our report for the following reasons. First, these techniques reduce interpretability of a learned model; in the case of linear regression, for example, the model itself enables not just *prediction* but also *inference* in that the coefficients estimate an aggregate relationship between features and the target. Second, some of the more powerful nonlinear techniques (such as decision trees and neural networks) have known "universal approximation" properties, which mean that learning may be aided by pre-processing that reduces skew, but it should also succeed without that pre-processing; pre-processing would aid some algorithms and not others, preventing an "apples-to-apples" comparison.

Question 3.

Bike Dataset

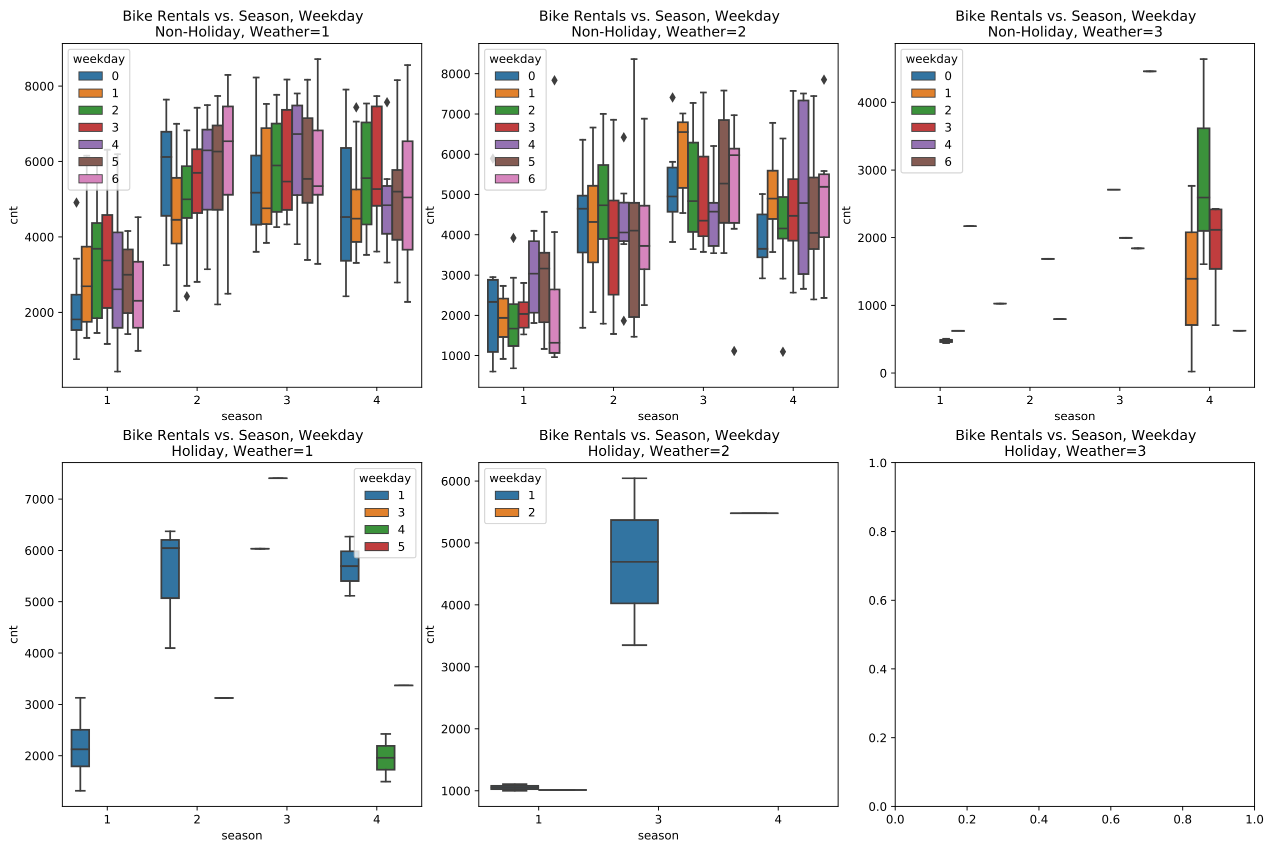


Figure 21: A boxplot of number of daily bike rentals separated within each plot by season and weekday, separated between plots by holiday and weather.

Figure 21 illustrates several aspects of the bike rental dataset.

- Data is much more available during non-holiday dates (because most dates are not holidays).
- Data is much more available when weather is "1" or "2". There are zero entries for `weathersit` value of 4. The rental rates seem slightly higher when `weathersit` is equal to 1, but not by much (by inspection). The data description on the UCI website notes that the `weathersit` variable uses the following code:

1. Clear, few flouds, partly cloudy
2. Mist + cloudy, cist + broken clouds, mist + few clouds, mist
3. Light snow, light rain + thunderstorm + scattered clouds, light rain + scattered clouds
4. Heavy rain + ice pallets + thunderstorm + mist, snow + fog.

- Season "3" consistently has the most bike rentals. This corresponds to summer. During the summer, Thursday through Saturday seem to have among the highest bike rentals, probably due to recreational use; during the winter and spring, midweek days seem to have the highest rate of bike rental, perhaps due to consistent commute customers.

Video Dataset

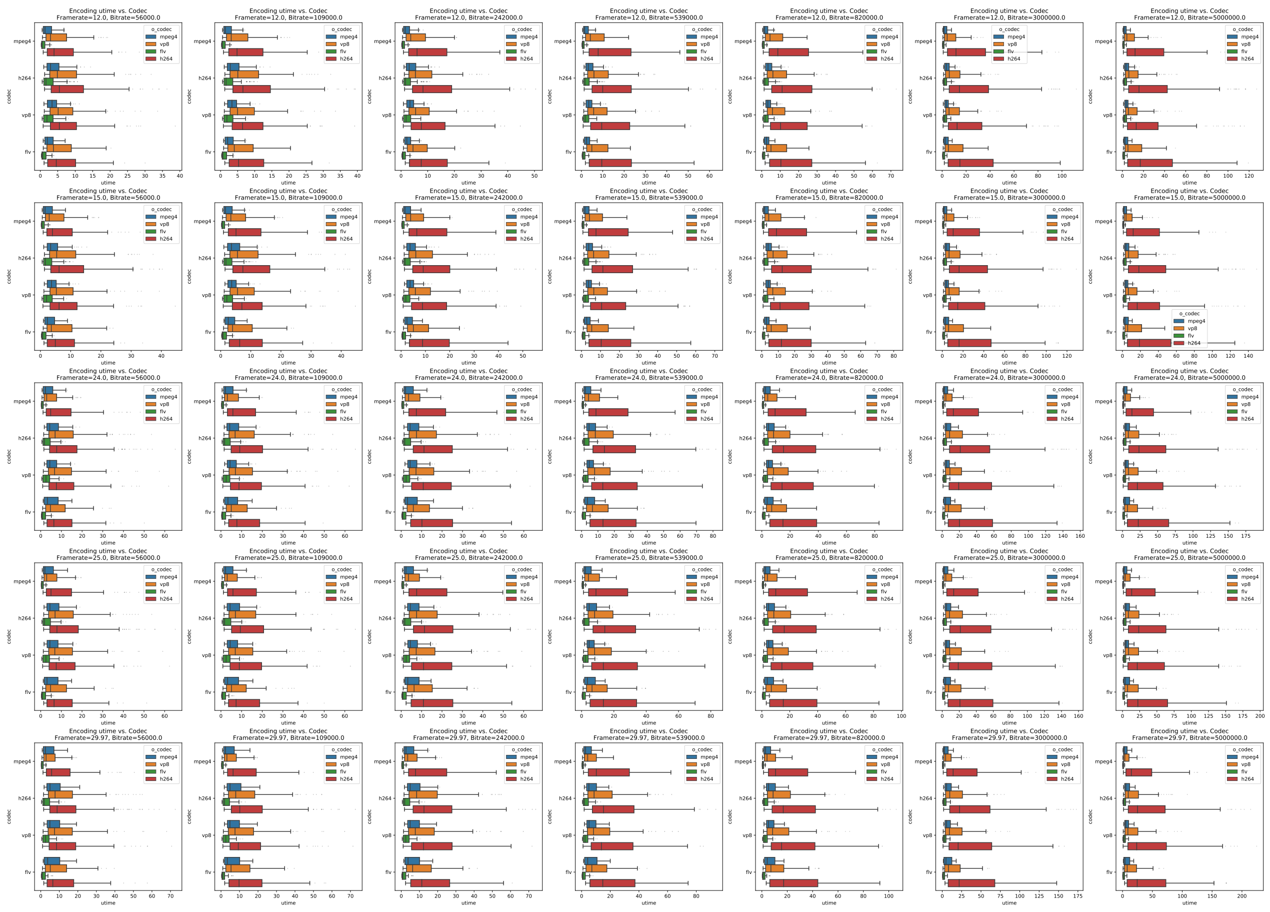


Figure 22: Video transcoding times versus output bitrate and framerate, separated by input and output codec.

Figure 22 illustrates several aspects of the video transcoding dataset:

- Output codec (`o_codec`): h264 encoding consistently takes the most time, followed by vp8, then mpeg4, then flv.

- Input codec (`codec`): flv generally takes the most time, followed by vp8, h264, then mpeg4. This is partially an inversion of the output codec times. Perhaps flv is different from the others in some fundamental way such that it is easy to transcode *to* but difficult to transcode *from*; the opposite seems to be true of h264 and mpeg4.
- As the bitrate increases (moving from left to right across the plots), the transcoding time generally increases for all formats.
- As the framerate increases (top to bottom across the plots), transcoding time generally increases slightly, though the range increases more than the mean.
- A plot of encoding *memory usage* (`umem`) shows similar trends (not pictured).

Suicide Dataset

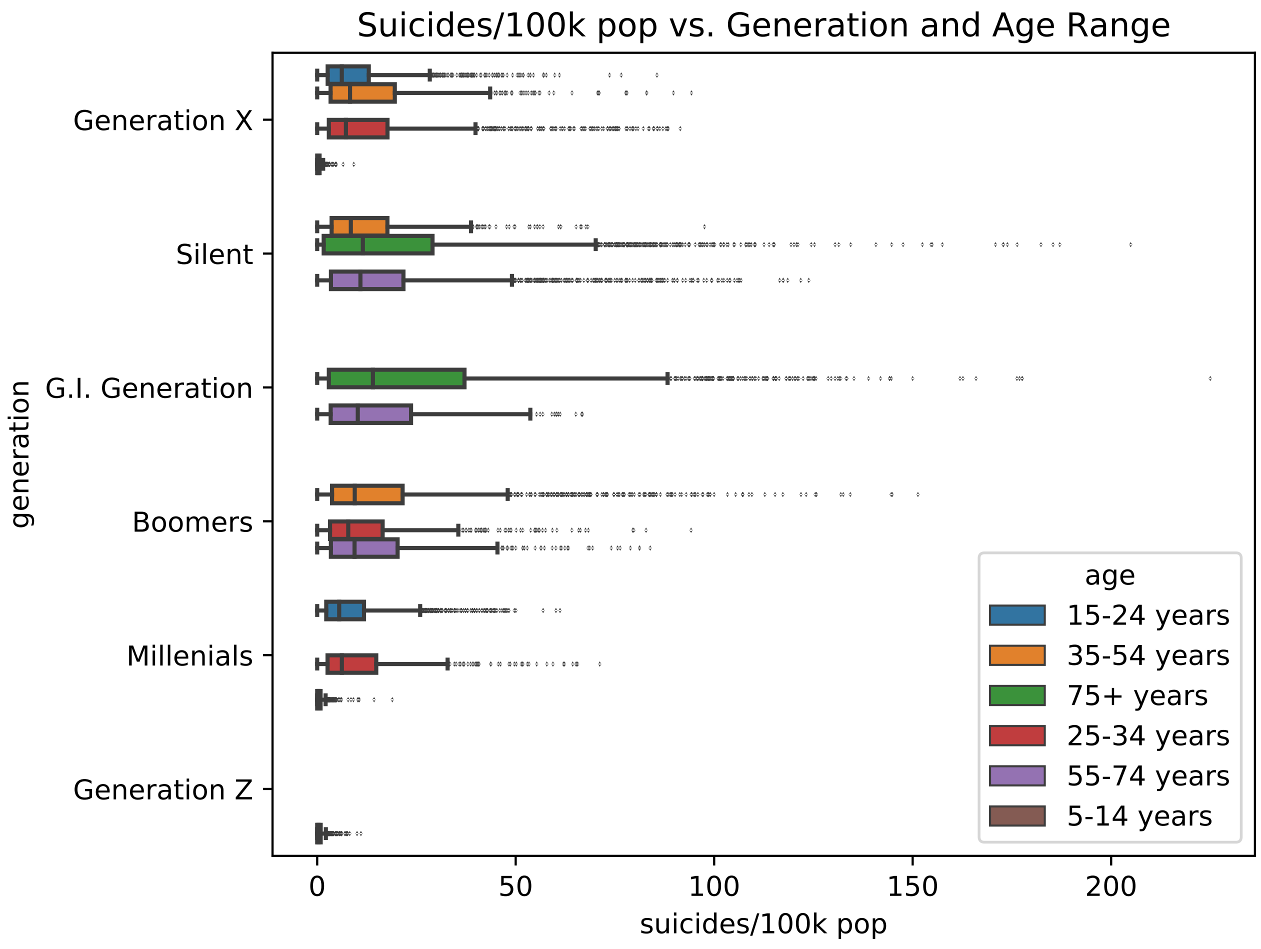


Figure 23: Suicides separated by generation and age range. Note that the true order of the generations in "G.I. Generation", "Silent", "Boomers", "Generation X", "Millenials", "Generation Z".

Figure 23 illustrates several aspects of the suicide dataset:

- For obvious reasons, data on certain generations in certain age groups are not as available as others. There are no Millenial or Generation Z individuals who are older then 25-34 years (any such observations are essentially noise).
- From the WHO website, it is hard to know how complete the data is over the entire timespan of the dataset. For example, the G.I. generation (the chronologically first generation included) suicides peak in old age; it seems possible that for that generation, earlier data was unavailable (though it is also possible that their participation in World War II affected propensity to suicide or suicide data availability in other ways).
- It seems that in all generations (in particular "Silent", "Boomers", "Generation X"), suicides increase as age increases.

Question 4.

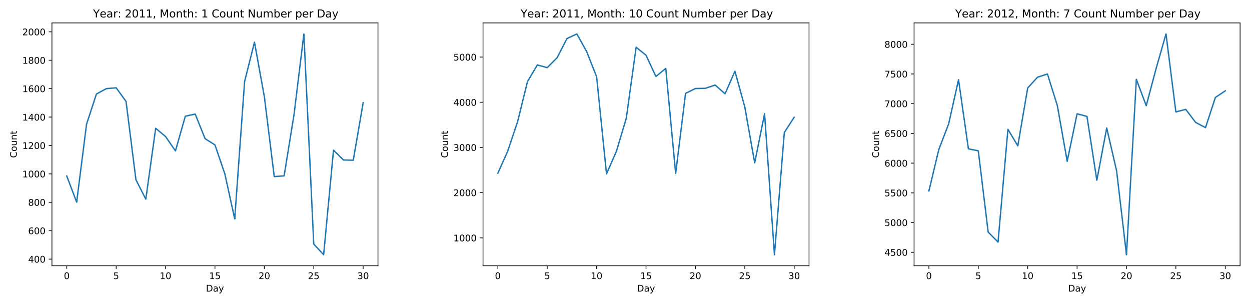


Figure 24: Rentals per day during three separate months in 2011 and 2012.

Figure 24 illustrates that there is often a rough weekly cycle in bike rentals (3-5 peaks and troughs in a month). Figure 21 (box plot of rentals separated by day, season, weather, and holiday) confirms that there is great variation in overall rentership on different weekdays. Figure 21 makes it clear that the most popular weekday changes with the seasons: during summer, weekends are most popular, whereas during winter, weekdays are, probably corresponding to the preponderance of commuters versus recreational renters. The isolated months in Figure 24 demonstrates these cycles in January, October, and July, respectively. Weekdays are not demarcated on these plots, but the presence of clear cyclicality confirms that the aggregate trend mentioned above can often be observed within a given month. Generally, we note there is large amount of fluctuation in the count number throughout the month.

Question 5.

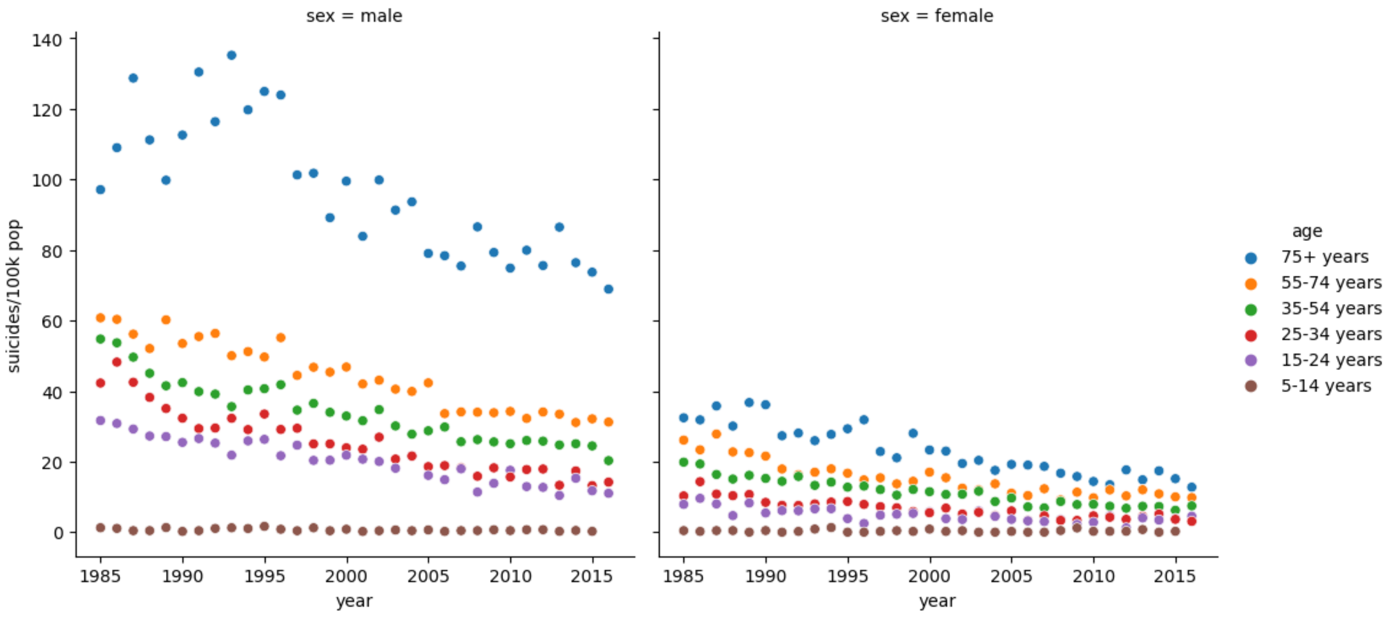


Figure 25: Suicides over time for Austria.

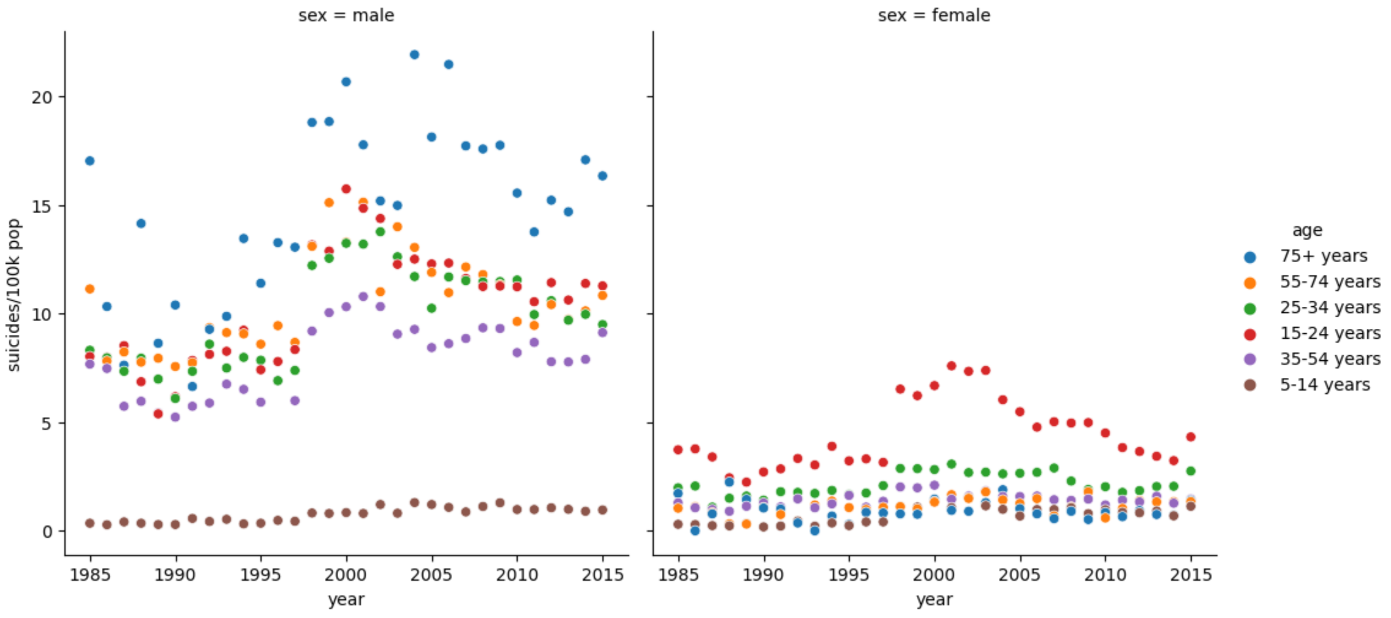


Figure 26: Suicides over time for Columbia.

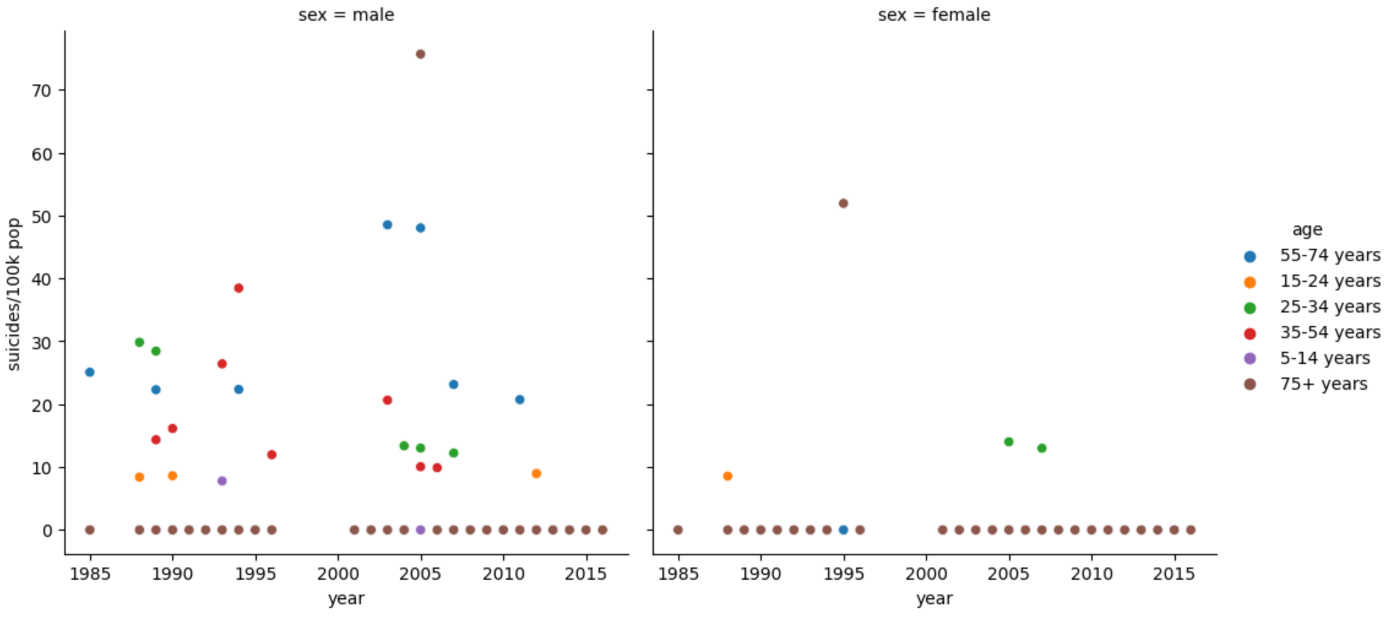


Figure 27: Suicides over time for Grenada.

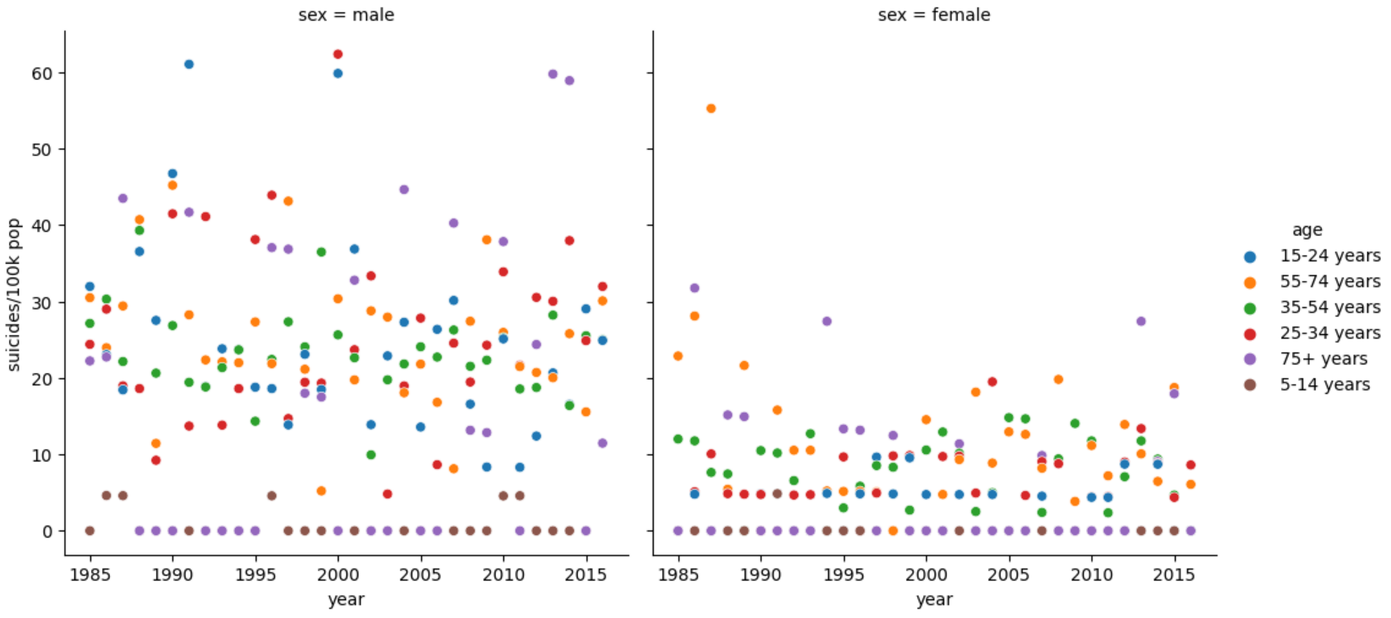


Figure 28: Suicides over time for Iceland.

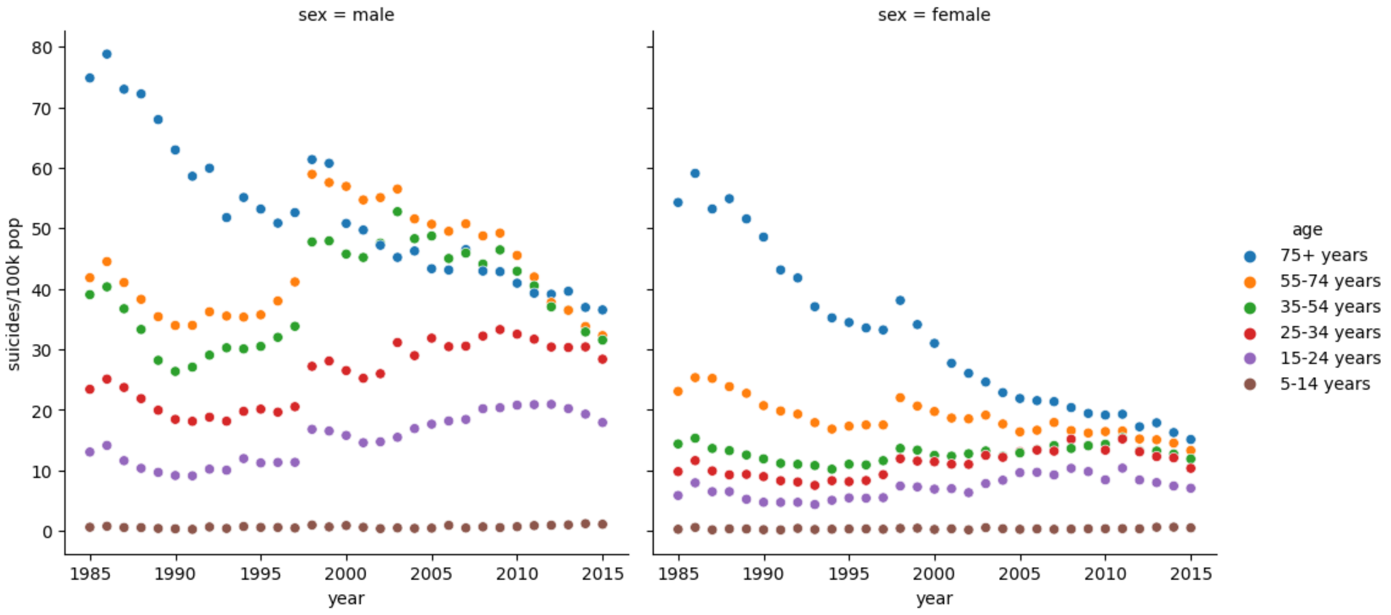


Figure 29: Suicides over time for Japan.

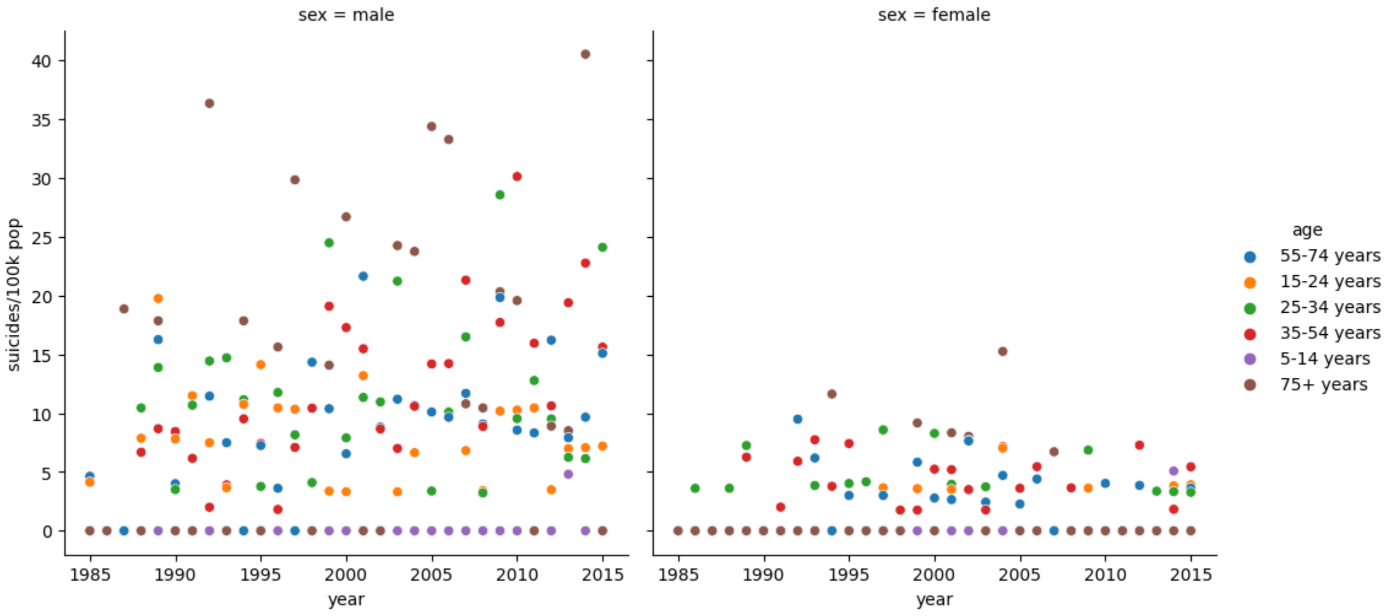


Figure 30: Suicides over time for Malta.

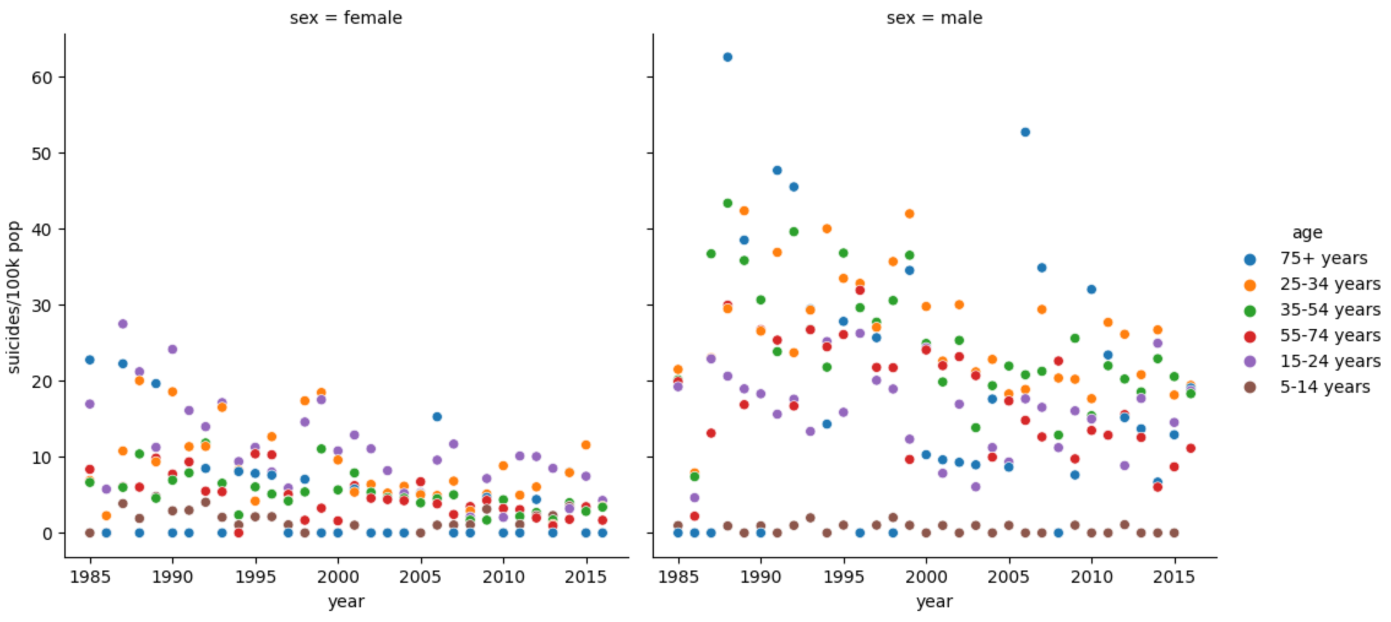


Figure 31: Suicides over time for Mauritius.

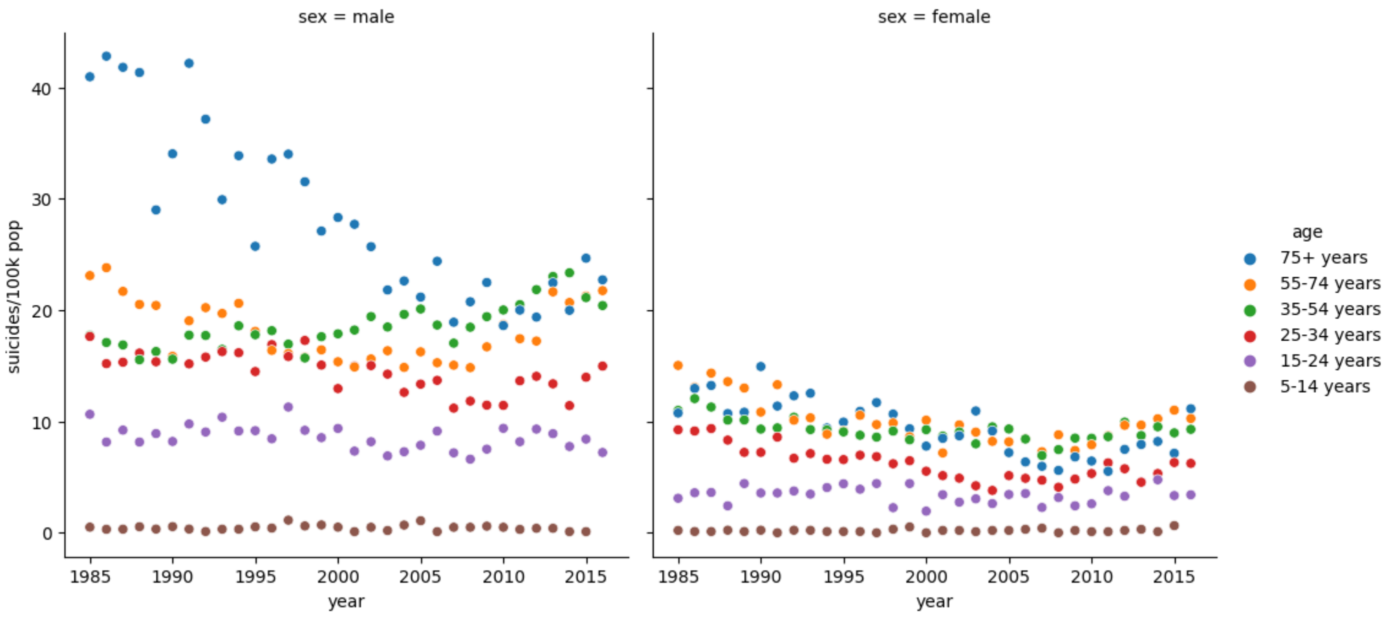


Figure 32: Suicides over time for Netherlands.

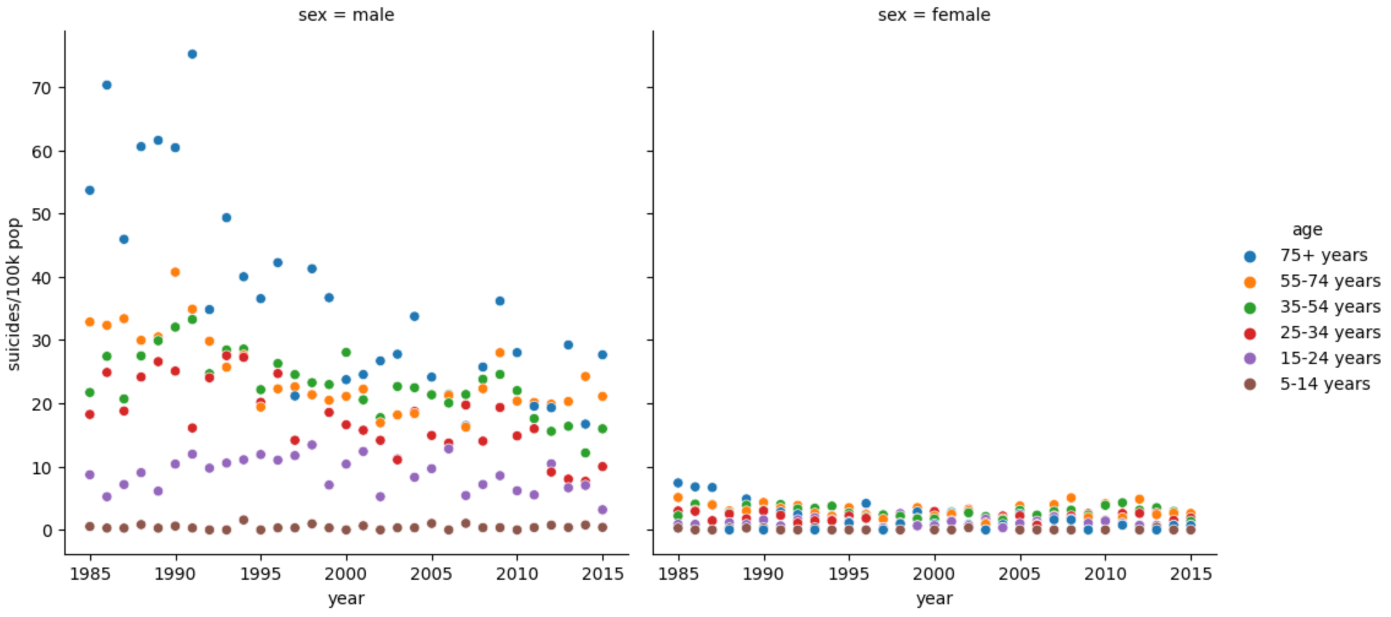


Figure 33: Suicides over time for Puerto Rico.

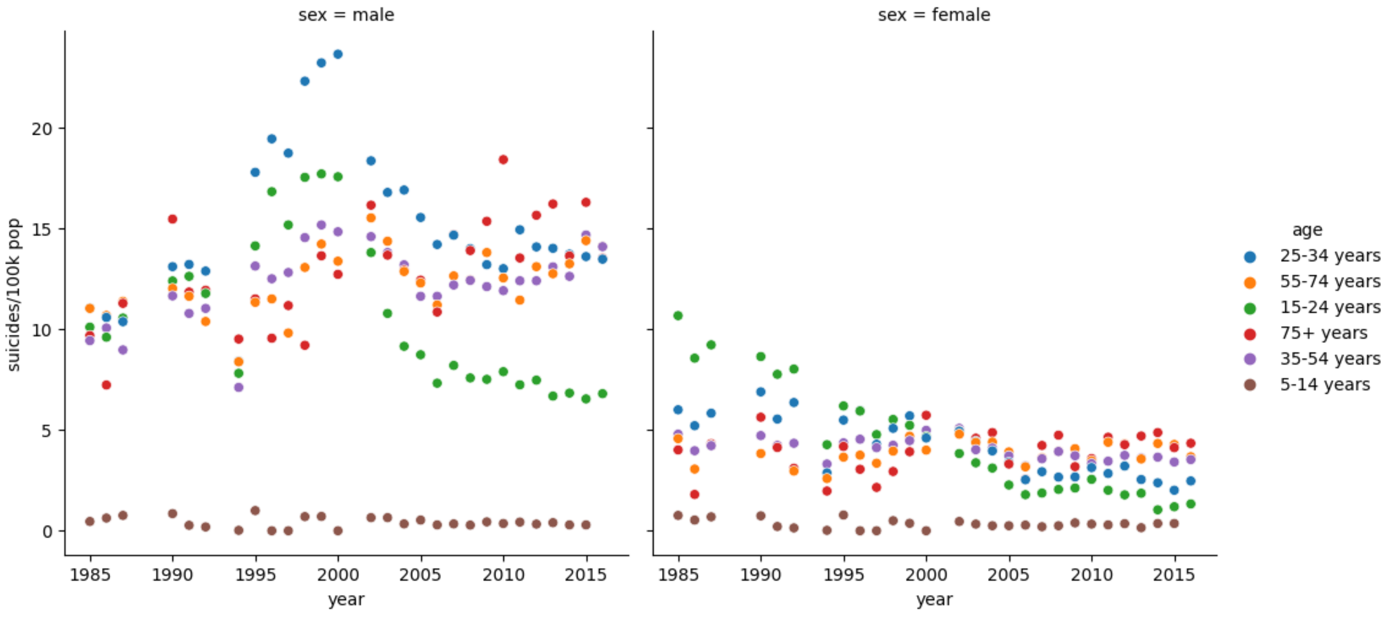


Figure 34: Suicides over time for Thailand.

Figure 35: Suicides over time for the ten countries with the longest timespans of available data.

Figure 35 demonstrates that in many countries (those for which the most data is available), there seems to be a net downward trend in number of suicides per year, at all age groups. In particular, Japan and Austria have strong downward trends after relatively high rates in the 1980s. Several countries seem to have experienced a slight uptick in the late 1990s and early 2000s. Overall, women have lower rates of suicides than men, and older individuals seem to have the highest rates of suicide in almost all countries.

Question 6.

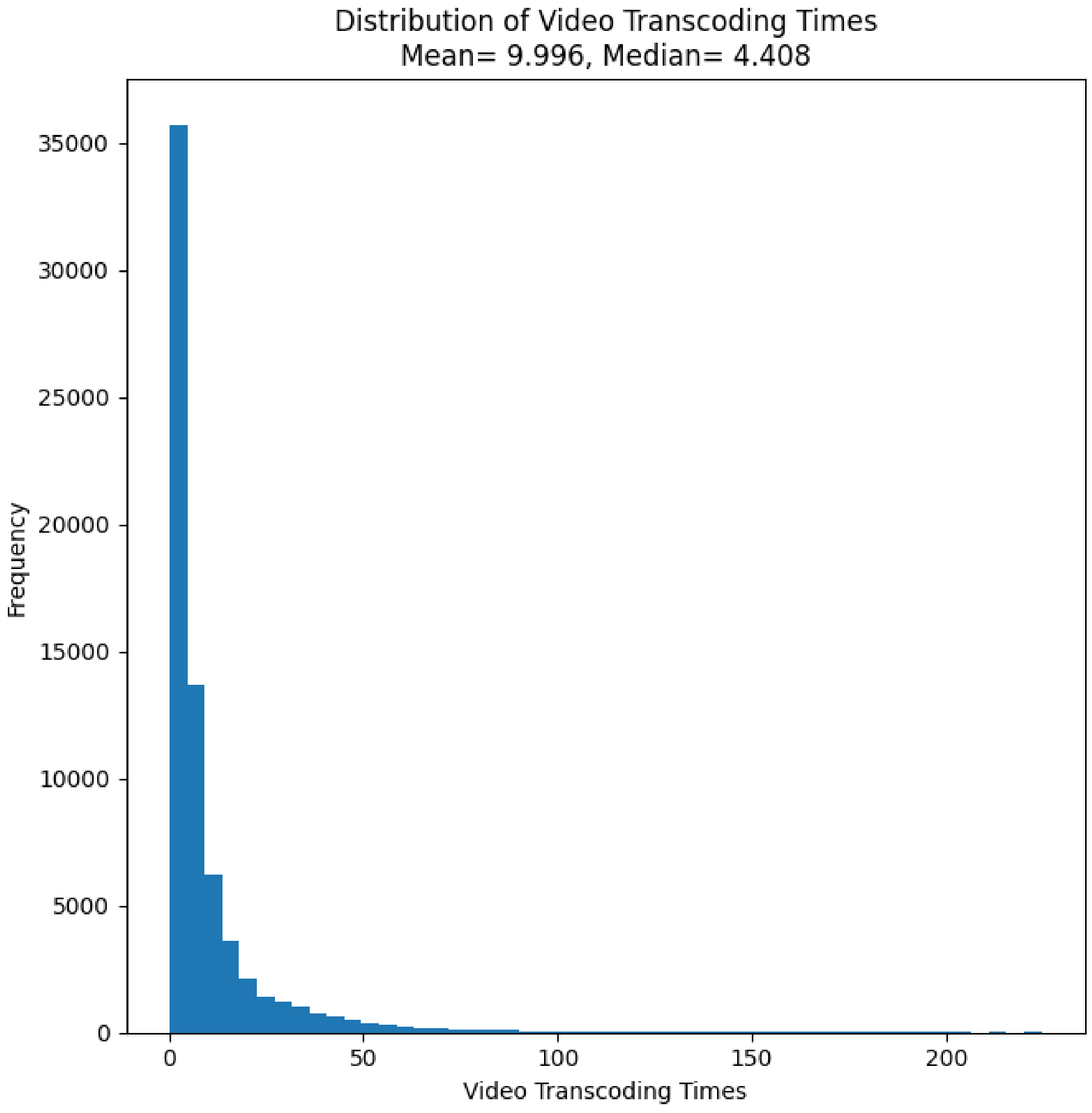


Figure 36: Distribution of video encoding times.

The distribution of video transcoding times is skewed to the right. This means that there is a significantly higher quantity of data that has a relatively low transcoding time in comparison to the quantity of data that has a high transcoding time. Essentially, there is much more data that has lower transcoding time.

Question 7.

Almost all data can be characterized in terms of a hierarchy of "level of measurement":

- *Nominal* data is purely categorical, such as "cat" and "dog".

- *Ordinal* data has an order but no units of measurement, such as “cold” and “hot”.

- *Interval* data has both an order and units of measurements, but no concept of *zero*, such as “30ºF” and “60ºF”, or “credit score of 720”. In these cases, *differences* between observations are meaningful (e.g. “hotter by 30ºF”), but ratios are not (e.g. “2 times as hot!”).

- *Ratio* data is numerical with meaningful zero measurements, such as “length” and “net worth”. In this case, both differences and ratios are meaningful.

Of the various levels of measurement, the only gray area in terms of one-hot versus scalar encoding would occur with features that are technically ordinal, but whose order may be irrelevant to measurements. Of course, scalar encoding should never be used if there is no inherent order to a feature. One-hot encoding of “generation” in the suicide dataset, for example, would discard the fact that the “G.I. Generation” preceded all the others; scalar encoding the generations to values of 1 through 6, on the other hand, would impose a somewhat arbitrary interval structure to the generations. The difference between a member of “Generation Z” and a “Boomer” is not necessarily the same as the difference between a “Millenial” and a member of the “Silent Generation” both because the spans of the generations are not consistent and because “generation” as a concept inherently attempts to capture a cultural cohort with qualitative differences rather than a numerically defined one.

Cyclical but technically ordered features, such as weekday or month, may at times be useful to encode in either way, depending on the method. For example, if the week is defined to start at Monday, there is a decreasing “number of days until the next Sunday”, which might be relevant to, say, a club that meets on Sundays (like a church), in which case a scalar encoding of weekdays may be warranted.

Essentially, by choosing one-hot encoding, we discard the possibility of including relationships between different data for a particular feature and by choosing to perform scalar encoding instead we should strongly assume that there is some existing relationship between the different data for a particular feature. An example would be the “generation” feature where there is a time-based relationship for the scalar encoding. The encoding for the feature of $1 < 2 < 3 < 4 < 5 < 6$ will help represent the generation’s order in time and which generation came before or after each other generation.

Question 8.

We standardized the feature columns and prepared them for trading. Standardization ensures that the individual features will look like standard normally distributed data, i.e Gaussian with zero mean and unit variance. If we do not standardize, certain features will dominate the objective function and prevent the model from learning from other features.

The code for this is as follows.

```
def scale_bike():
    return encode_bike()
```

```
def scale_suicide():
    suicide_df_final = encode_suicide()
    scaler = StandardScaler()
    suicide_df_final[' gdp_for_year ($) '] = suicide_df_final[' gdp_for_year ($) '].apply(lambda x:
        float(x.split()[0].replace(',', '')))
    # Standardize feature columns of suicide data.
    suicide_df_final[['year', 'population', ' gdp_for_year ($) ', 'gdp_per_capita ($)']] =
        scaler.fit_transform(suicide_df_final[['year', 'population', ' gdp_for_year ($) ', 'gdp_per_capita ($)']])
    return suicide_df_final
```

```
def scale_video():
    video_df_final = encode_video()
    scaler = StandardScaler()
    video_df_final[['duration', 'height', 'width', 'bitrate','framerate', 'i', 'p', 'b', 'frames', 'i_size', 'p_size',
        'b_size','size', 'o_bitrate', 'o_framerate', 'o_width', 'o_height']] =
        scaler.fit_transform(video_df_final[['duration', 'height', 'width','bitrate', 'framerate', 'i', 'p', 'b',
        'frames', 'i_size', 'p_size', 'b_size', 'size','o_bitrate', 'o_framerate', 'o_width', 'o_height']])
    return video_df_final
```

Question 9.

Rather than selecting one set of features to use with all the regression methods (linear, polynomial, neural network, tree), we decided to perform model selection separately for different methods. There were several motivations. First, we wished to use the `sklearn.feature_selection` module's `f_regression` implementation where it is most appropriate, which is in the case of linear regressors, but we felt it necessary to use `mutual_info_regression` for all nonlinear regression techniques. Second, we recognized that in a real application, even if several regression techniques are being considered, model selection decisions will ultimately follow the choice of regression method, and so our model selection too is performed in a way that is aware of the regression method.

For all of our regression methods, we used the same backwards stepwise model selection scheme as follows:

1. We select the least informative or "weakest" feature with respect to the target variable. If the regression method was linear (including regularized linear regression using LASSO and ridge regression, as well as polynomial regression), we selected the "weakest" feature as the one whose F score (from the Wald F-test implemented by `f_regression`) had the highest p-value. If the regression method was nonlinear (broadly, including the multilayer perceptron and decision tree), we identified the least informative feature as the one with the lowest mutual information score with respect to the target variable.

2. We calculated the average out-of-sample (test) performance of the regression method both *with* and *without* the "weakest" feature via 10-fold cross-validation. If the out-of-sample performance improved after removing the feature, we keep it removed and return to Step 1. If the performance was worse after removing this feature, we terminated the model selection procedure without removing that feature.

Including more variables in a model always results in lower training error (RMSE), but including too many features in a model can result in overfitting: worse performance on test (out-of-sample) data after fitting to noise (or collinear variables). Removing too many variables, however, naturally results in a loss of information.

Our backwards stepwise model selection procedure is designed to achieve a good out-of-sample RMSE. With M total possible features, it takes at most M iterations of cross-validation. Instead of stepwise model selection, the *best model selection* technique evaluated all 2^M models (by including and excluding every feature) and evaluates out-of-sample performance on each one (or sometimes simply evaluates the p-values of F-statistics); this is obviously much more computationally expensive and does not necessarily yield significant improvements, especially when most variables are either at least somewhat informative or completely uninformative.

We settled on 10-fold cross-validation for model selection because, unlike 5-fold, it produced relatively consistent model selection decisions between applications on the same data. Out-of-sample performance is of course affected by the random splitting of the data, but with more folds, one can interpret out-of-sample performance average as a reasonable proxy for performance overall on unseen data.

Figure 46 demonstrates the output from this model selection procedure with several regression methods. For all linear models, model selection was based on the LASSO regularized linear regression. Note that the more powerful regression methods, such as the multilayer perceptron, tended to prune fewer features, while the simplest methods, unregularized and regularized linear regression, tended to prune the most. This may be in part because those more complicated models are able to learn from nonlinear relationships between features and targets that cause linear models to perform badly, yielding improved out-of-sample performance. Some features were totally uninformative and entirely equal to zero (such as `b_size` in the video dataset), and it was esentially random whether models had better or worse out-of-sample performance with and without those features. They could have been removed manually, but we elected to let this automated system operate without intervention for the remainder of our project.

```
Model is 'LinearRegression()', selecting using p-Values of F-statistics
Number of features BEFORE model selection:
{'bike_cnt': 12, 'video_utime': 25, 'suicide_per_100k': 13}
From bike_cnt pruned: 'day_number'
From video_utime pruned: ''
From suicide_per_100k pruned: 'population', 'gdp_per_capita ($)'
Number of features AFTER model selection:
{'bike_cnt': 11, 'video_utime': 25, 'suicide_per_100k': 11}

Model is 'Lasso(alpha=0.5)', selecting using p-Values of F-statistics
Number of features BEFORE model selection:
{'bike_cnt': 12, 'video_utime': 25, 'suicide_per_100k': 13}
From bike_cnt pruned: 'day_number'
From video_utime pruned: 'b', 'duration', 'codec_flv', 'b_size'
From suicide_per_100k pruned: 'population', 'gdp_per_capita ($)'
Number of features AFTER model selection:
{'bike_cnt': 11, 'video_utime': 21, 'suicide_per_100k': 11}

Model is 'Ridge(alpha=0.5)', selecting using p-Values of F-statistics
Number of features BEFORE model selection:
{'bike_cnt': 12, 'video_utime': 25, 'suicide_per_100k': 13}
From bike_cnt pruned: 'day_number'
From video_utime pruned: 'b', 'duration', 'codec_flv', 'b_size'
From suicide_per_100k pruned: 'population', 'gdp_per_capita ($)'
Number of features AFTER model selection:
{'bike_cnt': 11, 'video_utime': 21, 'suicide_per_100k': 11}

Model is 'MLPRegressor()', selecting using Mutual Information
Number of features BEFORE model selection:
{'bike_cnt': 12, 'video_utime': 25, 'suicide_per_100k': 13}
From bike_cnt pruned: ''
From video_utime pruned: ''
From suicide_per_100k pruned: 'year', 'Continent_Oceania'
Number of features AFTER model selection:
{'bike_cnt': 12, 'video_utime': 25, 'suicide_per_100k': 11}

Model is 'RandomForestRegressor(max_depth=30, n_estimators=10)', selecting using Mutual Information
Number of features BEFORE model selection:
{'bike_cnt': 12, 'video_utime': 25, 'suicide_per_100k': 13}
From bike_cnt pruned: ''
From video_utime pruned: ''
From suicide_per_100k pruned: ''
Number of features AFTER model selection:
{'bike_cnt': 12, 'video_utime': 25, 'suicide_per_100k': 13}
```

Figure 37: Model selection output.

Preliminaries for Linear Regression

For Linear Regression, the objective function is $\frac{1}{N}\sum_{i=1}^{N}(y_i - x_i^T w)$ where N is the amount of data and $w \in \mathbb{R}^M$ where M is the dimension of the data.

For Lasso Regression, the objective function is $\frac{1}{N}\sum_{i=1}^{N}(y_i - x_i^T w) + \alpha \sum_{j=1}^{M} |w_j|$ where N is the amount of data and $w \in \mathbb{R}^M$ where M is the dimension of the data. α is the optimal penalty parameter.

For Ridge Regression, the objective function is $\frac{1}{N}\sum_{i=1}^{N}(y_i - x_i^T w) + \alpha \sum_{j=1}^{M} w_j^2$ where N is the amount of data and $w \in \mathbb{R}^M$ where M is the dimension of the data. α is the optimal penalty parameter.

NOTE WE REFER TO AVG. VALIDATION RMSE AS AVG. TEST RMSE

Before we post the results, we would like to state that we ran two different types of 10-fold cross-validations. In the first 10-fold cross-validation, we did not shuffle the data. The results for Linear Regression, Lasso Regression, and Ridge Regression under this scheme are as follows. The test and training RMSE are reported.

```
Linear Regression Results:
bike_cnt: Avg. Training RMSE Error: 858.3992946263372, Avg. Test RMSE Error: 968.5311654716763
video_utime: Avg. Training RMSE Error: 10.997562076001017, Avg. Test RMSE Error: 514346015.3039211
suicide_per_100k: Avg. Training RMSE Error: 15.411745704533828, Avg. Test RMSE Error: 15.462239361119572
```

Figure 38: Linear Regression Average Training and Test RMSE.

```
Lasso Results for 0.1:
bike_cnt: Avg. Training RMSE Error: 858.4146638664439, Avg. Test RMSE Error: 966.7730654036126
video_utime: Avg. Training RMSE Error: 11.025732559781668, Avg. Test RMSE Error: 11.03714614532262
suicide_per_100k: Avg. Training RMSE Error: 15.443477798429985, Avg. Test RMSE Error: 15.381608639984808
Lasso Results for 0.2:
bike_cnt: Avg. Training RMSE Error: 858.4312276293136, Avg. Test RMSE Error: 966.2579193721401
video_utime: Avg. Training RMSE Error: 11.058002885463091, Avg. Test RMSE Error: 11.057549706674925
suicide_per_100k: Avg. Training RMSE Error: 15.46171293476075, Avg. Test RMSE Error: 15.364193519275394
Lasso Results for 0.5:
bike_cnt: Avg. Training RMSE Error: 858.4823912507554, Avg. Test RMSE Error: 966.164535700412
video_utime: Avg. Training RMSE Error: 11.2523963787722, Avg. Test RMSE Error: 11.242823923710452
suicide_per_100k: Avg. Training RMSE Error: 15.559767151651005, Avg. Test RMSE Error: 15.44481150863931
Lasso Results for 0.75:
bike_cnt: Avg. Training RMSE Error: 858.558259110852, Avg. Test RMSE Error: 966.1222332372769
video_utime: Avg. Training RMSE Error: 11.530232567219468, Avg. Test RMSE Error: 11.515811299935939
suicide_per_100k: Avg. Training RMSE Error: 15.685354324305129, Avg. Test RMSE Error: 15.536603394267956
Lasso Results for 1.0:
bike_cnt: Avg. Training RMSE Error: 858.6643252126651, Avg. Test RMSE Error: 966.1115351117472
video_utime: Avg. Training RMSE Error: 11.763476758162959, Avg. Test RMSE Error: 11.74479724869594
suicide_per_100k: Avg. Training RMSE Error: 15.818253718061133, Avg. Test RMSE Error: 15.635351043488054
Lasso Results for 2.0:
bike_cnt: Avg. Training RMSE Error: 859.3903476637545, Avg. Test RMSE Error: 966.3894637127984
video_utime: Avg. Training RMSE Error: 12.966345466495875, Avg. Test RMSE Error: 12.964952341164807
suicide_per_100k: Avg. Training RMSE Error: 16.551348958759352, Avg. Test RMSE Error: 16.333953688782906
Lasso Results for 5.0:
bike_cnt: Avg. Training RMSE Error: 864.4762716939279, Avg. Test RMSE Error: 969.4772883871125
video_utime: Avg. Training RMSE Error: 14.606082921744783, Avg. Test RMSE Error: 14.556751873559216
suicide_per_100k: Avg. Training RMSE Error: 17.905796514170003, Avg. Test RMSE Error: 17.659428057881996
Lasso Results for 10.0:
bike_cnt: Avg. Training RMSE Error: 879.0722820787902, Avg. Test RMSE Error: 979.6595799789047
video_utime: Avg. Training RMSE Error: 16.106139737795253, Avg. Test RMSE Error: 16.056460831129616
suicide_per_100k: Avg. Training RMSE Error: 18.611121483038087, Avg. Test RMSE Error: 18.343385391304135
Best Alphas for each Dataset:
bike_cnt(Alpha: 1.0, Avg. Training RMSE: 858.6643252126651, Avg. Test RMSE: 966.1115351117472)
video_utime(Alpha: 0.1, Avg. Training RMSE: 11.025732559781668, Avg. Test RMSE: 11.03714614532262)
suicide_per_100k(Alpha: 0.2, Avg. Training RMSE: 15.46171293476075, Avg. Test RMSE: 15.364193519275394)
```

Figure 39: Lasso Regression Average Training and Test RMSE for various parameters.

```
Ridge Results for 0.1:
bike_cnt: Avg. Training RMSE Error: 858.5458726275226, Avg. Test RMSE Error: 963.8010221972997
video_utime: Avg. Training RMSE Error: 10.997600127515732, Avg. Test RMSE Error: 11.051462527858124
suicide_per_100k: Avg. Training RMSE Error: 15.41120173118214, Avg. Test RMSE Error: 15.463116095084606
Ridge Results for 0.2:
bike_cnt: Avg. Training RMSE Error: 858.6988625938366, Avg. Test RMSE Error: 962.744109319165
video_utime: Avg. Training RMSE Error: 10.997608774204727, Avg. Test RMSE Error: 11.051318387604644
suicide_per_100k: Avg. Training RMSE Error: 15.411201734447705, Avg. Test RMSE Error: 15.463100574544459
Ridge Results for 0.5:
bike_cnt: Avg. Training RMSE Error: 859.2238984185991, Avg. Test RMSE Error: 961.7314839154363
video_utime: Avg. Training RMSE Error: 10.997616881686424, Avg. Test RMSE Error: 11.051118653737474
suicide_per_100k: Avg. Training RMSE Error: 15.411201757298135, Avg. Test RMSE Error: 15.463054049334364
Ridge Results for 0.75:
bike_cnt: Avg. Training RMSE Error: 859.7807072098306, Avg. Test RMSE Error: 961.6033124446997
video_utime: Avg. Training RMSE Error: 10.997620648767398, Avg. Test RMSE Error: 11.05100182182149
suicide_per_100k: Avg. Training RMSE Error: 15.411201791286453, Avg. Test RMSE Error: 15.463015320000025
Ridge Results for 1.0:
bike_cnt: Avg. Training RMSE Error: 860.4284998655185, Avg. Test RMSE Error: 961.7867960016977
video_utime: Avg. Training RMSE Error: 10.997624175409777, Avg. Test RMSE Error: 11.050901911020182
suicide_per_100k: Avg. Training RMSE Error: 15.411201838849854, Avg. Test RMSE Error: 15.46297662849968
Ridge Results for 2.0:
bike_cnt: Avg. Training RMSE Error: 863.5630212190721, Avg. Test RMSE Error: 964.2597518945182
video_utime: Avg. Training RMSE Error: 10.99763959032028, Avg. Test RMSE Error: 11.050593313488669
suicide_per_100k: Avg. Training RMSE Error: 15.411202164621567, Avg. Test RMSE Error: 15.462822239875678
Ridge Results for 5.0:
bike_cnt: Avg. Training RMSE Error: 874.78837936183, Avg. Test RMSE Error: 979.0208418062745
video_utime: Avg. Training RMSE Error: 10.997690913813045, Avg. Test RMSE Error: 11.050051230479006
suicide_per_100k: Avg. Training RMSE Error: 15.411204436624823, Avg. Test RMSE Error: 15.462362670882916
Ridge Results for 10.0:
bike_cnt: Avg. Training RMSE Error: 895.5826692875683, Avg. Test RMSE Error: 1012.3486916126327
video_utime: Avg. Training RMSE Error: 10.997765134349772, Avg. Test RMSE Error: 11.049565757237742
suicide_per_100k: Avg. Training RMSE Error: 15.411212495190165, Avg. Test RMSE Error: 15.461608532411066
Best Alphas for each Dataset:
bike_cnt(Alpha: 0.75, Avg. Training RMSE: 859.7807072098306, Avg. Test RMSE: 961.6033124446997)
video_utime(Alpha: 10.0, Avg. Training RMSE: 10.997765134349772, Avg. Test RMSE: 11.049565757237742)
suicide_per_100k(Alpha: 10.0, Avg. Training RMSE: 15.411212495190165, Avg. Test RMSE: 15.461608532411066)
```

Figure 40: Ridge Regression Average Training and Test RMSE for various parameters.

From this, we immediately notice certain things. For Linear Regression, Lasso Regression, and Ridge Regression we notice that the Average Test RMSE is generally higher than the Average Training RMSE. This makes sense as since we train our model on the training data we are more likely to overfit to the training data. The test data is "new" unseen data which the model hasn't interacted with before so the performance on this set of data will be lower. **For the bike dataset, including regularization improves the Average Test RMSE and Ridge Regression with an optimal penalty parameter of .75 performs the best.** For the video dataset, we noticed that the Average Test RMSE for Linear Regression was abnormally high. We examined the test error for each split for the Linear Regression model and found

```
array([1.16495703e+01, 1.07933894e+01, 1.09094885e+01, 1.14032516e+01,
       1.12182208e+01, 1.22335231e+01, 1.01737965e+01, 1.09059309e+01,
       9.54406450e+00, 5.14346005e+09])
```

We noticed that the performance on the last split was abnormally poor and was the source of the poor test error. We address this later with our second cross-validation scheme. However, we also noticed that this is not an issue for the Lasso and Ridge Regression models as the average test error is normal. This seems to suggest that for the last split, the Linear Regression model overfit to training data and the regularization from the Lasso and Ridge Regression models helped the model generalize better. **For**

the video dataset, the Lasso model with an optimal penalty parameter of 0.1 performed the best. For the suicide dataset, the Lasso model with an optimal penalty parameter of 0.2 performed the best.

To amend the extremely large Avg. Test MSE from the Linear Regression model trained on the video dataset, we found an alternate approach where we shuffled the data before creating our training/test splits for cross validation. The results are as follows:

```
Linear Regression Shuffling Data Results:
bike_cnt: Avg. Training RMSE Error: 863.7235738228258, Avg. Test RMSE Error: 894.54087313473
video_utime: Avg. Training RMSE Error: 11.001123804928229, Avg. Test RMSE Error: 11.00042389379597
suicide_per_100k: Avg. Training RMSE Error: 15.428338341150823, Avg. Test RMSE Error: 15.423805325551916
```

Figure 41: Linear Regression Shuffled Data Average Training and Test RMSE.

```
Lasso Shuffling Data Results for 0.1:
bike_cnt: Avg. Training RMSE Error: 863.7394573893305, Avg. Test RMSE Error: 890.9952880866119
video_utime: Avg. Training RMSE Error: 11.028838492873719, Avg. Test RMSE Error: 11.02586444035639
suicide_per_100k: Avg. Training RMSE Error: 15.45877266293876, Avg. Test RMSE Error: 15.452613481132152
Lasso Shuffling Data Results for 0.2:
bike_cnt: Avg. Training RMSE Error: 863.7860418483735, Avg. Test RMSE Error: 887.7690318805835
video_utime: Avg. Training RMSE Error: 11.05970906852912, Avg. Test RMSE Error: 11.055849415130297
suicide_per_100k: Avg. Training RMSE Error: 15.473533080376665, Avg. Test RMSE Error: 15.466892937136453
Lasso Shuffling Data Results for 0.5:
bike_cnt: Avg. Training RMSE Error: 864.0084256685559, Avg. Test RMSE Error: 882.0658054974216
video_utime: Avg. Training RMSE Error: 11.253839624344991, Avg. Test RMSE Error: 11.249344154061548
suicide_per_100k: Avg. Training RMSE Error: 15.571493331598878, Avg. Test RMSE Error: 15.56459800340543
Lasso Shuffling Data Results for 0.75:
bike_cnt: Avg. Training RMSE Error: 864.0833276361411, Avg. Test RMSE Error: 882.1832078123085
video_utime: Avg. Training RMSE Error: 11.533133192774274, Avg. Test RMSE Error: 11.528434376290331
suicide_per_100k: Avg. Training RMSE Error: 15.703439090938414, Avg. Test RMSE Error: 15.696221653495002
Lasso Shuffling Data Results for 1.0:
bike_cnt: Avg. Training RMSE Error: 864.1880438419021, Avg. Test RMSE Error: 882.3305839101656
video_utime: Avg. Training RMSE Error: 11.764255755193897, Avg. Test RMSE Error: 11.75922966823086
suicide_per_100k: Avg. Training RMSE Error: 15.827681892407238, Avg. Test RMSE Error: 15.819278499973226
Lasso Shuffling Data Results for 2.0:
bike_cnt: Avg. Training RMSE Error: 864.90487146758, Avg. Test RMSE Error: 883.2195233583761
video_utime: Avg. Training RMSE Error: 12.965075786979302, Avg. Test RMSE Error: 12.959517856908647
suicide_per_100k: Avg. Training RMSE Error: 16.568762930002496, Avg. Test RMSE Error: 16.55981044402459
Lasso Shuffling Data Results for 5.0:
bike_cnt: Avg. Training RMSE Error: 869.9019890359175, Avg. Test RMSE Error: 888.7483528811579
video_utime: Avg. Training RMSE Error: 14.607226161525071, Avg. Test RMSE Error: 14.600594882706934
suicide_per_100k: Avg. Training RMSE Error: 17.91193650749951, Avg. Test RMSE Error: 17.903837570340368
Lasso Shuffling Data Results for 10.0:
bike_cnt: Avg. Training RMSE Error: 883.8909088768484, Avg. Test RMSE Error: 902.6287962501283
video_utime: Avg. Training RMSE Error: 16.10720779470957, Avg. Test RMSE Error: 16.1005317090175
suicide_per_100k: Avg. Training RMSE Error: 18.616827385881482, Avg. Test RMSE Error: 18.609014033884904
Best Alphas for each Dataset:
bike_cnt(Alpha: 0.5, Avg. Training RMSE: 864.0084256685559, Avg. Test RMSE: 882.0658054974216)
video_utime(Alpha: 0.1, Avg. Training RMSE: 11.028838492873719, Avg. Test RMSE: 11.02586444035639)
suicide_per_100k(Alpha: 0.1, Avg. Training RMSE: 15.45877266293876, Avg. Test RMSE: 15.452613481132152)
```

Figure 42: Lasso Regression Shuffled Data Average Training and Test RMSE for various parameters.

```
Ridge Shuffled Data Results for 0.1:
bike_cnt: Avg. Training RMSE Error: 864.0105378667861, Avg. Test RMSE Error: 881.563076833331
video_utime: Avg. Training RMSE Error: 11.001101756126971, Avg. Test RMSE Error: 10.999892044915596
suicide_per_100k: Avg. Training RMSE Error: 15.428068568214178, Avg. Test RMSE Error: 15.423543249579183
Ridge Shuffled Data Results for 0.2:
bike_cnt: Avg. Training RMSE Error: 864.2152822873531, Avg. Test RMSE Error: 879.7034085988658
video_utime: Avg. Training RMSE Error: 11.001110268711209, Avg. Test RMSE Error: 10.99989071372252
suicide_per_100k: Avg. Training RMSE Error: 15.428068571277928, Avg. Test RMSE Error: 15.423543025917516
Ridge Shuffled Data Results for 0.5:
bike_cnt: Avg. Training RMSE Error: 864.7655563401298, Avg. Test RMSE Error: 878.6358643436763
video_utime: Avg. Training RMSE Error: 11.001117944381408, Avg. Test RMSE Error: 10.999883387267108
suicide_per_100k: Avg. Training RMSE Error: 15.428068592717432, Avg. Test RMSE Error: 15.423542367389672
Ridge Shuffled Data Results for 0.75:
bike_cnt: Avg. Training RMSE Error: 865.3139153489259, Avg. Test RMSE Error: 878.6760295021741
video_utime: Avg. Training RMSE Error: 11.001121251160166, Avg. Test RMSE Error: 10.999877634376995
suicide_per_100k: Avg. Training RMSE Error: 15.428068624609304, Avg. Test RMSE Error: 15.423541832881261
Ridge Shuffled Data Results for 1.0:
bike_cnt: Avg. Training RMSE Error: 865.946109001318, Avg. Test RMSE Error: 878.9761195891246
video_utime: Avg. Training RMSE Error: 11.001124239879474, Avg. Test RMSE Error: 10.999872691935554
suicide_per_100k: Avg. Training RMSE Error: 15.428068669241839, Avg. Test RMSE Error: 15.42354131133077
Ridge Shuffled Data Results for 2.0:
bike_cnt: Avg. Training RMSE Error: 868.9917924011804, Avg. Test RMSE Error: 881.249681195424
video_utime: Avg. Training RMSE Error: 11.001137123217614, Avg. Test RMSE Error: 10.999859833167722
suicide_per_100k: Avg. Training RMSE Error: 15.42806897499363, Avg. Test RMSE Error: 15.423539354519338
Ridge Shuffled Data Results for 5.0:
bike_cnt: Avg. Training RMSE Error: 879.8960009519, Avg. Test RMSE Error: 891.2063799536702
video_utime: Avg. Training RMSE Error: 11.001181165613804, Avg. Test RMSE Error: 10.999854742338604
suicide_per_100k: Avg. Training RMSE Error: 15.428071108570697, Avg. Test RMSE Error: 15.423534721131807
Ridge Shuffled Data Results for 10.0:
bike_cnt: Avg. Training RMSE Error: 900.2440293954265, Avg. Test RMSE Error: 911.0449289448704
video_utime: Avg. Training RMSE Error: 11.001247227282665, Avg. Test RMSE Error: 10.999876898167276
suicide_per_100k: Avg. Training RMSE Error: 15.42807868398659, Avg. Test RMSE Error: 15.4235310866848
Best Alphas for each Dataset:
bike_cnt(Alpha: 0.5, Avg. Training RMSE: 864.7655563401298, Avg. Test RMSE: 878.6358643436763)
video_utime(Alpha: 5.0, Avg. Training RMSE: 11.001181165613804, Avg. Test RMSE: 10.999854742338604)
suicide_per_100k(Alpha: 10.0, Avg. Training RMSE: 15.42807868398659, Avg. Test RMSE: 15.4235310866848)
```

Figure 43: Ridge Regression Shuffled Data Average Training and Test RMSE for various parameters.

As we can see from this data, shuffling the data prevented the large Average Test RMSE found from the Linear Regression model when training on the video dataset. This is probably because by shuffling the data, we spread around the data from the original last split to prevent overfitting. This is the scheme that we have decided to base our findings and answers on. **In this case, we find the following. For the bike dataset, the best regularization scheme was Ridge Regression with an optimal penalty parameter of 0.5. For the video dataset, the best regularization scheme was Ridge Regression with an optimal penalty parameter of 5.0. For the suicide dataset, the best regularization scheme was Ridge Regression with an optimal penalty parameter of 10.**

Question 10.

Lasso and Ridge regression both work to reduce model complexity by using a regularization parameter to penalize large weights learned by the model which would cause the model to overfit to the training data. If it overfits to the training data, the model will generalize poorly and prevent the model from performing well on out-of-sample data (data it has not yet seen). However, too strong of a regularization parameter can lead to a bad model with poor performance on the test data as it could limit the complexity of the model. Lasso regression works by adding the following term to the objective function for least squares: $\alpha \sum_i |w_i|$ where w is the coefficients and w_i is each individual coefficient. Ridge regression works by adding $\alpha \sum_i w_i^2$. The difference is that L1 Regularization shrinks the less important feature's coefficients to zero which removes some features entirely. On the other hand, L2 regularization only reduces the value of those coefficients meaning that those features will still contribute to the model. These differences mean that L1 regularization works particularly well for feature selection when there is a lot of features as it will essentially reduce the number of features. L2 regularization shrinks the coefficients evenly which means that it will help with co-linear/codependent features as codependent features increase coefficient variance but L2 regularization reduces this variance.

The Lasso and Ridge Regression models overall improved the test RMSE for all 3 datasets in comparison to just Linear Regression as we can see in the data before although the training RMSE suffered a little which is not an issue. This is in line with our logic above as including a regularization parameter should help with our model's generalization ability and prevents overfitting.

Question 11.

The following answer is based off our alternate approach cross validation where we shuffled the data before creating our training/test splits for cross validation.

For the bike dataset, the best regularization scheme was Ridge Regression with an optimal penalty parameter of 0.5. For the video dataset, the best regularization scheme was Ridge Regression with an optimal penalty parameter of 5.0. For the suicide dataset, the best regularization scheme is Ridge Regression with an optimal penalty parameter of 10.0. However, we note that the Avg. Test RMSE when compared to the best from Lasso Regression is extremely close. In addition, while tuning the optimal parameter gave some improvement, it was not exactly extremely major. The Average Training and Test RMSEs for these best models can be seen in the logs from before in the Linear Regression preliminary section in the Question 10 section.

To find the optimal penalty parameter, we used k-fold cross-validation. We chose a range of parameter values to test (0.1, 0.2, 0.5, 0.75, 1.0, 2.0, 5.0, 10.0). We then split our data into k folds (for our specific case we chose k=10) and choose k-1 folds to be our training data and the remaining fold to test on/be the validation data set. We alternate which fold to use as the validation data set a total of k times. By doing it this way, we then have k Training and Test RMSEs which we average to find the Average Training and Test RMSE. We follow this procedure for the all the optimal penalty parameters we wish to test on and are left with multiple Test RMSEs to compare. We determine which penalty parameter is "optimal" by finding the one that corresponded with the model with the lowest Avg. Test RMSE.

Question 12.

To determine whether feature scaling plays a role, we reused the parameters of the best models for Linear Regression, Lasso Regression, and Ridge Regression found when we performed feature scaling on the data and retrained Linear Regression, Lasso Regression, and Ridge Regression models using these parameters with with data that is not featured scaled.

```
No Feature Scaling Test for Linear Regression; Using the best models from Feature Scaling
Bike Sharing Dataset: Avg. Training RMSE Error: 858.3992946263372, Avg. Test RMSE Error: 968.5311654716763
Video Dataset: Avg. Training RMSE Error: 10.997560341082401, Avg. Test RMSE Error: 11.091471352756638
Suicide Dataset: Avg. Training RMSE Error: 15.411201730174634, Avg. Test RMSE Error: 15.463128691256141
```

Figure 44: Linear Regression no Feature Scaling Average Training and Test RMSE.

```
No Feature Scaling Test for Lasso Regression; Using the best models from Feature Scaling
Bike Sharing Dataset: Avg. Training RMSE Error: 858.4823912507554, Avg. Test RMSE Error: 966.164535700412
Video Dataset: Avg. Training RMSE Error: 11.014739159272656, Avg. Test RMSE Error: 11.057294281259278
Suicide Dataset: Avg. Training RMSE Error: 15.437205794728683, Avg. Test RMSE Error: 15.397108825949909
```

Figure 45: Lasso Regression no Feature Scaling Average Training and Test RMSE.

```
No Feature Scaling Test for Ridge Regression; Using the best models from Feature Scaling
Bike Sharing Dataset: Avg. Training RMSE Error: 864.7655563401298, Avg. Test RMSE Error: 878.6358643436763
Video Dataset: Avg. Training RMSE Error: 11.00105754324312, Avg. Test RMSE Error: 11.000387308237787
Suicide Dataset: Avg. Training RMSE Error: 15.428077928133268, Avg. Test RMSE Error: 15.423531257948804
```

Figure 46: Ridge Regression no Feature Scaling Average Training and Test RMSE.

For Linear Regression with no regularization, we find that by not performing feature scaling, the Average Test RMSE was higher than if we performed feature scaling for all three datasets. On the other hand, for Lasso and Ridge Regression, we did not see that significant, if any, of a difference between the Lasso and Ridge Regression models trained on data using feature scaling and not using feature scaling. This makes sense as the Lasso and Ridge Regression models include a penalty for penalizing weights learned by the model which are too large. If we don't feature scale, we might cause features that have a variance that is orders of magnitude larger than others to dominate the objective function and make the estimator unable to learn from other features correctly. The penalty parameter from Lasso and Ridge regression serve a similar function which is why we think we don't see a noticeable difference when we train Lasso and Ridge Regression models using feature scaled data and not using feature scaled data.

Question 13.

The p-value for each feature tests the null hypothesis of whether the coefficient corresponding to that feature is equal to zero (having no effect). A low p-value (<0.05) allows us to reject the null hypothesis.

This means that features with a low p-value will probably be valuable for our model and that changes in our feature will affect the target. On the other hand, a larger p-value means that changing the value of the feature will not really affect our target. The most significant features would be those with low p-values.

We computed the p-values for each data set as follows.

```
Bike
                                 OLS Regression Results
=======================================================================================
Dep. Variable:                      y   R-squared (uncentered):                   0.967
Model:                            OLS   Adj. R-squared (uncentered):              0.967
Method:                 Least Squares   F-statistic:                              1932.
Date:                Fri, 19 Mar 2021   Prob (F-statistic):                        0.00
Time:                        03:50:37   Log-Likelihood:                         -5999.5
No. Observations:                 731   AIC:                                  1.202e+04
Df Residuals:                     720   BIC:                                  1.207e+04
Df Model:                          11
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
season       545.9072     55.796      9.784      0.000     436.364     655.450
yr          2123.6219     65.351     32.496      0.000    1995.320    2251.923
mnth         -40.3511     17.504     -2.305      0.021     -74.716      -5.986
holiday     -372.5598    204.592     -1.821      0.069    -774.228      29.109
weekday       94.4204     16.156      5.844      0.000      62.702     126.139
workingday   206.3674     72.383      2.851      0.004      64.260     348.475
weathersit  -643.3490     80.136     -8.028      0.000    -800.678    -486.020
temp        -159.9773   1391.149     -0.115      0.908   -2891.170    2571.215
atemp       6400.8083   1558.609      4.107      0.000    3340.847    9460.769
hum            6.9611    272.058      0.026      0.980    -527.160     541.082
windspeed   -852.3779    370.177     -2.303      0.022   -1579.133    -125.623
==============================================================================
Omnibus:                      111.698   Durbin-Watson:                   1.013
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              254.785
Skew:                          -0.835   Prob(JB):                     4.72e-56
Kurtosis:                       5.362   Cond. No.                         536.
==============================================================================
```

Figure 47: p-values for Bike Dataset

For the bike dataset, we found that the most significant features include season, year, month, weekday, workingday, atemp, and windspeed. The insignificant features are holiday, hum and temp. The temp feature being irrelevant is not surprising considering atemp is already accounted for so it would be redundant.

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.334
Model:                            OLS   Adj. R-squared:                  0.334
Method:                 Least Squares   F-statistic:                     1551.
Date:                Fri, 19 Mar 2021   Prob (F-statistic):               0.00
Time:                        21:17:48   Log-Likelihood:            -1.1567e+05
No. Observations:               27820   AIC:                         2.314e+05
Df Residuals:                   27810   BIC:                         2.314e+05
Df Model:                           9
Covariance Type:            nonrobust
============================================================================================
                              coef    std err          t      P>|t|      [0.025      0.975]
--------------------------------------------------------------------------------------------
Continent_Africa           -3.2873      0.456     -7.206      0.000      -4.181      -2.393
Continent_Asia             -0.1389      0.223     -0.624      0.532      -0.575       0.297
Continent_Europe            6.4046      0.184     34.886      0.000       6.045       6.764
Continent_North America    -3.9170      0.216    -18.112      0.000      -4.341      -3.493
Continent_Oceania           0.5065      0.429      1.181      0.238      -0.334       1.348
Continent_South America     0.4206      0.265      1.584      0.113      -0.100       0.941
year                        0.0043      0.001      8.334      0.000       0.003       0.005
sex                       -14.8410      0.186    -79.986      0.000     -15.205     -14.477
age                         3.4788      0.137     25.438      0.000       3.211       3.747
 gdp_for_year ($)        5.098e-13   6.46e-14      7.897      0.000    3.83e-13    6.36e-13
generation                 -0.7267      0.165     -4.408      0.000      -1.050      -0.404
==============================================================================
Omnibus:                    16816.956   Durbin-Watson:                   1.053
Prob(Omnibus):                  0.000   Jarque-Bera (JB):           256862.961
Skew:                           2.644   Prob(JB):                         0.00
Kurtosis:                      16.915   Cond. No.                     1.57e+15
==============================================================================
```

Figure 48: p-values for Suicide Dataset

For the suicide dataset, we found the Continent Africa, Continent Europe, Continent North America, year, sex, age, gpd for year, and generation were the most significant features. The insignificant features are Continent Asia, Continent Oceania, and Continent South America.

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.534
Model:                            OLS   Adj. R-squared:                  0.533
Method:                 Least Squares   F-statistic:                     3932.
Date:                Fri, 19 Mar 2021   Prob (F-statistic):               0.00
Time:                        21:08:13   Log-Likelihood:            -2.6255e+05
No. Observations:               68784   AIC:                         5.251e+05
Df Residuals:                   68763   BIC:                         5.253e+05
Df Model:                          20
Covariance Type:            nonrobust
=================================================================================
                    coef    std err          t      P>|t|      [0.025      0.975]
---------------------------------------------------------------------------------
codec_h264        0.0047      0.178      0.027      0.979      -0.344       0.353
codec_mpeg4      -1.8973      0.198     -9.559      0.000      -2.286      -1.508
codec_vp8        -0.8102      0.213     -3.806      0.000      -1.227      -0.393
height           -0.0108      0.002     -5.813      0.000      -0.014      -0.007
width             0.0052      0.001      5.593      0.000       0.003       0.007
bitrate        2.258e-06   9.22e-08     24.489      0.000    2.08e-06    2.44e-06
framerate         0.0600      0.010      6.269      0.000       0.041       0.079
i                -0.0081      0.002     -5.057      0.000      -0.011      -0.005
p                -0.0013      0.001     -1.765      0.078      -0.003       0.000
frames            0.0014      0.001      1.985      0.047    1.82e-05       0.003
i_size         6.056e-07   9.19e-07      0.659      0.510    -1.2e-06    2.41e-06
p_size         7.547e-07   9.18e-07      0.822      0.411   -1.05e-06    2.55e-06
size          -7.514e-07   9.18e-07     -0.818      0.413   -2.55e-06    1.05e-06
o_codec_flv     -16.7625      0.346    -48.476      0.000     -17.440     -16.085
o_codec_h264      1.9156      0.346      5.540      0.000       1.238       2.593
o_codec_mpeg4   -14.0543      0.346    -40.632      0.000     -14.732     -13.376
o_codec_vp8      -8.1342      0.346    -23.523      0.000      -8.812      -7.456
o_bitrate      1.429e-06    2.4e-08     59.580      0.000    1.38e-06    1.48e-06
o_framerate       0.2507      0.006     39.851      0.000       0.238       0.263
o_width           0.0160      0.001     24.306      0.000       0.015       0.017
o_height         -0.0041      0.001     -3.256      0.001      -0.007      -0.002
==============================================================================
Omnibus:                    62400.323   Durbin-Watson:                   1.352
Prob(Omnibus):                  0.000   Jarque-Bera (JB):          3594267.220
Skew:                           4.221   Prob(JB):                         0.00
Kurtosis:                      37.393   Cond. No.                     1.34e+09
==============================================================================
```

Figure 49: p-values for Video Dataset

For the video dataset, we found that the significant features are codec mpeg4, codec vp8, height, width, bitrate, framerate, i, frames, o codec flv, o codec h264, o codec mpeg4, o codec vp8, o bitrate, o framerate, o width, and o height. The insignificant features include code h264, p, i size, p size, and size.

Question 14.

We have got the 2-degree polynomial features to see the most salient features for three data sets. Before we get the p-value, three datasets were transformed into polynomial features (78, 253, 78). From these features, to filter the most significant features, we set the threshold of the p-value < 0.01 and interpret the relationship between the two.

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.900
Model:                            OLS   Adj. R-squared:                  0.889
Method:                 Least Squares   F-statistic:                     81.05
Date:                Sat, 20 Mar 2021   Prob (F-statistic):         1.48e-281
Time:                        21:26:19   Log-Likelihood:                -5727.9
No. Observations:                 731   AIC:                         1.160e+04
Df Residuals:                     657   BIC:                         1.194e+04
Df Model:                          73
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const      -2153.1092   1271.814     -1.693      0.091   -4650.419     344.201
x1          1312.3545    439.419      2.987      0.003     449.520    2175.189
x2           529.8758    199.144      2.661      0.008     138.840     920.912
x3           -63.5397    141.835     -0.448      0.654    -342.045     214.965
x4          -998.5420    723.439     -1.380      0.168   -2419.073     421.989
x5            76.2105    145.276      0.525      0.600    -209.051     361.472
x6           235.0714    218.017      1.078      0.281    -193.023     663.166
x7           878.8446    552.300      1.591      0.112    -205.641    1963.331
x8          1.103e+05   1.64e+04      6.732      0.000    7.81e+04    1.42e+05
x9         -1.051e+05   1.85e+04     -5.695      0.000   -1.41e+05   -6.89e+04
x10         9911.4347   2456.460      4.035      0.000    5087.975    1.47e+04
x11        -7316.1892   4098.615     -1.785      0.075   -1.54e+04     731.775
...
x26           53.1890     25.285      2.104      0.036       3.540     102.838
...
x30         1.167e+04   4251.725      2.745      0.006    3323.338       2e+04
...
x36          105.9881     33.134      3.199      0.001      40.927     171.050
...
x53        -2191.5237    904.918     -2.422      0.016   -3968.403    -414.644
x54         2541.0774   1027.897      2.472      0.014     522.717    4559.437
...
x73          3.67e+04   2.09e+04      1.756      0.080   -4344.907    7.77e+04
x74          6.83e+04   2.81e+04      2.431      0.015    1.31e+04    1.23e+05
...
x76        -9058.5501   3660.737     -2.475      0.014   -1.62e+04   -1870.396
x77        -1290.3948   3990.175     -0.323      0.747   -9125.428    6544.639
==============================================================================
Omnibus:                       81.470   Durbin-Watson:                   1.406
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              182.198
Skew:                          -0.632   Prob(JB):                     2.73e-40
Kurtosis:                       5.093   Cond. No.                     1.25e+16
==============================================================================
```

Figure 50: p-values for 2D Bike Dataset

For the bike dataset, x30(year*atemp) and x36(mnth*workingday) features have 0.006 and 0.001 p-values. For feature x30(year*atemp), we know that year can either take a value of 0 or 1. This means that the data belonging to the second year and the temperature of that particular day plays a huge determining factor in the count number. For feature x36(mnth*workingday), we can see that the whether the day is a working day in tandem with the day occurs plays an important factor in the total count. This makes sense as the total count includes casual users who will be less likely to ride a bike during the workday and the month can give information about the season which is also related to the weather conditions of that day.

```
OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.849
Model:                            OLS   Adj. R-squared:                  0.848
Method:                 Least Squares   F-statistic:                     1936.
Date:                Sat, 20 Mar 2021   Prob (F-statistic):               0.00
Time:                        21:26:21   Log-Likelihood:            -2.2378e+05
No. Observations:               68784   AIC:                         4.480e+05
Df Residuals:                   68584   BIC:                         4.498e+05
Df Model:                         199
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const         12.3286      5.277      2.336      0.019       1.986      22.671
x1            -5.5915      3.419     -1.636      0.102     -12.292       1.109
x2           -74.1710      8.742     -8.484      0.000     -91.305     -57.037
x3            -5.9767      3.354     -1.782      0.075     -12.550       0.596
x4             7.4587      5.765      1.294      0.196      -3.840      18.757
x5            -7.4754      3.663     -2.041      0.041     -14.655      -0.295
x6            53.5660     41.244      1.299      0.194     -27.273     134.405
x7            -0.7003      2.079     -0.337      0.736      -4.775       3.374
x8             0.5000      1.882      0.266      0.790      -3.188       4.188
x9            37.1478    156.946      0.237      0.813    -270.466     344.761
x10          -28.2543    161.912     -0.175      0.861    -345.602     289.093
x11          475.0359   1666.085      0.285      0.776   -2790.488    3740.560
x12         5702.9546   1.94e+04      0.294      0.769   -3.24e+04    4.38e+04
x13        -6088.6115   2.06e+04     -0.295      0.768   -4.65e+04    3.43e+04
x14           -0.9783      1.318     -0.742      0.458      -3.562       1.606
x15            9.1350      1.368      6.680      0.000       6.454      11.816
x16            0.5188      1.299      0.399      0.690      -2.027       3.064
x17            3.6531      1.317      2.775      0.006       1.073       6.234
x18            3.4964      0.093     37.590      0.000       3.314       3.679
x19            1.5260      0.499      3.057      0.002       0.548       2.504
x20            7.3310      2.404      3.049      0.002       2.619      12.043
x21           -2.4615      3.494     -0.704      0.481      -9.310       4.387
...
x28            0.8438      0.279      3.021      0.003       0.296       1.391
...
x39           -0.2548      0.102     -2.503      0.012      -0.454      -0.055
...
x65          -11.0914      3.896     -2.847      0.004     -18.727      -3.456
...
x92            4.1973      1.510      2.779      0.005       1.237       7.158
x93           -4.7902      1.503     -3.187      0.001      -7.736      -1.844
x94            4.4428      1.512      2.938      0.003       1.479       7.406
...
x109          -3.3127      1.009     -3.284      0.001      -5.290      -1.335
...
x130           0.1587      0.058      2.746      0.006       0.045       0.272
x131           1.4599      0.552      2.645      0.008       0.378       2.542
...
x144           0.1047      0.040      2.633      0.008       0.027       0.183
...
x238           3.6531      1.317      2.775      0.006       1.073       6.234
...
x250          62.0849     46.974      1.322      0.186     -29.983     154.153
x251         -98.0628     74.317     -1.320      0.187    -243.723      47.597
x252          38.9364     28.545      1.364      0.173     -17.012      94.885
==============================================================================
Omnibus:                    47058.956   Durbin-Watson:                   1.352
Prob(Omnibus):                  0.000   Jarque-Bera (JB):          3569463.480
Skew:                           2.574   Prob(JB):                         0.00
Kurtosis:                      37.913   Cond. No.                     5.75e+16
==============================================================================
```

Figure 51: p-values for 2D Video Transcoding Dataset

For the video transcoding dataset,
x65(codec_vp8*width), x92(height*o_codec_flv),

```
x93(height*o_codec_h264), x94(height*o_codec_mpeg4),
x109(width*o_codec_flv),x130(bitrate*o_framerate),
x131(bitrate*o_width) and x144(framerate*o_bitrate)
features have less than 0.01 p-value
```

Feature x65(codec vp8*width) seems to indicate that the input dimension combined with the input taking a particular coding standard is important in determining the total transcoding time for transcoding. Features x92(height*o codec flv), x93(height*o codec h264), x94(height*o codec mpeg4), x109(width*o codec flv), seem to indicate that the input dimension combined with the output coding standard is important in determining the total transcoding time for transcoding. The means in general the input/output coding time and dimension plays a huge role in the transcoding time. Features x130(bitrate*o framerate) and x144(framerate*o bitrate) seem to indicate the quality of the input/output video as a high frame rate will make the video seem less choppy while a high bitrate is how many bits we are sending allowing us to have a higher quality video. These two values together would signify the quality of the video and would have an obvious impact on the transcoding time.

```
OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.453
Model:                            OLS   Adj. R-squared:                  0.452
Method:                 Least Squares   F-statistic:                     469.1
Date:                Sat, 20 Mar 2021   Prob (F-statistic):               0.00
Time:                        21:26:21   Log-Likelihood:            -1.1294e+05
No. Observations:               27820   AIC:                         2.260e+05
Df Residuals:                   27770   BIC:                         2.264e+05
Df Model:                          49
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const        -91.0930     21.037     -4.330      0.000    -132.326     -49.860
x1           -22.7398      5.423     -4.193      0.000     -33.369     -12.111
x2           -16.5335      3.952     -4.184      0.000     -24.279      -8.788
x3           -14.2982      3.775     -3.788      0.000     -21.697      -6.900
x4           -10.7627      3.895     -2.763      0.006     -18.398      -3.128
x5           -14.3385      5.230     -2.742      0.006     -24.589      -4.088
x6           -12.4202      4.148     -2.994      0.003     -20.550      -4.290
x7           -12.0750      4.146     -2.912      0.004     -20.201      -3.949
x8            10.6337      1.491      7.132      0.000       7.711      13.556
x9            27.6150      6.151      4.490      0.000      15.559      39.671
x10           -7.0233      1.683     -4.173      0.000     -10.322      -3.724
x11           31.2204      7.887      3.958      0.000      15.761      46.680
...
x20            3.9053      1.457      2.679      0.007       1.049       6.762
x21          -36.1376      5.175     -6.983      0.000     -46.281     -25.994
x22            6.3126      1.864      3.387      0.001       2.659       9.966
x24         4.954e-13   1.57e-13      3.150      0.002    1.87e-13    8.04e-13
...
x37           -2.0629      0.748     -2.758      0.006      -3.529      -0.597
...
x49            3.8691      1.435      2.697      0.007       1.057       6.681
...
x54            3.9722      1.419      2.799      0.005       1.191       6.754
x55           11.5817      1.846      6.272      0.000       7.963      15.201
x56            6.1608      1.820      3.386      0.001       2.594       9.728
...
x60            4.0324      1.174      3.435      0.001       1.731       6.333
x61           -4.2440      1.228     -3.456      0.001      -6.651      -1.837
x62            4.5428      1.506      3.017      0.003       1.592       7.494
...
x66            0.7804      0.228      3.424      0.001       0.334       1.227
x67            2.1868      0.795      2.752      0.006       0.629       3.744
...
x71           -3.1524      0.530     -5.952      0.000      -4.191      -2.114
...
x74           -3.9115      1.170     -3.343      0.001      -6.205      -1.618
x75           -0.4310      0.067     -6.440      0.000      -0.562      -0.300
x76            0.1568      0.269      0.584      0.559      -0.370       0.683
x77           -2.7959      0.759     -3.685      0.000      -4.283      -1.309
==============================================================================
Omnibus:                    16512.291   Durbin-Watson:                   1.026
Prob(Omnibus):                  0.000   Jarque-Bera (JB):           338922.076
Skew:                           2.467   Prob(JB):                         0.00
Kurtosis:                      19.372   Cond. No.                     2.01e+16
==============================================================================
```

Figure 52: p-values for 2D Suicide Dataset

For the suicide dataset,
x20(Continent_Africa *age),x22(Continent_Africa *genaration),
x24(Continent_Asia*Continent_Europe),x37(Continent_Europe*year),
x49(Continent_North America*generation),x54(Continent_Oceania*age),
x56(Continent_Oceania*generation),x60(Continent_South America*age),

```
x61(Continent_South America*gdp_for_year ($)),x62(Continent_South America*generation),
x66(year*gdp_for_year ($)), and x67(year*generation)
features have less than 0.01 p-value
```

For x20(Continent Africa *age),x22(Continent Africa *genaration), x37(Continent Europe*year), x49(Continent North America*generation),x54(Continent Oceania*age), x56(Continent Oceania*generation), x60(Continent South America*age), x61(Continent South America*gdp for year), x62(Continent South America*generation), we see the features that are important in determining the suicides/100k for people from each continent. For people from Africa, it would be age and generation so it seems the age of the group is the important factor in determining suicide rate there. For people from Europe, it seems the year is important so there is a possibility that there was a big historical event that is tied to a high suicide rate there. For people from North America, it seems that generation is an important factor in determining suicide rate there. For people from Oceania, it seems that their age and generation plays a huge role in the suicide rate there. For people from South America, it seems age, generation, and gdp for year are the important factors in determining suicide rate there. The feature x24(Continent Asia*Continent Europe) seems to imply that being from either Asia or Europe plays an important role in determining suicide rate (obviously cannot be both because of one-hot encoding). Thus, the features seems to imply that the continent people are from and other specific features will correlate with suicide rate. In addition, it is no big surprise that age seems to be the big determining feature for people from all continents so people in general based on our answer in Question 5. The feature x66(year*gdp for year) seems to imply that the strength of the economy of a country which may have a tie to the general lifestyle and wealth conditions of the people has a correlation with suicide rate. The year could also be included to account for something such as inflation/deflation and the changing value of money over time. For feature x67(year*generation), we interpret that there may be some huge historical event such as a war, famine, or economic recession that could have resulted in larger suicide rate or it could simply be the age of the group.

Question 15.

Ideally, a higher degree polynomial could estimate quite complex features of the complex data set. However, for the first polynomial model without regularization, our test case falls into overfitting at only degree two and even ran out of memory at just degree five.

```
Polyniomial Linear Regression Results For 1 Degree:
bike_cnt: Avg. Training RMSE Error: 1655.0481337667584, Avg. Test RMSE Error: 1840.578871872201
video_utime: Avg. Training RMSE Error: 11.018591815110735, Avg. Test RMSE Error: 11.0524714871738
suicide_per_100k: Avg. Training RMSE Error: 15.467460393454868, Avg. Test RMSE Error: 15.43468480103569

Polyniomial Linear Regression Results For 2 Degree:
bike_cnt: Avg. Training RMSE Error: 955.1877686077107, Avg. Test RMSE Error: 1164.7716517045242
video_utime: Avg. Training RMSE Error: 6.378722071350994, Avg. Test RMSE Error: 7.602731191122558
suicide_per_100k: Avg. Training RMSE Error: 14.097598611762184, Avg. Test RMSE Error: 14.6756653201716

Polyniomial Linear Regression Results For 3 Degree:
bike_cnt: Avg. Training RMSE Error: 816.8396174182108, Avg. Test RMSE Error: 1072.182615216248
video_utime: Avg. Training RMSE Error: 3.967604981954318, Avg. Test RMSE Error: 168.09096553330937
suicide_per_100k: Avg. Training RMSE Error: 13.590211913192098, Avg. Test RMSE Error: 23.02436873287775

Polyniomial Linear Regression Results For 4 Degree:
bike_cnt: Avg. Training RMSE Error: 631.6932214273243, Avg. Test RMSE Error: 1274.4712164295527
video_utime: Avg. Training RMSE Error: nan, Avg. Test RMSE Error: nan
suicide_per_100k: Avg. Training RMSE Error: 13.269027466733302, Avg. Test RMSE Error: 245.42008662905224
```

Figure 53: Polynomial Regression $\alpha = 500$, Average Training and Test RMSE.

Setting up our model with L_2 regularization, we searched results with degrees one to five. We chose $L-2$/Ridge Regression model because it had the best performance as seen in section 3.2.1. For the bike data set, the degree-three case is the best test RMSE, and for the other data sets, the degree-two case is the best. The degree-four case for the three data sets showed some limitations. With strong L_2 regularization($\alpha > 500$), they are not optimized. Otherwise, with low L_2 regularization($\alpha < 500$), they suffered from overfitting. Even more than the five-degree cases are impossible to calculate because the number of parameters soars like a skyrocket. If the dimension of the data set = (D, F), degree = L,

- Total number of weights= $D[\sum_{i=0}^{L} \binom{F+i-1}{i}]$
- Total number of weights for the case degree 5 with video utime data set $= 4,524,611,520$

```
Polyniomial Linear Regression Results For 2 Degree, 0 Alpha:
bike_cnt: Avg. Training RMSE Error: 621.4801717605267, Avg. Test RMSE Error: 1124.1789269612887
video_utime: Avg. Training RMSE Error: 6.345483957776277, Avg. Test RMSE Error: 273359161587.28802
suicide_per_100k: Avg. Training RMSE Error: 14.239983313549269, Avg. Test RMSE Error: 25.02606274294652
Polyniomial Linear Regression Results For 2 Degree, 1 Alpha:
bike_cnt: Avg. Training RMSE Error: 696.435489902334, Avg. Test RMSE Error: 952.9849721965577
video_utime: Avg. Training RMSE Error: 6.247846619907449, Avg. Test RMSE Error: 11.965214191275297
suicide_per_100k: Avg. Training RMSE Error: 13.976868697390254, Avg. Test RMSE Error: 22.757708839495507
Polyniomial Linear Regression Results For 2 Degree, 2 Alpha:
bike_cnt: Avg. Training RMSE Error: 718.686916174855, Avg. Test RMSE Error: 966.3529061306923
video_utime: Avg. Training RMSE Error: 6.249983267802606, Avg. Test RMSE Error: 11.83949509625911
suicide_per_100k: Avg. Training RMSE Error: 13.97851141619051, Avg. Test RMSE Error: 22.501204432197873
Polyniomial Linear Regression Results For 2 Degree, 780 Alpha:
bike_cnt: Avg. Training RMSE Error: 938.99016951045, Avg. Test RMSE Error: 1155.848484336174
video_utime: Avg. Training RMSE Error: 6.362235154467404, Avg. Test RMSE Error: 7.59736879395202
suicide_per_100k: Avg. Training RMSE Error: 14.081584284974099, Avg. Test RMSE Error: 14.964089046353124
Polyniomial Linear Regression Results For 2 Degree, 820 Alpha:
bike_cnt: Avg. Training RMSE Error: 942.0887948899701, Avg. Test RMSE Error: 1157.5494383681696
video_utime: Avg. Training RMSE Error: 6.3653168360744035, Avg. Test RMSE Error: 7.597306650210217
suicide_per_100k: Avg. Training RMSE Error: 14.084603315716922, Avg. Test RMSE Error: 14.898323913903777
Polyniomial Linear Regression Results For 2 Degree, 860 Alpha:
bike_cnt: Avg. Training RMSE Error: 945.1103813368894, Avg. Test RMSE Error: 1159.2068527048953
video_utime: Avg. Training RMSE Error: 6.368355293594765, Avg. Test RMSE Error: 7.5978017860269755
suicide_per_100k: Avg. Training RMSE Error: 14.087571557664123, Avg. Test RMSE Error: 14.83915895420689
Polyniomial Linear Regression Results For 2 Degree, 1500 Alpha:
bike_cnt: Avg. Training RMSE Error: 987.153158552321, Avg. Test RMSE Error: 1183.7228613694617
video_utime: Avg. Training RMSE Error: 6.414092681818458, Avg. Test RMSE Error: 7.638788439240761
suicide_per_100k: Avg. Training RMSE Error: 14.129717612549115, Avg. Test RMSE Error: 14.431722612060957
Polyniomial Linear Regression Results For 2 Degree, 1800 Alpha:
bike_cnt: Avg. Training RMSE Error: 1004.4336255713454, Avg. Test RMSE Error: 1195.251479269722
video_utime: Avg. Training RMSE Error: 6.435054547966804, Avg. Test RMSE Error: 7.665182105342536
suicide_per_100k: Avg. Training RMSE Error: 14.14677013200649, Avg. Test RMSE Error: 14.409404629689078
Polyniomial Linear Regression Results For 2 Degree, 2100 Alpha:
bike_cnt: Avg. Training RMSE Error: 1020.7066943486165, Avg. Test RMSE Error: 1207.0880206249185
video_utime: Avg. Training RMSE Error: 6.45622795291339, Avg. Test RMSE Error: 7.692463718027895
suicide_per_100k: Avg. Training RMSE Error: 14.16246372429506, Avg. Test RMSE Error: 14.421434112856215
```

Figure 54: Polynomial Regression for α grid search , Average Training and Test RMSE.

Through considering all conditions, we've chosen the degree-two model with $\alpha = 1, 820$, and 1800 L_2 regularization as our best polynomial regression model for the bike, video, and suicide. **We choose this by basically performing a grid search over a range of regularization parameters(0, 1, 2, 780, 820, 860, 1500, 1800, 2100** and degrees (1, 2 , 3, 4, 5) although we only put in certain sections from the logs that demonstrate our point. For each combination of parameters, we did k-fold (k=10 for our specific case) cross-validation. This means we split our data into k folds and choose k-1 folds to be our training data and the remaining fold to test on/be the validation data set. We alternate which fold to use as the validation data set a total of k times. By doing it this way, we then have k Training and Test RMSEs which we average to find the Average Training and Test RMSE. We follow this procedure for the all the combinations of parameters (degrees and alpha) we wish to test on and are left with multiple Test RMSEs to compare. We determine which penalty parameter is "optimal" by finding the one that corresponded with the model with the lowest Avg. Test RMSE.

For these models, the best average train and test RMSEs are

- Train: 696.43, Test: 952.98 for the bike data set at $\alpha = 1$
- Train: 6.365, Test: 7.597 for the video data set at $\alpha = 820$
- Train: 14.146, Test: 14.409 for the suicide data set at $\alpha = 1800$

In summary, increasing the degree of polynomial to too high of a value can cause our model to overfit even with regularization and have us run into memory issues.

Question 16.

```
Polyniomial Linear Regression Results For 2 Degree:
video_utime: Avg. Training RMSE Error: 6.341509696718951, Avg. Test RMSE Error: 7.673023191950785
Polyniomial Linear Regression Results For 2 Degree with extra features:
video_utime: Avg. Training RMSE Error: 6.340443011542751, Avg. Test RMSE Error: 7.757408874342083
```

Figure 55: Polynomial Regression $\alpha = 1000$ for feature addition, Average Training and Test RMSE.

If we craft features with an inverse value, we might add more of the non-linearity effects to our dataset with some correlations. For example, a combination of certain features such as $\frac{FramesHeightWidth}{Framerate}$ may have a strong correlation with duration. Then, we can add more feature columns to our polynomial model. Basically, we can try to capture non-linear relationships between features and use that to improve our model's performance. However, degree two with the $\alpha = 1000$ L-2 regularized model showed only less than 1 percent boosting at training accuracy and even dropped accuracy for the testing set. We reason that this is the case because we are using a linear model to try to capture non-linear relationships between features. We may need a more robust model such as a neural network to capture these relationships more effectively.

Question 17.

A neural network without activation functions and hidden layers is identical to linear regression. While we add hidden layers and activation functions such as ReLU in the neural network model, it can be comprehensive and encompassing non-linearity than linear regression.

Over the past three decades, several "universal approximation theorems" have been proven. Possibly the first was by G. Cybenko in 1989 [1], which proved that a multilayer perceptron with sigmoid activation nodes in finite linear combinations can

> "uniformly approximate any continuous function of n real variables with support in the unit hypercube; only mild conditions are imposed on the univariate function...In particular, we show that arbitrary decision regions can be arbitrarily well approximated by continuous feedforward neural networks with only a single internal, hidden layer and any continuous sigmoidal nonlinearity."

Other mathematically comparable results preceded this one by decades and even centuries, such as the representability of continuous functions as summations or integrals of sinusoids (Fourier series and the Fourier transform), by polynomials (Weierstrass Approximation Theorem), and more generally by sufficiently numerous families of functions that "separate points" (Stone-Weierstrass Theorem). The success in engineering due to Fourier analysis and polynomial approximations (e.g. Taylor series) is well-known. One distinction that sets apart the result regarding sigmoid feedforward networks is that they so closely resemble jump discontinuities, and in particular they approximate binary switches used to encode all programming logic. In other words, both continuous functions and complex logical rules can be closely approximated by relatively small neural networks.

Question 18.

We used a systematic if somewhat coarse grid-search strategy to find an optimal set of parameters. Note that our ranges for our α, layer sizes, and network depths were not too complicated as we simply desired better/comparable performance than some of the previous linear models and we lacked computational power. That is why our neural networks are more simple in nature but they still give the desired result. We used the following values:

- $\alpha \in \{10^{-k} | k = -2, -1, 0, 1, 2, 3, 4\}$,
- Layer sizes in $\{20, 50, 100\}$, and
- Network depths in $\{2, 7, 12, 17\}$.

This already coarse grid search was further simplified to only include networks in which each layer had the same size. That is, we considered networks with layers $(20, 20)$ (depth 2, layer size 20), as well as $(50, 50, 50, 50, 50, 50, 50)$ (depth 7, layer size 50), but not $(20, 50)$, for example. We mostly sought to investigate whether extreme parameter combinations produced better or worse results.

For each combination of parameters, we performed 10-fold cross-validation on each dataset only to simplify computation. We calculated and averaged out-of-sample (test) performance to evaluate a parameterization.

Surprisingly, the optimal network size was not at the extreme of any of the parameters. The optimal value of α was 0.001, 10, and 100, and the optimal network size was $(50, 50, 50, 50, 50, 50, 50)$, $(20, 20)$,and $(100, 100)$ for each dataset. More investigation might reveal that arbitrary regularization value α would be superior for each dataset, and it's highly likely that a different combination of layer sizes would also yield improvements. As for network size, our result shows that both a too-deep and a too-wide network can yield worse out-of-sample performance due to overfitting and the issues discussed in Question 20.. The console output results of this experiment can be seen in Figure 58.

For each dataset, the model for the best average train and test RMSE is

bike : $\alpha = 0.001$, $Layers = [50, 50, 50, 50, 50, 50, 50]$,
Best Average Train RMSE= 811.99, Best Average Test RMSE= 953.21

video : $\alpha = 10$, $Layers = [20, 20]$,
Best Average Train RMSE= 3.19, Best Average Test RMSE= 4.91

suicide : $\alpha = 100$, $Layers = [100, 100]$,
Best Average Train RMSE= 13.39, Best Average Test RMSE= 14.69

While we didn't see as big of an improvement for the neural network trained on the Bike Dataset as we had hoped in terms of Average Test RMSE, we definitely saw a huge improvement for the neural networks trained on the video and suicide dataset in terms of the Average Test RMSE in comparison to all of the previous Linear Models (Linear Regression/Lasso Regression/Ridge Regression/Polynomial Regression). With further tuning, we are confident we can find even better models that deliver stronger results in terms of Average Test RMSE.

```
alpha=0.1, networksize=(20, 20), avg_train_score=2119.495261210238, avg_test_score=2262.54187545605
alpha=0.1, networksize=(50, 50), avg_train_score=1965.397316438295, avg_test_score=2201.107657712272
alpha=0.1, networksize=(100, 100), avg_train_score=1712.7605642337833, avg_test_score=2061.3534407231386
alpha=0.1, networksize=(20, 20, 20, 20, 20, 20, 20), avg_train_score=1002.8747630741781, avg_test_score=1193.7710255317063
alpha=0.1, networksize=(50, 50, 50, 50, 50, 50, 50), avg_train_score=811.1666265754832, avg_test_score=972.2901529263631
alpha=0.1, networksize=(100, 100, 100, 100, 100, 100, 100), avg_train_score=790.9585668531872, avg_test_score=960.3553341670556
alpha=0.1, networksize=(20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20), avg_train_score=868.0031908531163, avg_test_score=993.533337392239
alpha=0.1, networksize=(50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50), avg_train_score=782.5782166976599, avg_test_score=999.1872725798596
alpha=0.1, networksize=(100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100), avg_train_score=744.1780336587098, avg_test_score=1034.719785220842
alpha=0.1, networksize=(20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20), avg_train_score=847.2181329308114, avg_test_score=1013.0867131757957
alpha=0.1, networksize=(50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50), avg_train_score=822.5247891554116, avg_test_score=1029.0976779347393
alpha=0.1, networksize=(100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100), avg_train_score=774.3828410930852,
    avg_test_score=1005.9403288534689
alpha=0.01, networksize=(20, 20), avg_train_score=2119.4743257220916, avg_test_score=2262.55261852144
alpha=0.01, networksize=(50, 50), avg_train_score=1962.2691248113344, avg_test_score=2198.8236069741342
alpha=0.01, networksize=(100, 100), avg_train_score=1716.2249339617256, avg_test_score=2066.472768573612
alpha=0.01, networksize=(20, 20, 20, 20, 20, 20, 20), avg_train_score=1006.0484977635562, avg_test_score=1197.3678090212388
alpha=0.01, networksize=(50, 50, 50, 50, 50, 50, 50), avg_train_score=809.9473953072666, avg_test_score=971.2358463966192
alpha=0.01, networksize=(100, 100, 100, 100, 100, 100, 100), avg_train_score=794.2152308922186, avg_test_score=964.8607155687789
alpha=0.01, networksize=(20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20), avg_train_score=873.618756917946, avg_test_score=1000.7860651523872
alpha=0.01, networksize=(50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50), avg_train_score=779.2575990139752, avg_test_score=974.5492123561518
alpha=0.01, networksize=(100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100), avg_train_score=788.448729957101, avg_test_score=1043.3973038145864
alpha=0.01, networksize=(20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20), avg_train_score=844.4840809742234, avg_test_score=991.4763307970218
alpha=0.01, networksize=(50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50), avg_train_score=819.818386015881, avg_test_score=1069.536420080876
alpha=0.01, networksize=(100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100), avg_train_score=788.2004252746077,
    avg_test_score=1040.4577936614687
alpha=0.001, networksize=(20, 20), avg_train_score=2119.461291531958, avg_test_score=2262.5629441552874
alpha=0.001, networksize=(50, 50), avg_train_score=1963.6308875084665, avg_test_score=2201.225547905477
alpha=0.001, networksize=(100, 100), avg_train_score=1718.4725464637982, avg_test_score=2070.482502789419
alpha=0.001, networksize=(20, 20, 20, 20, 20, 20, 20), avg_train_score=1039.3886848313434, avg_test_score=1278.8949534304147
alpha=0.001, networksize=(50, 50, 50, 50, 50, 50, 50), avg_train_score=811.9878940704411, avg_test_score=953.2092893696363
alpha=0.001, networksize=(100, 100, 100, 100, 100, 100, 100), avg_train_score=786.1190417715377, avg_test_score=964.017867686739
alpha=0.001, networksize=(20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20), avg_train_score=873.8669131213036, avg_test_score=1022.6375886154246
alpha=0.001, networksize=(50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50), avg_train_score=775.2503534289829, avg_test_score=993.0497503592038
alpha=0.001, networksize=(100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100), avg_train_score=796.697036546245, avg_test_score=995.8414999940089
alpha=0.001, networksize=(20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20), avg_train_score=849.8444155204029, avg_test_score=1015.0759850256163
alpha=0.001, networksize=(50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50), avg_train_score=819.9016887660687, avg_test_score=1043.9624086046024
alpha=0.001, networksize=(100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100), avg_train_score=781.91346226134,
    avg_test_score=1038.7791857087516
alpha=0.0001, networksize=(20, 20), avg_train_score=2119.5792634126447, avg_test_score=2260.9865724685205
alpha=0.0001, networksize=(50, 50), avg_train_score=1962.1487440118988, avg_test_score=2198.095679430303
alpha=0.0001, networksize=(100, 100), avg_train_score=1731.4686949593365, avg_test_score=2076.486071117267
alpha=0.0001, networksize=(20, 20, 20, 20, 20, 20, 20), avg_train_score=1100.1027714911509, avg_test_score=1417.9072585298068
alpha=0.0001, networksize=(50, 50, 50, 50, 50, 50, 50), avg_train_score=819.5275724490378, avg_test_score=969.3217793221206
alpha=0.0001, networksize=(100, 100, 100, 100, 100, 100, 100), avg_train_score=788.6802875787406, avg_test_score=971.616535195916
alpha=0.0001, networksize=(20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20), avg_train_score=888.4567196213865, avg_test_score=1032.7234866175088
alpha=0.0001, networksize=(50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50), avg_train_score=781.8110450450401, avg_test_score=993.5966807561036
alpha=0.0001, networksize=(100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100), avg_train_score=770.0646258798006, avg_test_score=1080.016376353978
alpha=0.0001, networksize=(20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20), avg_train_score=840.7860186664511,
    avg_test_score=1015.8484202219695
alpha=0.0001, networksize=(50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50), avg_train_score=829.0026221093433, avg_test_score=1122.032243510678
alpha=0.0001, networksize=(100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100), avg_train_score=797.7597061002961,
    avg_test_score=1003.163240520006
Best Average RMSE:  953.21
Best network parameters are:
alpha=0.001
Layers=[50, 50, 50, 50, 50, 50, 50]
```

Figure 56: Performance on out-of-sample (test) data over 4-fold cross-validation of neural networks with various parameter combinations applied to the bike rental dataset.

```
alpha=10, networksize=(20, 20), avg_train_score=3.1851149693003333, avg_test_score=4.915124711242685
alpha=10, networksize=(50, 50), avg_train_score=2.643468108478084, avg_test_score=5.692156205231669
alpha=10, networksize=(100, 100), avg_train_score=2.5604819620085624, avg_test_score=5.174019596840244
alpha=10, networksize=(20, 20, 20, 20, 20, 20, 20), avg_train_score=1.8808539813100353, avg_test_score=5.260893841633832
alpha=10, networksize=(50, 50, 50, 50, 50, 50, 50), avg_train_score=1.5840797030109937, avg_test_score=5.293394100179527
alpha=10, networksize=(100, 100, 100, 100, 100, 100, 100), avg_train_score=1.554334991323489, avg_test_score=5.448109821472685
alpha=10, networksize=(20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20), avg_train_score=1.9170621872388338, avg_test_score=5.748293484929499
alpha=10, networksize=(50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50), avg_train_score=1.3793852070521118, avg_test_score=5.503493402951124
alpha=10, networksize=(100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100), avg_train_score=1.2690822346511501, avg_test_score=5.012954829942867
alpha=10, networksize=(20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20), avg_train_score=1.8179708602
35265, avg_test_score=5.84914122960789
alpha=10, networksize=(50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50), avg_train_score=1.3747686177043574, avg_test_score=5.319102585711698
alpha=10, networksize=(100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100), avg_train_score=1.235778275173647,
    avg_test_score=5.150127235874928
alpha=1, networksize=(20, 20), avg_train_score=2.2741573628726472, avg_test_score=12.752411394506101
alpha=1, networksize=(50, 50), avg_train_score=1.5751152494825937, avg_test_score=22.258071624013386
alpha=1, networksize=(100, 100), avg_train_score=1.181476487975329, avg_test_score=9.779548752055273
alpha=1, networksize=(20, 20, 20, 20, 20, 20, 20), avg_train_score=1.772112667215917, avg_test_score=5.363693922256511
alpha=1, networksize=(50, 50, 50, 50, 50, 50, 50), avg_train_score=1.4050847372278845, avg_test_score=5.914674209439344
alpha=1, networksize=(100, 100, 100, 100, 100, 100, 100), avg_train_score=1.2479447371339187, avg_test_score=5.161487682
655982
alpha=1, networksize=(20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20), avg_train_score=1.9170621872388338, avg_test_score=5.770702243405449
alpha=1, networksize=(50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50), avg_train_score=1.3793852070521118, avg_test_score=5.523363960865723
alpha=1, networksize=(100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100), avg_train_score=1.2690899152511501, avg_test_score=5.032142847713091
alpha=1, networksize=(20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20), avg_train_score=1.865845689635265, avg_test_score=5.83453513180789
alpha=1, networksize=(50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50), avg_train_score=1.3747686177043574, avg_test_score=5.343937905411698
alpha=1, networksize=(100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100), avg_train_score=1.235778275173647,
    avg_test_score=5.147205795074553
alpha=0.1, networksize=(20, 20), avg_train_score=2.3090819816572874, avg_test_score=19.73111296072012
alpha=0.1, networksize=(50, 50), avg_train_score=1.5798119056074698, avg_test_score=34.16665214043647
alpha=0.1, networksize=(100, 100), avg_train_score=1.195753646952724, avg_test_score=15.399018199641354
alpha=0.1, networksize=(20, 20, 20, 20, 20, 20, 20), avg_train_score=1.6602965187282006, avg_test_score=5.410013823613378
alpha=0.1, networksize=(50, 50, 50, 50, 50, 50, 50), avg_train_score=1.259858054029303, avg_test_score=5.820229957690775
alpha=0.1, networksize=(100, 100, 100, 100, 100, 100, 100), avg_train_score=1.0959538868812206, avg_test_score=5.277487793217663
alpha=0.1, networksize=(20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20), avg_train_score=1.9388838559175317, avg_test_score=5.80347810818336
alpha=0.1, networksize=(50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50), avg_train_score=1.358208370503009, avg_test_score=5.434963506195887
alpha=0.1, networksize=(100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100), avg_train_score=1.0359817018455784, avg_test_score=5.292256111858478
alpha=0.1, networksize=(20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20), avg_train_score=1.7185099246183255, avg_test_score=5.788024746518309
alpha=0.1, networksize=(50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50), avg_train_score=1.3491611193835058, avg_test_score=5.444756087552748
alpha=0.1, networksize=(100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100), avg_train_score=1.4854055355783649,
    avg_test_score=4.966026273609041
alpha=0.01, networksize=(20, 20), avg_train_score=2.2480786040823246, avg_test_score=19.908948518708154
alpha=0.01, networksize=(50, 50), avg_train_score=1.5448164413488827, avg_test_score=35.6364257230326
alpha=0.01, networksize=(100, 100), avg_train_score=1.1006822245063879, avg_test_score=14.916204288560747
alpha=0.01, networksize=(20, 20, 20, 20, 20, 20, 20), avg_train_score=1.570542948899629, avg_test_score=5.4444697837546245
alpha=0.01, networksize=(50, 50, 50, 50, 50, 50, 50), avg_train_score=1.4162164439632314, avg_test_score=6.025779107185977
alpha=0.01, networksize=(100, 100, 100, 100, 100, 100, 100), avg_train_score=1.223322644945427, avg_test_score=5.178545778416406
alpha=0.01, networksize=(20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20), avg_train_score=1.8840190232123646, avg_test_score=5.587716204533368
alpha=0.01, networksize=(50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50), avg_train_score=1.1813338462381202, avg_test_score=5.413842180764901
alpha=0.01, networksize=(100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100), avg_train_score=1.0524056639485797, avg_test_score=5.003968346735365
alpha=0.01, networksize=(20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20), avg_train_score=1.5932294443989439, avg_test_score=6.297182638993736
alpha=0.01, networksize=(50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50), avg_train_score=1.3905972793703574, avg_test_score=5.682061124439248
alpha=0.01, networksize=(100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100), avg_train_score=1.255540906059037,
    avg_test_score=5.145758396703916
Best Average RMSE:  4.92
Best network parameters are:
alpha=10
Layers=[20, 20]
```

Figure 57: Performance on out-of-sample (test) data over 4-fold cross-validation of neural networks with various parameter combinations applied to the video transcoding dataset.

```
alpha=100, networksize=(20, 20), avg_train_score=13.61145246714245, avg_test_score=14.759198725720584
alpha=100, networksize=(50, 50), avg_train_score=13.42094996572151, avg_test_score=14.771994659332938
alpha=100, networksize=(100, 100), avg_train_score=13.386573696697923, avg_test_score=14.691921623579146
alpha=100, networksize=(20, 20, 20, 20, 20, 20, 20), avg_train_score=12.284375471141402, avg_test_score=15.952943649984157
alpha=100, networksize=(50, 50, 50, 50, 50, 50, 50), avg_train_score=11.44534002317468, avg_test_score=15.397197194913797
alpha=100, networksize=(100, 100, 100, 100, 100, 100, 100), avg_train_score=11.115232242701824, avg_test_score=15.749657123930561
alpha=100, networksize=(20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20), avg_train_score=12.182998925118804, avg_test_score=16.15691858535229
alpha=100, networksize=(50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50), avg_train_score=10.884846855894624, avg_test_score=15.877134341522286
alpha=100, networksize=(100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100), avg_train_score=10.698749733510336, avg_test_score=15.72820910982799
alpha=100, networksize=(20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20), avg_train_score=12.122536073950114, avg_test_score=15.815713068505776
alpha=100, networksize=(50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50), avg_train_score=11.139329605975998, avg_test_score=15.894023992564371
alpha=100, networksize=(100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100), avg_train_score=10.71739848329422,
    avg_test_score=16.175815524625907
alpha=10, networksize=(20, 20), avg_train_score=12.495048825176529, avg_test_score=15.471755955782825
alpha=10, networksize=(50, 50), avg_train_score=11.822208051117668, avg_test_score=15.211783066988694
alpha=10, networksize=(100, 100), avg_train_score=11.60278520715175, avg_test_score=15.351722751242722
alpha=10, networksize=(20, 20, 20, 20, 20, 20, 20), avg_train_score=11.519744111596173, avg_test_score=16.17141841607235
alpha=10, networksize=(50, 50, 50, 50, 50, 50, 50), avg_train_score=9.50439008703389, avg_test_score=16.815026657839812
alpha=10, networksize=(100, 100, 100, 100, 100, 100, 100), avg_train_score=8.783317039179314, avg_test_score=16.429126166607862
alpha=10, networksize=(20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20), avg_train_score=10.97100283291672, avg_test_score=15.84482922231078
alpha=10, networksize=(50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50), avg_train_score=9.871906376662507, avg_test_score=16.62093836064431
alpha=10, networksize=(100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100), avg_train_score=9.025465675130755, avg_test_score=16.783424198818178
alpha=10, networksize=(20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20), avg_train_score=10.894867440326749, avg_test_score=16.25163998529176
alpha=10, networksize=(50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50), avg_train_score=9.47168989051456, avg_test_score=16.274041633513107
alpha=10, networksize=(100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100), avg_train_score=9.295368569475542,
    avg_test_score=16.68704809212925
alpha=1, networksize=(20, 20), avg_train_score=12.239932573528673, avg_test_score=15.899388634569467
alpha=1, networksize=(50, 50), avg_train_score=11.360944513597083, avg_test_score=16.926940150583437
alpha=1, networksize=(100, 100), avg_train_score=10.995410427611773, avg_test_score=16.019950245741963
alpha=1, networksize=(20, 20, 20, 20, 20, 20, 20), avg_train_score=10.87387038106855, avg_test_score=16.37767855133781
alpha=1, networksize=(50, 50, 50, 50, 50, 50, 50), avg_train_score=8.939442029648934, avg_test_score=17.037735273022722
alpha=1, networksize=(100, 100, 100, 100, 100, 100, 100), avg_train_score=8.45804498012675, avg_test_score=16.671156943398433
alpha=1, networksize=(20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20), avg_train_score=10.504239509348883, avg_test_score=16.202514302123703
alpha=1, networksize=(50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50), avg_train_score=8.931108553432228, avg_test_score=16.67832738627633
alpha=1, networksize=(100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100), avg_train_score=8.84622244198448, avg_test_score=16.348427785295613
alpha=1, networksize=(20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20), avg_train_score=10.731329590561625, avg_test_score=16.73963738732615
alpha=1, networksize=(50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50), avg_train_score=9.164929724269328, avg_test_score=16.47744450629276
alpha=1, networksize=(100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100), avg_train_score=9.626529801590515,
    avg_test_score=17.261773644134976
Best Average RMSE:  14.69
Best network parameters are:
alpha=100
Layers=[100, 100]
```

Figure 58: Performance on out-of-sample (test) data over 4-fold cross-validation of neural networks with various parameter combinations applied to the suicide dataset.

Question 19.

We used a ReLU activation function for the network model, which is so common and usually applied for a neural network. The ReLU function ($f(x) = \max(0, x)$) has many benefits. Unlike the other activations such as Tanh, Signoid needs exponential and complex calculation, the ReLU is fast and simple at training because either the gradient is 1 or 0. There are two more benefits. By using the ReLU we not only prevent gradient vanishing problems but we can apply nonlinearity easiest way.

Question 20.

If we set up the hidden layer for the neural network model too deep, two problems could occur, vanishing gradient and degradation. A reason for the vanishing gradient is a chain rule at the backpropagation. When we calculate a gradient of the deep network, we multiply lots of gradients for backpropagation, and eventually, the gradient becomes too small. Then, the network learns nothing. Fortunately, it is addressed by using the ReLU activation function. The emerged problem, degradation is another phenomenon that a deeper layer has a bigger train and test error. This problem is also successfully addressed by the residual layer in ResNet.

In addition, more layers entail more parameters overall, which can lead to both identifiability problems and overfitting. With sufficient data, identifiability is not an issue, but overfitting can lead to poor out-of-sample performance, even with appropriate regularization.

Question 21.

We fine-tuned our model performing a grid search over a range of the hyperparameters maximum number of features, number of trees, and depth of each tree and evaluated the performance of each new model by using 10-fold cross validation and examining the Average Test RMSE. The range of the maximum number of features is (0.2, 0.4, 0.6, 0.8, 'auto', 'sqrt', 'log2'). The floats represent a fraction and $round(maxfeatures * nfeatures)$ features are considered each split. 'auto' means $maxfeatures = sqrt(nfeatures)$. 'sqrt' means $maxfeatures = log2(nfeatures)$. 'log2' means $maxfeatures = log2(nfeatures)$. For the number of trees we chose a range of (10, 50, 100, 150, 200). For the depth of each tree we chose a range of (1, 2, 3, 4, 5, 7). Our range of the number of trees and the depth of each tree had to be limited because they increased the computation time too much as we increased it. The results of our grid search are as follows

```
Bike:
Max Number Features: 0.8, Number of Trees: 100, Depth of Each Tree: 7,
Avg. Training RMSE: 426.7680411468512, Avg. Test RMSE: 836.4700790958702, OOB error: 0.126, $R^2: 0.9549$

Video:
Max Number Features: 0.8, Number of Trees: 200, Depth of Each Tree: 7,
Avg. Training RMSE: 4.306154785881811, Avg. Test RMSE: 5.126432683814302, OOB error: 0.0959, $R^2: 0.913$

Suicide:
Max Number Features: 0.4, Number of Trees: 50, Depth of Each Tree: 7,
Avg. Training RMSE: 11.864126899072724, Avg. Test RMSE: 13.835570288001104, OOB error: 0.404, $R^2: 0.62$
```

Figure 59: Random Forest Hyperparameters, Average Training, Average Test RMSE, OOB error for the best model.

A quick note is that the Random Forest model seemed to struggle with the suicide dataset. However, the Average Test RMSE is comparable to the prior models such as Linear Regression, Lasso Regression, and Ridge Regression and it works extremely well for the bike and video datasets.

The maximum number of features has the following effect on performance. Lower values of maximum features lead to more different and less correlated trees. This gives better stability when we aggregate. In addition, random forests using lower values of maximum features tend to make better use of features that do not have a immense effect on the target and would normally be overshadowed by features with a very strong effect/are highly correlated which might have been chosen for splitting if the maximum number of features was higher. This is similar to a regularization effect as it prevents features with a strong effect/highly from dominating and overshadowing other features. The tradeoff of using a smaller number of maximum features is that it could lead to trees that generally perform worse because we are purposely using "worse" features meaning that the trees could be using features that have little relevance to building a good model. This would cause our accuracy to suffer. As a result, there is a tradeoff for the number of maximum features in terms of stability and accuracy with fewer maximum features helping stability and more maximum features helping accuracy. There is no “right” number of maximum of features to use and we need to tune it using cross-validation.

In general, having a greater number of trees is always more beneficial and increases the Average Test RMSE. Increasing the number of trees also has a regularization effect. Our reasoning is that a random forest is essentially an aggregation of various decision trees that all contribute to the model. Since there are various different models that work together, a Random Forest is more flexible and is more capable of handing new, unseen data. Increasing the number of the these trees only serves to increase the flexibility and generality of the model which means the number of trees has a regularization effect. However, we notice that increasing the number of trees directly contributes to increased training time so there is a trade off there.

The depth of each tree is a parameter that affects the performance in two ways. If the depth of each tree is too small, then the complexity of each tree is too small and it will cause poor Average Test RMSE. However, if we increase the depth of each tree by too much, each tree will become very specialized and have poor generalization ability. Thus, the depth of each tree has a regularization effect and we need to tune this hyperparameter so that we can balance out the model complexity with generalization ability.

Question 22.

Random forests perform well for multiple reasons. Among the various reasons include that random forests are invariant to feature scaling, can handle data sets with high dimensionality, has automated feature selection by the nature of decision trees and by design incorporates regularization. As mentioned before, the fact that Random Forests use multiple decision trees gives it built in regularization to prevent overfitting and gives it good generalization ability. In addition, we can further prevent the random forest from overfitting through controlling the depth of each individual tree and the maximum number of features for each decision tree. We can flexibly control the complexity of the model by increasing the depth of each individual tree or increase generalization ability by decreasing the depth. Using fewer maximum features also encourages the Random Forest model to use features that have not as great of an effect on the target. We can increase the maximum number of features if the Average Test RMSE is too low which would signify that we are using too many “bad” features. In addition, Random Forests which are composed of decision tree is a non-linear model which means it can help capture non-linear effects and relationships between the features that are important for generating a good target value.

Question 23.

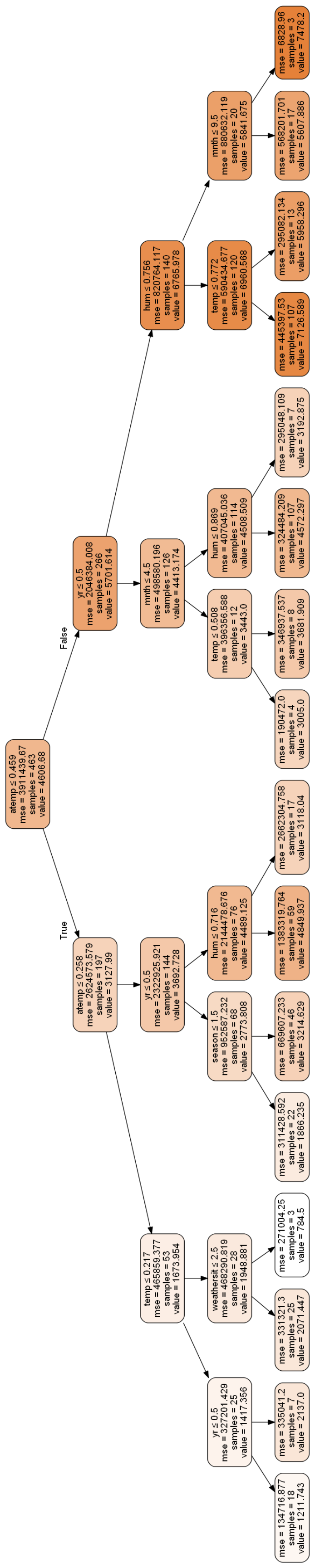


Figure 60: Random decision tree from Random Forest trained on the bike dataset.

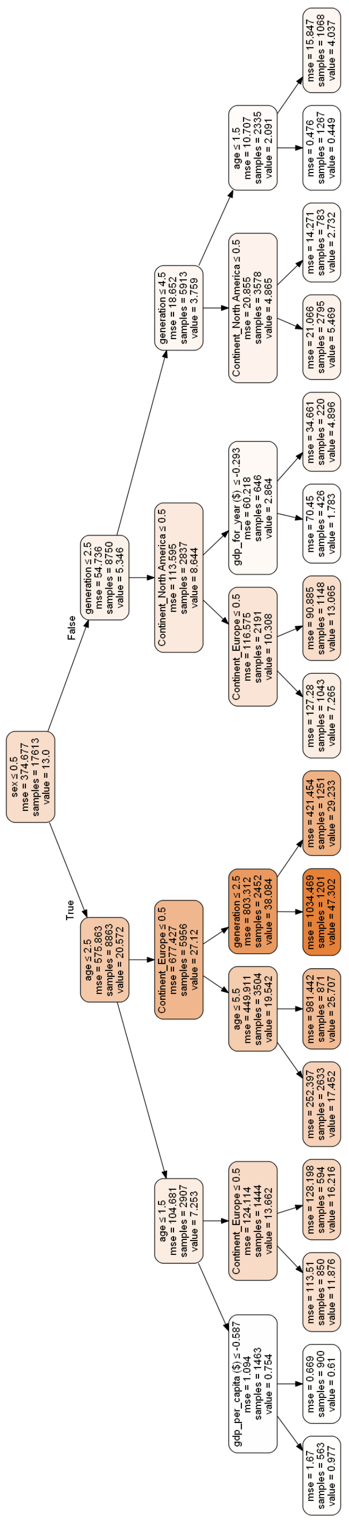


Figure 61: Random decision tree from Random Forest trained on the suicide dataset.

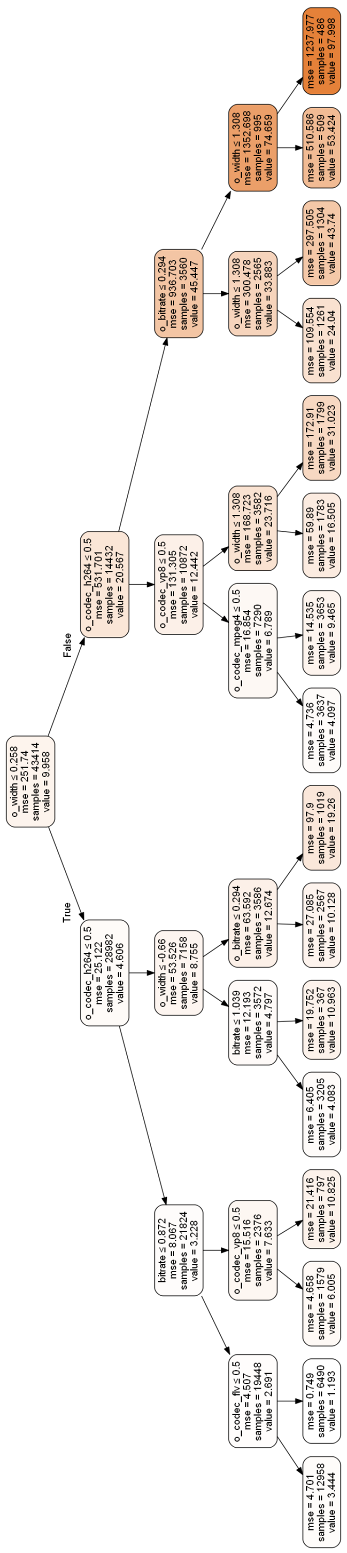


Figure 62: Random decision tree from Random Forest trained on the video dataset.

Because these plots are not clear in the report, we also generated logs for them shown as followed.

```
Decision Tree from Random Forest Trained on Bike Dataset
|--- atemp <= 0.46
|   |--- atemp <= 0.26
|   |   |--- temp <= 0.22
|   |   |   |--- yr <= 0.50
|   |   |   |   |--- value: [1211.74]
|   |   |   |--- yr >  0.50
|   |   |   |   |--- value: [2137.00]
|   |   |--- temp >  0.22
|   |   |   |--- weathersit <= 2.50
|   |   |   |   |--- value: [2071.45]
|   |   |   |--- weathersit >  2.50
|   |   |   |   |--- value: [784.50]
|   |--- atemp >  0.26
|   |   |--- yr <= 0.50
|   |   |   |--- season <= 1.50
|   |   |   |   |--- value: [1866.24]
|   |   |   |--- season >  1.50
|   |   |   |   |--- value: [3214.63]
|   |   |--- yr >  0.50
|   |   |   |--- hum <= 0.72
|   |   |   |   |--- value: [4849.94]
|   |   |   |--- hum >  0.72
|   |   |   |   |--- value: [3118.04]
|--- atemp >  0.46
|   |--- yr <= 0.50
|   |   |--- mnth <= 4.50
|   |   |   |--- temp <= 0.51
|   |   |   |   |--- value: [3005.00]
|   |   |   |--- temp >  0.51
|   |   |   |   |--- value: [3681.91]
|   |   |--- mnth >  4.50
|   |   |   |--- hum <= 0.87
|   |   |   |   |--- value: [4572.30]
|   |   |   |--- hum >  0.87
|   |   |   |   |--- value: [3192.88]
|   |--- yr >  0.50
|   |   |--- hum <= 0.76
|   |   |   |--- temp <= 0.77
|   |   |   |   |--- value: [7126.59]
|   |   |   |--- temp >  0.77
|   |   |   |   |--- value: [5958.30]
|   |   |--- hum >  0.76
|   |   |   |--- mnth <= 9.50
|   |   |   |   |--- value: [5607.89]
|   |   |   |--- mnth >  9.50
|   |   |   |   |--- value: [7478.20]
```

Figure 63: Logs of the Plot of a Decision Tree with a Maximum Depth 4 taken from a Random Forest Model trained on the bike dataset.

The feature that is selected for branching at the root node is atemp. We can infer that the features that are important are used at each level of the decision tree to split on. Basically, the important features are used in the decision tree to split on at each level. The important features based on the decision tree were atemp, yr, weathersit, season, and mnth and they were all found to be significant features in part 3.2.1. Oddly hum was chosen to split on in the decision tree but was not found to be significant in part 3.2.1. Thus, there is somewhat of a match but overall the important features match.

```
Decision Tree from Random Forest Trained on Suicide Dataset
|--- sex <= 0.50
|   |--- age <= 2.50
|   |   |--- age <= 1.50
|   |   |   |--- gdp_per_capita ($) <= -0.59
|   |   |   |   |--- value: [0.98]
|   |   |   |--- gdp_per_capita ($) >  -0.59
|   |   |   |   |--- value: [0.61]
|   |   |--- age >  1.50
|   |   |   |--- Continent_Europe <= 0.50
|   |   |   |   |--- value: [11.88]
|   |   |   |--- Continent_Europe >  0.50
|   |   |   |   |--- value: [16.22]
|   |--- age >  2.50
|   |   |--- Continent_Europe <= 0.50
|   |   |   |--- age <= 5.50
|   |   |   |   |--- value: [17.45]
|   |   |   |--- age >  5.50
|   |   |   |   |--- value: [25.71]
|   |   |--- Continent_Europe >  0.50
|   |   |   |--- generation <= 2.50
|   |   |   |   |--- value: [47.30]
|   |   |   |--- generation >  2.50
|   |   |   |   |--- value: [29.23]
|--- sex >  0.50
|   |--- generation <= 2.50
|   |   |--- Continent_North America <= 0.50
|   |   |   |--- Continent_Europe <= 0.50
|   |   |   |   |--- value: [7.27]
|   |   |   |--- Continent_Europe >  0.50
|   |   |   |   |--- value: [13.07]
|   |   |--- Continent_North America >  0.50
|   |   |   |---  gdp_for_year ($)  <= -0.29
|   |   |   |   |--- value: [1.78]
|   |   |   |---  gdp_for_year ($)  >  -0.29
|   |   |   |   |--- value: [4.90]
|   |--- generation >  2.50
|   |   |--- generation <= 4.50
|   |   |   |--- Continent_North America <= 0.50
|   |   |   |   |--- value: [5.47]
|   |   |   |--- Continent_North America >  0.50
|   |   |   |   |--- value: [2.73]
|   |   |--- generation >  4.50
|   |   |   |--- age <= 1.50
|   |   |   |   |--- value: [0.45]
|   |   |   |--- age >  1.50
|   |   |   |   |--- value: [4.04]
```

Figure 64: Logs of the Plot of a Decision Tree with a Maximum Depth 4 taken from a Random Forest Model trained on the suicide dataset.

The feature selected for branching at the root node is sex. Basically, the important features are used in the decision tree to split on at each level. The important features based on the decision tree here are sex, age, gdp per capita, Continent Europe, generation, and Continent North America. All of these features were found to be important in part 3.2.1 so they match what we got in part 3.2.1.

```
Decision Tree from Random Forest Trained on Video Dataset
|--- o_width <= 0.26
|   |--- o_codec_h264 <= 0.50
|   |   |--- bitrate <= 0.87
|   |   |   |--- o_codec_flv <= 0.50
|   |   |   |   |--- value: [3.44]
|   |   |   |--- o_codec_flv >  0.50
|   |   |   |   |--- value: [1.19]
|   |   |--- bitrate >  0.87
|   |   |   |--- o_codec_vp8 <= 0.50
|   |   |   |   |--- value: [6.01]
|   |   |   |--- o_codec_vp8 >  0.50
|   |   |   |   |--- value: [10.83]
|   |--- o_codec_h264 >  0.50
|   |   |--- o_width <= -0.66
|   |   |   |--- bitrate <= 1.04
|   |   |   |   |--- value: [4.08]
|   |   |   |--- bitrate >  1.04
|   |   |   |   |--- value: [10.96]
|   |   |--- o_width >  -0.66
|   |   |   |--- o_bitrate <= 0.29
|   |   |   |   |--- value: [10.13]
|   |   |   |--- o_bitrate >  0.29
|   |   |   |   |--- value: [19.26]
|--- o_width >  0.26
|   |--- o_codec_h264 <= 0.50
|   |   |--- o_codec_vp8 <= 0.50
|   |   |   |--- o_codec_mpeg4 <= 0.50
|   |   |   |   |--- value: [4.10]
|   |   |   |--- o_codec_mpeg4 >  0.50
|   |   |   |   |--- value: [9.47]
|   |   |--- o_codec_vp8 >  0.50
|   |   |   |--- o_width <= 1.31
|   |   |   |   |--- value: [16.50]
|   |   |   |--- o_width >  1.31
|   |   |   |   |--- value: [31.02]
|   |--- o_codec_h264 >  0.50
|   |   |--- o_bitrate <= 0.29
|   |   |   |--- o_width <= 1.31
|   |   |   |   |--- value: [24.04]
|   |   |   |--- o_width >  1.31
|   |   |   |   |--- value: [43.74]
|   |   |--- o_bitrate >  0.29
|   |   |   |--- o_width <= 1.31
|   |   |   |   |--- value: [53.42]
|   |   |   |--- o_width >  1.31
|   |   |   |   |--- value: [98.00]
```

Figure 65: Logs of the Plot of a Decision Tree with a Maximum Depth 4 taken from a Random Forest Model trained on the video dataset.

The featured selected for branching at the root node is o width. Basically, the important features are used in the decision tree to split on at each level. The important features based on the decision tree here are o width, o codec h264, bitrate, o codec flv, o codec vp8, o width, o code mpeg4, and o bitrate. All of these features were found to be important in part 3.2.1 so they match what we got in part 3.2.1.

Question 24.

We applied lightGBM and Catboost on bike sharing dataset with number of total rental bikes as the target. Bayesian optimization was used to tune the parameters. For lightGBM, the parameters we choose are number of leaves ranging from 2 to 50, maximum depth ranging from 1 to 10, learning rate ranging from 0 to 1, number of estimators ranging from 1 to 100 and regularization alpha ranging from 0 to 1. For catboost, the parameters we choose are depth ranging from 2 to 10, learning rate ranging from 0 to 1, number of estimators ranging from 1 to 100, l2 leaf regularization ranging from 0 to 1 and bagging temperature ranging from 1 t0 100.

Question 25.

The best parameters and corresponding RMSE found for these algorithms are shown in figure 14 and figure 15. The Catboost model has Avg. Test RMSE of 672.7, which is lower than that of lightGBM model (732.3)

```
Avg. Test RMSE Error: 734.2197756935816
Avg. Train RMSE Error:692.3831871920622
best_parameter: OrderedDict([('learning_rate', 0.24431439408569583), ('max_depth', 10), ('n_estimators', 100),
    ('num_leaves', 3), ('reg_alpha', 0.11145223325530218)])
```

Figure 66: Bayesian optimization result for lightBGM

```
Avg. Test RMSE Error: 681.9402044807986
Avg. Train RMSE Error:548.0529980411208
best_parameter: OrderedDict([('bagging_temperature', 0.01104190833817071), ('depth', 2), ('l2_leaf_reg',
    0.34252890238815836), ('learning_rate', 0.17716718143675902), ('n_estimators', 84)])
```

Figure 67: Bayesian optimization result for catboost

Question 26.

Figure 68-72 show the RMSE as function number of estimators, number of leaves, maximum depth, learning rate, regularization alpha for light light-gbm model. Figure 73-77 show the RMSE as function number of estimators, depth, bagging temperature, l2 regularization and learning rate for Catboost model. As we increase the number of estimators, the RMSE of the test set and training set for both light-gbm and Catboost models decrease. It shows that increase the number of estimators can improve the model performance as it is the main parameter to control model complexity. In addition, as we increase the number of estimators we see the gap between test and training accuracy increases showing that as we increase the model complexity, we have to pay for it in generalization ability (the test RMSE overall improves however with increased number of estimators). For learning rate and maximum depth for light-gbm and Catboost, the RMSE of test set first decrease then increase, while the RMSE of training set keeps becoming smaller. It shows learning rate and maximum depth have strong effect on regularization. The model becomes overfitting when these parameters are too large and underfitting when they are too small. We also find the RMSE of the model doesn't change with varying number of leaves and regularization alpha. It means these parameters doesn't affect the model performance or regularization. Figure 73-77 show the RMSE as function number of estimators, depth, bagging temperature, l2 regularization and learning rate for Catboost model. Following similar analysis, we find number of estimators is the key parameter to control the model performance. Depth,l2 regularization and learning rate have strong effect on regularization. Bagging temperature has no effect on model performance or regularization. At last, we show the effect of learning rate on the fitting efficiency using figure3 36. We can see large learning rate will reduce the time to fit the model, thus increasing the learning efficiency.

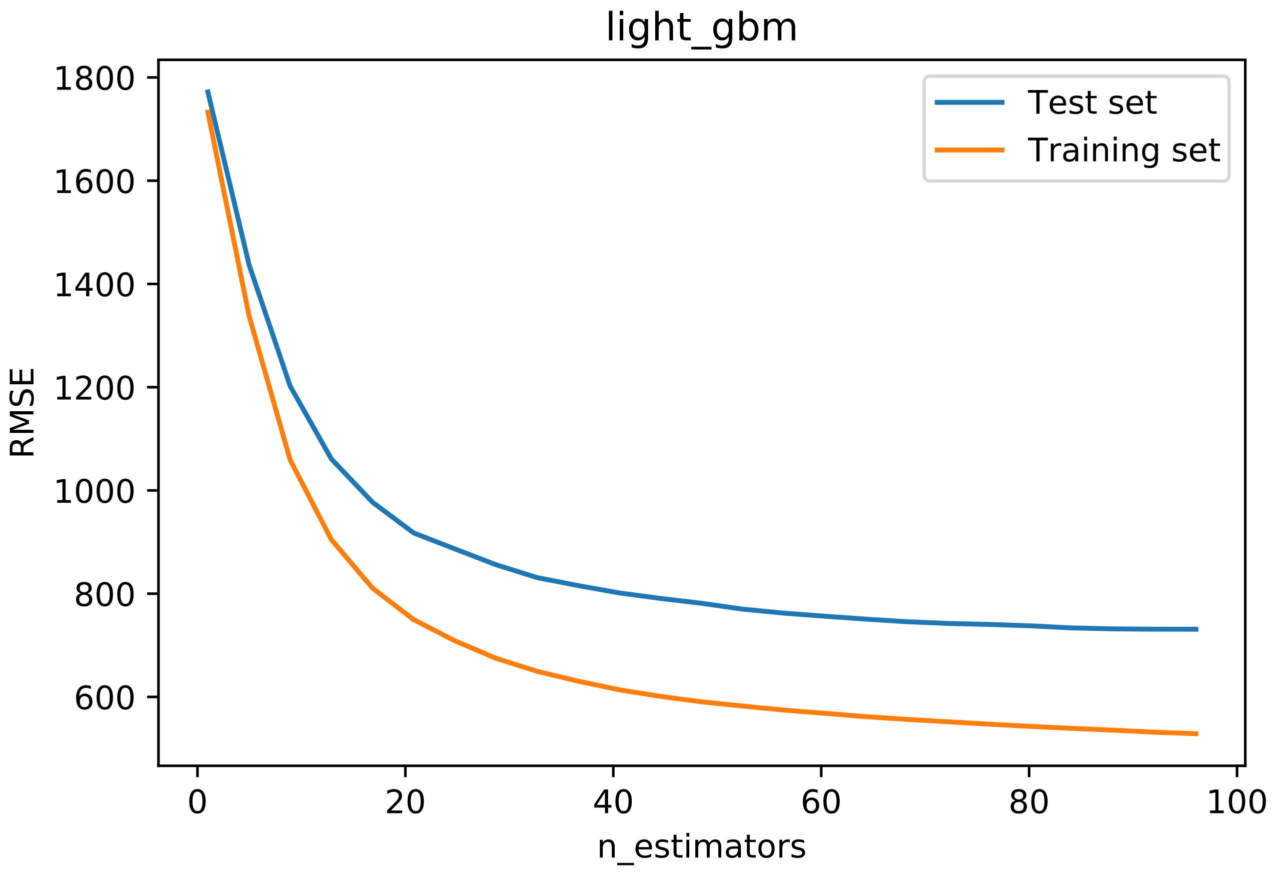


Figure 68: RMSE as function of number of estimators for lightgbm model

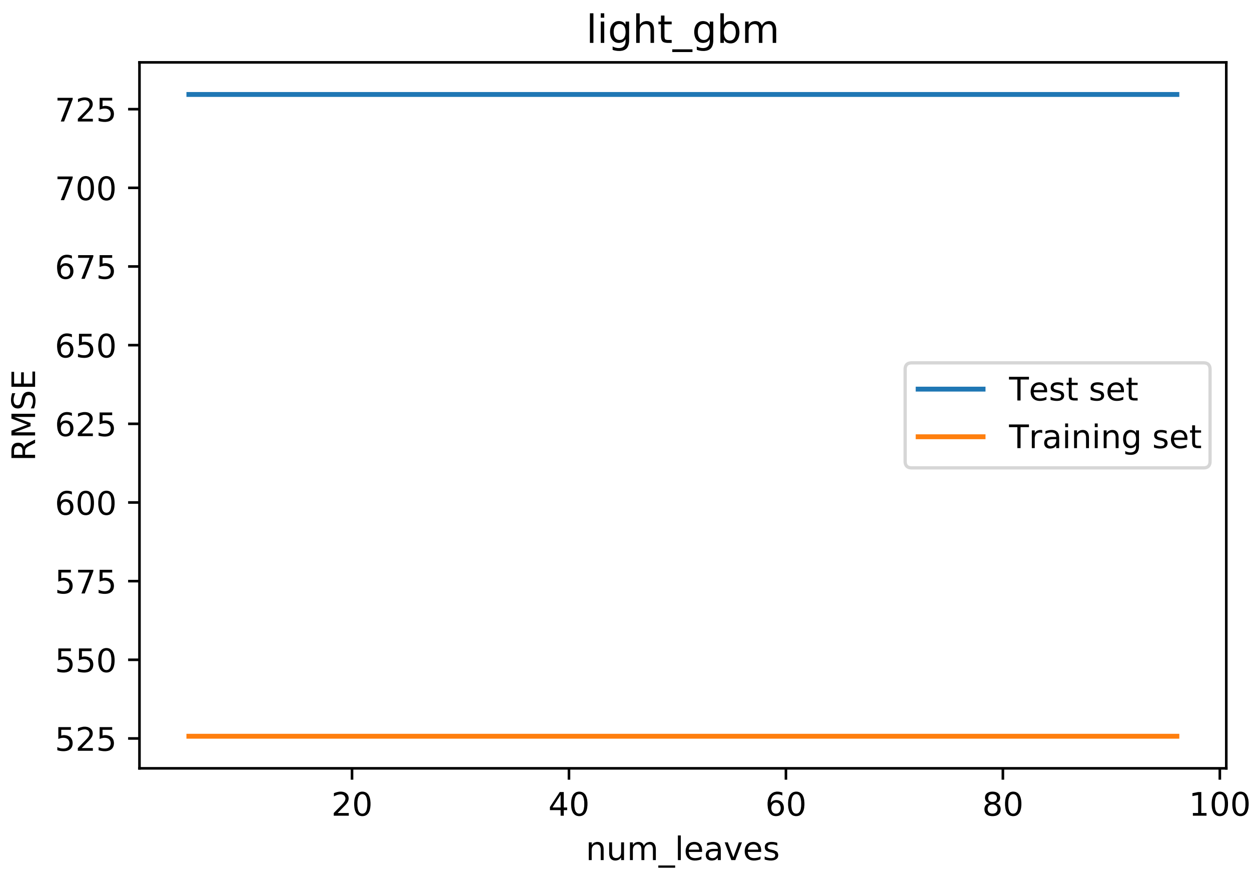


Figure 69: RMSE as function of number of leaves for lightgbm model

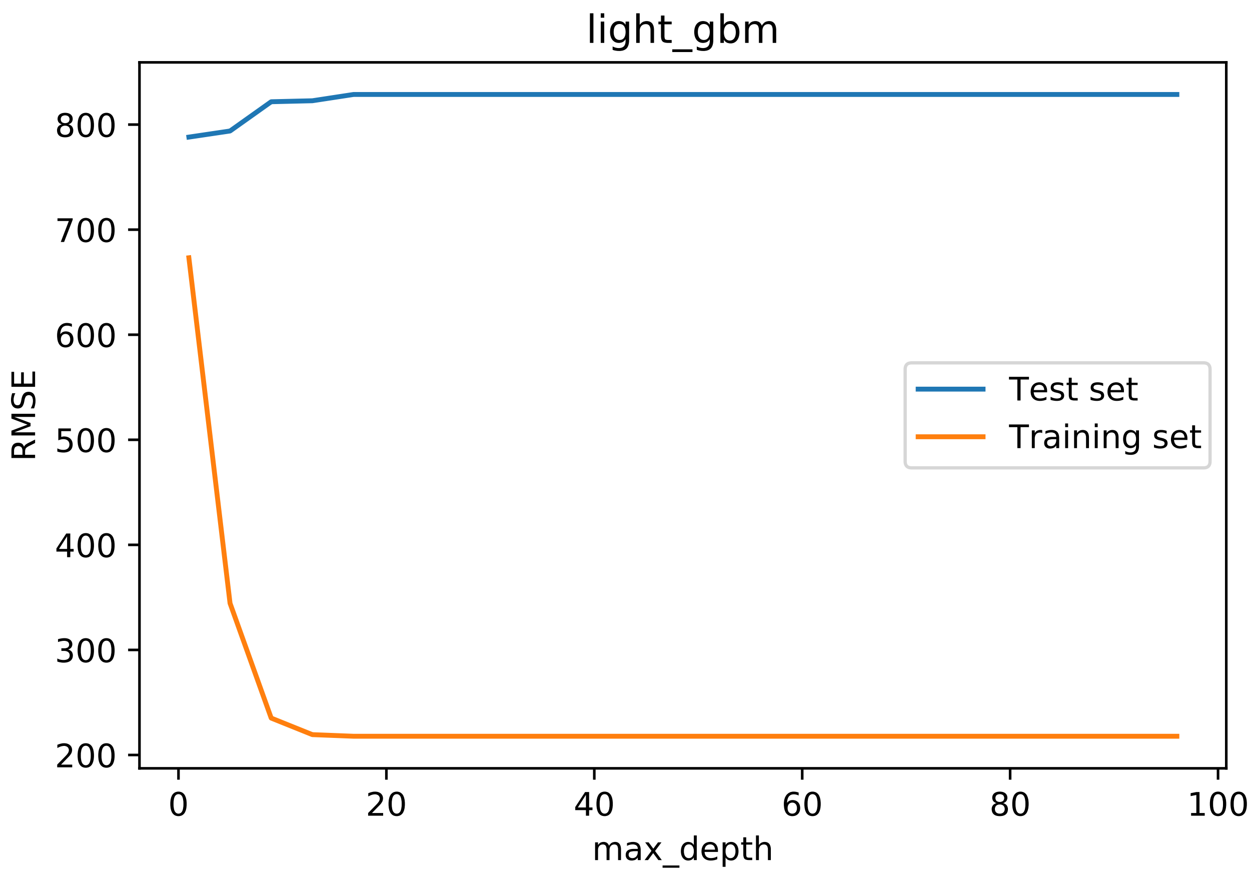


Figure 70: RMSE as function of Maximum depth for lightgbm model

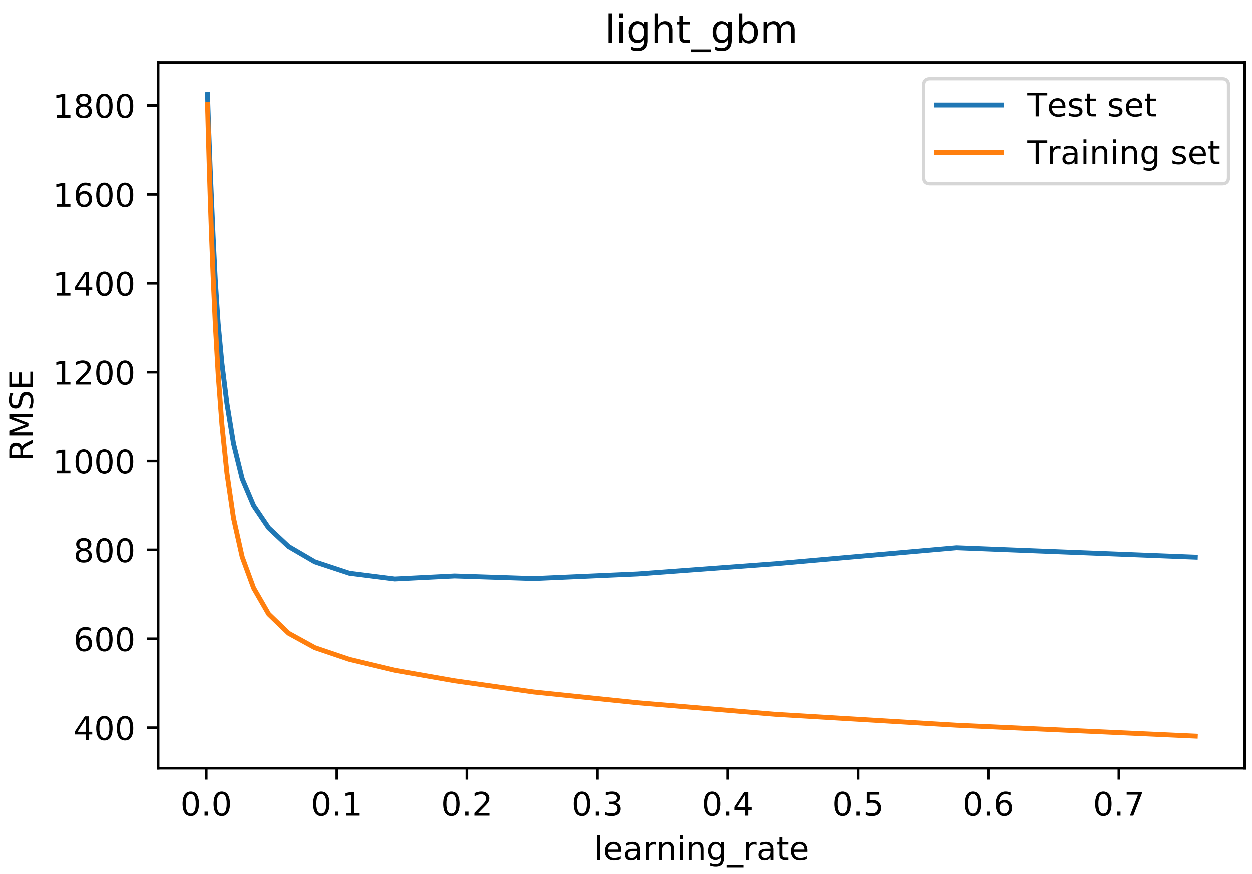


Figure 71: RMSE as function of learning rate for lightgbm model

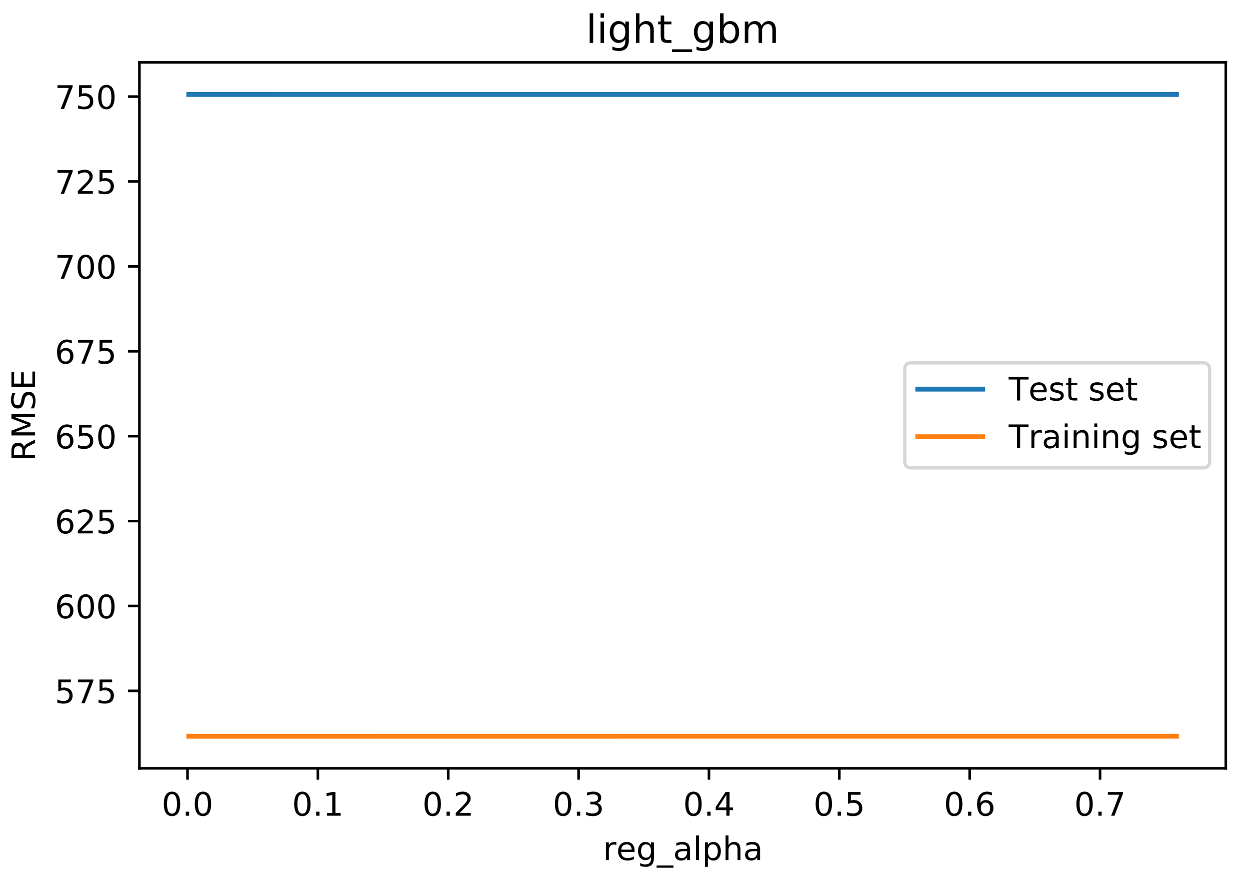


Figure 72: RMSE as function of regularization alpha for lightgbm model

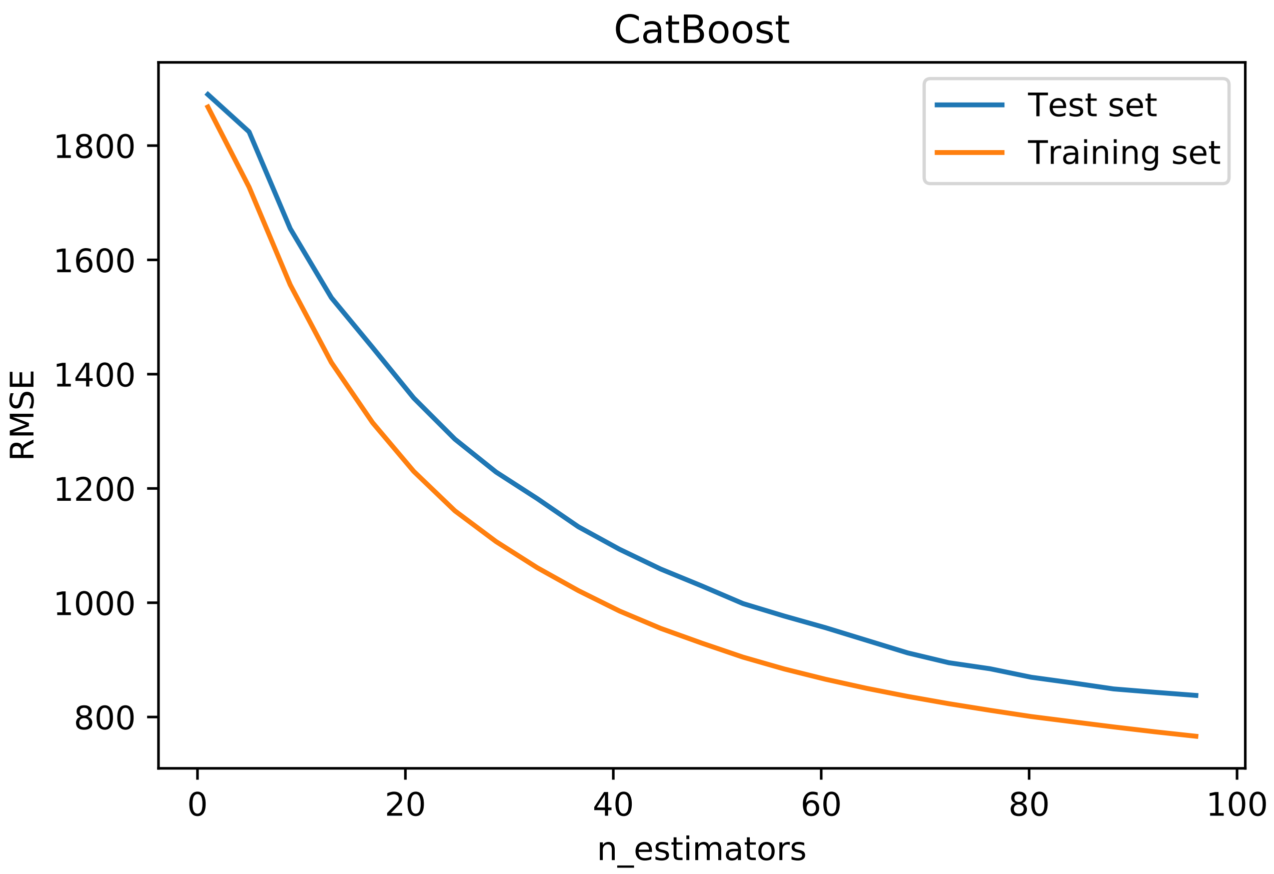


Figure 73: RMSE as function of number of estimators for catboost model

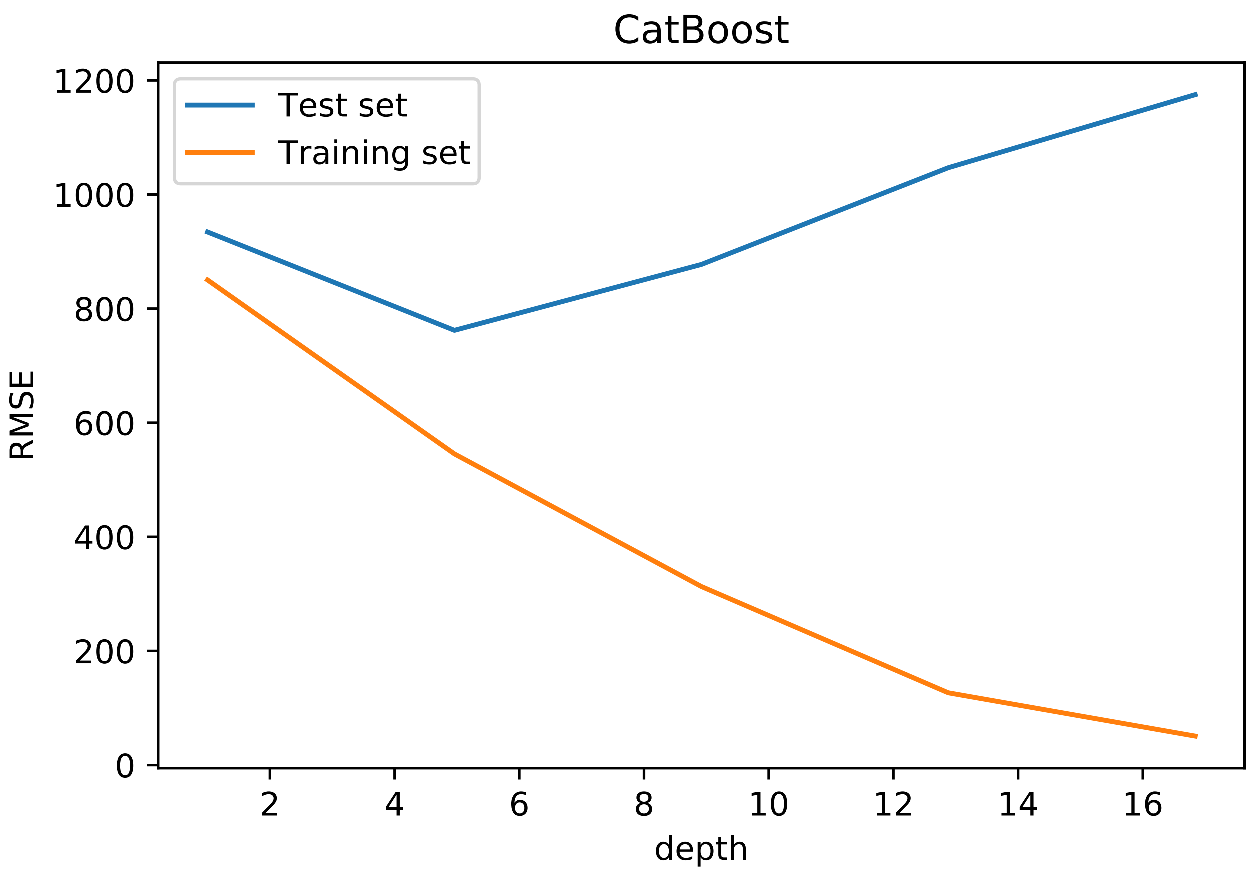


Figure 74: RMSE as function of depth for catboost model

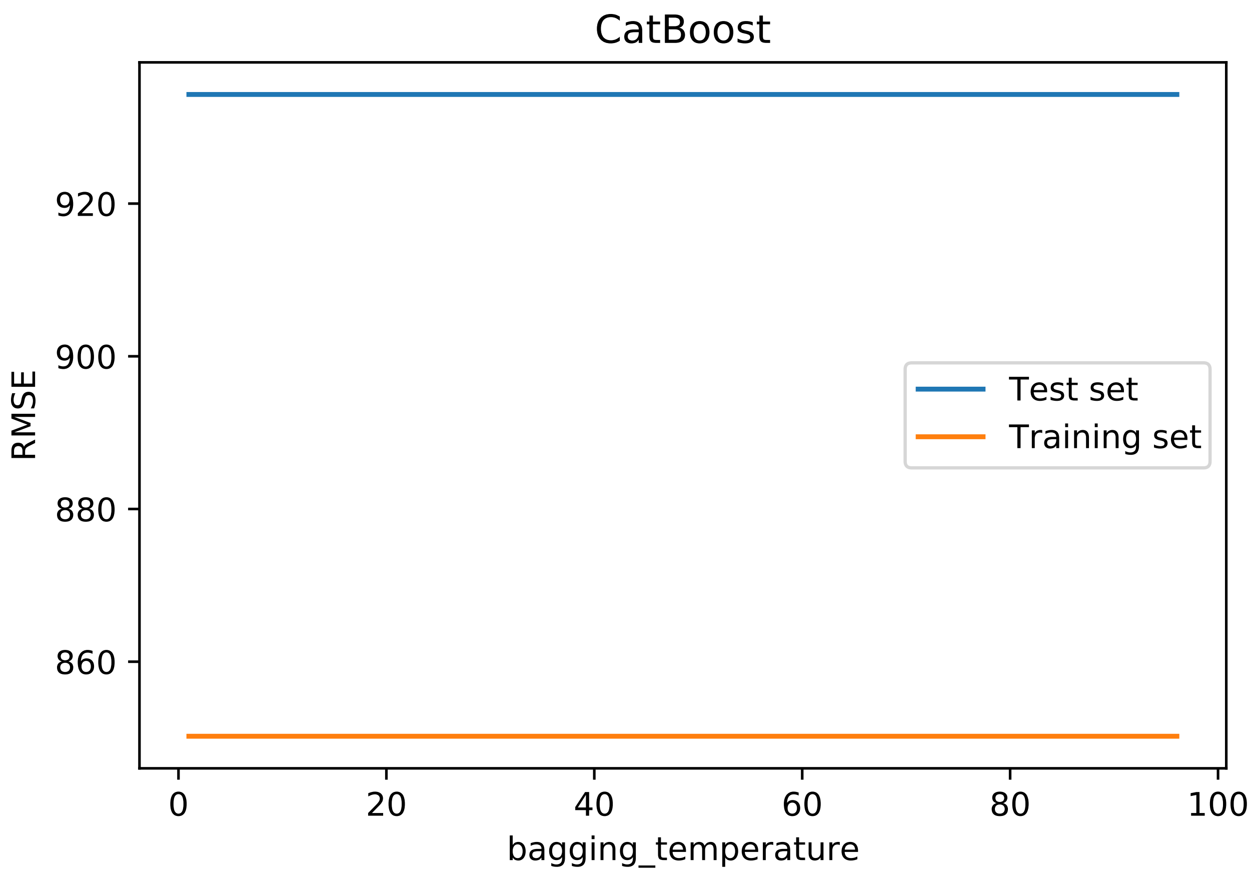


Figure 75: RMSE as function of bagging temperature for catboost model

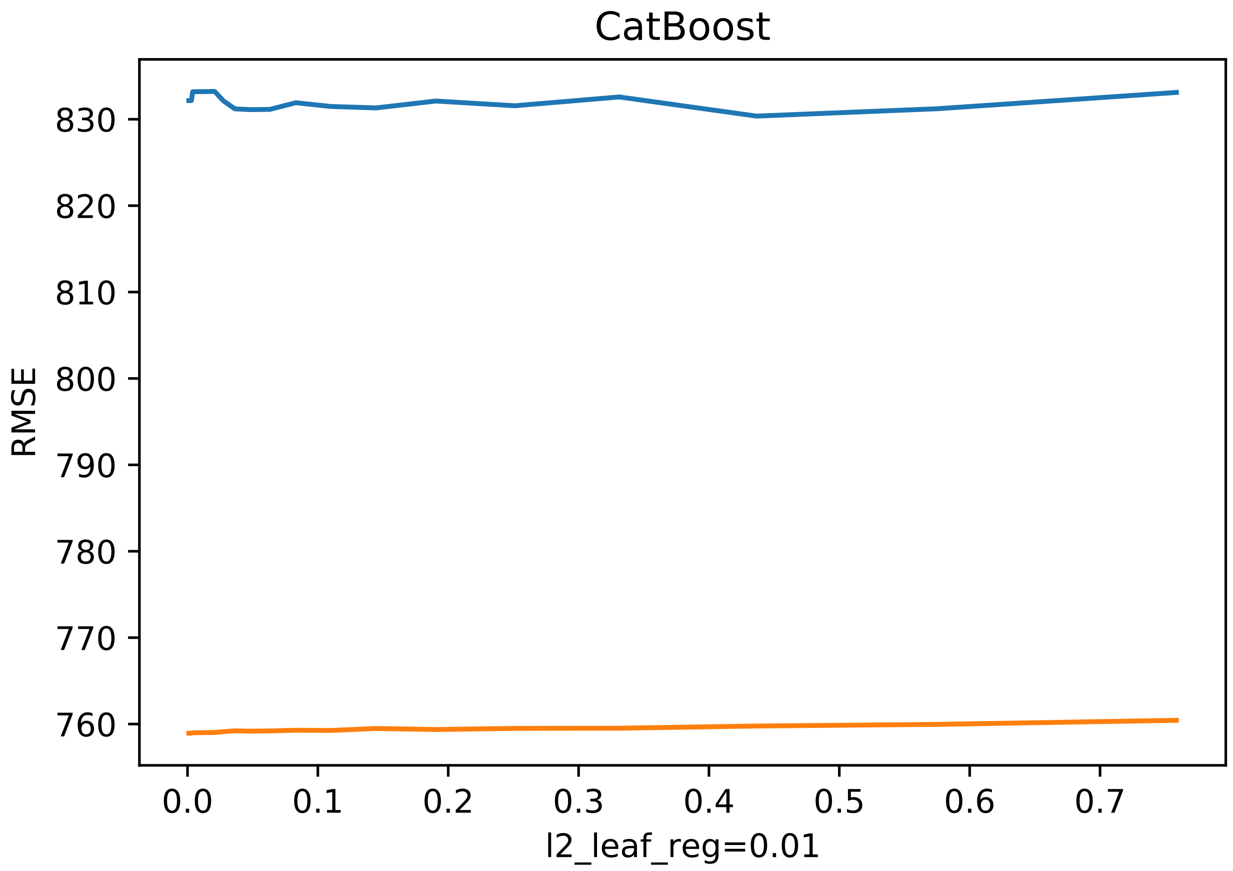


Figure 76: RMSE as function of l2 regularization for catboost model

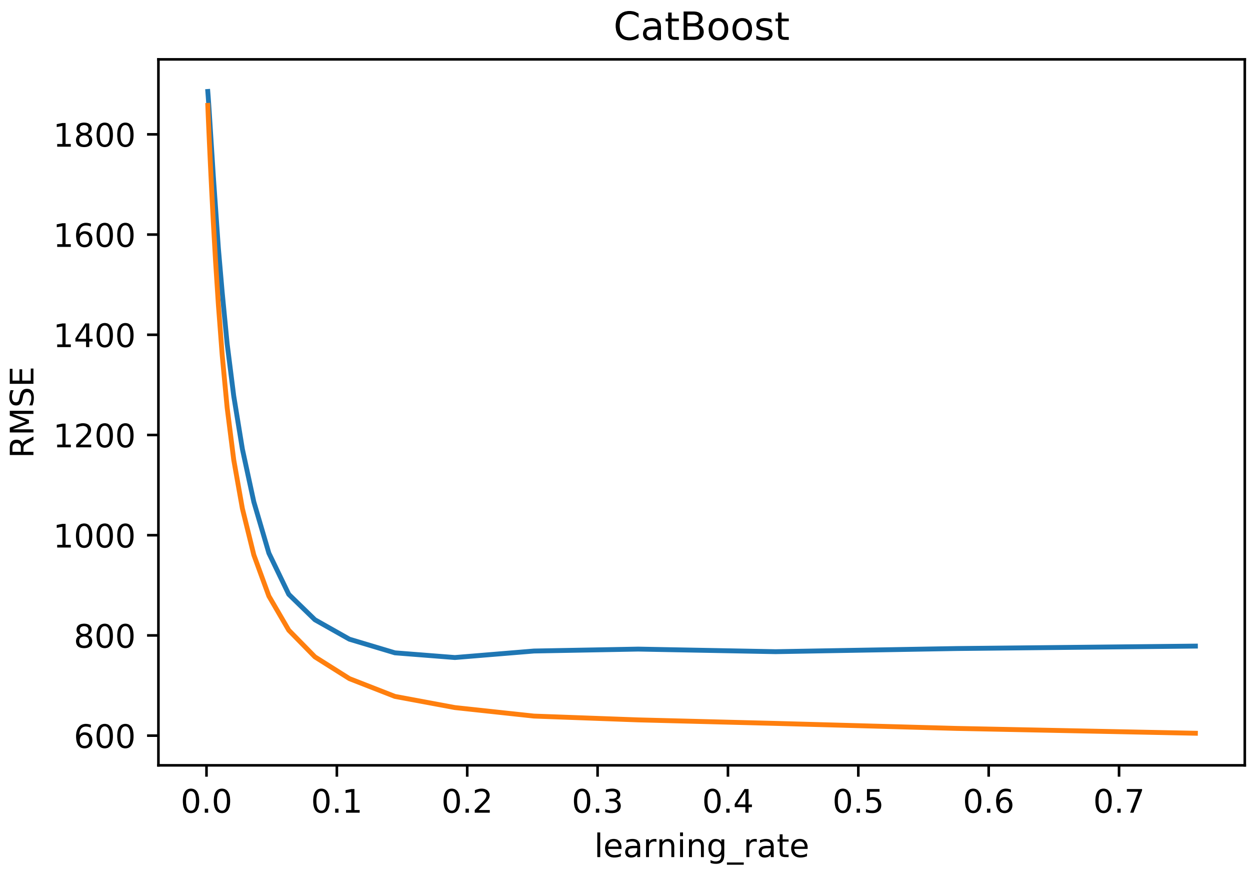
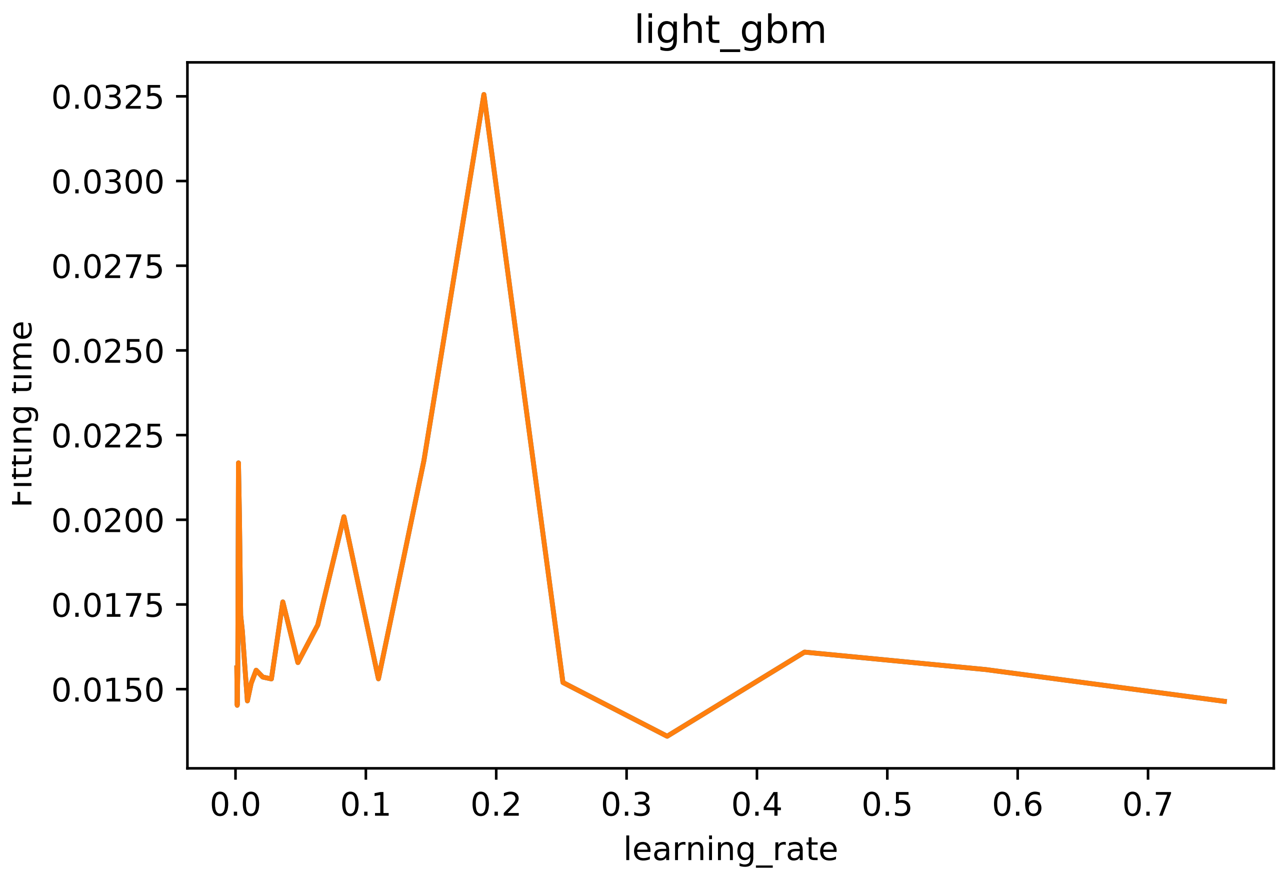


Figure 77: RMSE as function of learning rate for catboost model



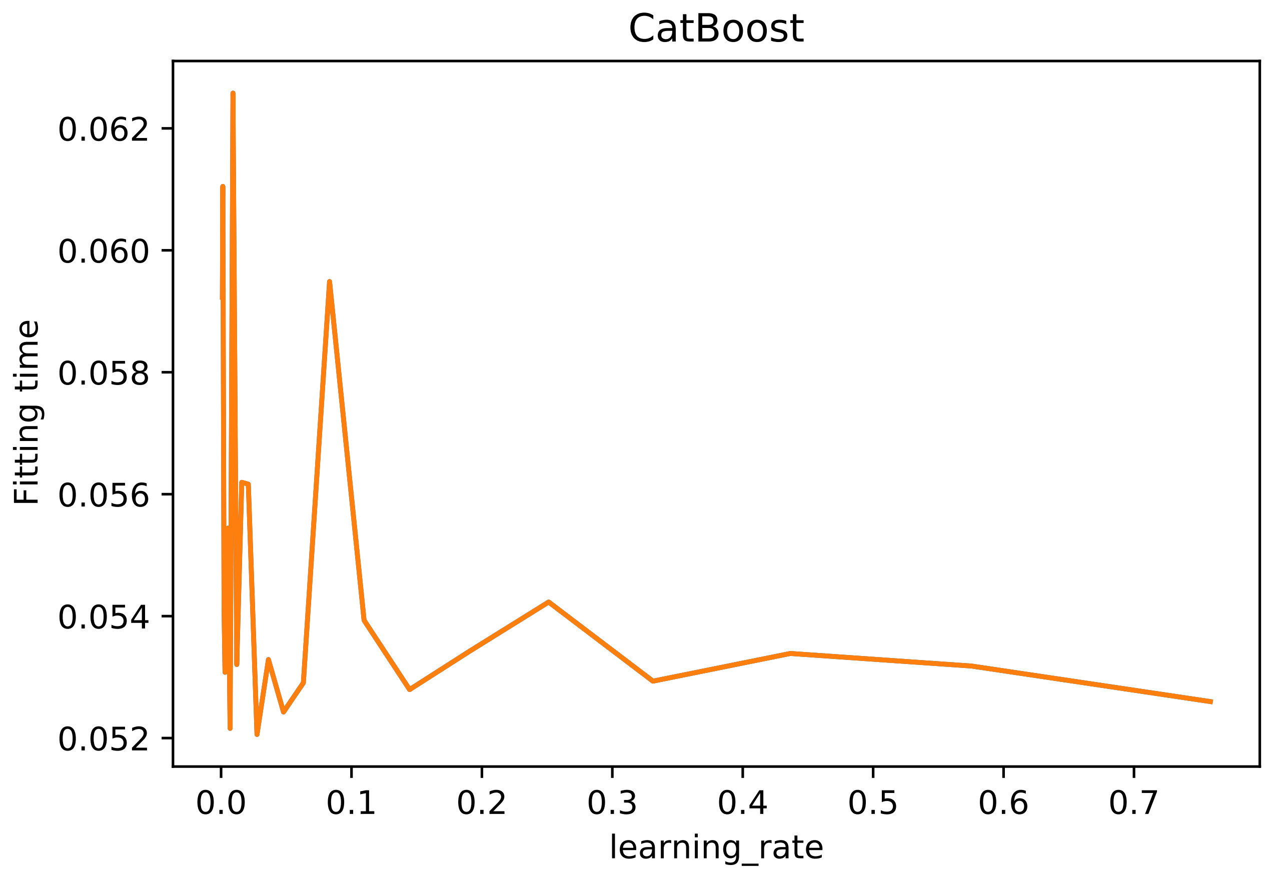


Figure 78: RMSE as function of learning rate for lightgbm models

Question 27.

The RMSE for training set is lower than RMSE of validation set because the training set is used to fit the parameters of the model. The validation set contains the data that the model has never seen before.Thus the model will have larger error when predicting the validation set.

All of the Average Training RMSE, Average Test RMSE, OOB Error (for Random Forest) for each dataset and model are in their respective sections. Please look there for discussion and analysis.

Question 28.

Random forest model employs subsampling technique to create training samples with replacement for the model to learn. The bootstrap sample is the data chosen to be "in-the-bag" by sampling with replacement. The OOB set is all data not chosen in the sampling process. In random forest, each of the OOB sample is evaluated by one decision tree that did not contain the OOB sample and a majority prediction is made for the OOB sample. The out-of-bag score is the mean prediction score from the out-of-bag sample.

Basically, in the Random Forest model, the training data is randomly sampled-with-replacement generating small subsets of data and this sampled data is called the "In the Bag" samples. These bootstrap samples are then fed to the decision trees that compose the Random Forest. Each decision tree is trained on a different sample. The samples that are not chosen are called the "Out of Bag" samples. After each decision tree is trained on its "In the Bag" samples, the "Out of Bag" samples from each decision tree are aggregated together. However, each sample is only considered out-of-bag for the trees that do not include it in their individual "In the Bag" sample. With this in mind, to calculate the out-of-bag error we will do the following.

- Find all trees that are not trained by the one "Out of Bag" sample.
- Take the average (this is regression not classification) of these models' result for the "Out of Bag" sample. Use it to compare to the true value of the "Out of Bag" sample.
- Repeat and aggregate the "Out of Bag" error for all samples in the aggregated "Out of Bag" samples.

The R^2 score is a statistical measure of how close the data are to the fitted regression line. It is the percentage of the response variable variation that is explained by a linear model. Mathematically it would be $R^2 = \frac{ExplainedVariation}{TotalVariation}$. This means that R^2 is always between 0 and 100%. 0% means that the linear model explains none of the variability of the responded data around its mean. 100% means that the model explains all the variability of the response data around its mean. However, R^2 score cannot determine whether the coefficient estimates and predictions are biased and cannot indicate whether the regression model is sufficient.

Appendix: Source Code

project4.py

```
from helpers import load_datasets, q4_helper, q5_helper, q6_helper, q7_helper, q8_helper, q12_helper,
    get_all_data_and_labels, get_all_data_and_labels_selected, get_all_data_and_labels_selected_no_scale
from pandas_profiling import ProfileReport
import pandas as pd
import pycountry_convert as pc
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from logger import logged
from sklearn.feature_selection import mutual_info_regression, f_regression
from sklearn.model_selection import KFold, cross_validate
from sklearn.linear_model import LinearRegression, Lasso, Ridge
from sklearn.neural_network import MLPRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.preprocessing import PolynomialFeatures
from sklearn.neural_network import MLPClassifier
from sklearn import tree
from sklearn.tree import export_graphviz
from caching import cached
from skopt import BayesSearchCV
from catboost import Pool, CatBoostRegressor
from skopt.space import Real, Categorical, Integer
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import GridSearchCV
import lightgbm as lgb
import statsmodels.api as sm
import warnings
import graphviz


import warnings
warnings.simplefilter("ignore")
# from sklearn.exceptions import ConvergenceWarning
# warnings.filterwarnings("ignore", category=ConvergenceWarning)


#TODO: UPLOAD TO OVERLEAF
def q1_q2():
    bike_df, suicide_df, video_df = load_datasets()
    profile_bike = ProfileReport(bike_df, title="Bike Sharing Report")
    profile_bike.to_file("output/Bike_Sharing_Report.html")
    profile_suicide = ProfileReport(suicide_df, title="Suicide Rates Overview Report")
    profile_suicide.to_file("output/Suicide Rates Overview Report.html")
    profile_video = ProfileReport(video_df, title="Video Transcoding Time Report")
    profile_video.to_file("output/Video Transcoding Time Report.html")

#TODO: CODE AND OVERLEAF
def q3():
    bike_df, suicide_df, video_df = load_datasets()
    fig, ax = plt.subplots()
    suicide_plot = sns.boxplot(data=suicide_df,x="suicides/100k pop",y="generation",hue="age",fliersize=0.001,ax=ax)
    ax.set_title(f"Suicides/100k pop vs. Generation and Age Range")
    fig.tight_layout(pad=0.25)
    plt.savefig(f"figures/q3_suicides.pdf")
    # plt.show()
    bike_df['weathersit'] = bike_df['weathersit'].astype('category')
    fig, axs = plt.subplots(2,3,figsize=(5* len(set(bike_df.weathersit.values)), 10))
    for holiday in (0,1):
        for j,weathersit in enumerate(set(bike_df.weathersit.values)):
            ax = axs[holiday,j]
            filtered = bike_df[(bike_df.weathersit == weathersit) & (bike_df.holiday == holiday)]
            if len(filtered) > 0:
                sns.boxplot(data=filtered,y="cnt",x="season",hue="weekday",ax=ax)
            ax.set_title(f"Bike Rentals vs. Season, Weekday\n{'Holiday' if holiday else 'Non-Holiday'},
                Weather={weathersit}")
    fig.tight_layout(pad=0.25)
    plt.savefig(f"figures/q3_bikes.pdf")
    # plt.show()
    # "codec", "bitrate", "framerate", "o_codec", "o_bitrate",
    framerates = sorted([ float(framerate) for framerate in set(video_df['o_framerate'].values)])
    bitrates = sorted([float(bitrate) for bitrate in set(video_df['o_bitrate'].values)])
    for target in ('utime','umem'):
        fig, axs = plt.subplots(len(framerates),len(bitrates),figsize=(5*len(bitrates),5*len(framerates)))
        for (i,framerate) in enumerate(framerates):
            for (j, bitrate) in enumerate(bitrates):
                ax = axs[i,j]
                filtered = video_df[(video_df['o_framerate'] == framerate) & (video_df['o_bitrate'] == bitrate)]
                if len(filtered) > 0:
                    sns.boxplot(data=filtered,x=target,y="codec",hue="o_codec",fliersize=0.001,ax=ax)
                    # sns.boxplot(data=filtered,x=target,y="codec",hue="o_codec",fliersize=0.001)
                ax.set_title(f"Encoding {target} vs. Codec\nFramerate={framerate}, Bitrate={bitrate}")
        fig.tight_layout(pad=0.25)
        plt.savefig(f"figures/q3_videos_{target}.pdf")



#TODO: UPLOAD TO OVERLEAF
def q4():
    max_year, smallest_month, max_month, count_list = q4_helper()
    for (i,j),current_list in count_list.items():
        plt.plot(list(range(len(current_list))), current_list)
        plt.xlabel('Day')
        plt.ylabel('Count')
        if i == 0:
            year = 2011
        else:
            year = 2012
        plt.title(f'Year: {year}, Month: {j} Count Number per Day')
        plt.savefig(f"figures/q4_{year}_{j}_count_number_per_day.pdf")
        plt.clf()

#TODO: UPLOAD TO OVERLEAF
def q5():
    suicide_df, ten_largest = q5_helper()
    for country_name in ten_largest:
        suicide_df_country = suicide_df[suicide_df['country'] == country_name]
        sns_plot = sns.relplot(data=suicide_df_country, x="year", y ="suicides/100k pop" , hue="age", col="sex")
        sns_plot.savefig(f"figures/q5_{country_name}_suicides100kpop_vs_year")



def q5_one_plot():
    suicide_df, ten_largest = q5_helper()
    suicide_df_country = suicide_df[suicide_df['country'].isin(ten_largest)]
    sns_plot = sns.relplot(data=suicide_df_country, x="year", y ="suicides/100k pop" , hue="age",
        col="sex",row='country',row_order=ten_largest)
    sns_plot.savefig(f"figures/q5__all_suicides100kpop_vs_year")

#TODO: UPLOAD TO OVERLEAF
@logged
def q6():
    video_df_utime = q6_helper()
    mean_utime  = video_df_utime['utime'].mean()
    median_utime =  video_df_utime['utime'].median()
    hist = video_df_utime.plot.hist(bins=50, title=f"Distribution of Video Transcoding Times\nMean={mean_utime : 0.3f},
        Median={median_utime : 0.3f}", legend=False, figsize=(7.5, 7.5))
    hist.set_xlabel("Video Transcoding Times")
    fig = hist.get_figure()
    fig.savefig("figures/q6_distribution_of_video_transcoding_times")
    mean, median = video_df_utime['utime'].mean(), video_df_utime['utime'].median()
    print(f"Mean Transcoding Time: {mean}, Median Transcoding Time: {median}")

#TODO: UPLOAD TO OVERLEAF
def q7():
    bike_df, suicide_df, video_df = q7_helper()
    return bike_df, suicide_df, video_df

#TODO: UPLOAD TO OVERLEAF
def q8():
    bike_df_final, suicide_df_final, video_df_final = q8_helper()
    return bike_df_final, suicide_df_final, video_df_final

#TODO: CODE AND OVERLEAF
@logged
def q9():
    # Need to analyze relevant features and decide which ones to return.
    data_and_labels = get_all_data_and_labels() #{'bike_cnt' : bike_df_cnt_data_labels(), 'bike_registered' :
        bike_df_registered_data_labels(), 'bike_casual' : bike_df_casual_data_labels(),  'video_umem' :
        video_df_umem_data_labels(),'video_utime' : video_df_utime_data_labels(),'suicide_per_100k' :
        suicide_per_100k_df_data_labels(),'suicide_total' : suicide_total_df_data_labels()}
    pruned_data = {}
    model = LinearRegression()
    kf = KFold(n_splits=10)
    for k, (features, target) in data_and_labels.items():
        all_features = features.columns
        f_scores, pvals =  f_regression(features.values, target.values.ravel())
        pruned_features = [feature for (feature,pval) in zip(features.columns,pvals) if pval > 0.05]

        test_score_before = np.mean(-cross_validate(model, features, target, scoring='neg_root_mean_squared_error',
            return_train_score=True, cv=kf)['test_score'])

        features.drop(pruned_features, axis=1,inplace=True)
        pruned_data[k] = (features, target)
        test_score_after = np.mean(-cross_validate(model, features, target, scoring='neg_root_mean_squared_error',
            return_train_score=True, cv=kf)['test_score'])

        print(f"{k}: After removing {len(pruned_features)}/{len(all_features)} insignificant features, Linear Regression
            RMSE on test set changes: {test_score_before} -> {test_score_after}")
    return pruned_data
    # return data_and_labels
        # features = features[[ feature for (feature,pval) in zip(features.columns,pvals) if pval < 0.05 ]]
        # data_and_labels[k] = (features,target)
     # k = 'video_umem'
    # features,target = data_and_labels[k]
    # mutual_infos = {}
    # f_regressions = {}
    # mutual_infos[k]= mutual_info_regression(features.values, target.values.ravel())
    # f_regressions[k] = f_regression(features.values, target.values.ravel())
    # Need to read through the data to determine the relevant features. Return the modified data.

    # return bike_df_final_features, bike_df_final_target, suicide_df_final_features, suicide_df_final_target,
        video_df_final_features, video_df_final_target

@logged
def q10():
# drop the feature with lowest mutual information iteratively as long as out-of-sample performance improves
    def stepwise_backward_selection_mutual_info(features,target,model,linear=False,n_splits=10):
        kf = KFold(n_splits=n_splits)
        test_score = np.mean(cross_validate(model, features.values, target.values.ravel(),
            scoring='neg_root_mean_squared_error', return_train_score=True, cv=kf)['test_score'])
        while True:
            if linear:
                weakest_feature = weakest_feature_linear(features,target)
            else:
                weakest_feature = weakest_feature_nonlinear(features,target)
            if weakest_feature is None:
                break
            new_features = features.drop([weakest_feature],axis=1)
            new_score  = np.mean(cross_validate(model, new_features.values, target.values.ravel(),
                scoring='neg_root_mean_squared_error', return_train_score=True, cv=kf)['test_score'])
            if new_score < test_score:
                break
            test_score = new_score
            features = new_features
        return features

    def weakest_feature_linear(features,target):
        f_scores, pvals =  f_regression(features.values, target.values.ravel())
        if max(pvals) < 0.05:
            return None
        return features.columns[np.argmax(pvals)]

    def weakest_feature_nonlinear(features,target):
        mutual_info = mutual_info_regression(features.values, target.values.ravel())
        return features.columns[np.argmin(mutual_info)]

    # @cached
    def prune_all(data_and_labels,model,linear=False):
        for k, (features, target) in data_and_labels.items():
            pruned_features = stepwise_backward_selection_mutual_info(features,target,model,linear=linear)
            data_and_labels[k] = (pruned_features,target)
        return data_and_labels


    # data_and_labels = get_all_data_and_labels()
    # k='video_umem'
    # features,target = data_and_labels[k]

    models = (LinearRegression(), Lasso(alpha = 0.5),Ridge(alpha = 0.5),
              MLPRegressor(), RandomForestRegressor(n_estimators = 10, max_depth = 30, max_features= "auto"))
    for model in models:
        linear = 'linear_model' in f"{type(model)}"
        # data_and_labels = model_selection_all(model=model,linear=linear)
        data_and_labels = get_all_data_and_labels()
        print(f"Model is '{model}', selecting using {'p-Values of F-statistics' if linear else 'Mutual Information'}")
        print(f"Number of features BEFORE model selection:\n{ {k : len(features.columns) for k, (features, target) in
            data_and_labels.items()} }")
        og_cols = {k : features.columns for k, (features,target) in data_and_labels.items()}
        data_and_labels = prune_all(data_and_labels,model,linear=linear)
        removed_cols = {k : "'{}'".format("', '".join(set(og_cols[k]) - set(features.columns))) for  k, (features,target)
            in data_and_labels.items()}
        for k, cols in removed_cols.items():
            print(f"\tFrom {k} pruned: {cols}")
        print(f"Number of features AFTER model selection:\n{ {k : len(features.columns) for k, (features, target) in
            data_and_labels.items()} }\n")

#TODO: UPLOAD TO OVERLEAF, RE-RUN WITH PRUNED DATA
@logged
def q10_q11_13_linear():
    data_and_labels = get_all_data_and_labels_selected(pruning_for='linear')
    # data_and_labels = get_all_data_and_labels()
    model = LinearRegression()
    kf = KFold(n_splits=10)
    print("Linear Regression Results: ")
    for k, (features, target) in data_and_labels.items():
        scores = cross_validate(model, features.values, target.values.ravel(), scoring='neg_root_mean_squared_error',
            return_train_score=True, cv=kf)
        avg_train_score = np.mean(-scores['train_score'])
        avg_test_score = np.mean(-scores['test_score'])
        print(f"{k}: Avg. Training RMSE Error: {avg_train_score}, Avg. Test RMSE Error: {avg_test_score}")

@logged
def q10_q11_13_linear_shuffle():
    data_and_labels = get_all_data_and_labels_selected(pruning_for='linear')
    # data_and_labels = get_all_data_and_labels()
    model = LinearRegression()
    kf = KFold(n_splits=10, shuffle=True, random_state=0)
    print("Linear Regression Shuffling Data Results: ")
    for k, (features, target) in data_and_labels.items():
        scores = cross_validate(model, features.values, target.values.ravel(), scoring='neg_root_mean_squared_error',
            return_train_score=True, cv=kf)
        avg_train_score = np.mean(-scores['train_score'])
        avg_test_score = np.mean(-scores['test_score'])
        print(f"{k}: Avg. Training RMSE Error: {avg_train_score}, Avg. Test RMSE Error: {avg_test_score}")

#TODO: UPLOAD TO OVERLEAF, RE-RUN WITH PRUNED DATA
@logged
def q10_q11_13_lasso():
    data_and_labels = get_all_data_and_labels_selected(pruning_for='linear')
    alpha = [0.1, 0.2, 0.5, 0.75, 1.0, 2.0, 5.0, 10.0]
    best_alpha = {}
    kf = KFold(n_splits=10)
    for parameter in alpha:
        model = Lasso(alpha = parameter)
        print(f"Lasso Results for {parameter}: ")
        for k, (features, target) in data_and_labels.items():
            scores = cross_validate(model, features.values, target.values.ravel(), scoring='neg_root_mean_squared_error',
                return_train_score=True, cv=kf)
            avg_train_score = np.mean(-scores['train_score'])
            avg_test_score = np.mean(-scores['test_score'])
            # The smaller the RMSE, the better.
            if k not in best_alpha: best_alpha[k] = (parameter, avg_train_score, avg_test_score)
            if best_alpha[k][2] > avg_test_score:
                best_alpha[k] = (parameter, avg_train_score, avg_test_score)
            print(f"{k}: Avg. Training RMSE Error: {avg_train_score}, Avg. Test RMSE Error: {avg_test_score}")
    print("Best Alphas for each Dataset: ")
    for key, value in best_alpha.items():
        print(key, value)
    pass

@logged
def q10_q11_13_lasso_shuffle():
    data_and_labels = get_all_data_and_labels_selected(pruning_for='linear')
    alpha = [0.1, 0.2, 0.5, 0.75, 1.0, 2.0, 5.0, 10.0]
    best_alpha = {}
    kf = KFold(n_splits=10, shuffle=True, random_state=0)
    for parameter in alpha:
        model = Lasso(alpha = parameter)
        print(f"Lasso Shuffling Data Results for {parameter}: ")
        for k, (features, target) in data_and_labels.items():
            scores = cross_validate(model, features.values, target.values.ravel(), scoring='neg_root_mean_squared_error',
                return_train_score=True, cv=kf)
            avg_train_score = np.mean(-scores['train_score'])
            avg_test_score = np.mean(-scores['test_score'])
            # The smaller the RMSE, the better.
            if k not in best_alpha: best_alpha[k] = (parameter, avg_train_score, avg_test_score)
            if best_alpha[k][2] > avg_test_score:
                best_alpha[k] = (parameter, avg_train_score, avg_test_score)
            print(f"{k}: Avg. Training RMSE Error: {avg_train_score}, Avg. Test RMSE Error: {avg_test_score}")
    print("Best Alphas for each Dataset: ")
    for key, value in best_alpha.items():
        print(key, value)
    pass

#TODO: UPLOAD TO OVERLEAF, RE-RUN WITH PRUNED DATA
@logged
def q10_q11_13_ridge():
    data_and_labels = get_all_data_and_labels_selected(pruning_for='linear')
    alpha = [0.1, 0.2, 0.5, 0.75, 1.0, 2.0, 5.0, 10.0]
    best_alpha = {}
    kf = KFold(n_splits=10)
    for parameter in alpha:
        model = Ridge(alpha = parameter)
        print(f"Ridge Results for {parameter}: ")
        for k, (features, target) in data_and_labels.items():
            scores = cross_validate(model, features.values, target.values.ravel(), scoring='neg_root_mean_squared_error',
                return_train_score=True, cv=kf)
            avg_train_score = np.mean(-scores['train_score'])
            avg_test_score = np.mean(-scores['test_score'])
            # The smaller the RMSE, the better.
            if k not in best_alpha: best_alpha[k] = (parameter, avg_train_score, avg_test_score)
            if best_alpha[k][2] > avg_test_score:
                best_alpha[k] = (parameter, avg_train_score, avg_test_score)
            print(f"{k}: Avg. Training RMSE Error: {avg_train_score}, Avg. Test RMSE Error: {avg_test_score}")
    print("Best Alphas for each Dataset: ")
    for key, value in best_alpha.items():
        print(key, value)
    pass

@logged
def q10_q11_13_ridge_shuffle():
    data_and_labels = get_all_data_and_labels_selected(pruning_for='linear')
    alpha = [0.1, 0.2, 0.5, 0.75, 1.0, 2.0, 5.0, 10.0]
    best_alpha = {}
    kf = KFold(n_splits=10, shuffle=True, random_state=0)
    for parameter in alpha:
        model = Ridge(alpha = parameter)
        print(f"Ridge Shuffled Data Results for {parameter}: ")
        for k, (features, target) in data_and_labels.items():
            scores = cross_validate(model, features.values, target.values.ravel(), scoring='neg_root_mean_squared_error',
                return_train_score=True, cv=kf)
            avg_train_score = np.mean(-scores['train_score'])
            avg_test_score = np.mean(-scores['test_score'])
            # The smaller the RMSE, the better.
            if k not in best_alpha: best_alpha[k] = (parameter, avg_train_score, avg_test_score)
            if best_alpha[k][2] > avg_test_score:
                best_alpha[k] = (parameter, avg_train_score, avg_test_score)
            print(f"{k}: Avg. Training RMSE Error: {avg_train_score}, Avg. Test RMSE Error: {avg_test_score}")
    print("Best Alphas for each Dataset: ")
    for key, value in best_alpha.items():
        print(key, value)
    pass

@logged
def q12_linear():
    # Don't do feature scaling. Load dataset. (Should do feature selection here.)
    data_and_labels = get_all_data_and_labels_selected_no_scale(pruning_for='linear')
    bike_features, bike_targets = data_and_labels['bike_cnt']
    video_features, video_targets = data_and_labels['video_utime']
    suicide_features, suicide_targets = data_and_labels['suicide_per_100k']

    model_bike = LinearRegression()
    model_video = LinearRegression()
    model_suicide = LinearRegression()

    # kf = KFold(n_splits=10, shuffle=True, random_state=0)
    kf = KFold(n_splits=10)

    scores_bike = cross_validate(model_bike, bike_features.values, bike_targets.values.ravel(),
        scoring='neg_root_mean_squared_error', return_train_score=True, cv=kf)
    scores_video = cross_validate(model_video, video_features.values, video_targets.values.ravel(),
        scoring='neg_root_mean_squared_error', return_train_score=True, cv=kf)
    scores_suicides = cross_validate(model_suicide, suicide_features.values, suicide_targets.values.ravel(),
        scoring='neg_root_mean_squared_error', return_train_score=True, cv=kf)

    bike_avg_train_score = np.mean(-scores_bike['train_score'])
    bike_avg_test_score = np.mean(-scores_bike['test_score'])

    video_avg_train_score = np.mean(-scores_video['train_score'])
    video_avg_test_score = np.mean(-scores_video['test_score'])

    suicide_avg_train_score = np.mean(-scores_suicides['train_score'])
    suicide_avg_test_score = np.mean(-scores_suicides['test_score'])

    print("No Feature Scaling Test for Linear Regression; Using the best models from Feature Scaling")
    print(f"Bike Sharing Dataset: Avg. Training RMSE Error: {bike_avg_train_score}, Avg. Test RMSE Error:
        {bike_avg_test_score}")
    print(f"Video Dataset: Avg. Training RMSE Error: {video_avg_train_score}, Avg. Test RMSE Error:
        {video_avg_test_score}")
    print(f"Suicide Dataset: Avg. Training RMSE Error: {suicide_avg_train_score}, Avg. Test RMSE Error:
        {suicide_avg_test_score}")

@logged
def q12_lasso():
    # Don't do feature scaling. Load dataset. (Should do feature selection here.)
    data_and_labels = get_all_data_and_labels_selected_no_scale(pruning_for='linear')
    bike_features, bike_targets = data_and_labels['bike_cnt']
    video_features, video_targets = data_and_labels['video_utime']
    suicide_features, suicide_targets = data_and_labels['suicide_per_100k']

    model_bike = Lasso(alpha = 2.0)
    model_video = Lasso(alpha = 0.1)
    model_suicide = Lasso(alpha = 0.1)

    # kf = KFold(n_splits=10, shuffle=True, random_state=0)
    kf = KFold(n_splits=10)

    scores_bike = cross_validate(model_bike, bike_features.values, bike_targets.values.ravel(),
        scoring='neg_root_mean_squared_error', return_train_score=True, cv=kf)
    scores_video = cross_validate(model_video, video_features.values, video_targets.values.ravel(),
        scoring='neg_root_mean_squared_error', return_train_score=True, cv=kf)
    scores_suicides = cross_validate(model_suicide, suicide_features.values, suicide_targets.values.ravel(),
        scoring='neg_root_mean_squared_error', return_train_score=True, cv=kf)

    bike_avg_train_score = np.mean(-scores_bike['train_score'])
    bike_avg_test_score = np.mean(-scores_bike['test_score'])

    video_avg_train_score = np.mean(-scores_video['train_score'])
    video_avg_test_score = np.mean(-scores_video['test_score'])

    suicide_avg_train_score = np.mean(-scores_suicides['train_score'])
    suicide_avg_test_score = np.mean(-scores_suicides['test_score'])

    print("No Feature Scaling Test for Lasso Regression; Using the best models from Feature Scaling")
    print(f"Bike Sharing Dataset: Avg. Training RMSE Error: {bike_avg_train_score}, Avg. Test RMSE Error:
        {bike_avg_test_score}")
    print(f"Video Dataset: Avg. Training RMSE Error: {video_avg_train_score}, Avg. Test RMSE Error:
        {video_avg_test_score}")
    print(f"Suicide Dataset: Avg. Training RMSE Error: {suicide_avg_train_score}, Avg. Test RMSE Error:
        {suicide_avg_test_score}")


@logged
def q12_ridge():
    # Don't do feature scaling. Load dataset. (Should do feature selection here.)
    data_and_labels = get_all_data_and_labels_selected_no_scale(pruning_for='linear')
    bike_features, bike_targets = data_and_labels['bike_cnt']
    video_features, video_targets = data_and_labels['video_utime']
    suicide_features, suicide_targets = data_and_labels['suicide_per_100k']

    model_bike = Ridge(alpha = 0.5)
    model_video = Ridge(alpha = 5.0)
    model_suicide = Ridge(alpha = 10.0)

    kf = KFold(n_splits=10, shuffle=True, random_state=0)
    # kf = KFold(n_splits=10)

    scores_bike = cross_validate(model_bike, bike_features.values, bike_targets.values.ravel(),
        scoring='neg_root_mean_squared_error', return_train_score=True, cv=kf)
    scores_video = cross_validate(model_video, video_features.values, video_targets.values.ravel(),
        scoring='neg_root_mean_squared_error', return_train_score=True, cv=kf)
    scores_suicides = cross_validate(model_suicide, suicide_features.values, suicide_targets.values.ravel(),
        scoring='neg_root_mean_squared_error', return_train_score=True, cv=kf)
    bike_avg_train_score = np.mean(-scores_bike['train_score'])
    bike_avg_test_score = np.mean(-scores_bike['test_score'])

    video_avg_train_score = np.mean(-scores_video['train_score'])
    video_avg_test_score = np.mean(-scores_video['test_score'])

    suicide_avg_train_score = np.mean(-scores_suicides['train_score'])
    suicide_avg_test_score = np.mean(-scores_suicides['test_score'])

    print("No Feature Scaling Test for Ridge Regression; Using the best models from Feature Scaling")
    print(f"Bike Sharing Dataset: Avg. Training RMSE Error: {bike_avg_train_score}, Avg. Test RMSE Error:
        {bike_avg_test_score}")
    print(f"Video Dataset: Avg. Training RMSE Error: {video_avg_train_score}, Avg. Test RMSE Error:
        {video_avg_test_score}")
    print(f"Suicide Dataset: Avg. Training RMSE Error: {suicide_avg_train_score}, Avg. Test RMSE Error:
        {suicide_avg_test_score}")

@logged
def q14():
    degrees = [2]
    kf = KFold(n_splits=10)
    for degree in degrees:
        data_and_labels = get_all_data_and_labels_selected(pruning_for='linear')
        polynomial_features = PolynomialFeatures(degree=degree)
        for k, (features, target) in data_and_labels.items():
            poly_features = polynomial_features.fit_transform(features.values)
            mod = sm.OLS(target.values, poly_features)
            results = mod.fit()
            print(results.summary())

@logged
def q14_q15_16_poly():
    alpha = 1000.0
    degrees = [2]
    kf = KFold(n_splits=10)
    model = Ridge(alpha = alpha)
    for degree in degrees:
        data_and_labels = get_all_data_and_labels_selected(pruning_for='linear')
        print("Polyniomial Linear Regression Results For {} Degree:".format(degree))
        polynomial_features = PolynomialFeatures(degree=degree)
        for k, (features, target) in data_and_labels.items():
            if k == "video_utime":



                poly_features = polynomial_features.fit_transform(features.values)
                print(np.shape(poly_features))
                scores = cross_validate(model, poly_features, target.values.ravel(),
                                        scoring='neg_root_mean_squared_error', return_train_score=True, cv=kf)
                avg_train_score = np.mean(-scores['train_score'])
                avg_test_score = np.mean(-scores['test_score'])
                print(f"{k}: Avg. Training RMSE Error: {avg_train_score}, Avg. Test RMSE Error: {avg_test_score}")

        data_and_labels = get_all_data_and_labels_selected(pruning_for='linear')
        print("Polyniomial Linear Regression Results For {} Degree:".format(degree))
        polynomial_features = PolynomialFeatures(degree=degree)
        for k, (features, target) in data_and_labels.items():
            if k == "video_utime":
                #out = np.zeros_like(features.values)
                #mask = np.zeros_like(features.values)
                #for i in range(np.size(out, 1)):
                #    if i == (3 or 4 or 5 or 6 or 7 or 8 or 9 or 10 or 11 or 12 or 18 or 19 or 20 or 21):
                #            mask[i] = np.ones_like(np.size(out, 0))

                # np.reciprocal(features.values, where=np.multiply(features.values,mask) > 0.0, out=out)

                # features = np.hstack((features.values, out))

                temp = features.values
                temp1 = temp[:,12]
                temp2 = temp[:,6]
                temp3 = temp[:,9]
                temp4 = np.multiply(np.multiply(temp1,temp2), np.reciprocal(temp3, where=temp3 > 0.0))

                poly_features = polynomial_features.fit_transform(features.values)
                poly_features = np.column_stack((poly_features, temp4))
                # poly_features = np.concatenate((poly_features, temp4), axis=1)
                # poly_features = np.hstack((poly_features, temp4))
                print(np.shape(poly_features))
                scores = cross_validate(model, poly_features, target.values.ravel(),
                                        scoring='neg_root_mean_squared_error', return_train_score=True, cv=kf)
                avg_train_score = np.mean(-scores['train_score'])
                avg_test_score = np.mean(-scores['test_score'])
                print(f"{k}: Avg. Training RMSE Error: {avg_train_score}, Avg. Test RMSE Error: {avg_test_score}")

                #['codec_h264',
                #'codec_mpeg4', 'codec_vp8', 'height', 'width', 'bitrate', 'framerate', 'i', 'p',
                #'frames', 'i_size', 'p_size','size',
                #'o_codec_flv', 'o_codec_h264', 'o_codec_mpeg4', 'o_codec_vp8',
                #'o_bitrate', 'o_framerate', 'o_width', 'o_height','utime']
@logged
def q17_18_19_20_NN():
    kf = KFold(n_splits=10)
    model = MLPClassifier(solver='adam',
                          activation = 'relu',
                          alpha=0.0001,
                          hidden_layer_sizes=(100, 50),
                          learning_rate = 'adaptive',
                          learning_rate_init = 0.001,
                          batch_size='auto',
                          verbose = False,
                          max_iter = 200,
                          random_state=1)
    data_and_labels = get_all_data_and_labels_selected(pruning_for='nn')
    print("Fully Connected Network Results:")
    for k, (features, target) in data_and_labels.items():
        print(f"Cross-validating on {k}")
        scores = cross_validate(model, features.values, target.values.ravel(),
                                scoring='neg_root_mean_squared_error', return_train_score=True, cv=kf)
        avg_train_score = np.mean(-scores['train_score'])
        avg_test_score = np.mean(-scores['test_score'])
        print(f"{k}: Avg. Training RMSE Error: {avg_train_score}, Avg. Test RMSE Error: {avg_test_score}")


@logged
def q18():
    data_and_labels = get_all_data_and_labels_selected(pruning_for='nn')
    features, target = data_and_labels['bike_cnt'] #use the bike dataset to determine model parameters.
    kf = KFold(n_splits=10)
    alphas = [10**(-k) for k in range(1,5)]
    depths = list(range(2,20,5))
    widths = [20,50,100]
    networksizes = [ tuple(width for _ in range(depth) ) for depth in depths for width in widths] #e.g. [20, 20],
        [50,50,50,...,50], [100,100,100,100,100]
    test_performance = {}
    for alpha in alphas:
        for networksize in networksizes:
            model = MLPRegressor(solver='adam',
                                  activation = 'relu',
                                  alpha=alpha,
                                  hidden_layer_sizes=networksize,
                                  learning_rate = 'adaptive',
                                  learning_rate_init = 0.001,
                                  batch_size='auto',
                                  verbose = False,
                                  max_iter = 200,
                                  random_state=1)
            scores = cross_validate(model, features.values, target.values.ravel(),scoring='neg_root_mean_squared_error',
                return_train_score=True, cv=kf)
            # avg_train_score = np.mean(-scores['train_score'])
            avg_test_score = np.mean(-scores['test_score'])
            test_performance[alpha,networksize] = avg_test_score
            print(f"{alpha=}, {networksize=}, {avg_test_score=}")
    best_alpha, best_networksize = min(test_performance,key=test_performance.get)
    print(f"Best Average RMSE: {test_performance[best_alpha, best_networksize] : .02f}\nBest network parameters
        are:\nalpha={best_alpha}\nLayers=[{', '.join(str(layer) for layer in best_networksize)}]")

#TODO: UPLOAD TO OVERLEAF, RE-RUN WITH PRUNED DATA
@logged
def q21_22():
    data_and_labels = get_all_data_and_labels_selected(pruning_for='forest')
    kf = KFold(n_splits=10)
    max_features = [0.2, 0.4, 0.6, 0.8, "auto", "sqrt", "log2"]
    number_trees = [10, 50, 100, 150, 200]
    max_depth = [1, 2, 3, 4, 5, 7]
    best_parameters = {}
    for i in max_features:
        for j in number_trees:
            for k in max_depth:
                model = RandomForestRegressor(n_estimators = j, max_depth = k, max_features= i)
                print(f"Random Forest for Max Number Features: {i}, Number of Trees: {j}, Depth of each Tree: {k}")
                for data_set_name, (features, target) in data_and_labels.items():
                    scores = cross_validate(model, features.values, target.values.ravel(),
                        scoring='neg_root_mean_squared_error', return_train_score=True, cv=kf)
                    avg_train_score = np.mean(-scores['train_score'])
                    avg_test_score = np.mean(-scores['test_score'])
                    # The smaller the RMSE, the better.
                    if data_set_name not in best_parameters: best_parameters[data_set_name] = (i, j, k, avg_train_score,
                        avg_test_score)
                    if best_parameters[data_set_name][4] > avg_test_score:
                        best_parameters[data_set_name] = (i, j, k, avg_train_score, avg_test_score)
                    print(f"{data_set_name}: Avg. Training RMSE Error: {avg_train_score}, Avg. Test RMSE Error:
                        {avg_test_score}")
                pass
    print("Best Max Number Features, Number of Trees, Depth of Each Tree for each Dataset")
    for key, value in best_parameters.items():
        print(key, value)
    pass

@logged
```

```
def q21_22_oob():
    data_and_labels = get_all_data_and_labels_selected(pruning_for='forest')
    bike_features, bike_targets = data_and_labels['bike_cnt']
    video_features, video_targets = data_and_labels['video_utime']
    suicide_features, suicide_targets = data_and_labels['suicide_per_100k']

    bike_model = RandomForestRegressor(n_estimators = 100, max_depth = 7, max_features= 0.8, oob_score=True)
    bike_model.fit(bike_features.values, bike_targets.values.ravel())

    suicide_model = RandomForestRegressor(n_estimators = 200, max_depth = 7, max_features= 0.8, oob_score=True)
    suicide_model.fit(suicide_features.values, suicide_targets.values.ravel())
    video_model = RandomForestRegressor(n_estimators = 50, max_depth = 7, max_features= 0.4, oob_score=True)
    video_model.fit(video_features.values, video_targets.values.ravel())


    print('OOB Error for each Dataset')
    print(f"Bike Dataset: {1 - bike_model.oob_score_}")
    print(f"Suicide Dataset: {1 - suicide_model.oob_score_}")
    print(f"Video Dataset: {1 - video_model.oob_score_}")

@logged
def q21_22_r():
    data_and_labels = get_all_data_and_labels_selected(pruning_for='forest')
    bike_features, bike_targets = data_and_labels['bike_cnt']
    video_features, video_targets = data_and_labels['video_utime']
    suicide_features, suicide_targets = data_and_labels['suicide_per_100k']

    bike_model = RandomForestRegressor(n_estimators = 100, max_depth = 7, max_features= 0.8, oob_score=True)
    bike_model.fit(bike_features.values, bike_targets.values.ravel())

    suicide_model = RandomForestRegressor(n_estimators = 200, max_depth = 7, max_features= 0.8, oob_score=True)
    suicide_model.fit(suicide_features.values, suicide_targets.values.ravel())
    video_model = RandomForestRegressor(n_estimators = 50, max_depth = 7, max_features= 0.4, oob_score=True)
    video_model.fit(video_features.values, video_targets.values.ravel())

    print('R^2 Error for each Dataset')
    print(f"Bike Dataset: {bike_model.score(bike_features.values, bike_targets.values.ravel())}")
    print(f"Suicide Dataset: {suicide_model.score(suicide_features.values, suicide_targets.values.ravel())}")
    print(f"Video Dataset: {video_model.score(video_features.values, video_targets.values.ravel())}")

    pass

@logged
def q23():
    data_and_labels = get_all_data_and_labels_selected(pruning_for='forest')
    kf = KFold(n_splits=10)
    max_features = [0.2, 0.4, 0.6, 0.8, "auto", "sqrt", "log2"]
    number_trees = [10, 50, 100, 150, 200]
    best_parameters = {}
    for i in max_features:
        for j in number_trees:
            model = RandomForestRegressor(n_estimators = j, max_depth = 4, max_features= i)
            print(f"Random Forest for Max Number Features: {i}, Number of Trees: {j}, Depth of each Tree: 4")
            for data_set_name, (features, target) in data_and_labels.items():
                scores = cross_validate(model, features.values, target.values.ravel(),
                    scoring='neg_root_mean_squared_error', return_train_score=True, cv=kf)
                avg_train_score = np.mean(-scores['train_score'])
                avg_test_score = np.mean(-scores['test_score'])
                # The smaller the RMSE, the better.
                if data_set_name not in best_parameters: best_parameters[data_set_name] = (i, j, avg_train_score,
                    avg_test_score)
                if best_parameters[data_set_name][3] > avg_test_score:
                    best_parameters[data_set_name] = (i, j, avg_train_score, avg_test_score)
                print(f"{data_set_name}: Avg. Training RMSE Error: {avg_train_score}, Avg. Test RMSE Error:
                    {avg_test_score}")
            pass
    print("Best Max Number Features, Number of Trees, Depth of Each Tree for each Dataset")
    for key, value in best_parameters.items():
        print(key, value)
    pass

@logged
def q23_plot():
    data_and_labels = get_all_data_and_labels_selected(pruning_for='forest')
    bike_features, bike_targets = data_and_labels['bike_cnt']
    video_features, video_targets = data_and_labels['video_utime']
    suicide_features, suicide_targets = data_and_labels['suicide_per_100k']
    bike_model = RandomForestRegressor(n_estimators = 100, max_depth = 4, max_features= 0.6)
    bike_model.fit(bike_features.values, bike_targets.values.ravel())
    suicide_model = RandomForestRegressor(n_estimators = 150, max_depth = 4, max_features= 0.6)
    suicide_model.fit(suicide_features.values, suicide_targets.values.ravel())
    video_model = RandomForestRegressor(n_estimators = 100, max_depth = 4, max_features= 'auto')
    video_model.fit(video_features.values, video_targets.values.ravel())
    dot_data = tree.export_graphviz(bike_model.estimators_[0],
                    feature_names=bike_features.columns,
                    filled=True, rounded=True,
                    special_characters=True,
                    out_file=None,
                    )
    graph = graphviz.Source(dot_data)
    graph.format = "png"
    graph.render("figures/q23_bike_tree.png")

    dot_data = tree.export_graphviz(suicide_model.estimators_[0],
                feature_names=suicide_features.columns,
                filled=True, rounded=True,
                special_characters=True,
                out_file=None,
                )
    graph = graphviz.Source(dot_data)
    graph.format = "png"
    graph.render("figures/q23_suicide_tree.png")

    dot_data = tree.export_graphviz(video_model.estimators_[0],
                feature_names=video_features.columns,
                filled=True, rounded=True,
                special_characters=True,
                out_file=None,
                )
    graph = graphviz.Source(dot_data)
    graph.format = "png"
    graph.render("figures/q23_video_tree.png")
    print("Decision Tree from Random Forest Trained on Bike Dataset")
    text_representation = tree.export_text(bike_model.estimators_[0], feature_names=list(bike_features.columns))
    print(text_representation)
    print("Decision Tree from Random Forest Trained on Suicide Dataset")
    text_representation = tree.export_text(suicide_model.estimators_[0], feature_names=list(suicide_features.columns))
    print(text_representation)
    print("Decision Tree from Random Forest Trained on Video Dataset")
    text_representation = tree.export_text(video_model.estimators_[0], feature_names=list(video_features.columns))
    print(text_representation)
    # fig = plt.figure(figsize=(25,20))
    # tree.plot_tree(bike_model.estimators_[0], feature_names=bike_features.columns, filled=True)
    # fig.savefig('figures/q23_bike_tree.png')
    # fig = plt.figure(figsize=(25,20))
    # tree.plot_tree(suicide_model.estimators_[0], feature_names=suicide_features.columns, filled=True)
    # fig.savefig('figures/q23_suicide_tree.png')
    # fig = plt.figure(figsize=(25,20))
    # tree.plot_tree(video_model.estimators_[0], feature_names=video_features.columns, filled=True)
    # fig.savefig('figures/q23_video_tree.png')

# LightGBM: https://lightgbm.readthedocs.io/en/latest/
# CatBoost: https://catboost.ai/

# LightGBM: https://lightgbm.readthedocs.io/en/latest/
# CatBoost: https://catboost.ai/
def q25():
    data_and_labels = get_all_data_and_labels_selected(pruning_for='forest')
    # TODO: pick one dataset, apply LightGBM and CatBoost.
    X, y = data_and_label=get_all_data_and_labels_selected()['bike_cnt']
    opt = BayesSearchCV(
        lgb.LGBMRegressor(),
        {
        'num_leaves': Integer(2, 50),
        'learning_rate':Real(0.01, 1, 'log-uniform'),
        'n_estimators': Integer(1, 100),
        'reg_alpha':Real(0.01, 1, 'log-uniform'),
        'subsample':Real(0.01, 1, 'log-uniform'),
        'feature_fraction':Real(0.01, 1, 'log-uniform')

        },
        scoring='neg_root_mean_squared_error',
        n_iter=32,
        cv=10
    )

    opt.fit(X, y)

    print("Avg. Test RMSE Error: %s" % -opt.best_score_)
    print("best_parameter: %s" % opt.best_params_)

    X, y = data_and_label=get_all_data_and_labels_selected()['bike_cnt']

    # log-uniform: understand as search over p = exp(x) by varying x
    opt = BayesSearchCV(
        CatBoostRegressor(),
        {
        'depth': Integer(1, 10),
        'learning_rate':Real(0.01, 1, 'log-uniform'),
        'n_estimators': Integer(1, 100),
        'l2_leaf_reg':Real(0.01, 1, 'log-uniform'),
        'bagging_temperature':Real(0.01, 1, 'log-uniform')
        },
        n_iter=32,
        cv=10
    )

    opt.fit(X, y)
    print("Avg. Test RMSE Error: %s" % -opt.best_score_)
    print("best_parameter: %s" % opt.best_params_)

def q26():
    f1 = plt.figure()
    score_test=[]
    score_train=[]
    parameter=[]
    for i in np.linspace(1, 100, 25, endpoint=False):
        X, y = data_and_label=get_all_data_and_labels_selected(pruning_for='forest')['bike_cnt']
        model=lgb.LGBMRegressor(n_estimators= int(i),learning_rate=0.15,max_depth=2,num_leaves=18,reg_alpha=0.01)
        scores = cross_validate(model, X, y, scoring='neg_root_mean_squared_error', return_train_score=True, cv=10)
        score_test.append(-np.average(scores['test_score']))
        score_train.append(-np.average(scores['train_score']))
        parameter.append(i)
    plt.plot( parameter,score_test,label='Test set')
    plt.plot( parameter,score_train, label = 'Training set')
    plt.ylabel("RMSE")
    plt.xlabel("n_estimators")
    plt.title("light_gbm")
    plt.legend()
    plt.savefig(f"figures/q26_gbm_estimator.pdf")

    f2 = plt.figure()
    score_test=[]
    score_train=[]
    parameter=[]
    for i in np.linspace(1, 100, 25, endpoint=False):
        X, y = data_and_label=get_all_data_and_labels_selected(pruning_for='forest')['bike_cnt']
        model=lgb.LGBMRegressor(n_estimators= 100,learning_rate=0.15,max_depth=int(i),num_leaves=18,reg_alpha=0.01)
        scores = cross_validate(model, X, y, scoring='neg_root_mean_squared_error', return_train_score=True, cv=10)
        score_test.append(-np.average(scores['test_score']))
        score_train.append(-np.average(scores['train_score']))
        parameter.append(i)
    plt.plot( parameter,score_test,label='Test set')
    plt.plot( parameter,score_train, label = 'Training set')
    plt.ylabel("RMSE")
    plt.xlabel("max_depth")
    plt.title("light_gbm")
    plt.legend()
    plt.savefig(f"figures/q26_gbm_max_depth.pdf")

    f3 = plt.figure()
    score_test=[]
    score_train=[]
    parameter=[]
    for i in np.linspace(1, 100, 25, endpoint=False):
        X, y = data_and_label=get_all_data_and_labels_selected(pruning_for='forest')['bike_cnt']
        model=lgb.LGBMRegressor(n_estimators= 100,learning_rate=0.15,max_depth=2,num_leaves=int(i),reg_alpha=0.01)
        scores = cross_validate(model, X, y, scoring='neg_root_mean_squared_error', return_train_score=True, cv=10)
        score_test.append(-np.average(scores['test_score']))
        score_train.append(-np.average(scores['train_score']))
        parameter.append(i)
    plt.plot( parameter,score_test,label='Test set')
    plt.plot( parameter,score_train, label = 'Training set')
    plt.ylabel("RMSE")
    plt.xlabel("num_leaves")
    plt.title("light_gbm")
    plt.legend()
    plt.savefig(f"figures/q26_gbm_num_leaves.pdf")
```

```
f4 = plt.figure()
score_test=[]
score_train=[]
parameter=[]
for i in np.logspace(-3, 0, 25, endpoint=False):
    X, y = data_and_label=get_all_data_and_labels_selected(pruning_for='forest')['bike_cnt']
    model=lgb.LGBMRegressor(n_estimators= 100,learning_rate=i,max_depth=2,num_leaves=18,reg_alpha=0.01)
    scores = cross_validate(model, X, y, scoring='neg_root_mean_squared_error', return_train_score=True, cv=10)
    score_test.append(-np.average(scores['test_score']))
    score_train.append(-np.average(scores['train_score']))
    parameter.append(i)
plt.plot( parameter,score_test,label='Test set')
plt.plot( parameter,score_train, label = 'Training set')
plt.ylabel("RMSE")
plt.xlabel("learning_rate")
plt.title("light_gbm")
plt.legend()
plt.savefig(f"figures/q26_gbm_learning_rate.pdf")

f5 = plt.figure()
score_test=[]
score_train=[]
parameter=[]
for i in np.logspace(-3, 0, 25, endpoint=False):
    X, y = data_and_label=get_all_data_and_labels_selected(pruning_for='forest')['bike_cnt']
    model=lgb.LGBMRegressor(n_estimators= 64,learning_rate=0.15,max_depth=2,num_leaves=18,reg_alpha=i)
    scores = cross_validate(model, X, y, scoring='neg_root_mean_squared_error', return_train_score=True, cv=10)
    score_test.append(-np.average(scores['test_score']))
    score_train.append(-np.average(scores['train_score']))
    parameter.append(i)
plt.plot( parameter,score_test,label='Test set')
plt.plot( parameter,score_train, label = 'Training set')
plt.ylabel("RMSE")
plt.xlabel("reg_alpha")
plt.title("light_gbm")
plt.legend()
plt.savefig(f"figures/q26_gbm_reg_alpha.pdf")


f6 = plt.figure()
score_test=[]
score_train=[]
parameter=[]
for i in np.linspace(1, 100, 25, endpoint=False):
    X, y = data_and_label=get_all_data_and_labels_selected(pruning_for='forest')['bike_cnt']
    model=CatBoostRegressor(n_estimators= int(i),bagging_temperature=1,depth=1,
        l2_leaf_reg=0.01,learning_rate=0.0824119,)
    scores = cross_validate(model, X, y, scoring='neg_root_mean_squared_error', return_train_score=True, cv=10)
    score_test.append(-np.average(scores['test_score']))
    score_train.append(-np.average(scores['train_score']))
    parameter.append(i)
plt.plot( parameter,score_test,label='Test set')
plt.plot( parameter,score_train, label = 'Training set')
plt.ylabel("RMSE")
plt.xlabel("n_estimators")
plt.title("CatBoost")
plt.legend()
plt.savefig(f"figures/q26_CatBoost_n_estimators.pdf")
```

```
f7 = plt.figure()
score_test=[]
score_train=[]
parameter=[]
for i in np.linspace(1, 100, 25, endpoint=False):
    X, y = data_and_label=get_all_data_and_labels_selected(pruning_for='forest')['bike_cnt']
    model=CatBoostRegressor(n_estimators= 64,bagging_temperature=1,depth=int(i),
        l2_leaf_reg=0.01,learning_rate=0.0824119,)
    scores = cross_validate(model, X, y, scoring='neg_root_mean_squared_error', return_train_score=True, cv=10)
    score_test.append(-np.average(scores['test_score']))
    score_train.append(-np.average(scores['train_score']))
    parameter.append(i)
plt.plot( parameter,score_test,label='Test set')
plt.plot( parameter,score_train, label = 'Training set')
plt.ylabel("RMSE")
plt.xlabel("depth")
plt.title("CatBoost")
plt.legend()
plt.savefig(f"figures/q26_CatBoost_depth.pdf")


f8 = plt.figure()
score_test=[]
score_train=[]
parameter=[]
for i in np.linspace(1, 100, 25, endpoint=False):
    X, y = data_and_label=get_all_data_and_labels_selected(pruning_for='forest')['bike_cnt']
    model=CatBoostRegressor(n_estimators= 64,bagging_temperature=int(i),depth=1,
        l2_leaf_reg=0.01,learning_rate=0.0824119,)
    scores = cross_validate(model, X, y, scoring='neg_root_mean_squared_error', return_train_score=True, cv=10)
    score_test.append(-np.average(scores['test_score']))
    score_train.append(-np.average(scores['train_score']))
    parameter.append(i)
plt.plot( parameter,score_test,label='Test set')
plt.plot( parameter,score_train, label = 'Training set')
plt.ylabel("RMSE")
plt.xlabel("bagging_temperature")
plt.title("CatBoost")
plt.legend()
plt.savefig(f"figures/q26_CatBoost_bagging_temperatures.pdf")


f9 = plt.figure()
score_test=[]
score_train=[]
parameter=[]
for i in np.logspace(-3, 0, 25, endpoint=False):
    X, y = data_and_label=get_all_data_and_labels_selected(pruning_for='forest')['bike_cnt']
    model=CatBoostRegressor(n_estimators= 100,learning_rate=i ,bagging_temperature=1,depth=1, l2_leaf_reg=0.01)
    scores = cross_validate(model, X, y, scoring='neg_root_mean_squared_error', return_train_score=True, cv=10)
    score_test.append(-np.average(scores['test_score']))
    score_train.append(-np.average(scores['train_score']))
    parameter.append(i)
plt.plot( parameter,score_test,label='Test set')
plt.plot( parameter,score_train, label = 'Training set')
plt.ylabel("RMSE")
plt.xlabel("learning_rate")
plt.title("CatBoost")
plt.savefig(f"figures/q26_CatBoost_learning_rate.pdf")
```

```
f10 = plt.figure()
score_test=[]
score_train=[]
parameter=[]
for i in np.logspace(-3, 0, 25, endpoint=False):
    X, y = data_and_label=get_all_data_and_labels_selected(pruning_for='forest')['bike_cnt']
    model=CatBoostRegressor(n_estimators= 100,learning_rate=0.0824119 ,bagging_temperature=1,depth=1, l2_leaf_reg=i)
    scores = cross_validate(model, X, y, scoring='neg_root_mean_squared_error', return_train_score=True, cv=10)
    score_test.append(-np.average(scores['test_score']))
    score_train.append(-np.average(scores['train_score']))
    parameter.append(i)
plt.plot( parameter,score_test,label='Test set')
plt.plot( parameter,score_train, label = 'Training set')
plt.ylabel("RMSE")
plt.xlabel("l2_leaf_reg=0.01")
plt.title("CatBoost")
plt.savefig(f"figures/q26_CatBoost_l2_leaf_reg.pdf")

f11 = plt.figure()
score_test=[]
score_train=[]
parameter=[]
for i in np.logspace(-3, 0, 25, endpoint=False):
    X, y = data_and_label=get_all_data_and_labels_selected(pruning_for='forest')['bike_cnt']
    model=lgb.LGBMRegressor(n_estimators= 64,learning_rate=i,max_depth=2,num_leaves=18,reg_alpha=0.01)
    scores = cross_validate(model, X, y, scoring='neg_root_mean_squared_error', return_train_score=True, cv=10)
    score_test.append(np.average(scores['fit_time']))
    score_train.append(np.average(scores['fit_time']))
    parameter.append(i)
plt.plot( parameter,score_test,)
plt.plot( parameter,score_train,)
plt.ylabel("Fitting time")
plt.xlabel("learning_rate")
plt.title("light_gbm")
plt.legend()
plt.savefig(f"figures/q26_gbm_fitting.pdf")


f12 = plt.figure()
score_test=[]
score_train=[]
parameter=[]
for i in np.logspace(-3, 0, 25, endpoint=False):
    X, y = data_and_label=get_all_data_and_labels_selected(pruning_for='forest')['bike_cnt']
    model=CatBoostRegressor(n_estimators= 100,learning_rate=i ,bagging_temperature=1,depth=1, l2_leaf_reg=0.01)
    scores = cross_validate(model, X, y, scoring='neg_root_mean_squared_error', return_train_score=True, cv=10)
    score_test.append(np.average(scores['fit_time']))
    score_train.append(np.average(scores['fit_time']))
    parameter.append(i)
plt.plot( parameter,score_test,)
plt.plot( parameter,score_train,)
plt.ylabel("Fitting time")
plt.xlabel("learning_rate")
plt.title("CatBoost")
plt.savefig(f"figures/q26_CatBoost__fitting.pdf")
```

```
if __name__=="__main__":
    q1_q2()
    q3()
    q4()
    q5()
    q5_one_plot()
    q6()
    q7()
    q8()
    q9()
    q10()
    q10_q11_13_linear()
    q10_q11_13_lasso()
    q10_q11_13_ridge()
    q10_q11_13_linear_shuffle()
    q10_q11_13_lasso_shuffle()
    q10_q11_13_ridge_shuffle()
    q12_linear()
    q12_lasso()
    q12_ridge()
    q14()
    q14_q15_16_poly()
    q17_18_19_20_NN()
    q18()
    q21_22()
    q21_22_oob()
    q21_22_r()
    q23()
    q23_plot()
    q25()
    q26()
    pass
```

helpers.py

```
import pandas as pd
import pycountry_convert as pc
import numpy as np
from sklearn.preprocessing import StandardScaler

def load_datasets():
    bike_df = pd.read_csv('input/Bike-Sharing-Dataset/day.csv')
    suicide_df = pd.read_csv('input/master.csv')
    suicide_df[' gdp_for_year ($) '] = suicide_df[' gdp_for_year ($) '].apply(lambda x: float(x.split()[0].replace(',',
        '')))
    video_df = pd.read_csv('input/online_video_dataset/transcoding_mesurment.tsv',sep='\t')
    return bike_df, suicide_df, video_df


def q4_helper():
    bike_df = pd.read_csv('input/Bike-Sharing-Dataset/day.csv', usecols=['yr', 'mnth', 'cnt'])
    year_column = bike_df['yr']
    max_year = year_column.max()
    month_column = bike_df['mnth']
    smallest_month = month_column.min()
    max_month = month_column.max()

    count_list = {}
    for i in range(max_year + 1):
        for j in range(smallest_month, max_month + 1, 3):
            bike_df_month = bike_df[(bike_df['mnth'] == j) & (bike_df['yr'] == i)]
            days_list = []
            for _, rows in bike_df_month.iterrows():
                days_list.append(rows['cnt'])
            count_list[i,j] = days_list

    return max_year, smallest_month, max_month, count_list

def q5_helper():
    suicide_df = pd.read_csv('input/master.csv')
    countries = suicide_df.groupby(['country'])
    ten_largest = (countries['year'].max()-countries['year'].min()).sort_values(ascending=False)[0:10].index.tolist()
    return suicide_df, ten_largest
    # agg_df = pd.DataFrame(countries['year'].max()).rename(columns={'year':'maxyear'}).merge(
        pd.DataFrame(countries['year'].min()).rename(columns={"year":"minyear"}),left_index=True,right_index=True)
    # agg_df['diff'] = agg_df['maxyear']-agg_df['minyear']



def q6_helper():
    video_df = pd.read_csv('input/online_video_dataset/transcoding_mesurment.tsv',sep='\t')
    video_df_utime = pd.DataFrame(video_df['utime'])
    return video_df_utime

def country_to_continent(country_name):
    if country_name == 'Republic of Korea':
        return 'Asia'
    if country_name == "Saint Vincent and Grenadines":
        return "North America"
    country_code = pc.country_name_to_country_alpha2(country_name)
    continent_code = pc.country_alpha2_to_continent_code(country_code)
    continent_name = pc.convert_continent_code_to_continent_name(continent_code)
    return continent_name

def encode_bike():
    bike_df = pd.read_csv('input/Bike-Sharing-Dataset/day.csv')
    bike_df['dteday'] = pd.to_datetime(bike_df['dteday'])
    bike_df['day_number'] = bike_df['dteday'].dt.day
    bike_df = bike_df.drop(['instant', 'dteday'], axis=1) # Zach: I removed 'casual' and 'registered' because we should
        not drop those! We have to do analysis with them...
    return bike_df

def encode_suicide():
    suicide_df = pd.read_csv('input/master.csv')
    continents = ['Asia', 'North America', 'Europe', 'South America', 'Africa', 'Oceania']
    suicide_df['continent'] = suicide_df['country'].apply(country_to_continent)
    y = pd.get_dummies(suicide_df['continent'], prefix="Continent")
    suicide_df = pd.concat([suicide_df, y], axis=1)
    sexkeys = {'male' : 0, 'female' : 1}
    suicide_df['sex'] = suicide_df['sex'].apply(lambda sex: sexkeys[sex])
    agekeys = {'5-14 years' :1,'15-24 years' :2,'25-34 years' :3,'35-54 years' :4,'55-74 years' :5,'75+ years' :6}
    suicide_df['age'] = suicide_df['age'].apply(lambda agerange: agekeys[agerange])
    genkeys = {'G.I. Generation' :1, 'Silent' :2, 'Boomers' :3, 'Generation X' :4, 'Millenials' :5, 'Generation Z' :6 }
    suicide_df['generation'] = suicide_df['generation'].apply(lambda generation: genkeys[generation])
    suicide_df = suicide_df[["Continent_Africa","Continent_Asia", "Continent_Europe", "Continent_North America",
        "Continent_Oceania", "Continent_South America","year", "sex", "age", "population", " gdp_for_year ($) ",
        "gdp_per_capita ($)", "generation", "suicides_no", "suicides/100k pop"]]
    return suicide_df
```

```
def encode_video():
    # Handling Categorical Features of Online Video Dataset
    video_df = pd.read_csv('input/online_video_dataset/transcoding_mesurment.tsv',sep='\t')
    y = pd.get_dummies(video_df['codec'], prefix='codec')
    video_df = pd.concat([video_df, y], axis=1)
    y = pd.get_dummies(video_df['o_codec'], prefix='o_codec')
    video_df = pd.concat([video_df, y], axis=1)
    video_df = video_df[['duration', 'codec_flv', 'codec_h264',
    'codec_mpeg4', 'codec_vp8', 'height', 'width', 'bitrate', 'framerate', 'i', 'p', 'b',
    'frames', 'i_size', 'p_size', 'b_size', 'size',
    'o_codec_flv', 'o_codec_h264', 'o_codec_mpeg4', 'o_codec_vp8',
    'o_bitrate', 'o_framerate', 'o_width', 'o_height', 'umem', 'utime']]
    return video_df




def q7_helper():
    return encode_bike(), encode_suicide(), encode_video()

def scale_bike():
    return encode_bike()

def scale_suicide():
    suicide_df_final = encode_suicide()
    scaler = StandardScaler()
    suicide_df_final[' gdp_for_year ($) '] = suicide_df_final[' gdp_for_year ($) '].apply(lambda x:
        float(x.split()[0].replace(',', '')))
    # Standardize feature columns of suicide data.
    suicide_df_final[['year', 'population', ' gdp_for_year ($) ', 'gdp_per_capita ($)']] =
        scaler.fit_transform(suicide_df_final[['year', 'population', ' gdp_for_year ($) ', 'gdp_per_capita ($)']])
    return suicide_df_final

def scale_video():
    video_df_final = encode_video()
    scaler = StandardScaler()
    video_df_final[['duration', 'height', 'width', 'bitrate','framerate', 'i', 'p', 'b', 'frames', 'i_size', 'p_size',
        'b_size','size', 'o_bitrate', 'o_framerate', 'o_width', 'o_height']] =
        scaler.fit_transform(video_df_final[['duration', 'height', 'width','bitrate', 'framerate', 'i', 'p', 'b',
        'frames', 'i_size', 'p_size', 'b_size', 'size','o_bitrate', 'o_framerate', 'o_width', 'o_height']])
    return video_df_final

def q8_helper():
    return scale_bike(), scale_suicide(), scale_video()

def separate_target(df,target,drop_other=[]):
    for other in drop_other:
        df = df.drop([other], axis=1)
    return df.drop(target, axis=1), df[[target]]


# def suicide_df_data_labels():
#     suicide_df_final = scale_suicide()
#     suicide_df_final_features = suicide_df_final.drop(['suicides_no', 'suicides/100k pop'], axis=1)
#     suicide_df_final_target = suicide_df_final[['suicides_no', 'suicides/100k pop']]
#     return suicide_df_final_features, suicide_df_final_target


# def video_df_data_labels():
#     video_df_final = scale_video()
#     video_df_final_features = video_df_final.drop(['umem', 'utime'], axis=1)
#     video_df_final_target = video_df_final[['umem', 'utime']]
#     return video_df_final_features, video_df_final_target


# def bike_df_data_labels():
#     bike_df_final = scale_bike()
#     return separate_target(bike_df_final,'cnt')

def bike_df_cnt_data_labels():
    bike_df_final = scale_bike()
    return separate_target(bike_df_final,'cnt',drop_other=['casual','registered'])

# def bike_df_registered_data_labels():
#     bike_df_final = scale_bike()
#     return separate_target(bike_df_final,'registered',drop_other=['cnt','casual'])

# def bike_df_casual_data_labels():
#     bike_df_final = scale_bike()
#     return separate_target(bike_df_final,'casual',drop_other=['cnt','registered'])



# def suicide_total_df_data_labels():
#     suicide_df_final = scale_suicide()
#     return separate_target(suicide_df_final,'suicides_no',drop_other=['suicides/100k pop'])


def suicide_per_100k_df_data_labels():
    suicide_df_final = scale_suicide()
    return separate_target(suicide_df_final,'suicides/100k pop',drop_other=['suicides_no'])


# def video_df_umem_data_labels():
#     video_df_final = scale_video()
#     return separate_target(video_df_final,'umem',drop_other=['utime'])

def video_df_utime_data_labels():
    video_df_final = scale_video()
    return separate_target(video_df_final,'utime',drop_other=['umem'])

def get_all_data_and_labels():
    return {'bike_cnt' : bike_df_cnt_data_labels(),'video_utime' : video_df_utime_data_labels(),'suicide_per_100k' :
        suicide_per_100k_df_data_labels()}

def bike_df_cnt_data_labels_no_scale():
    bike_df_final = encode_bike()
    return separate_target(bike_df_final,'cnt',drop_other=['casual','registered'])

def suicide_per_100k_df_data_labels_no_scale():
    suicide_df_final = encode_suicide()
    suicide_df_final[' gdp_for_year ($) '] = suicide_df_final[' gdp_for_year ($) '].apply(lambda x:
        float(x.split()[0].replace(',', '')))
    return separate_target(suicide_df_final,'suicides/100k pop',drop_other=['suicides_no'])

def video_df_utime_data_labels_no_scale():
    video_df_final = encode_video()
    return separate_target(video_df_final,'utime',drop_other=['umem'])

def get_all_data_and_labels_no_scale():
    return {'bike_cnt' : bike_df_cnt_data_labels_no_scale(),'video_utime' :
        video_df_utime_data_labels_no_scale(),'suicide_per_100k' : suicide_per_100k_df_data_labels_no_scale()}

def get_all_data_and_labels_selected(pruning_for='linear'):
    data_and_labels = get_all_data_and_labels()
    if pruning_for=='linear':
        drop = { 'bike_cnt' : ['day_number'], 'video_utime': ['codec_flv', 'duration', 'b_size', 'b' ],
            'suicide_per_100k': ['population', 'gdp_per_capita ($)']} # determined based on LASSO
    elif pruning_for=='nn':
        drop = { 'bike_cnt' : [], 'video_utime': [ ], 'suicide_per_100k': ['year', 'Continent_Oceania']}
    elif pruning_for=='forest':
        drop = { 'bike_cnt' : [], 'video_utime': [ ], 'suicide_per_100k': []}
    else:
        drop = {}
    for k in drop:
        features,target = data_and_labels[k]
        features.drop(drop[k], axis=1,inplace=True)
    return data_and_labels

def get_all_data_and_labels_selected_no_scale(pruning_for='linear'):
    data_and_labels = get_all_data_and_labels_no_scale()
    if pruning_for=='linear':
        drop = { 'bike_cnt' : ['day_number'], 'video_utime': ['codec_flv', 'duration', 'b_size', 'b' ],
            'suicide_per_100k': ['population', 'gdp_per_capita ($)']} # determined based on LASSO
    elif pruning_for=='nn':
        drop = { 'bike_cnt' : [], 'video_utime': [ ], 'suicide_per_100k': ['year', 'Continent_Oceania']}
    elif pruning_for=='forest':
        drop = { 'bike_cnt' : [], 'video_utime': [ ], 'suicide_per_100k': []}
    else:
        drop = {}
    for k in drop:
        features,target = data_and_labels[k]
        features.drop(drop[k], axis=1,inplace=True)
    return data_and_labels

def q12_helper():
    bike_df_final, suicide_df_final, video_df_final = q7_helper()
    suicide_df_final[' gdp_for_year ($) '] = suicide_df_final[' gdp_for_year ($) '].apply(lambda x:
        float(x.split()[0].replace(',', '')))
    bike_df_final_features = bike_df_final.drop('cnt', axis=1)
    bike_df_final_target = bike_df_final[['cnt']]
    suicide_df_final_features = suicide_df_final.drop(['suicides_no', 'suicides/100k pop'], axis=1)
    suicide_df_final_target = suicide_df_final[['suicides_no', 'suicides/100k pop']]
    video_df_final_features = video_df_final.drop(['umem', 'utime'], axis=1)
    video_df_final_target = video_df_final[['umem', 'utime']]

    return bike_df_final_features, bike_df_final_target, suicide_df_final_features, suicide_df_final_target,
        video_df_final_features, video_df_final_target
```

References

[1] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2:303–314, 1989.

[2] Dario Radecic. Top 3 methods for handling skewed data, Jan 2020.