Assignment 5 - Kaggle Competition and Unsupervised Learning

Fides Regina Schwartz

Netid: fs113

Instructions for all assignments can be found here, and is also linked to from the course syllabus.

Total points in the assignment add up to 90; an additional 10 points are allocated to presentation quality.

Learning objectives

Through completing this assignment you will be able to...

1. Apply the full supervised machine learning pipeline of preprocessing, model selection, model performance evaluation and comparison, and model application to a real-world scale dataset
2. Apply clustering techniques to a variety of datasets with diverse distributional properties, gaining an understanding of their strengths and weaknesses and how to tune model parameters
3. Apply PCA and t-SNE for performing dimensionality reduction and data visualization

1

[40 points] Kaggle Classification Competition

You've learned a great deal about supervised learning and now it's time to bring together all that you've learned. You will be competing in a Kaggle Competition along with the rest of the class! Your goal is to predict hotel reservation cancellations based on a number of potentially related factors such as lead time on the booking, time of year, type of room, special requests made, number of children, etc. While you will be asked to take certain steps along the way to your submission, you're encouraged to try creative solutions to this problem and your choices are wide open for you to make your decisions on how to best make the predictions.

IMPORTANT: Follow the link posted on Ed to register for the competition

You can view the public leaderboard anytime here

The Data. The dataset is provided as `a5_q1.pkl` which is a pickle file format, which allows you to load the data directly using the code below; the data can be downloaded from the Kaggle competition website. A data dictionary for the project can be found here and the original paper that describes the dataset can be found here. When you load the data, 5 matrices are provided `X_train_original`, `y_train`, and `X_test_original`, which are the original, unprocessed features and labels for the training set and the test features (the test labels are not provided - that's what you're predicting). Additionally, `X_train_ohe` and `X_test_ohe` are provided which are one-hot-encoded (OHE) versions of the data. The OHE versions OHE processed every categorical variable. This is provided for convenience if you find it helpful, but you're welcome to reprocess the original data other ways if your prefer.

Scoring. You will need to achieve a minimum acceptable level of performance to demonstrate proficiency with using these supervised learning techniques. Beyond that, it's an open competition and scoring in the top three places of the *private leaderboard* will result in **5 bonus points in this assignment** (and the pride of the class!). Note: the Kaggle leaderboard has a public and private component. The public component is viewable throughout the competition, but the private leaderboard is revealed at the end. When you make a submission, you immediately see your submission on the public leaderboard, but that only represents scoring on a fraction of the total collection of test data, the rest remains hidden until the end of the competition to prevent overfitting to the test data through repeated submissions. You will be be allowed to hand-select two eligible submissions for private score, or by default your best two public scoring submissions will be selected for private scoring.

Requirements:

(a) Explore your data. Review and understand your data. Look at it; read up on what the features represent; think through the application domain; visualize statistics from the paper data to understand any key relationships. **There is no output required for this question**, but you are encouraged to explore the data personally before going further.

(b) Preprocess your data. Preprocess your data so it's ready for use for classification and describe what you did and why you did it. Preprocessing may include: normalizing data, handling missing or erroneous values, separating out a validation dataset, preparing categorical variables through one-hot-encoding, etc. To make one step in this process easier, you're provided with a one-hot-encoded version of the data already.

- Comment on each type of preprocessing that you apply and both how and why you apply it.

(c) Select, train, and compare models. Fit at least 5 models to the data. Some of these can be experiments with different hyperparameter-tuned versions of the same model, although all 5 should not be the same type of model. There are no constraints on the types of models, but you're encouraged to explore examples we've discussed in class including:

1. Logistic regression
2. K-nearest neighbors
3. Random Forests
4. Neural networks
5. Support Vector Machines
6. Ensembles of models (e.g. model bagging, boosting, or stacking). `Scikit-learn` offers a number of tools for assisting with this including those for bagging, boosting, and stacking. You're also welcome to explore options beyond the `sklean` universe; for example, some of you may have heard of XGBoost which is a very fast implementation of gradient boosted decision trees that also allows for parallelization.

When selecting models, be aware that some models may take far longer than others to train. Monitor your output and plan your time accordingly.

Assess the classification performance AND computational efficiency of the models you selected:

- Plot the ROC curves and PR curves for your models in two plots: one of ROC curves and one of PR curves. For each of these two plots, compare the performance of the models you selected above and trained on the training data, evaluating them on the validation data. Be sure to plot the line representing random guessing on each plot. One of these models should also be your BEST performing submission on the Kaggle public leaderboard (see below). In the legends of each, include the area under the curve for each model (limit to 3 significant figures). For the ROC curve, this is the AUC; for the PR curve, this is the average precision (AP).
- As you train and validate each model time how long it takes to train and validate in each case and create a plot that shows both the training and prediction time for each model included in the ROC and PR curves.
- Describe:
  - Your process of model selection and hyperparameter tuning
  - Which model performed best and your process for identifying/selecting it

(d) Apply your model "in practice". Make *at least* 5 submissions of different model results to the competition (more submissions are encouraged and you can submit up to 10 per day!). These do not need to be the same that you report on above, but you should select your *most competitive* models.

- Produce submissions by applying your model on the test data.
- Be sure to RETRAIN YOUR MODEL ON ALL LABELED TRAINING AND VALIDATION DATA before making your predictions on the test data for submission. This will help to maximize your performance on the test data.

- In order to get full credit on this problem you must achieve an AUC on the Kaggle public leaderboard above the "Benchmark" score on the public leaderboard.

Guidance:

1. **Preprocessing**. You may need to preprocess the data for some of these models to perform well (scaling inputs or reducing dimensionality). Some of this preprocessing may differ from model to model to achieve the best performance. A helpful tool for creating such preprocessing and model fitting pipelines is the sklearn `pipeline` module which lets you group a series of processing steps together.
2. **Hyperparameters**. Hyperparameters may need to be tuned for some of the model you use. You may want to perform hyperparameter tuning for some of the models. If you experiment with different hyperparameters that include many model runs, you may want to apply them to a small subsample of your overall data before running it on the larger training set to be time efficient (if you do, just make sure to ensure your selected subset is representative of the rest of your data).
3. **Validation data**. You're encouraged to create your own validation dataset for comparing model performance; without this, there's a significant likelihood of overfitting to the data. A common choice of the split is 80% training, 20% validation. Before you make your final predictions on the test data, be sure to retrain your model on the entire dataset.
4. **Training time**. This is a larger dataset than you've worked with previously in this class, so training times may be higher that what you've experienced in the past. Plan ahead and get your model pipeline working early so you can experiment with the models you use for this problem and have time to let them run.

Starter code

Below is some code for (1) loading the data and (2) once you have predictions in the form of confidence scores for those classifiers, to produce submission files for Kaggle.

(a) Explore your data. Review and understand your data. Look at it; read up on what the features represent; think through the application domain; visualize statistics from the paper data to understand any key relationships. **There is no output required for this question**, but you are encouraged to explore the data personally before going further.

In [ ]:

```
import sys
import os
import warnings

if not sys.warnoptions:
    warnings.simplefilter("ignore")
    os.environ["PYTHONWARNINGS"] = "ignore"
```

In [ ]:

```python
import pandas as pd
import numpy as np
import pickle

#################################
# Load the data
#################################
data = pickle.load(open("C:/Users/dm93/Desktop/IDS705/ids705-a5-2022/a5_q1.pkl", "rb"))

y_train = data["y_train"]
X_train_original = data["X_train"]  # Original dataset
X_train_ohe = data["X_train_ohe"]  # One-hot-encoded dataset

X_test_original = data["X_test"]
X_test_ohe = data["X_test_ohe"]
```

Review data

In [ ]:

```python
print(y_train)
```

```
0         0
2         0
3         0
4         0
5         0
         ..
119383    0
119384    0
119387    0
119388    0
119389    0
Name: is_canceled, Length: 95512, dtype: int64
```

In [ ]:

```python
print(X_train_ohe.shape)
print(y_train.shape)
print(X_test_ohe.shape)
```

```
(95512, 940)
(95512,)
(23878, 940)
```

The total observations from the paper describe 119,390 datapoints and that is the amount of data contained in the training and test datasets in total, too. This confirms that I have the full dataset available to me.

In [ ]:
```
X_train_ohe.head()
```

Out[ ]:

	lead_time	arrival_date_year	arrival_date_week_number	arrival_date_day_of_month	stays_in_weekend_nights	stays_in_week_nights	adults	chil
0	342	2015	27	1	0	0	2	
2	7	2015	27	1	0	1	1	
3	13	2015	27	1	0	1	1	
4	14	2015	27	1	0	2	2	
5	14	2015	27	1	0	2	2	

5 rows × 940 columns

In [ ]:
```
# Look at summary statistics
X_train_original.head()
```

Out[ ]:

	hotel	lead_time	arrival_date_year	arrival_date_month	arrival_date_week_number	arrival_date_day_of_month	stays_in_weekend_nights	stays_
0	Resort Hotel	342	2015	July	27	1	0	
2	Resort Hotel	7	2015	July	27	1	0	
3	Resort Hotel	13	2015	July	27	1	0	
4	Resort Hotel	14	2015	July	27	1	0	
5	Resort Hotel	14	2015	July	27	1	0	

5 rows × 29 columns

In [ ]:

```
X_train_original.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 95512 entries, 0 to 119389
Data columns (total 29 columns):
 #   Column                          Non-Null Count  Dtype  
---  ------                          --------------  -----  
 0   hotel                           95512 non-null  object 
 1   lead_time                       95512 non-null  int64  
 2   arrival_date_year               95512 non-null  int64  
 3   arrival_date_month              95512 non-null  object 
 4   arrival_date_week_number        95512 non-null  int64  
 5   arrival_date_day_of_month       95512 non-null  int64  
 6   stays_in_weekend_nights         95512 non-null  int64  
 7   stays_in_week_nights            95512 non-null  int64  
 8   adults                          95512 non-null  int64  
 9   children                        95510 non-null  float64
 10  babies                          95512 non-null  int64  
 11  meal                            95512 non-null  object 
 12  country                         95117 non-null  object 
 13  market_segment                  95512 non-null  object 
 14  distribution_channel            95512 non-null  object 
 15  is_repeated_guest               95512 non-null  int64  
 16  previous_cancellations          95512 non-null  int64  
 17  previous_bookings_not_canceled  95512 non-null  int64  
 18  reserved_room_type              95512 non-null  object 
 19  assigned_room_type              95512 non-null  object 
 20  booking_changes                 95512 non-null  int64  
 21  deposit_type                    95512 non-null  object 
 22  agent                           82431 non-null  float64
 23  company                         5453 non-null   float64
 24  days_in_waiting_list            95512 non-null  int64  
 25  customer_type                   95512 non-null  object 
 26  adr                             95512 non-null  float64
 27  required_car_parking_spaces     95512 non-null  int64  
 28  total_of_special_requests       95512 non-null  int64  
dtypes: float64(4), int64(15), object(10)
memory usage: 21.9+ MB
```

In [ ]:

```
look_up = {
    "January": "01",
    "February": "02",
    "March": "03",
```

```
    "April": "04",
    "May": "05",
    "June": "06",
    "July": "07",
    "August": "08",
    "September": "09",
    "October": "10",
    "November": "11",
    "December": "12",
}

X_train_original["arrival_date_month"] = X_train_original["arrival_date_month"].apply(
    lambda x: look_up[x]
)
```

In [ ]:

```
X_train_original.rename(
    columns={
        "arrival_date_year": "year",
        "arrival_date_month": "month",
        "arrival_date_day_of_month": "day",
    },
    inplace=True,
)
```

In [ ]:

```
X_train_original["Date"] = pd.to_datetime(X_train_original[["year", "month", "day"]])
X_train_original.head()
```

Out[ ]:

	hotel	lead_time	year	month	arrival_date_week_number	day	stays_in_weekend_nights	stays_in_week_nights	adults	children	...	booking
0	Resort Hotel	342	2015	07	27	1	0	0	2	0.0	...	
2	Resort Hotel	7	2015	07	27	1	0	1	1	0.0	...	
3	Resort Hotel	13	2015	07	27	1	0	1	1	0.0	...	
4	Resort Hotel	14	2015	07	27	1	0	2	2	0.0	...	
5	Resort Hotel	14	2015	07	27	1	0	2	2	0.0	...	

5 rows × 30 columns

In [ ]:

```
# Create a full dataframe that contains the is_canceled category = y_train
df = pd.merge(X_train_original, y_train, left_index=True, right_index=True)
```

In [ ]:

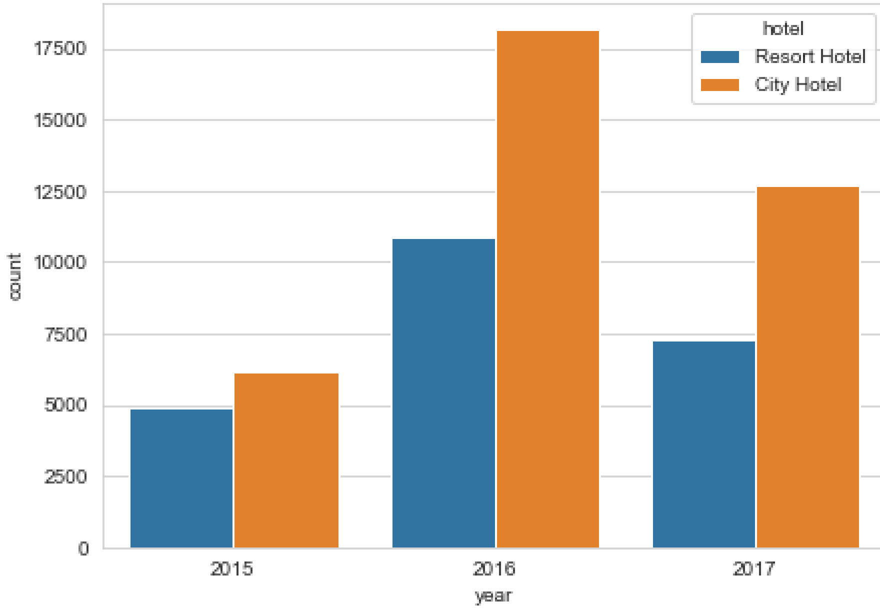
```
# Separate out only the data that is not canceled to look at trends
df_not_canceled = df[df["is_canceled"] == 0]
```

In [ ]:

```
import matplotlib.pyplot as plt
import numpy as np
from matplotlib import colors

# from matplotlib.ticker import PercentFormatter
import seaborn as sns

# import pycountry as pc
# import matplotlib.ticker as mtick

# Look at bookings that were not canceled for the years included in the analysis by type of hotel
plt.subplots(figsize=(7, 5))
sns.countplot(x="year", hue="hotel", data=df_not_canceled)
```

Out[ ]:

```
<AxesSubplot:xlabel='year', ylabel='count'>
```



There seem to be relevant differences between both the years and the hotel types in how many nights were booked and actually stayed at the hotels. These variables should probably be in the models.

In [ ]:

```
# Set up easy counting for the next steps
def get_count(series, limit=None):

    if limit != None:
        series = series.value_counts()[:limit]
    else:
        series = series.value_counts()

    x = series.index
    y = series / series.sum() * 100

    return x.values, y.values
```

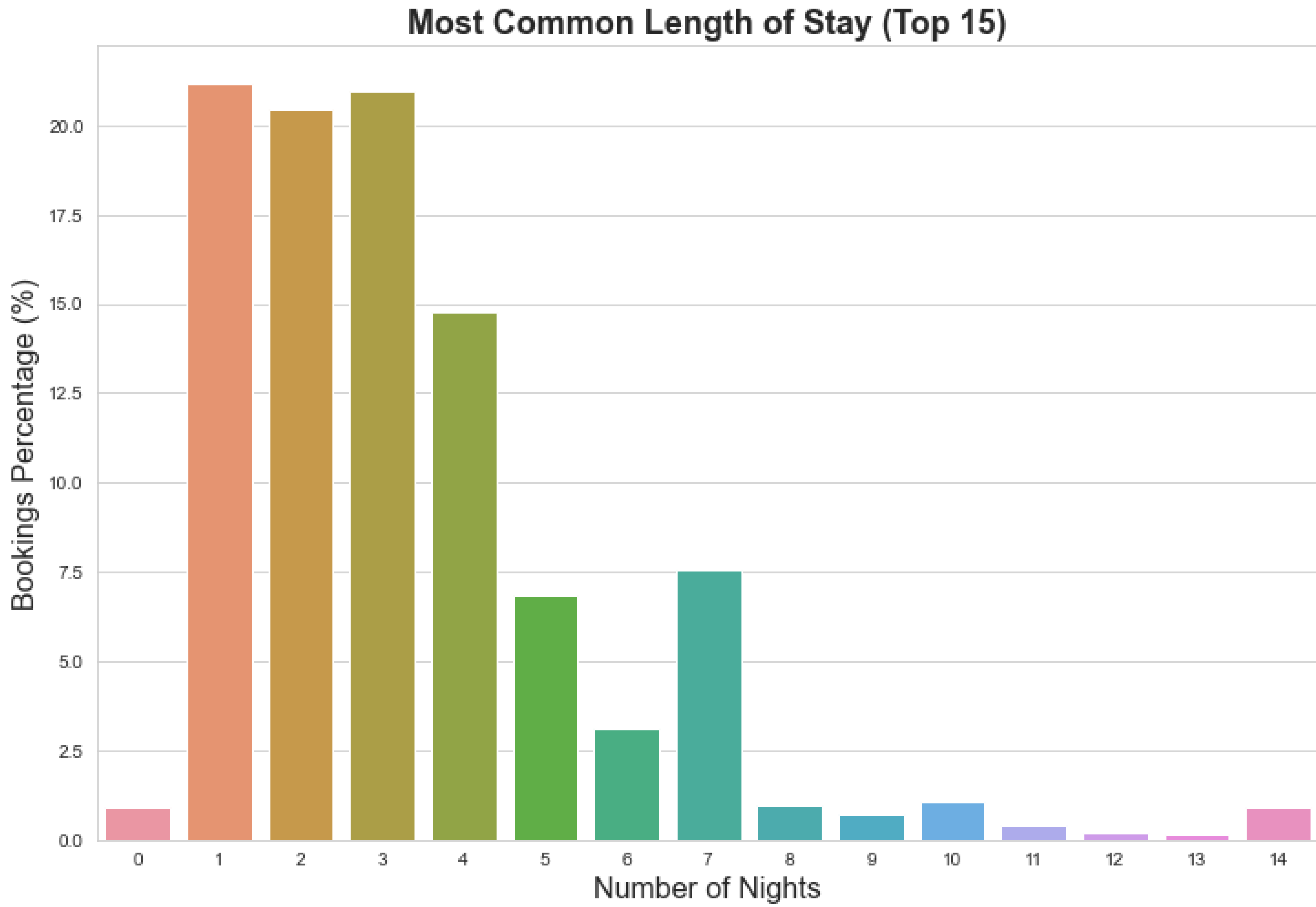
In [ ]:

```
# Set up various plot options for plotting going forward
def plot(x, y, x_label=None, y_label=None, title=None, figsize=(12, 8), type="bar"):

    sns.set_style("whitegrid")

    fig, ax = plt.subplots(figsize=figsize)
```

```python
    # ax.yaxis.set_major_formatter(mtick.PercentFormatter())

    if x_label != None:
        ax.set_xlabel(x_label, fontsize=16)

    if y_label != None:
        ax.set_ylabel(y_label, fontsize=16)

    if title != None:
        ax.set_title(title, fontsize=18, fontweight="bold")

    if type == "bar":
        sns.barplot(x, y, ax=ax)
    elif type == "line":
        sns.lineplot(x, y, ax=ax, sort=False)

    plt.show()
```

In [ ]:

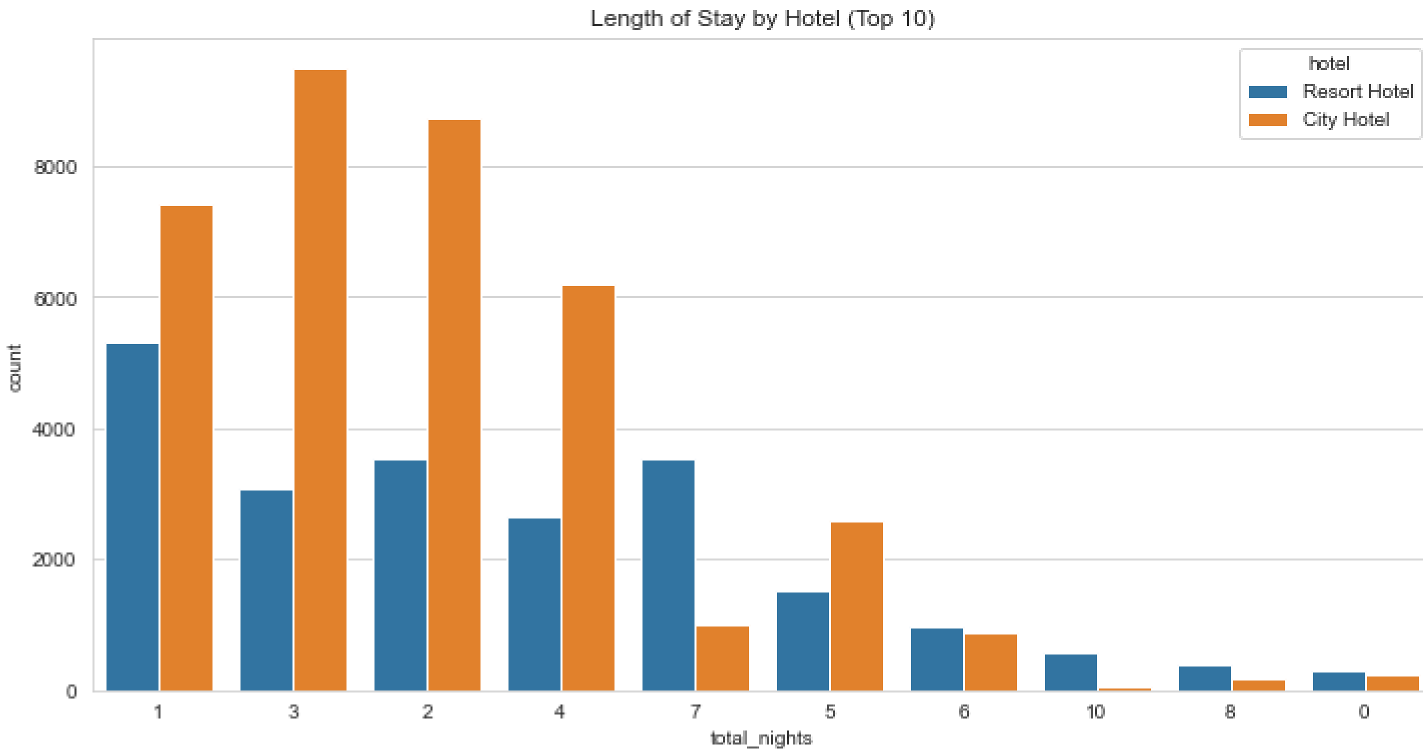
```python
# Look at distribution of non-canceled stays between weeknights and weekend nights
total_nights = (
    df_not_canceled["stays_in_weekend_nights"] + df_not_canceled["stays_in_week_nights"]
)
x, y = get_count(total_nights, limit=15)

plot(
    x,
    y,
    x_label="Number of Nights",
    y_label="Bookings Percentage (%)",
    title="Most Common Length of Stay (Top 15)",
    figsize=(12, 8),
)
```



There seem to be substantial differences in stays between weekend nights and weeknights, and longer stays seem to generally be less frequent, so this also seems like a good variable to keep for the models.

In [ ]:
```
# Look at length of stay based on hotel type
df_not_canceled.loc[:, "total_nights"] = (
    df_not_canceled["stays_in_weekend_nights"] + df_not_canceled["stays_in_week_nights"]
)

fig, ax = plt.subplots(figsize=(12, 6))
ax.set_xlabel("Number of Nights")
ax.set_ylabel("Number of Nights")
ax.set_title("Length of Stay by Hotel (Top 10)")
```

```
sns.countplot(
    x="total_nights",
    hue="hotel",
    data=df_not_canceled,
    order=df_not_canceled.total_nights.value_counts().iloc[:10].index,
    ax=ax,
)
```

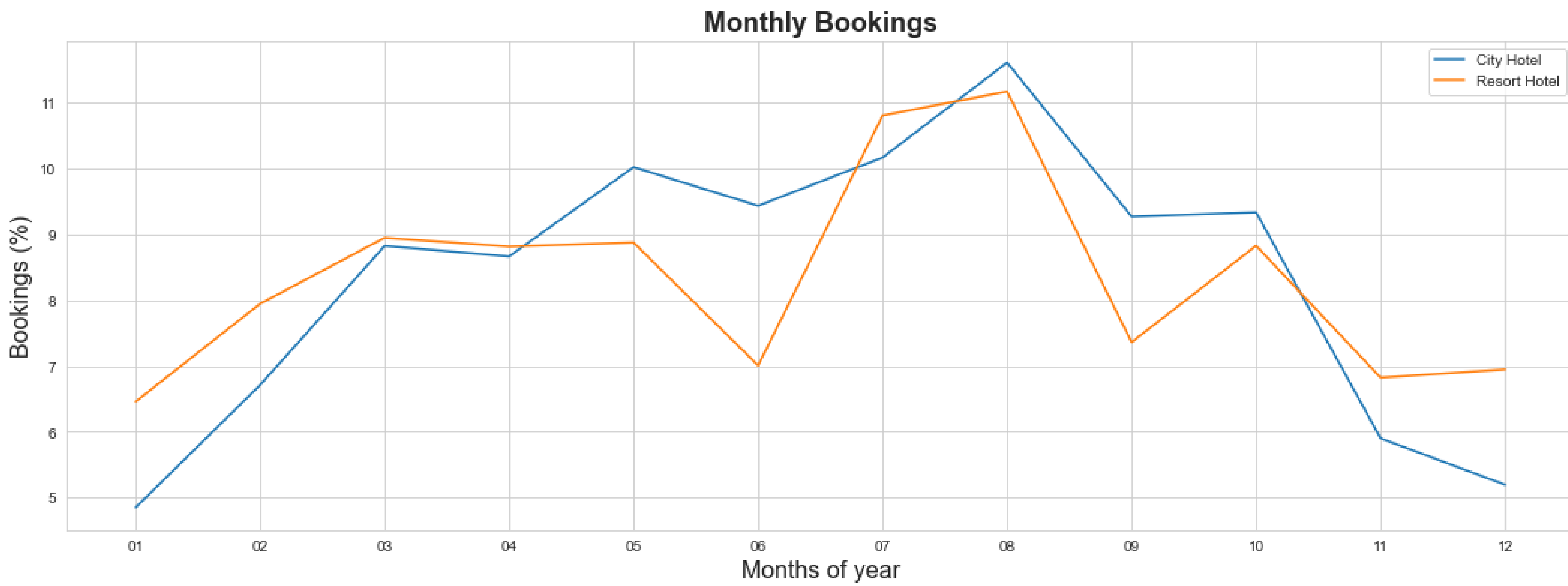


The length of stay varies between hotel types, with resort hotels mostly booked for weeklong stays and longer, whereas city hotels are booked for shorter stays on average.

In [ ]:

```
# Look at booking (not canceled) by month
# Look at booking trends separately for

# Sort months from January to December in ascending order
month_sort = ["01", "02", "03", "04", "05", "06", "07", "08", "09", "10", "11", "12"]

sorted_months = df_not_canceled["month"].value_counts().reindex(month_sort)
```

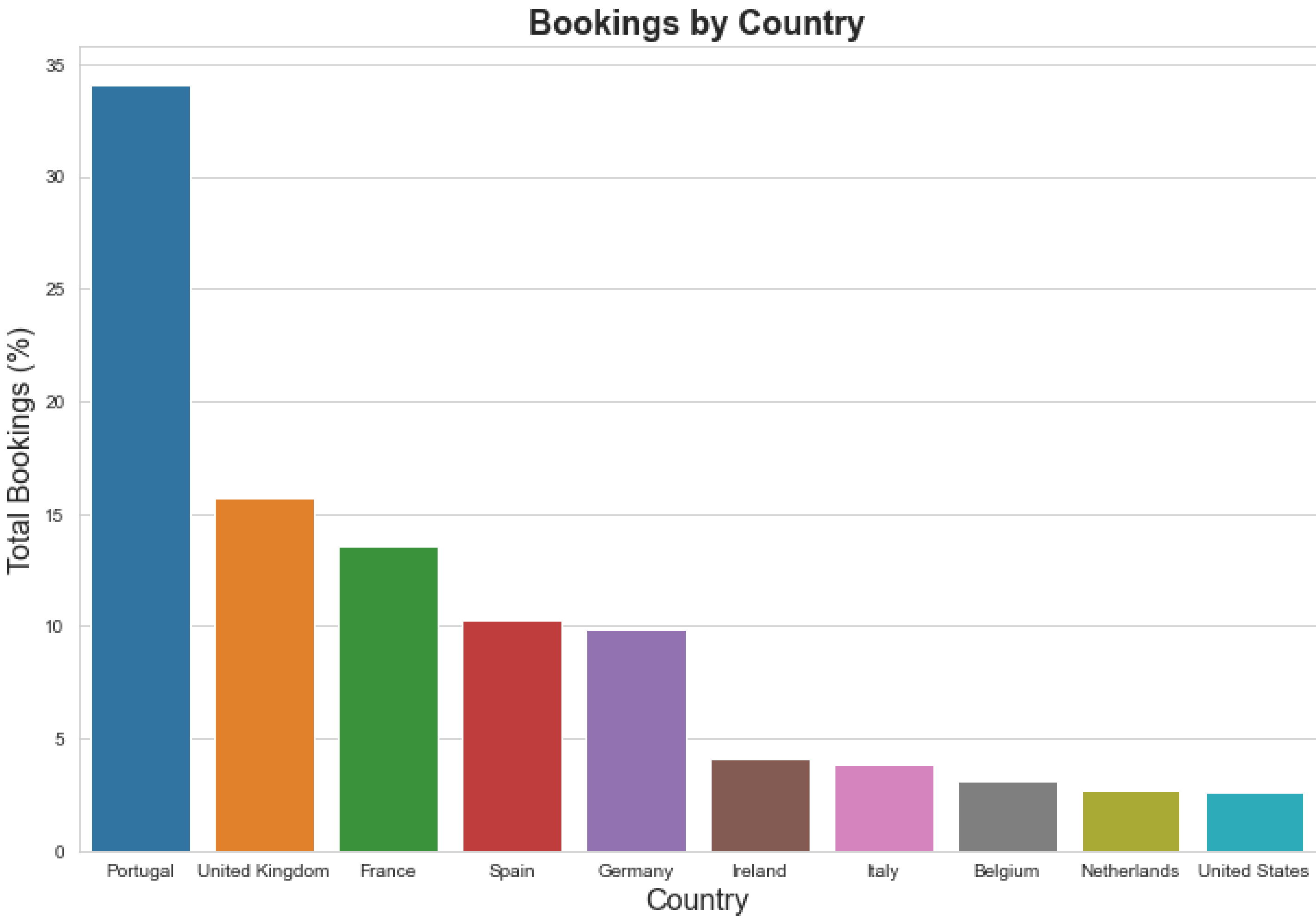
```
x = sorted_months.index
y = sorted_months / sorted_months.sum() * 100

# Select only City Hotel
sorted_months = (
    df_not_canceled.loc[df.hotel == "City Hotel", "month"]
    .value_counts()
    .reindex(month_sort)
)

x1 = sorted_months.index
y1 = sorted_months / sorted_months.sum() * 100

# Select only Resort Hotel
sorted_months = (
    df_not_canceled.loc[df.hotel == "Resort Hotel", "month"]
    .value_counts()
    .reindex(month_sort)
)

x2 = sorted_months.index
y2 = sorted_months / sorted_months.sum() * 100

# Draw the line plot
fig, ax = plt.subplots(figsize=(18, 6))

ax.set_xlabel("Months of year", fontsize=16)
ax.set_ylabel("Bookings (%)", fontsize=16)
ax.set_title("Monthly Bookings", fontsize=18, fontweight="bold")

sns.lineplot(x1, y1.values, label="City Hotel", sort=False)
sns.lineplot(x1, y2.values, label="Resort Hotel", sort=False)

plt.show()
```



There is clearly variation in the not-canceled bookings over the months and per hotel type. In addition, there seems to be a lot of variation in the weekend and weeknight stays over time, so it seems reasonable to include these as a variables for our models.

In [ ]:

```
# Look at distribution by country from which bookings came
import pycountry as pc

x, y = get_count(df_not_canceled["country"], limit=10)

## For each country code select the country name
country_name = [pc.countries.get(alpha_3=name).name for name in x]

plot(
    country_name,
    y,
    x_label="Country",
    y_label="Total Bookings (%)",
    title="Bookings by Country",
    figsize=(12, 8),
)
```

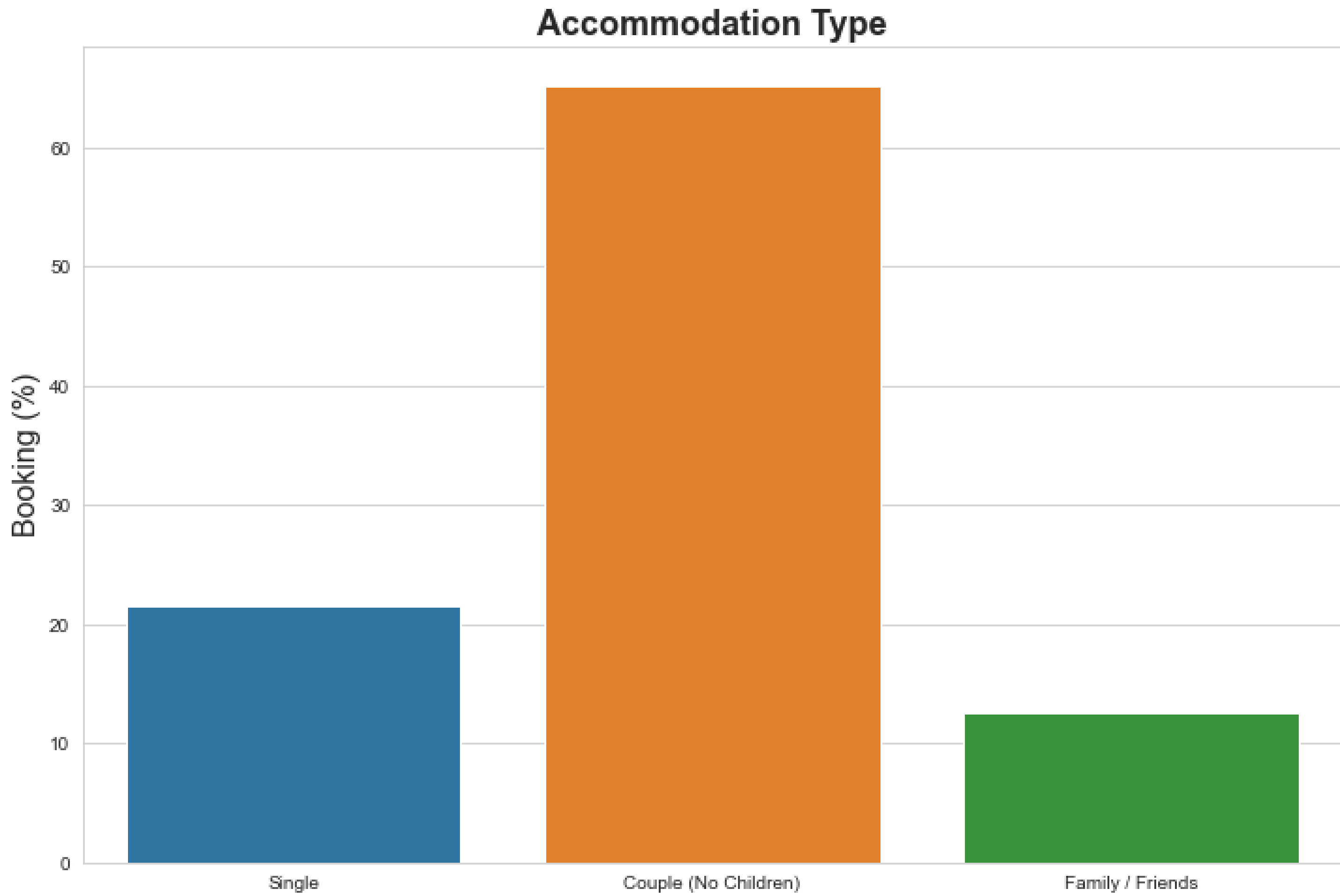


Since these are Portugese hotels, it makes sense that the largest percentage of bookings comes from there. There do seem to be substantial differences between the bookings from other countries, so this variable should also be considered for the models.

In [ ]:

```
# Look at distribution of non-canceled bookings based on number of adults and children

# Select single, couple, multiple adults and family
single = df_not_canceled[
    (df_not_canceled.adults == 1)
    & (df_not_canceled.children == 0)
    & (df_not_canceled.babies == 0)
]
couple = df_not_canceled[
```

```
    (df_not_canceled.adults == 2)
    & (df_not_canceled.children == 0)
    & (df_not_canceled.babies == 0)
]
family = df_not_canceled[
    df_not_canceled.adults + df_not_canceled.children + df_not_canceled.babies > 2
]


# Make the list of Category names, and their total percentage
names = ["Single", "Couple (No Children)", "Family / Friends"]
count = [single.shape[0], couple.shape[0], family.shape[0]]
count_percent = [x / df_not_canceled.shape[0] * 100 for x in count]


# Plot
plot(
    names,
    count_percent,
    y_label="Booking (%)",
    title="Accommodation Type",
    figsize=(12, 8),
)
```

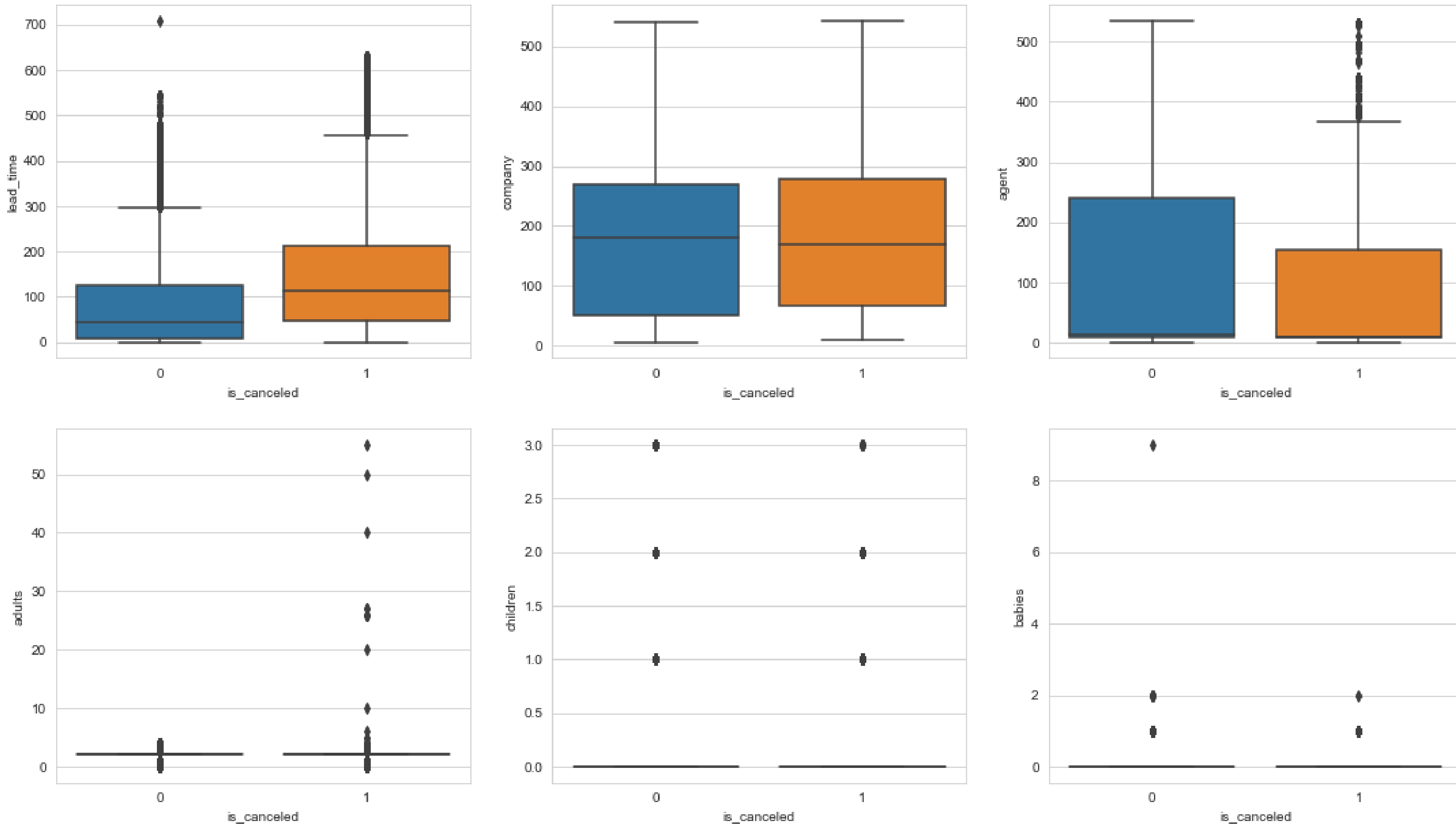


There are stark differences between the bookings made based on the number of adults and children, these are definitely categories that should be included in the models.

In [ ]:
```
# Look at differences between canceled and non-canceled for continous variables
f, axes = plt.subplots(2, 3)
f.set_size_inches(18.5, 10.5)
sns.boxplot(x=y_train, y=X_train_original["lead_time"], orient="v", ax=axes[0, 0])
sns.boxplot(x=y_train, y=X_train_original["company"], orient="v", ax=axes[0, 1])
sns.boxplot(x=y_train, y=X_train_original["agent"], orient="v", ax=axes[0, 2])
sns.boxplot(x=y_train, y=X_train_original["adults"], orient="v", ax=axes[1, 0])
sns.boxplot(x=y_train, y=X_train_original["children"], orient="v", ax=axes[1, 1])
sns.boxplot(x=y_train, y=X_train_original["babies"], orient="v", ax=axes[1, 2])
```

Out[ ]:

```
<AxesSubplot:xlabel='is_canceled', ylabel='babies'>
```



There seem to be substantial differences between mean and confidence intervals for lead time but not for the company or the agent that made the booking.

In [ ]:

```python
# Find the missing value, show the total null values for each column and sort it in descending order
# X_train_original.isnull().sum().sort_values(ascending=False)[:10]
X_train_original.isnull().sum() / X_train_original.shape[0]
```

Out[ ]:

```
hotel                         0.000000
lead_time                     0.000000
year                          0.000000
month                         0.000000
```

```
arrival_date_week_number          0.000000
day                               0.000000
stays_in_weekend_nights           0.000000
stays_in_week_nights              0.000000
adults                            0.000000
children                          0.000021
babies                            0.000000
meal                              0.000000
country                           0.004136
market_segment                    0.000000
distribution_channel              0.000000
is_repeated_guest                 0.000000
previous_cancellations            0.000000
previous_bookings_not_canceled    0.000000
reserved_room_type                0.000000
assigned_room_type                0.000000
booking_changes                   0.000000
deposit_type                      0.000000
agent                             0.136957
company                           0.942908
days_in_waiting_list              0.000000
customer_type                     0.000000
adr                               0.000000
required_car_parking_spaces       0.000000
total_of_special_requests         0.000000
Date                              0.000000
dtype: float64
```

The variables "agent" and "company" do not seem to show clear differences in the number of bookings canceled and also have the highest number of missing values, so they will be dropped from the analysis.

In [ ]:

```
# Look at percentage of canceled stays for categorical values
# make one merge for all following evaluations
new_df = pd.merge(X_train_original, y_train, left_index=True, right_index=True)
```

In [ ]:

```
# Variable market segment
market_segment = new_df.groupby("market_segment")["is_canceled"].mean().reset_index()
print(market_segment)
```

```
  market_segment  is_canceled
0       Aviation     0.202186
1  Complementary     0.128978
2      Corporate     0.187984
```

```
3         Direct     0.151967
4         Groups     0.610775
5   Offline TA/TO    0.342808
6       Online TA    0.367868
7       Undefined    1.000000
```

The market segments seem to vary strongly in their cancellation rates (e.g. groups with 61% and direct bookings with only 15%), so this should be included in the models.

In [ ]:

```
# Variable distribution channel
distribution_channel = new_df.groupby('distribution_channel')['is_canceled'].mean().reset_index()
distribution_channel
```

Out[ ]:

	distribution_channel	is_canceled
0	Corporate	0.218585
1	Direct	0.173026
2	GDS	0.180124
3	TA/TO	0.410754
4	Undefined	0.666667

There is some variation in cancellations between distribution channels, should be included.

In [ ]:

```
# Variable repeat guests
is_repeated_guest = new_df.groupby('is_repeated_guest')['is_canceled'].mean().reset_index()
is_repeated_guest
```

Out[ ]:

	is_repeated_guest	is_canceled
0	0	0.377916
1	1	0.143804

Repeat guests have fewer cancellations than first guests, this category should be included in the analysis.

In [ ]:

```
# Variable previous cancellations
previous_cancellations = new_df.groupby('previous_cancellations')['is_canceled'].mean().reset_index()
previous_cancellations
```

Out[ ]:

	previous_cancellations	is_canceled
0	0	0.339451
1	1	0.942958
2	2	0.333333
3	3	0.285714
4	4	0.222222
5	5	0.117647
6	6	0.187500
7	11	0.225806
8	13	0.875000
9	14	1.000000
10	19	1.000000
11	21	1.000000
12	24	1.000000
13	25	1.000000
14	26	1.000000

Previous cancellations vary strongly and seem to be a strong indicator for future cancellations, so this will be included in the model.

In [ ]:

```
# Variable previous non-cancellations
previous_bookings_not_canceled = new_df.groupby('previous_bookings_not_canceled')['is_canceled'].mean().reset_index()
previous_bookings_not_canceled
```

Out[ ]:

	previous_bookings_not_canceled	is_canceled
0	0	0.380301
1	1	0.053235
2	2	0.056769
3	3	0.057471

	previous_bookings_not_canceled	is_canceled
4	4	0.054348
...	...	...
63	68	0.000000
64	69	0.000000
65	70	0.000000
66	71	0.000000
67	72	0.000000

68 rows × 2 columns

This seems to be a weaker predictor than previous cancellations and seems to be somewhat redundant with the previous cancellations category, so this will be dropped.

In [ ]:

```
# Variable reserved room type
reserved_room_type = new_df.groupby('reserved_room_type')['is_canceled'].mean().reset_index()
reserved_room_type
```

Out[ ]:

	reserved_room_type	is_canceled
0	A	0.390524
1	B	0.329268
2	C	0.324900
3	D	0.320302
4	E	0.292299
5	F	0.299430
6	G	0.369423
7	H	0.433610
8	L	0.400000
9	P	1.000000

In [ ]:

```
# Variable assigned room type
assigned_room_type = new_df.groupby('assigned_room_type')['is_canceled'].mean().reset_index()
assigned_room_type
```

Out[ ]:

	assigned_room_type	is_canceled
0	A	0.443539
1	B	0.238506
2	C	0.183438
3	D	0.254098
4	E	0.253059
5	F	0.243463
6	G	0.310429
7	H	0.368696
8	I	0.006944
9	K	0.049774
10	L	1.000000
11	P	1.000000

Cancellations by booked room type do not seem to vary very strongly but they do vary substantially by assigned room type, so this might still be a valuable variable.

In [ ]:

```
# Variable booking changes
booking_changes = new_df.groupby('booking_changes')['is_canceled'].mean().reset_index()
booking_changes
```

Out[ ]:

	booking_changes	is_canceled
0	0	0.408474
1	1	0.143967
2	2	0.201125

	booking_changes	is_canceled
3	3	0.155102
4	4	0.160772
5	5	0.170455
6	6	0.270833
7	7	0.125000
8	8	0.230769
9	9	0.250000
10	10	0.166667
11	11	0.000000
12	12	0.000000
13	13	0.000000
14	14	0.250000
15	15	0.000000
16	16	0.500000
17	17	0.000000
18	18	0.000000
19	20	0.000000
20	21	0.000000

Changes made to the booking seem to be a strong indicator for cancellations and should be included in the models.

In [ ]:
```
# Variable deposit type
deposit_type = new_df.groupby('deposit_type')['is_canceled'].mean().reset_index()
deposit_type
```

Out[ ]:

	deposit_type	is_canceled
0	No Deposit	0.284135

	deposit_type	is_canceled
1	Non Refund	0.993900
2	Refundable	0.200000

There are marked differences between deposit type and percentage of cancellation, so this is definitely important for the models.

In [ ]:
```
# Variable days in wait list (before booking confirmation)
days_in_waiting_list  = new_df.groupby('days_in_waiting_list')['is_canceled'].mean().reset_index()
days_in_waiting_list
```

Out[ ]:

	days_in_waiting_list	is_canceled
0	0	0.362102
1	1	0.181818
2	2	0.200000
3	3	1.000000
4	4	0.210526
...	...	...
121	236	0.181818
122	259	0.000000
123	330	0.083333
124	379	0.642857
125	391	1.000000

126 rows × 2 columns

The length of time it takes for confirmation of a booking does seem to influence cancellations and should be kept for the models.

In [ ]:
```
# Variable customer type
customer_type  = new_df.groupby('customer_type')['is_canceled'].mean().reset_index()
customer_type
```

Out[ ]:

	customer_type	is_canceled
0	Contract	0.310557
1	Group	0.100218
2	Transient	0.407526
3	Transient-Party	0.254595

Cancellations vary strongly by customer type and should be kept for the models.

In [ ]:

```
# variable average daily rate
adr = new_df.groupby('adr')['is_canceled'].mean().reset_index()
adr
```

Out[ ]:

	adr	is_canceled
0	-6.38	0.000000
1	0.00	0.104220
2	0.26	0.000000
3	0.50	1.000000
4	1.00	0.166667
...	...	...
7953	426.25	0.000000
7954	450.00	1.000000
7955	451.50	0.000000
7956	510.00	0.000000
7957	5400.00	1.000000

7958 rows × 2 columns

Cancellations vary a lot by the average daily room rate, so should be kept as a variable.

In [ ]:

```
# Variable car parking spaces required by customer
```

```
required_car_parking_spaces = new_df.groupby('required_car_parking_spaces')['is_canceled'].mean().reset_index()
required_car_parking_spaces
```

Out[ ]:

	required_car_parking_spaces	is_canceled
0	0	0.39509
1	1	0.00000
2	2	0.00000
3	3	0.00000
4	8	0.00000

Cancellations by required parking spaces seem to be very different between people who do or do not require a parking space, so this should be kept.

In [ ]:

```
# Variable special requests
total_of_special_requests = new_df.groupby('total_of_special_requests')['is_canceled'].mean().reset_index()
total_of_special_requests
```

Out[ ]:

	total_of_special_requests	is_canceled
0	0	0.476766
1	1	0.220388
2	2	0.223002
3	3	0.181773
4	4	0.098113
5	5	0.033333

The number of special requests seem to influence differences between cancellation rates and should be kept for the models.

ANSWER (a) After reviewing the data, I believe that most factors in our original dataset are important to predict cancellations in Portugese hotels (Algarve vs Lisbon) but a few may be dropped to avoid over-fitting:

1. Agent
2. Company

3. Assigned room type
4. Reserved room type
5. Previous bookings not canceled

(b) Preprocess your data. Preprocess your data so it's ready for use for classification and describe what you did and why you did it. Preprocessing may include: normalizing data, handling missing or erroneous values, separating out a validation dataset, preparing categorical variables through one-hot-encoding, etc. To make one step in this process easier, you're provided with a one-hot-encoded version of the data already.

- Comment on each type of preprocessing that you apply and both how and why you apply it.

In [ ]:

```
# Look at where there are no adults, babies or children recorded for X_train
print(
    X_train_original[
        (X_train_original.adults + X_train_original.babies + X_train_original.children)
        == 0
    ].index
)
```

```
Int64Index([  2224,   2409,   3181,   3684,   3708,   9376,  31765,  32029,
             32827,  34849,
            ...
            111710, 112471, 112558, 113188, 114583, 114908, 114911, 115029,
            115091, 117087],
           dtype='int64', length=136)
```

In [ ]:

```
# Look at where there are no adults, babies or children recorded for y_train
print(
    y_train[
        (X_train_original.adults + X_train_original.babies + X_train_original.children)
        == 0
    ].index
)
```

```
Int64Index([  2224,   2409,   3181,   3684,   3708,   9376,  31765,  32029,
             32827,  34849,
            ...
            111710, 112471, 112558, 113188, 114583, 114908, 114911, 115029,
            115091, 117087],
           dtype='int64', length=136)
```

In [ ]:
```
# Copy the dataframe for safekeeping
X_train_original_subset = X_train_original.copy()
```

In [ ]:
```
# Drop Rows where there is no adult, baby and child
#X_train_original = X_train_original.drop(X_train_original[(X_train_original.adults+X_train_original.babies+X_train_origi
#y_train = y_train.drop(y_train[(X_train_original.adults+X_train_original.babies+X_train_original.children)==0].index)
```

In [ ]:
```
# Fill the two spots where there are Nan in the children column with the mean of all other children
X_train_original_subset["children"].fillna(
    round(X_train_original.children.mean()), inplace=True
)
```

In [ ]:
```
# Fill the two spots where there are Nan in the children column with the mean of all other children
X_train_ohe['children'].fillna(round(X_train_ohe.children.mean()), inplace=True)
```

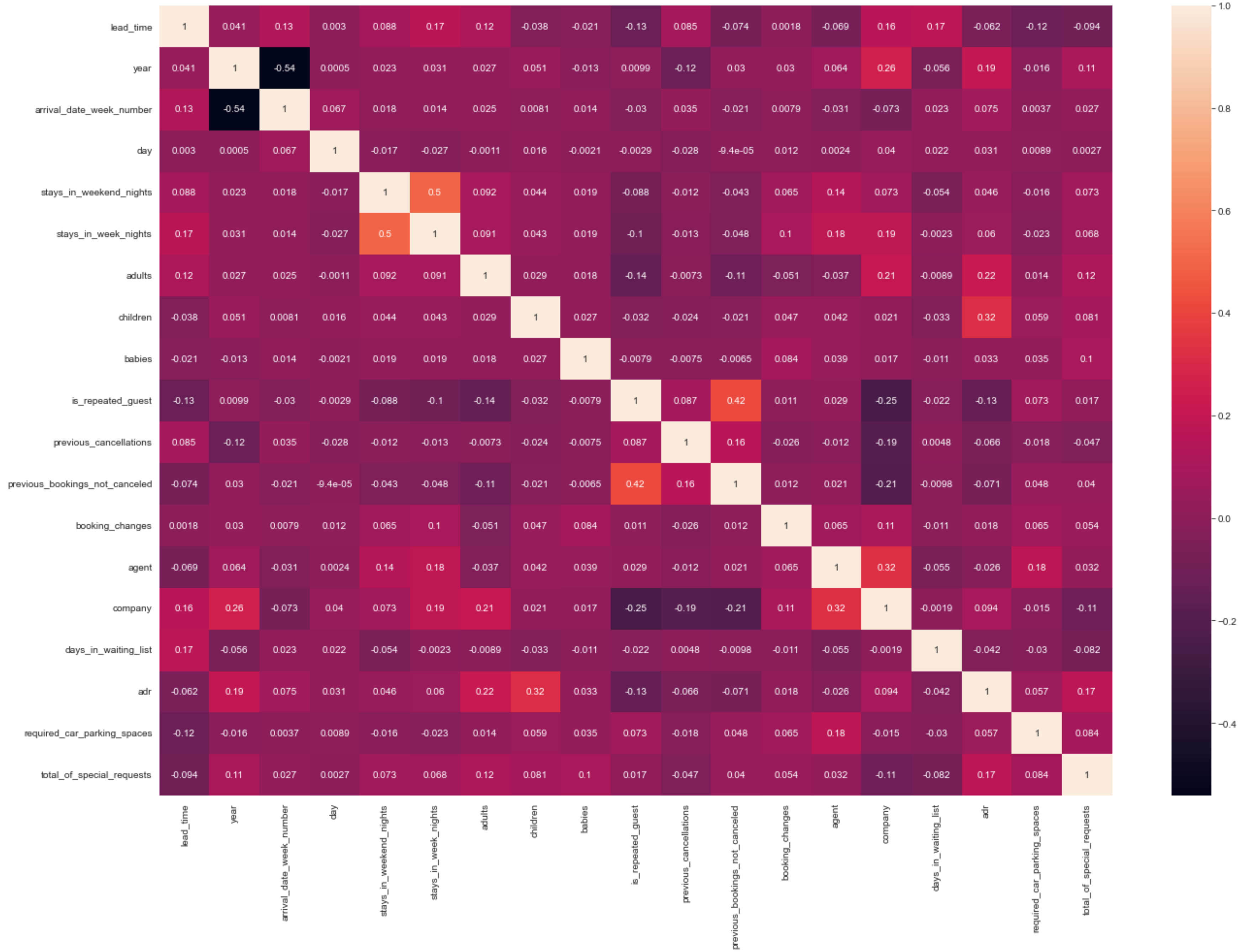
In [ ]:
```
# Fill the two spots where there are Nan in the children column with the mean of all other children
X_test_ohe["children"].fillna(round(X_test_ohe.children.mean()), inplace=True)
```

There are two NA values in the dataset in the column "children". I replace these with the mean of all the other rows.

In [ ]:
```
# convert datatype of float columns from float to integer
X_train_original_subset[["children"]] = X_train_original_subset[["children"]].astype(
    "int64"
)
```

In [ ]:
```
# Plot a heatmap to see correlation with columns (imitating the r-code from the paper)
fig, ax = plt.subplots(figsize=(22, 15))
sns.heatmap(X_train_original_subset.corr(), annot=True, ax=ax)
```

Out[ ]:
```
<AxesSubplot:>
```



This is the closest I can get to a similar output to the r-code that was used in the paper. I think it mostly confirms my EDA and will go ahead with dropping the planned variables.

In [ ]:

```
# Remove the less important features from training set
X_train_original_subset = X_train_original_subset.drop(
    [
        "agent",
        "arrival_date_week_number",
        "company",
        "assigned_room_type",
        "reserved_room_type",
        "previous_bookings_not_canceled",
    ],
    axis=1,
)
```

In [ ]:

```
# Remove the less important features from test set
X_test_original = X_test_original.drop(
    [
        "agent",
        "arrival_date_week_number",
        "company",
        "assigned_room_type",
        "reserved_room_type",
        "previous_bookings_not_canceled",
    ],
    axis=1,
)
```

In [ ]:

```
# turn all categorical values into one hot encoded values


def transform(dataframe):

    ## Import LabelEncoder from sklearn
    from sklearn.preprocessing import OneHotEncoder

    ohe = OneHotEncoder()

    ## Select all categorcial features
    categorical_features = list(dataframe.columns[dataframe.dtypes == object])

    ## Apply one hot Encoding on all categorcial features
    return dataframe[categorical_features].apply(lambda x: ohe.fit_transform(x))
```

```
X_train_original_subset = transform(X_train_original_subset)
```

In [ ]:

```
# turn all categorical values into one hot encoded values

def transform(dataframe):


    ## Import LabelEncoder from sklearn
    from sklearn.preprocessing import OneHotEncoder

    ohe = OneHotEncoder()


    ## Select all categorcial features
    categorical_features = list(dataframe.columns[dataframe.dtypes == object])


    ## Apply one hot Encoding on all categorical features
    return dataframe[categorical_features].apply(lambda x: ohe.fit_transform(x))

X_test_original = transform(X_test_original)
```

In [ ]:

```
# Split training dataset into training and validation dataset
from sklearn.model_selection import train_test_split

X_train, X_val, y_train2, y_val = train_test_split(
    X_train_original_subset, y_train, test_size = 0.2, random_state=2018
) # split with an 80/20 ratio
```

In [ ]:

```
X_train_original_subset.shape
```

Out[ ]:

```
(95512, 8)
```

ANSWER (b)

I did several iterations of preprocessing after my exploratory data analysis.

1. Normalizing the data:

i) I dropped the categories from my data that did not seem to be making a large impact on cancellations (e.g. agent, which had the same mean) or were included in other values (e.g. week of year, since I still have another date parameter).

2. Drop missing and erroneous values

   i) since there was no missing data, based on the database this data was pulled from, none could be dropped directly. Entries that were presented as NULL were considered as not applicable and dropped, though.

   ii) in addition, there were two NaN in the children column. I replaced those with the mean of the rest of the children's columns to avoid having to drop them altogether.

3. Prepare categorical variables

   i) I one hot encoded the categorical variables after the initial dropping and replacing of values in step 1 and 2.

4. Split training data into training and validation data using an 80/20 split. This is to evaluate performance of my models based on a held-out dataset before it gets applied to the true test data.

(c) Select, train, and compare models. Fit at least 5 models to the data. Some of these can be experiments with different hyperparameter-tuned versions of the same model, although all 5 should not be the same type of model.

In [ ]:

```
# Import all the important modules
from pandas import read_csv  # For dataframes
from pandas import DataFrame  # For dataframes
from numpy import ravel  # For matrices
import matplotlib.pyplot as plt  # For plotting data
import seaborn as sns  # For plotting data
from sklearn.model_selection import train_test_split  # For train/test splits
from sklearn.linear_model import (
    LogisticRegression,
)  # The logistic regression classifier
from sklearn.neighbors import KNeighborsClassifier  # The k-nearest neighbor classifier

from sklearn.feature_selection import VarianceThreshold  # Feature selector
from sklearn.pipeline import Pipeline  # For setting up pipeline

# Various pre-processing steps
from sklearn.preprocessing import (
    Normalizer,
    StandardScaler,
```

```python
    MinMaxScaler,
    PowerTransformer,
    MaxAbsScaler,
    LabelEncoder,
)
from sklearn.model_selection import GridSearchCV  # For optimization
from sklearn.preprocessing import StandardScaler, RobustScaler, QuantileTransformer
from sklearn.feature_selection import SelectKBest, f_regression
from sklearn.decomposition import PCA
from sklearn.linear_model import Ridge
```

1. Logistic regression

In [ ]:

```python
# set up tuning pipeline for logistic regression

logreg = LogisticRegression().fit(X_train, y_train2)

print("Training set score: " + str(logreg.score(X_train, y_train2)))
print("Validation set score: " + str(logreg.score(X_val, y_val)))

pipe = Pipeline(
    [
        ("scaler", StandardScaler()),
        ("classifier", LogisticRegression()),
    ]
)

pipe.fit(X_train, y_train2)

print("Training set score: " + str(pipe.score(X_train, y_train2)))
print("Validation set score: " + str(pipe.score(X_val, y_val)))

parameters = {
    "scaler": [StandardScaler(), MinMaxScaler(), Normalizer(), MaxAbsScaler()],
    "classifier__penalty": ["elasticnet", "none"],
    "classifier__tol": [1e-4, 1e-3, 1e-2],
}

grid = GridSearchCV(pipe, parameters, cv=2).fit(X_train, y_train2)

print("Training set score: " + str(grid.score(X_train, y_train2)))
print("Validation set score: " + str(grid.score(X_val, y_val)))
```

```
# Access the best set of parameters
best_params = grid.best_params_
print(best_params)
# Stores the optimum model in best_pipe
best_pipe_logreg = grid.best_estimator_
print(best_pipe_logreg)

result_df = DataFrame.from_dict(grid.cv_results_, orient="columns")
print(result_df.columns)
```

I wanted to see how logistic regression performs on this dataset because, if a relatively simple model could be used for this task, that would make the predictions faster. I set up a pipeline that I will be able to use for all of the other models that I will test and chose to change the penalty and tolerance in hyperparameter tuning.

In [ ]:
```
import time

# Time the model training
start_time=time.time()
best_pipe_logreg.fit(X_train, y_train2)
end_time=time.time()
time_taken_logreg = end_time-start_time

# Time model validation
start_time=time.time()
best_pipe_logreg.predict(X_val)
end_time=time.time()
time_taken_logreg2 = end_time-start_time
```

In [ ]:
```
print(
    f"The logistic regression model takes {time_taken_logreg} sec to train and {time_taken_logreg2} \
        sec to validate with these parameters: {best_pipe_logreg}."
)
```

```
The logistic regression model takes 0.5455076694488525 to train and 0.0019965171813964844 to validate with these paramete
rs: Pipeline(steps=[('scaler', Normalizer()),
                ('classifier', LogisticRegression(penalty='none'))]).
```

In [ ]:
```
final_logreg = best_pipe_logreg.fit(X_train_original_subset, y_train)
```

In [ ]:
```
print(final_logreg)
```

```
Pipeline(steps=[('scaler', Normalizer()),
                ('classifier', LogisticRegression(penalty='none'))])
```

In [ ]:

```
final_logreg.predict_proba(X_test_original)[:, 1]
```

Out[ ]:

```
array([0.97076472, 0.24445629, 0.31753734, ..., 0.34512487, 0.26207812,
       0.29508237])
```

In [ ]:

```
################################
# Produce submission general code
################################

def create_submission(confidence_scores, save_path):
    '''Creates an output file of submissions for Kaggle

    Parameters
    ----------
    confidence_scores : list or numpy array
        Confidence scores (from predict_proba methods from classifiers) or
        binary predictions (only recommended in cases when predict_proba is
        not available)
    save_path : string
        File path for where to save the submission file.

    Example:
    create_submission(my_confidence_scores, './data/submission.csv')

    '''
    import pandas as pd

    submission = pd.DataFrame({"score":confidence_scores})
    submission.to_csv(save_path, index_label="id")
```

In [ ]:

```
create_submission(final_logreg.predict_proba(X_test_original)[:, 1], "logreg.csv")
```

1. K-nearest neighbors

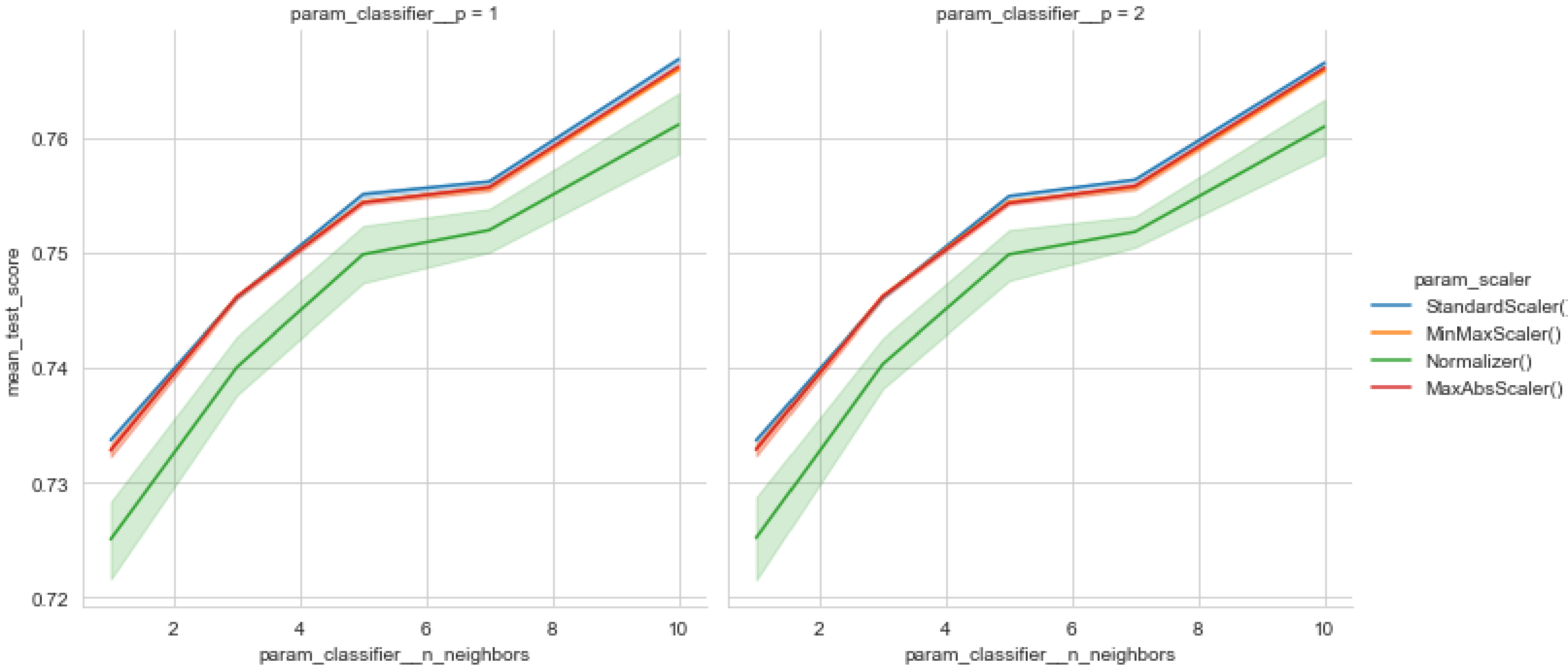
In [ ]:

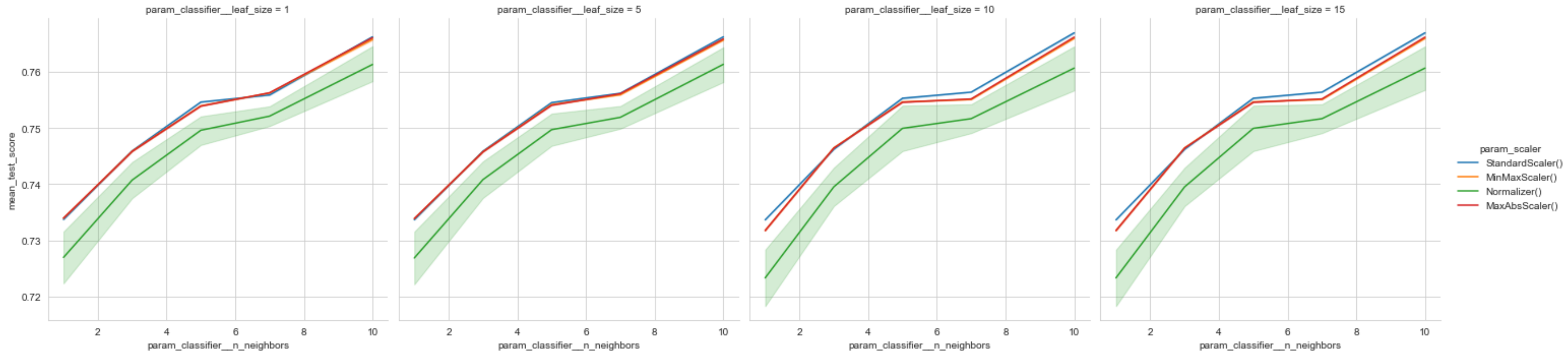
```
# set up tuning pipleine for KNN
knn = KNeighborsClassifier().fit(X_train, y_train2)
```

```
print("Training set score: " + str(knn.score(X_train, y_train2)))
print("Validation set score: " + str(knn.score(X_val, y_val)))

pipe = Pipeline(
    [
        ("scaler", StandardScaler()),
        ("selector", VarianceThreshold()),
        ("classifier", KNeighborsClassifier()),
    ]
)

pipe.fit(X_train, y_train2)

print("Training set score: " + str(pipe.score(X_train, y_train2)))
print("Validation set score: " + str(pipe.score(X_val, y_val)))

parameters = {
    "scaler": [StandardScaler(), MinMaxScaler(), Normalizer(), MaxAbsScaler()],
    "selector__threshold": [0, 0.01],
    "classifier__n_neighbors": [1, 5, 10],
    "classifier__p": [1, 2],
    "classifier__leaf_size": [1, 10, 15],
}

grid = GridSearchCV(pipe, parameters, cv=2).fit(X_train, y_train2)

print("Training set score: " + str(grid.score(X_train, y_train2)))
print("Validation set score: " + str(grid.score(X_val, y_val)))

# Access the best set of parameters
best_params = grid.best_params_
print(best_params)
# Stores the optimum model in best_pipe
best_pipe_knn = grid.best_estimator_
print(best_pipe)

result_df = DataFrame.from_dict(grid.cv_results_, orient="columns")
print(result_df.columns)
```

```
Training set score: 0.7701187032941146
Validation set score: 0.7567397790922892
Training set score: 0.7678807470324176
Validation set score: 0.7546458671412867
Training set score: 0.7808111609888887
Validation set score: 0.7698790765848296
```

```
{'classifier__leaf_size': 10, 'classifier__n_neighbors': 10, 'classifier__p': 1, 'scaler': StandardScaler(), 'selector__t
hreshold': 0}
Pipeline(steps=[('scaler', StandardScaler()),
                ('selector', VarianceThreshold(threshold=0)),
                ('classifier',
                 KNeighborsClassifier(leaf_size=10, n_neighbors=10, p=1))])
Index(['mean_fit_time', 'std_fit_time', 'mean_score_time', 'std_score_time',
       'param_classifier__leaf_size', 'param_classifier__n_neighbors',
       'param_classifier__p', 'param_scaler', 'param_selector__threshold',
       'params', 'split0_test_score', 'split1_test_score', 'mean_test_score',
       'std_test_score', 'rank_test_score'],
      dtype='object')
```





I wanted to evaluate KNN as it is still a relatively simple method but might produce better results than logistic regression. I tested various scalers, selector thresholds and varied the number of neighbors during hyperparameter tuning.

In [ ]:
```
import time

# Time the model training
start_time = time.time()
best_pipe.fit(X_train, y_train2)
end_time = time.time()
time_taken = end_time - start_time

# Time model validation
start_time = time.time()
best_pipe.predict(X_val)
end_time = time.time()
time_taken2 = end_time - start_time
```

```
0.48969101905822754
```

In [ ]:
```
print(f"The KNN model takes {time_taken} sec to train and {time_taken2} sec to validate with these parameters {best_pipe}
```

In [ ]:
```
final_knn = best_pipe.fit(X_train_original_subset, y_train)
```

In [ ]:
```
print(final_knn)
```

```
Pipeline(steps=[('scaler', StandardScaler()),
                ('selector', VarianceThreshold(threshold=0)),
```

```
                ('classifier',
                 KNeighborsClassifier(leaf_size=10, n_neighbors=10, p=1))])
```

In [ ]:
```
final_knn.predict_proba(X_test_original)[:, 1]
```

Out[ ]:
```
array([1. , 0.7, 0.3, ..., 0.5, 0.2, 0.6])
```

In [ ]:
```
# create submission KNN
create_submission(final_knn.predict_proba(X_test_original)[:, 1], "knn.csv")
```

1. Random Forests

In [ ]:
```
from sklearn.ensemble import RandomForestClassifier
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2

# set up tuning pipleine for Random Forest
rf = RandomForestClassifier().fit(X_train, y_train2)
print("Training set score: " + str(rf.score(X_train, y_train2)))
print("Validation set score: " + str(rf.score(X_val, y_val)))

pipe = Pipeline(
    [
        ("classifier", RandomForestClassifier()),
    ]
)

pipe.fit(X_train, y_train2)

print("Training set score: " + str(pipe.score(X_train, y_train2)))
print("Validation set score: " + str(pipe.score(X_val, y_val)))

parameters = {
    "classifier__n_estimators": [50, 100, 500, None],
    #'classifier__bootstrap': [True, False],
    #'classifier__max_features': ['auto', 'sqrt'],
    "classifier__min_samples_leaf": [1, 2, 4],
    #'classifier__min_samples_split': [2, 5, 10],
    "classifier__n_estimators": [200, 500, 1000],
}
```

```
grid = GridSearchCV(pipe, parameters, cv=2).fit(X_train, y_train2)

print("Training set score: " + str(grid.score(X_train, y_train2)))
print("Validation set score: " + str(grid.score(X_val, y_val)))

# Access the best set of parameters
best_params = grid.best_params_
print(best_params)
# Stores the optimum model in best_pipe
best_pipe_rf = grid.best_estimator_
print(best_pipe_rf)

result_df = DataFrame.from_dict(grid.cv_results_, orient="columns")
print(result_df.columns)
```

```
Training set score: 0.8030860238977083
Validation set score: 0.7843270690467465
Training set score: 0.8030729364341896
Validation set score: 0.7845888080406219
Training set score: 0.7941734612414768
Validation set score: 0.7840653300528713
{'classifier__min_samples_leaf': 4, 'classifier__n_estimators': 200}
Pipeline(steps=[('classifier',
                 RandomForestClassifier(min_samples_leaf=4, n_estimators=200))])
Index(['mean_fit_time', 'std_fit_time', 'mean_score_time', 'std_score_time',
       'param_classifier__min_samples_leaf', 'param_classifier__n_estimators',
       'params', 'split0_test_score', 'split1_test_score', 'mean_test_score',
       'std_test_score', 'rank_test_score'],
      dtype='object')
```

I tested random forest model as it is quite sophisticated in dealing with multiple different categories of continuous and categorical data. I had to remove some of the hyperparameter categories to make this run within a reasonable timeframe and the performance was not very impressive.

In [ ]:

```
# Time the model training
start_time=time.time()
best_pipe_rf.fit(X_train, y_train2)
end_time=time.time()
time_taken_rf = end_time-start_time

# Time model validation
start_time=time.time()
best_pipe_rf.predict(X_val)
end_time=time.time()
```

```
time_taken_rf2 = end_time-start_time

print(f"The random forest model takes {time_taken_rf} to train and {time_taken_rf2} \
     to validate with these parameters {best_pipe_rf}.")
```

```
The random forest model takes 5.524221420288086 to train and 0.5246322154998779 to validate with these parameters Pipelin
e(steps=[('classifier',
                 RandomForestClassifier(min_samples_leaf=4, n_estimators=200))]).
```

In [ ]:

```
# retrain on full dataset
final_rf = best_pipe_rf.fit(X_train_original_subset, y_train)
final_rf.predict_proba(X_test_original)[:, 1]
```

Out[ ]:

```
array([0.84694712, 0.44060116, 0.05109488, ..., 0.32336308, 0.04514153,
       0.33992993])
```

In [ ]:

```
# Create sumbission random forest
create_submission(final_rf.predict_proba(X_test_original)[:, 1], "rf.csv")
```

Random Forest with OHE data

In [ ]:

```
# Split training dataset into training and validation dataset
from sklearn.model_selection import train_test_split

X_train3, X_val3, y_train3, y_val3 = train_test_split(
    X_train_ohe, y_train, test_size = 0.2, random_state=2018
) # split with an 80/20 ratio
```

In [ ]:

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2

# set up tuning pipleine for Random Forest
rf2 = RandomForestClassifier().fit(X_train3, y_train3)
print('Training set score: ' + str(rf2.score(X_train3,y_train3)))
print('Validation set score: ' + str(rf2.score(X_val3,y_val3)))

pipe = Pipeline([
('classifier', RandomForestClassifier()),
])
```

```
pipe.fit(X_train3, y_train3)

print('Training set score: ' + str(pipe.score(X_train3,y_train3)))
print('Validation set score: ' + str(pipe.score(X_val3,y_val3)))

parameters = {
        'classifier__n_estimators': [50, 100, 200, 500, 1000],
}

grid = GridSearchCV(pipe, parameters, cv=2).fit(X_train3, y_train3)

print('Training set score: ' + str(grid.score(X_train3, y_train3)))
print('Validation set score: ' + str(grid.score(X_val3, y_val3)))

# Access the best set of parameters
best_params = grid.best_params_
print(best_params)
# Stores the optimum model in best_pipe
best_pipe_rf2 = grid.best_estimator_
print(best_pipe_rf2)

result_df = DataFrame.from_dict(grid.cv_results_, orient='columns')
print(result_df.columns)
```

```
Training set score: 0.9962046355795783
Validation set score: 0.8909595351515469
Training set score: 0.9962308105066157
Validation set score: 0.8922682301209234
Training set score: 0.9962308105066157
Validation set score: 0.892739360309899
{'classifier__n_estimators': 1000}
Pipeline(steps=[('classifier', RandomForestClassifier(n_estimators=1000))])
Index(['mean_fit_time', 'std_fit_time', 'mean_score_time', 'std_score_time',
       'param_classifier__n_estimators', 'params', 'split0_test_score',
       'split1_test_score', 'mean_test_score', 'std_test_score',
       'rank_test_score'],
      dtype='object')
```

Since none of the first three models I tested on my own data preprocessing produced very good results, I decided to try random forest with the provided one-hot encoded date for simplicity. To get this to run within less than two hours, I chose to only tune the n_estimator hyperparameter, which produced a very good result on the test dataset.

In [ ]:

```
import time
```

```
# Time the model training
start_time=time.time()
best_pipe_rf2.fit(X_train3, y_train3)
end_time=time.time()
time_taken_rf3 = end_time-start_time

# Time model validation
start_time=time.time()
best_pipe_rf2.predict(X_val3)
end_time=time.time()
time_taken_rf4 = end_time-start_time

print(f"The random forest model takes {time_taken_rf3} sec to train and {time_taken_rf4} \
    sec to validate with these parameters {best_pipe_rf2}.")
```

```
The random forest model takes 520.7484061717987 to train and 8.51977014541626 to validate with these parameters Pipeline
(steps=[('classifier', RandomForestClassifier(n_estimators=1000))]).
```

In [ ]:

```
# retrain on full dataset
final_rf2 = best_pipe_rf2.fit(X_train_ohe, y_train)
```

In [ ]:

```
final_rf2.predict_proba(X_test_ohe)[:, 1]
```

Out[ ]:

```
array([1.        , 0.126     , 0.2025    , ..., 0.696     , 0.035     ,
       0.47456667])
```

In [ ]:

```
from sklearn import metrics
import matplotlib.pyplot as plt

fpr, tpr, thresholds = metrics.roc_curve(y_val3, final_rf2.predict_proba(X_val3)[:,1])
auc = metrics.roc_auc_score(y_val3, final_rf2.predict(X_val3))
```

In [ ]:

```
# Create sumbission random forest for ohe data
create_submission(final_rf2.predict_proba(X_test_ohe)[:, 1], "rf2.csv")
```

1. Neural networks

In [ ]:

```
from sklearn.neural_network import MLPClassifier
```

```
# set up tuning pipleine for neural network
nn = MLPClassifier().fit(X_train, y_train2)
print("Training set score: " + str(nn.score(X_train, y_train2)))
print("Validation set score: " + str(nn.score(X_val, y_val)))

pipe = Pipeline(steps=[("classifier", MLPClassifier())])

pipe.fit(X_train, y_train2)

print("Training set score: " + str(pipe.score(X_train, y_train2)))
print("Validation set score: " + str(pipe.score(X_val, y_val)))

parameters = {
    "classifier__hidden_layer_sizes": [1, 50, 100],
    "classifier__learning_rate_init": [0.001, 0.01, 0.1],
    "classifier__max_iter": [100, 200],
    "classifier__tol": [1e-4, 1e-3, 1e-2],
}

grid = GridSearchCV(pipe, parameters, cv=2).fit(X_train, y_train2)

print("Training set score: " + str(grid.score(X_train, y_train2)))
print("Validation set score: " + str(grid.score(X_val, y_val)))

# Access the best set of parameters
best_params = grid.best_params_
print(best_params)
# Stores the optimum model in best_pipe
best_pipe_nn = grid.best_estimator_
print(best_pipe_nn)

result_df = DataFrame.from_dict(grid.cv_results_, orient="columns")
print(result_df.columns)
```

I tested a neural network on the original data, which also didn't perform very well, in spite of the level of sophistication these models provide. This leads me to use the ohe data for the following models. I chose several hyperparameters to tune but ended up with the default values for all of them.

In [ ]:

```
# Time the model training
start_time=time.time()
best_pipe_nn.fit(X_train, y_train2)
end_time=time.time()
time_taken_nn = end_time-start_time
```

```
# Time model validation
start_time=time.time()
best_pipe_nn.predict(X_val)
end_time=time.time()
time_taken_nn2 = end_time-start_time

print(f"The neural network takes {time_taken_nn} sec to train and {time_taken_nn2} \
    sec to validate with these parameters {best_pipe_nn}.")
```

```
The neural network takes 54.46628785133362 to train and 0.027926921844482422 to validate with these parameters Pipeline(s
teps=[('classifier', MLPClassifier(hidden_layer_sizes=100))]).
```

In [ ]:

```
# retrain on full dataset
final_nn = best_pipe_nn.fit(X_train_original_subset, y_train)
final_nn.predict_proba(X_test_original)[:, 1]
```

Out[ ]:

```
array([0.99637743, 0.25000901, 0.1999333 , ..., 0.2702155 , 0.12957874,
       0.23017665])
```

In [ ]:

```
# create submission neural network
create_submission(final_nn.predict_proba(X_test_original)[:, 1], "nn.csv")
```

1. Support Vector Machines

In [ ]:

```
from sklearn.svm import SVC
from sklearn.experimental import enable_halving_search_cv
from sklearn.model_selection import HalvingGridSearchCV

# set up tuning pipleine for SVC
SVM = SVC().fit(X_train3, y_train3)
print("Training set score: " + str(SVM.score(X_train3, y_train3)))
print("Validation set score: " + str(SVM.score(X_val3, y_val3)))

pipe = Pipeline([("scaler", StandardScaler()), ("classifier", SVC())])

pipe.fit(X_train3, y_train3)

print("Training set score: " + str(pipe.score(X_train3, y_train3)))
print("Validation set score: " + str(pipe.score(X_val3, y_val3)))

parameters = {
```

```
    "classifier__C": [0.1, 1, 1000],
    "classifier__kernel": ["rbf"],
    # "classifier__degree": [1, 3, 6],
    "classifier__gamma": [10, 1, 0.001],
}

grid = HalvingGridSearchCV(pipe, parameters, cv=2).fit(X_train3, y_train3)

print("Training set score: " + str(grid.score(X_train3, y_train3)))
print("Validation set score: " + str(grid.score(X_val3, y_val3)))

# Access the best set of parameters
best_params = grid.best_params_
print(best_params)

# Stores the optimum model in best_pipe
best_pipe_svm = grid.best_estimator_
print(best_pipe_svm)

result_df = DataFrame.from_dict(grid.cv_results_, orient="columns")
print(result_df.columns)
```

I tried to test the support vector machine because I thought a supervised model that is meant for classification and regression tasks in high dimensional spaces would potentially do better than a neural network or the simpler methods tested before. I tried several penalties to see whether a "hard" or "soft" boundary works better, and since I'm using the rbf classifier, I need to tune the gamma parameter simultaneously. This ran for more than 200 minutes before I aborted it and decided to proceed with ensembles of models instead.

1. Ensembles of models (e.g. model bagging, boosting, or stacking)

In [ ]:

```
from sklearn.ensemble import GradientBoostingClassifier
import time

# Time the model training
start_time=time.time()
GBC = GradientBoostingClassifier().fit(X_train3, y_train3)
end_time=time.time()
time_taken_svm = end_time-start_time

# Time model validation
start_time=time.time()
GBC.predict(X_val3)
end_time=time.time()
```

```python
time_taken_svm2 = end_time-start_time

print(f"The GBC takes {time_taken_svm} sec to train and {time_taken_svm2} sec to validate.")
```

```
The SVM takes 108.14406418800354 sec to train and 0.2543182373046875 sec to validate.
```

In [ ]:
```python
# retrain on full dataset
final_gbc = GBC.fit(X_train_ohe, y_train)
final_gbc.predict_proba(X_test_ohe)[:, 1]

fpr2, tpr2, thresholds2 = metrics.roc_curve(y_val3, final_gbc.predict_proba(X_val3)[:,1])
auc2 = metrics.roc_auc_score(y_val3, final_gbc.predict(X_val3))
```

In [ ]:
```python
# create submission SVM
create_submission(final_gbc.predict_proba(X_test_ohe)[:, 1], "gbc.csv")
```

In [ ]:
```python
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.neural_network import MLPClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import StackingClassifier

start_time=time.time()

estimators = [
    ('rf', RandomForestClassifier(min_samples_leaf=1, n_estimators=1000)),
    ('nn', MLPClassifier(hidden_layer_sizes=100)),
]
clf = StackingClassifier(
    estimators=estimators, final_estimator=GradientBoostingClassifier()
)
clf.fit(X_train3, y_train3)

end_time=time.time()
time_taken_stack = end_time-start_time
```

I tried stacking my best random forest, neural network and gradient boosting classifier to try and boost performance overall.

In [ ]:
```python
# Time model validation
start_time=time.time()
clf.predict(X_val3)
```

```
end_time=time.time()
time_taken_stack2 = end_time-start_time

print(f"The stacked model takes {time_taken_stack} sec to train and {time_taken_stack2} sec to validate.")
```

```
The stacked model takes 3575.090641260147 sec to train and 8.227524042129517 sec to validate.
```

In [ ]:

```
# retrain on full dataset
final_stack = clf.fit(X_train_ohe, y_train)
final_stack.predict_proba(X_test_ohe)[:, 1]

fpr3, tpr3, thresholds3 = metrics.roc_curve(y_val3, final_stack.predict_proba(X_val3)[:,1])
auc3 = metrics.roc_auc_score(y_val3, final_stack.predict(X_val3))
```

In [ ]:

```
# create submission stacked model
create_submission(final_stack.predict_proba(X_test_ohe)[:, 1], "stack2.csv")
```

In [ ]:

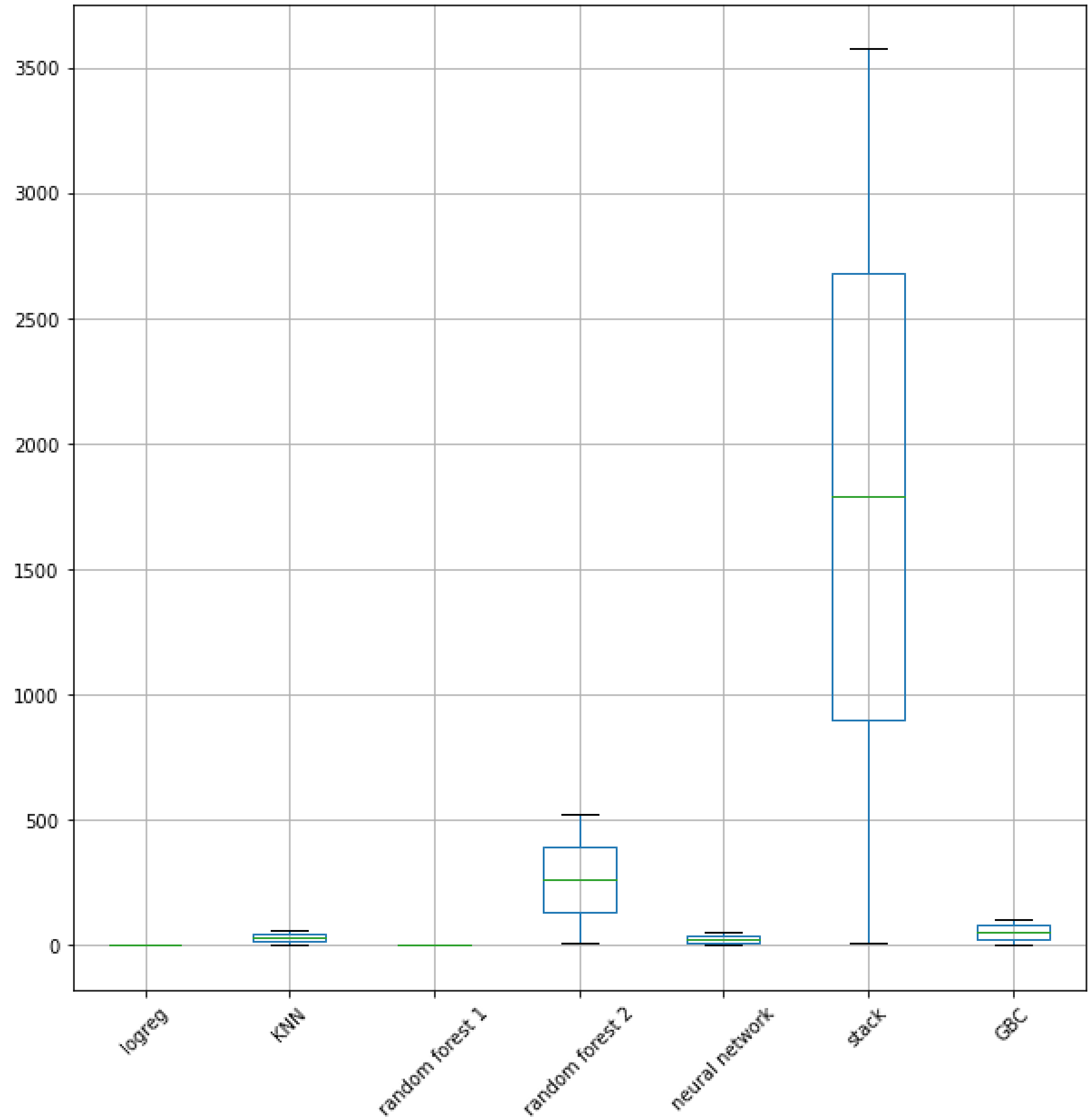
```
df = pd.DataFrame(
    [
        ("logreg", [0.5455076694488525, 0.0019965171813964844]),
        ("KNN", [57.75706325, 0.04]),
        ("random forest 1", [5.524221420288086, 0.5246322154998779]),
        ("random forest 2", [520.7484061717987, 8.51977014541626]),
        ("neural network", [54.46628785133362, 0.027926921844482422]),
        ("stack", [3575.090641260147, 8.227524042129517]),
        ("GBC", [108.14406418800354, 0.2543182373046875]),
    ],
    columns=["Model", "Values"],
).set_index("Model")
```

In [ ]:

```
df['Values'].apply(lambda x: pd.Series(x)).T.boxplot(figsize=(10,10),rot=45)
```

Out[ ]:

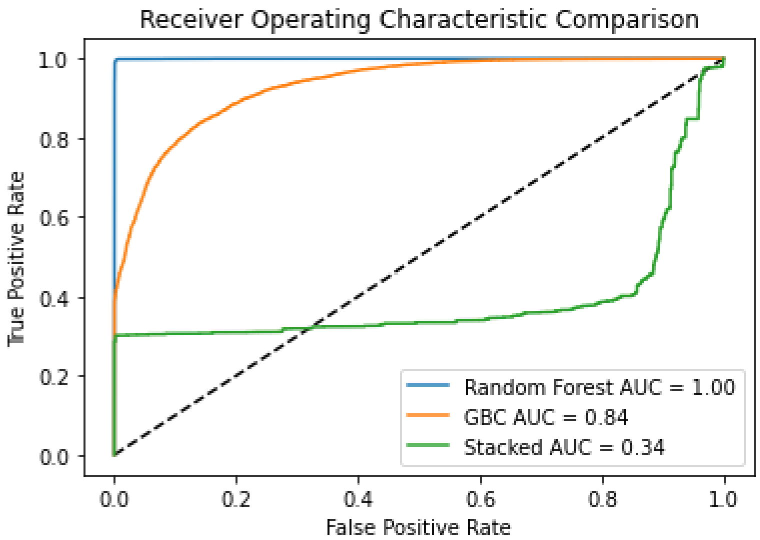
```
<AxesSubplot:>
```



Plot ROC and PR curves for the three best models - Including: Random guessing, AUC and AP + training and prediction time for full training dataset (train + val)

In [ ]:
```
from sklearn.metrics import plot_roc_curve
```

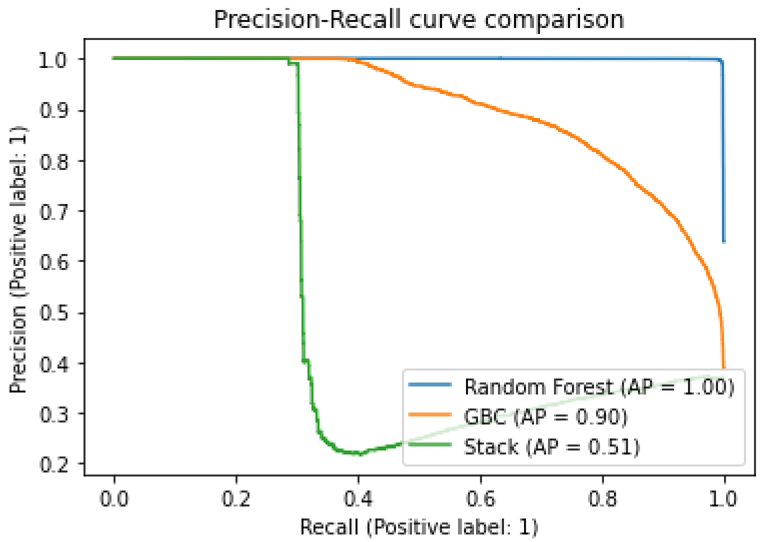
```
plt.plot([0, 1], [0, 1], "k--")
plt.plot(fpr, tpr, label="Random Forest AUC = %0.2f" % auc)
plt.plot(fpr2, tpr2, label="GBC AUC = %0.2f" % auc2)
plt.plot(fpr3, tpr3, label="Stacked AUC = %0.2f" % auc3)
plt.legend()
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("Receiver Operating Characteristic Comparison")
plt.legend(loc="lower right")
plt.show()
```



In [ ]:

```
from sklearn.metrics import precision_recall_curve
from sklearn.metrics import plot_precision_recall_curve
import matplotlib.pyplot as plt

plot_precision_recall_curve(
    final_rf2, X_val3, y_val3, ax=plt.gca(), name="Random Forest"
)

plot_precision_recall_curve(final_gbc, X_val3, y_val3, ax=plt.gca(), name="GBC")

plot_precision_recall_curve(final_stack, X_val3, y_val3, ax=plt.gca(), name="Stack")

plt.title("Precision-Recall curve comparison")
plt.legend(loc="lower right")
```

Out[ ]:
```
<matplotlib.legend.Legend at 0x2a8bd544f40>
```



ANSWER (c)

Model selection and hyperparameter tuning

I tested several different models, starting out with a logistic regression, KNN, neural network and random forest on the original dataset that I had made adjustments to. Neither of these achieved scores above 76% so I decided to use the one hot encoded dataset provided for further testing. I ran another random forest classifier on this data, as well as a gradient boosting classifier and a stacked model. Both the random forest and the gradient boosting classifier achieved scores over 90%. The stacked model was surprisingly the worst performing one that I tried with a score of just above random guessing.

Which model performed best, how did I identify/select it

I evaluated each model's AUC and PR in addition to the time it took to train and validate its performance. I selected the model with the highest AUC and best PR, which was the random forest classifier I trained on the ohe data.

(d) Apply your model "in practice". Make *at least* 5 submissions of different model results to the competition (more submissions are encouraged and you can submit up to 10 per day!). These do not need to be the same that you report on above, but you should select your *most competitive* models.

- Produce submissions by applying your model on the test data.

- Be sure to RETRAIN YOUR MODEL ON ALL LABELED TRAINING AND VALIDATION DATA before making your predictions on the test data for submission. This will help to maximize your performance on the test data.
- In order to get full credit on this problem you must achieve an AUC on the Kaggle public leaderboard above the "Benchmark" score (**0.94933**) on the public leaderboard.

ANSWER (d)

My final submission was for a model with a score of: 0.95912, which was the random forest model trained on the ohe dataset.

2

[25 points] Clustering

Clustering can be used to reveal structure between samples of data and assign group membership to similar groups of samples. This exercise will provide you with experience applying clustering algorithms and comparing these techniques on various datasets to experience the pros and cons of these approaches when the structure of the data being clustered varies. For this exercise, we'll explore clustering in two dimensions to make the results more tangible, but in practice these approaches can be applied to any number of dimensions.

(a) Run K-means and choose the number of clusters. Five datasets are provided for you below and the code to load them below.

- Scatterplot each dataset
- For each dataset run the k-means algorithm for values of k ranging from 1 to 10 and for each plot the "elbow curve" where you plot dissimilarity in each case. Here, you can measure dissimilarity using the within-cluster sum-of-squares, which in sklean is know as "inertia" and can be accessed through the `inertia_` attribute of a fit KMeans class instance.
- For each datasets, where is the elbow in the curve of within-cluster sum-of-squares and why? Is the elbow always clearly visible? When its not clear, you will have to use your judgement in terms of selecting a reasonable number of clusters for the data. *There are also other metrics you can use to explore to measure the quality of cluster fit (but do not have to for this assignment) including the silhouette score, the Calinski-Harabasz index, and the Davies-Bouldin, to name a few within sklearn alone. However, assessing quality of fit without "preferred" cluster assignments to compare against (that is, in a truly unsupervised manner) is challenging because measuring cluster fit quality is typically poorly-defined and doesn't generalize across all types of inter- and intra-cluster variation.*
- Plot your clustered data (different color for each cluster assignment) for your best k-means fit determined from both the elbow curve and your judgement for each dataset and your inspection of the dataset.

(b) Apply DBSCAN. Vary the `eps` and `min_samples` parameters to get as close as you can to having the same number of clusters as your choices with K-means. In this case, the black points are points that were not assigned to clusters.

(c) Apply Spectral Clustering. Select the same number of clusters as selected by k-means.

(d) Comment on the strengths and weaknesses of each approach. In particular, mention:

- Which technique worked "best" and "worst" (as defined by matching how human intuition would cluster the data) on each dataset?
- How much effort was required to get good clustering for each method (how much parameter tuning needed to be done)?

Note: for these clustering plots in this question, do NOT include legends indicating cluster assignment; instead just make sure the cluster assignments are clear from the plot (e.g. different colors for each cluster)

Code is provided below for loading the datasets and for making plots with the clusters as distinct colors

In [ ]:
```
################################
# Load the data
################################
import os
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs, make_moons

# Create / load the datasets:
n_samples = 1500
X0, _ = make_blobs(n_samples=n_samples, centers=2, n_features=2, random_state=0)
X1, _ = make_blobs(n_samples=n_samples, centers=5, n_features=2, random_state=0)

random_state = 170
X, y = make_blobs(n_samples=n_samples, random_state=random_state, cluster_std=1.3)
transformation = [[0.6, -0.6], [-0.2, 0.8]]
X2 = np.dot(X, transformation)
X3, _ = make_blobs(
    n_samples=n_samples, cluster_std=[1.0, 2.5, 0.5], random_state=random_state
)
X4, _ = make_moons(n_samples=n_samples, noise=0.12)

X = [X0, X1, X2, X3, X4]
# The datasets are X[i], where i ranges from 0 to 4
```

In [ ]:
```
################################
```

```
# Code to plot clusters
################################
def plot_cluster(ax, data, cluster_assignments):
    """Plot two-dimensional data clusters

    Parameters
    ----------
    ax : matplotlib axis
        Axis to plot on
    data : list or numpy array of size [N x 2]
        Clustered data
    cluster_assignments : list or numpy array [N]
        Cluster assignments for each point in data

    """
    clusters = np.unique(cluster_assignments)
    n_clusters = len(clusters)
    for ca in clusters:
        kwargs = {}
        if ca == -1:
            # if samples are not assigned to a cluster (have a cluster assignment of -1, color them gray)
            kwargs = {"color": "gray"}
            n_clusters = n_clusters - 1
        ax.scatter(
            data[cluster_assignments == ca, 0],
            data[cluster_assignments == ca, 1],
            s=5,
            alpha=0.5,
            **kwargs,
        )
        ax.set_xlabel("feature 1")
        ax.set_ylabel("feature 2")
        ax.set_title(f"No. Clusters = {n_clusters}")
        ax.axis("equal")
```

ANSWER Question 2

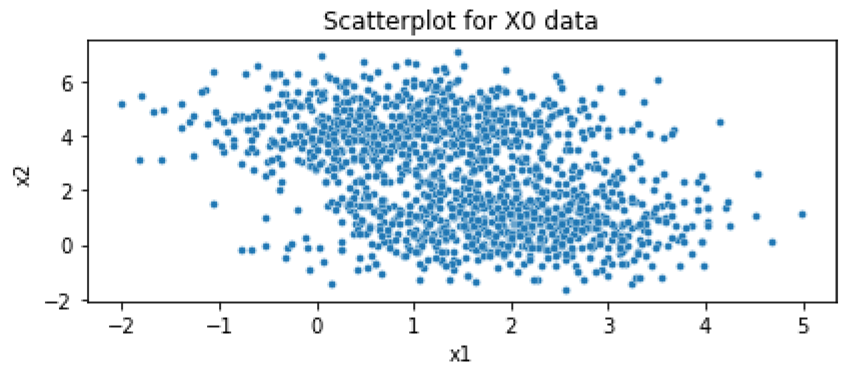
(a) Run K-means and choose the number of clusters.

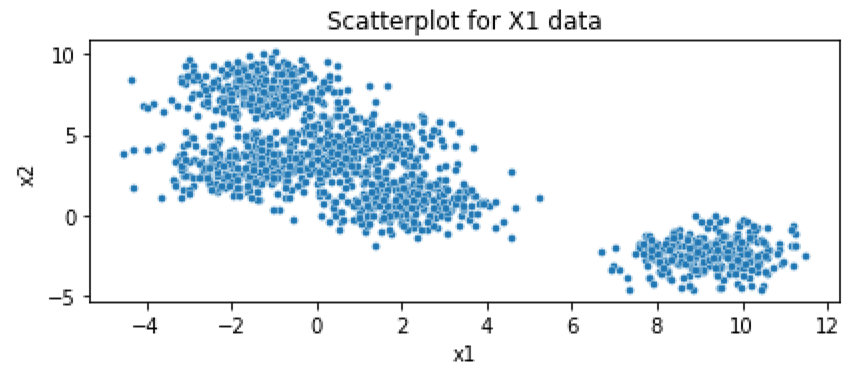
Scatterplot each dataset

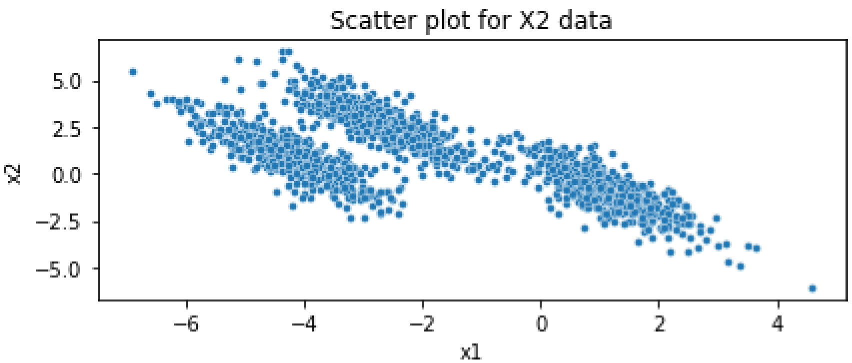
In [ ]:

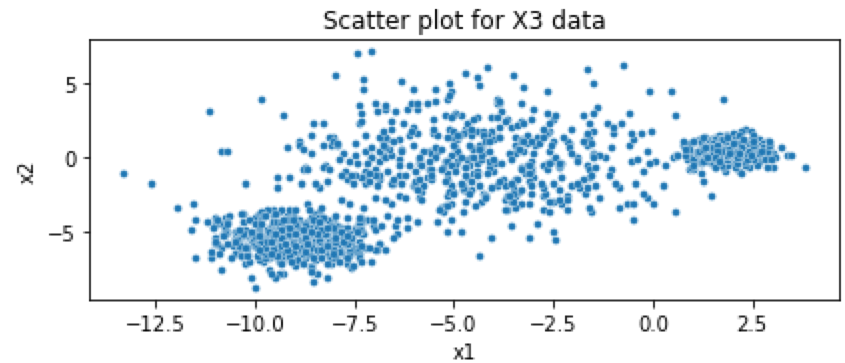
```
X = [X0, X1, X2, X3, X4]
labels = [
    "Scatterplot for X0 data",
```

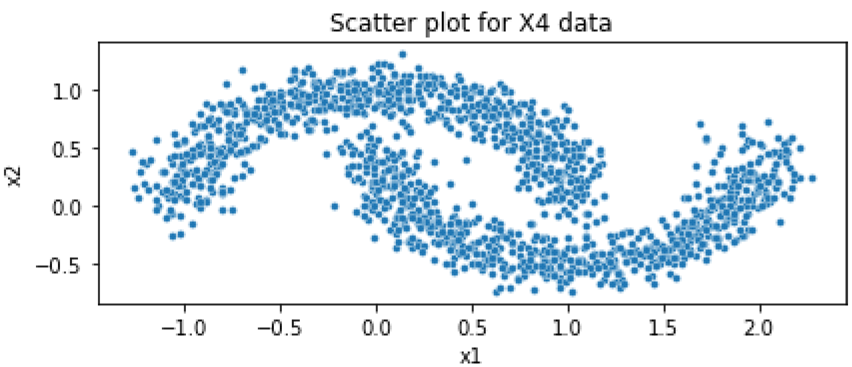
```
    "Scatterplot for X1 data",
    "Scatter plot for X2 data",
    "Scatter plot for X3 data",
    "Scatter plot for X4 data",
]

count = 1
plt.figure(figsize=(12, 8))
for i, lab in zip(X, labels):
    plt.subplot(3, 2, count)
    sns.scatterplot(i[:, 0], i[:, 1], s=16)
    sns.color_palette("pastel")

    count += 1
    plt.title(lab)
    plt.xlabel("x1")
    plt.ylabel("x2")

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









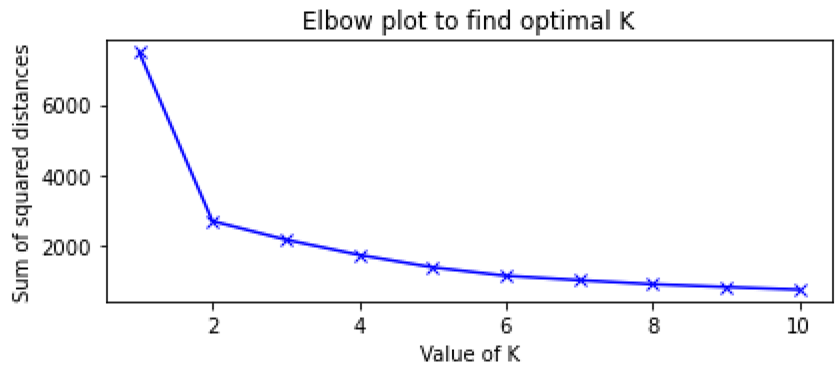


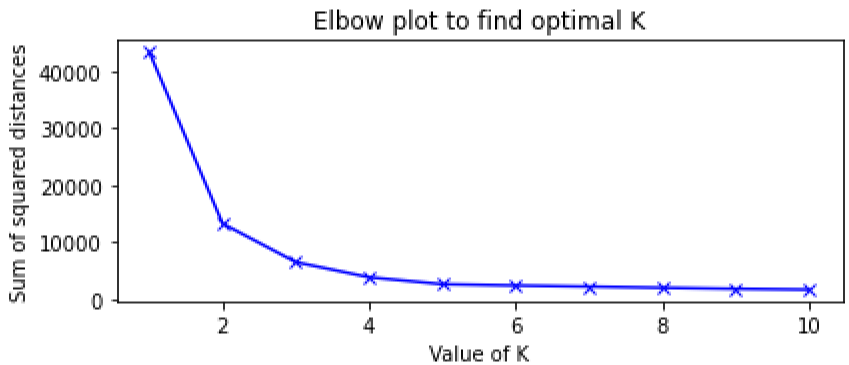
For each dataset run the k-means algorithm for values of k ranging from 1 to 10 and for each plot the "elbow curve" where you plot dissimilarity in each case. Here, you can measure dissimilarity using the within-cluster sum-of-squares, which in sklean is know as "inertia" and can be accessed through the `inertia_` attribute of a fit KMeans class instance.

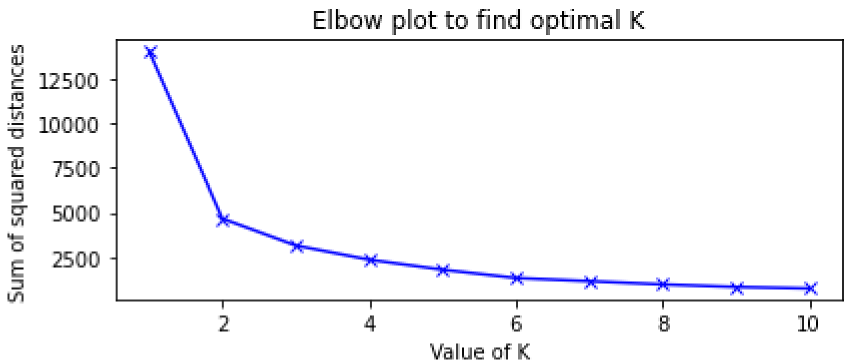
In [ ]:

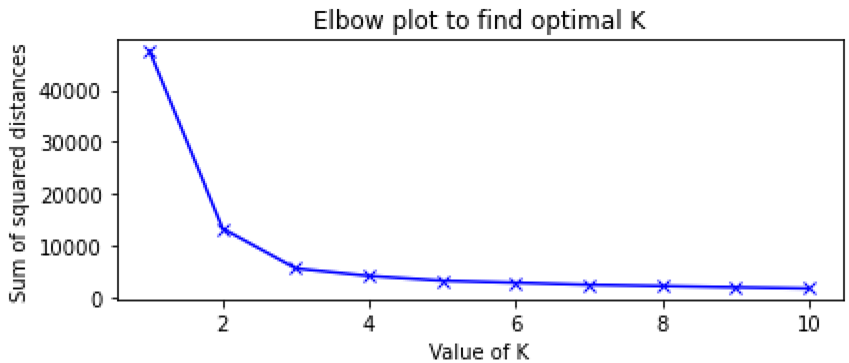
```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import sklearn
```

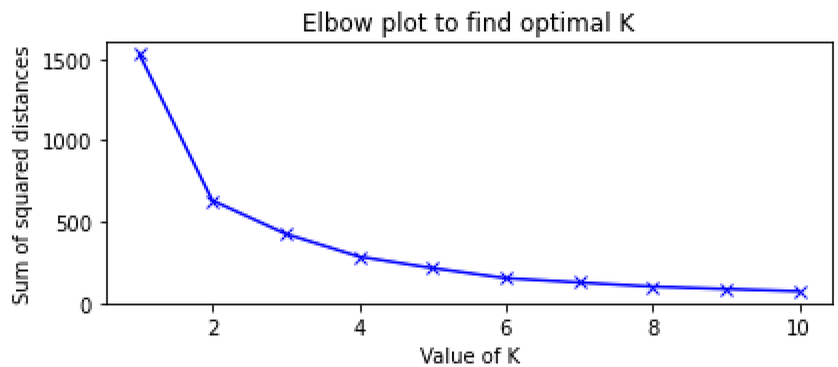
```
from sklearn.cluster import KMeans

Sum2dist = []

K = range(1, 11)
for i in [X0, X1, X2, X3, X4]:
    list = []
    for num_clusters in K:
        kmeans = KMeans(n_clusters=num_clusters)
        kmeans.fit(i)
        list.append(kmeans.inertia_)

    Sum2dist.append(list)
labels = [
    "Elbow plot for X0 data",
    "Elbow plot for X1 data",
    "Elbow plot for X2 data",
    "Elbow plot for X3 data",
    "Elbow plot for X4 data",
]
count = 1
plt.figure(figsize=(12, 8))

for i, lab in zip(Sum2dist, labels):
    plt.subplot(3, 2, count)
    plt.plot(K, i, "bx-")
    plt.xlabel("Value of K")
    plt.ylabel("Sum of squared distances")
    plt.title("Elbow plot to find optimal K")
    plt.tight_layout()
    count += 1
```









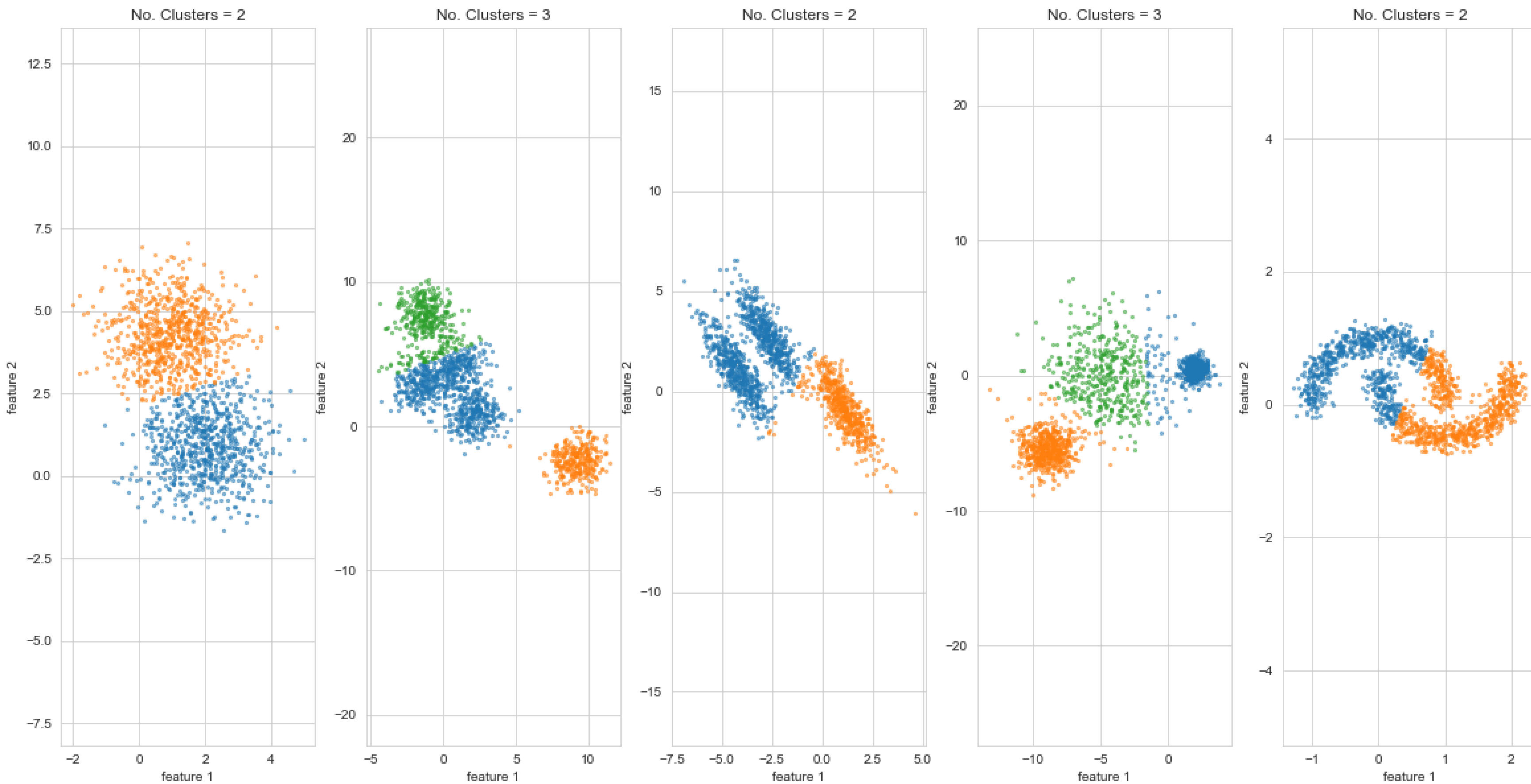


i) For each datasets, where is the elbow in the curve of within-cluster sum-of-squares and why? Is the elbow always clearly visible? When its not clear, you will have to use your judgement in terms of selecting a reasonable number of clusters for the data.

I would consider the elbow to be between 2 and 3 in each of the datasets (X0: 2, X1:, 3, X2: 2, X3: 3, X4: 2) but the elbow is not always clearly visible, so this seems to be more of a rough estimate method to determine the reasonable number of clusters.

Plot your clustered data (different color for each cluster assignment) for your best k-means fit determined from both the elbow curve and your judgement for each dataset and your inspection of the dataset.

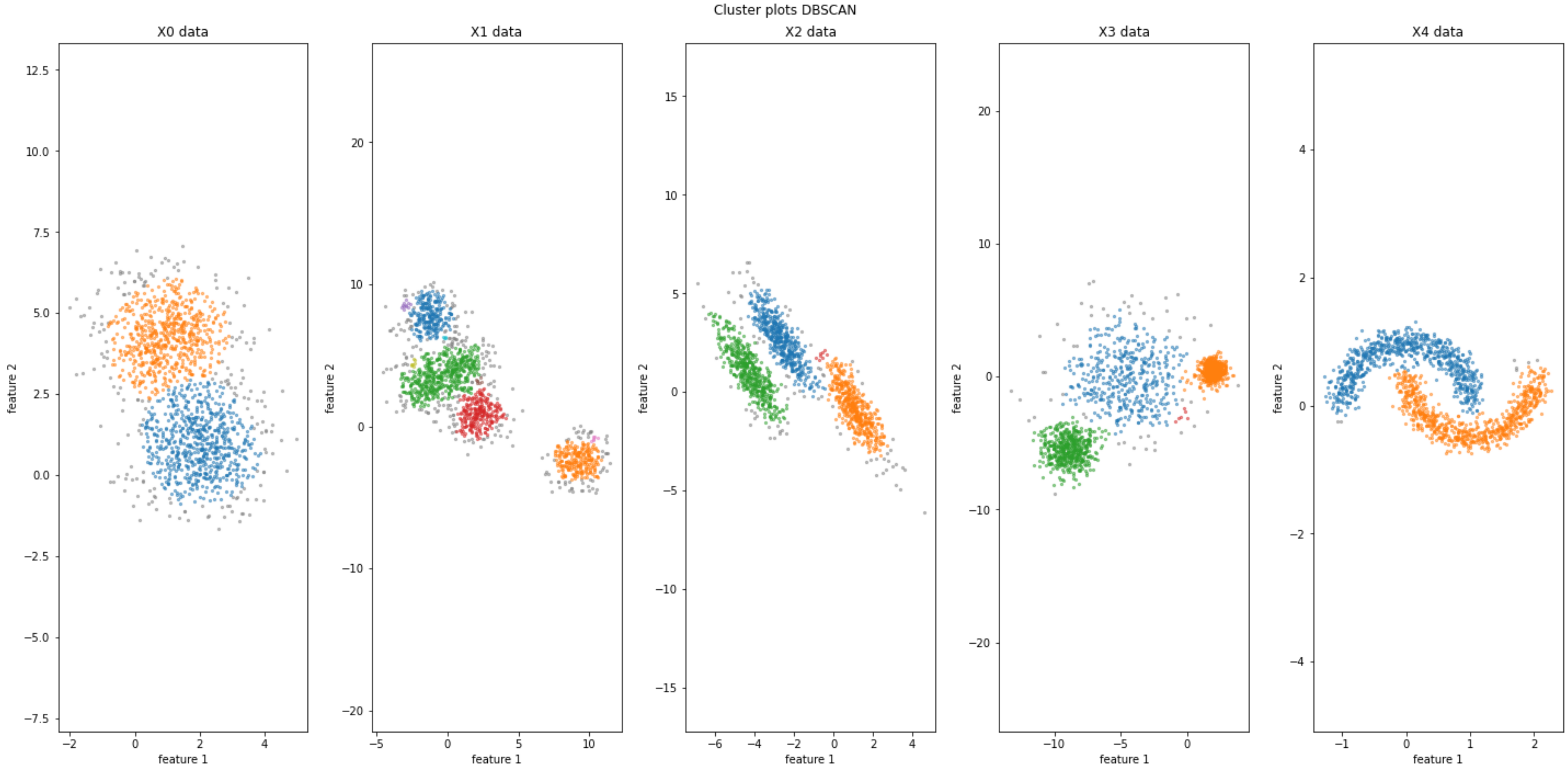
In [ ]:
```
# Plot data using elbow curve numbers (I go with: 2, 3, 2, 3, 2)
K1 = [2, 3, 2, 3, 2]

# Set up plot
fig, axs = plt.subplots(1, 5, figsize=(20, 10))

# Loop over features
for i, a in enumerate(axs):
    ca = KMeans(n_clusters=K1[i]).fit(X[i]).labels_
    plot_cluster(a, X[i], ca)
```



(b) Apply DBSCAN. Vary the `eps` and `min_samples` parameters to get as close as you can to having the same number of clusters as your choices with K-means. In this case, the black points are points that were not assigned to clusters.

In [ ]:
```
from sklearn.cluster import DBSCAN
```

```
# Set up plot
fig, axs = plt.subplots(1, 5, figsize=(20, 10))

# Assign clusters, hyperparameters same as in K-means
cluster0 = DBSCAN(eps=0.75, min_samples=108).fit(X0).labels_
cluster1 = DBSCAN(eps=0.31, min_samples=6).fit(X1).labels_
cluster2 = DBSCAN(eps=0.32, min_samples=6).fit(X2).labels_
cluster3 = DBSCAN(eps=0.70, min_samples=6).fit(X3).labels_
cluster4 = DBSCAN(eps=0.25, min_samples=68).fit(X4).labels_

# Plot clusters
plot_cluster(axs[0], X0, cluster0)
axs[0].set_title("X0 data")
plot_cluster(axs[1], X1, cluster1)
axs[1].set_title("X1 data")
plot_cluster(axs[2], X2, cluster2)
axs[2].set_title("X2 data")
plot_cluster(axs[3], X3, cluster3)
axs[3].set_title("X3 data")
plot_cluster(axs[4], X4, cluster4)
axs[4].set_title("X4 data")
plt.suptitle("Cluster plots DBSCAN")

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

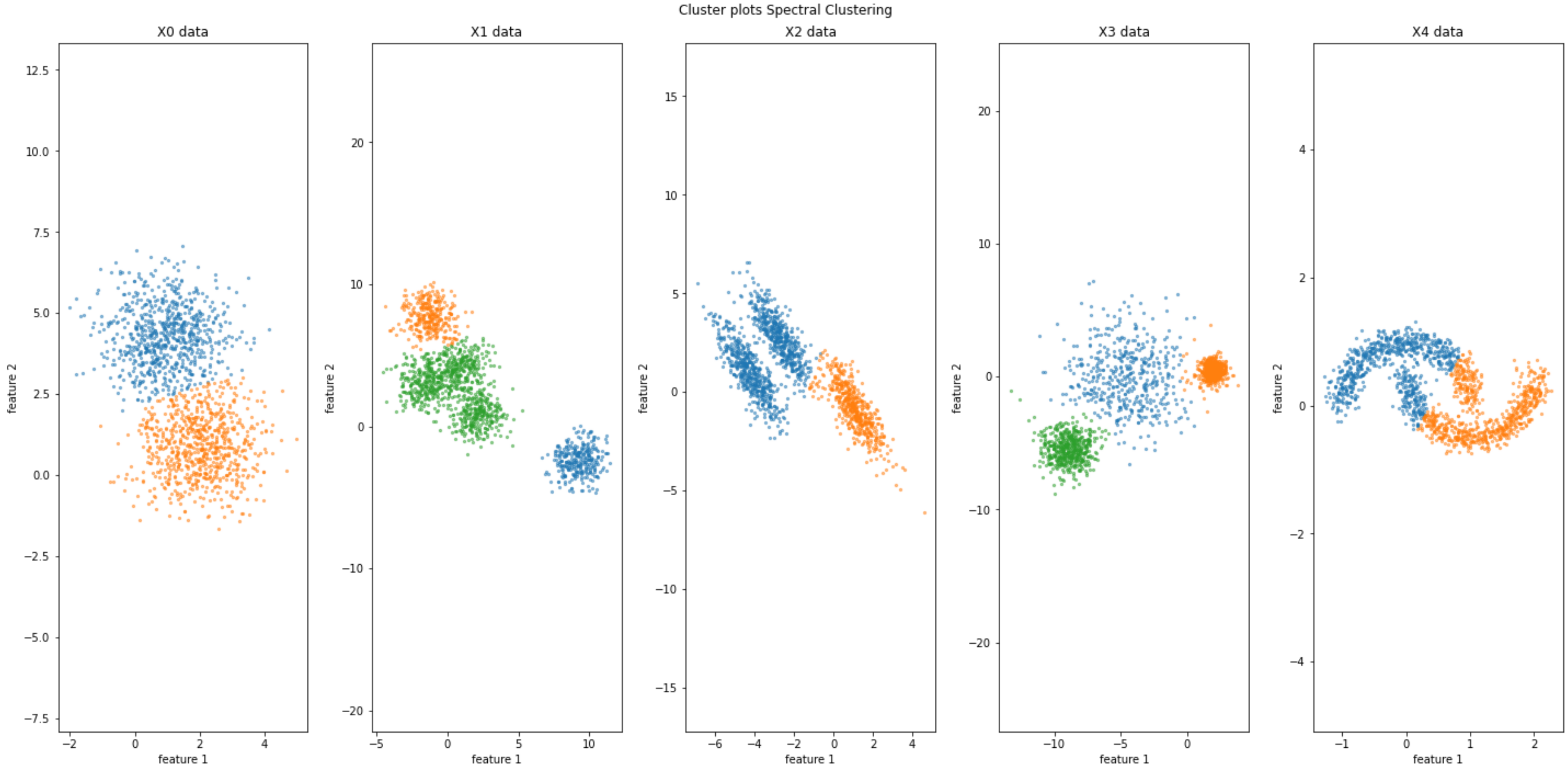


(c) Apply Spectral Clustering. Select the same number of clusters as selected by k-means.

In [ ]:

```
from sklearn.cluster import SpectralClustering

fig, axs = plt.subplots(1, 5, figsize=(20, 10))

# Assign clusters, hyperparameters same as in K-means
cluster0 = (
    SpectralClustering(n_clusters=2, assign_labels="discretize", random_state=2018)
    .fit(X0)
    .labels_
)
cluster1 = (
    SpectralClustering(n_clusters=3, assign_labels="discretize", random_state=2018)
    .fit(X1)
    .labels_
)
```

```
cluster2 = (
    SpectralClustering(n_clusters=2, assign_labels="discretize", random_state=2018)
    .fit(X2)
    .labels_
)
cluster3 = (
    SpectralClustering(n_clusters=3, assign_labels="discretize", random_state=2018)
    .fit(X3)
    .labels_
)
cluster4 = (
    SpectralClustering(n_clusters=2, assign_labels="discretize", random_state=2018)
    .fit(X4)
    .labels_
)

# Plot clusters
plot_cluster(axs[0], X0, cluster0)
axs[0].set_title("X0 data")
plot_cluster(axs[1], X1, cluster1)
axs[1].set_title("X1 data")
plot_cluster(axs[2], X2, cluster2)
axs[2].set_title("X2 data")
plot_cluster(axs[3], X3, cluster3)
axs[3].set_title("X3 data")
plot_cluster(axs[4], X4, cluster4)
axs[4].set_title("X4 data")
plt.suptitle("Cluster plots Spectral Clustering")

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



(d) Comment on the strengths and weaknesses of each approach. In particular, mention:

- Which technique worked "best" and "worst" (as defined by matching how human intuition would cluster the data) on each dataset?
- How much effort was required to get good clustering for each method (how much parameter tuning needed to be done)?

The DBSCAN method worked the best, matching my personal intuition on how I would separate the clusters. It did require quite a bit of trial and error on the parameter tuning side though. The worst performance is seen with the KMeans method after using the elbow plots as indicators. The spectral clustering method required little tuning and performed better than KMeans, though it did not do well on datasets X2 and X4 per my intution.

3

[25 points] Dimensionality reduction and visualization of digits with PCA and t-SNE

(a) Reduce the dimensionality of the data with PCA for data visualization. Load the `scikit-learn` digits dataset (code provided to do this below). Apply PCA and reduce the data (with the associated cluster labels 0-9) into a 2-dimensional space. Plot the data with labels in this two dimensional space (labels can be colors, shapes, or using the actual numbers to represent the data - definitely include a legend in your plot).

(b) Create a plot showing the cumulative fraction of variance explained as you incorporate from 1 through all D principal components of the data (where D is the dimensionality of the data).

- What fraction of variance in the data is UNEXPLAINED by the first two principal components of the data?
- Briefly comment on how this may impact how well-clustered the data are. *You can use the `explained_variance_` attribute of the PCA module in `scikit-learn` to assist with this question*

(c) Reduce the dimensionality of the data with t-SNE for data visualization. T-distributed stochastic neighborhood embedding (t-SNE) is a nonlinear dimensionality reduction technique that is particularly adept at embedding the data into lower 2 or 3 dimensional spaces. Apply t-SNE using the `scikit-learn` implementation to the digits dataset and plot it in 2-dimensions (with associated cluster labels 0-9). You may need to adjust the parameters to get acceptable performance. You can read more about how to use t-SNE effectively here.

(d) Briefy compare/contrast the performance of these two techniques.

- Which seemed to cluster the data best and why?
- Notice that t-SNE doesn't have a `fit` method, but only a `fit_transform` method. Why is this? What implications does this imply for using this method? *Note: Remember that you typically will not have labels available in most problems.*

Code is provided for loading the data below.

In [ ]:

```
################################
# Load the data
################################
from sklearn import datasets
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE

# Load dataset
digits = datasets.load_digits()
```

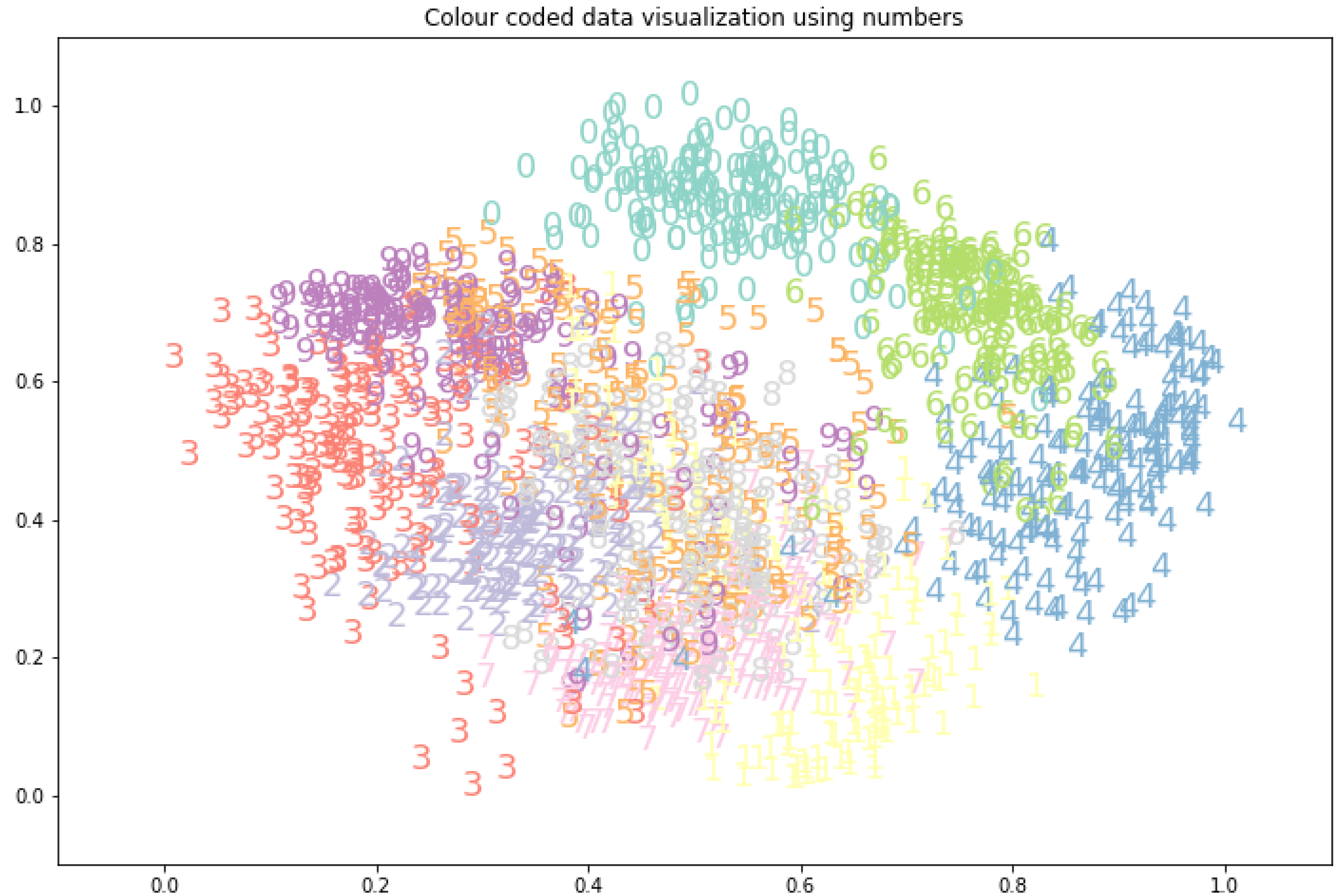
```
n_sample = digits.target.shape[0]
n_feature = digits.images.shape[1] * digits.images.shape[2]
X_digits = np.zeros((n_sample, n_feature))
for i in range(n_sample):
    X_digits[i, :] = digits.images[i, :, :].flatten()
y_digits = digits.target
```

ANSWER Question 3

(a) Reduce the dimensionality of the data with PCA for data visualization. Plot the data with labels in this two dimensional space (labels can be colors, shapes, or using the actual numbers to represent the data - definitely include a legend in your plot).

In [ ]:

```
from sklearn.decomposition import PCA

# apply pca

X_pca = PCA(n_components=2).fit_transform(X_digits)


def plot_components(X, y):  # define the plot components
    x_min, x_max = np.min(X, 0), np.max(X, 0)
    X = (X - x_min) / (x_max - x_min)

    plt.figure(figsize=(12, 8))  # set up the figure
    for i in range(X.shape[0]):
        plt.text(
            X[i, 0],
            X[i, 1],
            str(y[i]),  # visualize digits
            color=plt.cm.Set3(y[i]),  # add color
            fontdict={"size": 18},
        )

    plt.ylim([-0.1, 1.1])
    plt.xlim(-0.1, 1.1)
    plt.title("Colour coded data visualization using numbers")


plot_components(X_pca, y_digits)
```



(b) Create a plot showing the cumulative fraction of variance explained as you incorporate from 1 through all D principal components of the data (where D is the dimensionality of the data).

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

# get cumulative fraction of the explained variance

all_variance = []
for i in range(1, X_digits.shape[1] + 1):
    scaler = StandardScaler()
    pca = PCA(n_components=i)
    rt = pca.fit(scaler.fit(X_digits).transform(X_digits))
    all_variance.append(np.sum(rt.explained_variance_ratio_))
```

In [ ]:

```
# Look at fractions
np.cumsum(rt.explained_variance_ratio_)
```

Out[ ]:

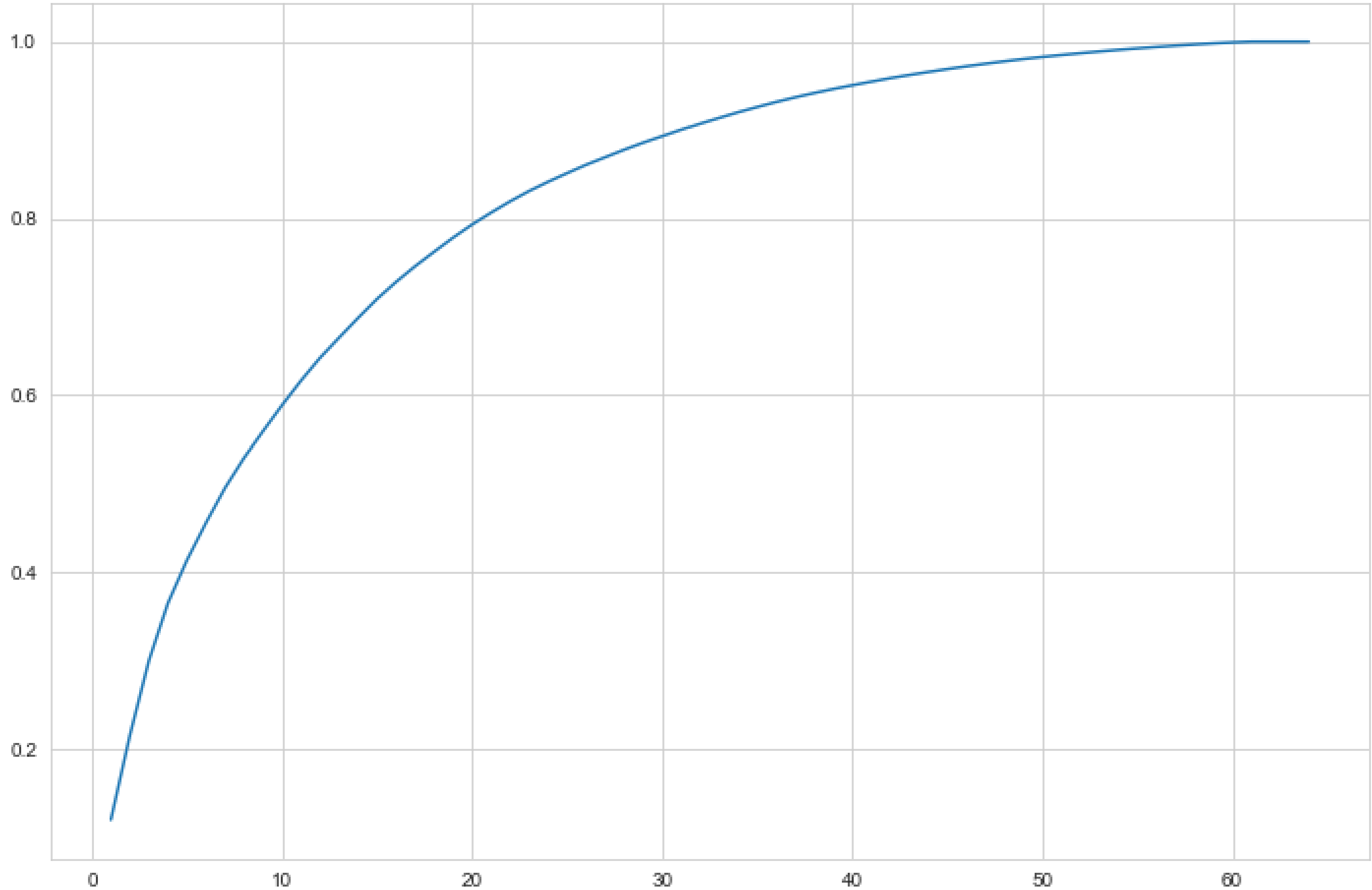
```
array([0.12033916, 0.21594971, 0.30039385, 0.36537793, 0.41397948,
       0.45612068, 0.49554151, 0.52943532, 0.55941753, 0.58873755,
       0.61655561, 0.64232616, 0.66507919, 0.68735099, 0.70900328,
       0.72814495, 0.74590042, 0.76228111, 0.77824572, 0.79313763,
       0.80661732, 0.81933664, 0.83099501, 0.84157148, 0.85132464,
       0.86077023, 0.86940036, 0.87776679, 0.88574372, 0.89320844,
       0.90046426, 0.90738337, 0.91392246, 0.92033038, 0.92624422,
       0.93195585, 0.93719222, 0.94201029, 0.94654748, 0.95077911,
       0.95483964, 0.95881049, 0.96237542, 0.9657833 , 0.96906165,
       0.97217197, 0.97505772, 0.97782262, 0.98041436, 0.98275919,
       0.98494176, 0.98697774, 0.98893286, 0.99076605, 0.99244551,
       0.99405787, 0.9955355 , 0.99688668, 0.99813769, 0.99917465,
       1.        , 1.        , 1.        , 1.        ])
```

In [ ]:

```
# plot the fractions
fig, axs = plt.subplots(1, 1, figsize = (12, 8))
axs.plot(range(1, X_digits.shape[1] + 1), all_variance)
```

Out[ ]:

```
[<matplotlib.lines.Line2D at 0x1868bfdc400>]
```



i) What fraction of variance in the data is UNEXPLAINED by the first two principal components of the data?

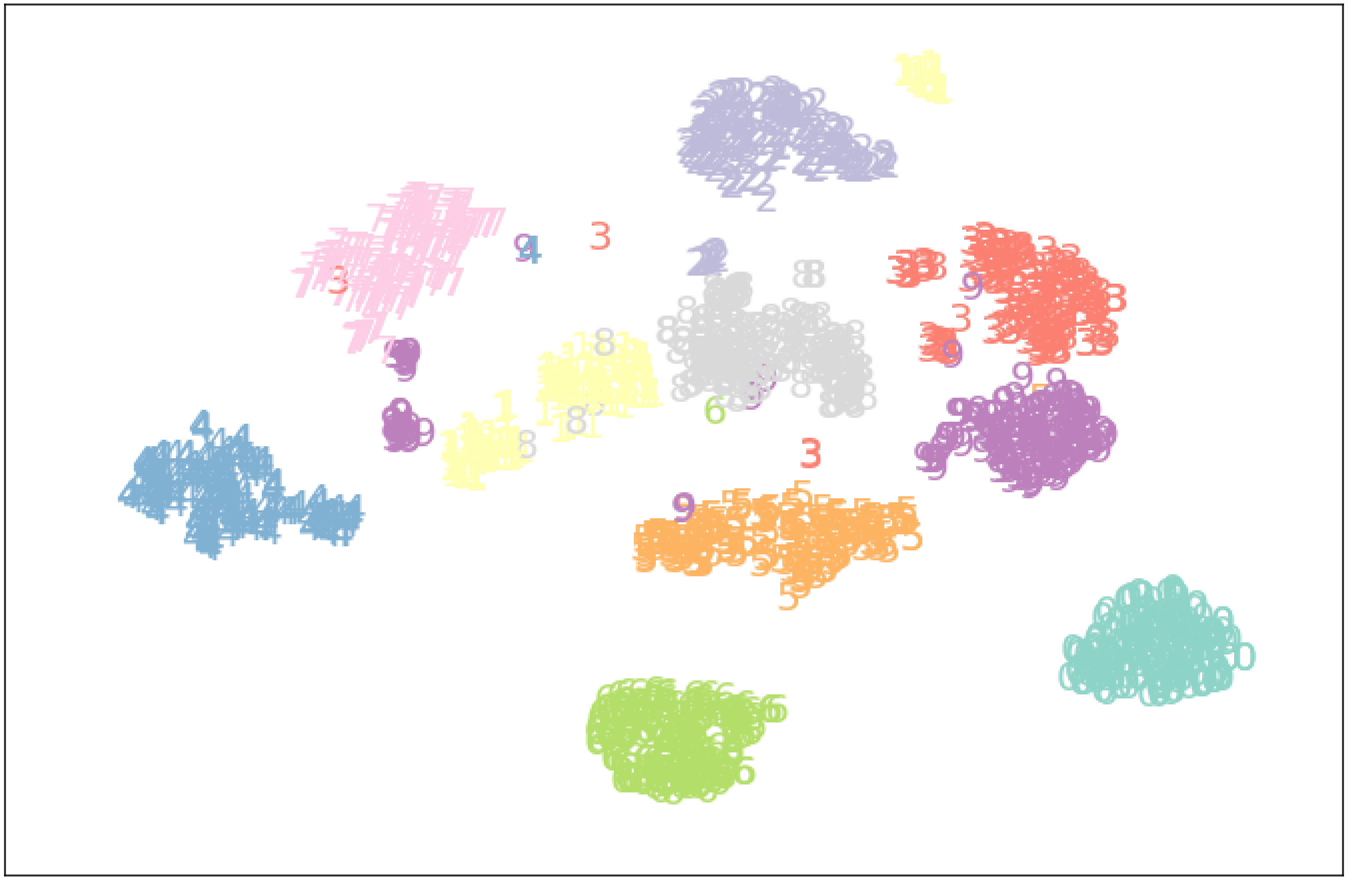
The first two principal components leave approximately 88% and 78% of the variance unexplained and only get to 100% explained variance in the last iteration.

ii) Briefly comment on how this may impact how well-clustered the data are.

Initially the data is not well clustered but becomes better clustered over time.

(c) Reduce the dimensionality of the data with t-SNE for data visualization. T-distributed stochastic neighborhood embedding (t-SNE) is a nonlinear dimensionality reduction technique that is particularly adept at embedding the data into lower 2 or 3 dimensional spaces. Apply t-SNE using the `scikit-learn` implementation to the digits dataset and plot it in 2-dimensions (with associated cluster labels 0-9). You may need to adjust the parameters to get acceptable performance. You can read more about how to use t-SNE effectively here.

In [ ]:
```
from sklearn.manifold import TSNE

# apply TSNE
X_embedded = TSNE(n_components=2, learning_rate="auto", init="random").fit_transform(
    X_digits
)


def plot_components(X, y):  # define plot components
    x_min, x_max = np.min(X, 0), np.max(X, 0)
    X = (X - x_min) / (x_max - x_min)

    plt.figure(figsize=(12, 8))  # set up plot
    for i in range(X.shape[0]):
        plt.text(
            X[i, 0], X[i, 1], str(y[i]), color=plt.cm.Set3(y[i]), fontdict={"size": 20}
        )

    plt.xticks([]),
    plt.yticks([]),
    plt.ylim([-0.1, 1.1])
    plt.xlim(-0.1, 1.1)


plot_components(X_embedded, y_digits)
```



(d) Briefy compare/contrast the performance of these two techniques. *i)* Which seemed to cluster the data best and why?

The data seemed to be clustered best by the TSNE method, because it separates the clusters out further from each other and only mis-attributes a very small number of digits.

ii) Notice that t-SNE doesn't have a `fit` method, but only a `fit_transform` method. Why is this? What implications does this imply for using this method?

TSNE is an unsupervised method, so there is no such thing as fitting on one dataset and then validating on a different dataset. Unsupervised methods can perform well, like in this case but cannot be used directly to train and test.