Final Assignment

CSU44061 - Machine Learning

James Deasley

January 4th, 2023

1 Introduction

The initial data contains 7 scores which we are trying to predict: rating, accuracy, checkin, cleanliness, communication, location, and value. The initial rating scores are visualised in figure 1 to give a reference of how the data for all of these scores are skewed. They are all similarly skewed toward higher scores.

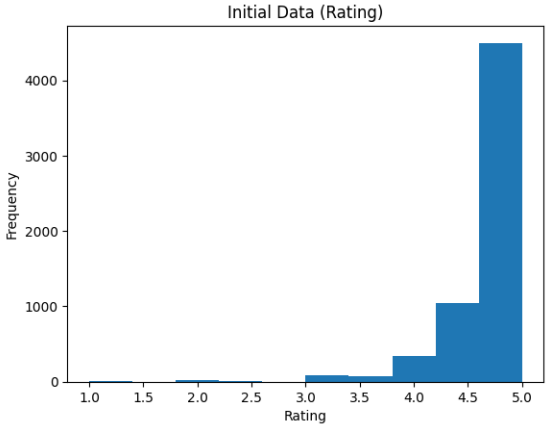


Figure 1: Example of initial target data (For Rating, just 1 of the 7 scores we are trying to predict.)

I extracted a few of the regular features from the listings dataset and combined them with text features from the review comments, then trained 2 machine learning models:

1. Ridge Regression model
2. k-Nearest Neighbours model

2 Feature Engineering

I went mostly with text features from the reviews, as I thought these would not only be very interesting to see but they are also available for most if not all listings, meaning we can get a representative sample. I did also add in a few extra features about the listing, namely whether or not the host is a superhost, the total number of listings the host has on AirBnB, the number of people the listing accommodates (as I thought it would be interesting to see if large groups or families review differently from individuals or couples.), and the total number of reviews the listing has.

I did some normalisation on each of these features, mapping ”t” and ”f” to 1 and 0 for the superhost feature, and using standard scaling (z-scores) on the rest of the numeric feature values.

Some of the troubles with text features, as mentioned in the assignment brief, were that not all reviews are in English, in fact there are many in a number of other languages, and some reviews may contain emojis and other non-conventional characters. Although admittedly it sounds lazy, I didn't do anything to get around these issues. After some thought, there are a few reasons why I thought they not only wouldn't be a big issue, but in fact may be helpful.

To start with the languages, if we are to get predictions based on a review, then we will eventually need to deal with reviews in other languages. I think it only makes sense to incorporate those other languages into the model with the training data. I also think it is interesting to leave other languages in because many languages, as long as they use the same alphabet, have similar stems for words with similar meanings. I thought it would be very interesting to see if stems which are used across languages had a bigger impact on the model.

Similarly, emojis and non-conventional characters, although they are non-conventional, still possess meaning. For example, a smiley-face emoji in a review I think would be a very good indicator of

whether or not the reviewer enjoyed their stay. An angry-face emoji makes it clear that they did not. I think most if not all characters will have some meaning, so I left these in too.

However, I didn't want to have thousands of features for every different language and character , making the model overly complex. So I controlled the number of text features using document frequency, which I will describe below.

I extracted my text features using a TF-IDF and Bag of Words approach. I started by tokenising and stemming the words in all of the reviews, then used a TF-IDF vectorizer on these stems. I used document frequency scores to control the number of features, setting a minimum DF of 0.06 and a maximum DF of 0.7, both of which were selected using k-fold cross-validation on a range of possible values, as seen in figure 2.

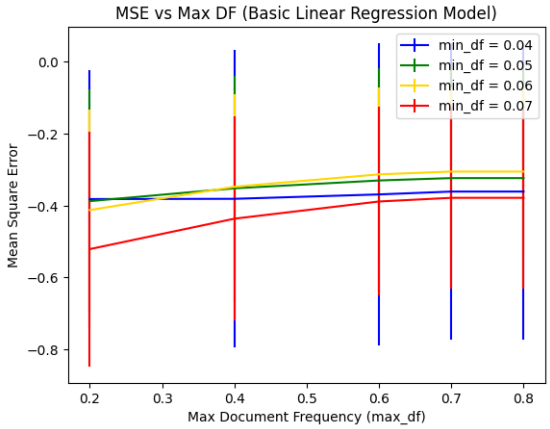


Figure 2: Negative Mean Square Error (MSE) vs Maximum Document Frequency (max_df) for a number of different Minimum Document Frequency values. Negative MSE, i.e. least error, peaks at min_df = 0.06 and max_df = 0.7. We will use these values going forward.

These were my initial features, which I worked with through to the point of model evaluation, however when I got to that point it became clear that, although the models had fairly small mean square error (around 0.11 stars on the overall listing rating for a Ridge regression model, compared to 0.16 for a baseline mean value predictor), the models did not perform very well on lower ratings, which are underrepresented in the dataset. It is also worth mentioning that the MSE values mentioned above were based on the underrepresentation of low ratings. The baselines performed noticeably worse when a more representative dataset was used, while I managed to keep the MSE of my models reasonably low.

To combat this, I split the data into rating ranges 1-2, 2-3, 3-4, and 4-5 (exclusive to inclusive, except for 1 which was also inclusive), and then did some resampling on these ranges, using 2000 samples from each range. I will note here that I am not particularly sure of the value of resampling when it comes to text features, as we are essentially copying reviews which I believe may have an impact on a KNN model specifically, which I did use. That said, resampling did seem to improve the performance of the model quite massively when it came to lower values ratings. You can see from figure 6 below that the KNN model in particular performed very well with the resampled data.

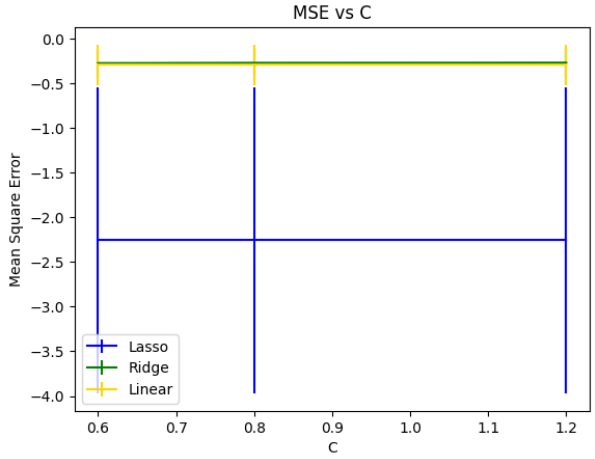
I opted not to use any polynomial features, as I felt the feature vector size was already quite large at 481 features, 477 of which were extracted from review text. Although 481 is large, I feel it is quite clear to understand what each feature is since each one is simply a word stem, whereas polynomial features would severely complicate feature meanings.

3 Methodology

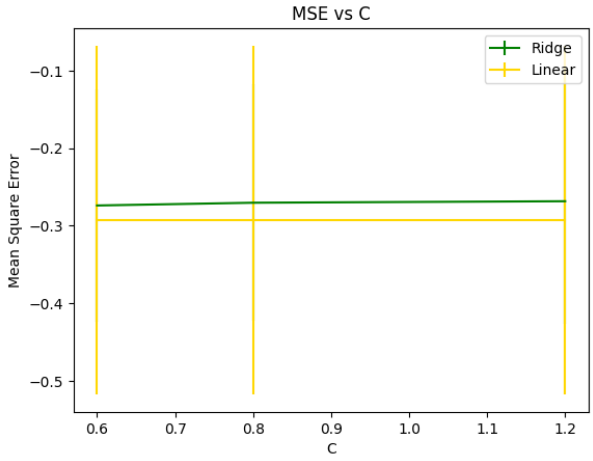
I decided to try 2 model types: linear regression, possibly with added regularisation making it Lasso or Ridge, as well as k-Nearest Neighbours. These made sense to me because this is a regression problem, so it wouldn't make much sense, I think to use a classification model. You could group ratings to make classes, for example ratings of 1-2, 2-3 ,3-4, and 4-5, but I think it is more interesting to predict the exact number, particularly because, as we know from the dataset, reviews tend to be skewed toward the 4-5 rating class, so I feel it would not be very interesting. For this reason, I wanted to do it as a regression problem. As for the kNN approach, I think this makes sense for the review text specifically. If 2 reviews are very similar, then I think it makes sense that the authors would have given similar ratings to

the listing, and so the listings review scores should be similar.

I also opted to use two baseline predictors: one which always predicts the mean value, and one which makes a constant prediction of 4, since the original data was skewed toward higher ratings, 4s and 5s.



(a) w/ Lasso model (zoomed out)



(b) w/o Lasso model (zoomed in)

Figure 3: Cross-validation to select regularisation penalty weight hyperparameter C

For my linear regression model, I started by comparing a no-regularisation model with Lasso (L_1 penalty) and Ridge (L_2 penalty) models, at varying values for the penalty-scaling hyperparameter C. I used cross-validation to select the optimal value of C for both models, and simultaneously compare their performance against each other and against the standard no-regularisation model. From the plot which can be seen in figure 3, you can see that Ridge regression with the L_2 penalty performed best, and did so at a value of around $C = 0.8$. I initially compared a wide range of values for C ranging from 0.01 up to 1000, and narrowed it down to this scale.

Next, for my kNN model I performed cross-validation to select a) the optimal number of neighbours k, and b) the optimal weight method for those neigbours, between uniform, distance-based, and Gaussian. This cross-validation can be seen in figure 4, which shows that the Gaussian weighting method performed best, and did so at a k value of 3. I also tested a small range of values for sigma in the Gaussian weighting method, but the large size of the dataset limited the range of values I could test. Regardless, 10 seemed to give the best performance.

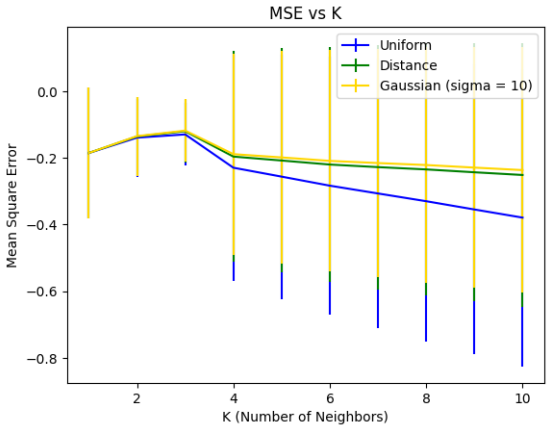


Figure 4: Cross-validation to select number of neighbours k, and weighting method, for kNN model

4 Evaluation

Figures 5 (below) and 6 (at the end) show the performance of all models at each target review score. You can see that both the Ridge and kNN models are fairly feasible predictors of review scores, with kNN performing slightly better. As this is as a regression problem, there are no true or false positive values with which to present ROC curves, so I have included the Mean Square Error (MSE) of each model next to it's name in the legend of each plot.

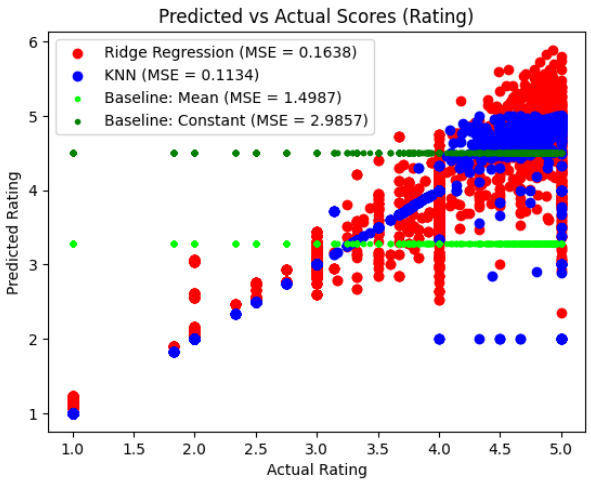


Figure 5: Rating

From the Ridge regression model, although it did not perform as well as the kNN model, I was able to extract some very interesting information about which features have an impact on review scores.

- Whether or not a host is a superhost seemed to have a small impact on overall rating (making approx. 0.1 stars of difference to the rating), accuracy (0.1 stars) cleanliness (0.2 stars), and value (0.2 stars).
- The number of listings the host has seemed to have an impact on communication (0.2 stars), inidicating that more experienced hosts have improved communication skills with their guests.
- Number of reviews on a listing also had a small impact on cleanliness (0.1 stars), which likely indicates that the owners learn what needs extra cleaning from reviews left on their listing.

No other regular features that I tested seemed to have a major impact. Presumably host response time would have some effect on communication and perhaps on checkin, but many listings were missing data for the response time column, so I didn't test it in order to avoid cutting out a lot of listings to maintain a representative sample size.

Some of the review text stems that have the most positive impact on the overall rating of a listing are as follows:

- "non" (+2.3 stars) - as in "second to none", "non-issue", "nonetheless", "non-touristy". (One of the properties also seems to be home to a dog named Nono, who may have had something of an impact on this)
- "propr" (+1.5 stars) - mostly from French word "propre", i.e. "clean". (This word also of course had a big impact on the cleanliness score.)
- "live" (+1.5 stars) - as in "lived up to", "lively", "live music". It also seems that people use the word "live" instead of the word "stay" when they feel like they've had a good time. As in they felt like they lived, rather than just survived.
- "sehr" (+1.4 stars) - from German, presumably "sehr gut", i.e. "very good". Probably does better than "gut" because something could also be "nicht gut", whereas people don't as often use the phrase "very bad", in favour of words like "terrible".

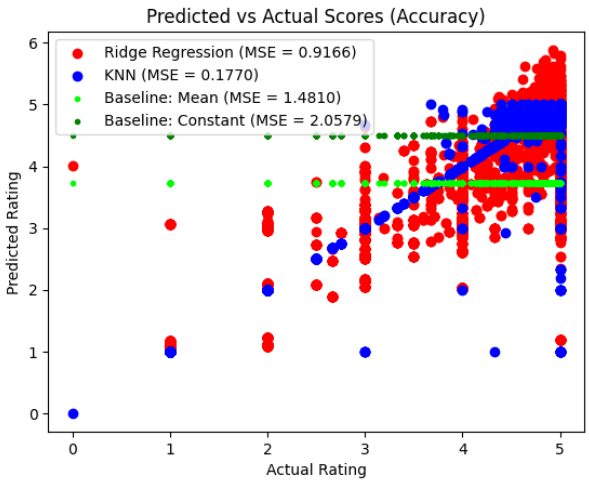
And some with the most negative impact are:

- "left" (-2.5 stars)
- "smell" (-2.3 stars)
- "run" (-2.0 stars)
- "rent" (-2.0 stars)

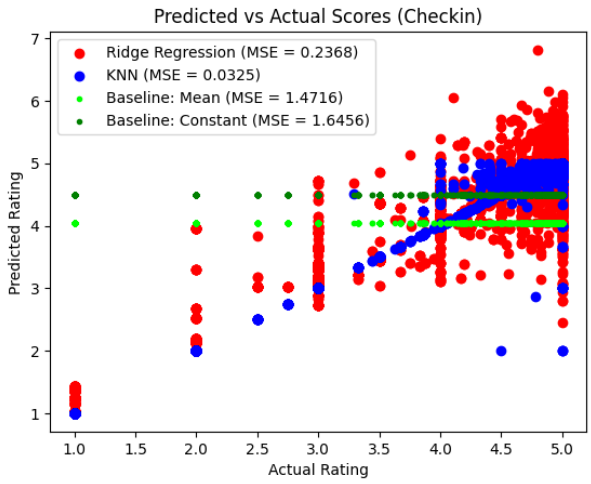
all of which seem fairly self-explanatory.

Impactful features for the other target ratings tend to see a lot of overlap with these, but some particularly interesting features are:

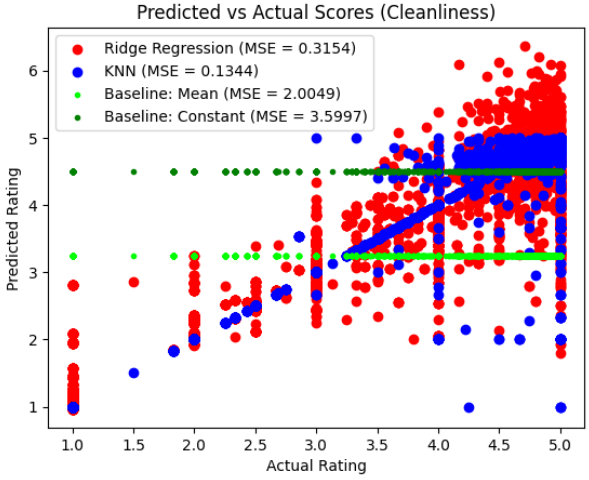
- Cleanliness - Negative impact: "smell" (-3.3 stars), "lack" (-2.9 stars), and "cleanli-" (-2.8 stars).
- Checkin - Positive impact: "late" (+2.3 stars), presumably for guests who arrived late but were still able to check-in.
- Location - Negative impact: "far" (-2.5 stars).
- Value - Positive impact: "pay" (+2.5 stars) and "free" (+2.1 stars). Negative impact: "bad" (-2.4 stars).



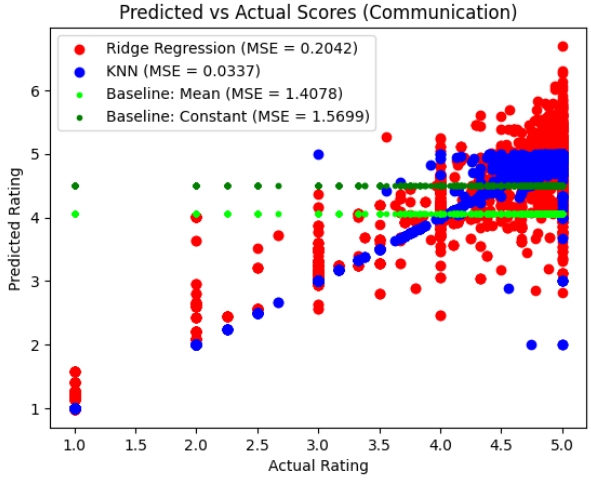
(a) Accuracy



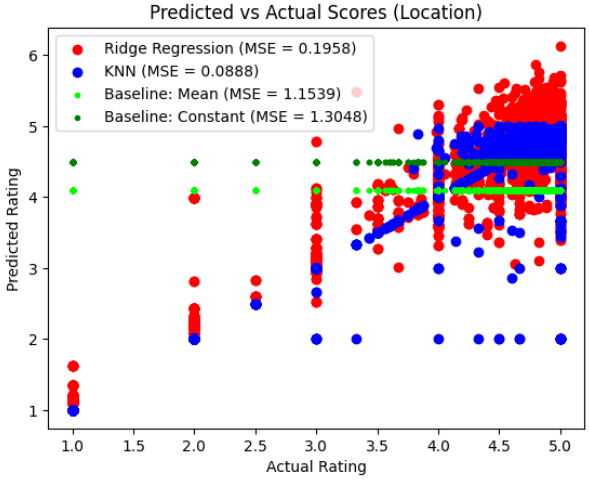
(b) Check-in



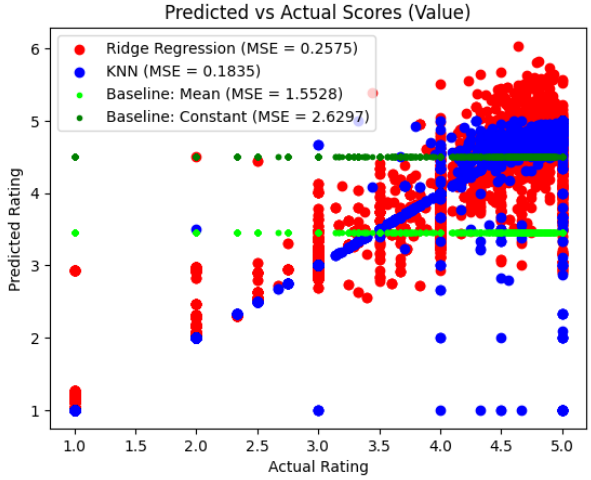
(c) Cleanliness



(d) Communication



(e) Location



(f) Value

Figure 6: Actual scores vs Predicted scores for each review score metric

5 Part 2 - Questions

(i) One situation that may cause inaccuracy in a logistic regression model is if we have imbalanced training data. If one class is far more common in our training data, then our model may lean toward predicting that more common classification purely because it is common, which is not useful to us when we are trying to predict a classification based on real data. In this case, a logistic regression model may not be any better than a baseline model which always predicts the most common classification.

Another situation that may cause inaccuracy is if there isn't a linear relationship between our input features and the target we wish to predict, since logistic regression looks for linear relationships similarly to linear regression. However, this issue could be worked around using feature engineering to find certain nonlinear relationships.

(ii) One advantage of a kNN classifier over an MLP neural net classifier is that typically a kNN classifier doesn't need as much training as an MLP does. Since a kNN classifier makes direct use of it's training data, it doesn't need to train weights or biases like an MLP, it simply finds the samples in the training data which are closest to the features of the input we want a prediction for.

In the same vein as this, a kNN classifier can also work quite well with a small training dataset, since it doesn't need a large amount of data to train on. Conversely, if we do have a large dataset, an MLP classifier might be better at learning complex relationships between input features and outputs.

Another practical advantage to kNN over MLP is interpretability. It is very clear what a kNN classifier is doing, whereas an MLP classifier may get highly complex and difficult to understand and interpret as a result.

(iii) Resampling gives us multiple estimates of cost function, allowing us to estimate the average and spread of values. The aim is to work around noise in the data by changing which parts of the data we use for training and testing in k models, and then taking the average of the prediction accuracy of all the models, effectively smoothing out the noise.

The larger the value of k we use the more representative our cross-validation will be. However, since we are fitting a new model k times, we don't want k to be so large that it is computationally unreasonable. The values k = 5 and k = 10 are often suggested as good choices because they are considered to strike a reasonably good balance in this trade-off.

(iv) Sometimes with time series data it can be useful to take the predictions of our model as feedback to make predictions even further ahead. We need to be careful doing this since our predictions are, as always, subject to error, which will build up the more we rely on this feedback. The further ahead we look with a feedback model like this, the less accurate we would expect our predictions to become.

As an example of this, suppose it is a Tuesday and we want to predict the weather for the coming Friday. We only have the actual weather data up to Tuesday, so we may start by predicting the weather for Wednesday. We can then propagate this prediction forward to use as an extra feature in order to predict the weather for Thursday, and then similarly propagate our predictions for both Wednesday and Thursday forward to use as features in our prediction for Friday. However, it might be unreasonable to use this method to make a prediction about a Tuesday 2 months from now, since all of the data in between will be predicted, not real data, and thus subject to error. We wouldn't expect such a prediction to be very accurate.

Appendix A Code

```
# %% [markdown]
# # Data setup and cleaning

# %%
import pandas as pd
import numpy as np

# %%
# HELPER FUNCTIONS - DATA CLEANING

import re
HMTL_TAG = re.compile('<.*?>')
WHITESPACE = re.compile(r'\s+')

def replace_html_tags(raw_html, replace_with = ''):
  clean_text = re.sub(HMTL_TAG, replace_with, str(raw_html))
  return clean_text

def replace_whitespace(raw_text, replace_with = '␣'):
  clean_text = re.sub(WHITESPACE, replace_with, str(raw_text))
  return clean_text

def clean_text(raw_text):
  clean_text = replace_html_tags(raw_text)
  clean_text = replace_whitespace(clean_text)
  return clean_text

# Clean text for every row in a pandas dataframe column. Returns new dataframe with
    cleaned text.
def clean_text_batch(df: pd.DataFrame, col: int) -> pd.DataFrame:
  df.iloc[:, col] = df.iloc[:, col].apply(lambda x: clean_text(x))
  return df

# %%
# HELPER FUNCTIONS - DATA ANALYSIS / INTERPRETATION

LISTING_FIELDS = ['id','listing_url','scrape_id','last_scraped','source','name','
   description','neighborhood_overview','picture_url','host_id','host_url','
   host_name','host_since','host_location','host_about','host_response_time','
   host_response_rate','host_acceptance_rate','host_is_superhost','
   host_thumbnail_url','host_picture_url','host_neighbourhood','host_listings_count
   ','host_total_listings_count','host_verifications','host_has_profile_pic','
   host_identity_verified','neighbourhood','neighbourhood_cleansed','
   neighbourhood_group_cleansed','latitude','longitude','property_type','room_type'
   ,'accommodates','bathrooms','bathrooms_text','bedrooms','beds','amenities','
   price','minimum_nights','maximum_nights','minimum_minimum_nights','
   maximum_minimum_nights','minimum_maximum_nights','maximum_maximum_nights','
   minimum_nights_avg_ntm','maximum_nights_avg_ntm','calendar_updated','
   has_availability','availability_30','availability_60','availability_90','
   availability_365','calendar_last_scraped','number_of_reviews','
   number_of_reviews_ltm','number_of_reviews_l30d','first_review','last_review','
   review_scores_rating','review_scores_accuracy','review_scores_cleanliness','
   review_scores_checkin','review_scores_communication','review_scores_location','
   review_scores_value','license','instant_bookable','
```

```
    calculated_host_listings_count','calculated_host_listings_count_entire_homes','
    calculated_host_listings_count_private_rooms','
    calculated_host_listings_count_shared_rooms','reviews_per_month']
REVIEW_FIELDS = ['listing_id','id','date','reviewer_id','reviewer_name','comments']

# function to quickly return index of a given string in LISTING_FIELDS
def index_in_listings(field: str) -> int:
    return LISTING_FIELDS.index(field)

# function to quickly return index of a given string in REVIEW_FIELDS
def index_in_reviews(field: str) -> int:
    return REVIEW_FIELDS.index(field)

# function to map a string reading 't' or 'f' to a boolean
def as_bool(t_f: str) -> bool:
    if t_f == 't':
        return True
    elif t_f == 'f':
        return False
    else:
        raise ValueError('Invalid value for boolean: ' + t_f)

# function to normalise a numpy array to floats between 0 and 1
def normalise(arr: np.ndarray) -> np.ndarray:
    return arr / arr.sum()



# %%
# CLEAN LISTINGS TEXT AND SAVE TO NEW CSV
df_listings = pd.read_csv("../data/listings.csv")

# print(df_listings.applymap(lambda x: isinstance(x, str)).all(0))

# Names and indices of columns to clean
COLS_TO_CLEAN = {
    'description': 6,
    'neighborhood_overview': 7,
    'host_about': 14
}

for col in COLS_TO_CLEAN:
    df_listings = clean_text_batch(df_listings, COLS_TO_CLEAN[col])

# df_listings.to_csv('../data/listings_clean.csv', index=False)

# %%
# SELECT ONLY TARGET COLUMNS FROM LISTINGS -> SAVE TO NEW CSV

df_listings = pd.read_csv("../data/listings.csv")

# col names: review_scores_rating,review_scores_accuracy,review_scores_cleanliness,
    review_scores_checkin,review_scores_communication,review_scores_location,
    review_scores_value
LISTING_COLS_TO_RETAIN = {
    'id': index_in_listings('id'),
```

```
    'review_scores_rating': index_in_listings('review_scores_rating'),
    'review_scores_accuracy': index_in_listings('review_scores_accuracy'),
    'review_scores_cleanliness': index_in_listings('review_scores_cleanliness'),
    'review_scores_checkin': index_in_listings('review_scores_checkin'),
    'review_scores_communication': index_in_listings('review_scores_communication')
        ,
    'review_scores_location': index_in_listings('review_scores_location'),
    'review_scores_value': index_in_listings('review_scores_value')
}

df_listings = df_listings.iloc[:, list(LISTING_COLS_TO_RETAIN.values())]
df_listings = df_listings.dropna()
df_listings = df_listings.sort_values(by=['id'])
df_listings.to_csv('../data/listings_targets.csv', index=False)

# NOTE: Some listings do not have some or all of the review scores. We will need to
    account for this when we train our model.

# print(df_listings.applymap(lambda x: isinstance(x, str)).all(0))


# %%
# CLEAN REVIEWS TEXT AND SELECT ONLY KEY COLUMNS -> SAVE TO NEW CSV

df_reviews = pd.read_csv("../data/reviews.csv")
df_reviews = clean_text_batch(df_reviews, 5)

REVIEW_COLS_TO_RETAIN = {
    'listing_id': index_in_reviews('listing_id'),
    'comments': index_in_reviews('comments')
}

df_reviews = df_reviews.iloc[:, list(REVIEW_COLS_TO_RETAIN.values())]

# Sort by listing_id, so reviews for a given listing are grouped together
df_reviews = df_reviews.sort_values(by=['listing_id'])
df_reviews.to_csv('../data/reviews_skeleton.csv', index=False)


# %%
comments = df_reviews.iloc[:, 5].values
print(comments.shape)
print(comments[0])

# %% [markdown]
# We are trying to predict the ratings of a listing based on reviews for that
    listing.
#
# So in order to train the model, we need to group all reviews for a particular
    listing with the appropriate listing id.

# %% [markdown]
# # Features and Targets

# %%
df_listings = pd.read_csv("../data/listings.csv")
```

```
# col names: review_scores_rating,review_scores_accuracy,review_scores_cleanliness,
   review_scores_checkin,review_scores_communication,review_scores_location,
   review_scores_value
LISTING_COLS_TO_RETAIN = {
    'id': index_in_listings('id'),
    'review_scores_rating': index_in_listings('review_scores_rating'),
    'review_scores_accuracy': index_in_listings('review_scores_accuracy'),
    'review_scores_cleanliness': index_in_listings('review_scores_cleanliness'),
    'review_scores_checkin': index_in_listings('review_scores_checkin'),
    'review_scores_communication': index_in_listings('review_scores_communication')
       ,
    'review_scores_location': index_in_listings('review_scores_location'),
    'review_scores_value': index_in_listings('review_scores_value')
}

targets = df_listings.iloc[:, list(LISTING_COLS_TO_RETAIN.values())]
targets = targets.dropna()
targets = targets.sort_values(by=['id'])

targ_ratings = targets.iloc[:, 1].values
targ_acc = targets.iloc[:, 2].values
targ_clean = targets.iloc[:, 3].values
targ_checkin = targets.iloc[:, 4].values
targ_comm = targets.iloc[:, 5].values
targ_loc = targets.iloc[:, 6].values
targ_value = targets.iloc[:, 7].values

# Save targets to csv
# targets.to_csv('../data/listings_targets_comp.csv', index=False)

# %%
print(targets.shape)

# %%
# Group review comments by listing_id
reviews = pd.read_csv("../data/reviews_skeleton.csv")
reviews = reviews.dropna()
reviews = reviews.sort_values(by=['listing_id'])
reviews_by_listing = reviews.groupby('listing_id').agg({'comments': '␣'.join})
reviews_by_listing = reviews_by_listing.reset_index()
reviews_by_listing.to_csv('../data/reviews_by_listing.csv', index=False)

# Tokenise and stem reviews
from nltk.tokenize import word_tokenize
from nltk.stem import PorterStemmer

stemmer = PorterStemmer()

# Function to map each string of comments to stems
def stem_review(review):
    tokens = word_tokenize(review)
    stems = [stemmer.stem(token) for token in tokens]
    return '␣'.join(stems)

reviews_by_listing['stems'] = reviews_by_listing['comments'].apply(stem_review)
```

```
stems_by_listing = reviews_by_listing[['listing_id', 'stems']]

# for index, row in reviews_by_listing.iterrows():
#     listing_id = row['listing_id']
#     comments = row['comments']
#     tokens = word_tokenize(comments)
#     stems = [stemmer.stem(token) for token in tokens]
#     ' '.join(stems)
#     stems_by_listing.append([listing_id, stems])

# stems_by_listing = pd.DataFrame(stems_by_listing, columns=['listing_id', 'stems
    '])

# Save stems to csv
stems_by_listing.to_csv('../data/stems_by_listing.csv', index=False)



# %%
# SELECT ONLY LISTINGS WITH BOTH REVIEWS AND TARGETS
reviews = pd.read_csv("../data/stems_by_listing.csv")
targets = pd.read_csv("../data/listings_targets_comp.csv")

# Sort both by listing_id
reviews = reviews.sort_values(by=['listing_id'])
targets = targets.sort_values(by=['id'])

# Drop rows where listing_id is not included in both
reviews = reviews[reviews['listing_id'].isin(targets['id'])]
targets = targets[targets['id'].isin(reviews['listing_id'])]


# %%
# Should have same number of rows (but different number of columns)
print(reviews.shape)
print(targets.shape)

# %%
stemmed_comments = reviews['stems'].values

import nltk
nltk.download('punkt')
nltk.download('stopwords')

from sklearn.feature_extraction.text import TfidfVectorizer

# df_max - exclude words which appear in too many documents
# df_min - exclude words which appear in too few documents
# Use cross validation to determine best values for these parameters.
# HAVE TO USE MIN_DF, otherwise feature vectors are:
# a) too large for my computer to handle.
# b) too full of zeros to be useful.
# vectorizer = TfidfVectorizer(stop_words='english', max_df=0.2, min_df=0.001) #
    93,193 features
```

```
vectorizer = TfidfVectorizer(stop_words='english') # 79,179 features for 6,078
    listings

# A single doc will be a list of sentences, in this case a single review.
# Each sentence will be a list of words, or in this case tokens, which have been
    stemmed.
X = vectorizer.fit_transform(stemmed_comments)
print(len(vectorizer.get_feature_names_out()))
print(X.shape)

# %%
Y = targets.iloc[:, 1:].values
print(Y.shape)

# Split into different categories
Y_rating = Y[:, 0]
Y_acc = Y[:, 1]
Y_clean = Y[:, 2]
Y_checkin = Y[:, 3]
Y_comm = Y[:, 4]
Y_loc = Y[:, 5]
Y_value = Y[:, 6]


# %%
counts, bins = np.histogram(Y_rating)
plt.stairs(counts, bins, fill=True)

plt.xlabel('Rating')
plt.ylabel('Frequency')
plt.title('Initial␣Data␣(Rating)')
plt.show()

# %%
print(X.shape)
print(X.shape[0])

# %% [markdown]
# Other features

# %%
listings = pd.read_csv("../data/listings_clean.csv")

# Pull out interesting features
FEATURES_OF_INTEREST = {
    'id': index_in_listings('id'),
    'host_is_superhost': index_in_listings('host_is_superhost'),
    'host_total_listings_count': index_in_listings('host_total_listings_count'),
    'accommodates': index_in_listings('accommodates'),
    'number_of_reviews': index_in_listings('number_of_reviews')
}

# Create a new dataframe with only the features of interest, and save to new csv
listings_extra_features = listings.iloc[:, list(FEATURES_OF_INTEREST.values())]
listings_extra_features.to_csv('../data/listings_extra_features.csv', index=False)
```

```
# %% [markdown]
# # Text Feature Extraction
#

# %%
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import KFold, cross_val_score
from sklearn.feature_extraction.text import TfidfVectorizer

colors = ['blue', 'green', 'gold', 'red', 'pink']

# df_min - exclude words which appear in too few documents
min_df_range = [0.2, 0.4, 0.5, 0.6]
# df_max - exclude words which appear in too many documents
max_df_range = [0.8, 0.9, 1.0]

for min_df_index, min_df in enumerate(min_df_range):
    print(f'min_df = {min_df}')

    mean_score=[]
    std_score=[]

    for max_df in max_df_range:
        print(f' - max_df = {max_df}')

        vectorizer = TfidfVectorizer(stop_words='english', min_df=min_df, max_df=
            max_df) # 79,179 features for 6,078 listings
        X = vectorizer.fit_transform(stemmed_comments)

        model = LinearRegression()
        kf = KFold(n_splits=5)

        scores = cross_val_score(model, X, Y_rating, cv=kf, scoring='
            neg_mean_squared_error')
        mean_score.append(np.array(scores).mean())
        std_score.append(np.array(scores).std())

    plt.errorbar(max_df_range,mean_score,yerr=std_score,color=colors[min_df_index],
        label=f'min_df = {min_df}')

plt.title('MSE vs Max DF (Basic Linear Regression Model)')
plt.xlabel('Max Document Frequency (max_df)')
plt.ylabel('Mean Square Error')
plt.legend()
plt.show()

# %%
# Set min_df and max_df to the values that gave the best results (lowest MSE)
MIN_DF = 0.4
MAX_DF = 0.9

vectorizer = TfidfVectorizer(stop_words='english', min_df=MIN_DF, max_df=MAX_DF) #
    79,179 features for 6,078 listings
```

```
X = vectorizer.fit_transform(stemmed_comments)


# %%
def normalise_feat(feat):
    """Normalise a feature so that it has mean 0 and standard deviation 1."""
    return np.log(feat) / (1 + np.log(feat))

# %%
print(X.shape)

# %% [markdown]
# # Modelling
# Now we have our text features and our targets ready to go, we can begin on our
    model.

# %%
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression, Lasso, Ridge
from sklearn.model_selection import KFold, cross_val_score
from sklearn.metrics import mean_squared_error
from sklearn.feature_extraction.text import TfidfVectorizer

colors = ['blue', 'green', 'gold', 'red', 'pink']
c_range = [0.1, 0.15, 0.2, 0.25, 0.3]
# c_range = [2000, 4000, 6000, 8000, 10000]

m_linear = LinearRegression()

me_lasso = []
me_ridge = []
me_linear = []

se_lasso = []
se_ridge = []
se_linear = []

for c in c_range:
    print(f'C = {c}')

    m_lasso = Lasso(alpha=1/c)
    m_ridge = Ridge(alpha=1/c)

    kf = KFold(n_splits=5)

    errors_lasso = cross_val_score(m_lasso, X, Y_rating, cv=kf, scoring='
        neg_mean_squared_error')
    errors_ridge = cross_val_score(m_ridge, X, Y_rating, cv=kf, scoring='
        neg_mean_squared_error')
    errors_linear = cross_val_score(model, X, Y_rating, cv=kf, scoring='
        neg_mean_squared_error')

    me_lasso.append(np.array(errors_lasso).mean())
    me_ridge.append(np.array(errors_ridge).mean())
    me_linear.append(np.array(errors_linear).mean())
```

```
    se_lasso.append(np.array(errors_lasso).std())
    se_ridge.append(np.array(errors_ridge).std())
    se_linear.append(np.array(errors_linear).std())

plt.errorbar(c_range,me_lasso,yerr=se_lasso,color=colors[0],label=f'Lasso')
plt.errorbar(c_range,me_ridge,yerr=se_ridge,color=colors[1],label=f'Ridge')
plt.errorbar(c_range,me_linear,yerr=se_linear,color=colors[2],label=f'Linear')

plt.title('MSE␣vs␣C')
plt.xlabel('C')
plt.ylabel('Mean␣Square␣Error')
plt.legend()
plt.show()

# %% [markdown]
# Standard Linear Regression (no regularization) works best. (Don't even really
    need to include this plot, since don't need cross-validation to select any C
    with no penalty)

# %%
def gaussian_kernel100(distances):
    weights = np.exp(-100*(distances**2))
    return weights/np.sum(weights)

def gaussian_kernel1000(distances):
    weights = np.exp(-1000*(distances**2))
    return weights/np.sum(weights)

def gaussian_kernel5000(distances):
    weights = np.exp(-5000*(distances**2))
    return weights/np.sum(weights)


# %%
# CROSS-VALIDATION TO GET K FOR KNN MODEL

from sklearn.neighbors import KNeighborsRegressor

k_range = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Want a model with uniform weights, a model with distance weights, and a model for
    each of the three kernels
# And perform cross-validation on each of these models to get the best k

me_unif = []
me_dist = []
me_gaus100 = []
me_gaus1000 = []
# me_gaus5000 = []

se_unif = []
se_dist = []
se_gaus100 = []
se_gaus1000 = []
# se_gaus5000 = []
```

```
for k in k_range:
    print(f'k = {k}')

    # Model with uniform weights, a model with distance weights, and a model for
        each of the three kernels
    m_unif = KNeighborsRegressor(n_neighbors=k, weights='uniform')
    m_dist = KNeighborsRegressor(n_neighbors=k, weights='distance')
    m_gaus100 = KNeighborsRegressor(n_neighbors=k, weights=gaussian_kernel100)
    m_gaus1000 = KNeighborsRegressor(n_neighbors=k, weights=gaussian_kernel1000)
    # m_gaus5000 = KNeighborsRegressor(n_neighbors=k, weights=gaussian_kernel5000)

    kf = KFold(n_splits=5)

    errors_unif = cross_val_score(m_unif, X, Y_rating, cv=kf, scoring='
        neg_mean_squared_error')
    errors_dist = cross_val_score(m_dist, X, Y_rating, cv=kf, scoring='
        neg_mean_squared_error')
    errors_gaus100 = cross_val_score(m_gaus100, X, Y_rating, cv=kf, scoring='
        neg_mean_squared_error')
    errors_gaus1000 = cross_val_score(m_gaus1000, X, Y_rating, cv=kf, scoring='
        neg_mean_squared_error')
    # errors_gaus5000 = cross_val_score(m_gaus5000, X, Y_rating, cv=kf, scoring='
        neg_mean_squared_error')

    me_unif.append(np.array(errors_unif).mean())
    me_dist.append(np.array(errors_dist).mean())
    me_gaus100.append(np.array(errors_gaus100).mean())
    me_gaus1000.append(np.array(errors_gaus1000).mean())
    # me_gaus5000.append(np.array(errors_gaus5000).mean())

    se_unif.append(np.array(errors_unif).std())
    se_dist.append(np.array(errors_dist).std())
    se_gaus100.append(np.array(errors_gaus100).std())
    se_gaus1000.append(np.array(errors_gaus1000).std())
    # se_gaus5000.append(np.array(errors_gaus5000).std())

plt.errorbar(k_range,me_unif,yerr=se_unif,color=colors[0],label=f'Uniform')
plt.errorbar(k_range,me_dist,yerr=se_dist,color=colors[1],label=f'Distance')
plt.errorbar(k_range,me_gaus100,yerr=se_gaus100,color=colors[2],label=f'Gaussian (
    sigma = 100)')
plt.errorbar(k_range,me_gaus1000,yerr=se_gaus1000,color=colors[3],label=f'Gaussian
    (sigma = 1000)')
# plt.errorbar(k_range,me_gaus5000,yerr=se_gaus5000,color=colors[4],label=f'
    Gaussian (sigma = 5000)')

plt.title('MSE vs K')
plt.xlabel('K (Number of Neighbors)')
plt.ylabel('Mean Square Error')
plt.legend()
plt.show()


# %% [markdown]
# k = 6 neighbours w/ uniform weighting seems best

# %%
```

```
print(X[0])

# %%
# COMPARE PERFORMANCE OF LINEAR AND KNN MODELS, ALONG WITH BASELINE PREDICTORS

# First get training and test data indices
from sklearn.model_selection import train_test_split
indices = np.arange(X.shape[0])
train, test = train_test_split(indices, test_size=0.2, random_state=0)


from sklearn.linear_model import LinearRegression
from sklearn.dummy import DummyRegressor
from sklearn.metrics import roc_curve, auc

# Linear Regression
m_linear = LinearRegression()
m_linear.fit(X.toarray()[train], Y_rating[train])

# KNN
m_knn = KNeighborsRegressor(n_neighbors=6, weights='uniform')
m_knn.fit(X.toarray()[train], Y_rating[train])

# Baseline predictors (mean value and constant 4)
m_mean = DummyRegressor(strategy='mean')
m_mean.fit(X.toarray()[train], Y_rating[train])
m_const = DummyRegressor(strategy='constant', constant=4)
m_const.fit(X.toarray()[train], Y_rating[train])

# Get predictions for all models
pred_linear = m_linear.predict(X.toarray()[test])
pred_knn = m_knn.predict(X.toarray()[test])
pred_mean = m_mean.predict(X.toarray()[test])
pred_const = m_const.predict(X.toarray()[test])

# Calculate MSE for all models
from sklearn.metrics import mean_squared_error
mse_linear = mean_squared_error(Y_rating[test], pred_linear)
mse_knn = mean_squared_error(Y_rating[test], pred_knn)
mse_mean = mean_squared_error(Y_rating[test], pred_mean)
mse_const = mean_squared_error(Y_rating[test], pred_const)

print(f'MSE␣for␣Linear␣Regression:␣{mse_linear}')
print(f'MSE␣for␣KNN:␣{mse_knn}')
print(f'MSE␣for␣Mean:␣{mse_mean}')
print(f'MSE␣for␣Constant:␣{mse_const}')

# Visualise performance of all models
plt.scatter(Y_rating[test], pred_linear, color=colors[0], label='Linear␣Regression'
    )
plt.scatter(Y_rating[test], pred_knn, color=colors[1], label='KNN')
plt.scatter(Y_rating[test], pred_mean, color=colors[2], label='Mean')
plt.scatter(Y_rating[test], pred_const, color=colors[3], label='Constant')

plt.title('Predicted␣vs␣Actual␣Ratings')
plt.xlabel('Actual␣Rating')
```

```
plt.xlim(2,5)
plt.ylabel('Predicted Rating')
plt.ylim(2,5)
plt.legend()
plt.show()



# %% [markdown]
# Not good :\

# %% [markdown]
# # Adding extra features
# Trying again with extra features

# %%
# Merge reviews with listings_extra_features where listing_id matches, and only
    keep one id column
all_features = pd.merge(listings_extra_features, reviews, left_on='id', right_on='
    listing_id', how='inner')
all_features = all_features.drop(columns=['listing_id'])

# Convert host_is_superhost to 0/1
all_features['host_is_superhost'] = all_features['host_is_superhost'].apply(lambda
    x: 1 if x == 't' else 0)

# Save to csv
all_features.to_csv('../data/all_features.csv', index=False)

all_features.dropna(inplace=True)


# %%
all_features.info()

# %% [markdown]
# Defining ColumnTransformer (CT)

# %%
# Create pipeline to use TF-IDF vectorizer on stems and passthrough other features
from sklearn.pipeline import Pipeline
from sklearn.compose import make_column_transformer
from sklearn.preprocessing import StandardScaler
from sklearn.feature_extraction.text import TfidfVectorizer

# Set min_df and max_df to the values that gave the best results (lowest MSE)
MIN_DF = 0.4
MAX_DF = 0.9

CT = make_column_transformer(
    (StandardScaler(), ['host_total_listings_count', 'accommodates', '
        number_of_reviews']),
    (TfidfVectorizer(stop_words='english', min_df=MIN_DF, max_df=MAX_DF), 'stems'),
    ('drop', 'id'),
    remainder='passthrough'
)
```

```
# pipeline = Pipeline([
#     ('transformer',  ct)
# ])

new_X = CT.fit_transform(all_features)

# print(new_X.shape)
# print(ct.get_feature_names())

# pipeline = pipe.fit_transform(all_features)



# %%
print(CT.get_feature_names_out())


# %% [markdown]
# Cross-validation for C

# %%
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression, Lasso, Ridge
from sklearn.model_selection import KFold, cross_val_score
from sklearn.metrics import mean_squared_error
from sklearn.pipeline import Pipeline

colors = ['blue', 'green', 'gold', 'red', 'pink']
c_range = [0.1, 0.2, 0.3, 0.4, 0.6]
# c_range = [2000, 4000, 6000, 8000, 10000]

m_linear = LinearRegression()

me_lasso = []
me_ridge = []
me_linear = []

se_lasso = []
se_ridge = []
se_linear = []

for c in c_range:
    print(f'C = {c}')

    m_lasso = Lasso(alpha=1/c)
    m_ridge = Ridge(alpha=1/c)

    kf = KFold(n_splits=5)

    errors_lasso = cross_val_score(m_lasso, new_X, Y_rating, cv=kf, scoring='
        neg_mean_squared_error')
    errors_ridge = cross_val_score(m_ridge, new_X, Y_rating, cv=kf, scoring='
        neg_mean_squared_error')
    errors_linear = cross_val_score(model, new_X, Y_rating, cv=kf, scoring='
        neg_mean_squared_error')
```

```
    me_lasso.append(np.array(errors_lasso).mean())
    me_ridge.append(np.array(errors_ridge).mean())
    me_linear.append(np.array(errors_linear).mean())

    se_lasso.append(np.array(errors_lasso).std())
    se_ridge.append(np.array(errors_ridge).std())
    se_linear.append(np.array(errors_linear).std())

plt.errorbar(c_range,me_lasso,yerr=se_lasso,color=colors[0],label=f'Lasso')
plt.errorbar(c_range,me_ridge,yerr=se_ridge,color=colors[1],label=f'Ridge')
plt.errorbar(c_range,me_linear,yerr=se_linear,color=colors[2],label=f'Linear')

plt.title('MSE␣vs␣C')
plt.xlabel('C')
plt.ylabel('Mean␣Square␣Error')
plt.legend()
plt.show()

# %% [markdown]
# Not much difference, but Ridge slightly better. No difference with C value above
    0.2

# %%
def gaussian_kernel10(distances):
    weights = np.exp(-10*(distances**2))
    return weights/np.sum(weights)

def gaussian_kernel50(distances):
    weights = np.exp(-50*(distances**2))
    return weights/np.sum(weights)

# %% [markdown]
# Cross-validation for K

# %%
# CROSS-VALIDATION TO GET K FOR KNN MODEL

from sklearn.neighbors import KNeighborsRegressor

k_range = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Want a model with uniform weights, a model with distance weights, and a model for
    each of the three kernels
# And perform cross-validation on each of these models to get the best k

me_unif = []
me_dist = []
me_gaus10 = []
me_gaus50 = []

se_unif = []
se_dist = []
se_gaus10 = []
se_gaus50 = []
```

```
for k in k_range:
    print(f'k = {k}')

    # Model with uniform weights, a model with distance weights, and a model for
       each of the three kernels
    m_unif = KNeighborsRegressor(n_neighbors=k, weights='uniform')
    m_dist = KNeighborsRegressor(n_neighbors=k, weights='distance')
    m_gaus10 = KNeighborsRegressor(n_neighbors=k, weights=gaussian_kernel10)
    m_gaus50 = KNeighborsRegressor(n_neighbors=k, weights=gaussian_kernel50)

    kf = KFold(n_splits=5)

    errors_unif = cross_val_score(m_unif, new_X, Y_rating, cv=kf, scoring='
       neg_mean_squared_error')
    errors_dist = cross_val_score(m_dist, new_X, Y_rating, cv=kf, scoring='
       neg_mean_squared_error')
    errors_gaus10 = cross_val_score(m_gaus10, new_X, Y_rating, cv=kf, scoring='
       neg_mean_squared_error')
    errors_gaus50 = cross_val_score(m_gaus50, new_X, Y_rating, cv=kf, scoring='
       neg_mean_squared_error')

    me_unif.append(np.array(errors_unif).mean())
    me_dist.append(np.array(errors_dist).mean())
    me_gaus10.append(np.array(errors_gaus10).mean())
    me_gaus50.append(np.array(errors_gaus50).mean())

    se_unif.append(np.array(errors_unif).std())
    se_dist.append(np.array(errors_dist).std())
    se_gaus10.append(np.array(errors_gaus10).std())
    se_gaus50.append(np.array(errors_gaus50).std())

plt.errorbar(k_range,me_unif,yerr=se_unif,color=colors[0],label=f'Uniform')
plt.errorbar(k_range,me_dist,yerr=se_dist,color=colors[1],label=f'Distance')
plt.errorbar(k_range,me_gaus10,yerr=se_gaus10,color=colors[2],label=f'Gaussian (
   sigma = 10)')
plt.errorbar(k_range,me_gaus50,yerr=se_gaus50,color=colors[3],label=f'Gaussian (
   sigma = 50)')

plt.title('MSE vs K')
plt.xlabel('K (Number of Neighbors)')
plt.ylabel('Mean Square Error')
plt.legend()
plt.show()


# %% [markdown]
# Still k = 6 w/ uniform weighting

# %%
# COMPARE PERFORMANCE OF LINEAR AND KNN MODELS, ALONG WITH BASELINE PREDICTORS (
   Y_rating)

# First get training and test data indices
from sklearn.model_selection import train_test_split
indices = np.arange(new_X.shape[0])
train, test = train_test_split(indices, test_size=0.2, random_state=0)
```

```
from sklearn.linear_model import LinearRegression
from sklearn.dummy import DummyRegressor
from sklearn.metrics import roc_curve, auc

# Linear Regression
# m_linear = LinearRegression()
# m_linear.fit(new_X[train], Y_rating[train])

# Ridge Regression
optimal_c = 0.2
m_ridge = Ridge(alpha=1/optimal_c)
m_ridge.fit(new_X[train], Y_rating[train])

# KNN
m_knn = KNeighborsRegressor(n_neighbors=6, weights='uniform')
m_knn.fit(new_X[train], Y_rating[train])

# Baseline predictors (mean value and constant 4)
m_mean = DummyRegressor(strategy='mean')
m_mean.fit(new_X[train], Y_rating[train])
m_const = DummyRegressor(strategy='constant', constant=4.5)
m_const.fit(new_X[train], Y_rating[train])

# Get predictions for all models
# pred_linear = m_linear.predict(new_X[test])
pred_ridge = m_ridge.predict(new_X[test])
pred_knn = m_knn.predict(new_X[test])
pred_mean = m_mean.predict(new_X[test])
pred_const = m_const.predict(new_X[test])

# Calculate MSE for all models
from sklearn.metrics import mean_squared_error
# mse_linear = mean_squared_error(Y_rating[test], pred_linear)
mse_ridge = mean_squared_error(Y_rating[test], pred_ridge)
mse_knn = mean_squared_error(Y_rating[test], pred_knn)
mse_mean = mean_squared_error(Y_rating[test], pred_mean)
mse_const = mean_squared_error(Y_rating[test], pred_const)

# print(f'MSE for Linear Regression: {mse_linear}')
print(f'MSE for Ridge Regression: {mse_ridge}')
print(f'MSE for KNN: {mse_knn}')
print(f'MSE for Mean: {mse_mean}')
print(f'MSE for Constant: {mse_const}')

# Visualise performance of all models
# plt.scatter(Y_rating[test], pred_linear, color=colors[0], label='Linear
    Regression')
plt.scatter(Y_rating[test], pred_ridge, color=colors[0], label=f'Ridge Regression (
    MSE = {mse_ridge:.4f})')
plt.scatter(Y_rating[test], pred_knn, color=colors[1], label=f'KNN (MSE = {mse_knn
    :.4f})')
plt.scatter(Y_rating[test], pred_mean, color=colors[2], label=f'Baseline: Mean (MSE
    = {mse_mean:.4f})')
```

```
plt.scatter(Y_rating[test], pred_const, color=colors[3], label=f'Baseline: Constant (MSE = {mse_const:.4f})')

plt.title('Predicted vs Actual Scores (Rating)')
plt.xlabel('Actual Rating')
# plt.xlim(2,5)
plt.ylabel('Predicted Rating')
# plt.ylim(2,5)
plt.legend()
plt.show()



# %%
# Print coefficients for extra features
# host_is_superhost
print(f'host_is_superhost: {m_linear.coef_[np.where(CT.get_feature_names_out() == "
    remainder__host_is_superhost")]}')
# host_total_listings_count
print(f'host_total_listings_count: {m_linear.coef_[np.where(CT.
    get_feature_names_out() == "standardscaler__host_total_listings_count")]}')
# accommodates
print(f'accommodates: {m_linear.coef_[np.where(CT.get_feature_names_out() == "
    standardscaler__accommodates")]}')
# number_of_reviews
print(f'number_of_reviews: {m_linear.coef_[np.where(CT.get_feature_names_out() == "
    standardscaler__number_of_reviews")]}')

# Find the most important features
# Get the indices of the top 5 features
top5 = np.argsort(m_linear.coef_)[-5:]
# Get the names of the top 5 features
top5_names = CT.get_feature_names_out()[top5]
# Get the values of the top 5 features
top5_values = m_linear.coef_[top5]
# Print the top 5 features
for i in range(len(top5)):
    print(f'{top5_names[i]}: {top5_values[i]}')


# %%
def print_key_coefs(coefs, transformer):
    # Print coefficients for extra features
    print('\nCoefs - Features of interest:\n----------------')

    # host_is_superhost
    print(f'host_is_superhost: {coefs[np.where(transformer.get_feature_names_out() 
        == "remainder__host_is_superhost")]}')
    # host_total_listings_count
    print(f'host_total_listings_count: {coefs[np.where(transformer.
        get_feature_names_out() == "standardscaler__host_total_listings_count")]}')
    # accommodates
    print(f'accommodates: {coefs[np.where(transformer.get_feature_names_out() == "
        standardscaler__accommodates")]}')
    # number_of_reviews
```

```
    print(f'number_of_reviews: {coefs[np.where(transformer.get_feature_names_out() ==
        "standardscaler__number_of_reviews")]}')


    # Find the 5 most increasing features
    print('\nCoefs - Top 5 increasing features:\n----------------')

    # Get the indices of the top 5 features
    top5 = np.argsort(coefs)[-5:]
    # Get the names of the top 5 features
    top5_names = transformer.get_feature_names_out()[top5]
    # Get the values of the top 5 features
    top5_values = coefs[top5]
    # Print the top 5 features
    for i in range(len(top5)):
        print(f'{top5_names[i]}: {top5_values[i]}')


    # Find the 5 most decreasing features
    print('\nCoefs - Top 5 decreasing features:\n----------------')

    # Get the indices of the top 5 features
    top5 = np.argsort(coefs)[:5]
    # Get the names of the top 5 features
    top5_names = transformer.get_feature_names_out()[top5]
    # Get the values of the top 5 features
    top5_values = coefs[top5]
    # Print the top 5 features
    for i in range(len(top5)):
        print(f'{top5_names[i]}: {top5_values[i]}')


# %% [markdown]
# This is actually interesting. Not bad!

# %% [markdown]
# Accuracy

# %%
# COMPARE PERFORMANCE OF LINEAR AND KNN MODELS, ALONG WITH BASELINE PREDICTORS (
    Y_acc)

# Ridge Regression
optimal_c = 0.2
m_ridge = Ridge(alpha=1/optimal_c)
m_ridge.fit(new_X[train], Y_acc[train])

# KNN
m_knn = KNeighborsRegressor(n_neighbors=6, weights='uniform')
m_knn.fit(new_X[train], Y_acc[train])

# Baseline predictors (mean value and constant 4)
m_mean = DummyRegressor(strategy='mean')
m_mean.fit(new_X[train], Y_acc[train])
m_const = DummyRegressor(strategy='constant', constant=4.5)
m_const.fit(new_X[train], Y_acc[train])
```

```
# Get predictions for all models
# pred_linear = m_linear.predict(new_X[test])
pred_ridge = m_ridge.predict(new_X[test])
pred_knn = m_knn.predict(new_X[test])
pred_mean = m_mean.predict(new_X[test])
pred_const = m_const.predict(new_X[test])

# Calculate MSE for all models
from sklearn.metrics import mean_squared_error
# mse_linear = mean_squared_error(Y_acc[test], pred_linear)
mse_ridge = mean_squared_error(Y_acc[test], pred_ridge)
mse_knn = mean_squared_error(Y_acc[test], pred_knn)
mse_mean = mean_squared_error(Y_acc[test], pred_mean)
mse_const = mean_squared_error(Y_acc[test], pred_const)

# print(f'MSE for Linear Regression: {mse_linear}')
print(f'MSE for Ridge Regression: {mse_ridge}')
print(f'MSE for KNN: {mse_knn}')
print(f'MSE for Mean: {mse_mean}')
print(f'MSE for Constant: {mse_const}')

# Visualise performance of all models
# plt.scatter(Y_acc[test], pred_linear, color=colors[0], label='Linear Regression')
plt.scatter(Y_acc[test], pred_ridge, color=colors[0], label=f'Ridge Regression (MSE
    = {mse_ridge:.4f})')
plt.scatter(Y_acc[test], pred_knn, color=colors[1], label=f'KNN (MSE = {mse_knn:.4f
    })')
plt.scatter(Y_acc[test], pred_mean, color=colors[2], label=f'Baseline: Mean (MSE =
    {mse_mean:.4f})')
plt.scatter(Y_acc[test], pred_const, color=colors[3], label=f'Baseline: Constant (
    MSE = {mse_const:.4f})')

plt.title('Predicted vs Actual Scores (Accuracy)')
plt.xlabel('Actual Rating')
# plt.xlim(2,5)
plt.ylabel('Predicted Rating')
# plt.ylim(2,5)
plt.legend()
plt.show()



# %%
# COMPARE PERFORMANCE OF LINEAR AND KNN MODELS, ALONG WITH BASELINE PREDICTORS (
    Y_clean)

# Ridge Regression
optimal_c = 0.2
m_ridge = Ridge(alpha=1/optimal_c)
m_ridge.fit(new_X[train], Y_clean[train])

# KNN
m_knn = KNeighborsRegressor(n_neighbors=6, weights='uniform')
m_knn.fit(new_X[train], Y_clean[train])
```

```
# Baseline predictors (mean value and constant 4)
m_mean = DummyRegressor(strategy='mean')
m_mean.fit(new_X[train], Y_clean[train])
m_const = DummyRegressor(strategy='constant', constant=4.5)
m_const.fit(new_X[train], Y_clean[train])

# Get predictions for all models
# pred_linear = m_linear.predict(new_X[test])
pred_ridge = m_ridge.predict(new_X[test])
pred_knn = m_knn.predict(new_X[test])
pred_mean = m_mean.predict(new_X[test])
pred_const = m_const.predict(new_X[test])

# Calculate MSE for all models
from sklearn.metrics import mean_squared_error
# mse_linear = mean_squared_error(Y_clean[test], pred_linear)
mse_ridge = mean_squared_error(Y_clean[test], pred_ridge)
mse_knn = mean_squared_error(Y_clean[test], pred_knn)
mse_mean = mean_squared_error(Y_clean[test], pred_mean)
mse_const = mean_squared_error(Y_clean[test], pred_const)

# print(f'MSE for Linear Regression: {mse_linear}')
print(f'MSE for Ridge Regression: {mse_ridge}')
print(f'MSE for KNN: {mse_knn}')
print(f'MSE for Mean: {mse_mean}')
print(f'MSE for Constant: {mse_const}')

# Visualise performance of all models
# plt.scatter(Y_clean[test], pred_linear, color=colors[0], label='Linear Regression
    ')
plt.scatter(Y_clean[test], pred_ridge, color=colors[0], label=f'Ridge Regression (
    MSE = {mse_ridge:.4f})')
plt.scatter(Y_clean[test], pred_knn, color=colors[1], label=f'KNN (MSE = {mse_knn
    :.4f})')
plt.scatter(Y_clean[test], pred_mean, color=colors[2], label=f'Baseline: Mean (MSE
    = {mse_mean:.4f})')
plt.scatter(Y_clean[test], pred_const, color=colors[3], label=f'Baseline: Constant
    (MSE = {mse_const:.4f})')

plt.title('Predicted vs Actual Scores (Cleanliness)')
plt.xlabel('Actual Rating')
# plt.xlim(2,5)
plt.ylabel('Predicted Rating')
# plt.ylim(2,5)
plt.legend()
plt.show()



# %%
# COMPARE PERFORMANCE OF LINEAR AND KNN MODELS, ALONG WITH BASELINE PREDICTORS (
    Y_checkin)

# Ridge Regression
optimal_c = 0.2
m_ridge = Ridge(alpha=1/optimal_c)
```

```
m_ridge.fit(new_X[train], Y_checkin[train])

# KNN
m_knn = KNeighborsRegressor(n_neighbors=6, weights='uniform')
m_knn.fit(new_X[train], Y_checkin[train])

# Baseline predictors (mean value and constant 4)
m_mean = DummyRegressor(strategy='mean')
m_mean.fit(new_X[train], Y_checkin[train])
m_const = DummyRegressor(strategy='constant', constant=4.5)
m_const.fit(new_X[train], Y_checkin[train])

# Get predictions for all models
# pred_linear = m_linear.predict(new_X[test])
pred_ridge = m_ridge.predict(new_X[test])
pred_knn = m_knn.predict(new_X[test])
pred_mean = m_mean.predict(new_X[test])
pred_const = m_const.predict(new_X[test])

# Calculate MSE for all models
from sklearn.metrics import mean_squared_error
# mse_linear = mean_squared_error(Y_checkin[test], pred_linear)
mse_ridge = mean_squared_error(Y_checkin[test], pred_ridge)
mse_knn = mean_squared_error(Y_checkin[test], pred_knn)
mse_mean = mean_squared_error(Y_checkin[test], pred_mean)
mse_const = mean_squared_error(Y_checkin[test], pred_const)

# print(f'MSE for Linear Regression: {mse_linear}')
print(f'MSE for Ridge Regression: {mse_ridge}')
print(f'MSE for KNN: {mse_knn}')
print(f'MSE for Mean: {mse_mean}')
print(f'MSE for Constant: {mse_const}')

# Visualise performance of all models
# plt.scatter(Y_checkin[test], pred_linear, color=colors[0], label='Linear
    Regression')
plt.scatter(Y_checkin[test], pred_ridge, color=colors[0], label=f'Ridge Regression
    (MSE = {mse_ridge:.4f})')
plt.scatter(Y_checkin[test], pred_knn, color=colors[1], label=f'KNN (MSE = {mse_knn
    :.4f})')
plt.scatter(Y_checkin[test], pred_mean, color=colors[2], label=f'Baseline: Mean (
    MSE = {mse_mean:.4f})')
plt.scatter(Y_checkin[test], pred_const, color=colors[3], label=f'Baseline:
    Constant (MSE = {mse_const:.4f})')

plt.title('Predicted vs Actual Scores (Checkin)')
plt.xlabel('Actual Rating')
# plt.xlim(2,5)
plt.ylabel('Predicted Rating')
# plt.ylim(2,5)
plt.legend()
plt.show()



# %% [markdown]
```

```
# # Resampled Data

# %% [markdown]
# ## Resampling

# %%
# Resample data to get more balanced target ratings
from sklearn.utils import resample

df_4_to_5stars = all_features[(Y_rating > 4) & (Y_rating <= 5)]
df_3_to_4stars = all_features[(Y_rating > 3) & (Y_rating <= 4)]
df_2_to_3stars = all_features[(Y_rating > 2) & (Y_rating <= 3)]
df_1_to_2stars = all_features[(Y_rating >= 1) & (Y_rating <= 2)]

# Print shape of each class
print('Original:\----')
print(f'4␣to␣5:␣{df_4_to_5stars.shape}')
print(f'3␣to␣4:␣{df_3_to_4stars.shape}')
print(f'2␣to␣3:␣{df_2_to_3stars.shape}')
print(f'1␣to␣2:␣{df_1_to_2stars.shape}')

# Resample each class to have 2000 samples
df_4_to_5stars_resampled = resample(df_4_to_5stars, replace=True, n_samples=4000,
    random_state=0)
df_3_to_4stars_resampled = resample(df_3_to_4stars, replace=True, n_samples=4000,
    random_state=0)
df_2_to_3stars_resampled = resample(df_2_to_3stars, replace=True, n_samples=4000,
    random_state=0)
df_1_to_2stars_resampled = resample(df_1_to_2stars, replace=True, n_samples=4000,
    random_state=0)

new_Y = targets.iloc[:,1:]

# Generate matching target ratings
Y_4_to_5stars_resampled = resample(new_Y[(Y_rating > 4) & (Y_rating <= 5)], replace
    =True, n_samples=4000, random_state=0)
Y_3_to_4stars_resampled = resample(new_Y[(Y_rating > 3) & (Y_rating <= 4)], replace
    =True, n_samples=4000, random_state=0)
Y_2_to_3stars_resampled = resample(new_Y[(Y_rating > 2) & (Y_rating <= 3)], replace
    =True, n_samples=4000, random_state=0)
Y_1_to_2stars_resampled = resample(new_Y[(Y_rating >= 1) & (Y_rating <= 2)],
    replace=True, n_samples=4000, random_state=0)

# Print new shape of each class
print('Resampled␣shapes:\----')
print(f'4␣to␣5:␣{df_4_to_5stars_resampled.shape}')
print(f'3␣to␣4:␣{df_3_to_4stars_resampled.shape}')
print(f'2␣to␣3:␣{df_2_to_3stars_resampled.shape}')
print(f'1␣to␣2:␣{df_1_to_2stars_resampled.shape}')

# Print new target ratings
# print('Target Ratings:\----')
# print(Y_4_to_5stars_upsampled)
# print(Y_3_to_4stars_upsampled)
# print(Y_2_to_3stars_upsampled)
# print(Y_1_to_2stars_upsampled)
```

```
# Combine resampled data
feats_resampled = pd.concat([df_4_to_5stars_resampled, df_3_to_4stars_resampled,
    df_2_to_3stars_resampled, df_1_to_2stars_resampled])
Y_resampled = pd.concat([Y_4_to_5stars_resampled, Y_3_to_4stars_resampled,
    Y_2_to_3stars_resampled, Y_1_to_2stars_resampled])

# Print shape of combined data
print('Combined␣shape:\----')
print(feats_resampled.shape)
print(Y_resampled.shape)

# Print head of combined data
# print('Combined head:\----')
# print(feats_resampled.head())
# print(Y_resampled.tail())

# Remove 'review_scores_' prefix from Y_resampled column names
Y_resampled.columns = Y_resampled.columns.str.replace('review_scores_', '')

# Print head of Y_resampled with new column names
print('New␣column␣names:\----')
print(feats_resampled.columns)
print(Y_resampled.columns)


# %% [markdown]
# ## Re-extracting text features

# %%
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import KFold, cross_val_score
from sklearn.feature_extraction.text import TfidfVectorizer

colors = ['blue', 'green', 'gold', 'red', 'pink']

# df_min - exclude words which appear in too few documents
min_df_range = [0.04, 0.05, 0.06, 0.07]
# df_max - exclude words which appear in too many documents
max_df_range = [0.2, 0.4, 0.6, 0.7, 0.8]

for min_df_index, min_df in enumerate(min_df_range):
    print(f'min_df␣=␣{min_df}')

    mean_score=[]
    std_score=[]

    for max_df in max_df_range:
        print(f'␣-␣max_df␣=␣{max_df}')

        vectorizer = TfidfVectorizer(stop_words='english', min_df=min_df, max_df=
            max_df) # 79,179 features for 6,078 listings
        test_X = vectorizer.fit_transform(feats_resampled['stems'])

        model = LinearRegression()
```

```
        kf = KFold(n_splits=5)

        scores = cross_val_score(model, test_X, Y_resampled['rating'], cv=kf,
            scoring='neg_mean_squared_error')
        mean_score.append(np.array(scores).mean())
        std_score.append(np.array(scores).std())

    plt.errorbar(max_df_range,mean_score,yerr=std_score,color=colors[min_df_index],
        label=f'min_df = {min_df}')

plt.title('MSE vs Max DF (Basic Linear Regression Model)')
plt.xlabel('Max Document Frequency (max_df)')
plt.ylabel('Mean Square Error')
plt.legend()
plt.show()

# %% [markdown]
# min 0.06 -> max 0.7

# %%
# Create pipeline to use TF-IDF vectorizer on stems and passthrough other features
from sklearn.pipeline import Pipeline
from sklearn.compose import make_column_transformer
from sklearn.preprocessing import StandardScaler
from sklearn.feature_extraction.text import TfidfVectorizer

# Set min_df and max_df to the values that gave the best results (lowest MSE)
RESAMPLED_MIN_DF = 0.06
RESAMPLED_MAX_DF = 0.7

RESAMPLED_CT = make_column_transformer(
    (StandardScaler(), ['host_total_listings_count', 'accommodates', '
        number_of_reviews']),
    (TfidfVectorizer(stop_words='english', min_df=RESAMPLED_MIN_DF, max_df=
        RESAMPLED_MAX_DF), 'stems'),
    ('drop', 'id'),
    remainder='passthrough'
)

# pipeline = Pipeline([
#     ('transformer',  ct)
# ])

resampled_X = RESAMPLED_CT.fit_transform(feats_resampled)
print(resampled_X.shape)
print(RESAMPLED_CT.get_feature_names_out())

# %% [markdown]
# ## Final Cross-Validation and Evaluation

# %% [markdown]
# ### Rating

# %%
# COMPARE PERFORMANCE OF LINEAR AND KNN MODELS, ALONG WITH BASELINE PREDICTORS (
    rating)
```

```
# Create train test split

resampled_indices = np.arange(resampled_X.shape[0])

from sklearn.model_selection import train_test_split
r_train, r_test = train_test_split(resampled_indices, test_size=0.2, random_state
    =0)

# Ridge Regression
optimal_c = 0.8
m_ridge = Ridge(alpha=1/optimal_c)
m_ridge.fit(resampled_X[r_train], Y_resampled['rating'].iloc[r_train])

# KNN
m_knn = KNeighborsRegressor(n_neighbors=3, weights='uniform')
m_knn.fit(resampled_X[r_train], Y_resampled['rating'].iloc[r_train])

# Baseline predictors (mean value and constant 4)
m_mean = DummyRegressor(strategy='mean')
m_mean.fit(resampled_X[r_train], Y_resampled['rating'].iloc[r_train])
m_const = DummyRegressor(strategy='constant', constant=4.5)
m_const.fit(resampled_X[r_train], Y_resampled['rating'].iloc[r_train])

# Get predictions for all models
pred_ridge = m_ridge.predict(resampled_X[r_test])
pred_knn = m_knn.predict(resampled_X[r_test])
pred_mean = m_mean.predict(resampled_X[r_test])
pred_const = m_const.predict(resampled_X[r_test])

# Calculate MSE for all models
from sklearn.metrics import mean_squared_error
mse_ridge = mean_squared_error(Y_resampled['rating'].iloc[r_test], pred_ridge)
mse_knn = mean_squared_error(Y_resampled['rating'].iloc[r_test], pred_knn)
mse_mean = mean_squared_error(Y_resampled['rating'].iloc[r_test], pred_mean)
mse_const = mean_squared_error(Y_resampled['rating'].iloc[r_test], pred_const)

# print(f'MSE for Linear Regression: {mse_linear}')
# print(f'MSE for Ridge Regression: {mse_ridge}')
# print(f'MSE for KNN: {mse_knn}')
# print(f'MSE for Mean: {mse_mean}')
# print(f'MSE for Constant: {mse_const}')

# Print top 5 features for Ridge model
print_key_coefs(m_ridge.coef_, RESAMPLED_CT)

# Visualise performance of all models
plt.scatter(Y_resampled['rating'].iloc[r_test], pred_ridge, color=colors[0], label=
    f'Ridge␣Regression␣(MSE␣=␣{mse_ridge:.4f})', marker='o')
plt.scatter(Y_resampled['rating'].iloc[r_test], pred_knn, color=colors[1], label=f'
    KNN␣(MSE␣=␣{mse_knn:.4f})', marker='o')
plt.scatter(Y_resampled['rating'].iloc[r_test], pred_mean, color=colors[2], label=f
    'Baseline:␣Mean␣(MSE␣=␣{mse_mean:.4f})', marker='.')
plt.scatter(Y_resampled['rating'].iloc[r_test], pred_const, color=colors[3], label=
    f'Baseline:␣Constant␣(MSE␣=␣{mse_const:.4f})', marker='.')
```

```
plt.title('Predicted vs Actual Scores (Rating)')
plt.xlabel('Actual Rating')
# plt.xlim(2,5)
plt.ylabel('Predicted Rating')
# plt.ylim(2,5)
plt.legend()
plt.show()


# %% [markdown]
# Cross validation for new K - it's looking a little rough on resampled data

# %%
def gaussian_kernel20(distances):
    weights = np.exp(-20*(distances**2))
    return weights/np.sum(weights)


# %%
# CROSS-VALIDATION TO GET K FOR KNN MODEL

from sklearn.neighbors import KNeighborsRegressor

k_range = [2, 3, 4]

# Want a model with uniform weights, a model with distance weights, and a model for
    each of the three kernels
# And perform cross-validation on each of these models to get the best k

me_unif = []
me_dist = []
me_gaus10 = []
me_gaus20 = []
# me_gaus50 = []

se_unif = []
se_dist = []
se_gaus10 = []
se_gaus20 = []
# se_gaus50 = []

for k in k_range:
    print(f'k = {k}')

    # Model with uniform weights, a model with distance weights, and a model for
       each of the three kernels
    m_unif = KNeighborsRegressor(n_neighbors=k, weights='uniform')
    m_dist = KNeighborsRegressor(n_neighbors=k, weights='distance')
    m_gaus10 = KNeighborsRegressor(n_neighbors=k, weights=gaussian_kernel10)
    m_gaus20 = KNeighborsRegressor(n_neighbors=k, weights=gaussian_kernel20)
    # m_gaus50 = KNeighborsRegressor(n_neighbors=k, weights=gaussian_kernel50)

    kf = KFold(n_splits=5)

    errors_unif = cross_val_score(m_unif, resampled_X, Y_resampled['rating'], cv=kf
       , scoring='neg_mean_squared_error')
```

```
    errors_dist = cross_val_score(m_dist, resampled_X, Y_resampled['rating'], cv=kf
        , scoring='neg_mean_squared_error')
    errors_gaus10 = cross_val_score(m_gaus10, resampled_X, Y_resampled['rating'],
        cv=kf, scoring='neg_mean_squared_error')
    errors_gaus20 = cross_val_score(m_gaus20, resampled_X, Y_resampled['rating'],
        cv=kf, scoring='neg_mean_squared_error')
    # errors_gaus50 = cross_val_score(m_gaus50, resampled_X, Y_resampled['rating'],
         cv=kf, scoring='neg_mean_squared_error')

    me_unif.append(np.array(errors_unif).mean())
    me_dist.append(np.array(errors_dist).mean())
    me_gaus10.append(np.array(errors_gaus10).mean())
    me_gaus20.append(np.array(errors_gaus20).mean())
    # me_gaus50.append(np.array(errors_gaus50).mean())

    se_unif.append(np.array(errors_unif).std())
    se_dist.append(np.array(errors_dist).std())
    se_gaus10.append(np.array(errors_gaus10).std())
    se_gaus20.append(np.array(errors_gaus20).std())
    # se_gaus50.append(np.array(errors_gaus50).std())

plt.errorbar(k_range,me_unif,yerr=se_unif,color=colors[0],label=f'Uniform')
plt.errorbar(k_range,me_dist,yerr=se_dist,color=colors[1],label=f'Distance')
plt.errorbar(k_range,me_gaus10,yerr=se_gaus10,color=colors[2],label=f'Gaussian␣(
    sigma␣=␣10)')
plt.errorbar(k_range,me_gaus20,yerr=se_gaus20,color=colors[3],label=f'Gaussian␣(
    sigma␣=␣20)')
# plt.errorbar(k_range,me_gaus50,yerr=se_gaus50,color=colors[3],label=f'Gaussian (
    sigma = 50)')

plt.title('MSE␣vs␣K')
plt.xlabel('K␣(Number␣of␣Neighbors)')
plt.ylabel('Mean␣Square␣Error')
plt.legend()
plt.show()


# %% [markdown]
# k = 3 w/ Gaussian weighting, sigma=10

# %% [markdown]
# Cross-validation for C in regression models

# %%
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression, Lasso, Ridge
from sklearn.model_selection import KFold, cross_val_score
from sklearn.metrics import mean_squared_error
from sklearn.pipeline import Pipeline

colors = ['gold', 'red', 'blue', 'green', 'pink']
# c_range = [0.1, 1, 10, 100, 1000]
c_range = [0.6, 0.8, 1.2]
# c_range = [2000, 4000, 6000, 8000, 10000]

m_linear = LinearRegression()
```

```
me_lasso = []
me_ridge = []
me_linear = []

se_lasso = []
se_ridge = []
se_linear = []

for c in c_range:
    print(f'C = {c}')

    m_lasso = Lasso(alpha=1/c)
    m_ridge = Ridge(alpha=1/c)

    kf = KFold(n_splits=5)

    errors_lasso = cross_val_score(m_lasso, resampled_X, Y_resampled['rating'], cv=
        kf, scoring='neg_mean_squared_error')
    errors_ridge = cross_val_score(m_ridge, resampled_X, Y_resampled['rating'], cv=
        kf, scoring='neg_mean_squared_error')
    errors_linear = cross_val_score(model, resampled_X, Y_resampled['rating'], cv=
        kf, scoring='neg_mean_squared_error')

    me_lasso.append(np.array(errors_lasso).mean())
    me_ridge.append(np.array(errors_ridge).mean())
    me_linear.append(np.array(errors_linear).mean())

    se_lasso.append(np.array(errors_lasso).std())
    se_ridge.append(np.array(errors_ridge).std())
    se_linear.append(np.array(errors_linear).std())

plt.errorbar(c_range,me_lasso,yerr=se_lasso,color=colors[0],label=f'Lasso')
plt.errorbar(c_range,me_ridge,yerr=se_ridge,color=colors[1],label=f'Ridge')
plt.errorbar(c_range,me_linear,yerr=se_linear,color=colors[2],label=f'Linear')

plt.title('MSE vs C')
plt.xlabel('C')
plt.ylabel('Mean Square Error')
plt.legend()
plt.show()

# %% [markdown]
# C = 0.8 looking best

# %% [markdown]
# ### Accuracy

# %%
# COMPARE PERFORMANCE OF LINEAR AND KNN MODELS, ALONG WITH BASELINE PREDICTORS (
    accuracy)

# Columns in Y_resampled
# 'rating', 'accuracy', 'cleanliness', 'checkin', 'communication', 'location', '
    value'
```

```
colors = ['red', 'blue', 'lime', 'green', 'pink']

# Ridge Regression
optimal_c = 0.8
m_ridge = Ridge(alpha=1/optimal_c)
m_ridge.fit(resampled_X[r_train], Y_resampled['rating'].iloc[r_train])

# KNN
m_knn = KNeighborsRegressor(n_neighbors=3, weights='uniform')
m_knn.fit(resampled_X[r_train], Y_resampled['accuracy'].iloc[r_train])

# Baseline predictors (mean value and constant 4)
m_mean = DummyRegressor(strategy='mean')
m_mean.fit(resampled_X[r_train], Y_resampled['accuracy'].iloc[r_train])
m_const = DummyRegressor(strategy='constant', constant=4.5)
m_const.fit(resampled_X[r_train], Y_resampled['accuracy'].iloc[r_train])

# Get predictions for all models
pred_ridge = m_ridge.predict(resampled_X[r_test])
pred_knn = m_knn.predict(resampled_X[r_test])
pred_mean = m_mean.predict(resampled_X[r_test])
pred_const = m_const.predict(resampled_X[r_test])

# Calculate MSE for all models
from sklearn.metrics import mean_squared_error
mse_ridge = mean_squared_error(Y_resampled['accuracy'].iloc[r_test], pred_ridge)
mse_knn = mean_squared_error(Y_resampled['accuracy'].iloc[r_test], pred_knn)
mse_mean = mean_squared_error(Y_resampled['accuracy'].iloc[r_test], pred_mean)
mse_const = mean_squared_error(Y_resampled['accuracy'].iloc[r_test], pred_const)

# print(f'MSE for Linear Regression: {mse_linear}')
# print(f'MSE for Ridge Regression: {mse_ridge}')
# print(f'MSE for KNN: {mse_knn}')
# print(f'MSE for Mean: {mse_mean}')
# print(f'MSE for Constant: {mse_const}')

# Print top 5 features for Ridge model
print_key_coefs(m_ridge.coef_, RESAMPLED_CT)

# Visualise performance of all models
plt.scatter(Y_resampled['accuracy'].iloc[r_test], pred_ridge, color=colors[0],
    label=f'Ridge Regression (MSE = {mse_ridge:.4f})', marker='o')
plt.scatter(Y_resampled['accuracy'].iloc[r_test], pred_knn, color=colors[1], label=
    f'KNN (MSE = {mse_knn:.4f})', marker='o')
plt.scatter(Y_resampled['accuracy'].iloc[r_test], pred_mean, color=colors[2], label
    =f'Baseline: Mean (MSE = {mse_mean:.4f})', marker='.')
plt.scatter(Y_resampled['accuracy'].iloc[r_test], pred_const, color=colors[3],
    label=f'Baseline: Constant (MSE = {mse_const:.4f})', marker='.')

plt.title('Predicted vs Actual Scores (Accuracy)')
plt.xlabel('Actual Rating')
# plt.xlim(2,5)
plt.ylabel('Predicted Rating')
# plt.ylim(2,5)
plt.legend()
plt.show()
```

```
# %% [markdown]
# ### Cleanliness

# %%
# COMPARE PERFORMANCE OF LINEAR AND KNN MODELS, ALONG WITH BASELINE PREDICTORS (
    cleanliness)

# Columns in Y_resampled
# 'rating', 'accuracy', 'cleanliness', 'checkin', 'communication', 'location', '
    value'

# Ridge Regression
optimal_c = 0.8
m_ridge = Ridge(alpha=1/optimal_c)
m_ridge.fit(resampled_X[r_train], Y_resampled['cleanliness'].iloc[r_train])

# KNN
m_knn = KNeighborsRegressor(n_neighbors=3, weights='uniform')
m_knn.fit(resampled_X[r_train], Y_resampled['cleanliness'].iloc[r_train])

# Baseline predictors (mean value and constant 4)
m_mean = DummyRegressor(strategy='mean')
m_const = DummyRegressor(strategy='constant', constant=4.5)
m_mean.fit( resampled_X[r_train], Y_resampled['cleanliness'].iloc[r_train])
m_const.fit(resampled_X[r_train], Y_resampled['cleanliness'].iloc[r_train])

# Get predictions for all models
pred_ridge =    m_ridge.predict(resampled_X[r_test])
pred_knn =      m_knn.predict(resampled_X[r_test])
pred_mean =     m_mean.predict(resampled_X[r_test])
pred_const =    m_const.predict(resampled_X[r_test])

# Calculate MSE for all models
from sklearn.metrics import mean_squared_error
mse_ridge = mean_squared_error(Y_resampled['cleanliness'].iloc[r_test], pred_ridge)
mse_knn =   mean_squared_error(Y_resampled['cleanliness'].iloc[r_test], pred_knn)
mse_mean =  mean_squared_error(Y_resampled['cleanliness'].iloc[r_test], pred_mean)
mse_const = mean_squared_error(Y_resampled['cleanliness'].iloc[r_test], pred_const)

# print(f'MSE for Linear Regression: {mse_linear}')
# print(f'MSE for Ridge Regression: {mse_ridge}')
# print(f'MSE for KNN: {mse_knn}')
# print(f'MSE for Mean: {mse_mean}')
# print(f'MSE for Constant: {mse_const}')

# Print top 5 features for Ridge model
print_key_coefs(m_ridge.coef_, RESAMPLED_CT)

# Visualise performance of all models
plt.scatter(Y_resampled['cleanliness'].iloc[r_test], pred_ridge, color=colors[0],
    label=f'Ridge␣Regression␣(MSE␣=␣{mse_ridge:.4f})', marker='o')
plt.scatter(Y_resampled['cleanliness'].iloc[r_test], pred_knn, color=colors[1],
    label=f'KNN␣(MSE␣=␣{mse_knn:.4f})', marker='o')
```

```
plt.scatter(Y_resampled['cleanliness'].iloc[r_test], pred_mean, color=colors[2],
    label=f'Baseline: Mean (MSE = {mse_mean:.4f})', marker='.')
plt.scatter(Y_resampled['cleanliness'].iloc[r_test], pred_const, color=colors[3],
    label=f'Baseline: Constant (MSE = {mse_const:.4f})', marker='.')

plt.title('Predicted vs Actual Scores (Cleanliness)')
plt.xlabel('Actual Rating')
# plt.xlim(2,5)
plt.ylabel('Predicted Rating')
# plt.ylim(2,5)
plt.legend()
plt.show()



# %% [markdown]
# ### Checkin

# %%
# COMPARE PERFORMANCE OF LINEAR AND KNN MODELS, ALONG WITH BASELINE PREDICTORS (
    checkin)

# Columns in Y_resampled
# 'rating', 'accuracy', 'cleanliness', 'checkin', 'communication', 'location', '
    value'

# Ridge Regression
optimal_c = 0.8
m_ridge = Ridge(alpha=1/optimal_c)
m_ridge.fit(resampled_X[r_train], Y_resampled['checkin'].iloc[r_train])

# KNN
m_knn = KNeighborsRegressor(n_neighbors=3, weights='uniform')
m_knn.fit(resampled_X[r_train], Y_resampled['checkin'].iloc[r_train])

# Baseline predictors (mean value and constant 4)
m_mean = DummyRegressor(strategy='mean')
m_const = DummyRegressor(strategy='constant', constant=4.5)
m_mean.fit( resampled_X[r_train], Y_resampled['checkin'].iloc[r_train])
m_const.fit(resampled_X[r_train], Y_resampled['checkin'].iloc[r_train])

# Get predictions for all models
pred_ridge =    m_ridge.predict(resampled_X[r_test])
pred_knn =      m_knn.predict(resampled_X[r_test])
pred_mean =     m_mean.predict(resampled_X[r_test])
pred_const =    m_const.predict(resampled_X[r_test])

# Calculate MSE for all models
from sklearn.metrics import mean_squared_error
mse_ridge = mean_squared_error(Y_resampled['checkin'].iloc[r_test], pred_ridge)
mse_knn =   mean_squared_error(Y_resampled['checkin'].iloc[r_test], pred_knn)
mse_mean =  mean_squared_error(Y_resampled['checkin'].iloc[r_test], pred_mean)
mse_const = mean_squared_error(Y_resampled['checkin'].iloc[r_test], pred_const)

# print(f'MSE for Linear Regression: {mse_linear}')
# print(f'MSE for Ridge Regression: {mse_ridge}')
```

```
# print(f'MSE for KNN: {mse_knn}')
# print(f'MSE for Mean: {mse_mean}')
# print(f'MSE for Constant: {mse_const}')

# Print top 5 features for Ridge model
print_key_coefs(m_ridge.coef_, RESAMPLED_CT)

# Visualise performance of all models
plt.scatter(Y_resampled['checkin'].iloc[r_test], pred_ridge, color=colors[0], label
    =f'Ridge␣Regression␣(MSE␣=␣{mse_ridge:.4f})', marker='o')
plt.scatter(Y_resampled['checkin'].iloc[r_test], pred_knn, color=colors[1], label=f
    'KNN␣(MSE␣=␣{mse_knn:.4f})', marker='o')
plt.scatter(Y_resampled['checkin'].iloc[r_test], pred_mean, color=colors[2], label=
    f'Baseline:␣Mean␣(MSE␣=␣{mse_mean:.4f})', marker='.')
plt.scatter(Y_resampled['checkin'].iloc[r_test], pred_const, color=colors[3], label
    =f'Baseline:␣Constant␣(MSE␣=␣{mse_const:.4f})', marker='.')

plt.title('Predicted␣vs␣Actual␣Scores␣(Checkin)')
plt.xlabel('Actual␣Rating')
# plt.xlim(2,5)
plt.ylabel('Predicted␣Rating')
# plt.ylim(2,5)
plt.legend()
plt.show()



# %% [markdown]
# ### Communication

# %%
# COMPARE PERFORMANCE OF LINEAR AND KNN MODELS, ALONG WITH BASELINE PREDICTORS (
    communication)

# Columns in Y_resampled
# 'rating', 'accuracy', 'cleanliness', 'checkin', 'communication', 'location', '
    value'

# Ridge Regression
optimal_c = 0.8
m_ridge = Ridge(alpha=1/optimal_c)
m_ridge.fit(resampled_X[r_train], Y_resampled['communication'].iloc[r_train])

# KNN
m_knn = KNeighborsRegressor(n_neighbors=3, weights='uniform')
m_knn.fit(resampled_X[r_train], Y_resampled['communication'].iloc[r_train])

# Baseline predictors (mean value and constant 4)
m_mean = DummyRegressor(strategy='mean')
m_const = DummyRegressor(strategy='constant', constant=4.5)
m_mean.fit( resampled_X[r_train], Y_resampled['communication'].iloc[r_train])
m_const.fit(resampled_X[r_train], Y_resampled['communication'].iloc[r_train])

# Get predictions for all models
pred_ridge =    m_ridge.predict(resampled_X[r_test])
pred_knn =      m_knn.predict(resampled_X[r_test])
```

```
pred_mean =     m_mean.predict(resampled_X[r_test])
pred_const =    m_const.predict(resampled_X[r_test])

# Calculate MSE for all models
from sklearn.metrics import mean_squared_error
mse_ridge = mean_squared_error(Y_resampled['communication'].iloc[r_test],
    pred_ridge)
mse_knn =   mean_squared_error(Y_resampled['communication'].iloc[r_test], pred_knn)
mse_mean =  mean_squared_error(Y_resampled['communication'].iloc[r_test], pred_mean
    )
mse_const = mean_squared_error(Y_resampled['communication'].iloc[r_test],
    pred_const)

# print(f'MSE for Linear Regression: {mse_linear}')
# print(f'MSE for Ridge Regression: {mse_ridge}')
# print(f'MSE for KNN: {mse_knn}')
# print(f'MSE for Mean: {mse_mean}')
# print(f'MSE for Constant: {mse_const}')

# Print top 5 features for Ridge model
print_key_coefs(m_ridge.coef_, RESAMPLED_CT)

# Visualise performance of all models
plt.scatter(Y_resampled['communication'].iloc[r_test], pred_ridge, color=colors[0],
     label=f'Ridge␣Regression␣(MSE␣=␣{mse_ridge:.4f})', marker='o')
plt.scatter(Y_resampled['communication'].iloc[r_test], pred_knn, color=colors[1],
    label=f'KNN␣(MSE␣=␣{mse_knn:.4f})', marker='o')
plt.scatter(Y_resampled['communication'].iloc[r_test], pred_mean, color=colors[2],
    label=f'Baseline:␣Mean␣(MSE␣=␣{mse_mean:.4f})', marker='.')
plt.scatter(Y_resampled['communication'].iloc[r_test], pred_const, color=colors[3],
     label=f'Baseline:␣Constant␣(MSE␣=␣{mse_const:.4f})', marker='.')

plt.title('Predicted␣vs␣Actual␣Scores␣(Communication)')
plt.xlabel('Actual␣Rating')
# plt.xlim(2,5)
plt.ylabel('Predicted␣Rating')
# plt.ylim(2,5)
plt.legend()
plt.show()



# %% [markdown]
# ### Location

# %%
# COMPARE PERFORMANCE OF LINEAR AND KNN MODELS, ALONG WITH BASELINE PREDICTORS (
    location)

# Columns in Y_resampled
# 'rating', 'accuracy', 'cleanliness', 'checkin', 'communication', 'location', '
    value'

# Ridge Regression
optimal_c = 0.8
m_ridge = Ridge(alpha=1/optimal_c)
```

```
m_ridge.fit(resampled_X[r_train], Y_resampled['location'].iloc[r_train])

# KNN
m_knn = KNeighborsRegressor(n_neighbors=3, weights='uniform')
m_knn.fit(resampled_X[r_train], Y_resampled['location'].iloc[r_train])

# Baseline predictors (mean value and constant 4)
m_mean = DummyRegressor(strategy='mean')
m_const = DummyRegressor(strategy='constant', constant=4.5)
m_mean.fit( resampled_X[r_train], Y_resampled['location'].iloc[r_train])
m_const.fit(resampled_X[r_train], Y_resampled['location'].iloc[r_train])

# Get predictions for all models
pred_ridge =    m_ridge.predict(resampled_X[r_test])
pred_knn =      m_knn.predict(resampled_X[r_test])
pred_mean =     m_mean.predict(resampled_X[r_test])
pred_const =    m_const.predict(resampled_X[r_test])

# Calculate MSE for all models
from sklearn.metrics import mean_squared_error
mse_ridge = mean_squared_error(Y_resampled['location'].iloc[r_test], pred_ridge)
mse_knn =   mean_squared_error(Y_resampled['location'].iloc[r_test], pred_knn)
mse_mean =  mean_squared_error(Y_resampled['location'].iloc[r_test], pred_mean)
mse_const = mean_squared_error(Y_resampled['location'].iloc[r_test], pred_const)

# print(f'MSE for Linear Regression: {mse_linear}')
# print(f'MSE for Ridge Regression: {mse_ridge}')
# print(f'MSE for KNN: {mse_knn}')
# print(f'MSE for Mean: {mse_mean}')
# print(f'MSE for Constant: {mse_const}')

# Print top 5 features for Ridge model
print_key_coefs(m_ridge.coef_, RESAMPLED_CT)

# Visualise performance of all models
plt.scatter(Y_resampled['location'].iloc[r_test], pred_ridge, color=colors[0],
    label=f'Ridge␣Regression␣(MSE␣=␣{mse_ridge:.4f})', marker='o')
plt.scatter(Y_resampled['location'].iloc[r_test], pred_knn, color=colors[1], label=
    f'KNN␣(MSE␣=␣{mse_knn:.4f})', marker='o')
plt.scatter(Y_resampled['location'].iloc[r_test], pred_mean, color=colors[2], label
    =f'Baseline:␣Mean␣(MSE␣=␣{mse_mean:.4f})', marker='.')
plt.scatter(Y_resampled['location'].iloc[r_test], pred_const, color=colors[3],
    label=f'Baseline:␣Constant␣(MSE␣=␣{mse_const:.4f})', marker='.')

plt.title('Predicted␣vs␣Actual␣Scores␣(Location)')
plt.xlabel('Actual␣Rating')
# plt.xlim(2,5)
plt.ylabel('Predicted␣Rating')
# plt.ylim(2,5)
plt.legend()
plt.show()



# %% [markdown]
# ### Value
```

```
# %%
# COMPARE PERFORMANCE OF LINEAR AND KNN MODELS, ALONG WITH BASELINE PREDICTORS (
    value)

# Columns in Y_resampled
# 'rating', 'accuracy', 'cleanliness', 'checkin', 'communication', 'location', '
    value'

# Ridge Regression
optimal_c = 0.8
m_ridge = Ridge(alpha=1/optimal_c)
m_ridge.fit(resampled_X[r_train], Y_resampled['value'].iloc[r_train])

# KNN
m_knn = KNeighborsRegressor(n_neighbors=3, weights='uniform')
m_knn.fit(resampled_X[r_train], Y_resampled['value'].iloc[r_train])

# Baseline predictors (mean value and constant 4)
m_mean = DummyRegressor(strategy='mean')
m_const = DummyRegressor(strategy='constant', constant=4.5)
m_mean.fit( resampled_X[r_train], Y_resampled['value'].iloc[r_train])
m_const.fit(resampled_X[r_train], Y_resampled['value'].iloc[r_train])

# Get predictions for all models
pred_ridge =    m_ridge.predict(resampled_X[r_test])
pred_knn =      m_knn.predict(resampled_X[r_test])
pred_mean =     m_mean.predict(resampled_X[r_test])
pred_const =    m_const.predict(resampled_X[r_test])

# Calculate MSE for all models
from sklearn.metrics import mean_squared_error
mse_ridge = mean_squared_error(Y_resampled['value'].iloc[r_test], pred_ridge)
mse_knn =   mean_squared_error(Y_resampled['value'].iloc[r_test], pred_knn)
mse_mean =  mean_squared_error(Y_resampled['value'].iloc[r_test], pred_mean)
mse_const = mean_squared_error(Y_resampled['value'].iloc[r_test], pred_const)

# print(f'MSE for Linear Regression: {mse_linear}')
# print(f'MSE for Ridge Regression: {mse_ridge}')
# print(f'MSE for KNN: {mse_knn}')
# print(f'MSE for Mean: {mse_mean}')
# print(f'MSE for Constant: {mse_const}')

# Print top 5 features for Ridge model
print_key_coefs(m_ridge.coef_, RESAMPLED_CT)

# Visualise performance of all models
plt.scatter(Y_resampled['value'].iloc[r_test], pred_ridge, color=colors[0], label=f
    'Ridge Regression (MSE = {mse_ridge:.4f})', marker='o')
plt.scatter(Y_resampled['value'].iloc[r_test], pred_knn, color=colors[1], label=f'
    KNN (MSE = {mse_knn:.4f})', marker='o')
plt.scatter(Y_resampled['value'].iloc[r_test], pred_mean, color=colors[2], label=f'
    Baseline: Mean (MSE = {mse_mean:.4f})', marker='.')
plt.scatter(Y_resampled['value'].iloc[r_test], pred_const, color=colors[3], label=f
    'Baseline: Constant (MSE = {mse_const:.4f})', marker='.')
```

```
plt.title('Predicted␣vs␣Actual␣Scores␣(Value)')
plt.xlabel('Actual␣Rating')
# plt.xlim(2,5)
plt.ylabel('Predicted␣Rating')
# plt.ylim(2,5)
plt.legend()
plt.show()
```