Assignment 5 - Kaggle Competition and Unsupervised Learning

Luopeiwen Yi

Netid: ly178

Names of students you worked with on this assignment: Jeremy Tan

Note: this assignment falls under collaboration Mode 2: Individual Assignment – Collaboration Permitted. Please refer to the syllabus for additional information.

Instructions for all assignments can be found here.

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

In [ ]:
```
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import numpy as np
import time

from sklearn.model_selection import PredefinedSplit
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.impute import SimpleImputer
from sklearn.model_selection import train_test_split
from scipy.stats import randint as sp_randint
```

```python
from sklearn.model_selection import RandomizedSearchCV
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.svm import SVC
from xgboost import XGBClassifier
from catboost import CatBoostClassifier
from sklearn.ensemble import AdaBoostClassifier
from scipy.stats import uniform, randint
from sklearn.metrics import (
    roc_curve,
    roc_auc_score,
    precision_recall_curve,
    auc as auc_score,
)
from sklearn.metrics import average_precision_score

# from lightgbm import LGBMClassifier

from sklearn.cluster import KMeans
from sklearn.cluster import DBSCAN
from sklearn.cluster import SpectralClustering
from sklearn.decomposition import PCA
from sklearn.metrics import pairwise_distances

import warnings

warnings.simplefilter(action="ignore", category=FutureWarning)

# Filter out the specific UserWarning related to loky's process executor
warnings.filterwarnings(
    "ignore",
    message="A worker stopped while some jobs were given to the executor.*",
    category=UserWarning,
)
warnings.filterwarnings(
    "ignore",
    message="X has feature names, but .* was fitted without feature names",
    category=UserWarning,
)

from sklearn.exceptions import ConvergenceWarning
```

```
# Ignore ConvergenceWarning
warnings.filterwarnings("ignore", category=ConvergenceWarning)
```

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 at the Kaggle website (see the Ed post).

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 (see Ed Discussions for the link). 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. You should plot all of the model's ROC curves on a single plot and the PR curves on a single plot. One of the 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 5 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.

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

```
################################
# Load the data
################################
data = pd.read_pickle("a5_q1.pkl.zip", compression="zip")
# data = pd.read_pickle("a5_q1.pkl")

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"]

################################
# Produce submission
################################


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")
```

ANSWER

(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.

> - My goal is to predict hotel cancellations in rooms booked from two hotels based on characteristics of the booking and those booking the room. See the assignment for more details.
> - The performance metric is area under the ROC curve (AUC).

In [ ]:
```
# Print dataset shapes
print(f"Training set shape: {X_train_original.shape}")
print(f"Test set shape: {X_test_original.shape}")
print(f"One-hot-encoded training set shape: {X_train_ohe.shape}")
print(f"One-hot-encoded test set shape: {X_test_ohe.shape}\n")
```

```
Training set shape: (95512, 29)
Test set shape: (23878, 29)
One-hot-encoded training set shape: (95512, 940)
One-hot-encoded test set shape: (23878, 940)
```

In [ ]:
```
# Display types of features
print("Type of Features:")
print(f"Original training dataset:\n{X_train_original.dtypes.value_counts()}\n")
print(f"One-hot-encoded training dataset:\n{X_train_ohe.dtypes.value_counts()}\n")
```

```
Type of Features:
Original training dataset:
int64      15
object     10
float64     4
Name: count, dtype: int64

One-hot-encoded training dataset:
uint8      923
int64       15
float64      2
Name: count, dtype: int64
```

In [ ]:
```
# Summary statistics
print("Summary Statistics for Original Training Dataset:")
X_train_original.describe().transpose()
```

```
Summary Statistics for Original Training Dataset:
```

Out[ ]:

	count	mean	std	min	25%	50%	75%	max
lead_time	95512.0	103.849768	106.722804	0.00	18.00	69.0	160.0	709.0
arrival_date_year	95512.0	2016.157205	0.707470	2015.00	2016.00	2016.0	2017.0	2017.0
arrival_date_week_number	95512.0	27.152902	13.611204	1.00	16.00	27.0	38.0	53.0
arrival_date_day_of_month	95512.0	15.823038	8.786777	1.00	8.00	16.0	23.0	31.0
stays_in_weekend_nights	95512.0	0.928491	0.999940	0.00	0.00	1.0	2.0	19.0
stays_in_week_nights	95512.0	2.503288	1.918017	0.00	1.00	2.0	3.0	50.0
adults	95512.0	1.855746	0.596925	0.00	2.00	2.0	2.0	55.0
children	95510.0	0.103696	0.397763	0.00	0.00	0.0	0.0	3.0
babies	95512.0	0.007748	0.093348	0.00	0.00	0.0	0.0	9.0
is_repeated_guest	95512.0	0.031598	0.174929	0.00	0.00	0.0	0.0	1.0
previous_cancellations	95512.0	0.087235	0.844491	0.00	0.00	0.0	0.0	26.0
previous_bookings_not_canceled	95512.0	0.140035	1.532968	0.00	0.00	0.0	0.0	72.0
booking_changes	95512.0	0.220621	0.653900	0.00	0.00	0.0	0.0	21.0
agent	82431.0	86.893778	110.839209	1.00	9.00	14.0	229.0	535.0
company	5453.0	188.237117	131.459182	6.00	62.00	178.0	269.0	543.0
days_in_waiting_list	95512.0	2.316348	17.651287	0.00	0.00	0.0	0.0	391.0
adr	95512.0	101.679313	50.906371	-6.38	69.29	94.5	126.0	5400.0
required_car_parking_spaces	95512.0	0.062673	0.246274	0.00	0.00	0.0	0.0	8.0
total_of_special_requests	95512.0	0.571310	0.792712	0.00	0.00	0.0	1.0	5.0

In [ ]:
```
print("Summary Statistics for One-Hot-Encoded Training Dataset:")
X_train_ohe.describe().transpose()
```

```
Summary Statistics for One-Hot-Encoded Training Dataset:
```

Out[ ]:

	count	mean	std	min	25%	50%	75%	max
lead_time	95512.0	103.849768	106.722804	0.0	18.0	69.0	160.0	709.0
arrival_date_year	95512.0	2016.157205	0.707470	2015.0	2016.0	2016.0	2017.0	2017.0
arrival_date_week_number	95512.0	27.152902	13.611204	1.0	16.0	27.0	38.0	53.0
arrival_date_day_of_month	95512.0	15.823038	8.786777	1.0	8.0	16.0	23.0	31.0
stays_in_weekend_nights	95512.0	0.928491	0.999940	0.0	0.0	1.0	2.0	19.0
...	...	...	...	...	...	...	...	...
company_543.0	95512.0	0.000021	0.004576	0.0	0.0	0.0	0.0	1.0
customer_type_Contract	95512.0	0.034017	0.181273	0.0	0.0	0.0	0.0	1.0
customer_type_Group	95512.0	0.004806	0.069157	0.0	0.0	0.0	0.0	1.0
customer_type_Transient	95512.0	0.750419	0.432773	0.0	1.0	1.0	1.0	1.0
customer_type_Transient-Party	95512.0	0.210759	0.407850	0.0	0.0	0.0	0.0	1.0

940 rows × 8 columns

In [ ]:
```
print("Summary Statistics for Original Testing Dataset:")
X_test_original.describe().transpose()
```

```
Summary Statistics for Original Testing Dataset:
```

Out[ ]:

	count	mean	std	min	25%	50%	75%	max
lead_time	23878.0	104.658012	107.422248	0.0	18.0	70.0	161.0	737.0
arrival_date_year	23878.0	2016.153949	0.707509	2015.0	2016.0	2016.0	2017.0	2017.0
arrival_date_week_number	23878.0	27.214256	13.581023	1.0	16.0	28.0	38.0	53.0
arrival_date_day_of_month	23878.0	15.699054	8.756480	1.0	8.0	16.0	23.0	31.0
stays_in_weekend_nights	23878.0	0.924030	0.993302	0.0	0.0	1.0	2.0	18.0
stays_in_week_nights	23878.0	2.488357	1.868844	0.0	1.0	2.0	3.0	42.0
adults	23878.0	1.859033	0.502434	0.0	2.0	2.0	2.0	20.0
children	23876.0	0.104666	0.401747	0.0	0.0	0.0	0.0	10.0
babies	23878.0	0.008753	0.112309	0.0	0.0	0.0	0.0	10.0
is_repeated_guest	23878.0	0.033169	0.179080	0.0	0.0	0.0	0.0	1.0
previous_cancellations	23878.0	0.086649	0.843734	0.0	0.0	0.0	0.0	26.0
previous_bookings_not_canceled	23878.0	0.125346	1.345930	0.0	0.0	0.0	0.0	66.0
booking_changes	23878.0	0.223134	0.645899	0.0	0.0	0.0	0.0	14.0
agent	20619.0	85.892235	110.514717	1.0	9.0	14.0	229.0	531.0
company	1344.0	193.444196	132.413576	9.0	67.0	193.0	274.0	539.0
days_in_waiting_list	23878.0	2.340355	17.366967	0.0	0.0	0.0	0.0	391.0
adr	23878.0	102.438355	49.021805	0.0	69.0	95.0	126.0	508.0
required_car_parking_spaces	23878.0	0.061898	0.241323	0.0	0.0	0.0	0.0	2.0
total_of_special_requests	23878.0	0.571572	0.793163	0.0	0.0	0.0	1.0	5.0

In [ ]:
```
print("Summary Statistics for One-Hot-Encoded Testing Dataset:")
X_test_ohe.describe().transpose()
```

```
Summary Statistics for One-Hot-Encoded Testing Dataset:
```

Out[ ]:

	count	mean	std	min	25%	50%	75%	max
lead_time	23878.0	104.658012	107.422248	0.0	18.0	70.0	161.0	737.0
arrival_date_year	23878.0	2016.153949	0.707509	2015.0	2016.0	2016.0	2017.0	2017.0
arrival_date_week_number	23878.0	27.214256	13.581023	1.0	16.0	28.0	38.0	53.0
arrival_date_day_of_month	23878.0	15.699054	8.756480	1.0	8.0	16.0	23.0	31.0
stays_in_weekend_nights	23878.0	0.924030	0.993302	0.0	0.0	1.0	2.0	18.0
...	...	...	...	...	...	...	...	...
company_543.0	23878.0	0.000000	0.000000	0.0	0.0	0.0	0.0	0.0
customer_type_Contract	23878.0	0.034634	0.182856	0.0	0.0	0.0	0.0	1.0
customer_type_Group	23878.0	0.004942	0.070125	0.0	0.0	0.0	0.0	1.0
customer_type_Transient	23878.0	0.751277	0.432282	0.0	1.0	1.0	1.0	1.0
customer_type_Transient-Party	23878.0	0.209146	0.406708	0.0	0.0	0.0	0.0	1.0

940 rows × 8 columns

In [ ]:

```
# Missing values in training dataset
missing_values_orig = X_train_original.isnull().sum()
missing_values_ohe = X_train_ohe.isnull().sum()

print("Missing Values in Original Training Dataset:")
print(
    missing_values_orig[missing_values_orig > 0], "\n"
)  # Only show columns with missing values

print("Missing Values in One-Hot-Encoded Training Dataset:")
print(
    missing_values_ohe[missing_values_ohe > 0], "\n"
)  # Only show columns with missing values
```

```
Missing Values in Original Training Dataset:
children        2
country       395
agent       13081
company     90059
dtype: int64

Missing Values in One-Hot-Encoded Training Dataset:
children    2
dtype: int64
```

In [ ]:
```python
# Checking for missing values in the original test dataset
missing_values_orig_test = X_test_original.isnull().sum()

# Checking for missing values in the one-hot-encoded test dataset
missing_values_ohe_test = X_test_ohe.isnull().sum()

print("Missing Values in Original Test Dataset:")
print(
    missing_values_orig_test[missing_values_orig_test > 0], "\n"
)  # Only show columns with missing values

print("Missing Values in One-Hot-Encoded Test Dataset:")
print(
    missing_values_ohe_test[missing_values_ohe_test > 0], "\n"
)  # Only show columns with missing values
```

```
Missing Values in Original Test Dataset:
children        2
country        93
agent        3259
company     22534
dtype: int64

Missing Values in One-Hot-Encoded Test Dataset:
children    2
dtype: int64
```

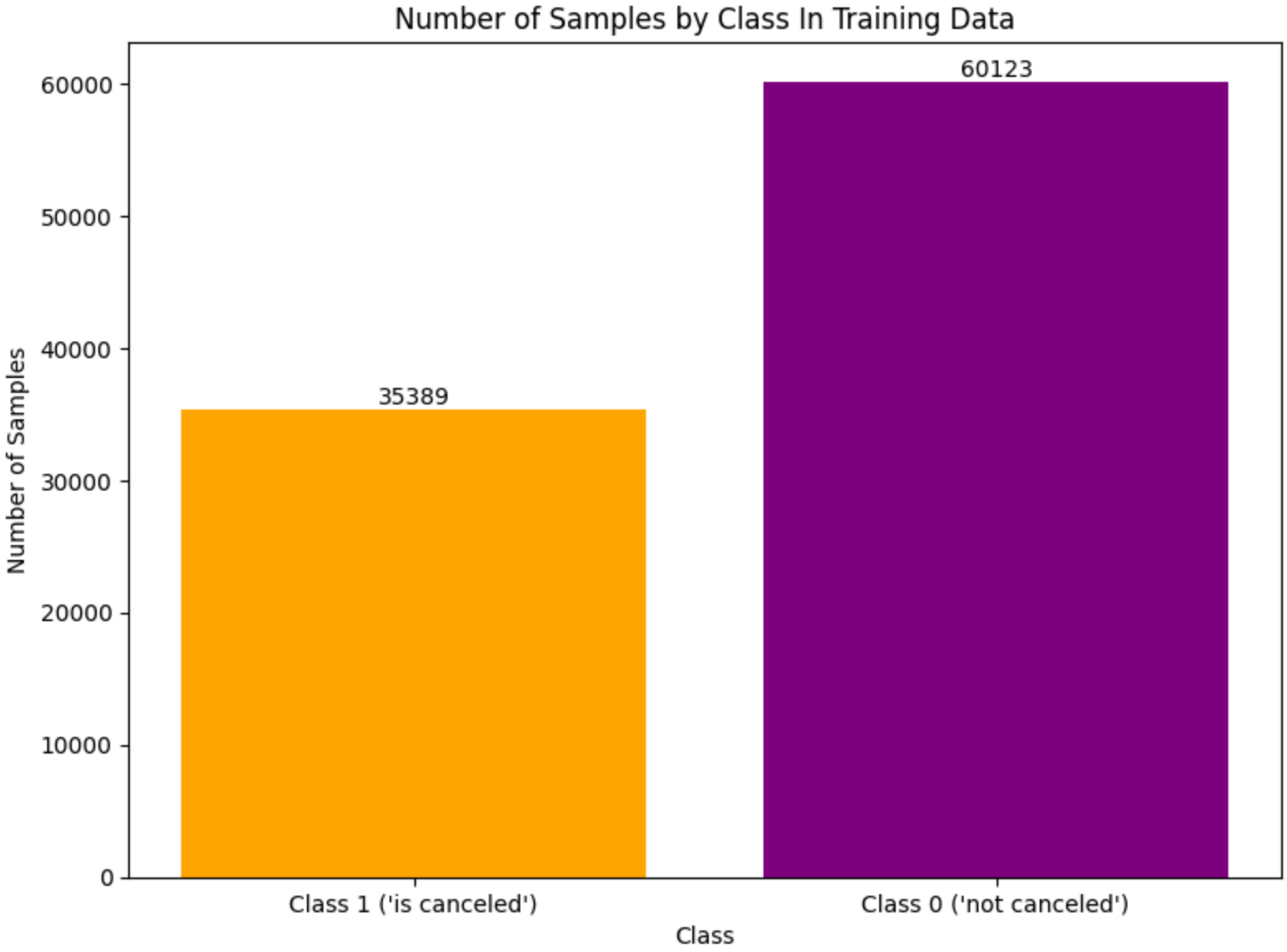
In [ ]:
```python
# Display the total number of labels in y_train
print(f"Total number of labels in training data: {y_train.shape[0]}")

# Display the count of each unique label value in y_train
print("\nDistribution of labels in training data:")
print(y_train.value_counts())
```

```
Total number of labels in training data: 95512

Distribution of labels in training data:
is_canceled
0    60123
1    35389
Name: count, dtype: int64
```

In [ ]:
```python
# For a binary classification where '1' indicates 'is canceled' and '0' indicates 'not canceled'
class_1 = y_train[y_train == 1]
class_0 = y_train[y_train == 0]
print(
    "Number of examples in class 1 ('is canceled') in training data:", class_1.shape[0]
)
print(
    "Number of examples in class 0 ('not canceled') in training data:", class_0.shape[0]
)
```

```
Number of examples in class 1 ('is canceled') in training data: 35389
Number of examples in class 0 ('not canceled') in training data: 60123
```

In [ ]:
```python
# Bar plot of samples by class

# define class and samples list
classes = ["Class 1 ('is canceled')", "Class 0 ('not canceled')"]
samples = [class_1.shape[0], class_0.shape[0]]

# Plotting
plt.figure(figsize=(8, 6))
bars = plt.bar(classes, samples, color=["orange", "purple"])

# Adding the count above the bar
for bar in bars:
    yval = bar.get_height()
    plt.text(
        bar.get_x() + bar.get_width() / 2, yval + 0.5, yval, ha="center", va="bottom"
    )

plt.xlabel("Class")
plt.ylabel("Number of Samples")
plt.title("Number of Samples by Class In Training Data")
plt.tight_layout()
plt.show()
```



In [ ]:
```
fraction_positive = class_1.shape[0] / (class_1.shape[0] + class_0.shape[0])
print(
    f"Fraction of samples that are positive being in Class 1 ('is canceled') is around: {fraction_positive:.2%}"
)
```

```
Fraction of samples that are positive being in Class 1 ('is canceled') is around: 37.05%
```

(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.

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. **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.

In [ ]:
```
random_state = 0
```

> **Separating Out a Validation Dataset**
>
> - I created my own validation dataset for comparing model performance; without this, there's a significant likelihood of overfitting to the data.
> - I chose to split 80% training, 20% validation. I also created train plus val for final performance evaluation.

In [ ]:
```
# Splitting the dataset while keeping stratification in mind, if applicable
X_train_1, X_val_1, y_train_1, y_val_1 = train_test_split(
    X_train_ohe, y_train, test_size=0.2, random_state=random_state
)

# For RandomSeachCV, we will need to combine training and validation sets then
#  specify which portion is training and which is validation
# Also, for the final performance evaluation, train on all of the training AND validation data
X_train_plus_val = np.concatenate((X_train_1, X_val_1), axis=0)
y_train_plus_val = np.concatenate((y_train_1, y_val_1), axis=0)

# Create a predefined train/test split for RandomSearchCV (to be used later)
# validation_fold = np.concatenate((-1 * np.ones(len(y_train_1)), np.zeros(len(y_val_1))))
# train_val_split = PredefinedSplit(validation_fold)
```

> **Handling Missing Values**
>
> - For the 'children' feature, there are 2 missing values.
> - I impute these missing values with the median, which is robust to outliers (less sensitive to outliers than the mean).
> - I use SimpleImputer(strategy="median") to impute missing values in the sklearn pipelines.

Normalizing Data

- According the description of the training data, numerical features's mean is not close to 0 and standard deviation is not close to 1. They have a different range of values and variability, indicating significant scale differences and need for normalization.
- I use StandardScaler() to normalize data in the sklearn pipelines.

Handling Imbalanced Data

- According to the description of the training data, it is an imbalanced dataset, where the number of samples in one class significantly outnumbers the samples in another class. Imbalanced dataset can lead to biased model that performs well on the majority class but performs poorly on the minority class. There's a risk of overfitting to the majority class while failing to generalize well to unseen data, especially for the minority class.
- I use the class weight techniques in some models to balance the data in the sklearn pipeline.

Initialize Model Pipelines

- Logistic regression
- K-nearest neighbors
- Random Forests
- Catboost
- XGBoost

In [ ]:
```
# generate initial pipelines for models

# logistic regression pipeline
logistic_regression_pipeline = Pipeline(
    [
        ("imputer", SimpleImputer(strategy="median")),  # Impute missing values
        ("scaler", StandardScaler()),
        (
            "classifier",
            LogisticRegression(class_weight="balanced", random_state=random_state),
        ),  # handle imbalanced data
    ]
)

# knn pipeline
```

```python
knn_pipeline = Pipeline(
    [
        ("imputer", SimpleImputer(strategy="median")),  # Impute missing values
        ("scaler", StandardScaler()),
        ("classifier", KNeighborsClassifier()),
    ]
)

# random forest pipeline
random_forest_pipeline = Pipeline(
    [
        ("imputer", SimpleImputer(strategy="median")),  # Impute missing values
        ("scaler", StandardScaler()),
        (
            "classifier",
            RandomForestClassifier(class_weight="balanced", random_state=random_state),
        ),  # handle imbalanced data
    ]
)

# catboost pipeline
catboost_pipeline = Pipeline(
    [
        ("imputer", SimpleImputer(strategy="median")),  # Impute missing values
        ("scaler", StandardScaler()),
        # handle imbalanced data
        (
            "classifier",
            CatBoostClassifier(
                auto_class_weights="Balanced", verbose=0, random_state=random_state
            ),
        ),
    ]
)

# xgboost pipeline
xgboost_pipeline = Pipeline(
    [
        ("imputer", SimpleImputer(strategy="median")),  # Impute missing values
        ("scaler", StandardScaler()),
        # handle imbalanced data
        (
            "classifier",
            XGBClassifier(
                scale_pos_weight=(len(y_train) - sum(y_train)) / sum(y_train),
                random_state=random_state,
            ),
        ),
```

```
    ]
)
```

(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.

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).

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

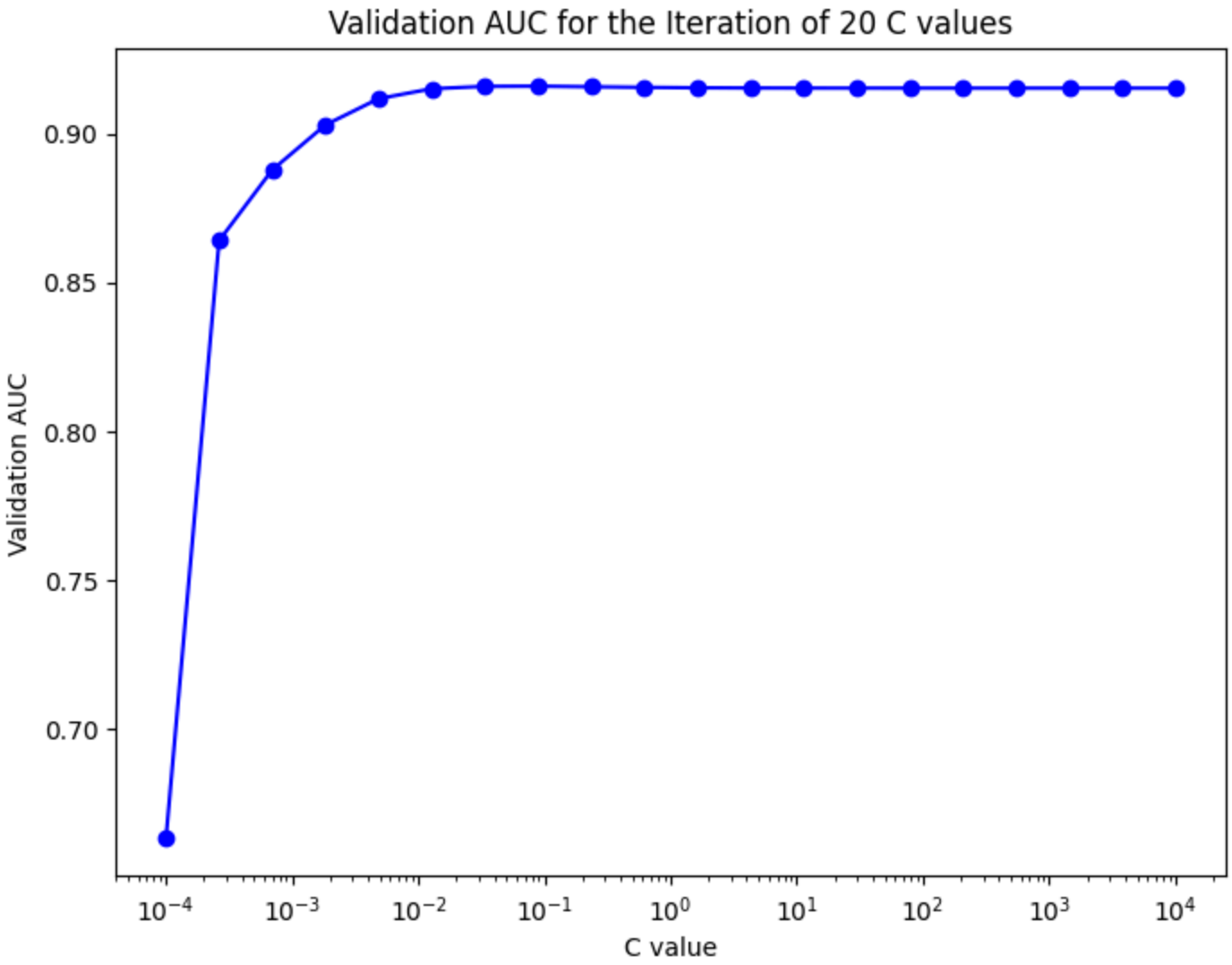
- Plot the ROC curves and PR curves for your models in two plots: one of ROC curves and one of PR curves. For each of these two plots, compare the performance of the models you selected above and trained on the training data, evaluating them on the validation data. Be sure to plot the line representing random guessing on each plot. You should plot all of the model's ROC curves on a single plot and the PR curves on a single plot. One of the 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

> Logistic Regression

In [ ]:
```
# Generate 20 logarithmically spaced values for C between 10^-4 and 10^4
C_values = np.logspace(-4, 4, num=20)

# Evaluate L1 regularization strengths for reducing features in final model
# As C (1/λ) decreases, more coefficients go to zero

# create list to store the auc
auc_list_lr = []

for c in C_values:
    # logistic regression with l1 penalty
    clf = Pipeline(
        [
            ("imputer", SimpleImputer(strategy="median")),
            ("scaler", StandardScaler()),
            (
                "classifier",
                LogisticRegression(
                    class_weight="balanced",  # balance class weight
                    penalty="l1",  # lasso
                    C=c,
                    solver="saga",  # large dataset
                    max_iter=1000,  # increase max iter to help with convergence
                    random_state=random_state,
                ),
            ),  # handle imbalanced data
        ]
    )
    # fit the model with training data
    clf.fit(X_train_1, y_train_1)
    # get probability estimates for all classes
    clf_score = clf.predict_proba(X_val_1)[:, 1]

    # Metrics calculation
    auc = roc_auc_score(y_val_1, clf_score)

    # Store metrics
    auc_list_lr.append(auc)
```

In [ ]:
```
# Plotting the AUC for different values of C

# set figure size
plt.figure(figsize=(8, 6))
```

```
# Plotting the AUC
plt.plot(C_values, auc_list_lr, marker="o", color="blue")
plt.xlabel("C value")
plt.ylabel("Validation AUC")
plt.title("Validation AUC for the Iteration of 20 C values")
plt.xscale("log")  # Set the x-axis to a logarithmic scale

# Set x-ticks
tick_values = [10**i for i in range(-4, 5)]  # Generates 10^-4, 10^-3, ..., 10^4
plt.xticks(tick_values, labels=[f"$10^{{{i}}}$" for i in range(-4, 5)])

plt.show()
```



In [ ]:
```
# Find the index of the maximum AUC score in the auc_list_lr
max_auc_index_lr = auc_list_lr.index(max(auc_list_lr))

# Use this index to find the corresponding best C value
best_c = C_values[max_auc_index_lr]

# Print the best AUC score and its corresponding C value
print(f"Best AUC Score: {round(max(auc_list_lr),4)}")
print(f"Corresponding C Value: {best_c}")
```

```
Best AUC Score: 0.9159
Corresponding C Value: 0.08858667904100823
```

In [ ]:
```
# Save the best C value to a file
with open("best_c_lr.pkl", "wb") as file:
    pickle.dump(best_c, file)
```

In [ ]:
```
with open("best_c_lr.pkl", "rb") as file:
    best_c = pickle.load(file)
```

In [ ]:
```
# Define the logistic regression pipeline with the best C value
best_lr_pipeline = Pipeline(
    [
        ("imputer", SimpleImputer(strategy="median")),
        ("scaler", StandardScaler()),
        (
            "classifier",
            LogisticRegression(
                class_weight="balanced",
                penalty="l1",
                C=best_c,  # Use the best C value here
                solver="saga",
                max_iter=1000,
                random_state=random_state,
            ),
        ),
    ]
)

# Fit the pipeline on the training data
start = time.time()
best_lr_pipeline.fit(X_train_1, y_train_1)
end = time.time()
time_train_lr = end - start
print(f"Time to train Logistic Regression: {round(time_train_lr, 4)}")

# Predict the probabilities on the validation data
```

```
start = time.time()
y_val_pred_proba_lr = best_lr_pipeline.predict_proba(X_val_1)[:, 1]
end = time.time()
time_val_lr = end - start
print(
    f"Time to predict on validation data for Logistic Regression: {round(time_val_lr, 4)}"
)

# Calculate the ROC AUC score for the logistic regression model on the validation data
roc_auc_lr_val = roc_auc_score(y_val_1, y_val_pred_proba_lr)
print(
    f"ROC AUC score for Logistic Regression on validation data: {round(roc_auc_lr_val, 4)}"
)
```

```
Time to train Logistic Regression: 999.5894
Time to predict on validation data for Logistic Regression: 0.3389
ROC AUC score for Logistic Regression on validation data: 0.9159
```

In [ ]:
```
# Compute ROC curve for Logistic Regression
fpr_lr, tpr_lr, _ = roc_curve(y_val_1, y_val_pred_proba_lr)
# Compute AUC score for ROC curve for Logistic Regression
roc_auc_lr = auc_score(fpr_lr, tpr_lr)

# Compute PR curve for Logistic Regression
precision_lr, recall_lr, _ = precision_recall_curve(y_val_1, y_val_pred_proba_lr)
# Compute the AP score for Logistic Regression
average_precision_lr = average_precision_score(y_val_1, y_val_pred_proba_lr)
```

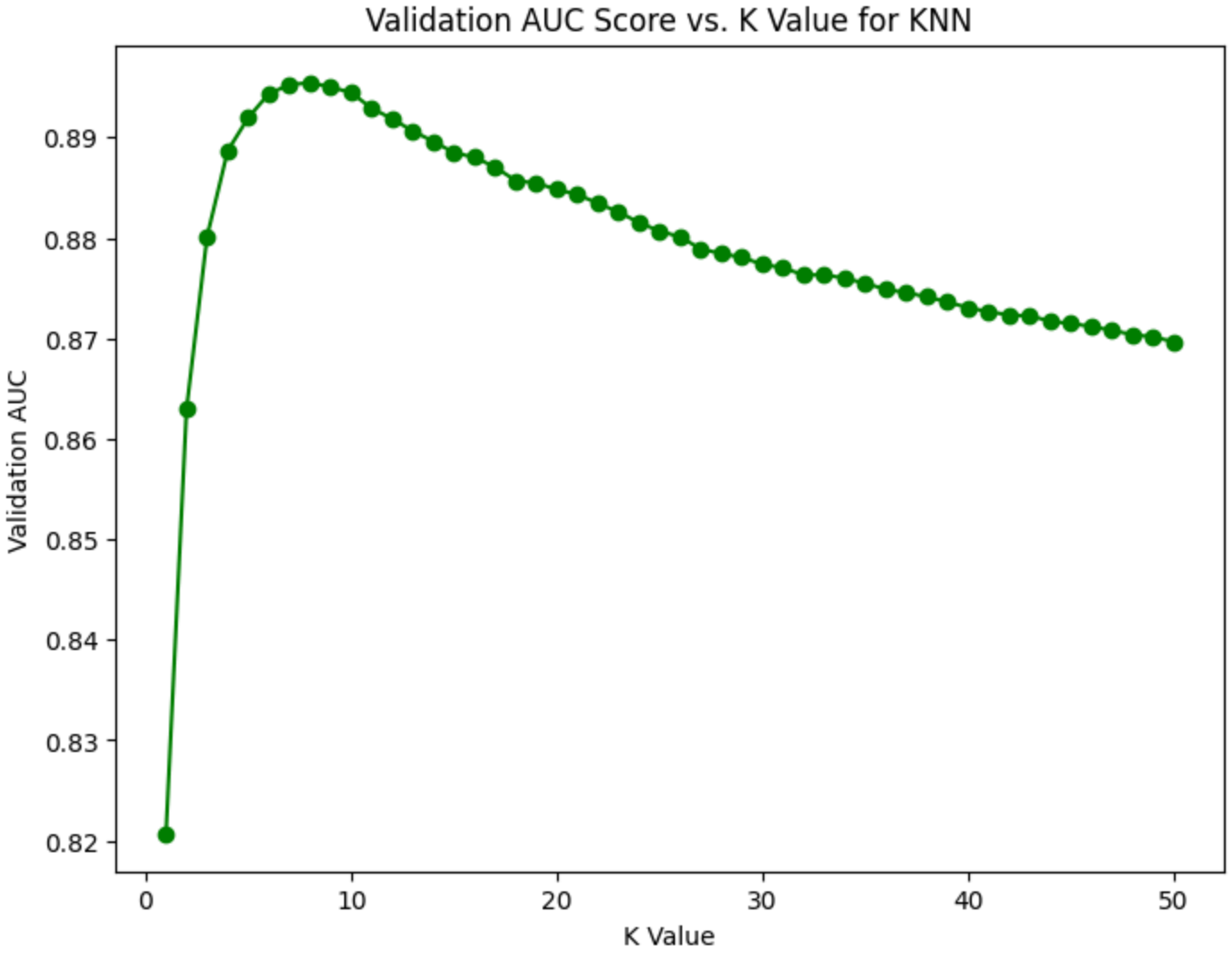
> KNN

In [ ]:
```
# Initialize list to store the AUC scores
auc_list_knn = []

# Evaluate K from 1 to 50
for k in range(1, 51):
    # Define the KNN pipeline with k neighbors
    knn_pipeline = Pipeline(
        [
            ("imputer", SimpleImputer(strategy="median")),
            ("scaler", StandardScaler()),
            ("classifier", KNeighborsClassifier(n_neighbors=k)),
        ]
    )

    # Fit the model with training data
    knn_pipeline.fit(X_train_1, y_train_1)
```

```
    # Predict probabilities on the validation data
    knn_score = knn_pipeline.predict_proba(X_val_1)[:, 1]

    # Calculate AUC score and store it
    auc = roc_auc_score(y_val_1, knn_score)
    auc_list_knn.append(auc)
```

In [ ]:

```
# Set figure size
plt.figure(figsize=(8, 6))

k_values = range(1, 51)  # K values from 1 to 50

# Plotting the AUC
plt.plot(k_values, auc_list_knn, marker="o", color="green")
plt.xlabel("K Value")
plt.ylabel("Validation AUC")
plt.title("Validation AUC Score vs. K Value for KNN")

plt.show()
```



In [ ]:
```
# find the index of the highest AUC
max_auc_knn = max(auc_list_knn)
max_auc_index_knn = np.argmax(auc_list_knn)
best_k = max_auc_index_knn + 1
print(
    f"The optimal value of k is {best_k} "
    f"because it maximizes the auc score, which is around {round(max_auc_knn,4)}, on validation dataset."
)
```

```
The optimal value of k is 8 because it maximizes the auc score, which is around 0.8954, on validation dataset.
```

In [ ]:
```
# Save the best k value to a file
with open("best_k_knn.pkl", "wb") as file:
    pickle.dump(best_k, file)
```

In [ ]:
```
# Open the best k value as a file
with open("best_k_knn.pkl", "rb") as file:
    best_k = pickle.load(file)
```

In [ ]:
```
# Define the KNN pipeline with the best parameters (n_neighbors=8)
best_knn_pipeline = Pipeline(
    [
        ("imputer", SimpleImputer(strategy="median")),
        ("scaler", StandardScaler()),
        ("classifier", KNeighborsClassifier(n_neighbors=best_k)),
    ]
)

# Fit the pipeline on the training data
start = time.time()
best_knn_pipeline.fit(X_train_1, y_train_1)
end = time.time()
time_train_knn = end - start
print(f"Time to train KNN: {round(time_train_knn, 4)}")

# Predict the probabilities on the validation data
start = time.time()
y_val_pred_proba_knn = best_knn_pipeline.predict_proba(X_val_1)[:, 1]
end = time.time()
time_val_knn = end - start
print(f"Time to predict on validation data for KNN: {round(time_val_knn, 4)}")

# Calculate the ROC AUC score for the KNN model on the validation data
roc_auc_knn_val = roc_auc_score(y_val_1, y_val_pred_proba_knn)
print(f"ROC AUC score for KNN on validation data: {round(roc_auc_knn_val, 4)}")
```

```
Time to train KNN: 8.1044
Time to predict on validation data for KNN: 17.1606
ROC AUC score for KNN on validation data: 0.8954
```

In [ ]:
```
# Compute ROC curve for KNN
fpr_knn, tpr_knn, _ = roc_curve(y_val_1, y_val_pred_proba_knn)
# Compute AUC score for ROC curve for KNN
roc_auc_knn = auc_score(fpr_knn, tpr_knn)

# Compute PR curve for KNN
precision_knn, recall_knn, _ = precision_recall_curve(y_val_1, y_val_pred_proba_knn)
# Compute the AP score for KNN
average_precision_knn = average_precision_score(y_val_1, y_val_pred_proba_knn)
```

> Random Forest

In [ ]:

```
# Define the Random Forest pipeline
random_forest_pipeline = Pipeline(
    [
        ("imputer", SimpleImputer(strategy="median")),
        ("scaler", StandardScaler()),
        (
            "classifier",
            RandomForestClassifier(class_weight="balanced", random_state=random_state),
        ),
    ]
)

# Define the parameter distributions to sample from
param_distributions = {
    "classifier__n_estimators": sp_randint(
        100, 500
    ),  # Randomly choose between 100 and 500 trees
    "classifier__max_depth": [None]
    + list(range(5, 51)),  # No limit or a depth between 5 and 50
    "classifier__min_samples_split": sp_randint(
        2, 11
    ),  # Minimum number of samples required to split, between 2 and 10
    "classifier__min_samples_leaf": sp_randint(
        1, 11
    ),  # Minimum number of samples required at a leaf node, between 1 and 10
}

# Setup RandomizedSearchCV
random_search_rf = RandomizedSearchCV(
    estimator=random_forest_pipeline,
    param_distributions=param_distributions,
    n_iter=50,
    cv=5,
    scoring="roc_auc",
    verbose=1,
    random_state=random_state,
    n_jobs=-1,
)

# Fit RandomizedSearchCV to the training data
random_search_rf.fit(X_train_1, y_train_1)

# Print the best parameters and the best score
print(f"Best parameters for Random Forest: {random_search_rf.best_params_}")
print(
```

```
    f"Best training ROC AUC for Random Forest: {round(random_search_rf.best_score_,4)}"
)
```

```
Fitting 5 folds for each of 50 candidates, totalling 250 fits
Best parameters for Random Forest: {'classifier__max_depth': 43, 'classifier__min_samples_leaf': 1, 'classifier__min_sa
mples_split': 3, 'classifier__n_estimators': 365}
Best training ROC AUC for Random Forest: 0.9543
```

In [ ]:
```
# Save the best parameters to a file
with open("best_params_rf.pkl", "wb") as file:
    pickle.dump(random_search_rf.best_params_, file)
```

In [ ]:
```
with open("best_params_rf.pkl", "rb") as file:
    best_params_rf = pickle.load(file)
```

In [ ]:
```
# Remove the 'classifier__' prefix from the parameter names
best_params_rf_cleaned = {
    k.replace("classifier__", ""): v for k, v in best_params_rf.items()
}

# Define the Random Forest pipeline with the best parameters
best_random_forest_pipeline = Pipeline(
    [
        ("imputer", SimpleImputer(strategy="median")),
        ("scaler", StandardScaler()),
        (
            "classifier",
            RandomForestClassifier(
                **best_params_rf_cleaned,  # Use the unpacked, cleaned parameters
                class_weight="balanced",
                random_state=random_state,
            ),
        ),
    ]
)

start = time.time()
# Fit the pipeline on the training data
best_random_forest_pipeline.fit(X_train_1, y_train_1)
end = time.time()

time_train_rf = end - start
print(f"Time to train Random Forest: {round(time_train_rf, 4)}")

start = time.time()
# Predict the probabilities on the validation data
y_val_pred_proba_rf = best_random_forest_pipeline.predict_proba(X_val_1)[:, 1]
```

```
end = time.time()

time_val_rf = end - start
print(f"Time to predict on validation data for Random Forest: {round(time_val_rf, 4)}")

# Calculate the ROC AUC score
roc_auc_rf_val = roc_auc_score(y_val_1, y_val_pred_proba_rf)

print(f"ROC AUC score for Random Forest on validation data: {round(roc_auc_rf_val, 4)}")
```

```
Time to train Random Forest: 143.8109
Time to predict on validation data for Random Forest: 2.371
ROC AUC score for Random Forest on validation data: 0.9606
```

In [ ]:
```
# Compute ROC curve for Random Forest
fpr_rf, tpr_rf, _ = roc_curve(y_val_1, y_val_pred_proba_rf)
# Compute AUC score for ROC curve for Random Forest
roc_auc_rf = auc_score(fpr_rf, tpr_rf)

# Compute PR curve for Random Forest
precision_rf, recall_rf, _ = precision_recall_curve(y_val_1, y_val_pred_proba_rf)
# Compute the AP score for Random Forest
average_precision_rf = average_precision_score(y_val_1, y_val_pred_proba_rf)
```

> Catboost

In [ ]:
```
# Define the CatBoost pipeline
catboost_pipeline = Pipeline(
    [
        ("imputer", SimpleImputer(strategy="median")),
        ("scaler", StandardScaler()),
        (
            "classifier",
            CatBoostClassifier(
                auto_class_weights="Balanced", verbose=0, random_state=random_state
            ),
        ),
    ]
)

# Define the parameter distributions to sample from
param_distributions = {
    "classifier__learning_rate": uniform(0.01, 0.3),  # Boosting learning rate
    "classifier__depth": [4, 6, 8, 10],  # Depth of the tree
    "classifier__iterations": [100, 200, 300, 400, 500],  # Number of trees
}
```

```
# Setup RandomizedSearchCV
random_search_catboost = RandomizedSearchCV(
    estimator=catboost_pipeline,
    param_distributions=param_distributions,
    n_iter=50,
    cv=5,
    scoring="roc_auc",
    verbose=1,
    random_state=random_state,
    n_jobs=-1,
)


# Fit RandomizedSearchCV to the training data
random_search_catboost.fit(X_train_1, y_train_1)

# Print the best parameters and the best score
print(f"Best parameters for CatBoost: {random_search_catboost.best_params_}")
print(
    f"Best training ROC AUC for CatBoost: {round(random_search_catboost.best_score_,4)}"
)
```

```
Fitting 5 folds for each of 50 candidates, totalling 250 fits
Best parameters for CatBoost: {'classifier__depth': 10, 'classifier__iterations': 500, 'classifier__learning_rate': 0.2
1454608973104503}
Best training ROC AUC for CatBoost: 0.9559
```

In [ ]:
```
best_params_catboost = random_search_catboost.best_params_

# Save the best parameters to a file
with open("best_params_catboost.pkl", "wb") as file:
    pickle.dump(best_params_catboost, file)
```

In [ ]:
```
# open the best parameters as a file
with open("best_params_catboost.pkl", "rb") as file:
    best_params_catboost = pickle.load(file)
```

In [ ]:
```
# Remove the 'classifier__' prefix from the parameter names
best_params_catboost_cleaned = {
    k.replace("classifier__", ""): v for k, v in best_params_catboost.items()
}

# Define the CatBoost pipeline with the dynamically applied best parameters
best_catboost_pipeline = Pipeline(
    [
        ("imputer", SimpleImputer(strategy="median")),
```

```
        ("scaler", StandardScaler()),
        (
            "classifier",
            CatBoostClassifier(
                **best_params_catboost_cleaned,  # Use the unpacked, cleaned parameters
                auto_class_weights="Balanced",
                verbose=0,
                random_state=random_state,  # Ensure random_state is defined
            ),
        ),
    ]
)

# Fit the pipeline on the training data
start = time.time()
best_catboost_pipeline.fit(X_train_1, y_train_1)
end = time.time()

time_train_catboost = end - start
print(f"Time to train CatBoost: {round(time_train_catboost, 4)} seconds")

# Predict the probabilities on the validation data
start = time.time()
y_val_pred_proba_cb = best_catboost_pipeline.predict_proba(X_val_1)[:, 1]
end = time.time()

time_val_catboost = end - start
print(
    f"Time to predict on validation data for CatBoost: {round(time_val_catboost, 4)} seconds"
)

# Calculate the ROC AUC score
roc_auc_catboost_val = roc_auc_score(y_val_1, y_val_pred_proba_cb)

print(
    f"ROC AUC score for CatBoost on validation data: {round(roc_auc_catboost_val, 4)}"
)
```

```
Time to train CatBoost: 22.3583 seconds
Time to predict on validation data for CatBoost: 0.3624 seconds
ROC AUC score for CatBoost on validation data: 0.9617
```

In [ ]:

```
# Compute ROC curve for CatBoost
fpr_cb, tpr_cb, _ = roc_curve(y_val_1, y_val_pred_proba_cb)
# Compute AUC score for ROC curve for CatBoost
roc_auc_cb = auc_score(fpr_cb, tpr_cb)

# Compute PR curve for CatBoost
```

```
precision_cb, recall_cb, _ = precision_recall_curve(y_val_1, y_val_pred_proba_cb)
# Compute the AP score for CatBoost
average_precision_cb = average_precision_score(y_val_1, y_val_pred_proba_cb)
```

> XGBoost

In [ ]:

```
# Define the XGBoost pipeline
xgboost_pipeline = Pipeline(
    [
        ("imputer", SimpleImputer(strategy="median")),
        ("scaler", StandardScaler()),
        (
            "classifier",
            XGBClassifier(
                scale_pos_weight=(len(y_train_1) - sum(y_train_1)) / sum(y_train_1),
                random_state=random_state,
            ),
        ),
    ]
)

param_distributions = {
    "classifier__n_estimators": randint(100, 1000),  # Number of trees
    "classifier__learning_rate": uniform(
        0.01, 0.3
    ),  # Step size shrinkage used to prevent overfitting
    "classifier__max_depth": randint(3, 10),  # Maximum tree depth
    "classifier__subsample": uniform(
        0.6, 0.4
    ),  # Subsample ratio of the training instances
    "classifier__colsample_bytree": uniform(
        0.6, 0.4
    ),  # Subsample ratio of columns when constructing each tree
}

# Setup RandomizedSearchCV
random_search_xgb = RandomizedSearchCV(
    estimator=xgboost_pipeline,
    param_distributions=param_distributions,
    n_iter=50,
    cv=5,
    scoring="roc_auc",
    verbose=1,
    random_state=random_state,
    n_jobs=-1,
)
```

```
# Fit RandomizedSearchCV to the training data
random_search_xgb.fit(X_train_1, y_train_1)

# Print the best parameters and the best score
print(f"Best parameters for XGBoost: {random_search_xgb.best_params_}")
print(f"Best training ROC AUC for XGBoost: {round(random_search_xgb.best_score_,4)}")
```

```
Fitting 5 folds for each of 50 candidates, totalling 250 fits
Best parameters for XGBoost: {'classifier__colsample_bytree': 0.9936168965456585, 'classifier__learning_rate': 0.088102
2782762214, 'classifier__max_depth': 8, 'classifier__n_estimators': 827, 'classifier__subsample': 0.779170468748292}
Best training ROC AUC for XGBoost: 0.9579
```

In [ ]:
```
best_params_xgb = random_search_xgb.best_params_

# Save the best parameters to a file
with open("best_params_xgb.pkl", "wb") as file:
    pickle.dump(best_params_xgb, file)
```

In [ ]:
```
# Open the best parameters to a file
with open("best_params_xgb.pkl", "rb") as file:
    best_params_xgb = pickle.load(file)
```

In [ ]:
```
# Remove the 'classifier__' prefix from the parameter names
best_params_xgb_cleaned = {
    k.replace("classifier__", ""): v for k, v in best_params_xgb.items()
}

# Define the XGBoost pipeline with the dynamically applied best parameters
best_xgboost_pipeline = Pipeline(
    [
        ("imputer", SimpleImputer(strategy="median")),
        ("scaler", StandardScaler()),
        (
            "classifier",
            XGBClassifier(
                **best_params_xgb_cleaned,  # Use the unpacked, cleaned parameters
                scale_pos_weight=(len(y_train_1) - sum(y_train_1)) / sum(y_train_1),
                random_state=random_state,
            ),
        ),
    ]
)

start = time.time()
# Fit the pipeline on the training data
```

```
best_xgboost_pipeline.fit(X_train_1, y_train_1)
end = time.time()

time_train_xgboost = end - start
print(f"Time to train XGBoost: {round(time_train_xgboost, 4)} seconds")

start = time.time()
# Predict the probabilities on the validation data
y_val_pred_proba_xgb = best_xgboost_pipeline.predict_proba(X_val_1)[:, 1]
end = time.time()

time_val_xgboost = end - start
print(
    f"Time to predict on validation data for XGBoost: {round(time_val_xgboost, 4)} seconds"
)

# Calculate the ROC AUC score
roc_auc_xgboost_val = roc_auc_score(y_val_1, y_val_pred_proba_xgb)

print(f"ROC AUC score for XGBoost on validation data: {round(roc_auc_xgboost_val, 4)}")
```

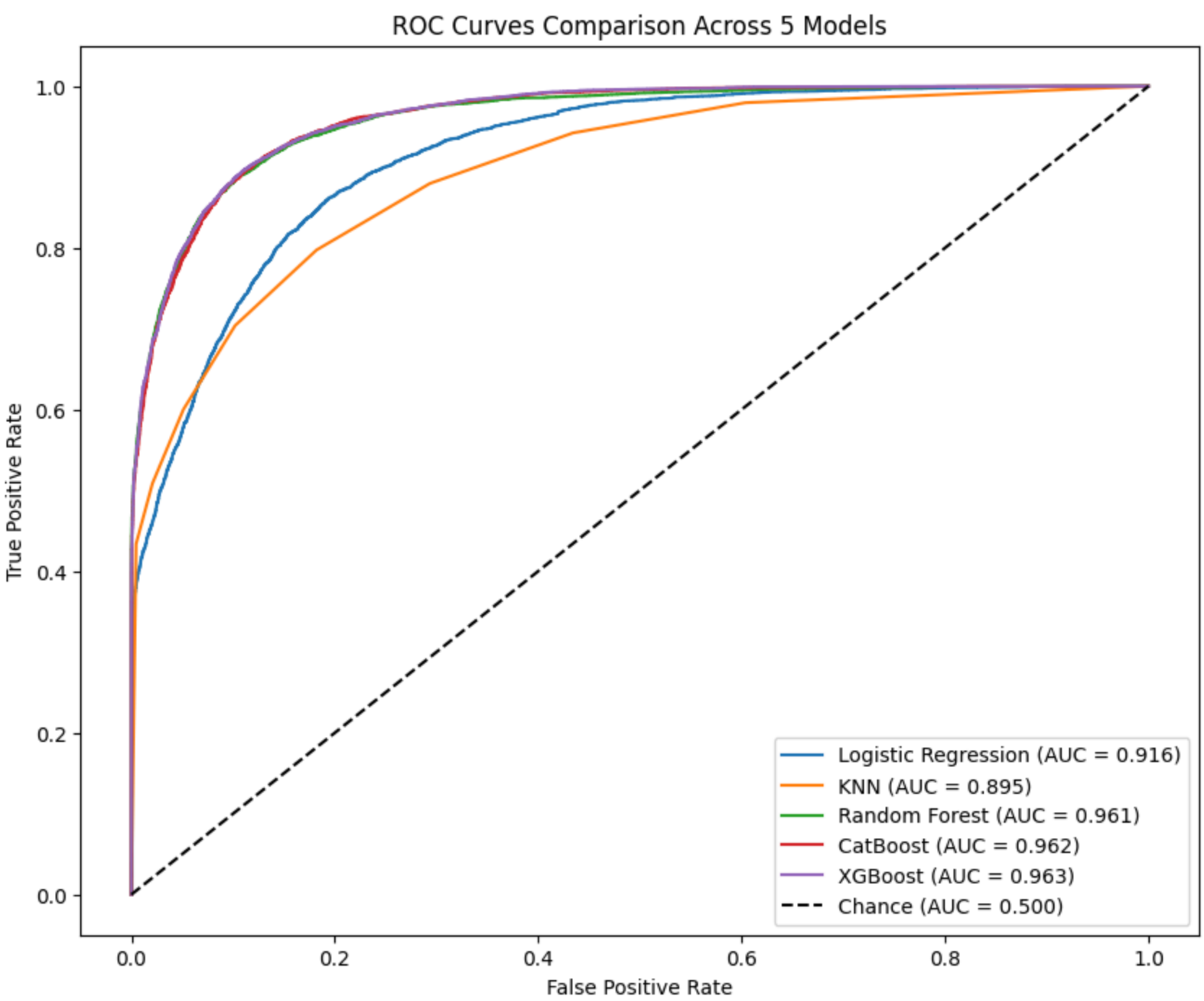
```
Time to train XGBoost: 34.9789 seconds
Time to predict on validation data for XGBoost: 0.5281 seconds
ROC AUC score for XGBoost on validation data: 0.9627
```

In [ ]:
```
# Compute ROC curve for XGBoost
fpr_xgb, tpr_xgb, _ = roc_curve(y_val_1, y_val_pred_proba_xgb)
# Compute AUC score for ROC curve for XGBoost
roc_auc_xgb = auc_score(fpr_xgb, tpr_xgb)

# Compute PR curve for XGBoost
precision_xgb, recall_xgb, _ = precision_recall_curve(y_val_1, y_val_pred_proba_xgb)
# Compute the AP score for XGBoost
average_precision_xgb = average_precision_score(y_val_1, y_val_pred_proba_xgb)
```

> ROC Curve Plot

In [ ]:
```
plt.figure(figsize=(10, 8))

# Plot ROC curve for Logistic Regression
plt.plot(fpr_lr, tpr_lr, label=f"Logistic Regression (AUC = {roc_auc_lr:.3f})")

# Plot ROC curve for KNN
plt.plot(fpr_knn, tpr_knn, label=f"KNN (AUC = {roc_auc_knn:.3f})")

# Plot ROC curve for Random Forest
```

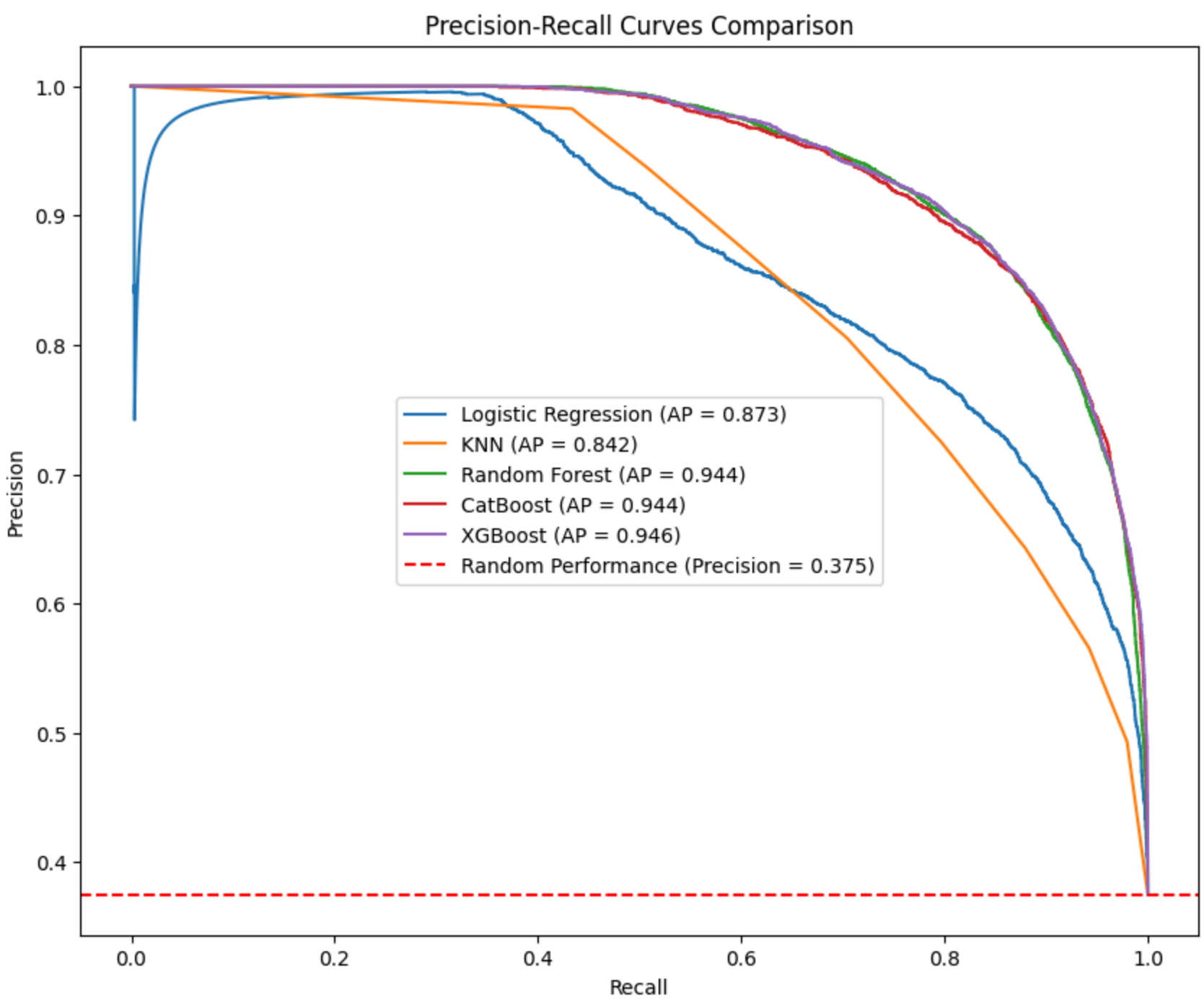
```
plt.plot(fpr_rf, tpr_rf, label=f"Random Forest (AUC = {roc_auc_rf:.3f})")

# Plot ROC curve for CatBoost
plt.plot(fpr_cb, tpr_cb, label=f"CatBoost (AUC = {roc_auc_cb:.3f})")

# Plot ROC curve for XGBoost
plt.plot(fpr_xgb, tpr_xgb, label=f"XGBoost (AUC = {roc_auc_xgb:.3f})")

# Plot chance line
plt.plot([0, 1], [0, 1], "k--", label="Chance (AUC = 0.500)")

plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curves Comparison Across 5 Models")
plt.legend(loc="lower right")
plt.show()
```



PR Curve Plot

In [ ]:
```
plt.figure(figsize=(10, 8))
```

```python
# Plot PR curve for Logistic Regression
plt.plot(
    recall_lr,
    precision_lr,
    label=f"Logistic Regression (AP = {average_precision_lr:.3f})",
)

# Plot PR curve for KNN
plt.plot(recall_knn, precision_knn, label=f"KNN (AP = {average_precision_knn:.3f})")

# Plot PR curve for Random Forest
plt.plot(
    recall_rf, precision_rf, label=f"Random Forest (AP = {average_precision_rf:.3f})"
)

# Plot PR curve for CatBoost
plt.plot(recall_cb, precision_cb, label=f"CatBoost (AP = {average_precision_cb:.3f})")

# Plot PR curve for XGBoost
plt.plot(recall_xgb, precision_xgb, label=f"XGBoost (AP = {average_precision_xgb:.3f})")

# Calculate and plot the random performance level line
P = sum(y_val_1 == 1)
N = sum(y_val_1 == 0)
random_classifier_precision = P / (P + N)
plt.axhline(
    y=random_classifier_precision,
    color="r",
    linestyle="--",
    label=f"Random Performance (Precision = {random_classifier_precision:.3f})",
)

plt.xlabel("Recall")
plt.ylabel("Precision")
plt.title("Precision-Recall Curves Comparison")
plt.legend(loc="best")
plt.show()
```

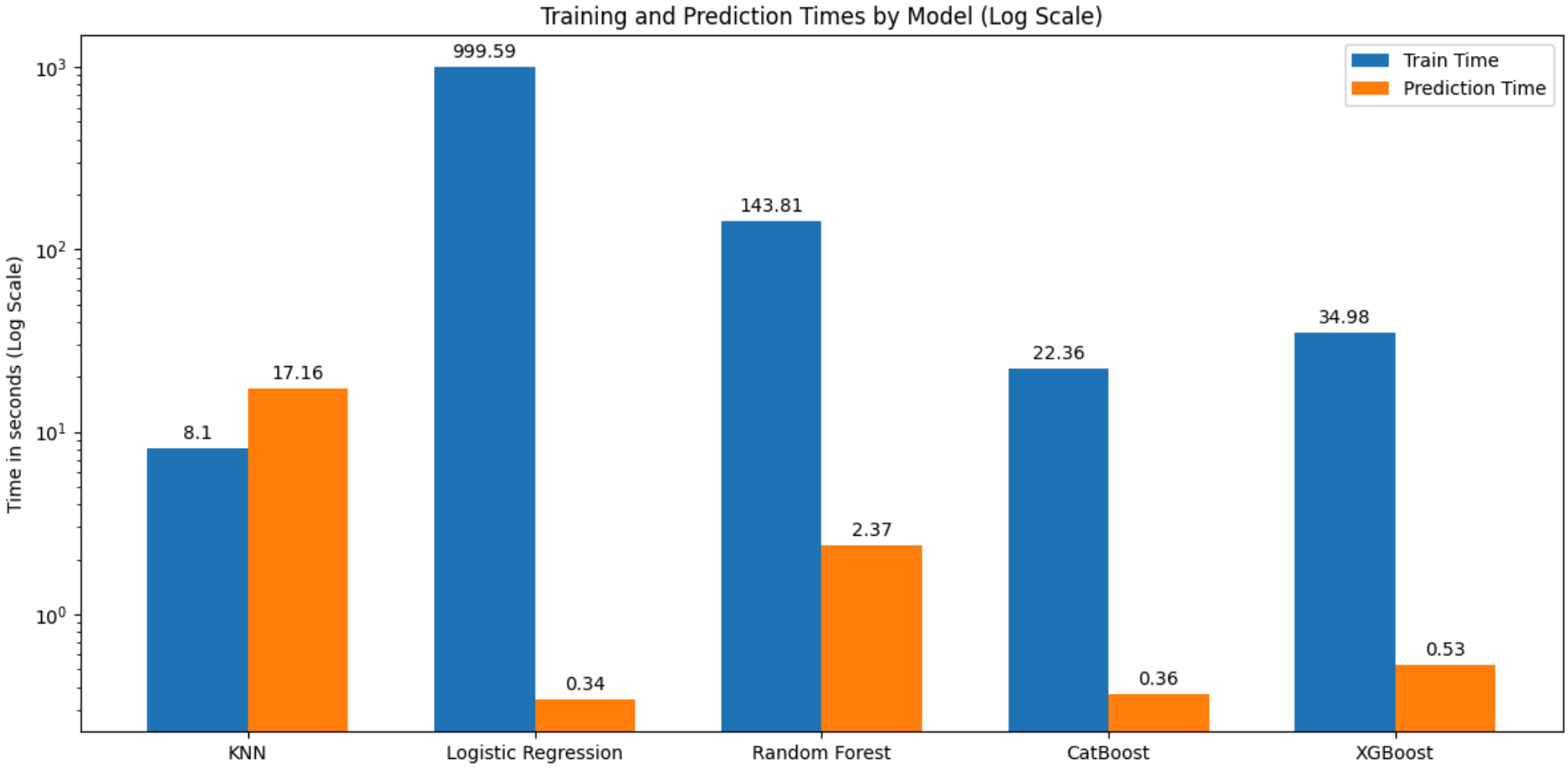


> Time Plot

In [ ]:
```
# training and prediction times in seconds
train_times = [
```

```
    time_train_knn,
    time_train_lr,
    time_train_rf,
    time_train_catboost,
    time_train_xgboost,
]
predict_times = [
    time_val_knn,
    time_val_lr,
    time_val_rf,
    time_val_catboost,
    time_val_xgboost,
]

# Model names
models = ["KNN", "Logistic Regression", "Random Forest", "CatBoost", "XGBoost"]

# Creating the plot
x = np.arange(len(models))  # the label locations
width = 0.35  # the width of the bars

fig, ax = plt.subplots(figsize=(12, 6))
rects1 = ax.bar(x - width / 2, train_times, width, label="Train Time")
rects2 = ax.bar(x + width / 2, predict_times, width, label="Prediction Time")

# Use a logarithmic scale for the y-axis
ax.set_yscale("log")

# Add some text for labels, title, and custom x-axis tick labels.
ax.set_ylabel("Time in seconds (Log Scale)")
ax.set_title("Training and Prediction Times by Model (Log Scale)")
ax.set_xticks(x)
ax.set_xticklabels(models)
ax.legend()


# autolabel bars
def autolabel(rects):
    """Attach a text label above each bar displaying its height."""
    for rect in rects:
        height = rect.get_height()
        ax.annotate(
            "{}".format(round(height, 2)),  # keep 2 deciaml
            xy=(rect.get_x() + rect.get_width() / 2, height),
            xytext=(0, 3),  # 3 points vertical offset
            textcoords="offset points",
            ha="center",
            va="bottom",
```

```
        )


autolabel(rects1)
autolabel(rects2)

fig.tight_layout()

plt.show()
```



Process on model selection and hyperparameter tuning

Logistic Regression

- The logistic regression algorithm is known for its simplicity and effectiveness in binary classification tasks, suitable for my goal of predicting hotel cancellations. For hyperparameter tuning, I focused on the regularization strength (C) and opted for L1 regularization (lasso). The choice of L1 regularization is particularly noteworthy as it not only helps in avoiding

overfitting by penalizing large coefficients but also aids in feature selection by driving some coefficients to zero, which is beneficial for interpretability and model simplicity.

- I employed a grid search approach over 20 logarithmically spaced values of C ranging from 10^{-4} to 10^{4}. This wide range ensures that I thoroughly explore the effect of regularization, from very weak to very strong. The solver 'saga' was chosen for its efficiency with large datasets and L1 penalty, coupled with an increased maximum iteration count to ensure convergence. The performance of each model configuration was evaluated using the AUC metric on the validation dataset. The choice of AUC as the metric aligns with my goal of maximizing the model's discriminatory ability, especially in a likely imbalanced scenario of hotel booking cancellations.
- After plotting the AUC scores against the corresponding C values on a logarithmic scale, I identified the C value that maximized the validation AUC. This process of visual inspection and empirical analysis led to selecting the best C value for the logistic regression model. The final model, configured with the optimal C value, underwent a final training phase. Finally, I timed the training and prediction processes and evaluated the model's performance on the validation data using the AUC score, ROC curve, Precision-Recall (PR) curve, and Average Precision (AP) score, providing a comprehensive overview of the model's ability to predict hotel cancellations.

KNN

- I applied the KNN model to predict hotel cancellations. KNN is a non-parametric, instance-based learning method where predictions for new instances are made based on the similarity to training samples. My hyperparameter tuning focused on finding the optimal number of neighbors (K) to use for predictions, which is crucial in balancing the bias-variance trade-off inherent to KNN. A smaller K can make the model overly sensitive to noise in the training data (high variance), while a larger K might make the model too general, potentially ignoring important subtleties (high bias).
- I conducted a grid search approach over K values from 1 to 50, which provides a broad overview of how the model's performance changes with different neighborhood sizes. I evaluated the model's performance using the AUC score on a validation dataset for each K value. The choice of AUC as the metric aligns with my goal of maximizing the model's discriminatory ability, especially in a likely imbalanced scenario of hotel booking cancellations.
- Upon plotting the AUC scores against their corresponding K values, I identified the K that maximized the validation AUC. The subsequent steps involved retraining the KNN model with the identified optimal K value, timing the training and prediction processes, and evaluating the final model's performance on the validation set using the AUC score, ROC curve, Precision-Recall (PR) curve, and Average Precision (AP) score, providing a comprehensive overview of the model's ability to predict hotel cancellations.

Random Forest

- The Random Forest model is a powerful and versatile machine learning algorithm suitable for predicting hotel cancellations. It operates by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes (classification) of the individual trees. Random Forests are particularly adept at handling tabular data,

which includes a variety of feature types and potentially complex relationships that might be difficult to model with simpler linear models.

- My approach to tuning the hyperparameters in the Random Forest model involved using RandomizedSearchCV, which randomly samples from the defined parameter distributions. This method is often more efficient than grid search, especially when dealing with a large number of hyperparameters and large datasets, as it can explore a wide parameter space without needing to evaluate every possible combination.
- I defined a comprehensive range of hyperparameters to explore:
- **Number of Trees (`n_estimators`):** I allowed for a broad range between 100 to 500 trees, understanding that more trees can improve model accuracy up to a point, beyond which gains diminish.
- **Maximum Depth of Trees (`max_depth`):** The depth of a tree is the length of the longest path from the root node down to a leaf node. By considering values from 5 to 50 as well as allowing for unlimited depth, I sought to find a balance between model complexity and the risk of overfitting.
- **Minimum Samples to Split (`min_samples_split`):** This parameter specifies the minimum number of samples required to split an internal node. Sampling from 2 to 10 ensures that trees do not grow too deep by splitting on too few samples, which can lead to overfitting.
- **Minimum Samples at Leaf Nodes (`min_samples_leaf`):** This parameter defines the minimum number of samples a leaf node must have. This range from 1 to 10 helps in making the model more general by preventing a tree from making splits that result in nodes containing too few samples.
- By using RandomizedSearchCV with these distributions, I effectively searched through a wide but randomized subset of the hyperparameter space. After identifying the best parameters based on the AUC score across a 5-fold cross-validation, I refined my Random Forest pipeline with these optimized parameters.
- After training the optimized Random Forest model, I evaluated its performance on the validation data, focusing on the AUC score, a critical metric for binary classification tasks in imbalanced datasets. I timed the training and prediction processes and evaluated the final model's performance on the validation set using the AUC score, ROC curve, Precision-Recall (PR) curve, and Average Precision (AP) score, providing a comprehensive overview of the model's ability to predict hotel cancellations.

Catboost

- The CatBoost model, a gradient-boosting decision tree algorithm, is specifically designed to handle categorical variables very efficiently, which makes it a compelling choice for predicting hotel cancellations. It utilizes gradient boosting, an ensemble technique that builds trees one at a time, where each new tree helps to correct errors made by previously trained trees. This approach can capture complex patterns and interactions in the data.
- My process of tuning hyperparameters for CatBoost involved RandomizedSearchCV, a probabilistic approach that selects random combinations of parameter values based on specified distributions:

- **Learning Rate (`learning_rate`):** I chose to explore a range of values from 0.01 to 0.3. The learning rate controls the speed at which the model learns, with lower values generally leading to more robust models at the cost of requiring more trees (iterations) to converge.
- **Depth of the Trees (`depth`):** By selecting among depths [4, 6, 8, 10], I allowed for a variety of model complexities, balancing between capturing data intricacies and preventing overfitting.
- **Number of Trees (`iterations`):** The choice of [100, 200, 300, 400, 500] iterations provided a spectrum for the model to combine multiple trees' decisions, optimizing between performance and computation time.
- This randomized search over a predefined parameter space allows for efficient exploration of potential models, identifying a combination that maximizes the AUC score. This metric, critical for binary classification tasks, especially in imbalanced datasets, serves as a robust indicator of model performance by measuring its ability to distinguish between the two classes.
- After training the optimized Catboost model, I evaluated its performance on the validation data, focusing on the AUC score, a critical metric for binary classification tasks in imbalanced datasets. I timed the training and prediction processes and evaluated the final model's performance on the validation set using the AUC score, ROC curve, Precision-Recall (PR) curve, and Average Precision (AP) score, providing a comprehensive overview of the model's ability to predict hotel cancellations.

XGboost

- XGBoost is a highly effective gradient-boosting decision tree algorithm for supervised learning tasks. XGBoost is built on the principle of gradient boosting, which involves sequentially adding predictors (decision trees) to an ensemble, each one correcting its predecessor. It's particularly well-suited for classification tasks like predicting hotel cancellations due to its efficiency, accuracy, and ability to handle large datasets with a mixture of categorical and numerical features.
- I've employed a comprehensive and strategic approach to tuning XGBoost's hyperparameters using RandomizedSearchCV, focusing on:
- `n_estimators` **(Number of Trees):** Varying from 100 to 1000, this range allows the model to explore both simpler (fewer trees) and more complex (more trees) models. A higher number of trees can improve model accuracy but also increases the risk of overfitting and computational cost.
- `learning_rate` **:** This controls the step size at each iteration while moving toward a minimum of a loss function. By exploring values from 0.01 to 0.3, I investigated how fast the model learns, balancing between quick convergence and the risk of overshooting the minimum.
- `max_depth` **:** This parameter sets the maximum depth of each tree. With values from 3 to 10, I looked to optimize the complexity of the model. Deeper trees can model complex patterns but might lead to overfitting.
- `subsample` **:** By setting this between 0.6 and 1, I controlled the fraction of the training dataset sampled without replacement for building each tree. Subsampling helps in making the model more robust against noise and overfitting.

- **`colsample_bytree`:** This parameter specifies the fraction of features to be randomly sampled for each tree. Setting this between 0.6 and 1, it helps in preventing overfitting and adds to the randomness of the model, encouraging feature diversity in the trees.
- By using RandomizedSearchCV with these distributions, I effectively searched through a wide but randomized subset of the hyperparameter space. After identifying the best parameters based on the AUC score across a 5-fold cross-validation, I refined my XGboost pipeline with these optimized parameters.
- After training the optimized XGboost model, I evaluated its performance on the validation data, focusing on the AUC score, a critical metric for binary classification tasks in imbalanced datasets. I timed the training and prediction processes and evaluated the final model's performance on the validation set using the AUC score, ROC curve, Precision-Recall (PR) curve, and Average Precision (AP) score, providing a comprehensive overview of the model's ability to predict hotel cancellations.

5 Model Performance Evaluation

- In the analysis of ROC curves, Random Forest, CatBoost, and XGBoost showcased exceptional performance, each achieving an AUC score in the vicinity of 0.96, indicating their strong capability in distinguishing between the classes. Despite their robust performance, a nuanced difference placed XGBoost slightly ahead with an AUC score of approximately 0.963, edging out its competitors in this metric. In contrast, KNN and Logistic Regression lagged in their performance, with KNN registering the lowest AUC score of about 0.895. Logistic Regression, on the other hand, recorded an AUC score of approximately 0.916, positioning it better than KNN but still behind the ensemble methods.
- The evaluation of PR curves further reinforced these findings, with Random Forest, CatBoost, and XGBoost once again performing comparably well, each securing an Average Precision (AP) score of around 0.94. This consistency across both ROC and PR analyses underscores their effectiveness in handling the prediction task, with XGBoost marginally surpassing the others with an AP score of about 0.946. Conversely, KNN and Logistic Regression lagged in their performance; KNN notably achieved the lowest AP score at approximately 0.842, and Logistic Regression's performance was marked by an initial dip in its PR curve, culminating in an AP score around 0.873.
- In consideration of computational efficiency, measured by training and prediction times. KNN uniquely required more time for prediction (17.16 seconds) than for training (8.1 seconds), an anomaly among the models evaluated. Logistic Regression, while taking the longest to train (999.59 seconds), boasted the quickest prediction time (0.34 seconds), presenting a stark contrast in its computational demands. Notably, CatBoost and XGBoost demonstrated exceptional efficiency, balancing expedient training times (22.36 seconds for CatBoost and 34.98 seconds for XGBoost) with swift prediction capabilities (under 1 second for both), evidencing their practical utility in time-sensitive applications.
- In conclusion, XGBoost emerges as the superior model among those evaluated considering the highest AUC and AP scores and good computational efficiency. Its top-tier performance metrics, coupled with its relatively fast training and prediction times, underscore its suitability for the task of predicting hotel cancellations.

(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 5 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.

> Catboost

In [ ]:
```
# the CatBoost pipeline with the best parameters
final_catboost_pipeline = Pipeline(
    [
        ("imputer", SimpleImputer(strategy="median")),
        ("scaler", StandardScaler()),
        (
            "classifier",
            CatBoostClassifier(
                **best_params_catboost_cleaned,  # Use the unpacked, cleaned parameters
                auto_class_weights="Balanced",
                verbose=0,
                random_state=random_state,  # Ensure random_state is defined
            ),
        ),
    ]
)

# Fit the pipeline on the training data + validation data
final_catboost_pipeline.fit(X_train_plus_val, y_train_plus_val)

# Predict the probabilities on the test data
y_test_pred_proba_cb = final_catboost_pipeline.predict_proba(X_test_ohe)[:, 1]

# Path to save the submission file
submission_save_path = "./submission_catboost.csv"

# Generate submission file
create_submission(y_test_pred_proba_cb, submission_save_path)
```

In [ ]:
```
catboost_result = pd.read_csv("submission_catboost.csv")
```

```
print(f"The shape of the CatBoost submission file is {catboost_result.shape}")
```

```
The shape of the CatBoost submission file is (23878, 2)
```

In [ ]:
```
print("The first few rows of the CatBoost submission file are:")
catboost_result.head()
```

```
The first few rows of the CatBoost submission file are:
```

Out[ ]:

	id	score
0	0	0.999568
1	1	0.000264
2	2	0.055418
3	3	0.002346
4	4	0.486375

> XGBoost

In [ ]:
```
# the XGBoost pipeline with the best parameters
final_xgboost_pipeline = Pipeline(
    [
        ("imputer", SimpleImputer(strategy="median")),
        ("scaler", StandardScaler()),
        (
            "classifier",
            XGBClassifier(
                **best_params_xgb_cleaned,  # Use the unpacked, cleaned parameters
                scale_pos_weight=(len(y_train_plus_val) - sum(y_train_plus_val))
                / sum(y_train_plus_val),
                random_state=random_state,
            ),
        ),
    ]
)

# Fit the pipeline on the training data + validation data
final_xgboost_pipeline.fit(X_train_plus_val, y_train_plus_val)

# Predict the probabilities on the test data
y_test_pred_proba_xgb = final_xgboost_pipeline.predict_proba(X_test_ohe)[:, 1]

# Path to save the submission file
submission_save_path = "./submission_xgboost.csv"
```

```
# Generate submission file
create_submission(y_test_pred_proba_xgb, submission_save_path)
```

In [ ]:
```
xgboost_result = pd.read_csv("submission_xgboost.csv")

print(f"The shape of the XGBoost submission file is {xgboost_result.shape}")
```

```
The shape of the XGBoost submission file is (23878, 2)
```

In [ ]:
```
print("The first few rows of the XGBoost submission file are:")
xgboost_result.head()
```

```
The first few rows of the XGBoost submission file are:
```

Out[ ]:

	id	score
0	0	0.999896
1	1	0.000158
2	2	0.214674
3	3	0.000218
4	4	0.680717

> Random Forest

In [ ]:
```
# Define the Random Forest pipeline with the best parameters
final_random_forest_pipeline = Pipeline(
    [
        ("imputer", SimpleImputer(strategy="median")),
        ("scaler", StandardScaler()),
        (
            "classifier",
            RandomForestClassifier(
                **best_params_rf_cleaned,  # Use the unpacked, cleaned parameters
                class_weight="balanced",
                random_state=random_state,
            ),
        ),
    ]
)

# Fit the pipeline on the training data + validation data
final_random_forest_pipeline.fit(X_train_plus_val, y_train_plus_val)
```

```
# Predict the probabilities on the test data
y_test_pred_proba_rf = final_random_forest_pipeline.predict_proba(X_test_ohe)[:, 1]

# Path to save the submission file
submission_save_path_rf = "./submission_random_forest.csv"

# Generate submission file
create_submission(y_test_pred_proba_rf, submission_save_path_rf)
```

In [ ]:
```
random_forest_result = pd.read_csv("submission_random_forest.csv")

print(f"The shape of the Random Forest submission file is {random_forest_result.shape}")
```

```
The shape of the Random Forest submission file is (23878, 2)
```

In [ ]:
```
print("The first few rows of the Random Forest submission file are:")
random_forest_result.head()
```

```
The first few rows of the Random Forest submission file are:
```

Out[ ]:

	id	score
0	0	0.999915
1	1	0.162338
2	2	0.212855
3	3	0.026141
4	4	0.647628

> KNN

In [ ]:
```
# Define the final KNN pipeline with the best parameters (n_neighbors=8)
final_knn_pipeline = Pipeline(
    [
        ("imputer", SimpleImputer(strategy="median")),
        ("scaler", StandardScaler()),
        ("classifier", KNeighborsClassifier(n_neighbors=best_k)),  # use best k value
    ]
)

# Fit the final pipeline on the combined training and validation data
final_knn_pipeline.fit(X_train_plus_val, y_train_plus_val)
```

```
# Predict the probabilities on the test data
y_test_pred_proba_knn = final_knn_pipeline.predict_proba(X_test_ohe)[:, 1]

# Path to save the submission file for the final KNN model
submission_save_path_knn = "./submission_knn.csv"

# Generate the submission file with predictions from the final KNN model
create_submission(y_test_pred_proba_knn, submission_save_path_knn)
```

In [ ]:
```
knn_result = pd.read_csv("submission_knn.csv")

print(f"The shape of the KNN submission file is {knn_result.shape}")
```

```
The shape of the KNN submission file is (23878, 2)
```

In [ ]:
```
print("The first few rows of the KNN submission file are:")
knn_result.head()
```

```
The first few rows of the KNN submission file are:
```

Out[ ]:

	id	score
0	0	1.000
1	1	0.750
2	2	0.000
3	3	0.000
4	4	0.375

> Logistic Regression

In [ ]:
```
# Define the final logistic regression pipeline with the best parameters
final_lr_pipeline = Pipeline(
    [
        ("imputer", SimpleImputer(strategy="median")),
        ("scaler", StandardScaler()),
        (
            "classifier",
            LogisticRegression(
                class_weight="balanced",
                penalty="l1",
                C=best_c,  # Use the best C value
                solver="saga",
                max_iter=1000,
```

```
                random_state=random_state,
            ),
        ),
    ]
)

# Fit the final pipeline on the combined training and validation data
final_lr_pipeline.fit(X_train_plus_val, y_train_plus_val)

# Predict the probabilities on the test data
y_test_pred_proba_lr = final_lr_pipeline.predict_proba(X_test_ohe)[:, 1]

# Define the path to save the submission file for the final logistic regression model
final_submission_save_path_lr = "./submission_lr.csv"

# Generate the submission file with predictions from the final logistic regression model
create_submission(y_test_pred_proba_lr, final_submission_save_path_lr)
```

In [ ]:
```
lr_result = pd.read_csv("submission_lr.csv")

print(f"The shape of the Logistic Regression submission file is {lr_result.shape}")
```

```
The shape of the Logistic Regression submission file is (23878, 2)
```

In [ ]:
```
print("The first few rows of the Logistic Regression submission file are:")
lr_result.head()
```

```
The first few rows of the Logistic Regression submission file are:
```

Out[ ]:

	id	score
0	0	0.998231
1	1	0.000807
2	2	0.000721
3	3	0.000055
4	4	0.445107

2

[25 points] Clustering