Machine Learning for Predicting Airline Customer Satisfaction

Introduction

Customer experience is an important concern for airline industries, as dissatisfied customers translate to less revenue for these industries.

Airlines can utilize customer feedback surveys to discover customers' perceptions of the services they offer. Through Data Analysis, airlines can derive valuable insights from customer feedback, make informed business decisions to meet customers' expectations and attract new customers while retaining existing ones.

Machine learning can be used to predict customer satisfaction when a service is provided.

The aim of this project is to build a machine learning model to predict customer satisfaction. We will also use the LIME explainability technique to understand the contribution of each feature in our prediction.

Dataset

The Dataset for this project is from Kaggle. The data is from an airline organization whose actual name is not given for various reasons, therefore, the airline is given the pseudonym Invistico airlines.

The dataset consists of (23 columns and 129880 entries) details of customers who have already flown with them. The feedback of the customers on various contexts and their flight data has been consolidated.

The main purpose of this dataset is to predict whether a future customer would be satisfied with their service given the details of the other parameters / values.

In addition, airlines need to know which aspects of their services to emphasize to generate more satisfied customers.

Exploration and Data Analysis

We explored the details of the customers surveyed in this dataset to have an understanding of their unique features and any salient aspects that affected customer flight satisfaction.

Our participants were a relatively young group overall with participants ranging from 7 to 85 years and an average age of approximately 39 years.

Overall, the dataset showed more satisfied fliers than otherwise, with 54.7% of the surveyed customers reporting satisfaction with their experiences

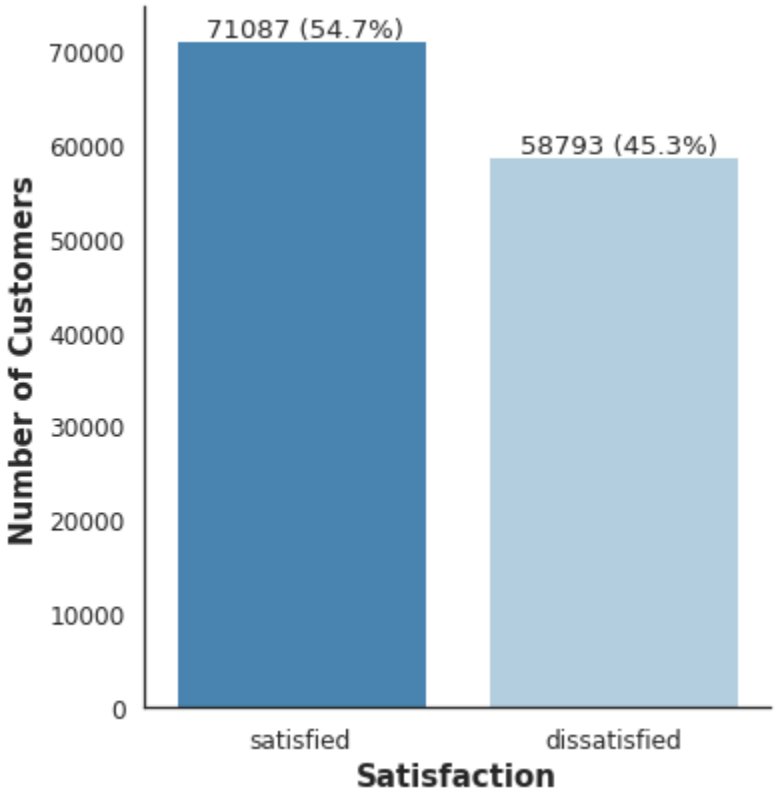

Figure 1:Barchart showing Customer Satisfaction

There were more female travelers than males in our participant sample and more female fliers reported satisfaction with their experiences while a higher proportion of the male participants had unsatisfactory experiences.

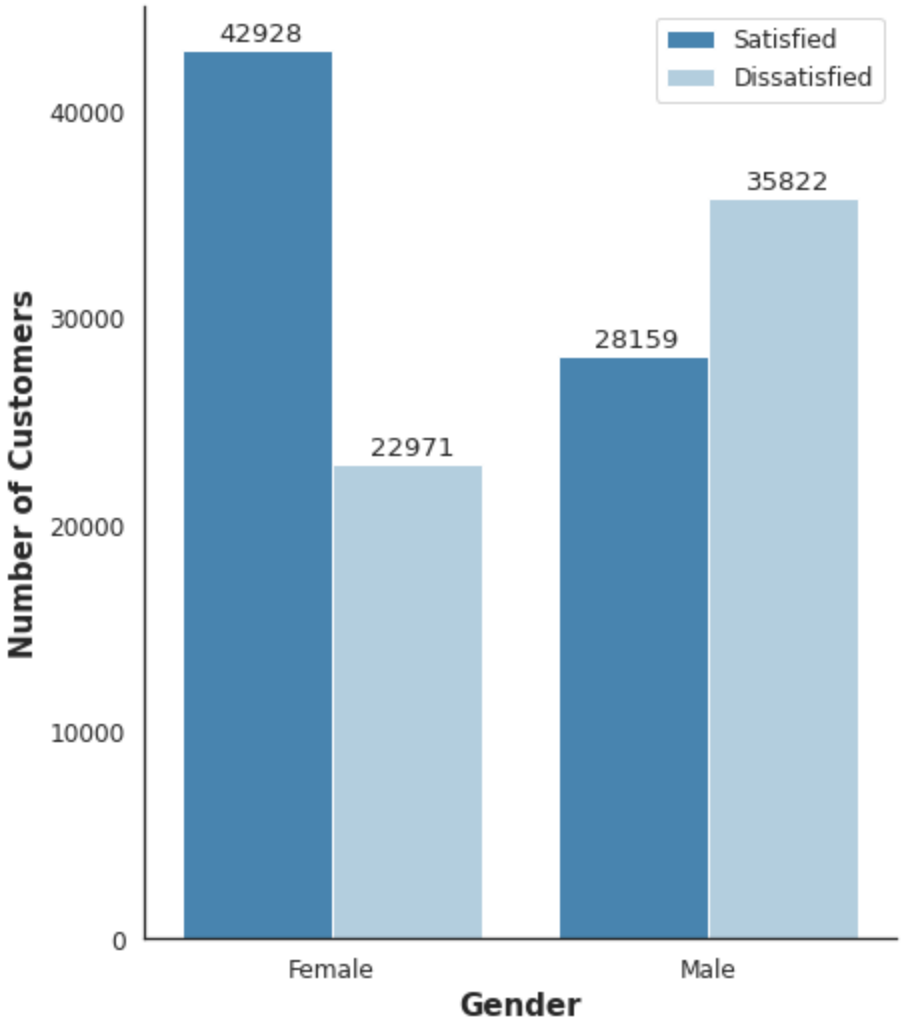

Figure 2: Barchart showing Gender Distribution of Satisfied/Dissatisfied Customers

Most of our participants traveled for business rather than for personal reasons and satisfaction was proportionately higher in business travelers as more individuals who traveled for personal reasons had unsatisfactory experiences than otherwise.

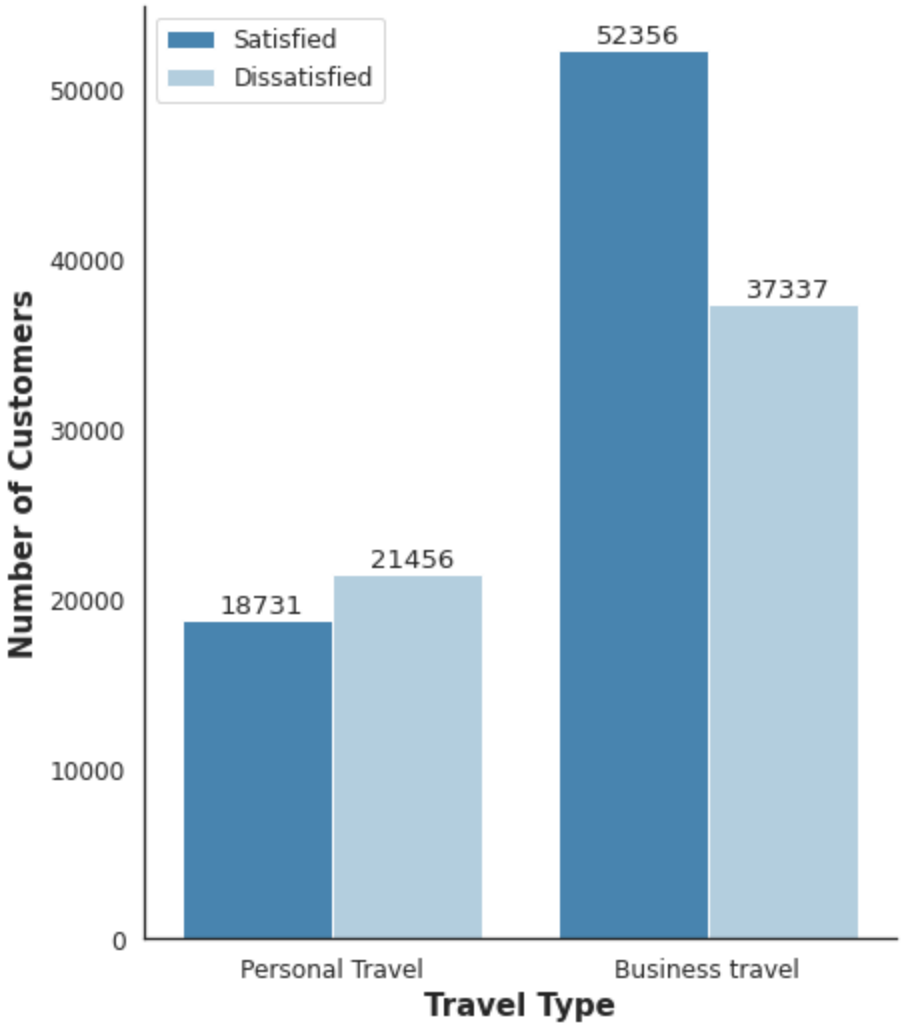

Figure 3: Barchart showing Distribution of Travel Type and Satisfaction

In terms of flight seat classes, the business class seats were the only category of the three with a higher proportion of satisfied customers than otherwise. This could point to either a better experience overall in those class of seats or certain deficiencies in the Eco and Eco plus seats.

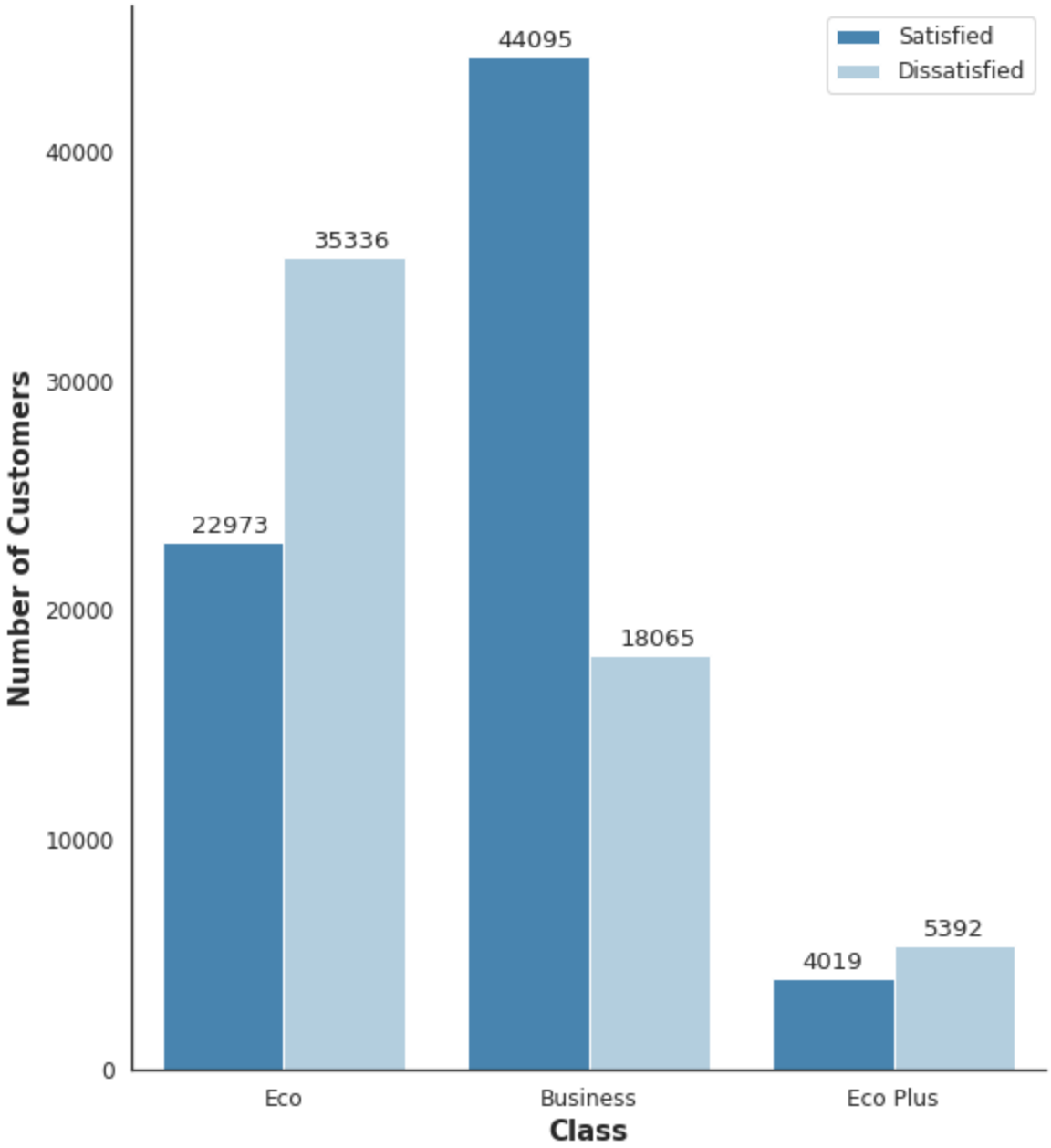

Figure 4: Barchart showing Class Distribution and Customer Satisfaction

The highest direct correlation with satisfaction in customers was the presence of the inflight entertainment (0.52). Delay periods, on arrival or departure, surprisingly had a weak correlation with satisfaction.

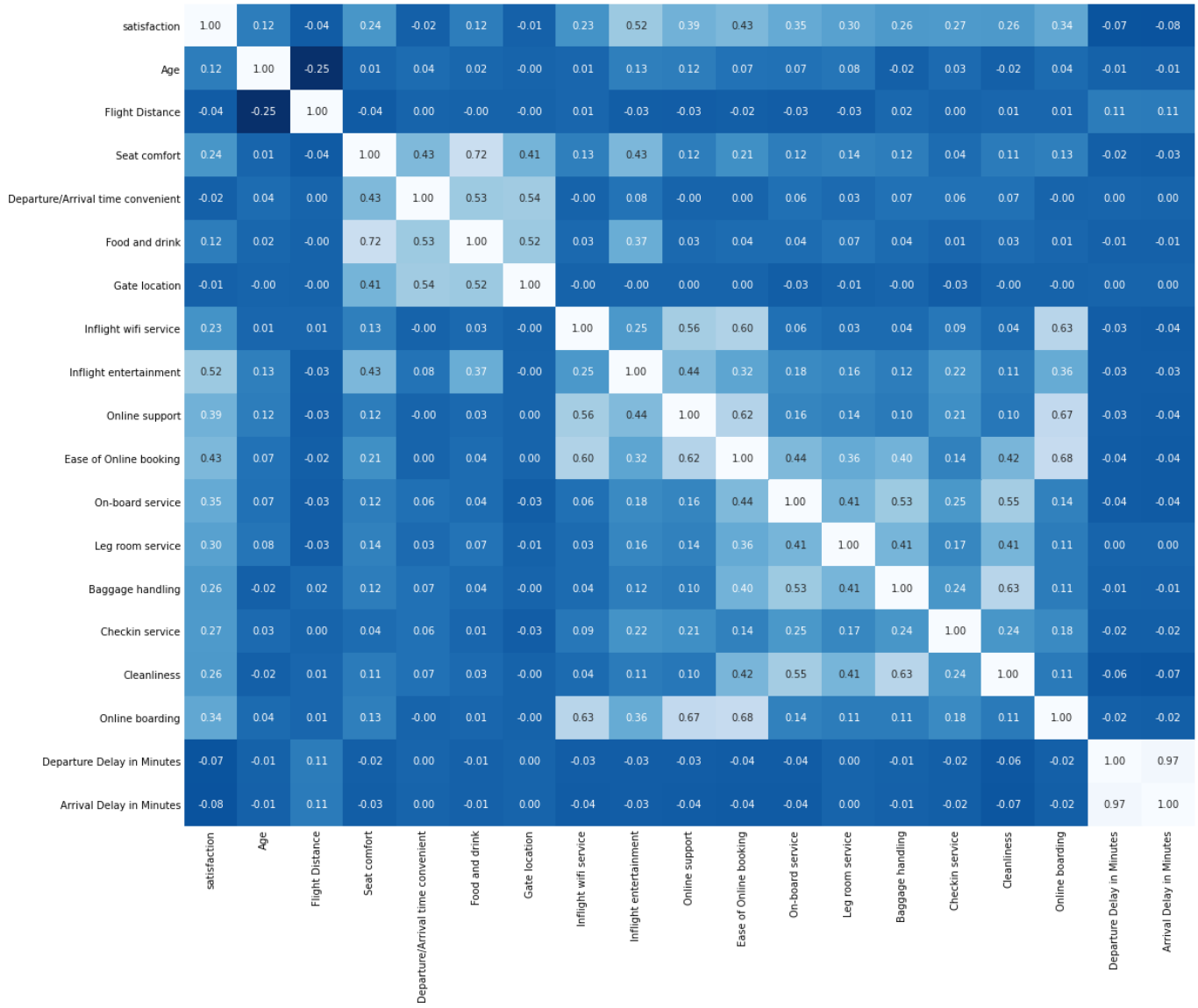

Figure 5: Feature Correlation

Preprocessing and Modelling

To make the data fit our machine learning model, we performed the following preprocessing steps:

1. Removing outliers

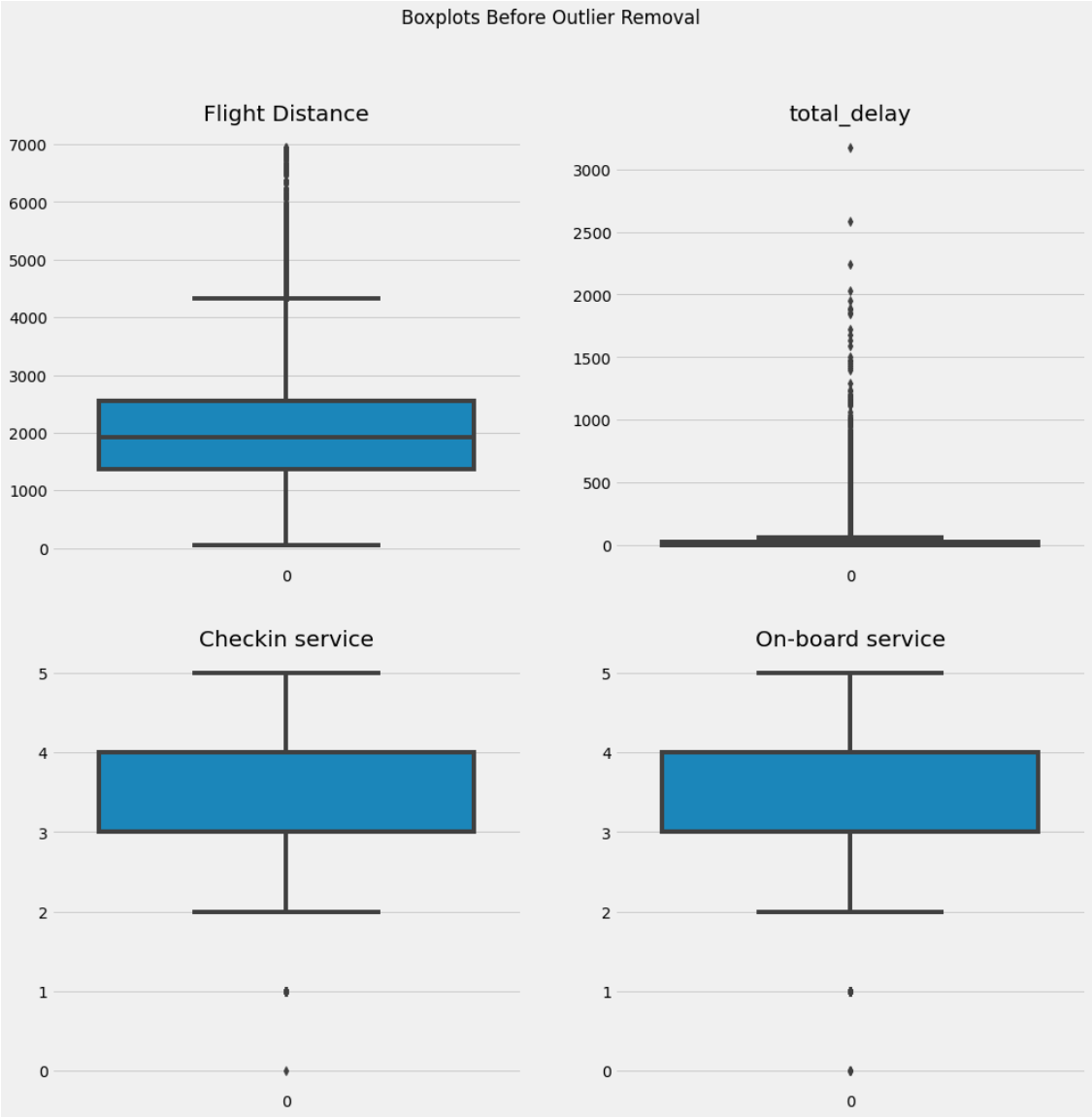

Figure 6: Boxplots of Features with Outliers before Outlier removal

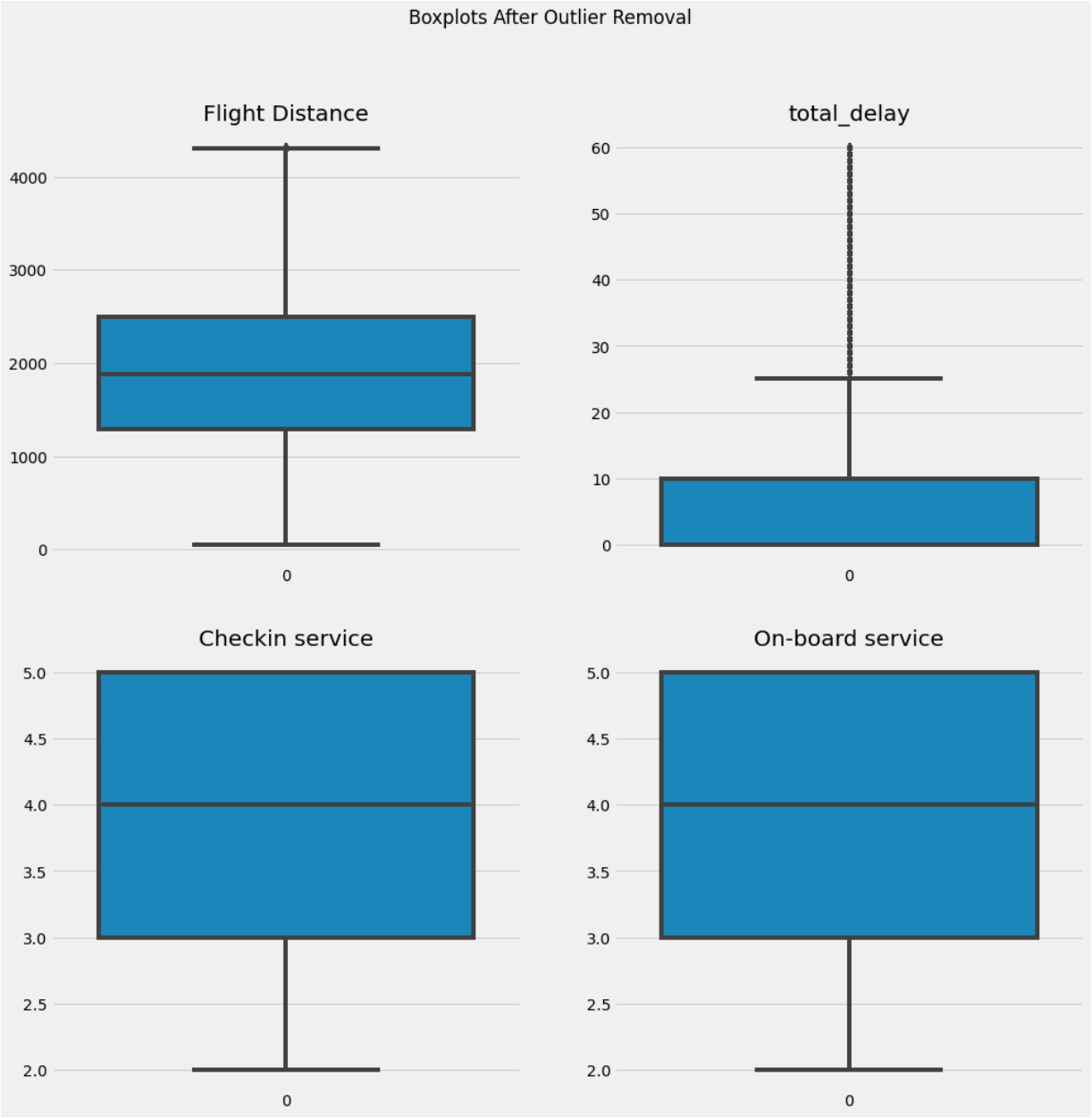

Figure 7: Boxplots of Features with Outliers after Outlier removal

2. Dropping rows with null values
3. Converting categorical features to numbers: The categorical features were mapped as follows:

Categorical Feature	**Mapping**
Satisfaction	Satisfied: 1 Dissatisfied: 0

Gender	Male: 1 Female: 2
Customer Type	Loyal Customer:1 Disloyal Customer: 0
Type of Travel	Business Travel: 1 Personal Travel: 2
Class	Business: 1 Eco Plus: 2 Eco: 3

We trained our dataset on a tensorflow model with an accuracy of approximately 90%.

Tensorflow Operator for Model Training

An Operator is a method of packaging, deploying and managing a stateful Kubernetes application which in this context is a Machine Learning Job. Operators are software written to encapsulate all operational considerations for a specific Kubernetes application and ensure that all aspects of its lifecycle, from configuration and deployment to upgrades, monitoring, and failure-handling, are integrated right into the Kubernetes framework and invoked when needed. An ML Operator can be made for a range of actions from basic functionalities to specific logic for an ML Job.

Tensorflow Operator is one of the operators offered by Kubeflow to make it easy to run and monitor both distributed and non-distributed tensorflow jobs on Kubernetes. Training tensorflow models using tf-operator relies on centralized parameter servers for coordination between workers. It supports the tensorflow framework only.

Tensorflow Operator for the Airline Customer Satisfaction Dataset

Here, we go through the process of creating a TensorFlow Operator with our Dataset:

1. Check that the right image, TensorFlow is available:

```
! pip3 list | grep tensorflow
! pip3 install --user tensorflow==2.4.0
! pip3 install --user ipywidgets nbconvert
!python -m pip install --user --upgrade pip
!pip3 install pandas scikit-learn keras --user
```

2. To package the trainer in a container image, we shall need a file (on our cluster) that contains the code as well as a file with the resource definition of the job for the Kubernetes cluster:

```
TRAINER_FILE = "tfjobairline.py"
KUBERNETES_FILE = "tfjob-airline.yaml"
```

3. Define a helper function to capture output from a cell with %%capture that looks like some-resource created:

```
import re

from IPython.utils.capture import CapturedIO


def get_resource(captured_io: CapturedIO) -> str:
    """
    Gets a resource name from `kubectl apply -f
<configuration.yaml>`.

    :param str captured_io: Output captured by using `%%capture`
cell magic
    :return: Name of the Kubernetes resource
    :rtype: str
    :raises Exception: if the resource could not be created
    """
    out = captured_io.stdout
    matches = re.search(r"^(.+)\s+created", out)
    if matches is not None:
        return matches.group(1)
    else:
        raise Exception(f"Cannot get resource as its creation
failed: {out}. It may already exist.")
```

4. Load and Inspect the Data:

```
import pandas as  pd
data = pd.read_csv("Invistico_Airline.csv")
data.head()
```

5. Train the Model in the Notebook:

We trained the model in a distributed fashion and put all the code in a single cell. That way we could save the file and include it in a container image.

```
%%writefile $TRAINER_FILE
import argparse
import logging
import json
import os
import warnings
warnings.filterwarnings("ignore", category=DeprecationWarning)

import numpy as np
import pandas as pd

from sklearn.model_selection import train_test_split as tts
from sklearn.preprocessing import StandardScaler

from numpy.random import seed

import tensorflow as tf
tf.random.set_seed(221)
from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout,
BatchNormalization
from tensorflow.keras.optimizers import SGD, Adam, RMSprop

logging.getLogger().setLevel(logging.INFO)

def make_datasets_unbatched():
    df = pd.read_csv("Invistico_Airline.csv")
    #drop rows with missing values
    df.dropna(inplace=True)
```

```
    #new column total delay
    df['total_delay'] = df['Departure Delay in Minutes'] +
df['Arrival Delay in Minutes']

    #drop 'Departure Delay in Minutes',and 'Arrival Delay in Minutes'
    df.drop(columns=['Departure Delay in Minutes','Arrival Delay in
Minutes'], inplace=True)

        #satisfied and dissatisfied in number
    satisfaction_map = {"satisfied": 1,"dissatisfied": 0 }
    df['satisfaction']  = df['satisfaction'].map(satisfaction_map)

    #Male and Female in number
    Gender_map = {"Male": 1,"Female": 2 }
    df['Gender']  = df['Gender'].map(Gender_map)

    #Loyal and disloyal in number
    Customer_Type_map = {"Loyal Customer": 1,"disloyal Customer": 0 }
    df['Customer Type']  = df['Customer Type'].map(Customer_Type_map)

    #Business travel and Business travel in number
    Type_of_Travel_map = {"Business travel": 1,"Personal Travel": 2 }
    df['Type of Travel']  = df['Type of
Travel'].map(Type_of_Travel_map)

    #Business and Eco and Eco plus in number
    Class_map = {"Business": 1,"Eco": 3, "Eco Plus": 2 }
    df['Class']  = df['Class'].map(Class_map)

    cols = ['Flight Distance', 'total_delay', 'Checkin service',
'On-board service']

    Q1 = df[cols].quantile(0.25)
    Q3 = df[cols].quantile(0.75)
    IQR = Q3 - Q1

    df = df[~((df[cols] < (Q1 - 1.5 * IQR)) |(df[cols] > (Q3 + 1.5 *
IQR))).any(axis=1)]
```

```
    #Split dataset

    X = df.drop('satisfaction',axis=1)
    y = df['satisfaction']
    X_train,X_test, y_train, y_test = train_test_split(X,y, test_size
= 0.3, random_state = 111)


    train_dataset = tf.data.Dataset.from_tensor_slices((X_train,
y_train))
    test_dataset = tf.data.Dataset.from_tensor_slices((X_test,
y_test))
    train = train_dataset.cache().shuffle(2000).repeat()
    return train, test_dataset

def model(args):
    seed(1)
    model = Sequential()
    model.add(Dense(100, activation='relu', input_dim=21))
    model.add(BatchNormalization())
    model.add(Dense(40, activation='relu'))
    model.add(Dropout(0.2))
    model.add(Dense(1, activation='sigmoid'))

    model.summary()
    opt = args.optimizer
    model.compile(optimizer=opt,
                loss='binary_crossentropy',
                metrics=['accuracy'])
    tf.keras.backend.set_value(model.optimizer.learning_rate,
args.learning_rate)
    return model


def main(args):
    # MultiWorkerMirroredStrategy creates copies of all variables in
the model's
    # layers on each device across all workers
    strategy =
```

```
tf.distribute.experimental.MultiWorkerMirroredStrategy(

communication=tf.distribute.experimental.CollectiveCommunication.AUTO
)
    logging.debug(f"num_replicas_in_sync:
{strategy.num_replicas_in_sync}")
    BATCH_SIZE_PER_REPLICA = args.batch_size
    BATCH_SIZE = BATCH_SIZE_PER_REPLICA *
strategy.num_replicas_in_sync

    # Datasets need to be created after instantiation of
`MultiWorkerMirroredStrategy`
    train_dataset, test_dataset = make_datasets_unbatched()
    train_dataset = train_dataset.batch(batch_size=BATCH_SIZE)
    test_dataset = test_dataset.batch(batch_size=BATCH_SIZE)

    # See:
https://www.tensorflow.org/api_docs/python/tf/data/experimental/Distr
ibuteOptions
    options = tf.data.Options()
    options.experimental_distribute.auto_shard_policy = \
    tf.data.experimental.AutoShardPolicy.DATA

    train_datasets_sharded  = train_dataset.with_options(options)
    test_dataset_sharded = test_dataset.with_options(options)

    with strategy.scope():
        # Model building/compiling need to be within
`strategy.scope()`.
        multi_worker_model = model(args)
        # Keras' `model.fit()` trains the model with specified number
of epochs and
        # number of steps per epoch.
        multi_worker_model.fit(train_datasets_sharded,
                               epochs=50,
                               steps_per_epoch=30)

        eval_loss, eval_acc =
multi_worker_model.evaluate(test_dataset_sharded,
```

```
                                                    verbose=0,
steps=10)
        # Log metrics for Katib
        logging.info("loss={:.4f}".format(eval_loss))
        logging.info("accuracy={:.4f}".format(eval_acc))

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument("--batch_size",
                        type=int,
                        default=128,
                        metavar="N",
                        help="Batch size for training (default: 128)")
    parser.add_argument("--learning_rate",
                        type=float,
                        default=0.1,
                        metavar="N",
                        help='Initial learning rate')
    parser.add_argument("--optimizer",
                        type=str,
                        default='adam',
                        metavar="N",
                        help='optimizer')
    parsed_args, _ = parser.parse_known_args()
    main(parsed_args)
```

That saves the file as defined by TRAINER_FILE but it does not run it.

6. Create a Docker Image:

The Docker file looks as follows:

```
FROM tensorflow/tensorflow:2.4.0
RUN pip install pandas scikit-learn keras
COPY tfjobairline.py /
ENTRYPOINT ["python", "/tfjobairline.py", "--batch_size", "64",
"--learning_rate", "0.1", "--optimizer", "adam"]
```

7. Check if the code is correct by running it from within the notebook:

```
%run $TRAINER_FILE --optimizer 'adam'
```

8. Create a Distributed TFJob:

For large training jobs, we wish to run our trainer in a distributed model. Once the notebook server cluster can access the Docker image from the registry, we can launch a distributed TF job.

The specification for a distributed TFJob is defined using YAML:

```
%%writefile $KUBERNETES_FILE
apiVersion: "kubeflow.org/v1"
kind: "TFJob"
metadata:
  name: "airline"
  namespace: - # your-user-namespace
spec:
  cleanPodPolicy: None
  tfReplicaSpecs:
    Worker:
      replicas: 2
      restartPolicy: OnFailure
      template:
        metadata:
          annotations:
            sidecar.istio.io/inject: "false"
        spec:
          containers:
          - name: tensorflow
            # modify this property if you would like to use a custom
image
            image: mavencodev/tfjob_airline:v.0.1
            command:
                - "python"
                - "/tfjobairline.py"
                - "--batch_size=64"
                - "--learning_rate=0.1"
                - "--optimizer=adam"
```

9. Deploy the distributed training job:

```
%%capture tf_output --no-stderr
! kubectl create -f $KUBERNETES_FILE
```

```
TF_JOB = get_resource(tf_output)
```

10. See the job status:

```
! kubectl describe $TF_JOB
```

11. See the created pods:

```
! kubectl get pods -l job-name=airline
```

12. Stream logs from the worker-0 pod to check the training progress:

```
! kubectl logs -f airline-worker-0
```

13. Delete the job:

```
#! kubectl delete $TF_JOB
```

14. Check to see if the pod is still up and running:

```
#! kubectl -n ekemini logs -f airline
```

Hyperparameter Tuning with Katib

Hyperparameter tuning is the process of optimizing a model's hyperparameter values to maximize the predictive quality of the model. Katib automates the Hyperparameter Tuning process thereby eliminating errors that arise from manual intervention and also saves much-needed resources. Katib is agnostic to ML Frameworks and supports a variety of traditional Hyperparameter Tuning Algorithms. Its concepts are Experiments, Suggestions, Trials, and WorkerJob which are all Custom Resource Definitions integrated on the Kubernetes Engine.

This section shows how to create and configure an Experiment for the TensorFlow training job. In terms of Kubernetes, such an experiment is a Custom Resource Definition (CRD) run by the Katib operator.

How to Create Experiments

1. Set up a few basic definitions that can be reused:

```
TF_EXPERIMENT_FILE = "katibairline.yaml"
```

```
import re

from IPython.utils.capture import CapturedIO


def get_resource(captured_io: CapturedIO) -> str:
    """
    Gets a resource name from `kubectl apply -f
<configuration.yaml>`.

    :param str captured_io: Output captured by using
`%%capture` cell magic
    :return: Name of the Kubernetes resource
    :rtype: str
    :raises Exception: if the resource could not be created
    """
    out = captured_io.stdout
    matches = re.search(r"^(.+)\s+created", out)
    if matches is not None:
        return matches.group(1)
    else:
        raise Exception(f"Cannot get resource as its creation
failed: {out}. It may already exist.")
```

2. TensorFlow: Katib TFJob Experiment

The TFJob definition for this example is based on the tensorflow operator notebook shown earlier. For our experiment, we focused on the learning rate, batch-size and optimizer. The following YAML file describes an Experiment object:

```
%%writefile $TF_EXPERIMENT_FILE
apiVersion: "kubeflow.org/v1alpha3"
```

```yaml
kind: Experiment
metadata:
  namespace: admin
  name: airline
spec:
  parallelTrialCount: 3
  maxTrialCount: 30
  maxFailedTrialCount: 3
  objective:
    type: maximize
    goal: 0.99
    objectiveMetricName: accuracy
  algorithm:
    algorithmName: random
  metricsCollectorSpec:
    kind: StdOut
  parameters:
    - name: learning_rate
      parameterType: double
      feasibleSpace:
        min: "0.01"
        max: "0.1"
    - name: batch_size
      parameterType: int
      feasibleSpace:
        min: "50"
        max: "100"
    - name: optimizer
      parameterType: categorical
      feasibleSpace:
        list:
          - rmsprop
          - adam
  trialTemplate:
    primaryContainerName: tensorflow
    trialParameters:
      - name: learningRate
        description: Learning rate for the training model
        reference: learning_rate
```

```
      - name: batchSize
        description: Batch Size
        reference: batch_size
      - name: optimizer
        description: Training model optimizer (sdg, adam)
        reference: optimizer
    trialSpec:
      apiVersion: "kubeflow.org/v1"
      kind: TFJob
      spec:
        tfReplicaSpecs:
          Worker:
            replicas: 1
            restartPolicy: OnFailure
            template:
              metadata:
                annotations:
                  sidecar.istio.io/inject: "false"
              spec:
                containers:
                  - name: tensorflow
                    image: mavencodevv/tfjob_airline:v.0.3
                    command:
                      - "python"
                      - "/tfjobairline.py"
                      -
"--batch_size=${trialParameters.batchSize}"
                      -
"--learning_rate=${trialParameters.learningRate}"
                      -
"--optimizer=${trialParameters.optimizer}"
```

3. Run and Monitor Experiments:

You can either execute these commands on your local machine with kubectl or on the notebook server:

```
%%capture kubectl_output --no-stderr
! kubectl apply -f $TF_EXPERIMENT_FILE
```

The cell magic grabs the output of the kubectl command and stores it in an object named kubectl_output. From there we can use the utility function we defined earlier:

```
EXPERIMENT = get_resource(kubectl_output)
```

4. See experiment status:

```
! kubectl describe $EXPERIMENT
```

5. Get the list of created experiments:

```
! kubectl get experiments
```

6. Get the list of created trials:

```
! kubectl get trials
```

7. After the experiment is completed, use describe to get the best trial results:

```
! kubectl logs $EXPERIMENT
```

8. Delete Katib job to free up resources:

```
! kubectl delete -f $TF_EXPERIMENT_FILE
```

9. Check to see if the pod is still up and running:

```
! kubectl -n admin logs -f airline
```

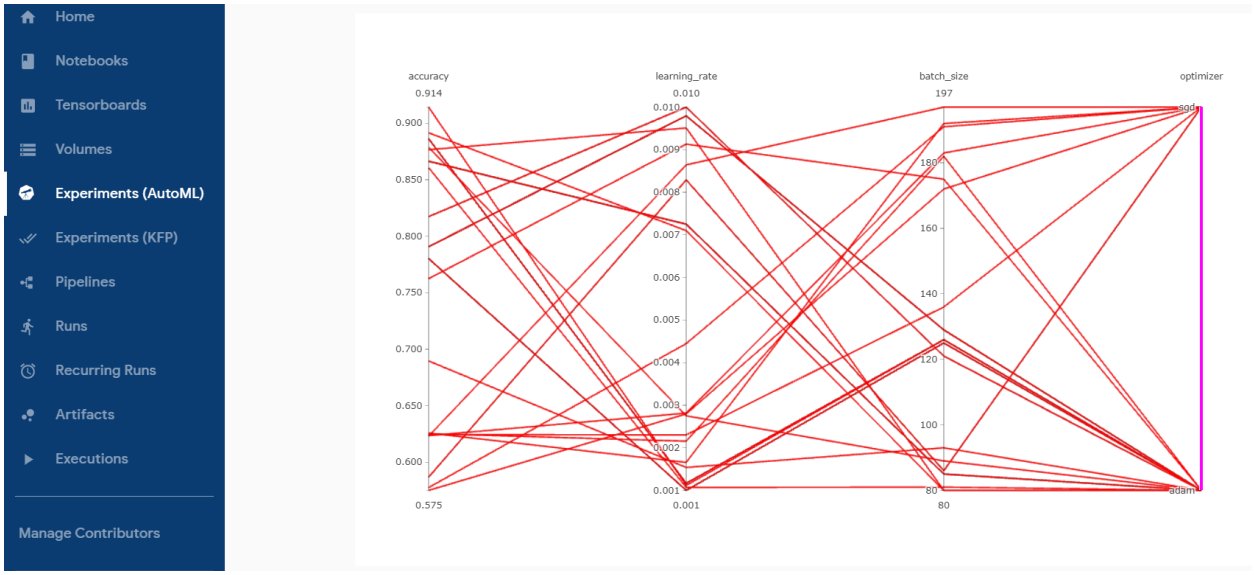

Figure 8: Result of the Katib Experiment

Model Explainability with LIME

Model explainability is the process of explaining and interpreting machine learning and deep learning models. It helps to better understand and interpret the model's behavior and to identify what features the model considers most important. For this project, we used LIME to explain our model.

LIME is an acronym for Local Interpretable Model-agnostic Explanations. It is used to explain predictions of your machine learning model. The process is described below:

1. Install LIME:

```
pip install lime
```

2. Get predictions:

```
def prob(df):
    print(df.shape)
    y_pred=keras_model.predict(df).reshape(-1, 1)
    #y_pred =(y_pred>0.5)

print(np.array(list(zip(1-y_pred.reshape(df.shape[0]),y_pred.re
shape(df.shape[0])))))
```

3. Indicate Categorical Features:

```
feature_names = ['Gender', 'Customer Type', 'Age', 'Type of
Travel',
       'Class', 'Flight Distance', 'Seat comfort',
       'Departure/Arrival time convenient', 'Food and drink',
'Gate location',
       'Inflight wifi service', 'Inflight entertainment',
'Online support',
       'Ease of Online booking', 'On-board service', 'Leg room
service',
       'Baggage handling', 'Checkin service', 'Cleanliness',
'Online boarding',
```

```
       'Total_delay']

categorical_names = ['Gender', 'Customer Type', 'Type of
Travel', 'Class', 'Seat comfort', 'Departure/Arrival time
convenient', 'Food and drink', 'Gate location', 'Inflight wifi
service', 'Inflight entertainment', 'Online support', 'Ease of
Online booking', 'On-board service', 'Leg room service',
'Baggage handling', 'Checkin service', 'Cleanliness', 'Online
boarding' ]
categorical_features = [0, 1, 3, 4,6, 7, 8, 9, 10, 11, 12, 13,
14, 15, 16, 17, 18, 19]
```

4. Import necessary Libraries:

```
import lime
import lime.lime_tabular
```

5. Create the explainer:

```
explainer =
lime.lime_tabular.LimeTabularExplainer(X[list(X.columns)].astyp
e(int).values,
mode='classification',training_labels=df['satisfaction'],featur
e_names=list(X.columns), class_names=['Dissatisfied',
'Satisfied'], categorical_features=categorical_features,
categorical_names=categorical_names)
```

6. Explain an instance:

```
i = 19
exp =
explainer.explain_instance(X.loc[i,X.columns].astype(int).value
s, prob, num_features=10)
```

7. Show explanation in notebook:

```
i = 19
exp =
explainer.explain_instance(X.loc[i,X.columns].astype(int).value
s, prob, num_features=10)
```

8. Generate a bar chart of feature contribution for this instance:

```
import matplotlib.pyplot as plt

with plt.style.context("ggplot"):
    exp.as_pyplot_figure()
```

Sample Explanation Output:

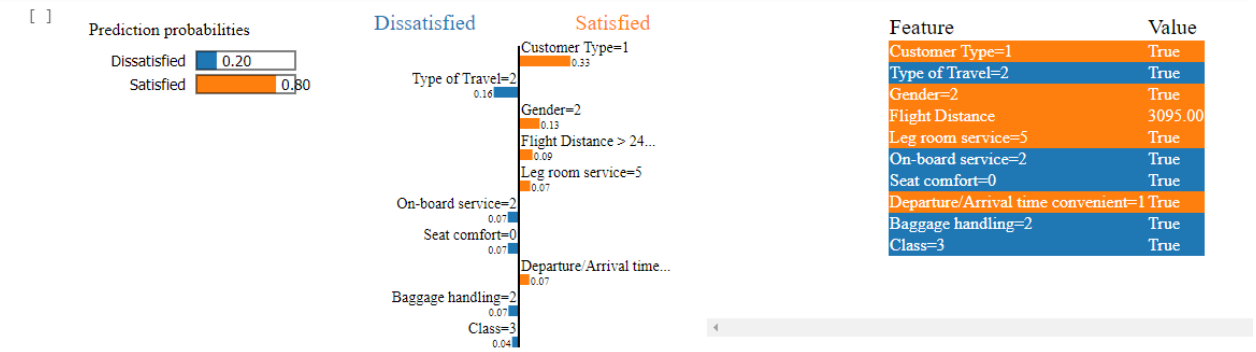

Figure 9: Sample Lime Explanation Output

The row we are explaining is displayed on the right side, in a table format with 2 columns; Feature and Values. Only the features used in the explanation (in this case, the 10 most important features) are displayed. The value column displays the original value for each feature at that instance. The explanations are based not only on features, but on feature-value pairs. For example, we are saying that Customer Type=1 is indicative of a satisfied customer (Recall that in our EDA, loyal customers (Customer Type 1) were more satisfied than Customer Type 2 (disloyal customers)).

Attributes in orange support satisfied customers and those in blue support dissatisfied customers. Flight Distance > 2497 supports dissatisfied customers, that is, it has a negative impact towards predicting the instance as 'satisfied'. Float point numbers on the horizontal bars represent the relative importance of each feature. This is represented in the chart below:

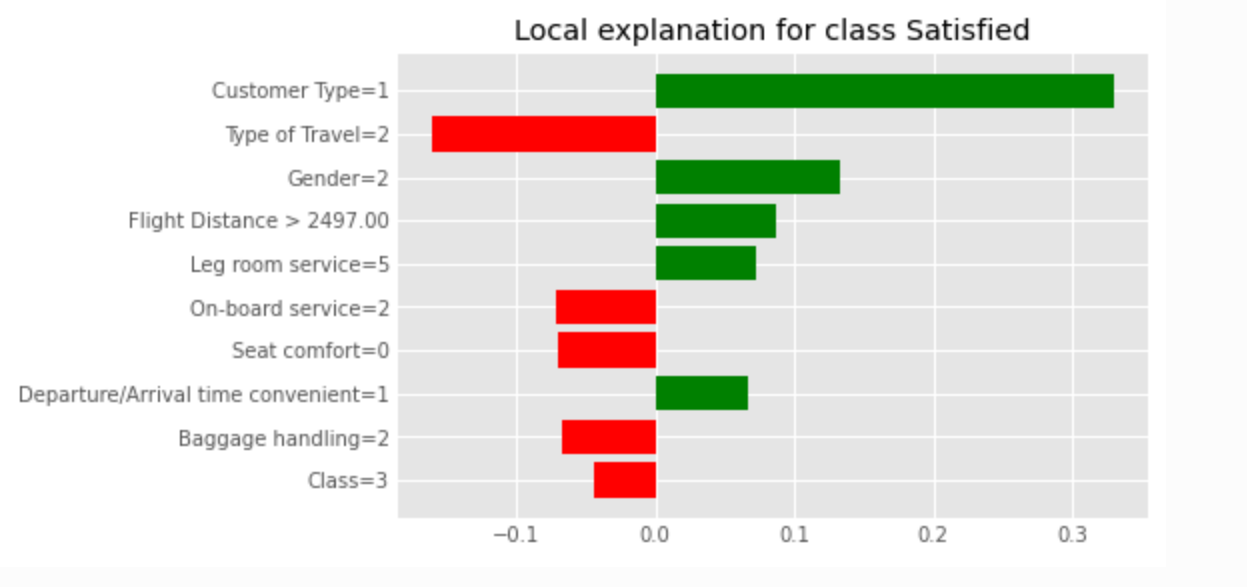

Figure 10: Barchart showing Feature Contributions for a Prediction

We created explanations for a few more instances, and our observation was that the most important features are: Customer Type, Type of Travel, Gender, Seat Comfort, and Departure/Arrival Time Convenient.

Model deployment using KfServing

Kubeflow also provides an umbrella implementation; KFServing, which generalizes the model inference concerns of autoscaling, networking, health checking, and server configuration. One of the primary tenancies of KFServing is extending serverless application development to model serving. Serverless helps build and run applications and services without provisioning, scaling, or managing any servers. Therefore, model serving puts a trained model behind a service that can handle prediction requests.

To efficiently serve our model, we built a Kubeflow pipeline for the end-to-end process of training to serving our model. The creation process is outlined below:

1. Check to see if kfp is installed

```
! pip3 show kfp
```

2. Configure model storage credentials: In order for KFServing to access MinIO or your cloud storage of choice, the credentials must be added to the default service account.

```
%%writefile minio_secret.yaml
apiVersion: v1
kind: Secret
metadata:
  name: minio-s3-secret
  annotations:
     serving.kubeflow.org/s3-endpoint:
minio-service.kubeflow:9000
     serving.kubeflow.org/s3-usehttps: "0" # Default: 1. Must
be 0 when testing with MinIO!
type: Opaque
data:
  AWS_ACCESS_KEY_ID: bWluaW8=
  AWS_SECRET_ACCESS_KEY: bWluaW8xMjM=
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: default
secrets:
  - name: minio-s3-secret
```

```
! kubectl apply -f minio_secret.yaml
```

3. Configure access MinIO: First, we configure credentials for the MinIO command-line client (mc). We then use it to create a bucket, upload the dataset to it, and set an access policy so that the pipeline can download it from MinIO.

```
# download MinIO client

! wget https://dl.min.io/client/mc/release/linux-amd64/mc
! chmod +x mc
! ./mc --help
```

4. Connect to the MinIO server

```
! ./mc alias set minio http://minio-service.kubeflow:9000 minio
minio123
```

5. Create a bucket to store your data and export your model to Minio

```
! ./mc mb minio/airlinecust1
```

6. Upload the dataset to your bucket in Minio.

```
! tar --dereference -czf datasets.tar.gz ./datasets
! ./mc cp datasets.tar.gz minio/airlinecust1/datasets.tar.gz
! ./mc policy set download minio/airlinecust1
```

7. Minio Server URL and Credentials

```
MINIO_SERVER='minio-service.kubeflow:9000'
MINIO_ACCESS_KEY='minio'
```

```
MINIO_SECRET_KEY='minio123'
```

Implementing Kubeflow Pipelines Components

8. Import the necessary libraries and assign the namespace

```
from typing import NamedTuple
import kfp
import kfp.components as components
import kfp.dsl as dsl
from kfp.components import InputPath, OutputPath
#helps define the input & output between the components
NAMESPACE = 'sooter'
```

9. Component 1: Download the Data Set

```
def download_dataset(minio_server: str, data_dir:
OutputPath(str)):
    """Download the data set to the KFP volume to share it
among all steps"""
    import urllib.request
    import tarfile
    import os
    import subprocess

    if not os.path.exists(data_dir):
        os.makedirs(data_dir)

    url = f'http://{minio_server}/airlinecust1/datasets.tar.gz'
    print(url)
    stream = urllib.request.urlopen(url)
    print('done downloading')
    tar = tarfile.open(fileobj=stream, mode="r|gz")
    tar.extractall(path=data_dir)
```

```
    print('done extracting')

    subprocess.call(["ls", "-dlha", data_dir])
```

10. Component 2: Preprocess Dataset

```
def preprocess(data_dir: InputPath(str), clean_data_dir:
OutputPath(str)):

    import numpy as np
    import sys, subprocess;
    subprocess.run([sys.executable, '-m', 'pip', 'install',
'pandas'])
    subprocess.run([sys.executable, '-m', 'pip', 'install',
'scikit-learn'])
    from sklearn.model_selection import KFold
    from sklearn.model_selection import train_test_split  #
splitting the data
    import pandas as pd
    import pickle
    import os

    # Get data

    df =
pd.read_csv(f"{data_dir}/datasets/Invistico_Airline.csv")

    #print(data)

    #drop rows with missing values
    df.dropna(inplace=True)
    #new column total delay
    df['total_delay'] = df['Departure Delay in Minutes'] +
df['Arrival Delay in Minutes']
```

```
    #drop 'Departure Delay in Minutes',and 'Arrival Delay in
Minutes'
    df.drop(columns=['Departure Delay in Minutes','Arrival
Delay in Minutes'], inplace=True)

        #satisfied and dissatisfied in number
    satisfaction_map = {"satisfied": 1,"dissatisfied": 0 }
    df['satisfaction']  =
df['satisfaction'].map(satisfaction_map)

    #Male and Female in number
    Gender_map = {"Male": 1,"Female": 2 }
    df['Gender']  = df['Gender'].map(Gender_map)

    #Loyal and disloyal in number
    Customer_Type_map = {"Loyal Customer": 1,"disloyal
Customer": 0 }
    df['Customer Type']  = df['Customer
Type'].map(Customer_Type_map)

    #Business travel and Business travel in number
    Type_of_Travel_map = {"Business travel": 1,"Personal
Travel": 2 }
    df['Type of Travel']  = df['Type of
Travel'].map(Type_of_Travel_map)

    #Business and Eco and Eco plus in number
    Class_map = {"Business": 1,"Eco": 3, "Eco Plus": 2 }
    df['Class']  = df['Class'].map(Class_map)

    cols = ['Flight Distance', 'total_delay', 'Checkin
service', 'On-board service']

    Q1 = df[cols].quantile(0.25)
    Q3 = df[cols].quantile(0.75)
    IQR = Q3 - Q1

    df = df[~((df[cols] < (Q1 - 1.5 * IQR)) |(df[cols] > (Q3 +
```

```
1.5 * IQR))).any(axis=1)]

    #Split dataset

    X = df.drop('satisfaction',axis=1)
    y = df['satisfaction']
    X_train,X_test, y_train, y_test = train_test_split(X,y,
test_size = 0.3, random_state = 111)


    data = {"X_train": X_train,"X_test": X_test, "y_train":
y_train,"y_test": y_test}

    os.makedirs(clean_data_dir, exist_ok=True)

    with open(os.path.join(clean_data_dir,'clean_data.pickle'),
'wb') as f:
        pickle.dump(data, f)

    print(f"clean_data.pickle {clean_data_dir}")

    print(os.listdir(clean_data_dir))

    print("Preprocessing Done")
```

11. Component 3: Training the data with Tensorflow Model

```
def train_model(clean_data_dir: InputPath(str), model_dir:
OutputPath(str)):

    # Install all the dependencies inside the function
    import numpy as np
    import pickle
    import os
    import sys, subprocess;
    subprocess.run([sys.executable, '-m', 'pip', 'install',
'tensorflow==0.24.2'])
```

```
    import pandas as pd
    # import libraries for training

    from numpy.random import seed

    import tensorflow as tf
    tf.random.set_seed(221)
    from tensorflow import keras
    from tensorflow.keras.models import Sequential
    from tensorflow.keras.layers import Dense, Dropout,
BatchNormalization
    from tensorflow.keras.optimizers import SGD, Adam, RMSprop

    #load the preprocessed data

    print(clean_data_dir)
    with open(os.path.join(clean_data_dir,'clean_data.pickle'),
'rb') as f:
        data = pickle.load(f)

    print(data)

    X_train = data['X_train']
    y_train = data['y_train']

    seed(1)
    model = Sequential()
    model.add(Dense(100, activation='relu', input_dim=21))
    model.add(BatchNormalization())
    model.add(Dense(40, activation='relu'))
    model.add(Dropout(0.2))
    model.add(Dense(1, activation='sigmoid'))

    model.summary()
    #opt = args.optimizer
    model.compile(loss='binary_crossentropy',
                optimizer='adam',
                metrics=['accuracy'])
```

```
    # Fit the model to the training data
    model.fit(X_train, y_train, epochs=30)

    #Save the model to the designated
    os.makedirs(model_dir, exist_ok=True)

    #with open(os.path.join(model_dir,'model.pickle'), 'wb') as
f:
        #pickle.dump(model, f)


    model.save(model_dir)

    print(f"Model saved {model_dir}")

    print(os.listdir(model_dir))
```

12. Component 4: Evaluate Model

```
def prediction(clean_data_dir: InputPath(str), model_dir:
InputPath(str), metrics_path: OutputPath(str)
) -> NamedTuple("EvaluationOutput", [("mlpipeline_metrics",
"Metrics")]):
    import pickle
    import os
    import sys, subprocess;
    import numpy as np

    import json
    import tensorflow as tf
    import tensorflow_datasets as tfds
    from collections import namedtuple


    #Evaluate the model and print the results
    print(model_dir)
```

```
    model = tf.keras.models.load_model(model_dir)

    print(model)

    print(clean_data_dir)
    with open(os.path.join(clean_data_dir,'clean_data.pickle'),
'rb') as f:
        data = pickle.load(f)
    print(data)

    X_test = data['X_test']
    y_test = data['y_test']
    X_train = data['X_train']
    y_train = data['y_train']


    (loss, accuracy) = model.evaluate(X_test,y_test, verbose=0)

    metrics = {
        "metrics": [
            {"name": "loss", "numberValue": str(loss),
"format": "PERCENTAGE"},
            {"name": "accuracy", "numberValue": str(accuracy),
"format": "PERCENTAGE"},
        ]
    }

    with open(metrics_path, "w") as f:
        json.dump(metrics, f)

    out_tuple = namedtuple("EvaluationOutput",
["mlpipeline_metrics"])

    return out_tuple(json.dumps(metrics))
```

13. Component 5: Export the Model

```
def export_model(
    model_dir: InputPath(str),
    metrics: InputPath(str),
    export_bucket: str,
    model_name: str,
    model_version: int,
    minio_server: str,
    minio_access_key: str,
    minio_secret_key: str,
):
    import os
    import boto3
    from botocore.client import Config

    s3 = boto3.client(
        "s3",
        endpoint_url=f'http://{minio_server}',
        aws_access_key_id=minio_access_key,
        aws_secret_access_key=minio_secret_key,
        config=Config(signature_version="s3v4"),
    )

    # Create export bucket if it does not yet exist
    response = s3.list_buckets()
    export_bucket_exists = False

    for bucket in response["Buckets"]:
        if bucket["Name"] == export_bucket:
            export_bucket_exists = True

    if not export_bucket_exists:
        s3.create_bucket(ACL="public-read-write",
Bucket=export_bucket)

    # Save model files to S3
    for root, dirs, files in os.walk(model_dir):
        for filename in files:
            local_path = os.path.join(root, filename)
            s3_path = os.path.relpath(local_path, model_dir)
```

```
            s3.upload_file(
                local_path,
                export_bucket,
                f"{model_name}/{model_version}/{s3_path}",
                ExtraArgs={"ACL": "public-read"},
            )

    response = s3.list_objects(Bucket=export_bucket)
    print(f"All objects in {export_bucket}:")
    for file in response["Contents"]:
        print("{}/{}".format(export_bucket, file["Key"]))
```

14. Component: Serve Model

Kubeflow Pipelines comes with a pre-defined KFServing component which can be imported from GitHub repo and reused across the pipelines without the need to define it every time. We included a copy with the tutorial to make it work in an air-gapped environment. Here's what the import looks like:

```
kfserving =
components.load_component_from_file("kfserving-component.yaml")
```

15. Combine the Components into a Pipeline

```
def train_model_pipeline(
    data_dir: str,
    clean_data_dir: str,
    model_dir: str,
    export_bucket: str,
    model_name: str,
    model_version: int,
    minio_server: str,
```

```
    minio_access_key: str,
    minio_secret_key: str,
):
    # For GPU support, please add the "-gpu" suffix to the base
image
    BASE_IMAGE = "mavencodev/minio:v.0.1"

    downloadOp = components.func_to_container_op(
        download_dataset, base_image=BASE_IMAGE
    )(minio_server)

    preprocessOp = components.func_to_container_op(preprocess,
base_image=BASE_IMAGE)(
        downloadOp.output
    )

    trainOp = components.func_to_container_op(train_model,
base_image=BASE_IMAGE)(
        preprocessOp.output
    )

    predictionOp = components.func_to_container_op(prediction,
base_image=BASE_IMAGE)(
        preprocessOp.output, trainOp.output
    )

    exportOp = components.func_to_container_op(export_model,
base_image=BASE_IMAGE)(
        trainOp.output, predictionOp.output, export_bucket,
        model_name, model_version, minio_server,
minio_access_key, minio_secret_key
    )

    kfservingOp = kfserving(
        action="apply",
        default_model_uri=f"s3://{export_bucket}/{model_name}",
        model_name="airlinecust1",
        namespace= NAMESPACE,
        framework="tensorflow",
```

```
    )
    kfservingOp.after(exportOp)
```

16. Model Serving Pipeline

```
def op_transformer(op):
    op.add_pod_annotation(name="sidecar.istio.io/inject",
value="false")
    return op


@dsl.pipeline(
    name="Serving Customer Satisfaction Prediction model",
    description="A KFServing pipeline",
)
def satisfaction_pipeline(
    model_dir: str = "/train/model",
    data_dir: str = "/train/data",
    clean_data_dir: str= "/train/data",
    export_bucket: str = "airlinecust1",
    model_name: str = "airlinecust1",
    model_version: int = 1,
):
    MINIO_SERVER='minio-service.kubeflow:9000'
    MINIO_ACCESS_KEY='minio'
    MINIO_SECRET_KEY='minio123'


    train_model_pipeline(
        data_dir=data_dir,
        clean_data_dir=clean_data_dir,
        model_dir=model_dir,
        export_bucket=export_bucket,
        model_name=model_name,
        model_version=model_version,
        minio_server=MINIO_SERVER,
```

```
        minio_access_key=MINIO_ACCESS_KEY,
        minio_secret_key=MINIO_SECRET_KEY,
    )

    dsl.get_pipeline_conf().add_op_transformer(op_transformer)
```

17. Compiling pipeline as a YAML file

```
pipeline_func = satisfaction_pipeline
run_name = pipeline_func.__name__ + " run"
experiment_name = "End-to-End-Demo"

kfp.compiler.Compiler().compile(pipeline_func,
'airline16.yaml')
```

Upload the generated YAML file to create a pipeline in Kubeflow UI to generate a pipeline run

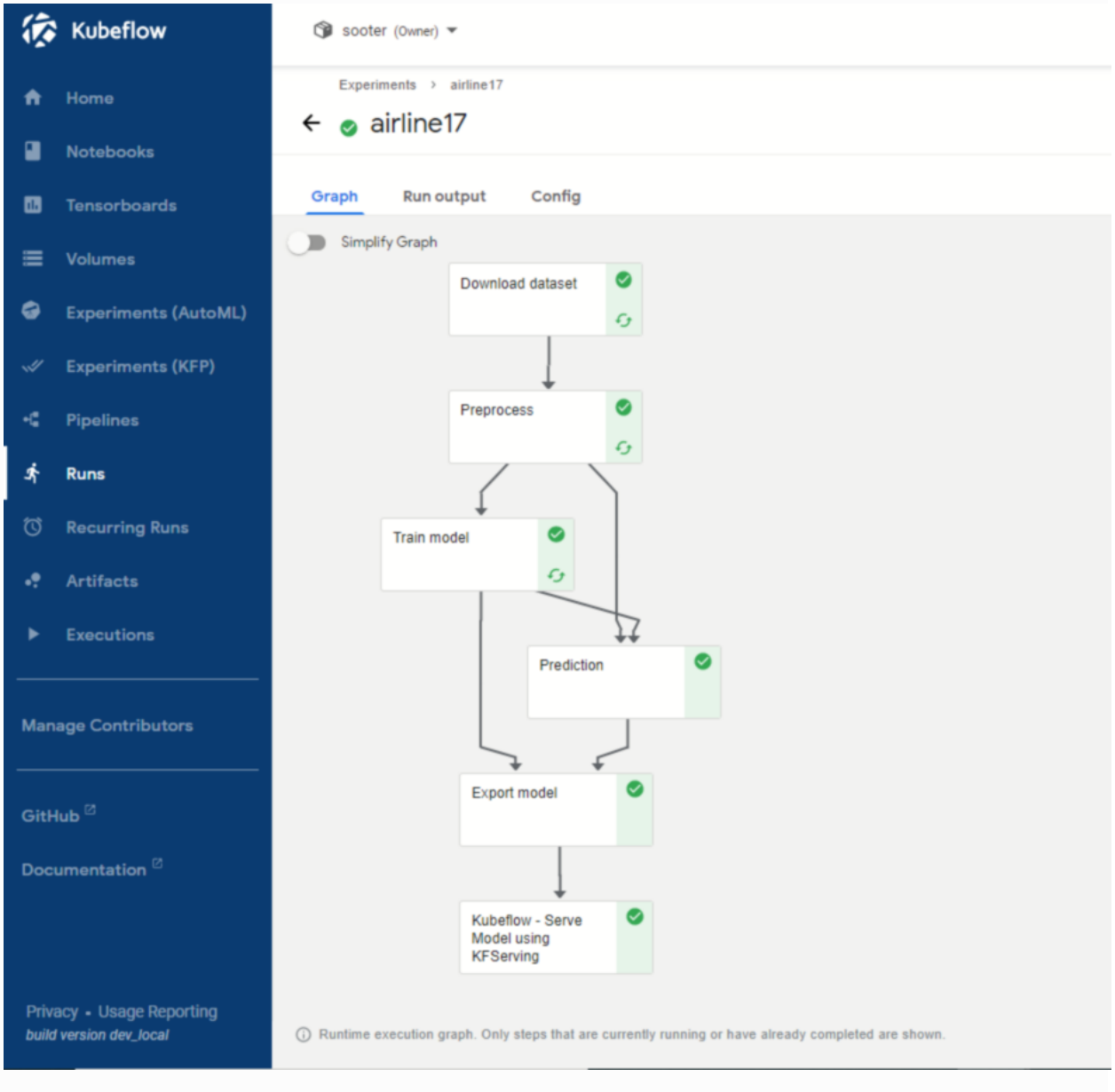

Figure 11: Model Serving Pipeline

Conclusion

We have analyzed data from Invistico Airlines and built a machine learning model to predict customer satisfaction. To improve our model's performance, we performed hyperparameter tuning using Katib, getting an overall accuracy of over 90% of the

variation. We also used LIME explainers to explain our model and identify the most important features in our model's prediction such as Customer Type, Type of Travel, Gender, Seat Comfort and Flight Distance.