

电子科技大学信息与软件工程学院

项目报告

课程名称 大数据分析 & 智能计算

理论教师 罗瑜

实验教师 罗瑜

学生信息：

序号	学号	姓名
1	*****	*****
2	*****	*****
3	*****	*****
4	*****	*****
5	*****	*****
6	*****	*****

电子科技大学

项目报告

指导教师： 罗瑜

地点：二教 205

一、项目名称：航空公司延误和取消分析项目

二、项目时间：*****

三、项目原理

Apache Spark 是一个开源的分布式计算框架，专为大规模数据处理而设计，是专为大规模数据处理而设计的快速通用的计算引擎。它具有以下特点：

- 快速：Spark 可以将中间结果保存在内存中，避免了频繁的磁盘 I/O 操作，从而提高了计算效率。
- 通用：Spark 支持多种编程语言，包括 Python、Java、Scala 和 SQL。它还支持多种数据处理场景，包括批处理、迭代算法、交互式查询和流处理。
- 易于使用：Spark 提供了丰富的 API，可以方便地进行数据预处理、分析和建模。它还支持 Jupyter Notebook 等工具，可以进行交互式数据分析和可视化。

Spark 的核心是弹性分布式数据集 (RDD)，它是一个不可变、可分区、可并行操作的集合，可以存储在内存或磁盘上。这种设计使其能够高效地处理海量数据，并通过并行计算和分布式存储来加速数据处理过程。

Spark 的这些特点使其成为大数据分析的强大工具，能够高效地处理海量数据，并支持多种数据处理场景，例如批处理、迭代算法、交互式查询和流处理等。因此，选用 Spark 作为本项目数据分析、处理的主要框架工具。

PySpark 是 Spark 提供的 Python 接口，利用 PySpark，可以调用 Spark 的 API，实现航班延误数据集的数据预处理、数据分析与数据建模预测。结合 Jupyter Notebook，我们可以在 Web 界面输入 Python 命令后立刻看到结果，并将数据分析的过程和运行后的命令与结果存储成笔记本，方便后续查阅、分析和复现。

Python 是数据分析最常用的语言之一，而 Apache Spark 是一个开源的强大的分布式查询和处理引擎。本实验要求基于 Python 语言进行 Spark Application 编程，完成数据获取、处理、数据分析及可视化方面常用的数据分析方法与技巧，让学生掌握使用 PySpark 来分析数据。

四、 项目内容

本项目旨在利用 PySpark 框架和 Python 语言，对航班延误和取消数据进行分析和预测。

首先在本机安装并测试 PySpark (在前序实验中已完成 hadoop 伪分布式配置以及 spark 的安装); 并完成项目开展前的原理学习、需求分析、模块设计工作。

随后开展项目。首先，读取航班数据集。接下来，对数据进行一系列数据分析，包括查看飞机延误时间最长的前 10 名航班、计算延误的和没有延误的航空公司的比例、分析一天中、一周中延误最严重的飞行时间、分析短途航班和长途航班的取消情况，并使用列表、图形等形式展示数据分析结果，分析相关可能原因。

为了预测未来航班的取消情况，选择合适的机器学习算法模型在训练集中进行模型训练然后，使用测试数据集评估模型性能，例如准确率、召回率、ROC 曲线、AUC 等。

最终，记录项目开展的过程，包括数据预处理、数据分析、模型建立和预测结果等。分析实验结果，并分析模型预测准确率。总结项目经验和心得体会，并提出改进建议。

五、 需求分析与设计

5.1 项目背景

航班延误和取消是航空业面临的重要问题，它不仅影响旅客的出行体验，还造成航空公司巨大的经济损失。分析航班延误和取消的原因，并预测未来航班取消情况，对于航空公司优化运营、提高服务质量和降低运营成本具有重要意义。

5.2 需求概述

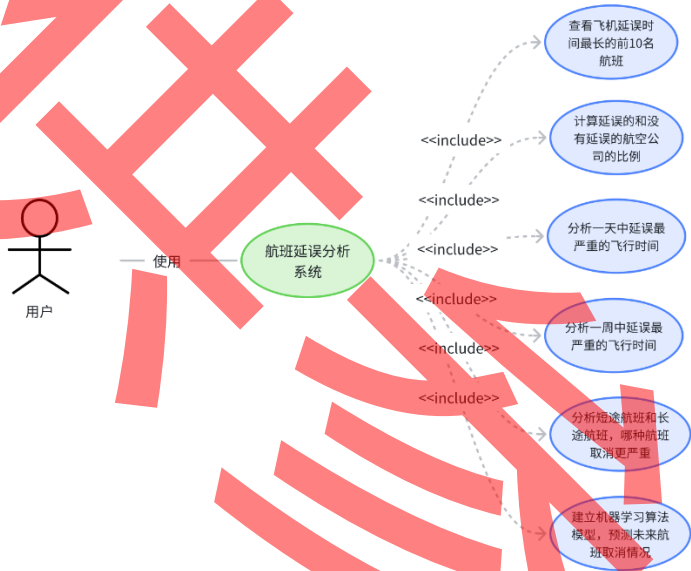
本项目旨在利用 PySpark 和 Python 语言，深入分析航班延误和取消数据，并建立机器学习模型预测未来航班取消情况。通过数据预处理、分析和可视化，分析航班延误和取消的可能原因和规律，并利用机器学习模型预测未来航班取消情况，为航空公司和旅客提供决策支持。综上，小组将完成以下任务：

需求分类	具体说明
处理	读取航班数据集，分析数据标签
分析	① 查看飞机延误时间最长的前 10 名航班
	② 计算延误的和没有延误的航空公司的比例
	③ 分析一天中延误最严重的飞行时间

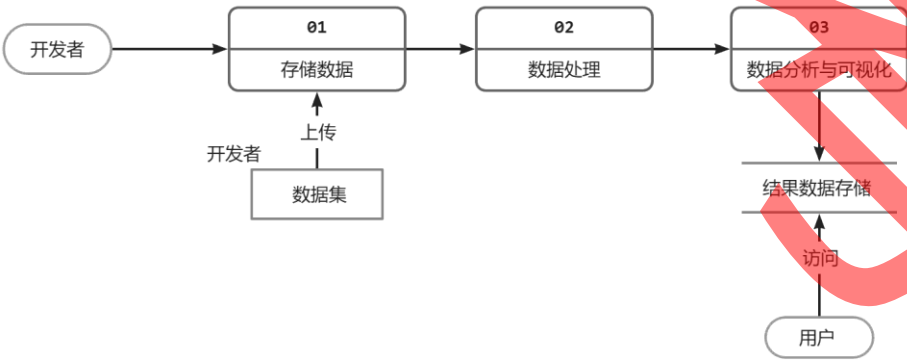
	④ 分析一周中延误最严重的飞行时间
	⑤ 分析短途航班和长途航班，哪种航班取消更严重
可视化	使用列表和图形展示上述①~⑤的数据分析结果，例如柱状图、折线图、饼图、数据表等，以便更直观地了解航班延误和取消的情况。并解释可能的原因。
预测	建立多种机器学习算法模型，预测未来航班取消情况
	结合相关指标，在测试集中评估预测模型的预测准确性

5.3 图示说明

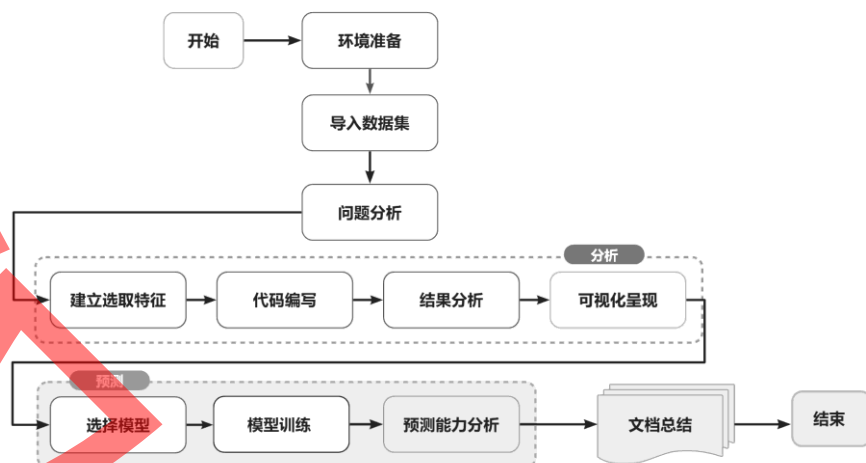
本项目的用例图如下所示——



数据流图如下所示——



本项目流程图如下所示——



5.4 人员分工

本组将项目进行拆解，每人完成其中一个或几个部分。每个人的报告反映了其本人的工作量情况，不同的人的报告内容不同，合起来为一个完整项目。

小组成员	工作量
***	<ul style="list-style-type: none"> 分析一天中延误最严重的飞行时间并可视化分析说明 使用梯度提升树(GBDT)模型对航班取消进行预测
***	<ul style="list-style-type: none"> 计算延误的和没有延误的航空公司的比例并可视化分析说明 使用随机森林模型对航班取消情况进行预测分析
***	<ul style="list-style-type: none"> 查看飞机延误时间最长的前 10 名航班并可视化分析说明 使用 SVM 支持向量机模型对航班取消进行预测
***	<ul style="list-style-type: none"> 分析一周中延误最严重的飞行时间并可视化分析说明 使用线性判别分析模型对航班取消进行预测
***	<ul style="list-style-type: none"> 分析短途航班和长途航班，哪种航班取消更严重？并完成可视化
***	<ul style="list-style-type: none"> 使用逻辑回归模型对航班取消情况进行预测分析

六、项目计划

本项目的大致计划如下——

- 1) 进行需求分析，绘制用例图等详细描述需求分析
- 2) 环境配置，安装 PySpark、Jupyter Notebook
- 3) 测试实验环境
- 4) 准备实验数据集
- 5) 针对数据集，编写代码，进行相应的数据分析
- 6) 选择合适的模型，进行模型训练，预测未来航班的取消情况
- 7) 分析数据结果，给出相应的解释和分析。同时，针对预测模型，根据相应指标分析模型预测准确率。
- 8) 总结，记录实验完整过程，完成报告撰写。

七、项目环境配置管理

7.1 操作系统

本项目主要使用 Linux 操作系统环境,具体为:基于 VMware 虚拟机的 Ubuntu 22.04 系统。

7.2 开发工具

本项目主要使用 Python 编程语言,结合 PySpark 大数据处理框架,Anaconda3 的科学计算环境,Jupyter Notebook 的交互式编程界面,以及 VSCode 代码编辑器进行开发。主要选择的开发工具如下——

- Python
- PySpark
- Anaconda3
- Jupyter Notebook
- VSCode
- VMWare

7.3 配置过程

(1) 环境准备

首先,检查确认 Spark 是否完成伪分布式配置(实验 1、2 要求的 spark 环境)。输入以下命令,登录 hadoop 用户

```
su - Hadoop
```

输入以下信息,输出 SPARK_HOME 环境变量,查看 spark 环境变量配置是否正确:

```
echo $SPARK_HOME
```

```
hadoop@jjq-ubuntu:~$ echo $SPARK_HOME
/hadoop/spark
hadoop@jjq-ubuntu:~$
```

可以看到成功输出 Spark 的环境变量指向 spark 的安装地址,Spark 配置正确

(2) 安装 Anaconda

在 hadoop 目录下,新建 anaconda3 文件夹

```
mkdir anaconda3
cd /hadoop/anaconda3
```

在终端使用 wget 命令,下载安装脚本

```
$ wget https://repo.anaconda.com/archive/Anaconda3-2022.10-Linux-x86_64.sh
```

```
-Linux-x86_64.sh
--2024-11-23 18:18:40-- https://repo.anaconda.com/archive/Anaconda3-2022.10-Linux-x86_64.sh
正在解析主机 repo.anaconda.com (repo.anaconda.com)... 104.16.191.158, 104.16.32.241, 2606:4700::6810:20f1, ...
正在连接 repo.anaconda.com (repo.anaconda.com)|104.16.191.158|:443... 已连接。
已发出 HTTP 请求，正在等待回应... 200 OK
长度: 773428196 (738M) [application/x-sh]
正在保存至: 'Anaconda3-2022.10-Linux-x86_64.sh'

Anaconda3-2022.10-L 100%[=====] 737.60M 8.74MB/s 用时 88s

2024-11-23 18:20:08 (8.43 MB/s) - 已保存 'Anaconda3-2022.10-Linux-x86_64.sh' [773428196/773428196]
```

下载完成后，使用如下指令安装

```
bash Anaconda3-2022.10-Linux-x86_64.sh
```

安装过程截图如下：

```
jjq@jjq-ubuntu: /hadoop/anaconda3
zope          pkgs/main/linux-64::zope-1.0-py39h06a4308_1
zope.interface pkgs/main/linux-64::zope.interface-5.4.0-py39h7f8727e_0
zstd          pkgs/main/linux-64::zstd-1.5.2-ha4553b6_0

Preparing transaction: done
Executing transaction: |

Installed package of scikit-learn can be accelerated using scikit-learn-intelex.
More details are available here: https://intel.github.io/scikit-learn-intelex

For example:

$ conda install scikit-learn-intelex
$ python -m sklearn my_application.py

done
Installation finished.
WARNING:
You currently have a PYTHONPATH environment variable set. This may cause
unexpected behavior when running the Python interpreter in Anaconda3.
For best results, please verify that your PYTHONPATH only points to
directories of packages that are compatible with the Python interpreter
in Anaconda3: /home/jjq/anaconda3
jjq@jjq-ubuntu: /hadoop/anaconda3$
```

接着，配置 PySpark driver，用来和 spark 本身进行交互。

```
sudo gedit /hadoop/.bash_profile
```

通过上述命令打开 spark 的环境配置文件，配置项编写如下——

```
export PATH=$PATH:$SPARK_HOME/bin: /hadoop/anaconda3/anaconda/bin
export ANACONDA_PATH=/hadoop/anaconda3/anaconda
export PYSARK_DRIVER_PYTHON=$ANACONDA_PATH/bin/ipython
export PYSARK_PYTHON=$ANACONDA_PATH/bin/python
export PYSARK_DRIVER_PYTHON=jupyter
export PYSARK_DRIVER_PYTHON_OPTS='notebook'
```



```
.bash_profile
1 #HADOOP START
2 export JAVA_HOME=/hadoop/jdk1.8.0_391
3 export HADOOP_HOME=/hadoop/hadoop-3.3.6
4 export PATH=$PATH:$HADOOP_HOME/bin:$HADOOP_HOME/sbin:$JAVA_HOME/bin
5 export HADOOP_MAPRED_HOME=$HADOOP_HOME
6 export HADOOP_COMMON_HOME=$HADOOP_HOME
7 export HADOOP_HDFS_HOME=$HADOOP_HOME
8 export YARN_HOME=$HADOOP_HOME
9 export HADOOP_COMMON_LIB_NATIVE_DIR=$HADOOP_HOME/lib/native
10 export HADOOP_OPTS="-Djava.library.path=$HADOOP_HOME/lib"
11 #HADOOP END
12
13 # SPARK START
14 export SPARK_HOME=/hadoop/spark
15 export PATH=$PATH:$SPARK_HOME/bin
16 # check the version of py4j
17 export PYTHONPATH=$SPARK_HOME/python:$SPARK_HOME/python/lib/py4j-0.10.9.7-src.zip:$PYTHONPATH
18 # if run pyspark on Ubuntu 20.04 LTS, it shows the error "python: command not found"
19 export PYSPARK_PYTHON=python3
20 # SPARK END
21 export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
22 export HDFS_CONF_DIR=$HADOOP_HOME/etc/hadoop
23 export YARN_CONF_DIR=$HADOOP_HOME/etc/hadoop
24
25 export PATH=$PATH:$SPARK_HOME/bin:/hadoop/anaconda3/anaconda/bin
26 export ANACONDA_PATH=/hadoop/anaconda3/anaconda
27 export PYSPARK_DRIVER_PYTHON=$ANACONDA_PATH/bin/ipython
28 export PYSPARK_PYTHON=$ANACONDA_PATH/bin/python
29 export PYSPARK_DRIVER_PYTHON=jupyter
30 export PYSPARK_DRIVER_PYTHON_OPTS='notebook'
```

保存关闭后，执行以下命令使得环境变量生效：

```
source /hadoop/.bash_profile
```

随后，输入以下指令查看 **anaconda** 的版本信息，以验证安装是否成功。

```
conda --version
```

```
jjq@jjq-ubuntu:/hadoop/anaconda3$ conda --version
conda 22.9.0
jjq@jjq-ubuntu:/hadoop/anaconda3$
```

可以看到，成功安装了 22.9.0 版本的 **anaconda**。安装配置正确

(3) 安装并配置 Jupyter

Jupyter Notebook 是一个交互式计算工具，用于编写代码、分析数据和展示结果，支持多种编程语言。

在 **anaconda** 安装时，已经预先装载了 Jupyter。使用如下命令可以查看进行验证——

```
conda list | grep jupyter
```

```
jjq@jjq-ubuntu:/hadoop/anaconda3$ conda list | grep jupyter
jupyter                    1.0.0                    py39h06a4308_8
jupyter_client              7.3.4                    py39h06a4308_0
jupyter_console             6.4.3                    pyhd3eb1b0_0
jupyter_core                4.11.1                   py39h06a4308_0
jupyter_server              1.18.1                   py39h06a4308_0
jupyterlab                  3.4.4                    py39h06a4308_0
jupyterlab_pygments         0.1.2                    py_0
jupyterlab_server           2.10.3                   pyhd3eb1b0_1
jupyterlab_widgets          1.0.0                    pyhd3eb1b0_1
jjq@jjq-ubuntu:/hadoop/anaconda3$
```


(4) 尝试在 Notebook 中使用 Spark

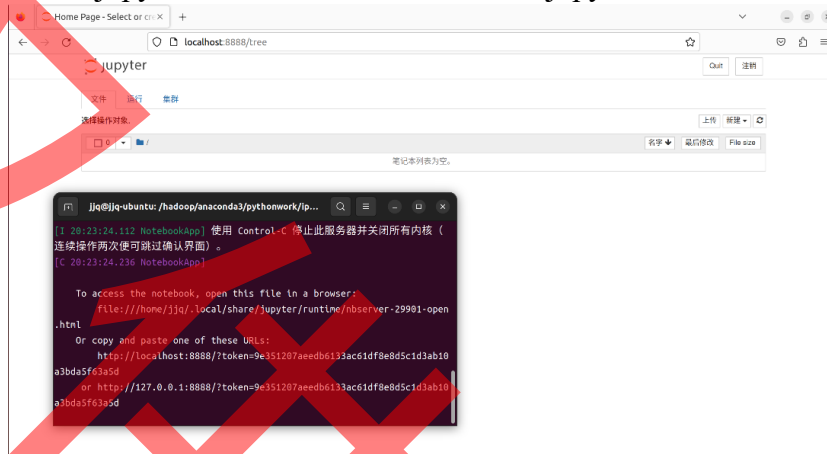
可以在“终端”程序中输入下列命令，创建并切换 ipynotebook 工作目录。

```
mkdir -p ./pythonwork/ipynotebook
cd ./pythonwork/ipynotebook
```

进入工作目录后，输入以下命令

```
pyspark
```

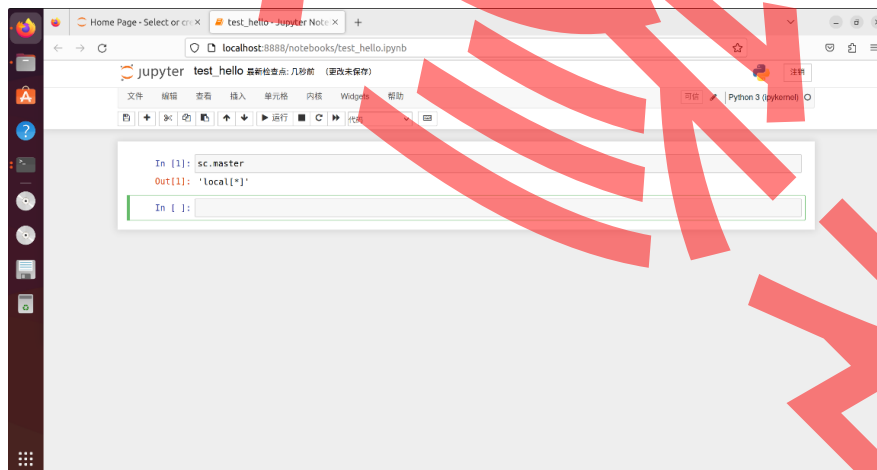
就会自动启动 jupyter，在浏览器中就可以使用 jupyter 了。如图所示——



然后从浏览器启动 Notebook 界面，默认是以 spark 的本地模式运行。例如在新建的 notebook 中，输入代码作为测试：

```
sc.master
```

点击运行，运行结果如下图所示：



输出结果为“local[*]”表明正在以 Spark 本地模式运行。配置正确。

八、项目实践过程

在项目中，我主要负责分析一天中延误最严重的飞行时间并可视化分析说明，以及用梯度提升树(GBDT)模型对航班取消进行预测两个主要任务工作。下面详细介绍实践过程（对实践结果的分析 and 图示见第九部分）。

8.1 数据读取与准备

首先，将所需的数据集拷贝至工作目录中

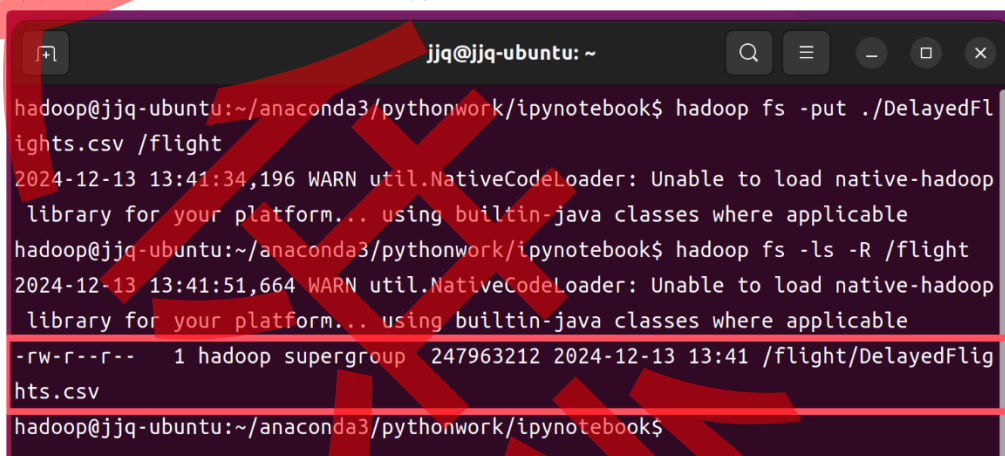
```
cp DelayedFlights.csv /hadoop/anaconda3/pythonwork/ipynotebook/
```

然后将数据上传至 hadoop 中，HDBS/flight 文件夹下

相关命令为——

```
hadoop fs -mkdir /flight
hadoop fs -put ./DelayedFlights.csv /flight
hadoop fs -ls -R /flight
```

这三条命令在 Hadoop 中创建一个 /flight 目录、上传数据集 CSV 文件到该目录，并递归显示目录中的所有文件和子目录。



```
jjq@jjq-ubuntu: ~
hadoop@jjq-ubuntu:~/anaconda3/pythonwork/ipynotebook$ hadoop fs -put ./DelayedFlights.csv /flight
2024-12-13 13:41:34,196 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
hadoop@jjq-ubuntu:~/anaconda3/pythonwork/ipynotebook$ hadoop fs -ls -R /flight
2024-12-13 13:41:51,664 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
-rw-r--r-- 1 hadoop supergroup 247963212 2024-12-13 13:41 /flight/DelayedFlights.csv
hadoop@jjq-ubuntu:~/anaconda3/pythonwork/ipynotebook$
```

从运行截图可知，数据集已经正确上传至 HDBS 中。

从数据集中读取数据。代码如下所示。

代码 1: 读取数据集代码

```
from numpy import select
from pyspark import sql
from pyspark.sql import SQLContext
from pyspark import SparkContext
import pyspark.sql.functions as F
from typing import ForwardRef
from pyspark.sql.functions import concat,concat_ws,bround
import pandas as pd
import matplotlib.pyplot as plt
import datetime
import matplotlib.dates as mdate

# 读取数据
data = spark.read\
    .option("header","true")\
    .option("inferSchema","true")\
```

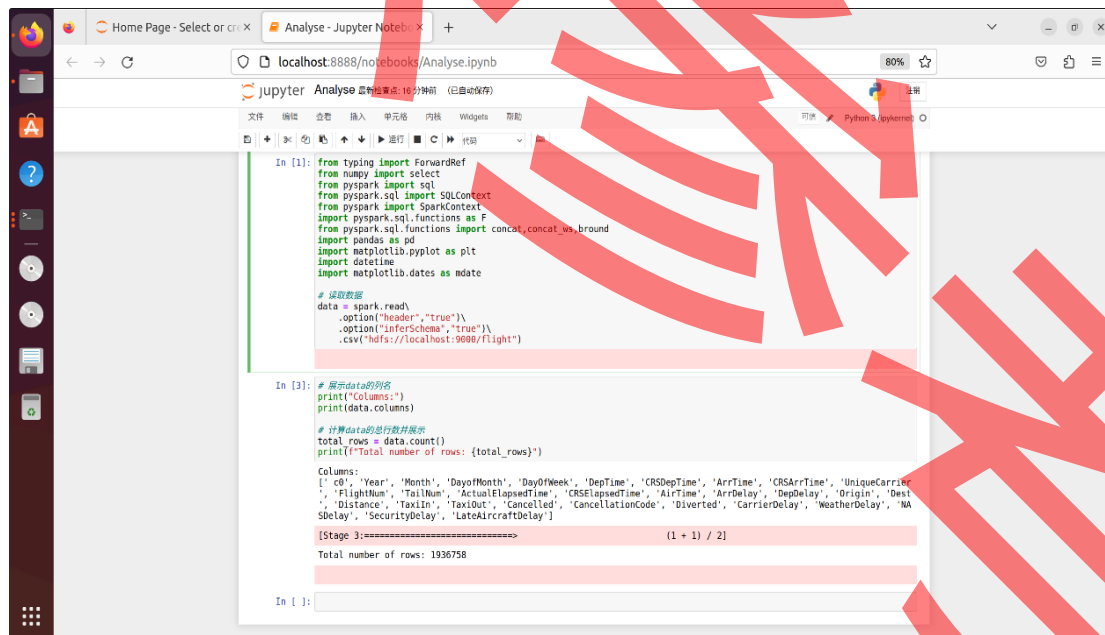
```
.csv("hdfs://localhost:9000/flight")
# 展示 data 的列名
print("Columns:")
print(data.columns)

# 计算 data 的总行数并展示
total_rows = data.count()
print(f"Total number of rows: {total_rows}")
```

这段代码使用 PySpark 进行数据集的读取和准备工作。首先导入了必要的库，包括 NumPy、pyspark.sql 模块、SparkContext 以及一些用于数据操作的 pyspark.sql.functions。

随后，通过调用 spark.read 方法，程序读取存储在 HDFS 上的数据集 CSV 文件。在读取过程中，代码指定了两个选项：header="true" 和 inferSchema="true"。header="true" 表示 CSV 文件的第一行包含列名，而 inferSchema="true" 告诉 Spark 自动推断每列的数据类型。读取操作完成后，数据被存储在一个名为 data 的 DataFrame 对象中。DataFrame 是 Spark SQL 中的关键抽象，它代表一个分布式数据集，类似于 RDBMS 中的表或 Python 的 pandas 库中的 DataFrame。在读取 HDFS 上的 CSV 文件后，打印出数据的列名，并计算数据表的总行数。

代码运行的结果如图：



```
In [1]: from typing import ForwardRef
from numpy import select
from pyspark import sql
from pyspark.sql import SQLContext
from pyspark import SparkContext
import pyspark.sql.functions as F
from pyspark.sql.functions import concat,concat_ws,bround
import pandas as pd
import matplotlib.pyplot as plt
import datetime
import matplotlib.dates as mdate

# 读取数据
data = spark.read\
    .option("header","true")\
    .option("inferSchema","true")\
    .csv("hdfs://localhost:9000/flight")

In [3]: # 展示data的列名
print("Columns:")
print(data.columns)

# 计算data的总行数并展示
total_rows = data.count()
print(f"Total number of rows: {total_rows}")

Columns:
['c0', 'Year', 'Month', 'DayOfMonth', 'DayOfWeek', 'DepTime', 'CRSDepTime', 'ArrTime', 'CRSArrTime', 'UniqueCarrier',
 'FlightNum', 'TailNum', 'ActualElapsedTime', 'CRSElapsedTime', 'AirTime', 'ArrDelay', 'DepDelay', 'Origin', 'Dest',
 'Distance', 'TaxiIn', 'TaxiOut', 'Cancelled', 'CancellationCode', 'Diverted', 'CarrierDelay', 'WeatherDelay', 'NASDelay',
 'SecurityDelay', 'LateAircraftDelay']

[Stage 3]:===== (1 + 1) / 2]
Total number of rows: 1936758

In [ ]:
```

从截图中可以看出，已成功从 HDFS 中读取了 1,936,758 条航班数据，数据量较为庞大，覆盖了多个关键维度。这些记录包含了航班的基本信息，如航班号、所属航空公司、出发和到达时间等，还涵盖了延误时间、是否取消、取消原因（如航空公司、天气、空管或安全原因）等详细信息。此外，数据中还涉及航班的实际飞行时间、计划飞行时间、起飞和降落的滑行时间等。数据价值大。

8.2 分析一天中延误最严重的飞行时间

(1) 任务总述

本任务需要分析了一天中延误最严重的飞行时间。为了具体探究，我将其细分一些子任务，具体包括：

- 分析一天中各时段延误架次情况
- 可视化呈现一天各时段延误架次的折线图变化
- 可视化呈现一天各时段延误架次占总架次的饼图比例
- 分析一天各时段延误原因统计及可视化呈现分组柱状图。

下面进行具体分析

(2) 按时间段划分，分析一天中各时段延误架次情况

为了识别一天中航班延误最严重的时段，我采用了以“小时”为单位的时间划分方法，将一天划分为 24 个独立的时间段，并计算了每个小时内航班的延误次数。在此分析中，我关注了两个核心变量：航班的起飞时间（DepTime）和航班的抵达延误时间（ArrDelay）。

航班的起飞时间（DepTime）标志着航班计划的启动，它与航空公司的调度能力和机场的运营效率紧密相关，并且能够反映出航路积压对航班时间的影响。因此，我依据起飞时间来划分时间段进行分析。同时，航班的抵达延误时间（ArrDelay）是判断航班是否延误的决定性指标。

考虑到 DepTime 在提供的数据集中是以“数”的形式存储的（格式为 hhmm，保留一位小数），其在数据集中的表现形式因时间段而异，例如凌晨 00:23 在延误数据集中表示为 23.0，早上 7:45 在数据中表示为 745.0，而下午 15:13 则被表示为 1513.0。为了从中提取准确的小时信息，我首先将起飞时间转换为字符串格式，并根据其位数进行解析，从而将起飞时间转换为对应的小时数。

完成这一转换后，我按照小时对数据进行分组，并统计了每个小时内的总延误航班数。通过比较一天中各个时间段的延误航班数量，可以确定延误最为严重的时间段。具体来说，延误航班数最多的时间段即为延误最为严重的时段。以下是我的代码实现：

代码 2：分析一天中延误最严重的飞行时间

```
def analyzeDelayByDay():  
    """  
    Task: 分析一天中延误最严重的时间段  
    """  
    # 选择所需的列：起飞时间和到达延误时间  
    columnsNeeded = ['DepTime', 'ArrDelay']  
    dataSelected = data.select(columnsNeeded)
```

```

# 数据清洗: 移除到达延误为负值或缺失的航班 (即没有延误的航班)
dataCleaned = dataSelected.filter(col('ArrDelay').cast('int') >=
0)

# 从 DepTime 中提取小时信息, 并处理不同长度的起飞时间格式
dataCleaned = dataCleaned.withColumn('DepTimeStr', expr("cast(DepTime as string)"))
dataWithHour = dataCleaned.withColumn(
    'Hour',
    expr("""
CASE
    # 长度为5, 说明类似于7: 54 (即754.0), 第一位是小时
    WHEN length(DepTimeStr) = 5 THEN cast(substring(DepTimeStr, 0, 1) AS INT)
    # 长度为6, 说明类似于17: 32 (即1732.0), 第1、2位是小时
    WHEN length(DepTimeStr) = 6 THEN cast(substring(DepTimeStr, 0, 2) AS INT)
    # 长度小于5, 说明类似于00: 12 (即12.0), 小时为0
    WHEN length(DepTimeStr) < 5 THEN 0
    ELSE NULL
END
"""))
) # 将起飞时间转换为字符串格式, 并根据其位数进行解析, 从而将起飞时间转换为对应的小时数
# 计算每个小时的总延误航班数
sumDelayByHour = (
    dataWithHour
    .filter(col('ArrDelay') > 0) # 只保留有延误的航班
    .groupBy('Hour') # 按小时分组
    .agg(count('*').alias('SumOfDelayInOneHour')) # 统计每小时的延误航班数
)
# 按小时排序并返回结果
delayList = sumDelayByHour.orderBy('Hour')
delayList.show(24) # 显示一天中所有小时的延误情况

return delayList

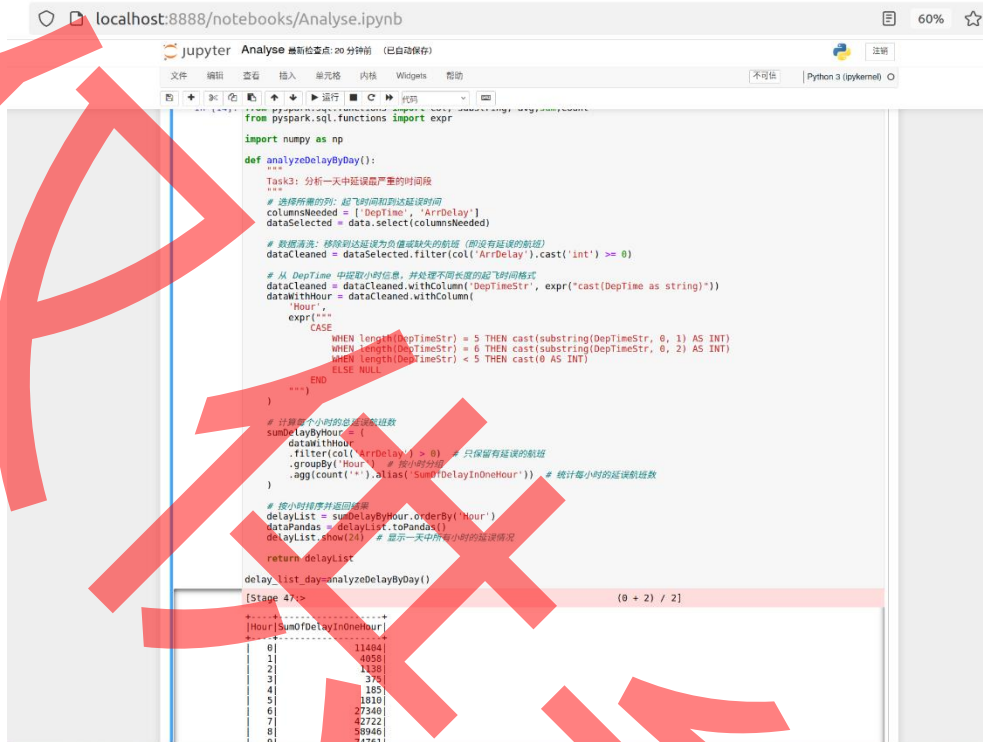
delay_list_day=analyzeDelayByDay()

```

函数代码 `analyzeDelayByDay` 的主要功能即是分析一天中航班延误最严重的时间段。首先, 它从原始数据中选择了 `DepTime` (起飞时间) 和 `ArrDelay` (到达延误时间) 两列, 并过滤掉没有到达延误的航班数据。接着, 通过提取 `DepTime` 中的小时信息, 按起飞时间的小时数画段。然后, 函数统计每个小时段内延误航班

的总数，并按小时排序，最终返回一个包含每个小时延误航班数量的结果集。通过这个分析，可以直观地看出一天中哪些时间段航班延误最为严重，为航空公司或机场优化运营提供参考依据。

运行过程如图，详细结果分析写入第九部分。



```
def analyzeDelayByDay():
    """
    Task3: 分析一天中延误最严重的时间段
    """
    # 选择所需的列: 起飞时间和到达延误时间
    columnsNeeded = ['DepTime', 'ArrDelay']
    dataSelected = data.select(columnsNeeded)

    # 数据清洗: 移除到达延误为负值或缺失的航班 (即没有延误的航班)
    dataCleaned = dataSelected.filter(col('ArrDelay').cast('int') >= 0)

    # 从 DepTime 中提取小时信息, 并处理不同长度的起飞时间格式
    dataCleaned = dataCleaned.withColumn('DepTimeStr', expr("cast(DepTime as string)"))
    dataWithHour = dataCleaned.withColumn(
        'Hour',
        expr("""
        CASE
            WHEN length(DepTimeStr) = 5 THEN cast(substring(DepTimeStr, 0, 1) AS INT)
            WHEN length(DepTimeStr) = 6 THEN cast(substring(DepTimeStr, 0, 2) AS INT)
            WHEN length(DepTimeStr) < 5 THEN cast(0 AS INT)
            ELSE NULL
        END
        """))
    """
    # 计算每个小时的总延误架次
    sumDelayByHour = (
        dataWithHour
        .filter(col('ArrDelay') > 0) # 只保留有延误的航班
        .groupBy('Hour') # 按小时分组
        .agg(count('*').alias('SumOfDelayInOneHour')) # 统计每个小时的延误架次数
    )

    # 按小时排序并返回结果
    delayList = sumDelayByHour.orderBy('Hour')
    dataPandas = delayList.toPandas()
    delayList.show(24) # 展示一天中所有小时的延误情况

    return delayList

delay_list_day=analyzeDelayByDay()
[Stage 47]:> (0 + 2) / 2]
+-----+-----+
|Hour|SumOfDelayInOneHour|
+-----+-----+
|0|11484|
|1|4058|
|2|1138|
|3|379|
|4|185|
|5|3010|
|6|27340|
|7|42722|
|8|38946|
|9|47611|
+-----+-----+
```

(2) 可视化展现一天中各时段延误架次情况折线图

为了更好的可视化呈现数据运行结果，使用 `matplotlib` 进行绘图，可视化展现了各个时间段的延误架次信息，相关代码如下——

代码 3: 可视化一天中延误最严重的飞行时间（折线图）

```
def plotDelayByDay(delays):
    """
    绘制延误时间图表
    """

    pandas_data = delays.toPandas()
    plt.style.use('seaborn-poster')
    # 绘制柱状图
    plt.figure(figsize=(10, 8))
    ax = pandas_data.plot(kind='line', x='Hour',
                           y='SumOfDelayInOneHour', marker='o', linestyle='--',
                           legend=False, color='#15aabf')
    # 设置标题和轴标签
    plt.title('Delay by Hour of the Day', fontsize=18)
```



```

plt.xlabel('Hour', fontsize=16)
plt.ylabel('Flight delays per hour', fontsize=16)
# 设置刻度标签
yticks = ax.get_yticks()
ax.set_yticklabels(['{:,.}{}'.format(int(y)) for y in yticks])
# 显示图表
plt.tight_layout()
plt.show()

plotDelayByDay(delay_list_day)

```

代码用于绘制一天中不同小时段的航班延误情况图表。首先将 Spark 数据集转换为 Pandas DataFrame。接着，通过绘制折线图，并进行配置，纵轴展示了每个小时段的航班延误总数，折线图中使用圆点标记每个数据点，并以实线连接。

(3) 可视化展现一天中各时段延误架次占总延误架次比例饼图

当然，也可以用圆饼图展示每个小时延误架次占总延误架次的大小情况。圆饼图能够以一种直观且易于理解的形式，突出各个数据的百分比情况，快速识别出延误严重（延误比例大）的时间段。具体代码如下——

代码 4：可视化一天各时段延误架次比例（饼图）

```

def portionDelayByDay(delays):
    """
    绘图：绘制一个饼图，分析每小时延误架次比例
    """
    pandasData = delays.toPandas()
    plt.style.use('seaborn-poster')
    # 计算每个小时的延误架次占总架次的百分比
    delaysSum = pandasData['SumOfDelayInOneHour'].sum()
    percentages = pandasData['SumOfDelayInOneHour'] / delaysSum * 100
    # 将小于 1% 的区域聚合为一个整体，方便审查
    smallAreas = pandasData[percentages < 2]
    largeAreas = pandasData[percentages >= 2]
    otherSum = smallAreas['SumOfDelayInOneHour'].sum()
    otherPercentage = otherSum / delaysSum * 100
    # 构建新的数据集，包含“其他”类别
    newData = largeAreas.copy()
    newData = newData.append({
        'Hour': '(23~6)',
        'SumOfDelayInOneHour': otherSum
    }, ignore_index=True)
    # 绘制饼图
    plt.figure(figsize=(10, 8))

```



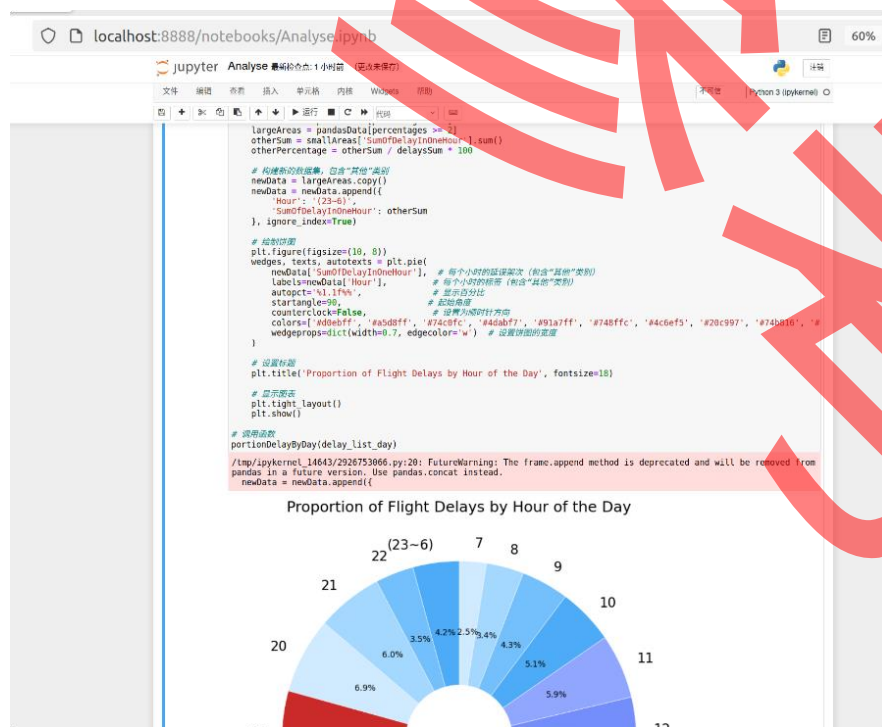
```

wedges, texts, autotexts = plt.pie(
    newData['SumOfDelayInOneHour'], # 每个小时的延误架次
    labels=newData['Hour'],         # 每个小时的标签（包含“其他”类别）
    autopct='%1.1f%%',              # 显示百分比
    startangle=90,                   # 起始角度
    counterclock=False,              # 设置为顺时针方向
    colors=['#d0ebff', '#a5d8ff', '#74c0fc', '#4dabf7', '#91a7ff',
            '#748ffc', '#4c6ef5', '#20c997', '#74b816', '#f59f00', '#f76707',
            '#fa5252', '#c92a2a'],
    wedgeprops=dict(width=0.7, edgecolor='w') # 设置饼图的宽度
)
# 设置标题
plt.title('Proportion of Flight Delays by Hour of the Day', fontsize=18)
plt.tight_layout()
plt.show()
# 调用函数
portionDelayByDay(delay_list_day)

```

这段代码用于分析每小时的航班延误架次占总延误架次的比例。代码首先将延误数据转换为 **Pandas DataFrame**，并计算每个小时的延误架次占总延误架次的百分比。为了提高图表的可读性，代码将延误比例小于 2% 的小区域聚合为一个“其他”类别（标记为 (23~6)），并在饼图中聚合显示。通过使用 **plt.pie** 函数，代码绘制了一个顺时针方向的饼图，从顶部（12 点钟方向）开始。

相关运行截图如图，详细的运行结果见文档第九部分——



(4) 分析并可视化展现各时段延误的原因

为了更全面分析每个时段航班延误的具体原因。我选择了 CarrierDelay, WeatherDelay, NASDelay, SecurityDelay, LateAircraftDelay 这 5 个变量进行分析, 其分别代表了由航空公司、天气状况、空中交通管理、安全检查以及前序飞机晚到等因素引起的延误, 通过统计每个时段这些原因造成的延误次数, 可以深入分析造成各个时段延误的主要原因, 从而识别和比较不同时间段的延误模式。

我仍然以 ArrDelay (到达延误分钟数) 作为衡量航班是否延误的直接指标。以 DepTime (起飞时间) 作为 0~23 时间段划分依据。在每个时间段内, 统计由 CarrierDelay (航空公司延误)、WeatherDelay (天气延误)、NASDelay (空中交通管理系统延误)、SecurityDelay (安全检查延误) 以及 LateAircraftDelay (飞机晚到延误) 这五种主要延误原因所引起的航班延误次数。值得注意的是, 一个航班的延误可能由多种因素共同导致, 例如, 同一航班既可能受到恶劣天气的影响, 也会由于前序航班晚点而延误, 因此, 当统计各时段内不同延误原因的架次总数时, 有可能会超过该时段内实际发生延误的航班总数。具体代码如下:

代码 5: 各时段航班延误原因统计

```
def analyzeDelayByHourReason():
    """
    分析每个小时因不同延误原因导致的延误架次
    """
    # 选择所需的列
    columnsNeeded = ['DepTime', 'ArrDelay', 'CarrierDelay', 'WeatherDelay', 'NASDelay', 'SecurityDelay', 'LateAircraftDelay']
    # 选择列并进行数据清洗
    dataSelected = data.select(columnsNeeded)
    # 确保所有延误原因列是数值类型, 并将 null 值替换为 0
    delayReasons = ['CarrierDelay', 'WeatherDelay', 'NASDelay', 'SecurityDelay', 'LateAircraftDelay']
    for reason in delayReasons:
        dataSelected = dataSelected.withColumn(reason, F.coalesce(F.col(reason).cast('float'), F.lit(0.0)))
    dataCleaned = dataSelected.filter(col('ArrDelay') > 0)
    # 从 DepTime 中提取小时信息, 并处理不同长度的起飞时间格式
    dataCleaned = dataCleaned.withColumn('DepTimeStr', expr("cast(DepTime as string)"))
    dataWithHour = dataCleaned.withColumn(
        'Hour',
        expr("""
        CASE
            WHEN length(DepTimeStr) = 5 THEN cast(substring(DepTimeStr, 0, 1) AS INT)
        """))
```

```

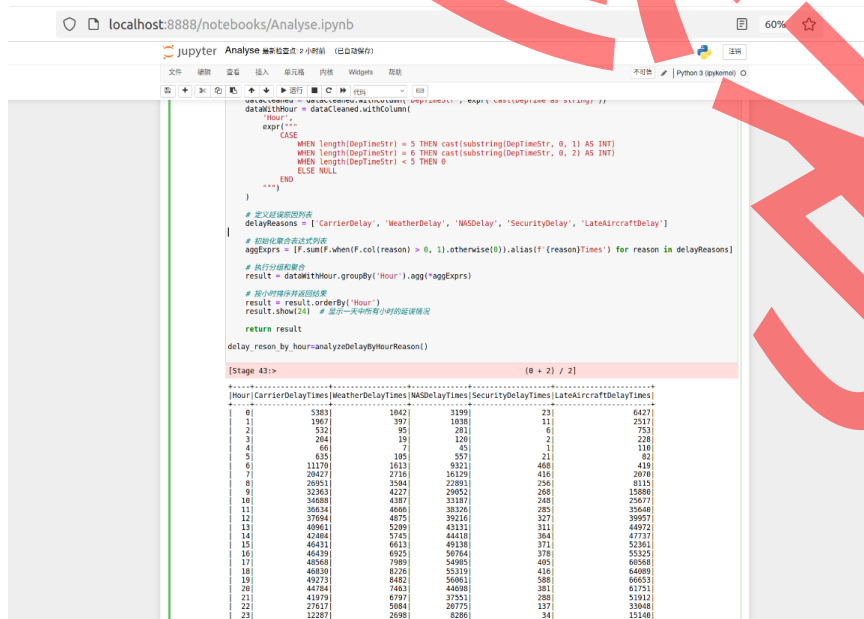
        WHEN length(DepTimeStr) = 6 THEN cast(substring(DepTimeStr, 0, 2) AS INT)
        WHEN length(DepTimeStr) < 5 THEN 0
        ELSE NULL
    END
)
)
# 定义延误原因列表
delayReasons = ['CarrierDelay', 'WeatherDelay', 'NASDelay', 'SecurityDelay', 'LateAircraftDelay']
# 初始化聚合表达式列表
aggExprs = [F.sum(F.when(F.col(reason) > 0, 1).otherwise(0)).alias(f'{reason}Times') for reason in delayReasons]
# 执行分组和聚合
result = dataWithHour.groupBy('Hour').agg(*aggExprs)
result = result.orderBy('Hour')
result.show(24) # 显示一天中所有小时的延误情况
return result

delay_reson_by_hour=analyzeDelayByHourReason()

```

这段代码用于分析航空数据集中每个小时因不同延误原因导致的延误次数。首先选择和清洗数据，并过滤掉没有延误的航班。随后从起飞时间中提取小时信息，并为每个延误原因（航空公司延误、天气延误、空中交通管理系统延误、安全检查延误以及飞机晚到延误）创建聚合表达式，表达式将计算每个小时内由于每个特定原因导致的延误航班次数。通过调用 `groupBy` 和 `agg` 方法，函数按小时对数据进行分组，并应用聚合表达式来计算每个原因的延误次数。

运行过程截图如下，详细结果及分析见第九部分——



The screenshot shows a Jupyter Notebook interface with the following code and output:

```

def analyzeDelayByHourReason():
    # ... (code from the previous block) ...
    return result

delay_reson_by_hour=analyzeDelayByHourReason()

```

The output is a table showing the number of delays for each reason across 24 hours. The table has 6 columns: Hour, CarrierDelayTimes, WeatherDelayTimes, NASDelayTimes, SecurityDelayTimes, and LateAircraftDelayTimes.

Hour	CarrierDelayTimes	WeatherDelayTimes	NASDelayTimes	SecurityDelayTimes	LateAircraftDelayTimes
0	33831	1842	31399	231	6427
1	1967	397	1038	11	2517
2	532	95	281	6	753
3	204	19	120	2	228
4	66	7	45	1	116
5	635	185	597	21	82
6	11170	1013	9321	466	419
7	20427	2716	16129	416	2076
8	20951	3584	22891	256	5115
9	32363	4227	24952	268	1586
10	34688	4387	33387	248	25677
11	38634	4666	38376	285	3546
12	37864	4675	39216	327	39671
13	40861	5298	43313	311	44972
14	42484	5745	44418	364	47737
15	46432	6613	49138	371	52361
16	46439	6925	50764	378	53325
17	48568	7989	54985	495	6056
18	48830	8226	55319	416	64089
19	49273	8482	56061	588	66053
20	44764	7463	44008	381	61751
21	41979	6797	37551	288	51912
22	27517	5684	20775	137	33048
23	12287	2698	8286	34	15148

下面的代码呈现了一个分组柱状图，可视化展现了各时段航班延误原因统计：

代码 6：可视化各时段航班延误原因统计

```
pandas_df = delay_reson_by_hour.toPandas()
pandas_df = pandas_df[pandas_df['Hour'].between(0, 23)]
plt.figure(figsize=(16, 9))
# 定义 x 轴的位置
x = np.arange(len(pandas_df['Hour']))
num_bars_per_group = len(['CarrierDelayTimes', 'WeatherDelayTimes', 'NASDelayTimes', 'SecurityDelayTimes', 'LateAircraftDelayTimes'])
total_width = 0.7 # 每个组的总宽度（包括组内所有柱子）
bar_width = total_width / num_bars_per_group # 单个柱子的宽度
# 创建分组柱状图
fig, ax = plt.subplots()

# 为每个延误原因绘制柱状图
delay_reasons = ['CarrierDelayTimes', 'WeatherDelayTimes', 'NASDelayTimes', 'SecurityDelayTimes', 'LateAircraftDelayTimes']
colors = ['skyblue', 'orange', 'lightgreen', 'red', 'purple']
for i, reason in enumerate(delay_reasons):
    ax.bar(x + (i - num_bars_per_group / 2 + 0.5) * bar_width, pandas_df[reason], bar_width, label=reason, color=colors[i])
ax.set_xlabel('Hour')
ax.set_ylabel('DelayCounts')
ax.set_title('Flight delays per hour due to various reasons')
ax.set_xticks(x + width) # 设置 x 轴刻度位置
ax.set_xticklabels(pandas_df['Hour'])
ax.legend() # 显示图例
# 显示图表
plt.tight_layout()
plt.show()
```

代码使用 Matplotlib 绘制了一个分组柱状图，展示了每个小时因不同延误原因（如 CarrierDelay、WeatherDelay 等）导致的航班延误数量。最后，通过设置标题、轴标签和图例，增强了图表的可读性。

可视化的具体结果见第九部分。

8.3 使用梯度提升树(GBDT)模型预测未来航班取消情况

我选用梯度提升树(GBDT)这一典型机器学习算法进行预测。

（1）预处理航班数据集

首先读取了名为 DelayedFlights.csv 的 CSV 文件，并将其存储在变量 flightData 中。然后，代码定义了可预测的特征和目标变量，其中可预测特征包括年份、月

份、日、星期、计划起飞时间、计划到达时间、唯一承运人、航班号、机尾号、计划飞行时间和距离等，而目标变量为 **Cancelled**（是否取消）。

接下来，代码从数据集中选择了特征和目标变量，并删除了含有缺失值的行，以确保特征矩阵和目标变量索引的一致性。随后，代码定义了哪些特征是分类特征，哪些是数值特征，并创建了一个 **ColumnTransformer** 来分别对这两类特征进行处理。数值特征通过 **StandardScaler** 进行标准化，而分类特征则通过 **OneHotEncoder** 进行独热编码。

在预处理完成后，代码将处理后的特征矩阵应用于训练集和测试集的分割，其中测试集占总数据的 20%，并设置了随机种子以确保结果的可重复性。至此，数据预处理和分割工作已完成，为后续模型训练和预测做好了准备。

具体代码如下——

代码 7：数据预处理与训练集划分

```
import pandas as pd
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.model_selection import train_test_split
# 读取数据
flightData = pd.read_csv('./DelayedFlights.csv')
data=flightData
data.head(10)

# 定义可预知的特征和目标变量
predictable_features = ['Year', 'Month', 'DayOfMonth', 'DayOfWeek',
                        'CRSDepTime', 'CRSArrTime', 'UniqueCarrier', 'FlightNum', 'TailNum',
                        'CRSElapsedTime', 'Distance', 'Origin', 'Dest']
target_variable = 'Cancelled'
# 选择特征和目标变量
X = data[predictable_features]
y = data[target_variable]
# 删除含有缺失值的行
X = X.dropna()
y = y[X.index] # 确保 y 与 X 的索引对齐

# 定义分类特征和数值特征
categorical_features = ['UniqueCarrier', 'FlightNum', 'TailNum',
                        'Origin', 'Dest']
numerical_features = ['Year', 'Month', 'DayOfMonth', 'DayOfWeek',
                      'CRSDepTime', 'CRSArrTime', 'CRSElapsedTime', 'Distance']
# 创建预处理管道
preprocessor = ColumnTransformer(
    transformers=[
```

```

        ('num', StandardScaler(), numerical_features),
        ('cat', OneHotEncoder(handle_unknown='ignore'), categorical_features)
    ])
# 划分训练集和验证集
X_train, X_test, y_train, y_test = train_test_split(X_train, y_train, test_size=0.25, random_state=42)

```

```

# 数据处理
In [1]: import pandas as pd
        from sklearn.preprocessing import StandardScaler, OneHotEncoder
        from sklearn.compose import ColumnTransformer
        from sklearn.model_selection import train_test_split

        flightData = pd.read_csv('DelayedFlights.csv')
        data = flightData

In [2]: # 统计Cancelled的个数并打印
        cancel_count = data[data['Cancelled'] == 1].shape[0]
        print("Number of 'cancel' entries with value 1: cancel count:")
        Number of 'cancel' entries with value 1: 635

In [3]: data.head(10)
Out[3]:
   Unnamed: 0  Year  Month  DayOfMonth  DayOfWeek  DepTime  CRSDepTime  ArrTime  CRSArrTime  UniqueCarrier  TailNum  TaxiOut  Cancelled  CancellationTime
0          0    1     2008         1         3         4      2008.0      1905.0  2791.0      2055.0      NAK      4.0      0.0      0.0
1          1    1     2008         1         3         4      174.0       735.0  1002.0      1000.0      NAK      5.0      0.0      0.0
2          2    2     2008         1         3         4      628.0       620.0      804.0       790.0      NAK      3.0      1.0      0.0
3          3    4     2008         1         3         4      1028.0      1155.0  1559.0      1625.0      NAK      3.0      0.0      0.0
4          4    8     2008         1         3         4      1945.0      1935.0  2521.0      2110.0      NAK      4.0      10.0      0.0
5          5    9     2008         1         3         4      1607.0      1830.0  2007.0      1940.0      NAK      3.0      7.0      0.0
6          6   10     2008         1         3         4      708.0       700.0      916.0       815.0      NAK      5.0      19.0      0.0
7          7   14     2008         1         3         4      1644.0      1530.0  1641.0      1715.0      NAK      0.0      0.0      0.0
8          8   15     2008         1         3         4      1038.0      1030.0  1551.0      1610.0      NAK      0.0      0.0      0.0
9          9   15     2008         1         3         4      1421.0      1420.0  1641.0      1655.0      NAK      1.0      0.0      0.0

10 rows x 30 columns

In [4]: # 定义特征和标签
        predictable_features = ['Year', 'Month', 'DayOfMonth', 'DayOfWeek', 'CRSDepTime', 'CRSArrTime', 'UniqueCarrier', 'TailNum', 'CRSDepTime']
        target_variable = 'Cancelled'

        # 训练数据
        X = data[predictable_features]
        y = data[target_variable]

        # 训练模型

```

(2) 梯度提升树(GBDT)进行模型训练

然后，使用梯度提升树（Gradient Boosting Decision Tree, GBDT）算法来预测航班是否取消。代码逻辑如下：首先，然后创建一个预处理管道，管道包括数值特征的标准化和类别特征的编码。随后，计算样本权重以处理数据不平衡问题。

紧接着，我使用“早停策略”来推进训练循环。早停策略通过设定一个“耐心值”来预防过拟合现象，这在处理不平衡数据集（如本数据集中取消航班的样本数量相对较少）时尤为重要。在这种数据分布下，模型有可能会偏向于多数类，而忽略对少数类的学习。长时间的训练可能导致模型在多数类上过度拟合。早停策略通过监测验证集上的性能，一旦发现性能不再提升，便立即停止，从而避免了对多数类的过拟合。代码如下——

代码 8：梯度提升树 GBDT 训练

```

from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report, roc_curve, auc
from sklearn.preprocessing import StandardScaler
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.utils.class_weight import compute_class_weight
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.utils.class_weight import compute_sample_weight
import numpy as np

```



```

# 创建梯度提升树分类器
gbdt_classifier = GradientBoostingClassifier(n_estimators=1000,
learning_rate=0.1, max_depth=3, random_state=42, warm_start=True)
# 创建管道并计算样本权重
pipeline = Pipeline(steps=[('preprocessor', preprocessor),
                           ('classifier', gbdt_classifier)])
sample_weights = compute_sample_weight(class_weight='balanced', y=y_train)
# 初始化验证集的最佳性能和迭代次数
best_val_score = 0
best_iter = 0
patience = 10
patience_counter = 0
# 训练循环，实现早停
for i in range(1, 1001):
    pipeline.named_steps['classifier'].n_estimators = i
    pipeline.fit(X_train, y_train, classifier__sample_weight=sample_weights)
    # 验证集性能
    val_predictions = pipeline.predict(X_val)
    val_score = accuracy_score(y_val, val_predictions)
    # 如果验证集性能提升，更新最佳性能和迭代次数
    if val_score > best_val_score:
        best_val_score = val_score
        best_iter = i
        patience_counter = 0
    else:
        patience_counter += 1
    # 如果耐心耗尽，停止训练
    if patience_counter >= patience:
        print(f"Early stopping at iteration {i}, best validation score:
{best_val_score}")
        break
pipeline.named_steps['classifier'].n_estimators = best_iter # 用最佳迭代更新模型

```

(3) 模型预测与评价

下面进行预测和评价。首先调用 `predict` 方法对测试集进行预测，并计算模型的准确率和分类报告。接着，它通过二值化标签计算了 ROC 曲线和 AUC 值，并打印了召回率和 AUC。最后，代码绘制了 ROC 曲线图，展示了模型在不同阈值下的真阳性率和假阳性率。

代码 9：梯度提升树 GBDT 的预测与评估

```

# 预测测试集
y_pred = pipeline.predict(X_test)
# 评估模型性能
accuracy = accuracy_score(y_test, y_pred)

```



```

classification_rep = classification_report(y_test, y_pred)
# 打印准确率和分类报告
print(f"Accuracy: {accuracy}")
print(f"Classification Report:\n{classification_rep}")
y_test_bin = label_binarize(y_test, classes=[0, 1])
y_pred_bin = label_binarize(y_pred, classes=[0, 1])

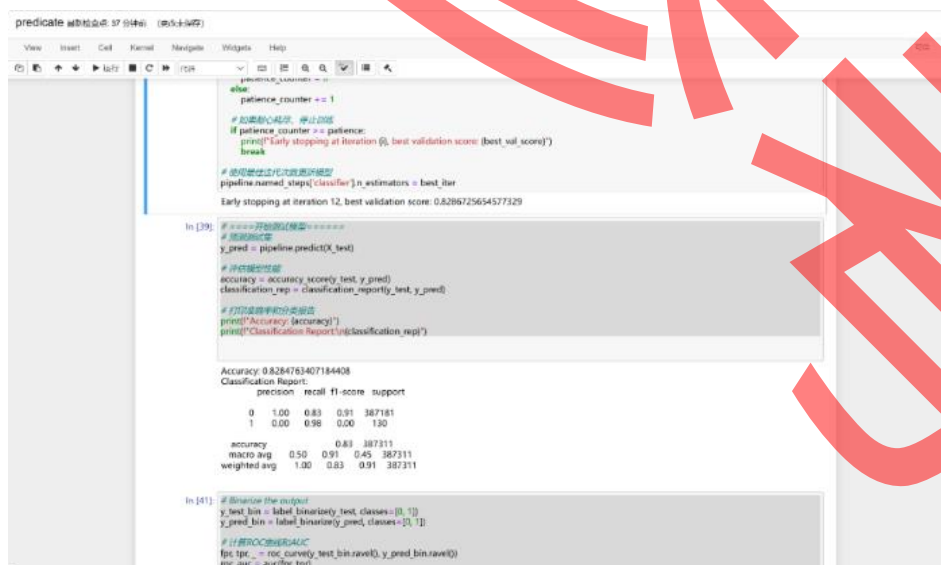
# 计算ROC曲线和AUC
fpr, tpr, _ = roc_curve(y_test_bin.ravel(), y_pred_bin.ravel())
roc_auc = auc(fpr, tpr)

# 打印召回率和AUC
print(f"Recall: {tpr[1]}")
print(f"AUC: {roc_auc}")

# 绘制ROC曲线
plt.figure()
plt.plot(fpr, tpr, color='#9775fa', lw=2, label=f'ROC curve (area = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='#20c997', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Gradient Boosting Decision Tree Receiver Operating Characteristic')
plt.legend(loc="lower right")
plt.show()

```

运行过程截图如下（具体结果见第九部分阐述）——



```

predicate 训练模型: 37 分钟前 (默认未保存)
View Insert Cell Kernel Navigate Widgets Help

else:
    patience_counter += 1
# 训练耐心结束，停止训练
if patience_counter >= patience:
    print("Early stopping at iteration 0, best validation score: (best_val_score)")
    break

# 使用训练好的模型进行预测
pipeline.named_steps['classifier'].n_estimators = best_iter
Early stopping at iteration 12, best validation score: 0.8284763407184408

In [39]:
# 开始测试模型
# 预测结果
y_pred = pipeline.predict(X_test)
# 计算准确率
accuracy = accuracy_score(y_test, y_pred)
classification_rep = classification_report(y_test, y_pred)
# 打印准确率和分类报告
print(f"Accuracy: {accuracy}")
print(f"Classification Report:\n{classification_rep}")

Accuracy: 0.8284763407184408
Classification Report:
  precision    recall  f1-score   support

    0       1.00      0.83      0.91      367181
    1       0.00      0.98      0.00         130

 accuracy          0.83   0.87311
 macro avg       0.50   0.91   0.45   367311
 weighted avg    1.00   0.83   0.91   367311

In [41]:
# Binarize the output
y_test_bin = label_binarize(y_test, classes=[0, 1])
y_pred_bin = label_binarize(y_pred, classes=[0, 1])
# 计算ROC曲线和AUC
fpr, tpr, _ = roc_curve(y_test_bin.ravel(), y_pred_bin.ravel())
roc_auc = auc(fpr, tpr)

```

至此，完成了梯度提升树模型的训练与预测，具体的分析结果和评测指标将在第九部分详细阐述。

九、项目结果与分析（含重要数据结果分析或核心代码流程分析）

Task 3. 分析一天中延误最严重的飞行时间。

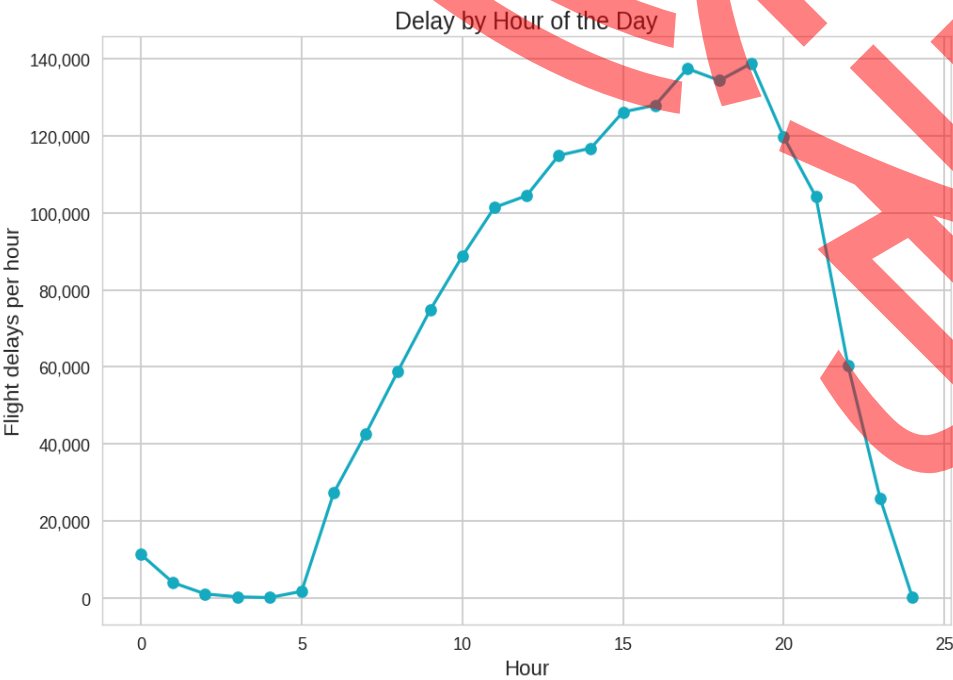
- 各时段延误架次、延误率分析及可视化

为了分析一天中延误最严重的飞行时间，我将一天按每小时划分为 24 个时间段（以起飞时间作为时段划分依据）。统计各个时间段航班延误频次（架次）。

Hour	SumOfDelayInOneHour
0	11404
1	4058
2	1138
3	375
4	185
5	1810
6	27340
7	42722
8	58946
9	74761
10	88691
11	101355
12	104397
13	114844
14	116727
15	126114
16	127853
17	137365
18	134269
19	138750
20	119608
21	104130
22	60374
23	25912

only showing top 24 rows

如上表所示，第一列代表时间段（0~23 小时），第二列为该时段对应的延误频次。可直观感受到延误情况呈现出由轻到重，随后又逐渐减轻的动态趋势。

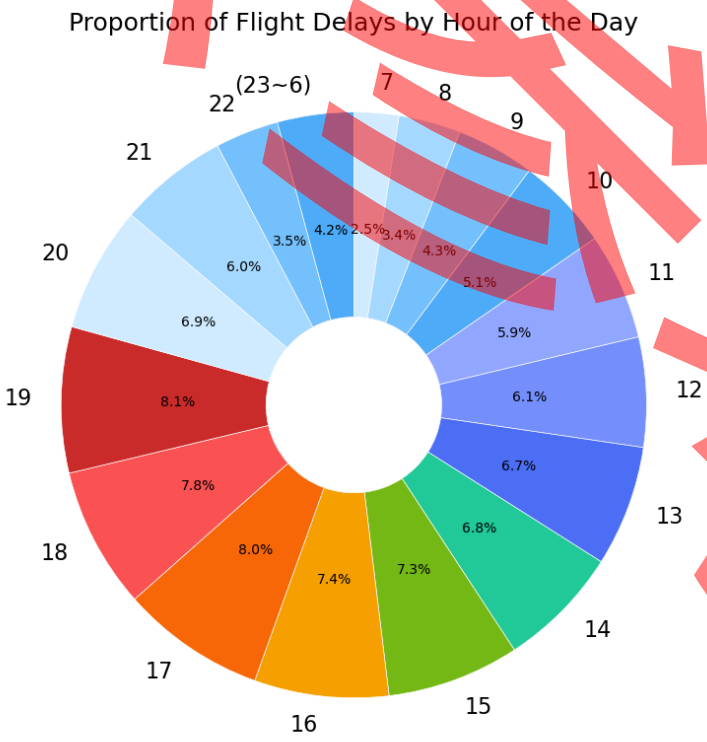


为了更清晰的反应一天中各个时段的延误情况，我绘制了如上的折线图以更直观的反应。图中横轴表示一天的每个小时（0~23），纵轴表示每小时的航班延误架次，且纵轴的刻度值以千位分隔符格式显示。

可以明显观察到，在凌晨时段（0 时至 5 时），航班的延误情况相对较少。这主要是因为在这一时间段，大多数人正处于休息状态，导致航班数量减少，航路较为畅通，因此延误事件的发生率较低；随着清晨的到来，白天的活动逐渐展开，航班的延误情况也开始逐渐增加。特别是在上午 11 点至晚上 21 点这一时间段，延误的航班数量激增至 100,000 班次。这一现象可能与乘客更倾向于在这个时间段出行，导致机场客流量增大，航路出现积压有关。随后，夜幕降临，乘客的数量逐渐减少，导致航班的飞行次数也随之降低。因此，航班的延误率也逐渐下降，恢复了更为平稳的运行状态。

延误情况的顶峰出现在傍晚 17 时至 19 时，这一时段被认为是“晚高峰”阶段，延误航班数高达 138,750 班次。许多乘客选择在这个时间段进行回程旅行或商务出行，进一步加剧了航班的延误情况。因此该时段（17 时~19 时）可以认为是一天中延误最严重的飞行时间。

另外，从“各小时起飞延误架次占总延误架次的比例”饼图（如下图）中也能看出延误情况分布。图中外围 0~23 标识各个时间段，内部的百分比数字标识该时段起飞延误架次占总延误架次的比例。



图示展示了每个小时（0~23）航班起飞延误架次占总延误架次的比例。延误比例较高的时段主要集中在 15:00 到 19:00 之间，尤其是 17:00、18:00、

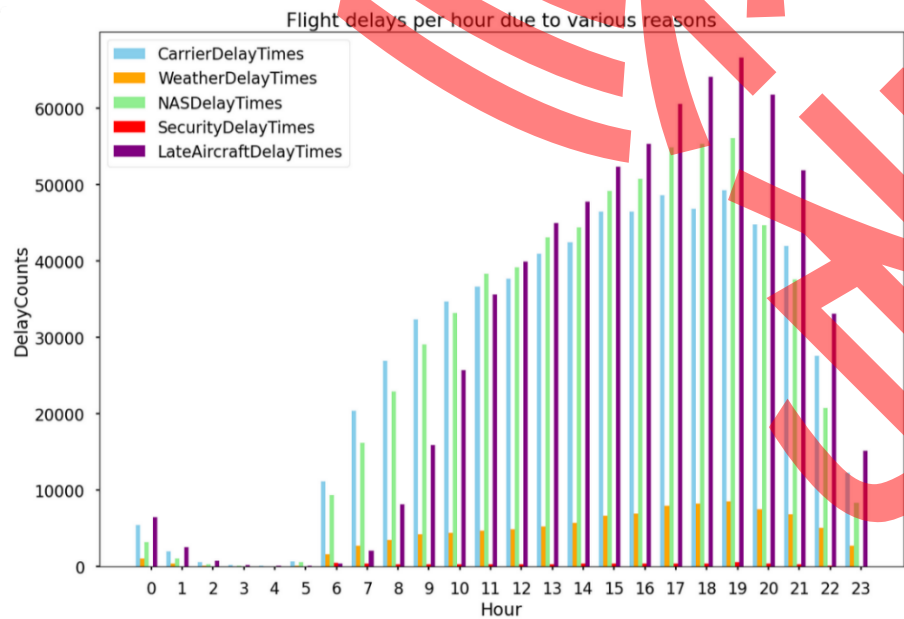
19:00，这些时段的延误比例普遍在 7.0%到 8.1%之间。其中，19:00 的延误比例最高，达到 8.1%，因此该时段（17 时~19 时）可以认为是一天中延误最严重的飞行时间。18:00、17:00 的延误比例也较高，分别为 7.8%、8.0%，表明这些时段的航班延误情况也较为突出。延误比例较低的时段主要集中在 23:00 到 6:00，延误比例总和仅占到 4.2%，这些时段的航班延误情况相对较少。

各时段延误原因统计分析及分组可视化

Hour	CarrierDelayTimes	WeatherDelayTimes	NASDelayTimes	SecurityDelayTimes	LateAircraftDelayTimes
0	5383	1842	3199	23	6427
1	1967	397	1038	11	2517
2	532	95	281	6	753
3	204	19	120	2	228
4	66	7	45	1	110
5	635	105	557	21	82
6	11170	1613	9321	468	419
7	20427	2716	16129	416	2070
8	26951	3504	22891	256	8115
9	32363	4227	29052	268	15880
10	34688	4387	33187	248	25677
11	36634	4666	38326	285	35640
12	37694	4875	39216	327	39957
13	40961	5209	43131	311	44972
14	42404	5745	44418	364	47737
15	46431	6613	49138	371	52361
16	46439	6925	50764	378	55325
17	48568	7989	54905	405	60568
18	46838	8226	55319	416	64089
19	49273	8482	56061	588	66653
20	44784	7463	44698	381	61751
21	41979	6797	37551	288	51912
22	27617	5084	20775	137	33048
23	12287	2698	8286	34	15140

only showing top 24 rows

为了探究各时段延误情况背后的原因，我选择了五种主要延误原因（航空公司延误、天气延误、空中交通管理系统延误、安全检查延误以及飞机晚到延误）进行统计，如上表所示，第一列代表时间段（0~23 小时），第二列为该时段对应的因某种原因延误的航班架次。



绘制分组柱状图，用五种颜色分别代表五种不同原因的航班延误架次。

从图表中可以明显观察到，在 11 点至 20 点之间，LateAircraftDelay（前序

飞机晚点延误）成为了造成航班延误的最主要原因。这种延误类型具有显著的“连锁效应”，即单一航班的延误往往会波及到后续多个航班的正常运作。特别是在航班密集的 17 至 19 点期间，这种连锁反应被显著放大，导致 "LateAircraftDelay" 达到高峰，进一步加剧了 17~19 点的延误情况。

当然，CarrierDelay（航空公司延误）、NASDelay（空管系统延误）也是造成延误情况的主要因素，特别是随着高峰时段（11 点~20 点）的到来，航路压力较大，因 NASDelay（空重管制）造成的延误次数明显升高。

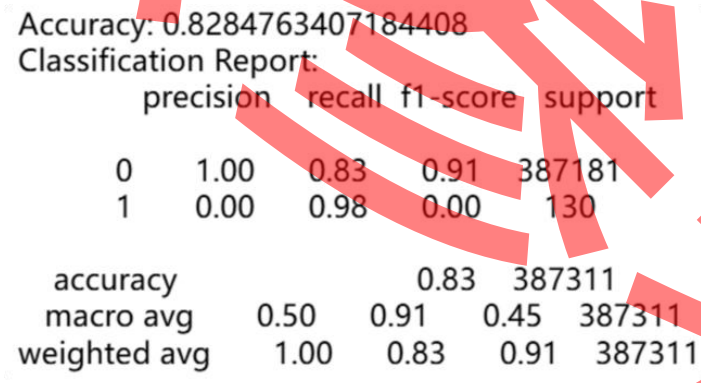
而 WeatherDelay（天气延误）和 SecurityDelay（安全延误）虽在高峰时段也有一定影响，但整体相对较小，属于次要延误原因。

• **结论总结**

综上所述，可以得出结论：17 时到 19 时是一天中航班延误最为严重的时段。这是因为：许多乘客选择在这个时间段进行回程旅行，尤其是从商务出行或长途旅行中返回的旅客，导致航班需求量激增，进一步加剧了航线的拥挤程度，导致航路交通流量管控而延误；同时，此前的航班晚点导致的连锁反应进一步延长了延误时间，加剧了整体延误的严重程度；此外，在傍晚时分，天气变化较为频繁，可能影响航班的正常运行。

Task 6. 建立机器学习算法模型，预测未来航班取消情况。

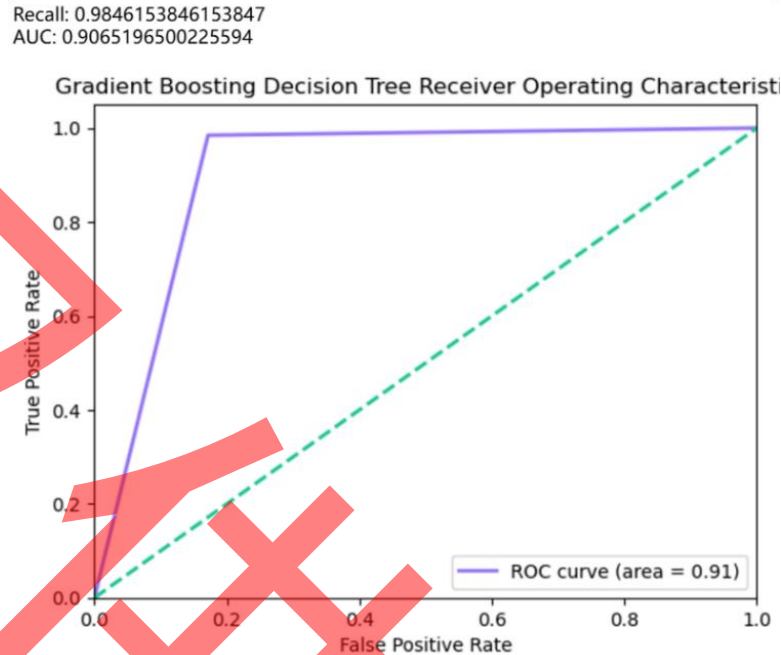
我使用了梯度提升树(GBDT)进行预测，下表给出了其预测效果评估以及一些关键指标。



可以看到梯度提升树模型在预测航班取消情况时表现出了较高的准确性和有效性。模型的准确率（Accuracy）为 0.828，这意味着模型正确预测了约 83% 的航班取消情况。在分类报告中，对于未取消的航班（标签为 0），模型的精确度为 1.00，召回率为 0.83，F1 分数为 0.91，这表明模型在预测未取消航班方面表现非常好。对于取消的航班（标签为 1），召回率高达 0.98，说明模型在识别取消航班方面几乎不会漏掉任何真实取消的航班。

我绘制了 ROC 曲线，如下图所示，其中绿色线条代表了随机预测的基准线，而紫色线条则展示了模型所得到的 ROC 曲线。梯度提升树的 AUC 值为 0.907，

说明模型在区分取消与非取消航班方面的能力较强。因此整体来看，梯度提升树模型在预测航班取消情况时具有较好的效果。



十一、总结及心得体会

在本次针对航空公司延误和取消情况的实验分析中，小组同学通过集体讨论，明确了任务分工，确保了实验的有序推进。

通过项目，我能够熟练运用大数据分析工具，能够基于 PySpark 进行数据处理。在本次项目中，我实操了“分析一天中延误最严重的飞行时间并可视化分析”和“使用梯度提升树(GBDT)模型对航班取消情况进行预测”两个主要任务，发掘出 17~19 点是一天中延误情况最严重的时间段，并给出了相关原因，同时训练了一个 AUC 为 0.9 的 GBDT 模型，能有效预测航班取消情况。工作量充实。这不仅巩固了我在理论课程中学到的知识，也加深了我对大数据分析实践的理解。

本次项目的一个显著特点是理论与实践的紧密结合。虽然在理论课程中学习了大数据分析与智能计算相关算法，但实际操作中遇到的挑战让我意识到理论与实践之间的差距。我也深深明白了“纸上得来终觉浅，绝知此事要躬行”。

通过这次实践，我不仅锻炼了独立思考和协作的能力，还对大数据分析与计算领域有了更深入的认识。

十二、对本项目过程及方法、手段的改进建议

本实验所给出的数据集中，航班取消的样本数量较少（航班取消与航班未取消样本不均衡）。因此，在建立机器学习算法模型，预测未来航班取消情况时需要进一步考虑不平衡数据集的影响。