正文

具体目标

- （1）解析网络流量数据文件，提取网络流量信息，分析网络流量特性。
- （2）以T为周期，ARMA为预测算法，根据历史周期内到达的数据帧数目信息预测当前周期内到达的数据帧数目。对比该预测结果和真实到达的数据帧数并通过RMSE、MAE等技术指标量化对比。
- （3）改变参数T的值，研究T的变化对于预测算法的影响。
- （4）模拟该网络流量作用下节能以太网的模式转换过程，统计EEEP策略下的节能效果和延时开销。
- （5）设计新的预测算法替代EEEP策略的ARMA预测算法，分析节能效果和延时开销的变化。

实验内容及具体步骤

实验步骤一：

- ①下载安装Wireshark工具并处理trace1数据；
- ②在保证实验效果的前提下可以切割其中一段trace作为实验数据，提取网络流量信息，包括但不限于数据包的大小、到达时间和一段固定时间T内达到的数据包个数等；
- ③分析网络流量特性，包括但不限于计算截取trace的数据包的平均大小、流量的负载和数据包的平均达到速率等；

1.

2. 利用python实现对 trace1 数据的切割，具体代码如下：

```
import csv
from datetime import datetime
import dpkt
import socket
from concurrent.futures import ThreadPoolExecutor

# 定义一个函数，用于处理每个数据包
```

```python
def process_packet(ts, buf):
    # 将数据包解析为Ethernet帧
    pkt = dpkt.ethernet.Ethernet(buf)
    row = []
    # 计算时间戳与参考时间戳之间的差值，并添加到行中
    row.append(float(ts) - 1588136400.238425)

    if isinstance(pkt.data, dpkt.ip.IP):
        # 如果是IPv4数据包，解析源IP地址、目的IP地址、协议类型和数据包长度，并添加到行中
        ip = pkt.data
        src_ip = socket.inet_ntoa(ip.src)
        dst_ip = socket.inet_ntoa(ip.dst)
        row.extend([src_ip, dst_ip, ip.p, len(pkt)])
    elif isinstance(pkt.data, dpkt.ip6.IP6):
        # 如果是IPv6数据包，解析源IPv6地址和目的IPv6地址，并添加到行中
        ip6 = pkt.data
        src_ip6 = socket.inet_ntop(socket.AF_INET6, ip6.src)
        dst_ip6 = socket.inet_ntop(socket.AF_INET6, ip6.dst)
        row.extend([src_ip6, dst_ip6])

        # 根据IPv6数据包的下一层协议类型，添加相应的协议类型值到行中
        if isinstance(ip6.data, dpkt.icmp.ICMP):
            row.append(1)  # ICMP协议类型为1
        elif isinstance(ip6.data, dpkt.tcp.TCP):
            row.append(6)  # TCP协议类型为6
        elif isinstance(ip6.data, dpkt.udp.UDP):
            if isinstance(ip6.data.data, dpkt.dns.DNS):
                row.append(53)  # DNS协议类型为53
            else:
                row.append(17)  # UDP协议类型为17
        elif isinstance(ip6.data, dpkt.esp.ESP):
            row.append(50)  # ESP协议类型为50
        else:
            row.append(ip6.data.__class__.__name__)  # 其他协议类型的名称
        row.append(len(pkt))
    else:
        # 如果不是IPv4或IPv6数据包，将NA（不适用）添加到行中
        row.extend(['NA', 'NA', 'NA', 'NA'])

    return row

# 定义一个函数，用于将行写入CSV文件
def write_to_csv(rows):
    with open('output1.csv', 'a+', newline='') as csv_f:
        csv_writer = csv.writer(csv_f)
        csv_writer.writerows(rows)

# 定义一个函数，用于并行处理数据包
def process_packets_in_parallel(pcap_path):
    final = []
    with open(pcap_path, 'rb') as pcap_file:
        pcap = dpkt.pcap.Reader(pcap_file)
        with ThreadPoolExecutor() as executor:
```

```
            for ts, buf in pcap:
                # 提交处理数据包的任务给线程池，并将返回的Future对象添加到final列表中
                row = executor.submit(process_packet, ts, buf)
                final.append(row.result())

                # 每处理1000个数据包，将final列表中的行写入CSV文件，并清空final列表
                if len(final) == 1000:
                    executor.submit(write_to_csv, final)
                    final = []

        # 处理剩余的数据包并写入CSV文件
        executor.submit(write_to_csv, final)

# 定义要处理的pcap文件路径
pcap_path = 'trace1.pcap'

# 创建一个新的CSV文件，并写入表头
with open('output1.csv', 'w', newline='') as csv_f:
    csv_writer = csv.writer(csv_f)
    csv_writer.writerow(['Time', 'Source', 'Destination', 'Protocol', 'Length'])

# 调用函数以并行处理数据包
process_packets_in_parallel(pcap_path)
```

> 这段代码实现了读取pcap文件中的数据包，并并行处理每个数据包。处理过程包括解析数据包的各个字段，并将结果写入CSV文件。代码使用dpkt库进行pcap文件的解析，使用ThreadPoolExecutor实现并行处理。

3. 分析网络流量特性

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from pylab import mpl
import statsmodels.api as sm
from sklearn.metrics import mean_squared_error, mean_absolute_error
from pmdarima import auto_arima
from scipy import stats
```

```
# 设置显示中文字体，需要注意此字体在mac电脑不存在，需要另行下载或者指定别的字体
mpl.rcParams["font.sans-serif"] = ["SimHei"]
# 设置正常显示符号
mpl.rcParams["axes.unicode_minus"] = False
```

```
trace = pd.read_csv('output1.csv')
trace.tail(5)
```

	Time	Source	Destination	Protocol	Length
26103995	259.399009	80.74.125.250	203.243.107.226	6	54.0
26103996	259.399023	71.253.154.177	203.243.20.108	1	42.0
26103997	259.399035	194.34.107.145	163.94.112.2	6	54.0
26103998	259.399047	133.4.234.208	3.178.180.83	6	54.0
26103999	259.399055	203.243.20.108	183.123.200.190	1	42.0

> (1). 求数据包的平均大小

```
trace['Length'].mean()
```

```
48.95719999874347
```

> (2). 求数据包的平均到达速率
>
> > 计算总时间，即最后一个数据包的时间减去第一个数据包的时间
> > 计算数据包的总数
> > 将数据包的总数除以总时间，即可得到平均速率

```
# 计算总时间
total_time = trace['Time'].max() - trace['Time'].min()
# 计算平均速率
average_rate = trace.shape[0] / total_time
average_rate
```

```
100632.59482416158
```

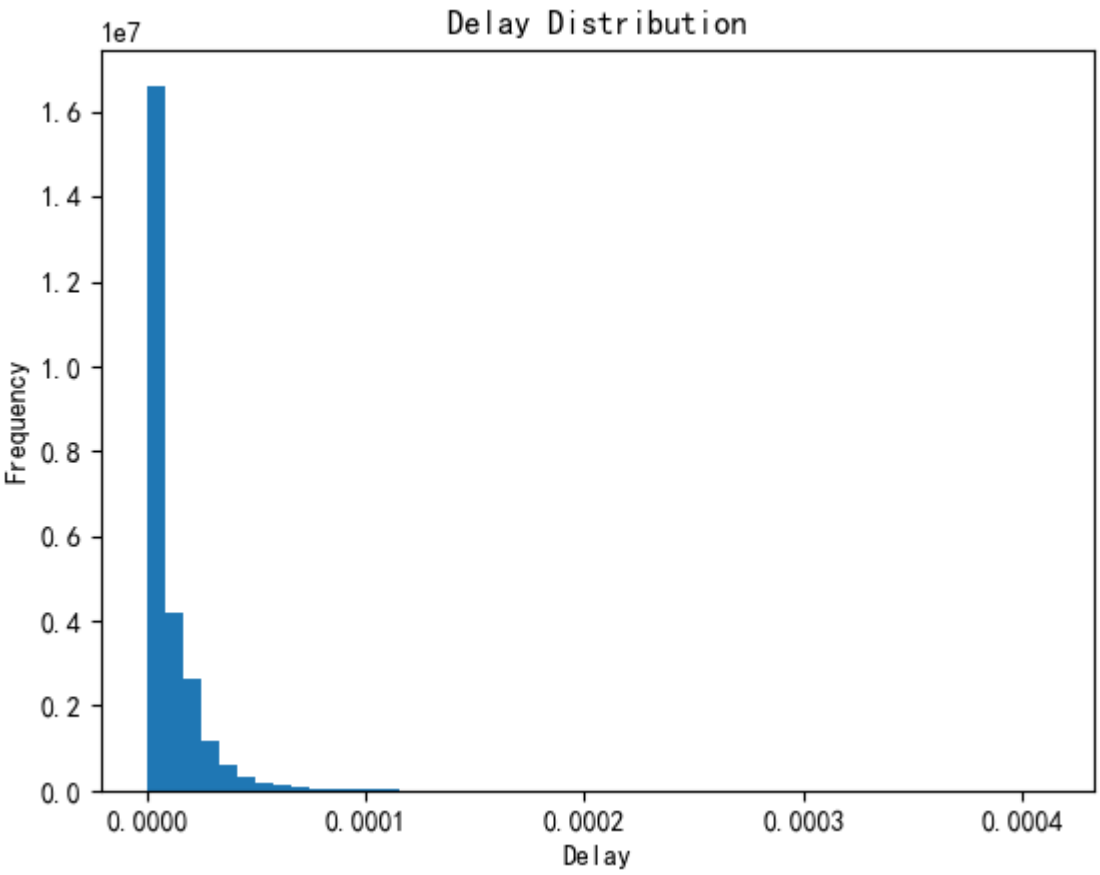
> (3). 求流量的总负载
>
> > 将数据包的长度相加即可获得总负载

```
traffic_load = trace.Length.sum()
traffic_load
```

```
1277957893.0
```

> (4). 延迟分析

```
# 计算延迟
df_delay = trace['Time'].diff()
# 移除第一个数据包的延迟，因为没有前一个数据包
df_delay.iloc[0] = 0
# 统计延迟
min_delay = df_delay.min()
max_delay = df_delay.max()
mean_delay = df_delay.mean()
# 绘制延迟分布图
plt.hist(df_delay, bins=50)
plt.xlabel("Delay")
plt.ylabel("Frequency")
plt.title("Delay Distribution")
plt.show()

print("最小延迟: ", min_delay)
print("最大延迟: ", max_delay)
print("平均延迟: ", mean_delay)
```

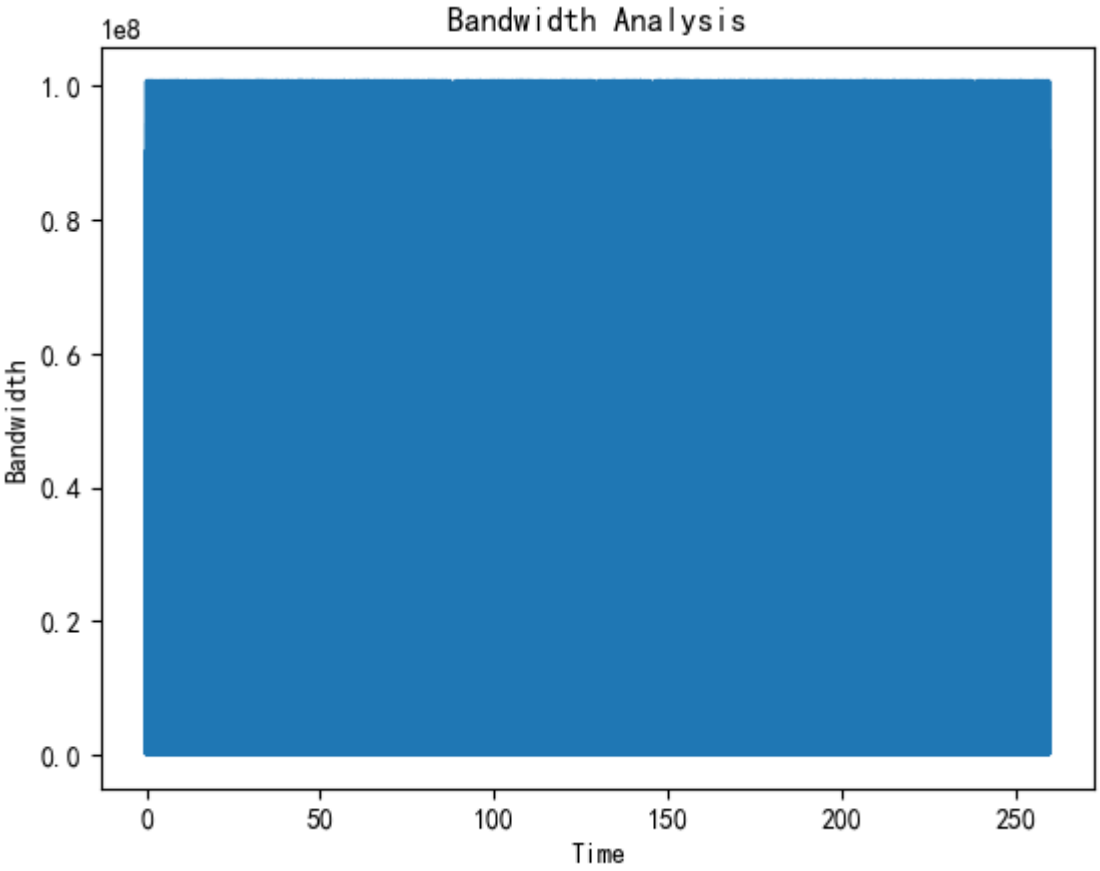


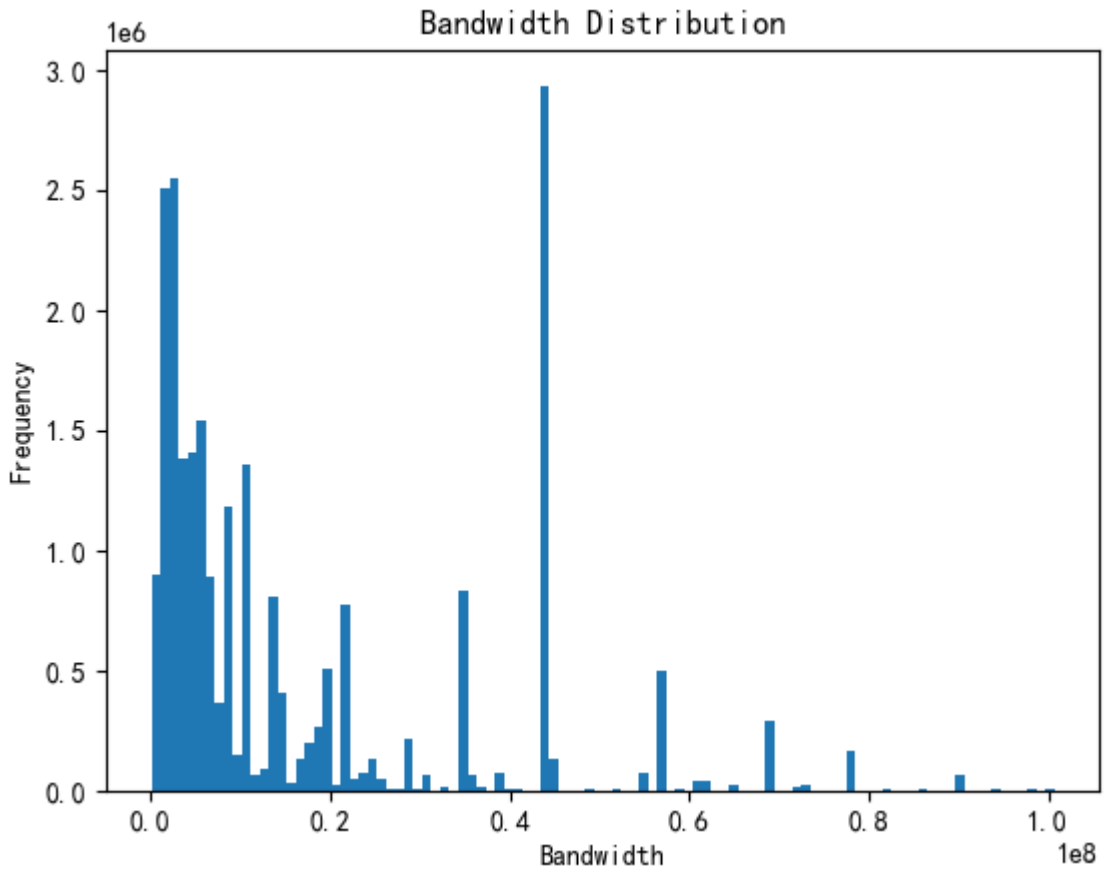

```
最小延迟:  0.0
最大延迟:  0.00041222572326660156
平均延迟:  9.937138178214828e-06
```

> (5). 带宽分析
>
> > 带宽是指在单位时间内通过网络的数据量

```
# 计算带宽
df_bandwidth = pd.DataFrame({"bandwidth": trace.Length / df_delay, "Time": trace.Time})
# 去除inf数据
df_bandwidth.replace([np.inf, -np.inf], np.nan, inplace=True)
df_bandwidth.dropna(inplace=True)
# 统计带宽
min_bandwidth = df_bandwidth.bandwidth.min()
max_bandwidth = df_bandwidth.bandwidth.max()
mean_bandwidth = df_bandwidth.bandwidth.mean()
# 绘制带宽随时间的变化曲线
plt.plot(df_bandwidth.Time, df_bandwidth.bandwidth)
plt.xlabel("Time")
plt.ylabel("Bandwidth")
plt.title("Bandwidth Analysis")
plt.show()

# 绘制带宽分布直方图
```

```
plt.hist(df_bandwidth.bandwidth, bins=100)
plt.xlabel("Bandwidth")
plt.ylabel("Frequency")
plt.title("Bandwidth Distribution")
plt.show()

print(min_bandwidth)
print(max_bandwidth)
print(mean_bandwidth)
```

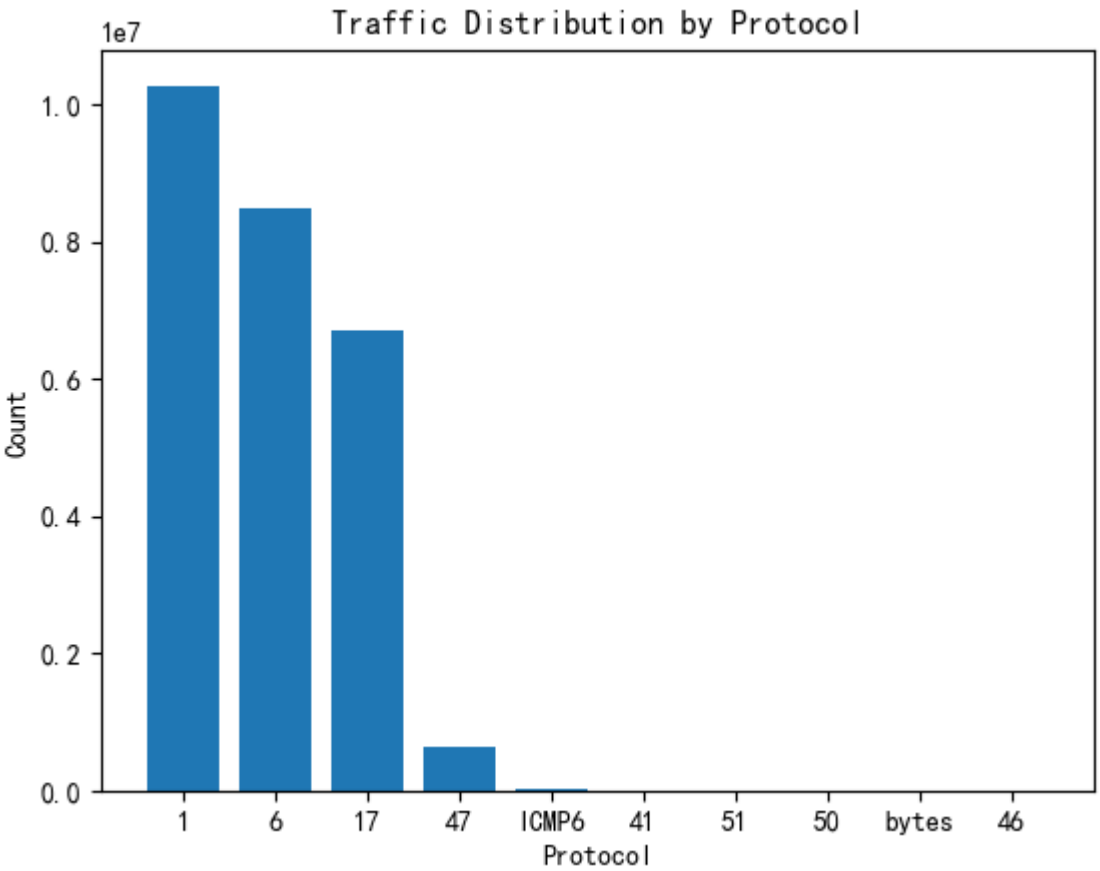





```
102447.08045977012
100663302.00000036
16770185.483744103
```

> (6). 根据协议类型进行流量分析¶
>
> > 使用 value_counts() 函数对 Protocol 列进行计数，以计算每个协议的流量分布

```
# 计算流量分布
df_protocol = trace.Protocol.value_counts()
# 绘制流量分布柱状图
plt.bar(df_protocol.index, df_protocol.values)
plt.xlabel("Protocol")
plt.ylabel("Count")
plt.title("Traffic Distribution by Protocol")
plt.show()
```



> (7). 统计一段固定时间T内达到的数据包个数

```
trace.Time[(trace.Time >= 10) & (trace.Time <= 20)].count()
```

```
1036584
```

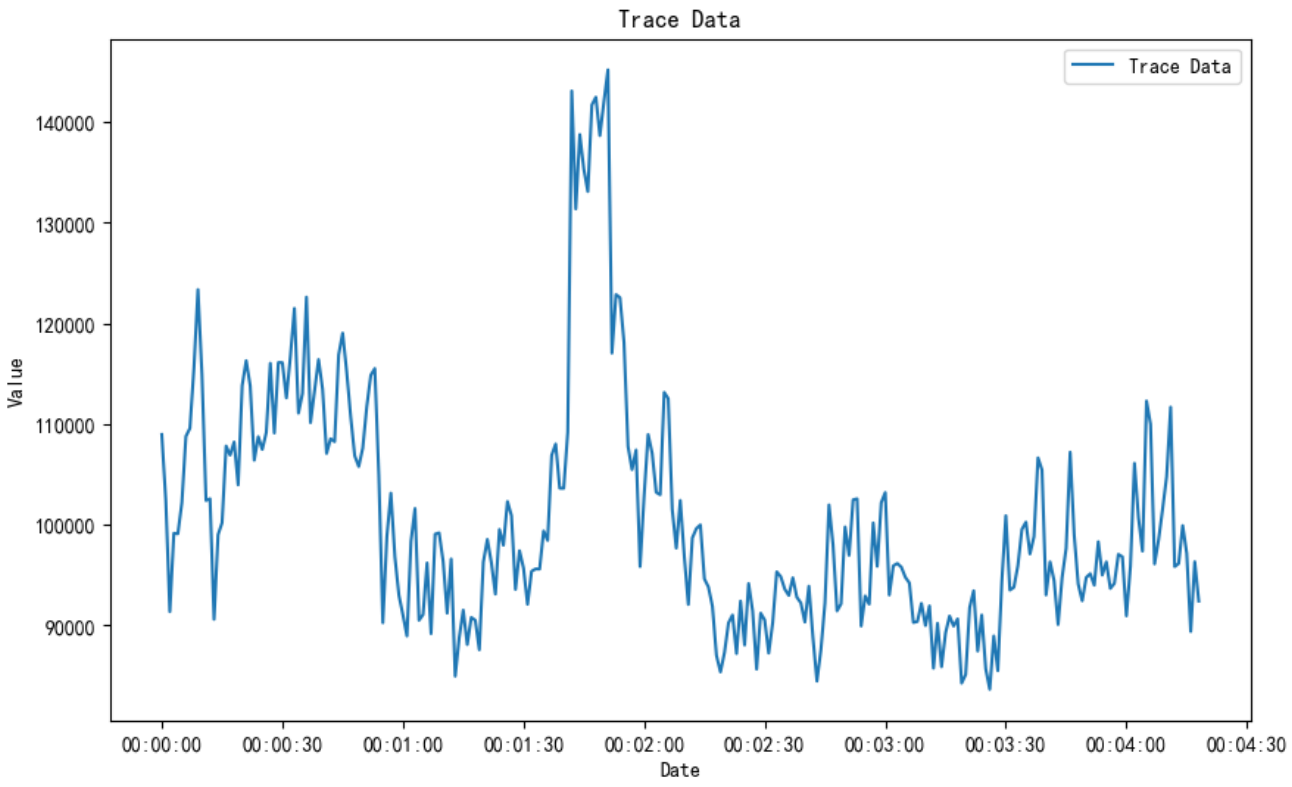
实验步骤二：

- ①将步骤一中统计的一段时间T内达到的数据包个数作为ARMA预测算法的输入，在时间序列上预测下一个时间T内可能达到的数据包个数；
- ②使用RMSE、MAE等方法计算预测结果和统计的T内真实数据；
- ③画出预测结果和真实数据的时间序列图，以此评估、比较预测的效果，从理论上来说，误差越小，预测的效果越好。

> (1)设置时间段T为 1 s，使用ARMA预测算法在时间序列上预测下一个时间T内可能到达的数据包个数

```
# 将Time列转换为datetime类型，设置为dataframe类型
trace_time = pd.DataFrame(pd.to_datetime(trace.Time, unit='s', origin='2024-01-01'))
# 设置Time列为索引
trace_time.set_index("Time", inplace=True)
spell = 1
# 计算每段时间spell到达的数据包个数
trace_time_1 = pd.Series(trace_time.resample(f"{spell}S").size())
# 由于resample会生成与原索引相邻的时间点，所以通常最后一个计数点是不需要的（因为它代表的是不完整的时间间隔）
# 因此，这里移除最后一个计数值
time_count_1 = trace_time_1[:-1]
```

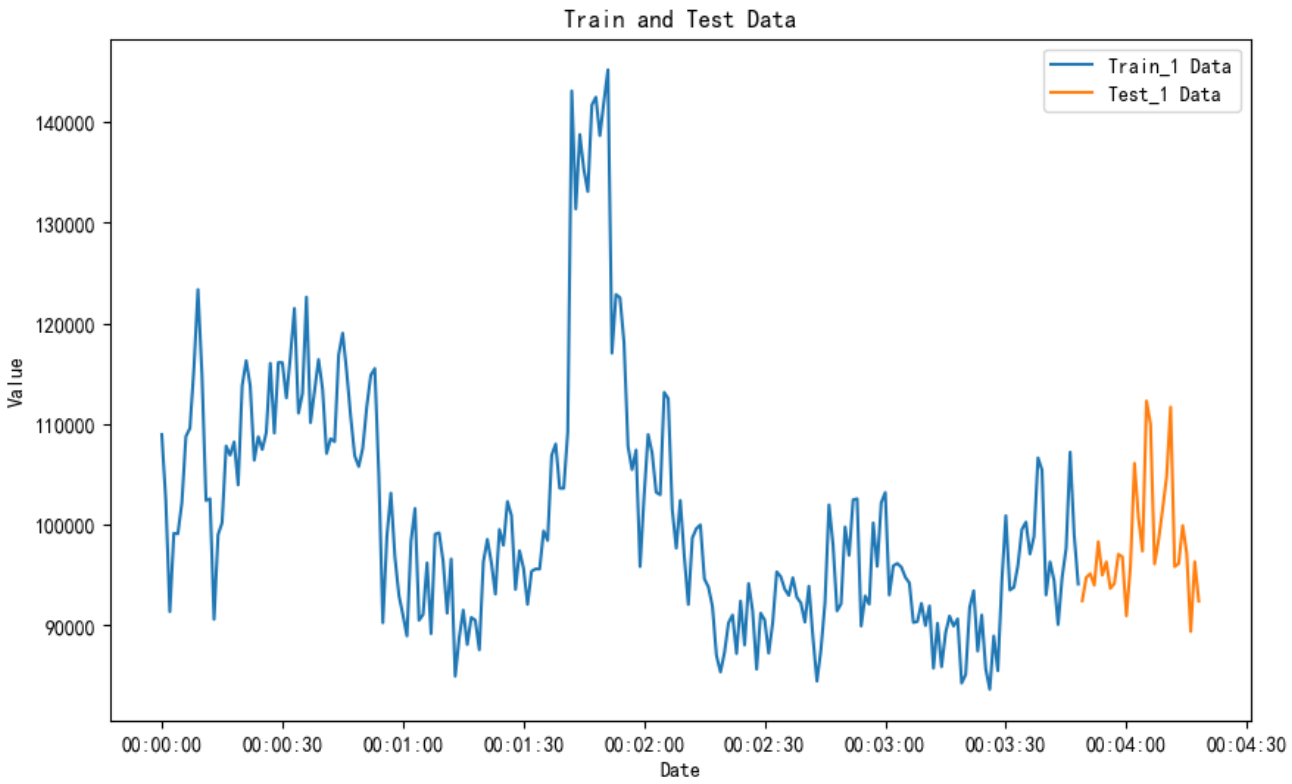
```
plt.figure(figsize=(10, 6))  # 设置图表大小为宽10英寸，高6英寸
plt.plot(time_count_1.index, time_count_1, label='Trace Data')  # 绘制时间序列数据，使用索引作为X轴（日期），值作为Y轴
plt.xlabel('Date')  # X轴标签设置为'Date'
plt.ylabel('Value')  # Y轴标签设置为'Value'
plt.title('Trace Data')  # 图表标题设置为'Trace Data'
plt.legend()  # 显示图例
plt.show()  # 显示图表
```



> 划分训练集和测试集(由于ARMA计算不适合长期预测，所以这里测试集数据划分的数量会较少一些)

```
train_1 = time_count_1[:-30]  # 训练集包含除了最后30个数据点以外的所有数据
test_1 = time_count_1[-30:]   # 测试集包含最后30个数据点
```

```
plt.figure(figsize=(10, 6))  # 设置图形大小为宽10英寸，高6英寸
plt.plot(train_1, label='Train_1 Data')  # 绘制训练集数据，使用索引作为X轴（日期），值作为Y轴
plt.plot(test_1, label='Test_1 Data')  # 绘制测试集数据，同样使用索引作为X轴，值作为Y轴
plt.xlabel('Date')  # X轴标签设置为'Date'
plt.ylabel('Value')  # Y轴标签设置为'Value'
plt.title('Train and Test Data')  # 图表标题更改为'Train and Test Data'
plt.legend()  # 显示图例
plt.show()  # 显示图表
```




```
# 进行Augmented Dickey-Fuller (ADF) 单位根检验。
# 单位根检验是用来判断时间序列数据是否平稳（即是否存在趋势或随机波动）的一种统计方法。

# 对 train 准备进行单位根检验的时间序列数据
result = sm.tsa.stattools.adfuller(train_1)

# 输出 ADF 检验的结果
print('ADF Statistic: %f' % result[0])
print('p-value: %f' % result[1])
for key, value in result[4].items():
    print('\t%s: %.3f' % (key, value))

# 这里，adfuller()函数返回一个包含多个值的元组，其中：
# -第0个元素是ADF统计量。
# -第1个元素是p值，用来判断序列是否具有单位根（通常如果p值小于0.05，则拒绝原假设，认为序列是平稳的）。
# -第4个元素是一个字典，包含了临界值信息，可以用来与ADF统计量比较以做出决策。
```

```
ADF Statistic: -2.515847
p-value: 0.111666
    1%: -3.461
    5%: -2.875
    10%: -2.574
```

> 因为 -2.515847 都大于 p-value 的三个指标值，所以可以预料的是不使用差分的话，即单单使用ARMA算法进行预测得到的拟合结果并不能较好的满意
> 但是根据题目要求选择使用ARMA如下：

```
# 如果某个滞后阶数的p值小于显著性水平（例如0.05），则可能表明在该滞后阶上存在自相关性
sm.stats.diagnostic.acorr_ljungbox(train_1, 20)
```

	lb_stat	lb_pvalue
1	178.114261	1.250736e-40
2	326.806837	1.083420e-71
3	463.800610	3.334389e-100
4	584.014026	4.465228e-125
5	691.733361	3.010358e-147
6	788.927820	3.799155e-167
7	870.520759	1.113298e-183
8	936.486967	7.594309e-197
9	992.961051	5.679724e-208
10	1034.399993	7.255770e-216
11	1073.372282	3.081097e-223
12	1108.640878	8.039764e-230
13	1133.498610	3.569135e-234
14	1151.323349	5.029503e-237
15	1166.711026	2.307869e-239
16	1179.911131	3.048126e-241
17	1190.338580	1.546570e-242
18	1199.963641	1.139227e-243
19	1205.452329	6.308379e-244
20	1208.894850	9.349461e-244

> p 值远远小于 0.05 ，说明了在滞后阶上具备自相关性，即便是 1 阶，则AR模型可以成立

> 使用auto_arima()寻找最佳参数，并获取最终确定的ARMA模型阶数(这里仅仅用于参考)

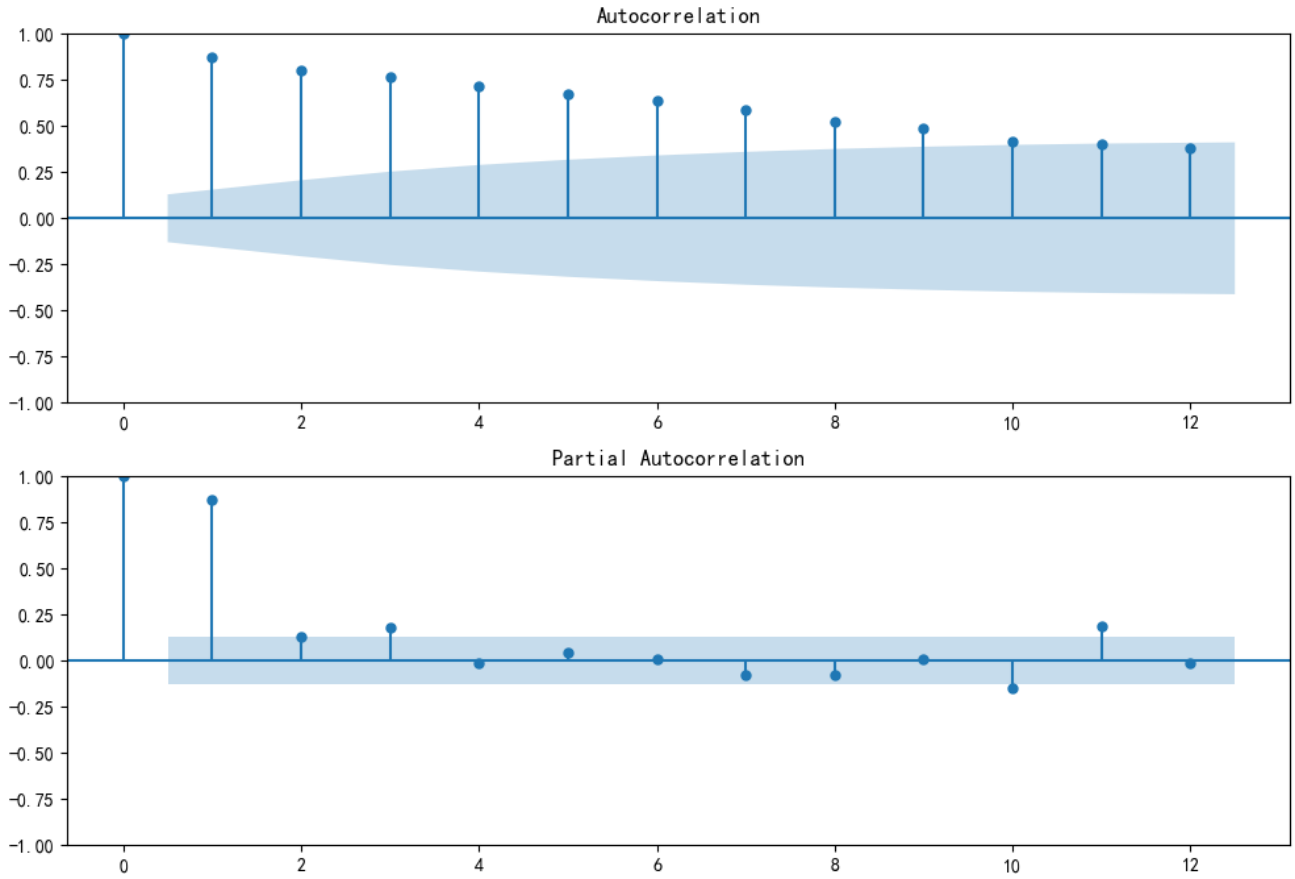
```
model_auto = auto_arima(train_1, seasonal=False, trace=True, d=0)
# 这里的seasonal=False表示我们正在处理非季节性时间序列数据，trace=True会输出一个表格，
# 显示不同ARIMA模型的AIC (Akaike Information Criterion) 或其他指定信息准则值，
# 从而帮助我们观察模型选择过程。
# 最后，通过model_auto.order可以查看自动选择的最佳ARIMA模型的阶数 (p,d,q) 。
model_auto.order
```

```
Performing stepwise search to minimize aic
 ARIMA(2,0,2)(0,0,0)[0]             : AIC=4638.023, Time=0.26 sec
 ARIMA(0,0,0)(0,0,0)[0]             : AIC=5932.691, Time=0.02 sec
 ARIMA(1,0,0)(0,0,0)[0]             : AIC=inf, Time=0.03 sec
 ARIMA(0,0,1)(0,0,0)[0]             : AIC=5769.695, Time=0.05 sec
 ARIMA(1,0,2)(0,0,0)[0]             : AIC=4636.815, Time=0.08 sec
 ARIMA(0,0,2)(0,0,0)[0]             : AIC=5732.941, Time=0.08 sec
 ARIMA(1,0,1)(0,0,0)[0]             : AIC=4640.780, Time=0.06 sec
 ARIMA(1,0,3)(0,0,0)[0]             : AIC=4638.384, Time=0.25 sec
 ARIMA(0,0,3)(0,0,0)[0]             : AIC=5723.913, Time=0.14 sec
 ARIMA(2,0,1)(0,0,0)[0]             : AIC=4653.970, Time=0.33 sec
 ARIMA(2,0,3)(0,0,0)[0]             : AIC=4639.162, Time=0.41 sec
 ARIMA(1,0,2)(0,0,0)[0] intercept   : AIC=4628.442, Time=0.28 sec
 ARIMA(0,0,2)(0,0,0)[0] intercept   : AIC=4820.777, Time=0.09 sec
 ARIMA(1,0,1)(0,0,0)[0] intercept   : AIC=4630.205, Time=0.07 sec
 ARIMA(2,0,2)(0,0,0)[0] intercept   : AIC=4628.620, Time=0.26 sec
 ARIMA(1,0,3)(0,0,0)[0] intercept   : AIC=4629.237, Time=0.16 sec
 ARIMA(0,0,1)(0,0,0)[0] intercept   : AIC=4792.904, Time=0.19 sec
 ARIMA(0,0,3)(0,0,0)[0] intercept   : AIC=4803.653, Time=0.13 sec
 ARIMA(2,0,1)(0,0,0)[0] intercept   : AIC=4631.472, Time=0.27 sec
 ARIMA(2,0,3)(0,0,0)[0] intercept   : AIC=4630.209, Time=0.60 sec

Best model:  ARIMA(1,0,2)(0,0,0)[0] intercept
Total fit time: 3.795 seconds
```

```
(1, 0, 2)
```

> 接下来进行对自相关系数图和偏自相关系数图分析，以确定最后的AR和MA的阶数

```
fig = plt.figure(figsize=(12,8))
ax1 = fig.add_subplot(211)
fig = sm.graphics.tsa.plot_acf(train_1.values.squeeze(), lags=12, ax=ax1)  # 自相关系数图，截断在12阶滞后
ax2 = fig.add_subplot(212)
fig = sm.graphics.tsa.plot_pacf(train_1, lags=12, ax=ax2) # 偏自相关系数图，截断在12阶滞
```



> 由 (1, 0, 2) 和 (11, 0, 2) 综合可得应选参数 (11, 0, 2)

```
model_1 = sm.tsa.ARIMA(train_1, order=(11, 0, 2))
model_fit_1 = model_1.fit()
# 模型预测
prediction_1 = model_fit_1.forecast(steps=len(test_1))
fit_1 = model_fit_1.predict(start=1, end=len(train_1))[1:]
```

> (2) 使用RMSE、MAE等方法计算预测结果和统计的T内真实数据;

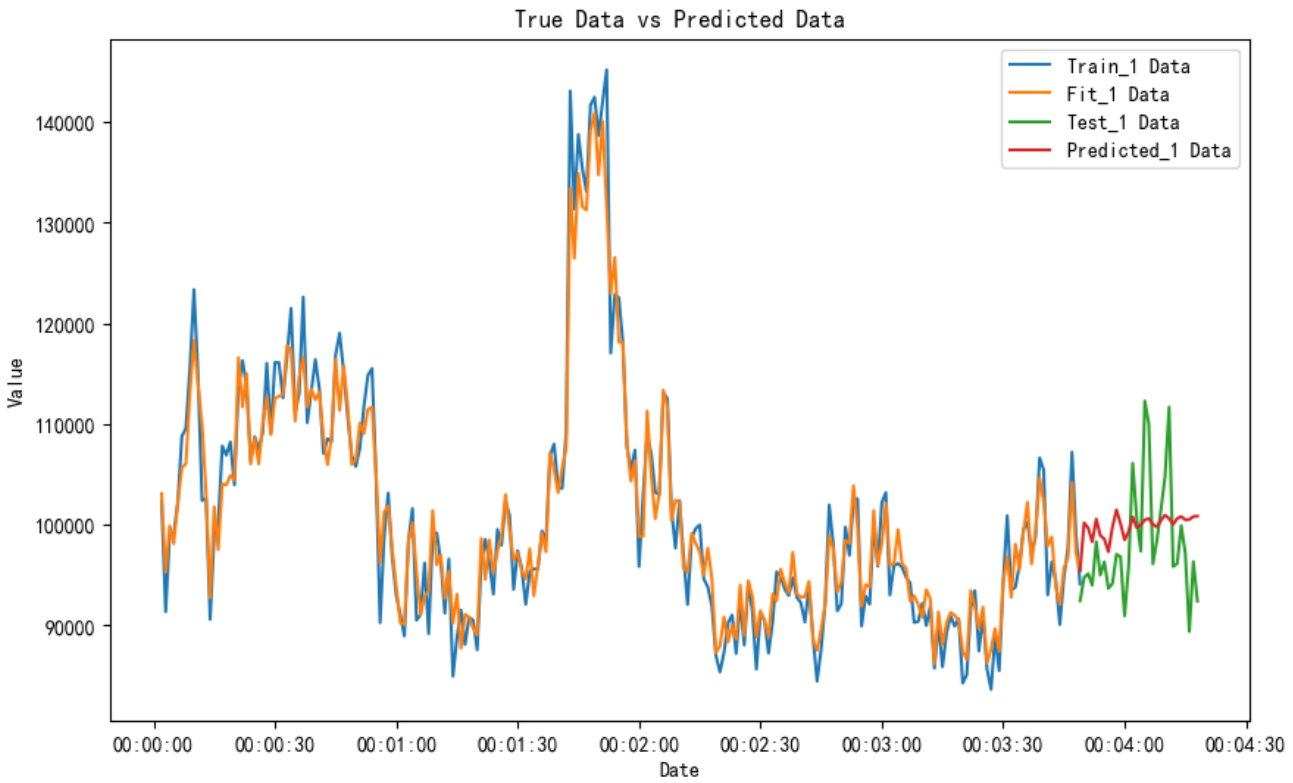
```
# 计算RMSE（均方根误差）
rmse = np.sqrt(mean_squared_error(test_1, prediction_1))
# 计算MAE（平均绝对误差）
mae = mean_absolute_error(test_1, prediction_1)
# 输出RMSE, MAE
print("数据包的RMSE：", rmse)
print("数据包的MAE：", mae)
```

```
数据包的RMSE：  5569.31914926447
数据包的MAE：  4731.689114840352
```

> RMSE为5569，意味着模型的平均预测误差约为5569个单位。
> MAE为4731，表示模型的平均绝对误差约为4731个单位。

(3) 画出预测结果和真实数据的时间序列图，以此评估、比较预测的效果，从理论上来说，误差越小，预测的效果越好

```
# 绘制时间序列图
plt.figure(figsize=(10, 6))
plt.plot(fit_1.index, train_1.values[1:], label = 'Train_1 Data')
plt.plot(fit_1.index, fit_1.values, label = 'Fit_1 Data')
plt.plot(prediction_1.index, test_1.values, label='Test_1 Data')
plt.plot(prediction_1.index, prediction_1.values, label='Predicted_1 Data')

plt.xlabel('Date')
plt.ylabel('Value')
plt.title('True Data vs Predicted Data')
plt.legend()
plt.show()
```



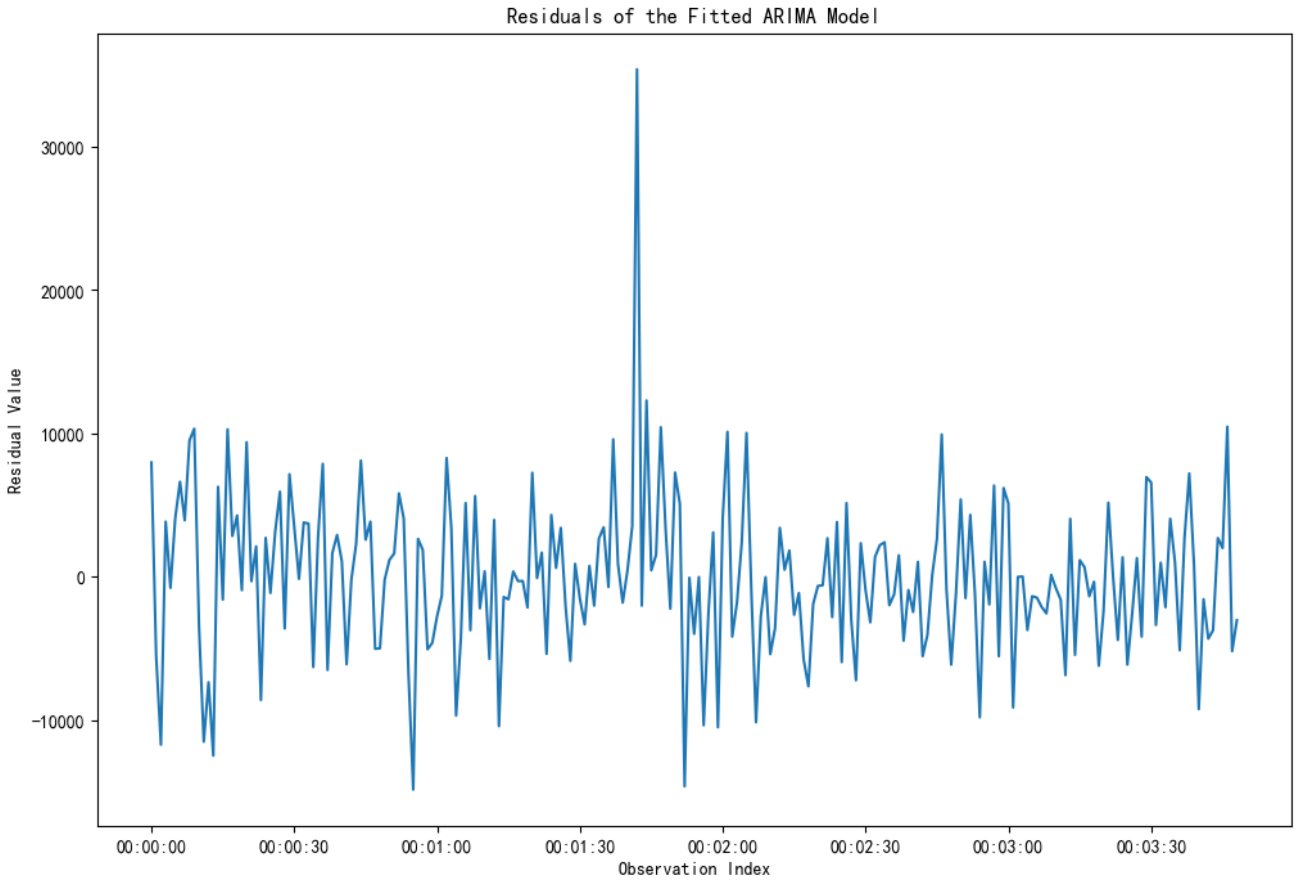
> 根据结果显示
>
> - 短期预测曲线走向跟测试集数据前部分是一致的
> - 长期预测曲线会趋于均值与真实曲线走向不一致
> - 而且全过程预测曲线整体数值偏高了一些

> 对拟合数据（训练集）进行残差分析

```
resid_1 = model_fit_1.resid  # 获取残差

# 可视化残差
plt.figure(figsize=(12, 8))
```

```
plt.plot(resid_1)
plt.title('Residuals of the Fitted ARIMA Model')
plt.xlabel('Observation Index')
plt.ylabel('Residual Value')

# 计算残差的平均值
mean_residual = np.mean(resid_1)
print("Mean Residual: ", mean_residual)

plt.show()
```

```
Mean Residual:  -18.770060664379876
```

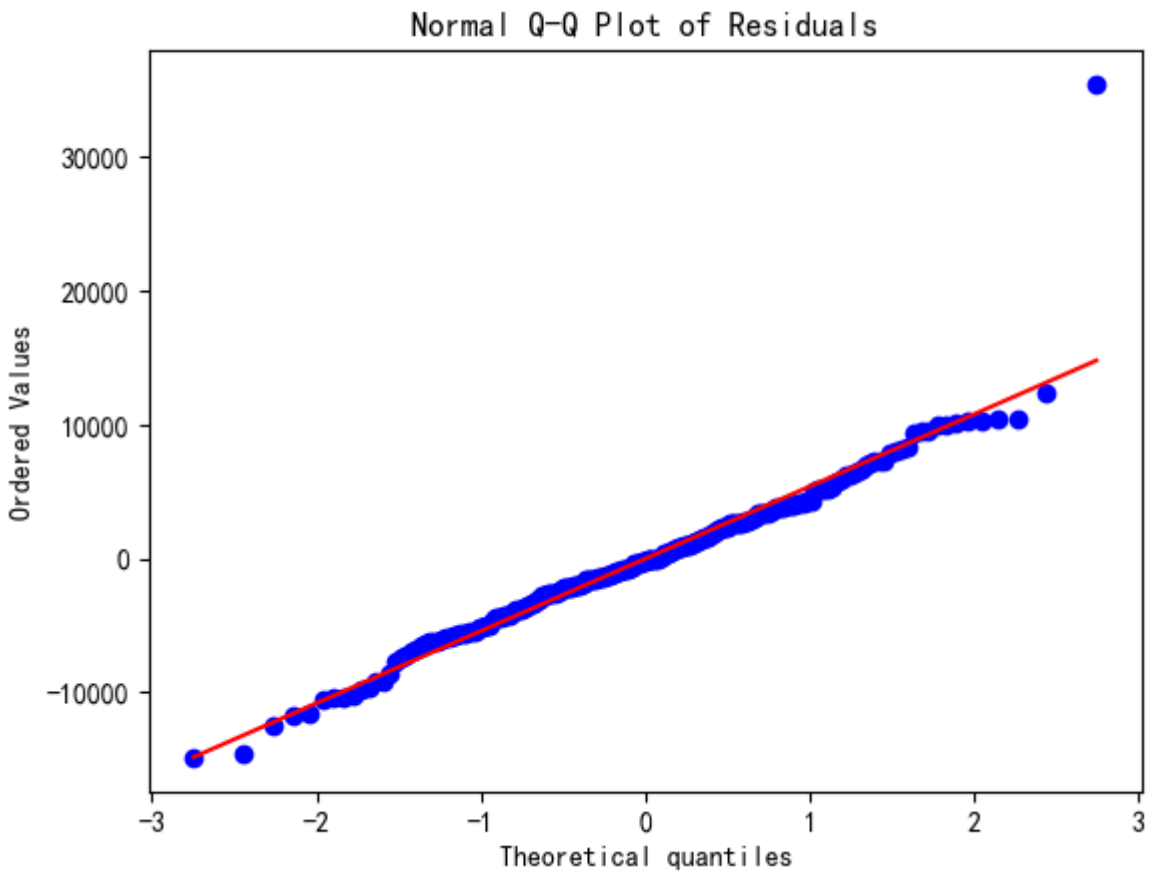


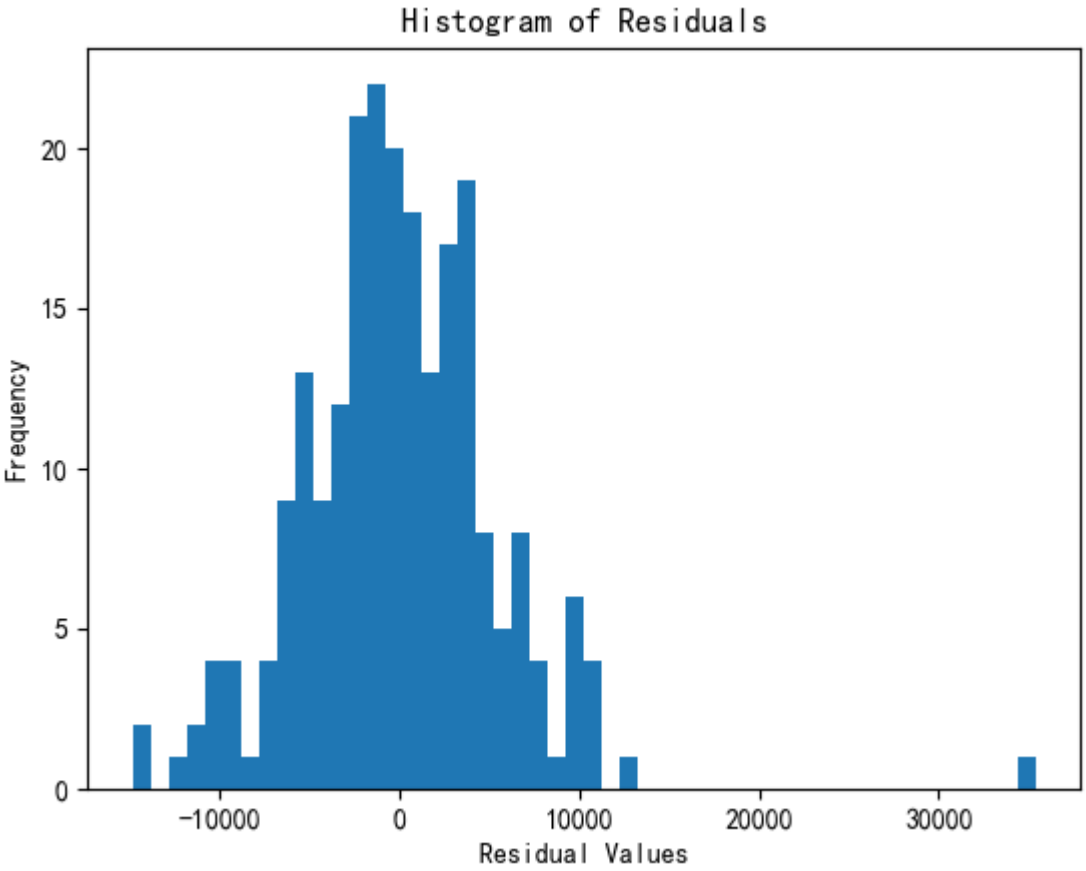
> 残差正态性检验

```
#检验序列残差是否为正态分布     pvalue<0.05  拒绝原假设 认为残差符合正太分布
stat, pvalue = stats.normaltest(resid_1)

if pvalue >= 0.05:
    print("残差不服从正态分布 (p-value: {:.3f})".format(pvalue))
else:
    print("残差服从正态分布 (p-value: {:.3f})".format(pvalue))

stats.probplot(resid_1, dist="norm", plot=plt)
plt.title('Normal Q-Q Plot of Residuals')
```

```
plt.show()

# 绘制残差的直方图，设置50个bins（区间）
plt.hist(resid_1, bins=50)
plt.xlabel('Residual Values')
plt.ylabel('Frequency')
plt.title('Histogram of Residuals')
plt.show()
```

```
残差服从正态分布（p-value: 0.000）
```





> 残差序列自相关 （残差序列是否独立）

```
# 残差序列自相关 （残差序列是否独立）
from statsmodels.stats.stattools import durbin_watson
durbin_watson(resid_1.values)      ##DW检验：靠近2—正常；靠近0—正自相关；靠近4—负自相关
```

```
1.970927694717104
```

> 综上残差的分析可以得出 正态分布的残差和残差序列的独立：
>
> - 表明模型能够很好地捕捉数据中的随机性和噪声，没有遗漏重要的信息。这可以作为模型选择和评估的一个指标，表明所选择的模型对数据的拟合程度较好。
> - 基于这个模型的预测也具有较高的准确性。因为正态分布的性质使得预测区间的计算更可靠，可以提供关于未来观测值的置信区间。

实验步骤三：

- 在步骤二的基础上，改变统计时间T的大小，重复步骤二并给出相应的分析结论
- 统计、打印信息等操作参考《基于NS3平台完成实验目标4-5》中第二、三部分

> (1) 设置统计时间T spell=2，重复步骤二

```
spell = 2
# 计算每段时间spell到达的数据包个数
trace_time_2 = pd.Series(trace_time.resample(f"{spell}S").size())
# 由于resample会生成与原索引相邻的时间点，所以通常最后一个计数点是不需要的（因为它代表的是不完整的时间间隔）
# 因此，这里移除最后一个计数值
time_count_2 = trace_time_2[:-1]
train_2 = time_count_2[:-15]  # 训练集包含除了最后15个数据点以外的所有数据
test_2 = time_count_2[-16:]   # 测试集包含最后16个数据点
# 如果某个滞后阶数的p值小于显著性水平（例如0.05），则可能表明在该滞后阶上存在自相关性
sm.stats.diagnostic.acorr_ljungbox(train_2, 20)
```

	lb_stat	lb_pvalue
1	85.364345	2.481552e-20
2	151.863264	1.055143e-33
3	205.700826	2.473360e-44
4	242.447718	2.756150e-51
5	266.741316	1.401808e-55
6	284.986457	1.344717e-58
7	295.152401	6.558498e-60
8	302.373193	1.286855e-60
9	307.231219	7.628003e-61
10	309.124472	1.828049e-60
11	309.798590	7.536595e-60
12	309.829340	4.044345e-59
13	310.664420	1.407177e-58
14	313.623339	1.695608e-58
15	317.995250	1.007217e-58
16	325.588984	1.251013e-59
17	334.597288	7.796766e-61
18	343.381837	5.353456e-62
19	353.089744	2.346888e-63
20	361.581095	1.821910e-64

> p 值远远小于 0.05（或 0.01）说明了该时间序列的滞后阶上存在自相关性

> 使用auto_arima()寻找最佳参数，并获取最终确定的ARMA模型阶数(这里仅仅用于参考)

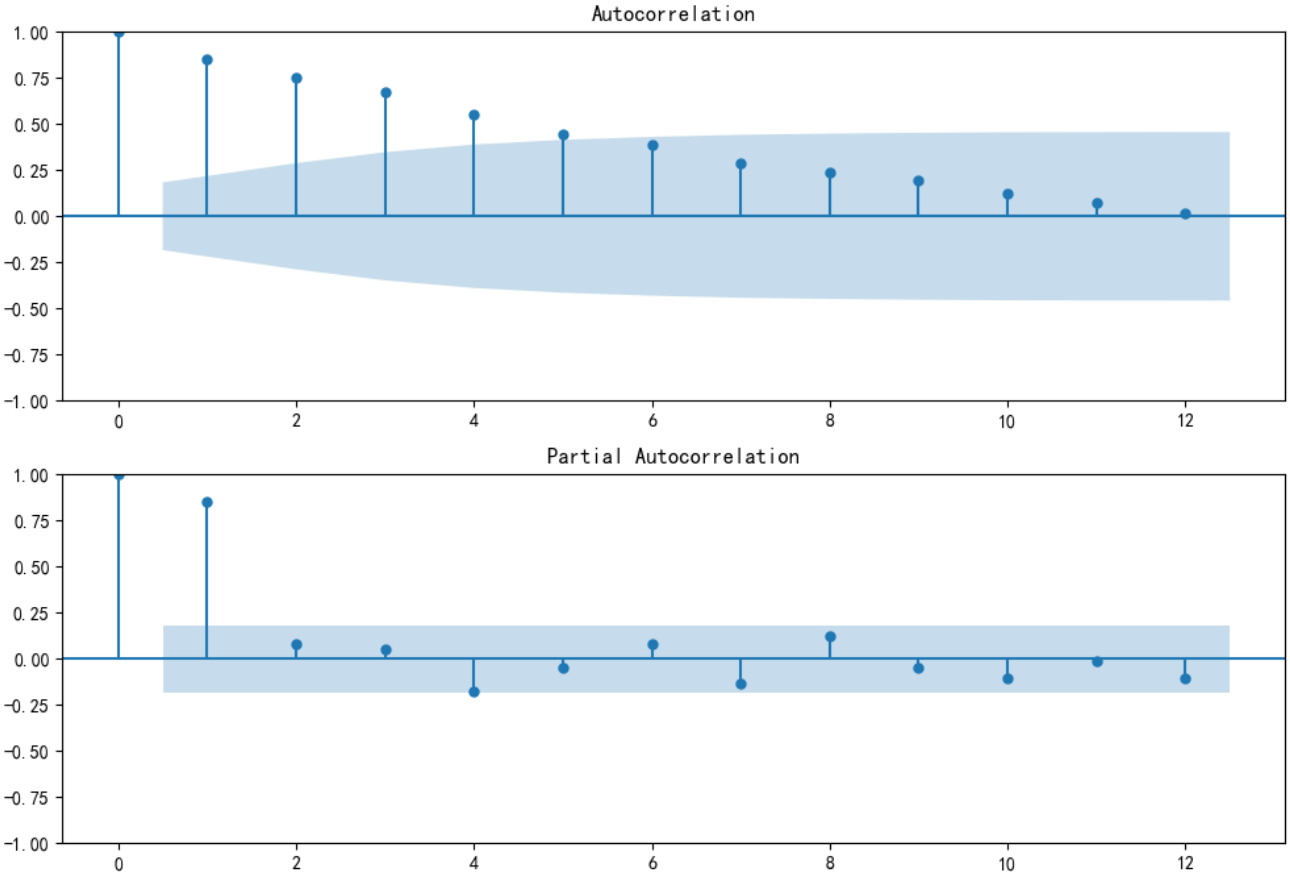
```
model_auto = auto_arima(train_2, seasonal=False, trace=True, d=0)
# 这里的seasonal=False表示我们正在处理非季节性时间序列数据，trace=True会输出一个表格，
# 显示不同ARIMA模型的AIC (Akaike Information Criterion) 或其他指定信息准则值，
# 从而帮助我们观察模型选择过程。
# 最后，通过model_auto.order可以查看自动选择的最佳ARIMA模型的阶数 (p,d,q) 。
model_auto.order
```

```
Performing stepwise search to minimize aic
 ARIMA(2,0,2)(0,0,0)[0]             : AIC=2496.572, Time=0.14 sec
 ARIMA(0,0,0)(0,0,0)[0]             : AIC=3112.426, Time=0.01 sec
 ARIMA(1,0,0)(0,0,0)[0]             : AIC=inf, Time=0.04 sec
 ARIMA(0,0,1)(0,0,0)[0]             : AIC=3033.860, Time=0.06 sec
 ARIMA(1,0,2)(0,0,0)[0]             : AIC=2493.996, Time=0.17 sec
 ARIMA(0,0,2)(0,0,0)[0]             : AIC=3016.375, Time=0.08 sec
 ARIMA(1,0,1)(0,0,0)[0]             : AIC=2492.282, Time=0.04 sec
 ARIMA(2,0,1)(0,0,0)[0]             : AIC=2494.197, Time=0.09 sec
 ARIMA(2,0,0)(0,0,0)[0]             : AIC=inf, Time=0.11 sec
 ARIMA(1,0,1)(0,0,0)[0] intercept   : AIC=2482.498, Time=0.08 sec
 ARIMA(0,0,1)(0,0,0)[0] intercept   : AIC=2570.777, Time=0.06 sec
 ARIMA(1,0,0)(0,0,0)[0] intercept   : AIC=2481.084, Time=0.03 sec
 ARIMA(0,0,0)(0,0,0)[0] intercept   : AIC=2627.292, Time=0.03 sec
 ARIMA(2,0,0)(0,0,0)[0] intercept   : AIC=2482.579, Time=0.08 sec
 ARIMA(2,0,1)(0,0,0)[0] intercept   : AIC=2484.491, Time=0.14 sec

Best model:  ARIMA(1,0,0)(0,0,0)[0] intercept
Total fit time: 1.183 seconds
```

```
(1, 0, 0)
```

> 接下来进行对自相关系数图和偏自相关系数图分析，以确定最后的AR和MA的阶数

```
fig = plt.figure(figsize=(12,8))
ax1 = fig.add_subplot(211)
fig = sm.graphics.tsa.plot_acf(train_2.values.squeeze(), lags=12, ax=ax1)  # 自相关系数图，截断在10阶滞后
ax2 = fig.add_subplot(212)
fig = sm.graphics.tsa.plot_pacf(train_2, lags=12, ax=ax2) # 偏自相关系数图，截断在10阶滞
```



> 综上，由 (1, 0, 0) 和 (5, 0, 1) 可选择 (5, 0, 1)

```
model_2 = sm.tsa.ARIMA(train_2, order=(5, 0, 1))
model_fit_2 = model_2.fit()
# 模型预测
prediction_2 = model_fit_2.forecast(steps=len(test_2))
fit_2 = model_fit_2.predict(start=1, end=len(train_2))[1:]
```

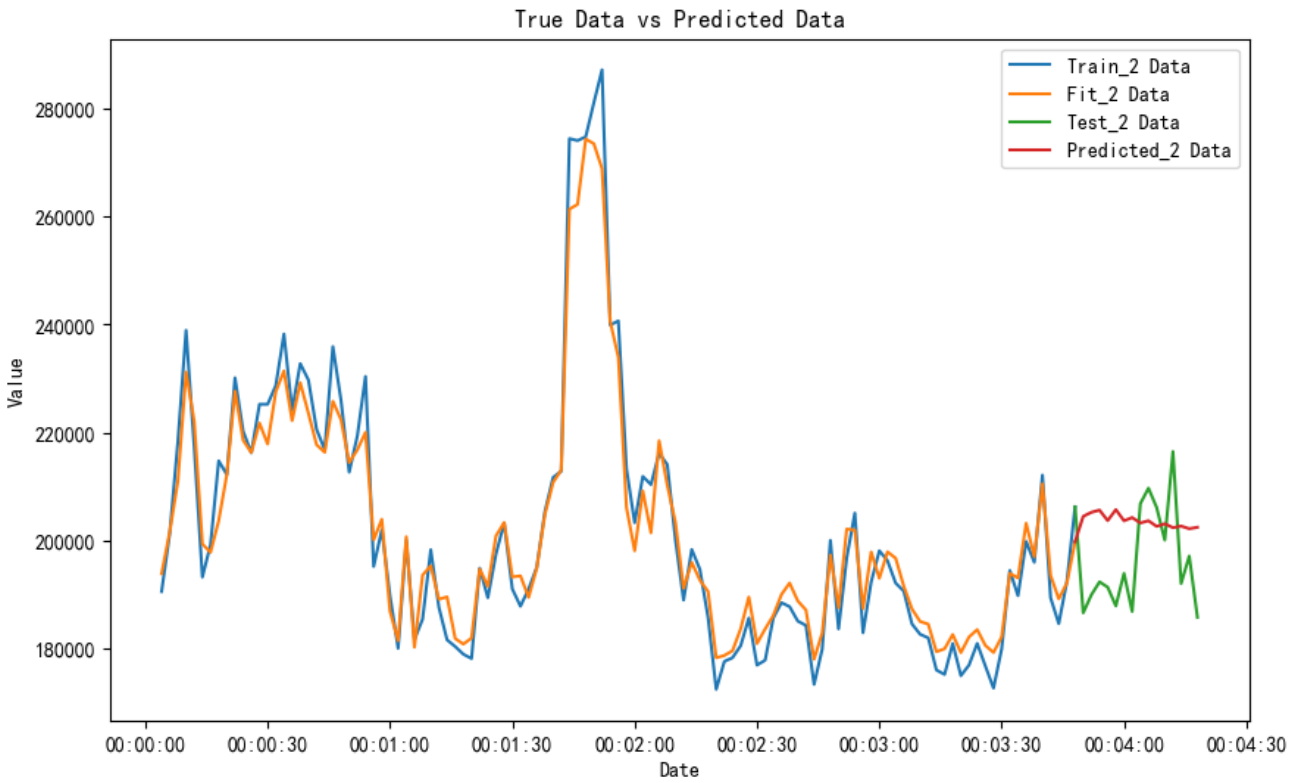
> 使用RMSE、MAE等方法计算预测结果和统计的T内真实数据；

```
# 计算RMSE（均方根误差）
rmse = np.sqrt(mean_squared_error(test_2, prediction_2))
# 计算MAE（平均绝对误差）
mae = mean_absolute_error(test_2, prediction_2)
# 输出RMSE, MAE
print("数据包的RMSE：", rmse)
print("数据包的MAE：", mae)
```

```
数据包的RMSE：  12063.286047636264
数据包的MAE：  10807.82034692179
```

> 画出预测结果和真实数据的时间序列图，以此评估、比较预测的效果，从理论上来说，误差越小，预测的效果越好

```
# 绘制时间序列图
plt.figure(figsize=(10, 6))
plt.plot(fit_2.index, train_2.values[1:], label = 'Train_2 Data')
plt.plot(fit_2.index, fit_2.values, label = 'Fit_2 Data')
plt.plot(prediction_2.index, test_2.values, label='Test_2 Data')
plt.plot(prediction_2.index, prediction_2.values, label='Predicted_2 Data')

plt.xlabel('Date')
plt.ylabel('Value')
plt.title('True Data vs Predicted Data')
plt.legend()
plt.show()
```



> (2) 设置统计时间T spell=3，重复步骤二

```
spell = 3
# 计算每段时间spell到达的数据包个数
trace_time_3 = pd.Series(trace_time.resample(f"{spell}S").size())
# 由于resample会生成与原索引相邻的时间点，所以通常最后一个计数点是不需要的（因为它代表的是不完整的时间间隔）
# 因此，这里移除最后一个计数值
time_count_3 = trace_time_3[:-1]
train_3 = time_count_3[:-10]  # 训练集包含除了最后10个数据点以外的所有数据
test_3 = time_count_3[-11:]   # 测试集包含最后11个数据点
# 如果某个滞后阶数的p值小于显著性水平（例如0.05），则可能表明在该滞后阶上存在自相关性
sm.stats.diagnostic.acorr_ljungbox(train_3, 20)
```

	lb_stat	lb_pvalue
1	55.242021	1.065660e-13
2	95.302452	2.019926e-21
3	117.466474	2.710362e-25
4	130.908694	2.489211e-27
5	137.120696	7.320810e-28
6	140.626303	7.392324e-28
7	141.525286	2.435024e-27
8	141.532015	1.139131e-26
9	142.799168	2.724922e-26
10	146.126995	2.331392e-26
11	152.144325	5.433126e-27
12	158.848394	9.014672e-28
13	165.535263	1.489229e-28
14	171.213149	3.894082e-29
15	175.266029	2.136611e-29
16	178.666548	1.558530e-29
17	179.765110	3.215485e-29
18	180.061229	9.326389e-29
19	180.378049	2.606449e-28
20	181.680474	4.550624e-28

> p 值远远小于 0.05（或 0.01）说明了该时间序列的滞后阶上存在自相关性

> 使用auto_arima()寻找最佳参数，并获取最终确定的ARMA模型阶数(这里仅仅用于参考)

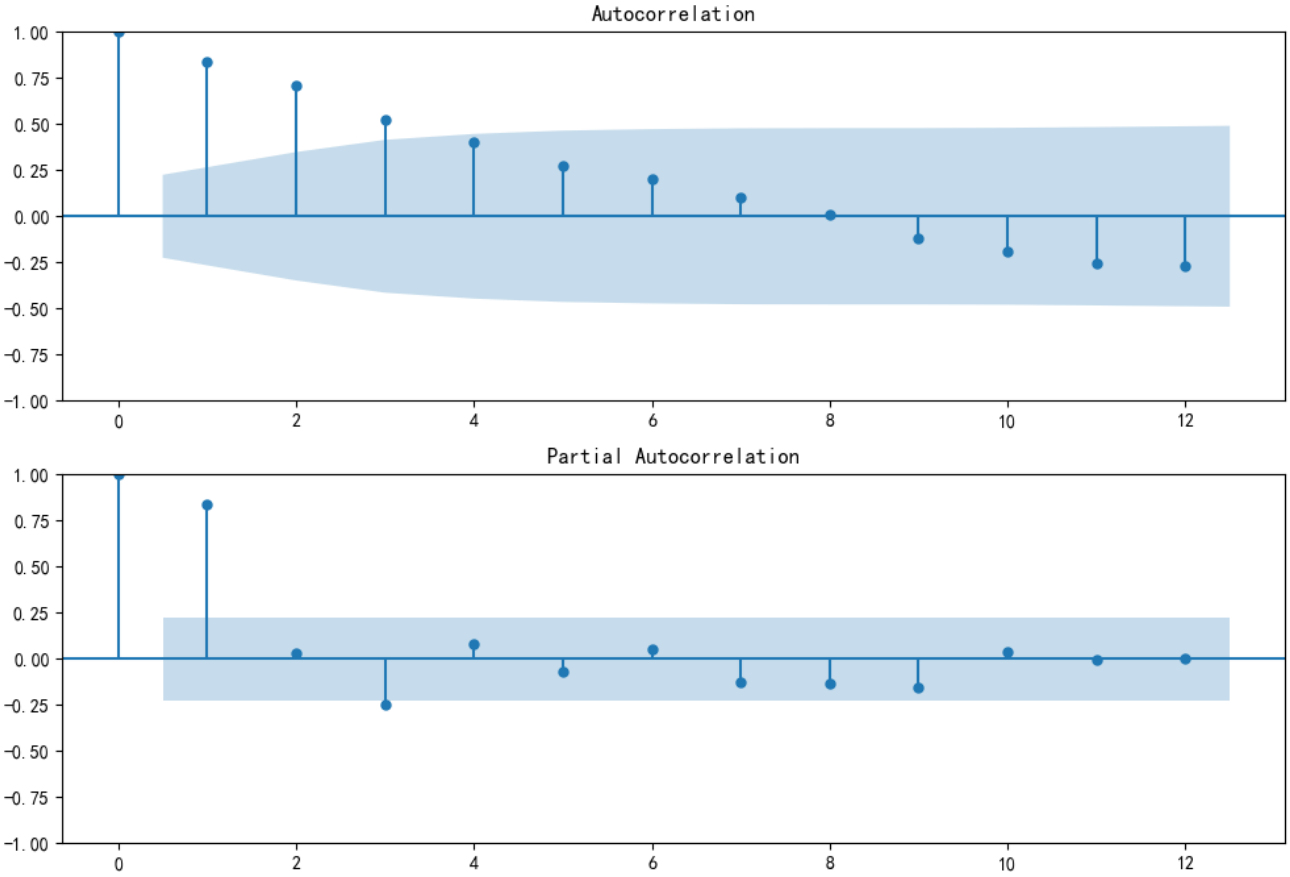
```
model_auto = auto_arima(train_3, seasonal=False, trace=True, d=0)
# 这里的seasonal=False表示我们正在处理非季节性时间序列数据，trace=True会输出一个表格，
# 显示不同ARIMA模型的AIC (Akaike Information Criterion) 或其他指定信息准则值，
# 从而帮助我们观察模型选择过程。
# 最后，通过model_auto.order可以查看自动选择的最佳ARIMA模型的阶数 (p,d,q) 。
model_auto.order
```

```
Performing stepwise search to minimize aic
 ARIMA(2,0,2)(0,0,0)[0]             : AIC=1733.251, Time=0.19 sec
 ARIMA(0,0,0)(0,0,0)[0]             : AIC=2137.196, Time=0.02 sec
 ARIMA(1,0,0)(0,0,0)[0]             : AIC=inf, Time=0.04 sec
 ARIMA(0,0,1)(0,0,0)[0]             : AIC=2086.583, Time=0.03 sec
 ARIMA(1,0,2)(0,0,0)[0]             : AIC=1733.967, Time=0.16 sec
 ARIMA(2,0,1)(0,0,0)[0]             : AIC=1731.825, Time=0.14 sec
 ARIMA(1,0,1)(0,0,0)[0]             : AIC=1733.656, Time=0.10 sec
 ARIMA(2,0,0)(0,0,0)[0]             : AIC=inf, Time=0.06 sec
 ARIMA(3,0,1)(0,0,0)[0]             : AIC=1733.170, Time=0.22 sec
 ARIMA(3,0,0)(0,0,0)[0]             : AIC=inf, Time=0.13 sec
 ARIMA(3,0,2)(0,0,0)[0]             : AIC=1735.339, Time=0.46 sec
 ARIMA(2,0,1)(0,0,0)[0] intercept   : AIC=1723.772, Time=0.20 sec
 ARIMA(1,0,1)(0,0,0)[0] intercept   : AIC=1724.619, Time=0.07 sec
 ARIMA(2,0,0)(0,0,0)[0] intercept   : AIC=1724.640, Time=0.13 sec
 ARIMA(3,0,1)(0,0,0)[0] intercept   : AIC=1723.320, Time=0.19 sec
 ARIMA(3,0,0)(0,0,0)[0] intercept   : AIC=1721.522, Time=0.09 sec
 ARIMA(4,0,0)(0,0,0)[0] intercept   : AIC=1723.072, Time=0.13 sec
 ARIMA(4,0,1)(0,0,0)[0] intercept   : AIC=1725.332, Time=0.22 sec

Best model:  ARIMA(3,0,0)(0,0,0)[0] intercept
Total fit time: 2.565 seconds
```

```
(3, 0, 0)
```

> 接下来进行对自相关系数图和偏自相关系数图分析，以确定最后的AR和MA的阶数

```
fig = plt.figure(figsize=(12,8))
ax1 = fig.add_subplot(211)
fig = sm.graphics.tsa.plot_acf(train_3.values.squeeze(), lags=12, ax=ax1)  # 自相关系数图，截断在10阶滞后
ax2 = fig.add_subplot(212)
fig = sm.graphics.tsa.plot_pacf(train_3, lags=12, ax=ax2) # 偏自相关系数图，截断在10阶滞
```



> 综上，由 (3, 0, 0) 和 (3, 0, 1) 可得 (3, 0, 1)

```
model_3 = sm.tsa.ARIMA(train_3, order=(3, 0, 1))
model_fit_3 = model_3.fit()
# 模型预测
prediction_3 = model_fit_3.forecast(steps=len(test_3))
fit_3 = model_fit_3.predict(start=1, end=len(train_3))[1:]
```

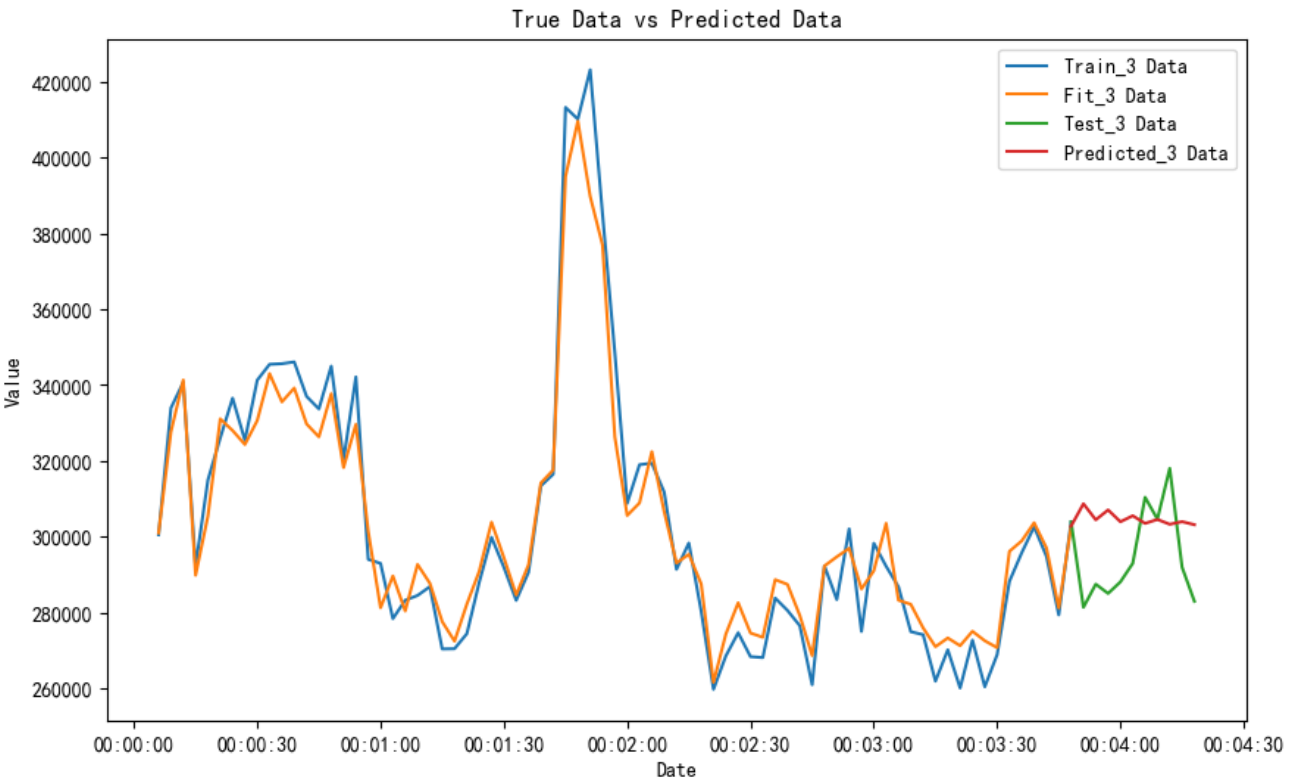
> 使用RMSE、MAE等方法计算预测结果和统计的T内真实数据；

```
# 计算RMSE（均方根误差）
rmse = np.sqrt(mean_squared_error(test_3, prediction_3))
# 计算MAE（平均绝对误差）
mae = mean_absolute_error(test_3, prediction_3)
# 输出RMSE, MAE
print("数据包的RMSE：", rmse)
print("数据包的MAE：", mae)
```

```
数据包的RMSE：  15809.65222776369
数据包的MAE：  13620.87241960981
```

> 画出预测结果和真实数据的时间序列图，以此评估、比较预测的效果，从理论上来说，误差越小，预测的效果越好

```
# 绘制时间序列图
plt.figure(figsize=(10, 6))
plt.plot(fit_3.index, train_3.values[1:], label = 'Train_3 Data')
plt.plot(fit_3.index, fit_3.values, label = 'Fit_3 Data')
plt.plot(prediction_3.index, test_3.values, label='Test_3 Data')
plt.plot(prediction_3.index, prediction_3.values, label='Predicted_3 Data')

plt.xlabel('Date')
plt.ylabel('Value')
plt.title('True Data vs Predicted Data')
plt.legend()
plt.show()
```



> 综上所示，得出结论：
>
> - 随着统计时间T的增加，模型捕捉数据中的随机性和噪声的能力下降，遗漏重要的信息更多。
> - 随着统计时间T的增加，模型预测的准确性也在降低。

> (3) 统计，打印操作具体在 scenario-ethernet-net-device.cc 文件中：

实验步骤四：

- ①从代码的角度模拟网络流量下节能以太网的模式转换过程；
- ②统计EEEP策略下的能耗和平均延时。

> 具体来说，模拟时节能以太网中到达的流量数据为处理过的trace1，它会为模拟过程提供所需要的数据包的大小、到达时间等信息，在上一周期结束后，在模拟的系统中统计上一周期内到达的数据包个数并预测下一周期达到的个数，并由此计算出下一周期内停留在低功耗状态的时间t，周期开始后首先在低功耗状态停留时间t后转至活跃状态传输数据包直到排队队列中数据包为空，周期结束，重复上述过程至流量数据在时间序列上结束。模拟结束并统计模拟过程中的平均延时、能耗等数据，由此计算节能效果和延时开销。信息统计等操作参考《基于NS3平台完成实验目标4-5》中第二、三部分

> (1) 使用流量文件designed_trace_R，并作出周期修改，导出数据到文件 TEST.txt 中，并用周期重命名文件。如下图和代码读取所示：
>
> > 这是在 scenario_File.cc 文件中：
> >
> > 这是在 switched-ethernet-net-device.cc文件中：

> (2) 模拟过程及结果如下图所示：
>
> >

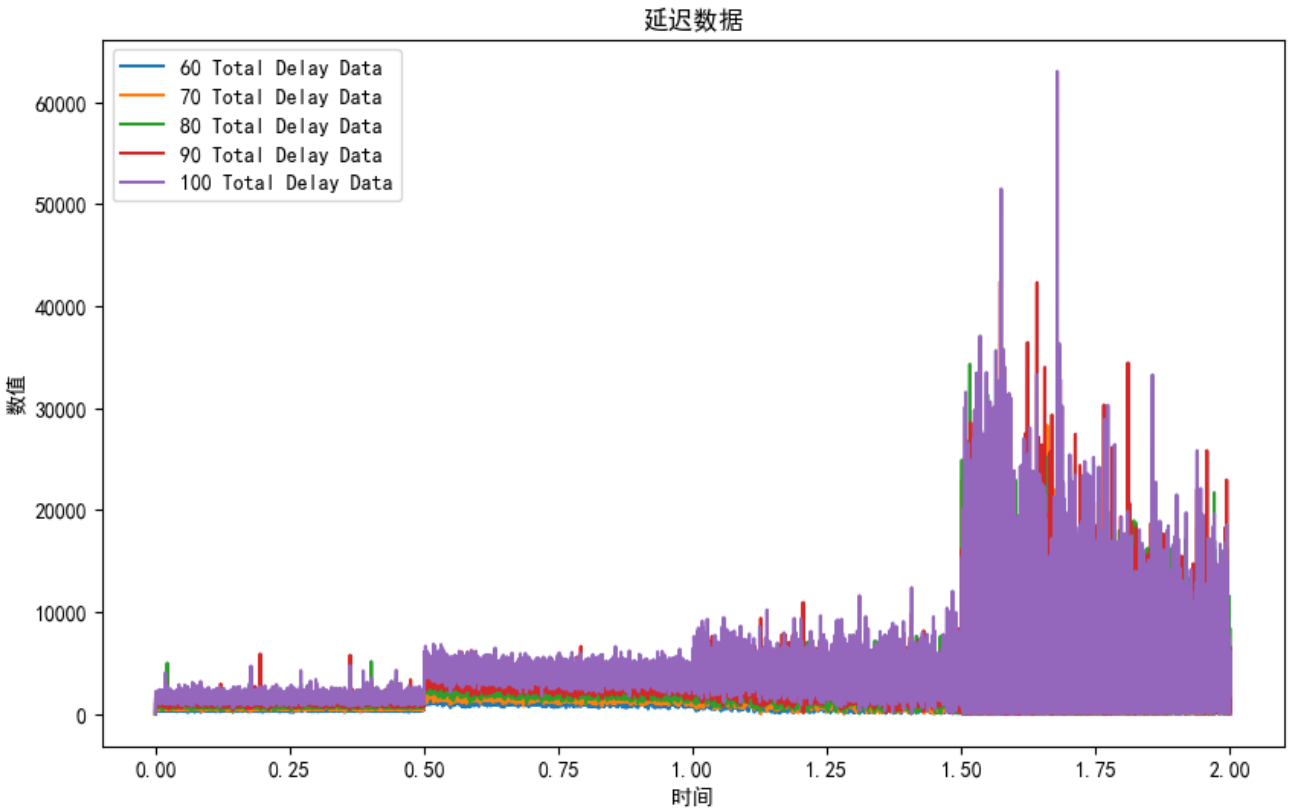
> (3) 统计模拟过程中的平均延时、能耗等数据，由此计算节能效果和延时开销，对其进行分析和可视化展示如下：

```
trace_60 = pd.read_csv('TEST_60.txt')  # 读取文件TEST_60.txt的数据并赋值给trace_60
trace_70 = pd.read_csv('TEST_70.txt')  # 读取文件TEST_70.txt的数据并赋值给trace_70
trace_80 = pd.read_csv('TEST_80.txt')  # 读取文件TEST_80.txt的数据并赋值给trace_80
trace_90 = pd.read_csv('TEST_90.txt')  # 读取文件TEST_90.txt的数据并赋值给trace_90
trace_100 = pd.read_csv('TEST_100.txt')  # 读取文件TEST_100.txt的数据并赋值给trace_100
print(trace_60.index[-1], trace_70.index[-1], trace_80.index[-1], trace_90.index[-1],
trace_100.index[-1])  # 打印最后一个索引值，即bao'shu
```

```
27302 23429 20521 18251 16437
```

> > 对每个窗口的总延迟数据进行分析

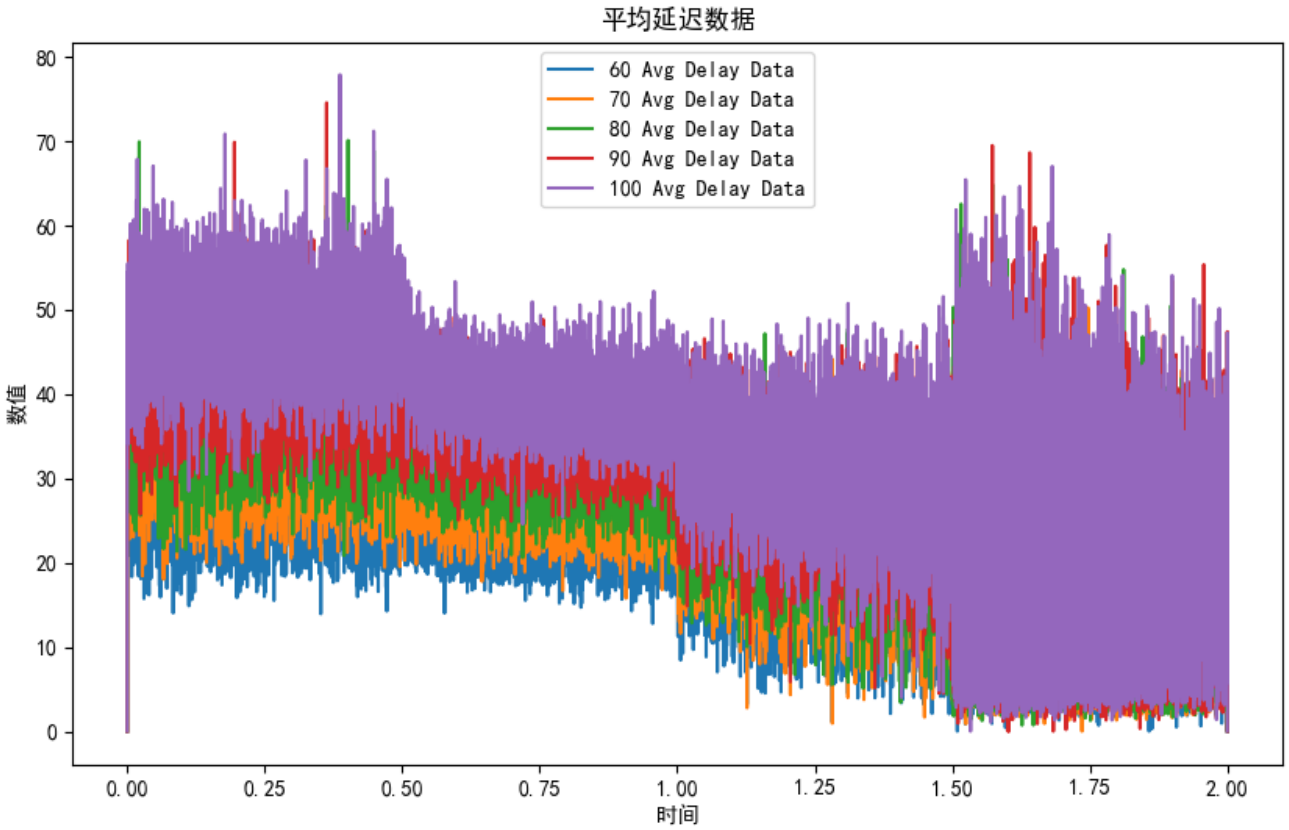
```
plt.figure(figsize=(10, 6))  # 创建一个大小为10x6的图像
plt.plot(trace_60.time, trace_60.total_delay, label='60 Total Delay Data')  # 绘制trace_60.time和trace_60.total_delay之间的关系图，并添加标签'60 Total Delay Data'
plt.plot(trace_70.time, trace_70.total_delay, label='70 Total Delay Data')  # 绘制trace_70.time和trace_70.total_delay之间的关系图，并添加标签'70 Total Delay Data'
plt.plot(trace_80.time, trace_80.total_delay, label='80 Total Delay Data')  # 绘制trace_80.time和trace_80.total_delay之间的关系图，并添加标签'80 Total Delay Data'
plt.plot(trace_90.time, trace_90.total_delay, label='90 Total Delay Data')  # 绘制trace_90.time和trace_90.total_delay之间的关系图，并添加标签'90 Total Delay Data'
plt.plot(trace_100.time, trace_100.total_delay, label='100 Total Delay Data')  # 绘制trace_100.time和trace_100.total_delay之间的关系图，并添加标签'100 Total Delay Data'
plt.xlabel('时间')  # 设置x轴标签为'时间'
plt.ylabel('数值')  # 设置y轴标签为'数值'
plt.title('延迟数据')  # 设置标题为'延迟数据'
plt.legend()  # 添加图例
plt.show()  # 显示图像
```



> - 具体来说，各个周期的总延迟数据的分布其实没有相差太多，但从细微来看，周期长度越大，其总延迟就越高。
> - 因为其中的数据包更多，产生的延迟就更多。
> - 因此该图表所展示的结果是合理的。

> 对每个窗口的平均延迟进行分析和可视化展示

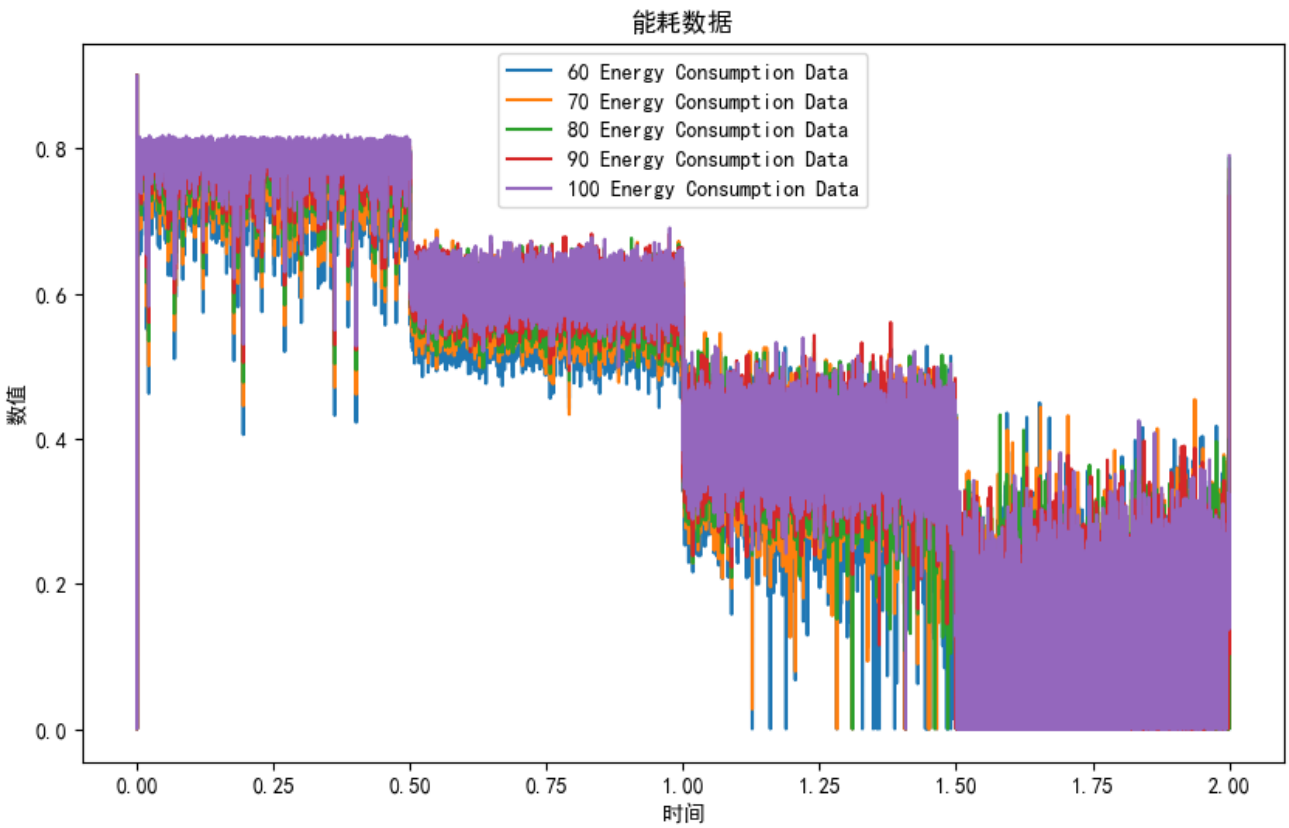
```
plt.figure(figsize=(10, 6))  # 创建一个大小为10x6的图像
plt.plot(trace_60.time, trace_60.avg_delay, label='60 Avg Delay Data')  # 绘制trace_60.time和
trace_60.avg_delay之间的关系图，并添加标签'60 Avg Delay Data'
plt.plot(trace_70.time, trace_70.avg_delay, label='70 Avg Delay Data')  # 绘制trace_70.time和
trace_70.avg_delay之间的关系图，并添加标签'70 Avg Delay Data'
plt.plot(trace_80.time, trace_80.avg_delay, label='80 Avg Delay Data')  # 绘制trace_80.time和
trace_80.avg_delay之间的关系图，并添加标签'80 Avg Delay Data'
plt.plot(trace_90.time, trace_90.avg_delay, label='90 Avg Delay Data')  # 绘制trace_90.time和
trace_90.avg_delay之间的关系图，并添加标签'90 Avg Delay Data'
plt.plot(trace_100.time, trace_100.avg_delay, label='100 Avg Delay Data')  # 绘制trace_100.time
和trace_100.avg_delay之间的关系图，并添加标签'100 Avg Delay Data'
plt.xlabel('时间')  # 设置x轴标签为'时间'
plt.ylabel('数值')  # 设置y轴标签为'数值'
plt.title('平均延迟数据')  # 设置标题为'平均延迟数据'
plt.legend()  # 添加图例
plt.show()  # 显示图像
```



> 整体上来看，随着周期长度的增加，平均延迟也会增加，具体原因可能是：
>
> - 当周期长度增加时，预测的时间范围也相应增加，这可能会导致更长的延迟。
> - 也可能是由于网络流量的变化、网络拥塞或其他因素导致的。

> 对能耗数据进行分析和可视化展示

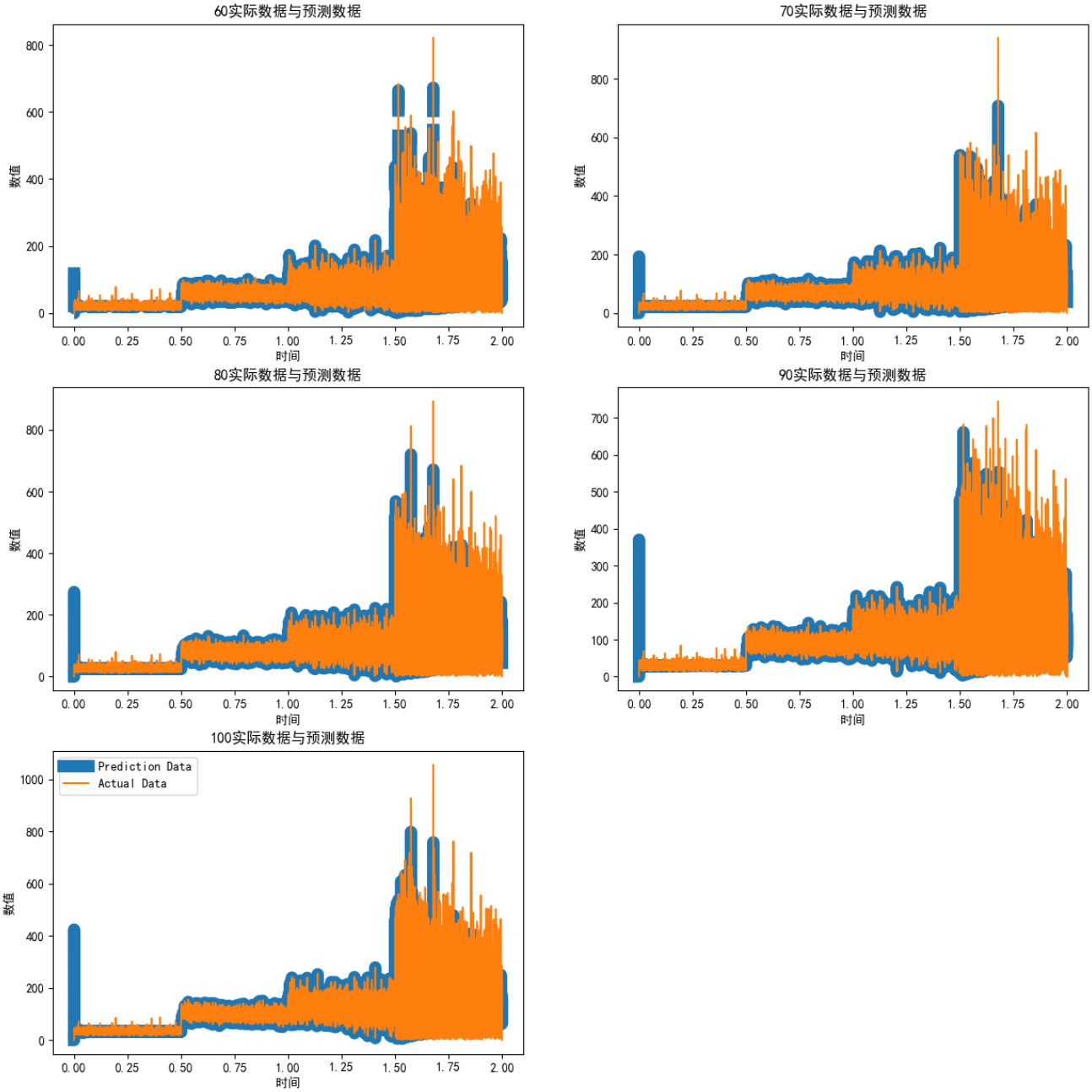
```
plt.figure(figsize=(10, 6))  # 创建一个大小为10x6的图像
plt.plot(trace_60.time, trace_60.energy_consumption, label='60 Energy Consumption Data')  # 绘制trace_60.time和trace_60.energy_consumption之间的关系图，并添加标签'60 Energy Consumption Data'
plt.plot(trace_70.time, trace_70.energy_consumption, label='70 Energy Consumption Data')  # 绘制trace_70.time和trace_70.energy_consumption之间的关系图，并添加标签'70 Energy Consumption Data'
plt.plot(trace_80.time, trace_80.energy_consumption, label='80 Energy Consumption Data')  # 绘制trace_80.time和trace_80.energy_consumption之间的关系图，并添加标签'80 Energy Consumption Data'
plt.plot(trace_90.time, trace_90.energy_consumption, label='90 Energy Consumption Data')  # 绘制trace_90.time和trace_90.energy_consumption之间的关系图，并添加标签'90 Energy Consumption Data'
plt.plot(trace_100.time, trace_100.energy_consumption, label='100 Energy Consumption Data')  # 绘制trace_100.time和trace_100.energy_consumption之间的关系图，并添加标签'100 Energy Consumption Data'
plt.xlabel('时间')  # 设置x轴标签为'时间'
plt.ylabel('数值')  # 设置y轴标签为'数值'
plt.title('能耗数据')  # 设置标题为'能耗数据'
plt.legend()  # 添加图例
plt.show()  # 显示图像
```



> 可以看出的是，随着周期长度的增加，其能耗的平均值也会上移（增加）。能耗的增加可能是由于以下几个原因：
>
> - 延迟增加导致设备需要更长时间运行以处理和传输数据包，从而消耗更多的能量。
> - 延迟增加可能导致设备需要进行更多的数据重传或处理错误，这会增加处理器和通信模块的负载，进而增加能耗。
> - 当网络拥塞或数据包传输延迟增加时，可能需要额外的网络设备或中继设备来处理和传输数据包，这些设备的工作量增加会导致能耗的增加。

> 对预测数据和真实数据进行可视化展示，绘制时间序列图

```
# 绘制时间序列图
plt.figure(figsize=(15, 15))

plt.subplot(3, 2, 1)  # 创建一个3x2的子图中的第一个子图
plt.plot(trace_60.time, trace_60.predict, label='Prediction Data', linewidth=10, linestyle="--")
 # 绘制trace_60.time和trace_60.predict之间的关系图，并添加标签'Prediction Data'，线宽为5
plt.plot(trace_60.time, trace_60.actual, label='Actual Data')  # 绘制trace_60.time和
trace_60.actual之间的关系图，并添加标签'Actual Data'
plt.xlabel('时间')  # 设置x轴标签为'时间'
plt.ylabel('数值')  # 设置y轴标签为'数值'
plt.title('60实际数据与预测数据')  # 设置标题为'60实际数据与预测数据'

plt.subplot(3, 2, 2)  # 创建一个3x2的子图中的第二个子图
plt.plot(trace_70.time, trace_70.predict, label='Prediction Data', linewidth=10)  # 绘制
trace_70.time和trace_70.predict之间的关系图，并添加标签'Prediction Data'，线宽为5
plt.plot(trace_70.time, trace_70.actual, label='Actual Data')  # 绘制trace_70.time和
trace_70.actual之间的关系图，并添加标签'Actual Data'
plt.xlabel('时间')  # 设置x轴标签为'时间'
plt.ylabel('数值')  # 设置y轴标签为'数值'
plt.title('70实际数据与预测数据')  # 设置标题为'70实际数据与预测数据'

plt.subplot(3, 2, 3)  # 创建一个3x2的子图中的第三个子图
plt.plot(trace_80.time, trace_80.predict, label='Prediction Data', linewidth=10)  # 绘制
trace_80.time和trace_80.predict之间的关系图，并添加标签'Prediction Data'，线宽为5
plt.plot(trace_80.time, trace_80.actual, label='Actual Data')  # 绘制trace_80.time和
trace_80.actual之间的关系图，并添加标签'Actual Data'
plt.xlabel('时间')  # 设置x轴标签为'时间'
plt.ylabel('数值')  # 设置y轴标签为'数值'
plt.title('80实际数据与预测数据')  # 设置标题为'80实际数据与预测数据'

plt.subplot(3, 2, 4)  # 创建一个3x2的子图中的第四个子图
plt.plot(trace_90.time, trace_90.predict, label='Prediction Data', linewidth=10)  # 绘制
trace_90.time和trace_90.predict之间的关系图，并添加标签'Prediction Data'，线宽为5
plt.plot(trace_90.time, trace_90.actual, label='Actual Data')  # 绘制trace_90.time和
trace_90.actual之间的关系图，并添加标签'Actual Data'
plt.xlabel('时间')  # 设置x轴标签为'时间'
plt.ylabel('数值')  # 设置y轴标签为'数值'
plt.title('90实际数据与预测数据')  # 设置标题为'90实际数据与预测数据'

plt.subplot(3, 2, 5)  # 创建一个3x2的子图中的第五个子图
plt.plot(trace_100.time, trace_100.predict, label='Prediction Data', linewidth=10)  # 绘制
trace_100.time和trace_100.predict之间的关系图，并添加标签'Prediction Data'，线宽为5
plt.plot(trace_100.time, trace_100.actual, label='Actual Data')  # 绘制trace_100.time和
trace_100.actual之间的关系图，并添加标签'Actual Data'
plt.xlabel('时间')  # 设置x轴标签为'时间'
plt.ylabel('数值')  # 设置y轴标签为'数值'
plt.title('100实际数据与预测数据')  # 设置标题为'100实际数据与预测数据'

plt.legend()  # 添加图例
plt.show()  # 显示图像
```



> - 可以看出的是，蓝色为预测数据，橙色为真实数据（当然由于设置算法的原因，初始点的预测值会很高，这个不应该作为分析的一部分）。预测值会比较贴合均衡的真实值，而对部分的离散点（如高峰值）的预测效果就没有那么好
> - 而随着周期长度的增加，预测值和真实值的拟合程度就更好，这是因为两者在一个窗口中的检测值的弹性也会更高，而恰好ARMA更趋向于均衡
> - 第二点对于之前步骤三利用 output1.csv 文件的多个周期的分析结果看起来是不一致的，当然这也仅仅是看起来不一致，实际上它所展示的图表中是有一定的缺陷的。而之所以讲一致，是因为在这些测试中都可以看出来无论是真实值还是预测值都是在趋于均衡的。

实验步骤五：

- ①设计新的预测算法替代EEEP策略中的ARMA预测算法；

- ②重复步骤四统计并分析节能效果和延时开销的变化。操作参考《基于NS3平台完成实验目标4-5》中第四部分。

> (1) 设计新的预测算法，对原文件的修改如下图所示：
>
> > 在 ARMA11.cc 文件中
> >
> > 在 ARMA11.h 文件中
> >
> > 在 scenario-ethernet-net-device.cc 文件中

> 算法修改部分解释：
>
> - 在PredictActiveTime()函数中，我们首先创建了一个ARIMA11对象arimaModel，并使用原始ARMA模型的参数a1和b1来设置ARIMA模型的参数a2和b2。
> - 然后，我们计算出中心观察值的差值centralArrivPackets。
> - 接下来，我们使用ARIMA模型进行预测，通过调用arimaModel.predictARIMA(centralArrivPackets)来获取下一个窗口的包数量的预测
> - 值。最后，我们进行必要的处理，如将预测值限制在非负范围
> - preNumber = -a1 * newObservation + b1 * interfere + a2 * initOberserve[0] + b2 * initOberserve[1] 这个公式通过组合先前观测值、干扰项和初始观测值的乘积来计算预测的观测值。在ARIMA算法中，通过使用这个公式，我们可以根据先前的观测值和其他参数来预测下一个观测值。内。

> (2) 使用修改后的模拟预测算法 (ARIMA) 后的测试和结果如下图所示
>
> > 对比之前在步骤四中的模拟测试结果，可以发现延迟会更低，非空时窗到达包数也会更少。

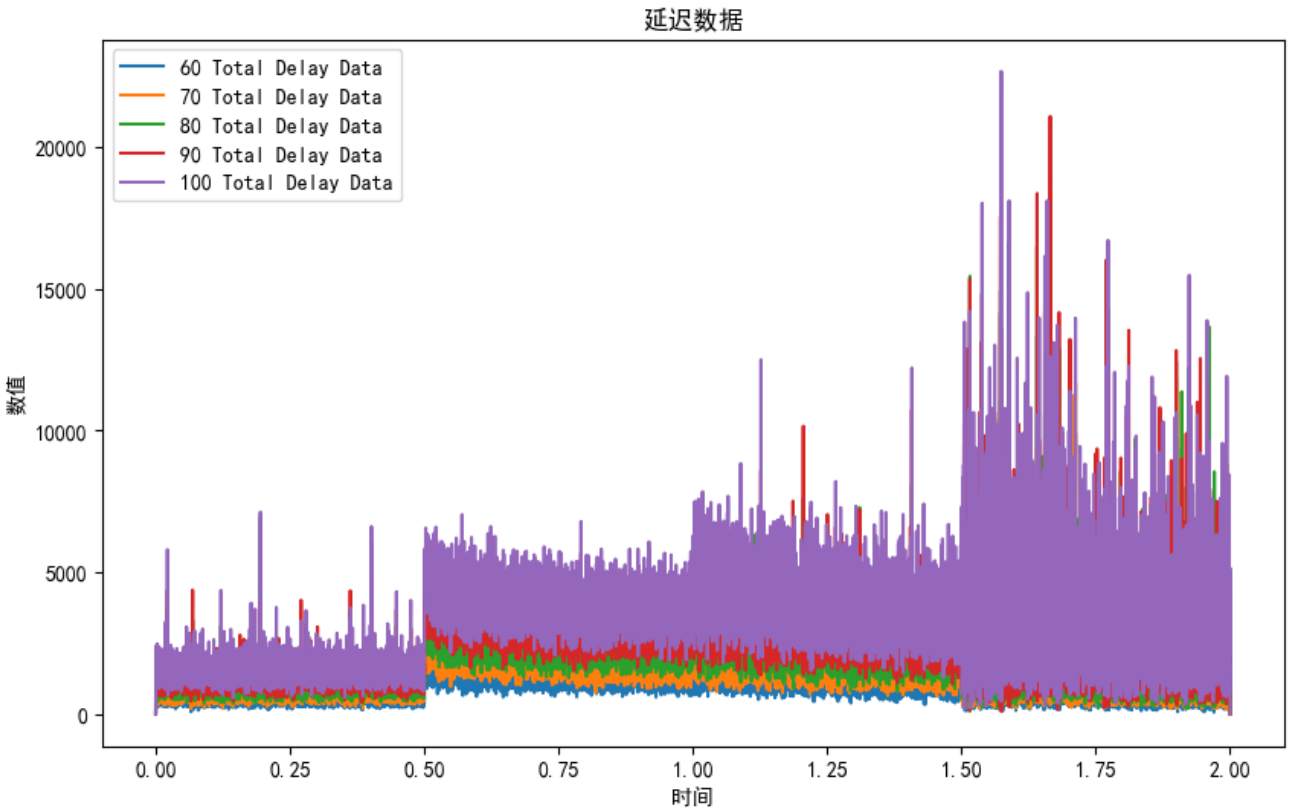
> (3) 重复步骤四统计并分析节能效果和延时开销的变化

```
re_60 = pd.read_csv('RETEST_60.txt')  # 读取文件TEST_60.txt的数据并赋值给trace_60
re_70 = pd.read_csv('RETEST_70.txt')  # 读取文件TEST_70.txt的数据并赋值给trace_70
re_80 = pd.read_csv('RETEST_80.txt')  # 读取文件TEST_80.txt的数据并赋值给trace_80
re_90 = pd.read_csv('RETEST_90.txt')  # 读取文件TEST_90.txt的数据并赋值给trace_90
re_100 = pd.read_csv('RETEST_100.txt')  # 读取文件TEST_100.txt的数据并赋值给trace_100
print(re_60.index[-1], re_70.index[-1], re_80.index[-1], re_90.index[-1], re_100.index[-1])  # 打印最后一个索引值，即bao'shu
```

```
28328 24268 21202 18814 16926
```

> > 对每个窗口的总延迟数据进行分析

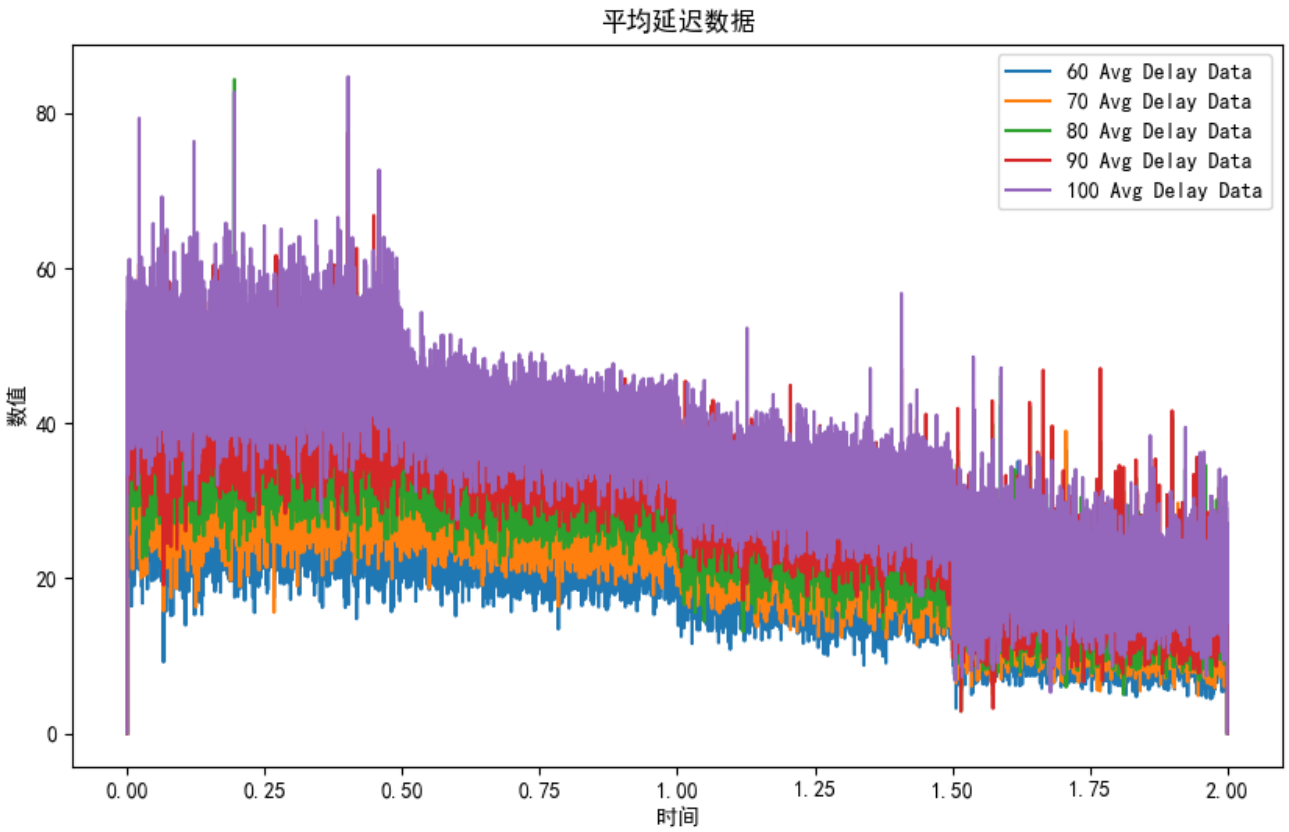
```
plt.figure(figsize=(10, 6))  # 创建一个大小为10x6的图像
plt.plot(re_60.time, re_60.total_delay, label='60 Total Delay Data')  # 绘制trace_60.time和
trace_60.total_delay之间的关系图，并添加标签'60 Total Delay Data'
plt.plot(re_70.time, re_70.total_delay, label='70 Total Delay Data')  # 绘制trace_70.time和
trace_70.total_delay之间的关系图，并添加标签'70 Total Delay Data'
plt.plot(re_80.time, re_80.total_delay, label='80 Total Delay Data')  # 绘制trace_80.time和
trace_80.total_delay之间的关系图，并添加标签'80 Total Delay Data'
plt.plot(re_90.time, re_90.total_delay, label='90 Total Delay Data')  # 绘制trace_90.time和
trace_90.total_delay之间的关系图，并添加标签'90 Total Delay Data'
plt.plot(re_100.time, re_100.total_delay, label='100 Total Delay Data')  # 绘制trace_100.time和
trace_100.total_delay之间的关系图，并添加标签'100 Total Delay Data'
plt.xlabel('时间')  # 设置x轴标签为'时间'
plt.ylabel('数值')  # 设置y轴标签为'数值'
plt.title('延迟数据')  # 设置标题为'延迟数据'
plt.legend()  # 添加图例
plt.show()  # 显示图像
```



> 对比为修改前的测试数据，可以发现在总延迟中，延迟更加均衡一些，较高延迟的（比如总延迟数据达到30000以上的）更少了，说明ARIMA算法确实较于ARMA算法对于模型有优化

> 对每个窗口的平均延迟进行分析和可视化展示

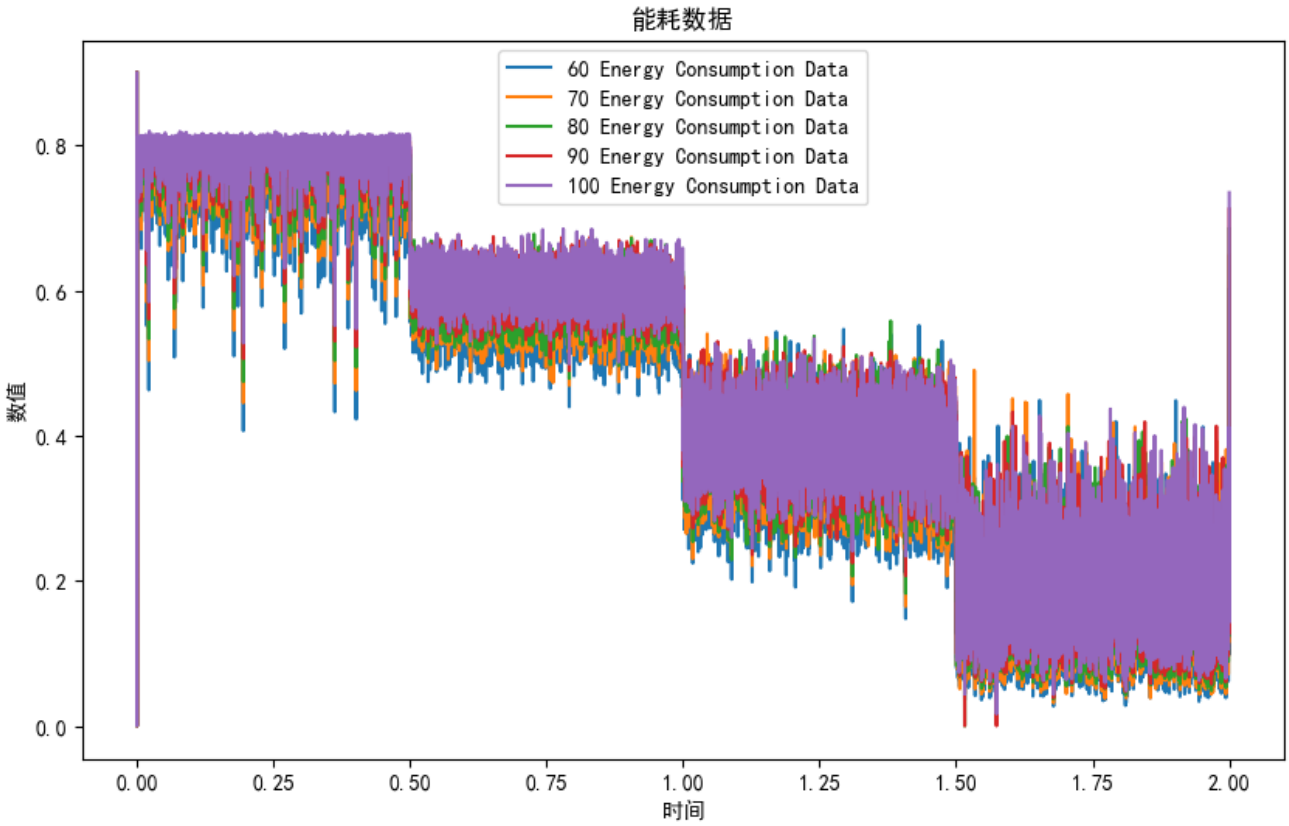
```
plt.figure(figsize=(10, 6))  # 创建一个大小为10x6的图像
plt.plot(re_60.time, re_60.avg_delay, label='60 Avg Delay Data')  # 绘制trace_60.time和
trace_60.avg_delay之间的关系图，并添加标签'60 Avg Delay Data'
plt.plot(re_70.time, re_70.avg_delay, label='70 Avg Delay Data')  # 绘制trace_70.time和
trace_70.avg_delay之间的关系图，并添加标签'70 Avg Delay Data'
plt.plot(re_80.time, re_80.avg_delay, label='80 Avg Delay Data')  # 绘制trace_80.time和
trace_80.avg_delay之间的关系图，并添加标签'80 Avg Delay Data'
plt.plot(re_90.time, re_90.avg_delay, label='90 Avg Delay Data')  # 绘制trace_90.time和
trace_90.avg_delay之间的关系图，并添加标签'90 Avg Delay Data'
plt.plot(re_100.time, re_100.avg_delay, label='100 Avg Delay Data')  # 绘制trace_100.time和
trace_100.avg_delay之间的关系图，并添加标签'100 Avg Delay Data'
plt.xlabel('时间')  # 设置x轴标签为'时间'
plt.ylabel('数值')  # 设置y轴标签为'数值'
plt.title('平均延迟数据')  # 设置标题为'平均延迟数据'
plt.legend()  # 添加图例
plt.show()  # 显示图像
```



> 分析：与之前的对比并未出现不同

> 对能耗数据进行分析和可视化展示

```
plt.figure(figsize=(10, 6))  # 创建一个大小为10x6的图像
plt.plot(re_60.time, re_60.energy_consumption, label='60 Energy Consumption Data')  # 绘制
trace_60.time和trace_60.energy_consumption之间的关系图，并添加标签'60 Energy Consumption Data'
plt.plot(re_70.time, re_70.energy_consumption, label='70 Energy Consumption Data')  # 绘制
trace_70.time和trace_70.energy_consumption之间的关系图，并添加标签'70 Energy Consumption Data'
plt.plot(re_80.time, re_80.energy_consumption, label='80 Energy Consumption Data')  # 绘制
trace_80.time和trace_80.energy_consumption之间的关系图，并添加标签'80 Energy Consumption Data'
plt.plot(re_90.time, re_90.energy_consumption, label='90 Energy Consumption Data')  # 绘制
trace_90.time和trace_90.energy_consumption之间的关系图，并添加标签'90 Energy Consumption Data'
plt.plot(re_100.time, re_100.energy_consumption, label='100 Energy Consumption Data')  # 绘制
trace_100.time和trace_100.energy_consumption之间的关系图，并添加标签'100 Energy Consumption Data'
plt.xlabel('时间')  # 设置x轴标签为'时间'
plt.ylabel('数值')  # 设置y轴标签为'数值'
plt.title('能耗数据')  # 设置标题为'能耗数据'
plt.legend()  # 添加图例
plt.show()  # 显示图像
```

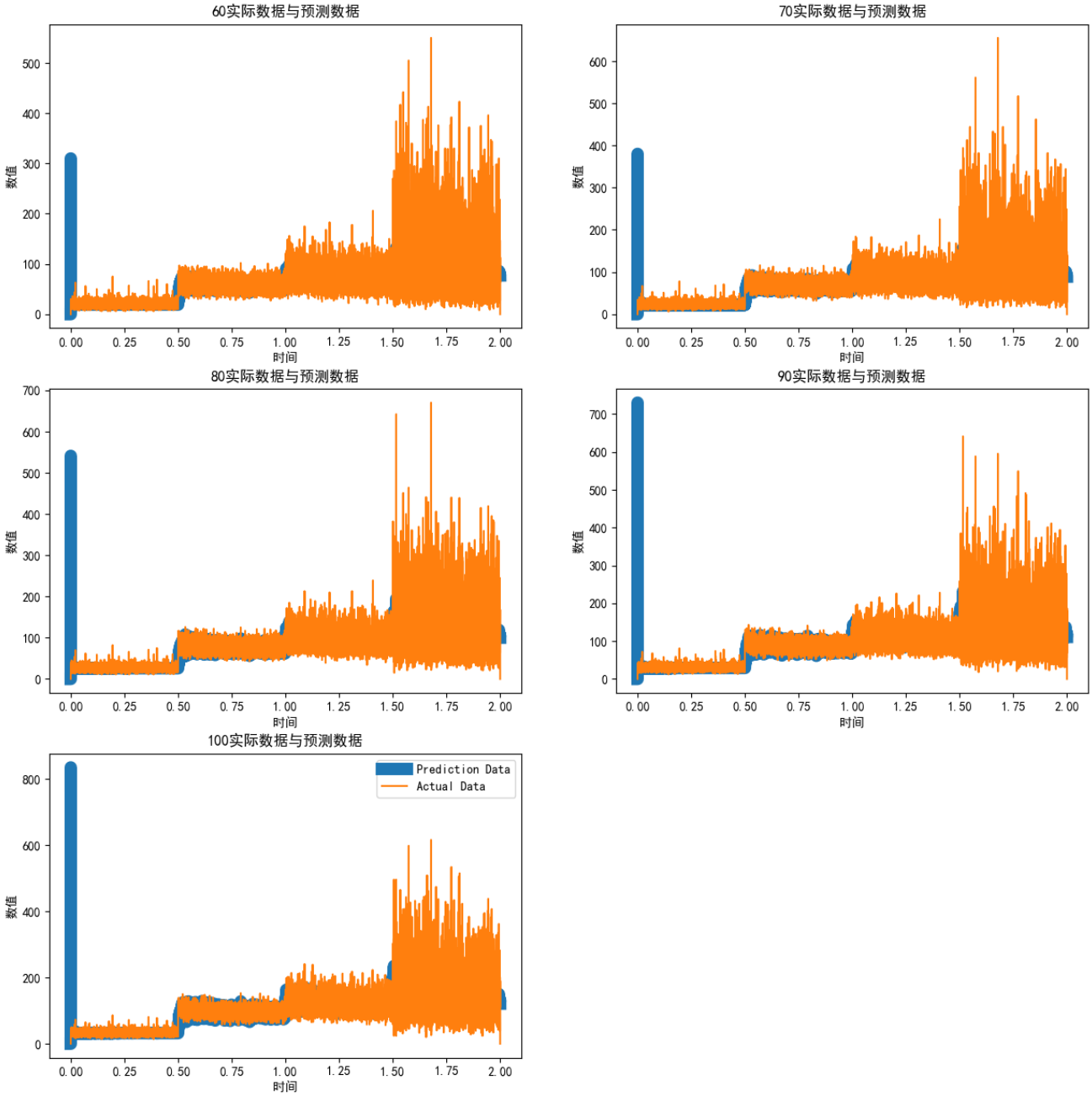


> 分析：与之前的相比，也没有较为明显的差别

> 对预测数据和真实数据进行可视化展示，绘制时间序列图

```
# 绘制时间序列图
plt.figure(figsize=(15, 15))

plt.subplot(3, 2, 1)  # 创建一个3x2的子图中的第一个子图
plt.plot(re_60.time, re_60.predict, label='Prediction Data', linewidth=10)  # 绘制trace_60.time
和trace_60.predict之间的关系图，并添加标签'Prediction Data'，线宽为5
```

```
plt.plot(re_60.time, re_60.actual, label='Actual Data')  # 绘制trace_60.time和trace_60.actual之间的关系图，并添加标签'Actual Data'
plt.xlabel('时间')  # 设置x轴标签为'时间'
plt.ylabel('数值')  # 设置y轴标签为'数值'
plt.title('60实际数据与预测数据')  # 设置标题为'60实际数据与预测数据'

plt.subplot(3, 2, 2)  # 创建一个3x2的子图中的第二个子图
plt.plot(re_70.time, re_70.predict, label='Prediction Data', linewidth=10)  # 绘制trace_70.time和trace_70.predict之间的关系图，并添加标签'Prediction Data'，线宽为5
plt.plot(re_70.time, re_70.actual, label='Actual Data')  # 绘制trace_70.time和trace_70.actual之间的关系图，并添加标签'Actual Data'
plt.xlabel('时间')  # 设置x轴标签为'时间'
plt.ylabel('数值')  # 设置y轴标签为'数值'
plt.title('70实际数据与预测数据')  # 设置标题为'70实际数据与预测数据'

plt.subplot(3, 2, 3)  # 创建一个3x2的子图中的第三个子图
plt.plot(re_80.time, re_80.predict, label='Prediction Data', linewidth=10)  # 绘制trace_80.time和trace_80.predict之间的关系图，并添加标签'Prediction Data'，线宽为5
plt.plot(re_80.time, re_80.actual, label='Actual Data')  # 绘制trace_80.time和trace_80.actual之间的关系图，并添加标签'Actual Data'
plt.xlabel('时间')  # 设置x轴标签为'时间'
plt.ylabel('数值')  # 设置y轴标签为'数值'
plt.title('80实际数据与预测数据')  # 设置标题为'80实际数据与预测数据'

plt.subplot(3, 2, 4)  # 创建一个3x2的子图中的第四个子图
plt.plot(re_90.time, re_90.predict, label='Prediction Data', linewidth=10)  # 绘制trace_90.time和trace_90.predict之间的关系图，并添加标签'Prediction Data'，线宽为5
plt.plot(re_90.time, re_90.actual, label='Actual Data')  # 绘制trace_90.time和trace_90.actual之间的关系图，并添加标签'Actual Data'
plt.xlabel('时间')  # 设置x轴标签为'时间'
plt.ylabel('数值')  # 设置y轴标签为'数值'
plt.title('90实际数据与预测数据')  # 设置标题为'90实际数据与预测数据'

plt.subplot(3, 2, 5)  # 创建一个3x2的子图中的第五个子图
plt.plot(re_100.time, re_100.predict, label='Prediction Data', linewidth=10)  # 绘制trace_100.time和trace_100.predict之间的关系图，并添加标签'Prediction Data'，线宽为5
plt.plot(re_100.time, re_100.actual, label='Actual Data')  # 绘制trace_100.time和trace_100.actual之间的关系图，并添加标签'Actual Data'
plt.xlabel('时间')  # 设置x轴标签为'时间'
plt.ylabel('数值')  # 设置y轴标签为'数值'
plt.title('100实际数据与预测数据')  # 设置标题为'100实际数据与预测数据'

plt.legend()  # 添加图例
plt.show()  # 显示图像
```



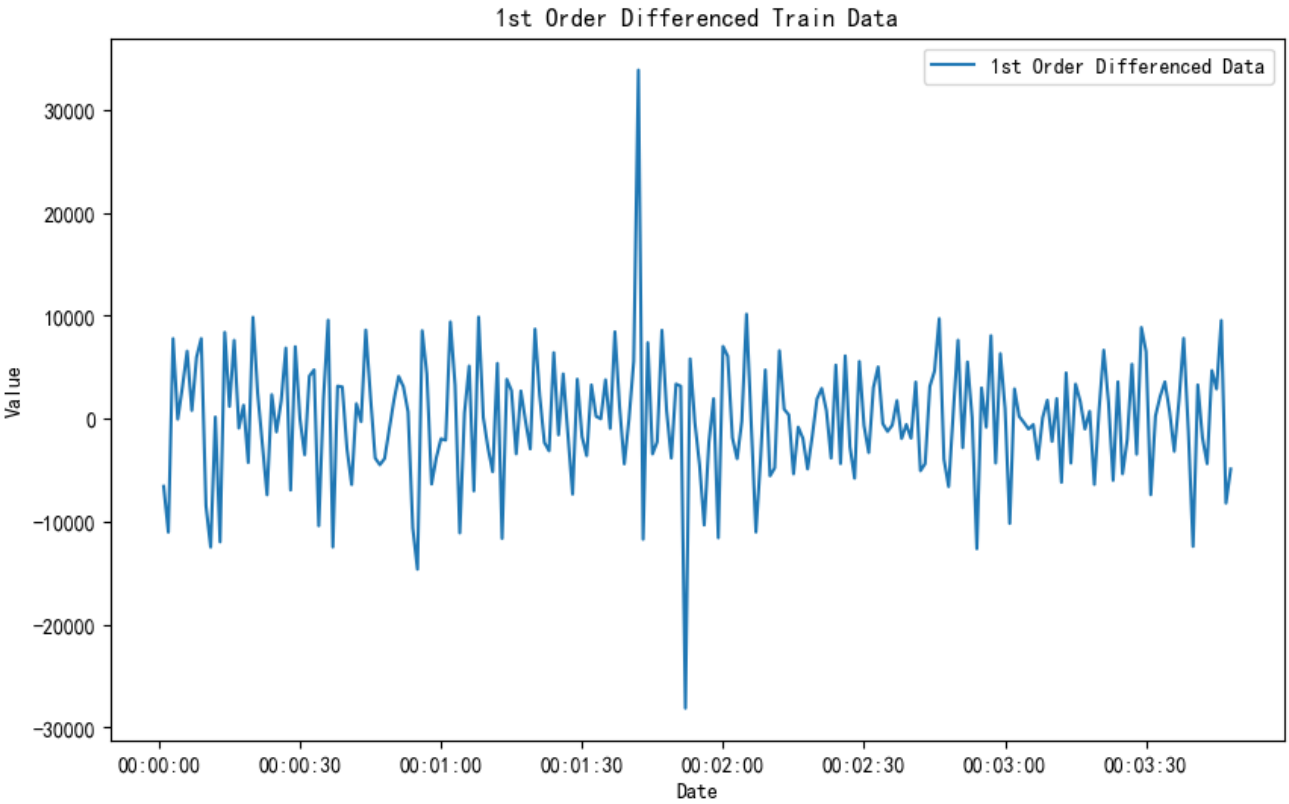
> 与之前的相比，可以看出的修改后的算法更趋于均衡，对于散点的抗干扰能力更强

> 综上所示，可以看出的是 ARIMA 算法相较于 ARMA 算法的均衡趋向的更强，意味着更适合于对预测的趋势判断而不是对于精确值的预测。

(4) 额外对 ARIMA 的测试，使用之前的 train_1 数据（之前在步骤二中是发现其 ADF 检验是大于标准的，应该进行差分操作）

> 通过对时间序列进行差分操作，可以消除趋势或季节性影响，有助于使非平稳的时间序列转换为平稳时间序列，这对于许多时间序列分析和预测模型来说是必要的预处理步骤。

```
diff1 = train_1.diff(1)  # 对时间序列数据进行一阶差分，即计算相邻时刻的数据差值
diff1.dropna(inplace=True)  # 删除包含缺失值的行

# 绘制差分后的时间序列
plt.figure(figsize=(10, 6))  # 设置图表大小为宽10英寸，高6英寸
plt.plot(diff1.index, diff1, label='1st Order Differenced Data')
plt.xlabel('Date')  # X轴标签设置为'Date'
plt.ylabel('Value')  # Y轴标签设置为'Value'
plt.title('1st Order Differenced Train Data')  # 图表标题设置为'1st Order Differenced Train Data'
plt.legend()  # 显示图例
plt.show()  # 显示图表
```



> 这段代码将对差分序列进行单位根检验，以确定是否通过差分操作使序列变得更加平稳。如果得到的p值小于0.05，则可以认为差分后的序列是平稳的，适合进一步进行时间序列分析和建模。

```
# 假设 diff1 是经过一阶差分处理后的时间序列数据
result_diff1 = sm.tsa.stattools.adfuller(diff1)

# 输出 ADF 检验的结果
print('ADF Statistic: %f' % result_diff1[0])
print('p-value: %f' % result_diff1[1])
for key, value in result_diff1[4].items():
    print('\t%s: %.3f' % (key, value))
```

```
ADF Statistic: -5.623617
p-value: 0.000001
        1%: -3.461
        5%: -2.875
        10%: -2.574
```

> 模型预测，由之前可知 p=11, q=2

```
# 在构建ARIMA模型时，指定的参数为(2, 1, 2)，表示ARIMA(p=2, d=1, q=2)模型，
# 其中p是自回归项数，d是一阶差分次数（在这里应该已经通过diff1完成了一阶差分），q是移动平均项数。
model = sm.tsa.ARIMA(train_1, order=(11, 1, 2))
model_fit = model.fit()
# 对未来数据进行预测
prediction = model_fit.forecast(steps=len(test_1))
# 提取训练集上的预测结果（从第二个值开始）
fit = model_fit.predict(start=1, end=len(train_1))
```

> 计算RMSE（均方根误差）和 MAE（平均绝对误差）

```
# 计算RMSE（均方根误差）
rmse = np.sqrt(mean_squared_error(test_1, prediction))
# 计算MAE（平均绝对误差）
mae = mean_absolute_error(test_1, prediction)
# 输出RMSE, MAE
print("数据包的RMSE：", rmse)
print("数据包的MAE：", mae)
```

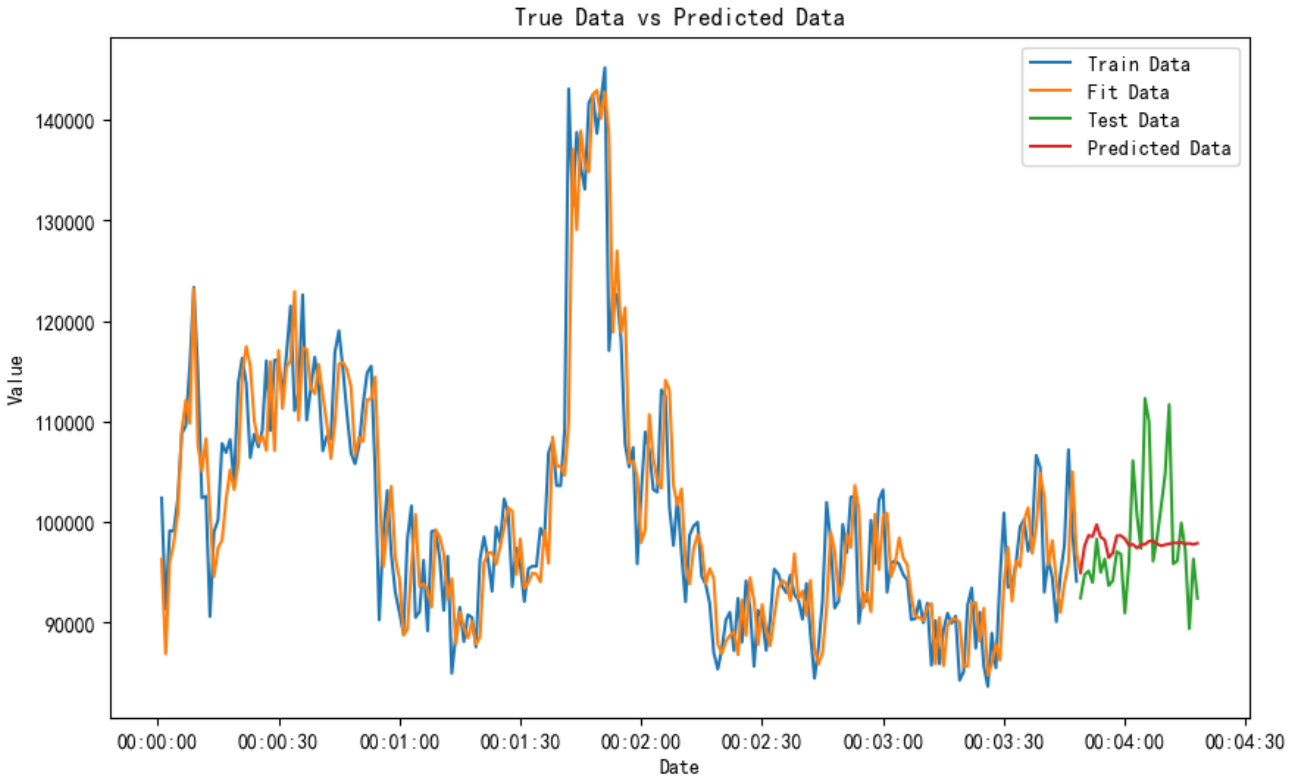
```
数据包的RMSE：  5638.707901851229
数据包的MAE：  4208.502801489712
```

> > 对比发现，RMSE差别不大，MAE比之前的4731低许多

> 绘制预测时间序列图

```
# train 是未经差分处理的原始训练数据（注意：这里的 fit 应该是从原始尺度上计算出的预测结果）
# test 是未经差分处理的原始测试数据
# prediction 是基于模型对整个未来时间段（包括测试集部分）的预测值，并且其索引与test数据的索引对应

# 绘制时间序列图
plt.figure(figsize=(10, 6))
plt.plot(train_1.index[1:], train_1.values[1:], label='Train Data')  # 训练数据从第二个值开始
plt.plot(fit.index, fit.values, label='Fit Data')  # 拟合数据应与训练数据的时间索引对齐
plt.plot(test_1.index, test_1.values, label='Test Data')  # 测试数据
plt.plot(prediction.index, prediction.values, label='Predicted Data')  # 预测数据

plt.xlabel('Date')
plt.ylabel('Value')
plt.title('True Data vs Predicted Data')
plt.legend()
```

```
plt.show()
```



> 对比发现，是用 ARIMA 进行差分后，其趋势会更快进入均衡状态，拟合程度也更贴近真实值，但是走向会因趋向均衡而丢失一些信息特征而较于 ARMA 不足

实验总结

- 本次实验展示了利用时间序列预测以太网数据包到达情况以实现节能的潜力。准确预测数据包到达情况可以实现动态资源分配和优化，从而降低能耗。实验结果强调将预测模型纳入网络管理策略以提高能效的重要性。进一步的研究和优化预测模型，并将其整合到网络系统中，可以在长期内实现显著的能源节约。

- 在本次实验中遇到了许多难点，在这里特别感谢助教的帮助和解答。

- 难点1：在初次编译过程中，按照实验指导书上步骤和方法并不能解决报错问题，后来在助教的指导下使用另外的命令完成编译：CXXFLAGS="-std=c++11" ./waf configure，具体编译成功图片如下：

- 难点2：使用切割的流量文件后缀为pcap，这个文件是不受程序支持的，而实验指导书没有指出文件具体要求如何，使得这一步非常艰难；同时，流量文件中的具体的数据代表的含义也没有指出；另外，实验所给的程序没有指导性和说明性的注释和注解，这令人头痛；最后，实验指导书的实验代码中的变量名是与学生实验所给的代码中的变量是不一致的，这意味着实验指导书存在一定的过时和不严谨性。

- 难点3：切割流量文件时，由于所给的流量文件太大，会把电脑内存占崩，所以尝试了多种方法，例如使用 wireshark 进行导出特定分组，这个方法只能导出到130秒左右的数据，并且很消耗人的精力和耐心，在尝试这个方法的时候整整用了两天的时间；最后选择使用 python 利用指针进行读取数据，跑了将近 40 分钟获得 output1.csv 文件将近270秒数据。