Mobile Phone Pricing Classifier

> **This notebook is in association with the Unified Mentor Machine Learning internship project submission.**

The task of the porject is to develop a system that can predict the price of a mobile phone using the data available on phones in the market. The mobile phones must be categorized as `0: low cost` / `1: medium cost` / `2: high cost` or `3: very high cost`.

Visit Deployed [Mobile Price Range Classifier Streamlit App](#)

1. Importing Dataset

In [1]:

```
!git clone https://github.com/PranayJagtap06/UFM_Mobile_Phone_Pricing.git
```

```
Cloning into 'UFM_Mobile_Phone_Pricing'...
remote: Enumerating objects: 32, done.
remote: Counting objects: 100% (32/32), done.
remote: Compressing objects: 100% (29/29), done.
remote: Total 32 (delta 11), reused 10 (delta 1), pack-reused 0 (from 0)
Receiving objects: 100% (32/32), 1.49 MiB | 3.98 MiB/s, done.
Resolving deltas: 100% (11/11), done.
```

In [2]:

```
import zipfile

zip_ref = zipfile.ZipFile("/content/UFM_Mobile_Phone_Pricing/mobile_phone_pricing.zip",
'r')
zip_ref.extractall("/content")
zip_ref.close()
```

2. Importing Libraries

In [3]:

```
!pip install icecream dagshub mlflow[jupyter]
```

```
Collecting icecream
  Downloading icecream-2.1.3-py2.py3-none-any.whl.metadata (1.4 kB)
Collecting dagshub
  Downloading dagshub-0.3.45-py3-none-any.whl.metadata (11 kB)
Collecting mlflow[jupyter]
  Downloading mlflow-2.18.0-py3-none-any.whl.metadata (29 kB)
Collecting colorama>=0.3.9 (from icecream)
  Downloading colorama-0.4.6-py2.py3-none-any.whl.metadata (17 kB)
Requirement already satisfied: pygments>=2.2.0 in /usr/local/lib/python3.10/dist-packages
(from icecream) (2.18.0)
Collecting executing>=0.3.1 (from icecream)
  Downloading executing-2.1.0-py2.py3-none-any.whl.metadata (8.9 kB)
Collecting asttokens>=2.0.1 (from icecream)
  Downloading asttokens-3.0.0-py3-none-any.whl.metadata (4.7 kB)
Requirement already satisfied: PyYAML>=5 in /usr/local/lib/python3.10/dist-packages (from
dagshub) (6.0.2)
Collecting appdirs>=1.4.4 (from dagshub)
  Downloading appdirs-1.4.4-py2.py3-none-any.whl.metadata (9.0 kB)
Requirement already satisfied: click>=8.0.4 in /usr/local/lib/python3.10/dist-packages (f
rom dagshub) (8.1.7)
Requirement already satisfied: httpx>=0.23.0 in /usr/local/lib/python3.10/dist-packages (
from dagshub) (0.27.2)
```

```
Requirement already satisfied: GitPython>=3.1.29 in /usr/local/lib/python3.10/dist-packag
es (from dagshub) (3.1.43)
Requirement already satisfied: rich>=13.1.0 in /usr/local/lib/python3.10/dist-packages (f
rom dagshub) (13.9.4)
Collecting dacite~=1.6.0 (from dagshub)
  Downloading dacite-1.6.0-py3-none-any.whl.metadata (14 kB)
Requirement already satisfied: tenacity>=8.2.2 in /usr/local/lib/python3.10/dist-packages
(from dagshub) (9.0.0)
Collecting gql[requests] (from dagshub)
  Downloading gql-3.5.0-py2.py3-none-any.whl.metadata (9.2 kB)
Collecting dataclasses-json (from dagshub)
  Downloading dataclasses_json-0.6.7-py3-none-any.whl.metadata (25 kB)
Requirement already satisfied: pandas in /usr/local/lib/python3.10/dist-packages (from da
gshub) (2.2.2)
Collecting treelib>=1.6.4 (from dagshub)
  Downloading treelib-1.7.0-py3-none-any.whl.metadata (1.3 kB)
Collecting pathvalidate>=3.0.0 (from dagshub)
  Downloading pathvalidate-3.2.1-py3-none-any.whl.metadata (12 kB)
Requirement already satisfied: python-dateutil in /usr/local/lib/python3.10/dist-packages
(from dagshub) (2.8.2)
Collecting boto3 (from dagshub)
  Downloading boto3-1.35.73-py3-none-any.whl.metadata (6.7 kB)
Collecting dagshub-annotation-converter>=0.1.0 (from dagshub)
  Downloading dagshub_annotation_converter-0.1.2-py3-none-any.whl.metadata (2.5 kB)
WARNING: mlflow 2.18.0 does not provide the extra 'jupyter'
Collecting mlflow-skinny==2.18.0 (from mlflow[jupyter])
  Downloading mlflow_skinny-2.18.0-py3-none-any.whl.metadata (30 kB)
Requirement already satisfied: Flask<4 in /usr/local/lib/python3.10/dist-packages (from m
lflow[jupyter]) (3.0.3)
Collecting alembic!=1.10.0,<2 (from mlflow[jupyter])
  Downloading alembic-1.14.0-py3-none-any.whl.metadata (7.4 kB)
Collecting docker<8,>=4.0.0 (from mlflow[jupyter])
  Downloading docker-7.1.0-py3-none-any.whl.metadata (3.8 kB)
Collecting graphene<4 (from mlflow[jupyter])
  Downloading graphene-3.4.3-py2.py3-none-any.whl.metadata (6.9 kB)
Requirement already satisfied: markdown<4,>=3.3 in /usr/local/lib/python3.10/dist-package
s (from mlflow[jupyter]) (3.7)
Requirement already satisfied: matplotlib<4 in /usr/local/lib/python3.10/dist-packages (f
rom mlflow[jupyter]) (3.8.0)
Requirement already satisfied: numpy<3 in /usr/local/lib/python3.10/dist-packages (from m
lflow[jupyter]) (1.26.4)
Requirement already satisfied: pyarrow<19,>=4.0.0 in /usr/local/lib/python3.10/dist-packa
ges (from mlflow[jupyter]) (17.0.0)
Requirement already satisfied: scikit-learn<2 in /usr/local/lib/python3.10/dist-packages
(from mlflow[jupyter]) (1.5.2)
Requirement already satisfied: scipy<2 in /usr/local/lib/python3.10/dist-packages (from m
lflow[jupyter]) (1.13.1)
Requirement already satisfied: sqlalchemy<3,>=1.4.0 in /usr/local/lib/python3.10/dist-pac
kages (from mlflow[jupyter]) (2.0.36)
Requirement already satisfied: Jinja2<4,>=2.11 in /usr/local/lib/python3.10/dist-packages
(from mlflow[jupyter]) (3.1.4)
Collecting gunicorn<24 (from mlflow[jupyter])
  Downloading gunicorn-23.0.0-py3-none-any.whl.metadata (4.4 kB)
Requirement already satisfied: cachetools<6,>=5.0.0 in /usr/local/lib/python3.10/dist-pac
kages (from mlflow-skinny==2.18.0->mlflow[jupyter]) (5.5.0)
Requirement already satisfied: cloudpickle<4 in /usr/local/lib/python3.10/dist-packages (
from mlflow-skinny==2.18.0->mlflow[jupyter]) (3.1.0)
Collecting databricks-sdk<1,>=0.20.0 (from mlflow-skinny==2.18.0->mlflow[jupyter])
  Downloading databricks_sdk-0.38.0-py3-none-any.whl.metadata (38 kB)
Requirement already satisfied: importlib-metadata!=4.7.0,<9,>=3.7.0 in /usr/local/lib/pyt
hon3.10/dist-packages (from mlflow-skinny==2.18.0->mlflow[jupyter]) (8.5.0)
Requirement already satisfied: opentelemetry-api<3,>=1.9.0 in /usr/local/lib/python3.10/d
ist-packages (from mlflow-skinny==2.18.0->mlflow[jupyter]) (1.28.2)
Requirement already satisfied: opentelemetry-sdk<3,>=1.9.0 in /usr/local/lib/python3.10/d
ist-packages (from mlflow-skinny==2.18.0->mlflow[jupyter]) (1.28.2)
Requirement already satisfied: packaging<25 in /usr/local/lib/python3.10/dist-packages (f
rom mlflow-skinny==2.18.0->mlflow[jupyter]) (24.2)
Requirement already satisfied: protobuf<6,>=3.12.0 in /usr/local/lib/python3.10/dist-pack
ages (from mlflow-skinny==2.18.0->mlflow[jupyter]) (4.25.5)
Requirement already satisfied: requests<3,>=2.17.3 in /usr/local/lib/python3.10/dist-pack
ages (from mlflow-skinny==2.18.0->mlflow[jupyter]) (2.32.3)
Requirement already satisfied: sqlparse<1,>=0.4.0 in /usr/local/lib/python3.10/dist-packa
```

```
ges (from mlflow-skinny==2.18.0->mlflow[jupyter]) (0.5.2)
Collecting Mako (from alembic!=1.10.0,<2->mlflow[jupyter])
  Downloading Mako-1.3.6-py3-none-any.whl.metadata (2.9 kB)
Requirement already satisfied: typing-extensions>=4 in /usr/local/lib/python3.10/dist-pac
kages (from alembic!=1.10.0,<2->mlflow[jupyter]) (4.12.2)
Requirement already satisfied: lxml in /usr/local/lib/python3.10/dist-packages (from dags
hub-annotation-converter>=0.1.0->dagshub) (5.3.0)
Requirement already satisfied: pillow in /usr/local/lib/python3.10/dist-packages (from da
gshub-annotation-converter>=0.1.0->dagshub) (11.0.0)
Requirement already satisfied: pydantic>=2.0.0 in /usr/local/lib/python3.10/dist-packages
(from dagshub-annotation-converter>=0.1.0->dagshub) (2.9.2)
Requirement already satisfied: urllib3>=1.26.0 in /usr/local/lib/python3.10/dist-packages
(from docker<8,>=4.0.0->mlflow[jupyter]) (2.2.3)
Requirement already satisfied: Werkzeug>=3.0.0 in /usr/local/lib/python3.10/dist-packages
(from Flask<4->mlflow[jupyter]) (3.1.3)
Requirement already satisfied: itsdangerous>=2.1.2 in /usr/local/lib/python3.10/dist-pack
ages (from Flask<4->mlflow[jupyter]) (2.2.0)
Requirement already satisfied: blinker>=1.6.2 in /usr/local/lib/python3.10/dist-packages
(from Flask<4->mlflow[jupyter]) (1.9.0)
Requirement already satisfied: gitdb<5,>=4.0.1 in /usr/local/lib/python3.10/dist-packages
(from GitPython>=3.1.29->dagshub) (4.0.11)
Collecting graphql-core<3.3,>=3.1 (from graphene<4->mlflow[jupyter])
  Downloading graphql_core-3.2.5-py3-none-any.whl.metadata (10 kB)
Collecting graphql-relay<3.3,>=3.1 (from graphene<4->mlflow[jupyter])
  Downloading graphql_relay-3.2.0-py3-none-any.whl.metadata (12 kB)
Requirement already satisfied: anyio in /usr/local/lib/python3.10/dist-packages (from htt
px>=0.23.0->dagshub) (3.7.1)
Requirement already satisfied: certifi in /usr/local/lib/python3.10/dist-packages (from h
ttpx>=0.23.0->dagshub) (2024.8.30)
Requirement already satisfied: httpcore==1.* in /usr/local/lib/python3.10/dist-packages (
from httpx>=0.23.0->dagshub) (1.0.7)
Requirement already satisfied: idna in /usr/local/lib/python3.10/dist-packages (from http
x>=0.23.0->dagshub) (3.10)
Requirement already satisfied: sniffio in /usr/local/lib/python3.10/dist-packages (from h
ttpx>=0.23.0->dagshub) (1.3.1)
Requirement already satisfied: h11<0.15,>=0.13 in /usr/local/lib/python3.10/dist-packages
(from httpcore==1.*->httpx>=0.23.0->dagshub) (0.14.0)
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/dist-packages
(from Jinja2<4,>=2.11->mlflow[jupyter]) (3.0.2)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.10/dist-package
s (from matplotlib<4->mlflow[jupyter]) (1.3.1)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.10/dist-packages (f
rom matplotlib<4->mlflow[jupyter]) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.10/dist-packag
es (from matplotlib<4->mlflow[jupyter]) (4.55.0)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.10/dist-packag
es (from matplotlib<4->mlflow[jupyter]) (1.4.7)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.10/dist-package
s (from matplotlib<4->mlflow[jupyter]) (3.2.0)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (f
rom pandas->dagshub) (2024.2)
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.10/dist-packages
(from pandas->dagshub) (2024.2)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from
python-dateutil->dagshub) (1.16.0)
Requirement already satisfied: markdown-it-py>=2.2.0 in /usr/local/lib/python3.10/dist-pa
ckages (from rich>=13.1.0->dagshub) (3.0.0)
Requirement already satisfied: joblib>=1.2.0 in /usr/local/lib/python3.10/dist-packages (
from scikit-learn<2->mlflow[jupyter]) (1.4.2)
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3.10/dist-pac
kages (from scikit-learn<2->mlflow[jupyter]) (3.5.0)
Requirement already satisfied: greenlet!=0.4.17 in /usr/local/lib/python3.10/dist-package
s (from sqlalchemy<3,>=1.4.0->mlflow[jupyter]) (3.1.1)
Collecting botocore<1.36.0,>=1.35.73 (from boto3->dagshub)
  Downloading botocore-1.35.73-py3-none-any.whl.metadata (5.7 kB)
Collecting jmespath<2.0.0,>=0.7.1 (from boto3->dagshub)
  Downloading jmespath-1.0.1-py3-none-any.whl.metadata (7.6 kB)
Collecting s3transfer<0.11.0,>=0.10.0 (from boto3->dagshub)
  Downloading s3transfer-0.10.4-py3-none-any.whl.metadata (1.7 kB)
Collecting marshmallow<4.0.0,>=3.18.0 (from dataclasses-json->dagshub)
  Downloading marshmallow-3.23.1-py3-none-any.whl.metadata (7.5 kB)
Collecting typing-inspect<1,>=0.4.0 (from dataclasses-json->dagshub)
```

```
  Downloading typing_inspect-0.9.0-py3-none-any.whl.metadata (1.5 kB)
Requirement already satisfied: yarl<2.0,>=1.6 in /usr/local/lib/python3.10/dist-packages
(from gql[requests]->dagshub) (1.17.2)
Collecting backoff<3.0,>=1.11.1 (from gql[requests]->dagshub)
  Downloading backoff-2.2.1-py3-none-any.whl.metadata (14 kB)
Requirement already satisfied: requests-toolbelt<2,>=1.0.0 in /usr/local/lib/python3.10/d
ist-packages (from gql[requests]->dagshub) (1.0.0)
Requirement already satisfied: exceptiongroup in /usr/local/lib/python3.10/dist-packages
(from anyio->httpx>=0.23.0->dagshub) (1.2.2)
Requirement already satisfied: google-auth~=2.0 in /usr/local/lib/python3.10/dist-package
s (from databricks-sdk<1,>=0.20.0->mlflow-skinny==2.18.0->mlflow[jupyter]) (2.27.0)
Requirement already satisfied: smmap<6,>=3.0.1 in /usr/local/lib/python3.10/dist-packages
(from gitdb<5,>=4.0.1->GitPython>=3.1.29->dagshub) (5.0.1)
Requirement already satisfied: zipp>=3.20 in /usr/local/lib/python3.10/dist-packages (fro
m importlib-metadata!=4.7.0,<9,>=3.7.0->mlflow-skinny==2.18.0->mlflow[jupyter]) (3.21.0)
Requirement already satisfied: mdurl~=0.1 in /usr/local/lib/python3.10/dist-packages (fro
m markdown-it-py>=2.2.0->rich>=13.1.0->dagshub) (0.1.2)
Requirement already satisfied: deprecated>=1.2.6 in /usr/local/lib/python3.10/dist-packag
es (from opentelemetry-api<3,>=1.9.0->mlflow-skinny==2.18.0->mlflow[jupyter]) (1.2.15)
Requirement already satisfied: opentelemetry-semantic-conventions==0.49b2 in /usr/local/l
ib/python3.10/dist-packages (from opentelemetry-sdk<3,>=1.9.0->mlflow-skinny==2.18.0->mlf
low[jupyter]) (0.49b2)
Requirement already satisfied: annotated-types>=0.6.0 in /usr/local/lib/python3.10/dist-p
ackages (from pydantic>=2.0.0->dagshub-annotation-converter>=0.1.0->dagshub) (0.7.0)
Requirement already satisfied: pydantic-core==2.23.4 in /usr/local/lib/python3.10/dist-pa
ckages (from pydantic>=2.0.0->dagshub-annotation-converter>=0.1.0->dagshub) (2.23.4)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist
-packages (from requests<3,>=2.17.3->mlflow-skinny==2.18.0->mlflow[jupyter]) (3.4.0)
Collecting mypy-extensions>=0.3.0 (from typing-inspect<1,>=0.4.0->dataclasses-json->dagsh
ub)
  Downloading mypy_extensions-1.0.0-py3-none-any.whl.metadata (1.1 kB)
Requirement already satisfied: multidict>=4.0 in /usr/local/lib/python3.10/dist-packages
(from yarl<2.0,>=1.6->gql[requests]->dagshub) (6.1.0)
Requirement already satisfied: propcache>=0.2.0 in /usr/local/lib/python3.10/dist-package
s (from yarl<2.0,>=1.6->gql[requests]->dagshub) (0.2.0)
Requirement already satisfied: wrapt<2,>=1.10 in /usr/local/lib/python3.10/dist-packages
(from deprecated>=1.2.6->opentelemetry-api<3,>=1.9.0->mlflow-skinny==2.18.0->mlflow[jupyt
er]) (1.16.0)
Requirement already satisfied: pyasn1-modules>=0.2.1 in /usr/local/lib/python3.10/dist-pa
ckages (from google-auth~=2.0->databricks-sdk<1,>=0.20.0->mlflow-skinny==2.18.0->mlflow[j
upyter]) (0.4.1)
Requirement already satisfied: rsa<5,>=3.1.4 in /usr/local/lib/python3.10/dist-packages (
from google-auth~=2.0->databricks-sdk<1,>=0.20.0->mlflow-skinny==2.18.0->mlflow[jupyter])
(4.9)
Requirement already satisfied: pyasn1<0.7.0,>=0.4.6 in /usr/local/lib/python3.10/dist-pac
kages (from pyasn1-modules>=0.2.1->google-auth~=2.0->databricks-sdk<1,>=0.20.0->mlflow-sk
inny==2.18.0->mlflow[jupyter]) (0.6.1)
Downloading icecream-2.1.3-py2.py3-none-any.whl (8.4 kB)
Downloading dagshub-0.3.45-py3-none-any.whl (252 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 252.2/252.2 kB 11.3 MB/s eta 0:00:00
Downloading mlflow_skinny-2.18.0-py3-none-any.whl (5.8 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 5.8/5.8 MB 68.5 MB/s eta 0:00:00
Downloading alembic-1.14.0-py3-none-any.whl (233 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 233.5/233.5 kB 18.0 MB/s eta 0:00:00
Downloading appdirs-1.4.4-py2.py3-none-any.whl (9.6 kB)
Downloading asttokens-3.0.0-py3-none-any.whl (26 kB)
Downloading colorama-0.4.6-py2.py3-none-any.whl (25 kB)
Downloading dacite-1.6.0-py3-none-any.whl (12 kB)
Downloading dagshub_annotation_converter-0.1.2-py3-none-any.whl (33 kB)
Downloading docker-7.1.0-py3-none-any.whl (147 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 147.8/147.8 kB 11.8 MB/s eta 0:00:00
Downloading executing-2.1.0-py2.py3-none-any.whl (25 kB)
Downloading graphene-3.4.3-py2.py3-none-any.whl (114 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 114.9/114.9 kB 9.6 MB/s eta 0:00:00
Downloading gunicorn-23.0.0-py3-none-any.whl (85 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 85.0/85.0 kB 7.6 MB/s eta 0:00:00
Downloading pathvalidate-3.2.1-py3-none-any.whl (23 kB)
Downloading treelib-1.7.0-py3-none-any.whl (18 kB)
Downloading boto3-1.35.73-py3-none-any.whl (139 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 139.2/139.2 kB 11.2 MB/s eta 0:00:00
Downloading dataclasses_json-0.6.7-py3-none-any.whl (28 kB)
Downloading mlflow-2.18.0-py3-none-any.whl (27.3 MB)
```

```
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 27.3/27.3 MB 61.5 MB/s eta 0:00:00
Downloading backoff-2.2.1-py3-none-any.whl (15 kB)
Downloading botocore-1.35.73-py3-none-any.whl (13.1 MB)
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 13.1/13.1 MB 88.8 MB/s eta 0:00:00
Downloading databricks_sdk-0.38.0-py3-none-any.whl (575 kB)
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 575.1/575.1 kB 35.6 MB/s eta 0:00:00
Downloading graphql_core-3.2.5-py3-none-any.whl (203 kB)
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 203.2/203.2 kB 16.9 MB/s eta 0:00:00
Downloading graphql_relay-3.2.0-py3-none-any.whl (16 kB)
Downloading jmespath-1.0.1-py3-none-any.whl (20 kB)
Downloading marshmallow-3.23.1-py3-none-any.whl (49 kB)
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 49.5/49.5 kB 3.9 MB/s eta 0:00:00
Downloading s3transfer-0.10.4-py3-none-any.whl (83 kB)
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 83.2/83.2 kB 6.6 MB/s eta 0:00:00
Downloading typing_inspect-0.9.0-py3-none-any.whl (8.8 kB)
Downloading gql-3.5.0-py2.py3-none-any.whl (74 kB)
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 74.0/74.0 kB 6.9 MB/s eta 0:00:00
Downloading Mako-1.3.6-py3-none-any.whl (78 kB)
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 78.6/78.6 kB 6.8 MB/s eta 0:00:00
Downloading mypy_extensions-1.0.0-py3-none-any.whl (4.7 kB)
Installing collected packages: appdirs, treelib, pathvalidate, mypy-extensions, marshmall
ow, Mako, jmespath, gunicorn, graphql-core, executing, dacite, colorama, backoff, asttoke
ns, typing-inspect, icecream, graphql-relay, docker, botocore, alembic, s3transfer, graph
ene, gql, dataclasses-json, databricks-sdk, dagshub-annotation-converter, boto3, mlflow-s
kinny, dagshub, mlflow
Successfully installed Mako-1.3.6 alembic-1.14.0 appdirs-1.4.4 asttokens-3.0.0 backoff-2.
2.1 boto3-1.35.73 botocore-1.35.73 colorama-0.4.6 dacite-1.6.0 dagshub-0.3.45 dagshub-ann
otation-converter-0.1.2 databricks-sdk-0.38.0 dataclasses-json-0.6.7 docker-7.1.0 executi
ng-2.1.0 gql-3.5.0 graphene-3.4.3 graphql-core-3.2.5 graphql-relay-3.2.0 gunicorn-23.0.0
icecream-2.1.3 jmespath-1.0.1 marshmallow-3.23.1 mlflow-2.18.0 mlflow-skinny-2.18.0 mypy-
extensions-1.0.0 pathvalidate-3.2.1 s3transfer-0.10.4 treelib-1.7.0 typing-inspect-0.9.0
```

In [4]:

```
import os
import plotly
import numpy as np
import pandas as pd
import plotly.express as px
import plotly.io as pio

pio.templates.default = "seaborn"
pio.renderers.default = "colab"

from icecream import ic
from urllib.parse import urlparse
from typing import Dict, Any, Optional
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report, prec
ision_recall_curve, roc_curve
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import StandardScaler

import dagshub
import mlflow
```

3. Load Dataset

In [5]:

```
pd.set_option("display.max_colwidth", None)
df = pd.read_csv("/content/Mobile Phone Pricing/dataset.csv")
df.head()
```

Out[5]:

	battery_power	blue	clock_speed	dual_sim	fc	four_g	int_memory	m_dep	mobile_wt	n_cores	...	px_height	px_width
0	842	0	2.2	0	1	0	7	0.6	188	2	...	20	756
1	1021	1	0.5	1	0	1	53	0.7	136	3	...	905	1988
2	563	1	0.5	1	2	1	41	0.9	145	5	...	1263	1716
3	615	1	2.5	0	0	0	10	0.8	131	6	...	1216	1786
4	1821	1	1.2	0	13	1	44	0.6	141	2	...	1208	1212

5 rows × 21 columns

4. Inspecting Dataset for `null` values

In [6]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2000 entries, 0 to 1999
Data columns (total 21 columns):
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   battery_power  2000 non-null   int64  
 1   blue           2000 non-null   int64  
 2   clock_speed    2000 non-null   float64
 3   dual_sim       2000 non-null   int64  
 4   fc             2000 non-null   int64  
 5   four_g         2000 non-null   int64  
 6   int_memory     2000 non-null   int64  
 7   m_dep          2000 non-null   float64
 8   mobile_wt      2000 non-null   int64  
 9   n_cores        2000 non-null   int64  
 10  pc             2000 non-null   int64  
 11  px_height      2000 non-null   int64  
 12  px_width       2000 non-null   int64  
 13  ram            2000 non-null   int64  
 14  sc_h           2000 non-null   int64  
 15  sc_w           2000 non-null   int64  
 16  talk_time      2000 non-null   int64  
 17  three_g        2000 non-null   int64  
 18  touch_screen   2000 non-null   int64  
 19  wifi           2000 non-null   int64  
 20  price_range    2000 non-null   int64  
dtypes: float64(2), int64(19)
memory usage: 328.2 KB
```

In [7]:

```
df.isnull().sum()
```

Out[7]:

	0
battery_power	0
blue	0
clock_speed	0
dual_sim	0
fc	0
four_g	0
int_memory	0
m_dep	0

mobile_wt	0
n_cores	0
pc	0
px_height	0
px_width	0
ram	0
sc_h	0
sc_w	0
talk_time	0
three_g	0
touch_screen	0
wifi	0
price_range	0

dtype: int64

The dataset is clean and ready to use.

5. Exploratory Data Analysis

In [8]:

```
df.describe()
```

Out[8]:

	battery_power	blue	clock_speed	dual_sim	fc	four_g	int_memory	m_dep	mobile_wt
count	2000.000000	2000.0000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000
mean	1238.518500	0.4950	1.522250	0.509500	4.309500	0.521500	32.046500	0.501750	140.249000
std	439.418206	0.5001	0.816004	0.500035	4.341444	0.499662	18.145715	0.288416	35.399655
min	501.000000	0.0000	0.500000	0.000000	0.000000	0.000000	2.000000	0.100000	80.000000
25%	851.750000	0.0000	0.700000	0.000000	1.000000	0.000000	16.000000	0.200000	109.000000
50%	1226.000000	0.0000	1.500000	1.000000	3.000000	1.000000	32.000000	0.500000	141.000000
75%	1615.250000	1.0000	2.200000	1.000000	7.000000	1.000000	48.000000	0.800000	170.000000
max	1998.000000	1.0000	3.000000	1.000000	19.000000	1.000000	64.000000	1.000000	200.000000

8 rows × 21 columns

Lets explore the target variable `price_range`.

In [9]:

```
df.price_range.value_counts()
```

Out[9]:

price_range	count
1	500
2	500
3	500

	0	co500
price_range		

dtype: int64

That's good, the dataset is balanced.

Let's see how many 4G phones there in the dataset.

In [10]:

```
df.four_g.value_counts()
```

Out[10]:

	count
four_g	
1	1043
0	957

dtype: int64

Let's see how many 3G phones are present.

In [11]:

```
df.three_g.value_counts()
```

Out[11]:

	count
three_g	
1	1523
0	477

dtype: int64

Checking the count of dual sim phones.

In [12]:

```
df.dual_sim.value_counts()
```

Out[12]:

	count
dual_sim	
1	1019
0	981

dtype: int64

Now let's plot some plots for better understanding of the dataset.

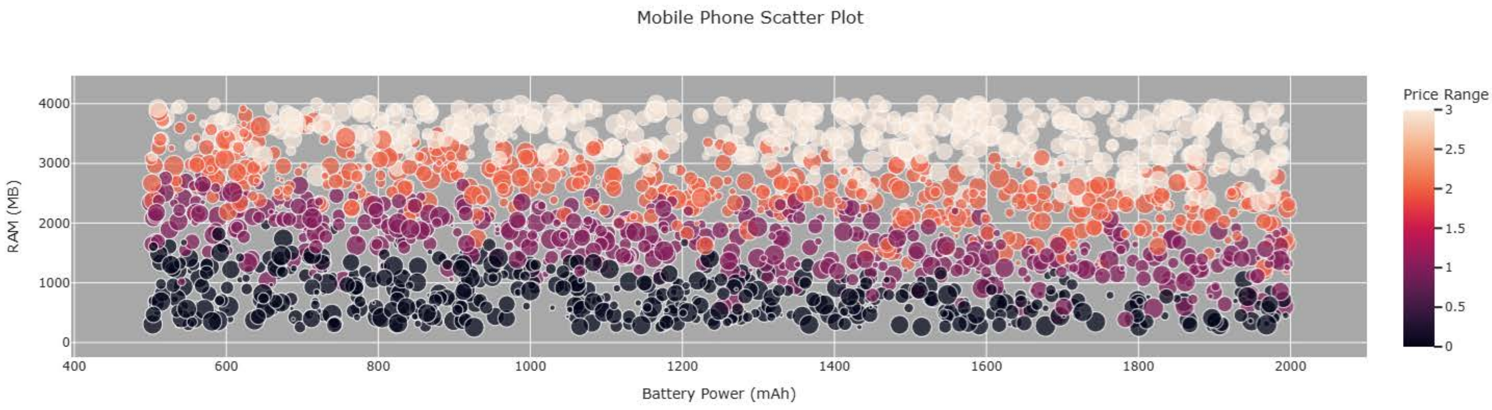
In [13]:

```
# Function for saving plotly plots as html to embed them later
with open('html_template.html', 'w') as f:
```

```
  f.write("""
  <!doctype html>
  <html>
  <head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  </head>

  <body>
  <!-- <h3>{{ heading }}</h3> -->
  {{ fig }}
  </body>
  </head>
  """)

def fig_to_html(fig: plotly.graph_objs._figure.Figure,
                # plot_heading: str,
                output_path: Optional[str]="output.html",
                template_path: Optional[str]="html_template.html") -> None:
  """
  Convert a plotly figure to an HTML.
  """
  # Create output directory if it doesn't exist
  output_dir = "plotly_html"
  os.makedirs(output_dir, exist_ok=True)

  from jinja2 import Template
  # Convert the figure to HTML
  plotly_jinja_data = {
      "fig": fig.to_html(full_html=False, include_plotlyjs="cdn"),
      # "heading": plot_heading
      }

  # Load the template
  with open(os.path.join(output_dir, output_path), "w", encoding="utf-8") as f:
    with open(template_path, "r", encoding="utf-8") as template_file:
      template = Template(template_file.read())
      f.write(template.render(plotly_jinja_data))
```

In [ ]:

```
# Mobile Phone Scatter plot
fig1 = px.scatter(
    df,
    x="battery_power",
    y="ram",
    color="price_range",
    size="int_memory",
    title="Mobile Phone Scatter Plot",
    labels={
        "battery_power": "Battery Power (mAh)",
        "ram": "RAM (MB)",
        "price_range": "Price Range",
        "int_memory": "Internal Memory (GB)"
    },
    hover_data=["battery_power", "ram", "price_range", "int_memory"],
)
fig1.update_layout(
    plot_bgcolor="darkgrey",
    template="seaborn"
)
fig_to_html(fig1, "mobile_phone_scatter_plot.html")
fig1.show()
```



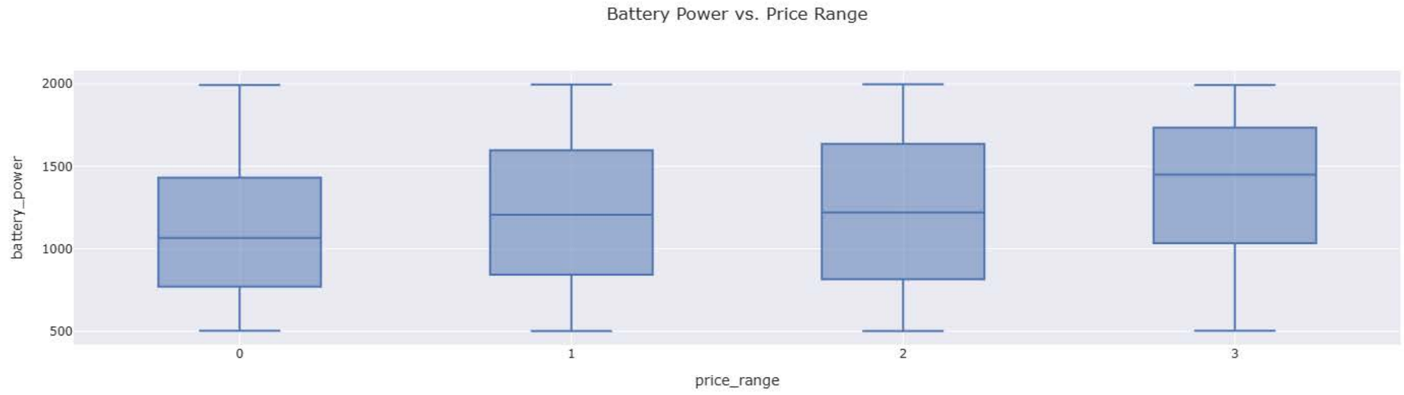
The above scatter plot visualizes the relationship between battery power and RAM for mobile, categorized by price range and sized by internal memory.

Observations:

1. **Relationship Between Battery Power and Price Range:** Battery power is evenly distributed across price ranges, with no strong positive or negative correlation. Phones in both lower (0) and higher (3) price ranges are spread across the entire spectrum of battery capacities (400 to 2000 mAh). This suggests that battery power is not a strong differentiating factor for pricing, as even budget phones offer competitive battery capacities.
2. **Relationship Between RAM and Price Range:** RAM increases consistently with price range. Phones in Price Range 0 (black points) are clustered at the lower end of the RAM spectrum (500–1500 MB). Phones in Price Range 3 (light orange points) are clustered at the higher end (3000–4000 MB). This indicates that RAM is a key differentiator for pricing, with higher RAM being a feature of premium phones.
3. **Impact of Internal Memory (Point Size):** Larger data points, representing higher internal memory, are more prevalent in higher price ranges. Phones in Price Range 3 not only have higher RAM but also tend to have higher internal memory (as seen from larger point sizes). Conversely, smaller data points (lower internal memory) are predominantly in Price Range 0 and Price Range 1. This indicates that internal memory, along with RAM, is another significant factor influencing phone pricing.
4. **Clustering:** Phones in lower price ranges (0 and 1) are clustered at the lower left, indicating a combination of low battery power, RAM, and internal memory. Phones in higher price ranges (2 and 3) dominate the upper region of the plot due to higher RAM and larger data points, representing more premium configurations.
5. **Insights:** Battery Power does not significantly affect price range, as high-capacity batteries are available across all ranges. RAM and Internal Memory are critical features for premium phones, as seen from their strong positive correlation with price range. Feature Trade-offs in Budget Phones (Price Range 0) tend to compromise on RAM and internal memory while offering competitive battery power.

In [ ]:

```
# 2. Battery Power vs. Price Range
fig2 = px.box(df, x="price_range", y="battery_power", title="Battery Power vs. Price Ran
ge")
fig_to_html(fig2, "battery_power_vs_price_range.html")
fig2.show()
```



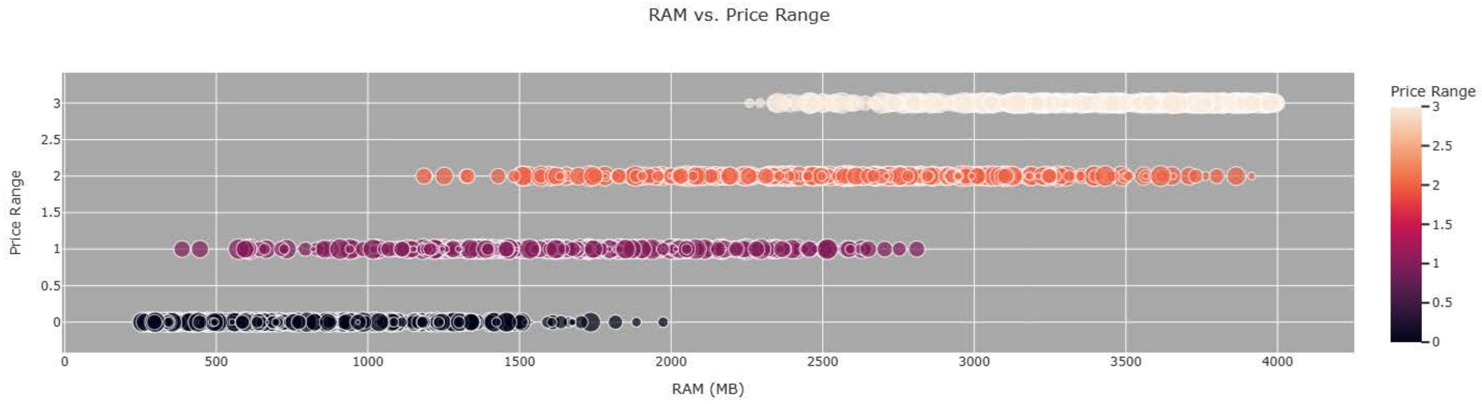
The above box plot gives insight into the distribution of battery power across different price categories.

Observations:

1. **Median Battery Power: The median battery power increases as the price range increases. Suggesting, that higher priced phones have higher battery power.**
2. **Price Range 0: Has a relatively lower median and interquartile range (IQR) for battery power compared to higher price ranges.**
3. **Price Range 1 to 3: There is an increasing trend where both the median and overall battery power distribution slightly increase, though the change is not drastic.**
4. **Spread & Variation: The whiskers of each box plot extend from the minimum to the maximum values, indicating the range of battery_power within each price range. There is a significant overlap in the range of battery power across all price categories, which indicates that some lower-priced devices may still offer comparable battery power to mid-range or high-priced devices.**
5. **Insights: Higher price ranges are generally associated with slightly higher battery power, but the overlap suggests that battery power is not a strong differentiator between price ranges. Manufacturers might prioritize other features besides battery power when justifying higher prices, or there might be diminishing returns in battery capacity for premium-priced devices.**

In [ ]:

```
# 3. RAM vs. Price Range
fig3 = px.scatter(
    df,
    x="ram",
    y="price_range",
    title="RAM vs. Price Range",
    color="price_range",
    size="int_memory",
    labels={
        "ram": "RAM (MB)",
        "price_range": "Price Range",
        "int_memory": "Internal Memory (GB)"
        },
    hover_data=["ram", "price_range", "int_memory"]
)
fig3.update_layout(
    plot_bgcolor="darkgrey",
    template="seaborn"
)
fig_to_html(fig3, "ram_vs_price_range.html")
fig3.show()
```



The scatter plot displays the relationship between RAM (in MB) and the price range of mobile phones.

Observations:

1. **RAM Influence:** The plot suggests that RAM is a significant factor influencing the price range of mobile phones. Phones with higher RAM tend to be more expensive.
2. **Clustering:** The clustering of data points suggests that there are specific price ranges associated with certain RAM configurations.
3. **Internal Memory:** The size of the data points reveals that internal memory is another important factor determining the price of mobile phones.
4. **Insights:** This plot indicates that RAM plays a vital role in pricing mobile phones. Higher RAM values lead to higher price ranges, and the size of the data points highlights the impact of internal memory on the cost.

In [ ]:

```
# 4. 3G/4G Availability by Price Range
fig4 = px.histogram(df, x="price_range", color="four_g", title="3G/4G Availability by Pri
ce Range",
                    barmode="group",
                    labels={"four_g": "4G Availability"},
                    # facet_col="four_g"
                   )
fig_to_html(fig4, "3g_4g_availability_by_price_range.html")
fig4.show()
```

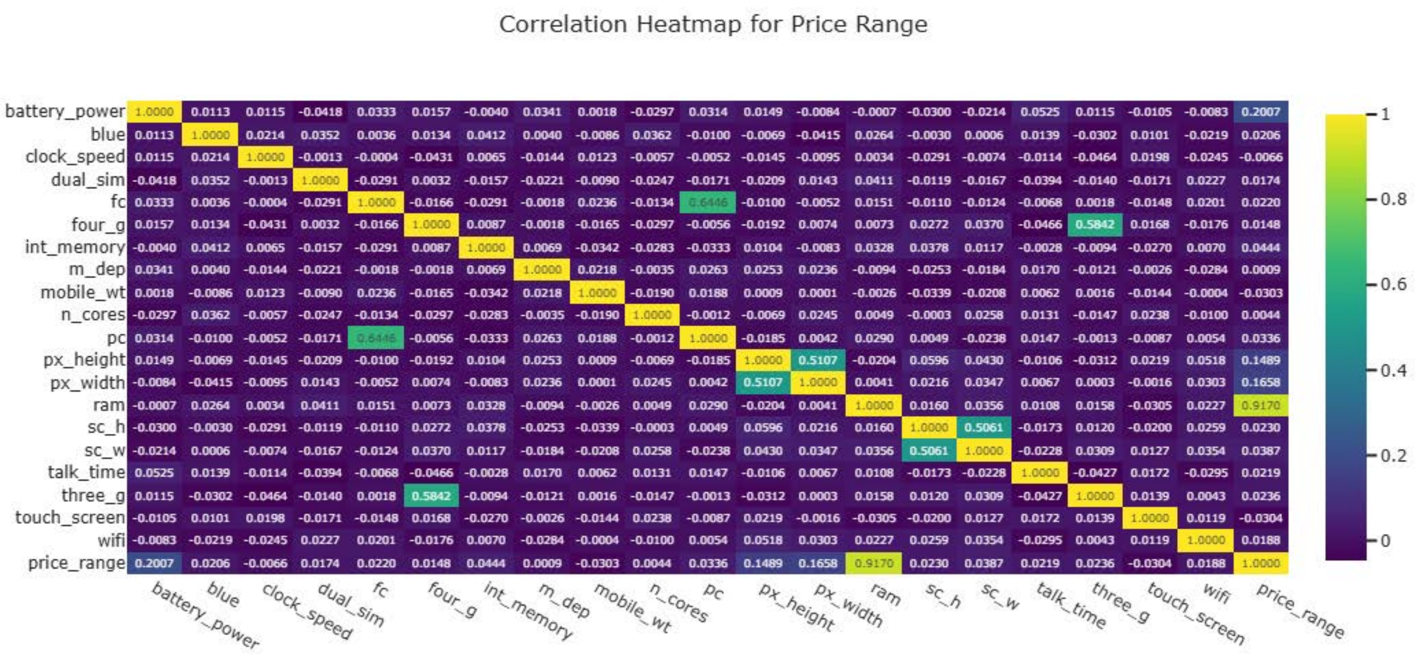
The scatter plot displays the relationship between RAM (in MB) and the price range of mobile phones.

Observations:

1. **Higher Price Range, Higher 4G Availability:** As the price range increases, the number of phones with 4G availability also increases. This trend is consistent across all price categories.
2. **Prevalence of 4G:** In all price ranges, a significant proportion of phones offer 4G capabilities. This indicates that 4G is a prevalent feature in the mobile phone market.
3. **Insights:** The plot suggests a strong correlation between price range and 4G availability. This indicates that phones with higher price tags are more likely to have 4G capabilities. This observation aligns with the expectation that newer and higher-end phones are more likely to incorporate advanced features like 4G.

In [ ]:

```
corr_matrix = df.corr()  # Calculate the correlation matrix

# Create the heatmap using px.imshow
fig5 = px.imshow(
    corr_matrix,
    text_auto=".4f",  # Display correlation values rounded to 2 decimal places
    aspect="auto",  # Adjust aspect ratio for better visualization
    color_continuous_scale="viridis",  # Choose a color scale
    title="Correlation Heatmap for Price Range"
)

fig5.show()
```



6. Preparing Training Data

In [14]:

```
# Copying dataset
ds = df.copy(deep=True)
```

In [15]:

```
# Separating features and target
X = ds.drop("price_range", axis=1)
y = ds.price_range
```

In [16]:

```
# Creating train-test-split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=ds.pri
ce_range, random_state=42)
ic(X_train.shape, X_test.shape, y_train.shape, y_test.shape);
```

```
ic| X_train.shape: (1600, 20)
    X_test.shape: (400, 20)
    y_train.shape: (1600,)
    y_test.shape: (400,)
```

Now we are ready to train classification models.

7. Setting Up DAGsHub & mlflow

First, let's set up dagshub for experiment tracking.

In [ ]:

```
!dagshub login
```

```
                          □□□ AUTHORIZATION REQUIRED □□□



Open the following link in your browser to authorize the client:
https://dagshub.com/login/oauth/authorize?state=fd6cd198-2afd-4b69-89a2-32dd30ed24aa&clie
nt_id=32b60ba385aa7cecf24046d8195a71c07dd345d9657977863b52e7748e0f0f28&middleman_request_
id=4d96426b0bfbc129b62ee5f94cc23f785902c514c0c4162c749344a0f35bac36


⠋ Waiting for authorization
□ OAuth token added
```

In [ ]:

```
from google.colab import userdata
repo_owner_ = userdata.get('REPO_OWNER')
repo_name_ = userdata.get('REPO_NAME')
tracking_uri = userdata.get('MLFLOW_TRACKING_URI')

os.makedirs('tmp', exist_ok=True)
```

```
# Creating function to log experiments to mlflow
def create_experiment(experiment_name: str,run_name: str, run_metrics: Dict[str, Any], mo
del, model_name: str = None, artifact_paths: Dict[str, str] = {}, run_params: Dict[str, A
ny] = None, tag_dict: Dict[str, str] = {"tag1":"Linear Regression", "tag2":"House Rent P
rediction"}):

    try:
        dagshub.init(repo_owner=f"{repo_owner_}", repo_name=f"{repo_name_}", mlflow=True
)

        # You can get your MLlfow tracking uri from your dagshub repo by opening "Remote"
dropdown menu, go to "Experiments" tab and copy the MLflow experiment tracking uri and pa
ste below
        mlflow.set_tracking_uri(f"{tracking_uri}")

        mlflow.set_experiment(experiment_name)

        with mlflow.start_run(run_name=run_name):

            # log params
            if not run_params == None:
                for param in run_params:
                    mlflow.log_param(param, run_params[param])

            # log metrics
            for metric, value in run_metrics.items():
                if isinstance(value, list):
                    # If the metric is a list, log each value as a separate step
                    for step, v in enumerate(value):
                        mlflow.log_metric(metric, v, step=step)
                else:
                    # If it's a single value, log it normally
                    mlflow.log_metric(metric, value)

            tracking_url_type_store = urlparse(mlflow.get_tracking_uri()).scheme

            # log artifacts
            for artifact_name, path in artifact_paths.items():
                if path and os.path.exists(path):
                    if tracking_url_type_store != "file":
                        mlflow.log_artifact(
                            path,
                            # artifact_name
                        )
                elif path:
                    print(f"Warning: Artifact file not found: {path}")

            # log model
            if tracking_url_type_store != "file":
                # mlflow.sklearn.save_model(model, save_path)
                mlflow.sklearn.log_model(model, "sk_model")

            mlflow.set_tags(tag_dict)

        print(f'Run - {run_name} is logged to Experiment - {experiment_name}')
    except Exception as e:
        print(f"An error occurred: {str(e)}")
        import traceback
        traceback.print_exc()
```

8. Model Training & Evaluation

8.1 Logistic Regression Classifier

8.1.1 Model Training

In [17]:

```
np.random.seed(42)

# Create a pipeline
pipe = make_pipeline(StandardScaler(), LogisticRegression(max_iter=1000, solver="liblinea
r"))

# Create a parameter grid
param_grid = {
    'logisticregression__C': [0.01, 0.1, 1, 10, 100],  # Regularization parameter
    'logisticregression__penalty': ['l1', 'l2']  # Penalty type
}

# Create a GridSearchCV object
grid_search = GridSearchCV(pipe, param_grid, cv=5, scoring="accuracy")

# Fit the model
grid_search.fit(X_train, y_train)
```

Out[17]:

```
▸ GridSearchCV  i ?
▸ best_estimator_: Pipeline
    ▸ StandardScaler ?
    ▸ LogisticRegression ?
```

In [18]:

```
# Best estimator
grid_search.best_estimator_
```

Out[18]:

```
▸ Pipeline  i ?
  ▸ StandardScaler ?
  ▸ LogisticRegression ?
```

In [19]:

```
# Best score
grid_search.best_score_
```

Out[19]:

```
0.859375
```

8.1.2 Model Evaluation

In [20]:

```
# Train Set Score (Accuracy)
train_acc = grid_search.score(X_train, y_train)
print(f"Training Accuracy: {train_acc*100:.2f}%")

# Test Set Score (Accuracy)
test_acc = grid_search.score(X_test, y_test)
print(f"Testing Accuracy: {test_acc*100:.2f}%")
```

```
Training Accuracy: 89.38%
Testing Accuracy: 84.00%
```

In [21]:

```
# Making predictions on y_test
np.random.seed(42)
y_preds = grid_search.best_estimator_.predict(X_test)
```

In [22]:

```
# Making predictions on test set
pred_price = grid_search.best_estimator_.predict(pd.DataFrame(X_test.iloc[15].to_numpy()
, index=X_test.columns).T)
pred_probs = grid_search.best_estimator_.predict_proba(pd.DataFrame(X_test.iloc[15].to_n
umpy(), index=X_test.columns).T)
true_price = y_test.iloc[15]

classes_ = np.array(["Low Cost", "Medium Cost", "High Cost", "Very High Cost"])

print("Price Range Prediction for Logistic Regression Model:")
print("\tTest Set:")
print(f"""\t\tPredicted Price Range: {pred_price[0]}({classes_[pred_price[0]]}) | True Pr
ice Range: {true_price} ({classes_[true_price]})""")
print(f"""\t\tModel's Confidence on Prediction: {np.max(pred_probs):.2%}""")
ds.loc[X_test.iloc[15].name]
```

```
Price Range Prediction for Logistic Regression Model:
 Test Set:
  Predicted Price Range: 3(Very High Cost) | True Price Range: 3 (Very High Cost)
  Model's Confidence on Prediction: 70.69%
```

Out[22]:

	1985
battery_power	1829.0
blue	1.0
clock_speed	2.1
dual_sim	0.0
fc	8.0
four_g	0.0
int_memory	59.0
m_dep	0.1
mobile_wt	91.0
n_cores	5.0
pc	15.0
px_height	1457.0
px_width	1919.0
ram	3142.0
sc_h	16.0
sc_w	6.0
talk_time	5.0
three_g	1.0
touch_screen	1.0
wifi	1.0
price_range	3.0

dtype: float64

In [ ]:

```
# Classification Report
print(f"Logistic Regression Classification Report:\n\n{classification_report(y_test, y_pr
```

```
eds)}")
```

```
Logistic Regression Classification Report:

              precision    recall  f1-score   support

           0       0.99      1.00      1.00       100
           1       0.72      0.68      0.70       100
           2       0.68      0.70      0.69       100
           3       0.96      0.98      0.97       100

    accuracy                           0.84       400
   macro avg       0.84      0.84      0.84       400
weighted avg       0.84      0.84      0.84       400
```

Okay, let's analyze the performance metrics of your Logistic Regression model:

Metrics Analysis:

1. **Training Accuracy (89.38%):** This metric represents the model's accuracy on the training data. It indicates that the model correctly predicted the price range for approximately 89.38% of the mobile phones in the training set.
2. **Testing Accuracy (84.00%):** This metric represents the model's accuracy on the testing data, which is unseen data during training. It indicates that the model correctly predicted the price range for approximately 84.00% of the mobile phones in the testing set.
3. **Precision:** Precision measures the proportion of correctly predicted positive instances (for a specific class) out of all instances predicted as positive for that class.

   - Precision for class 0 (Low Price) is very high (0.99), meaning that when the model predicts a phone as 'Low Price,' it is almost always correct.
   - Precision for class 3 (Very High Price) is also high (0.96), indicating good accuracy in predicting 'Very High Price' phones.
   - Precision for classes 1 (Medium Price) and 2 (High Price) are lower (0.72 and 0.68 respectively), suggesting that the model has more difficulty accurately identifying these price ranges.
4. **Recall:** Recall measures the proportion of correctly predicted positive instances (for a specific class) out of all actual positive instances for that class.

   - Recall for class 0 (Low Price) is perfect (1.00), indicating that the model correctly identifies all 'Low Price' phones.
   - Recall for class 3 (Very High Price) is also high (0.98), meaning it captures most 'Very High Price' phones.
   - Recall for classes 1 (Medium Price) and 2 (High Price) are lower (0.68 and 0.70 respectively), indicating that the model misses some phones belonging to these price ranges.
5. **F1-score:** The F1-score is the harmonic mean of precision and recall, providing a balanced measure of both metrics.

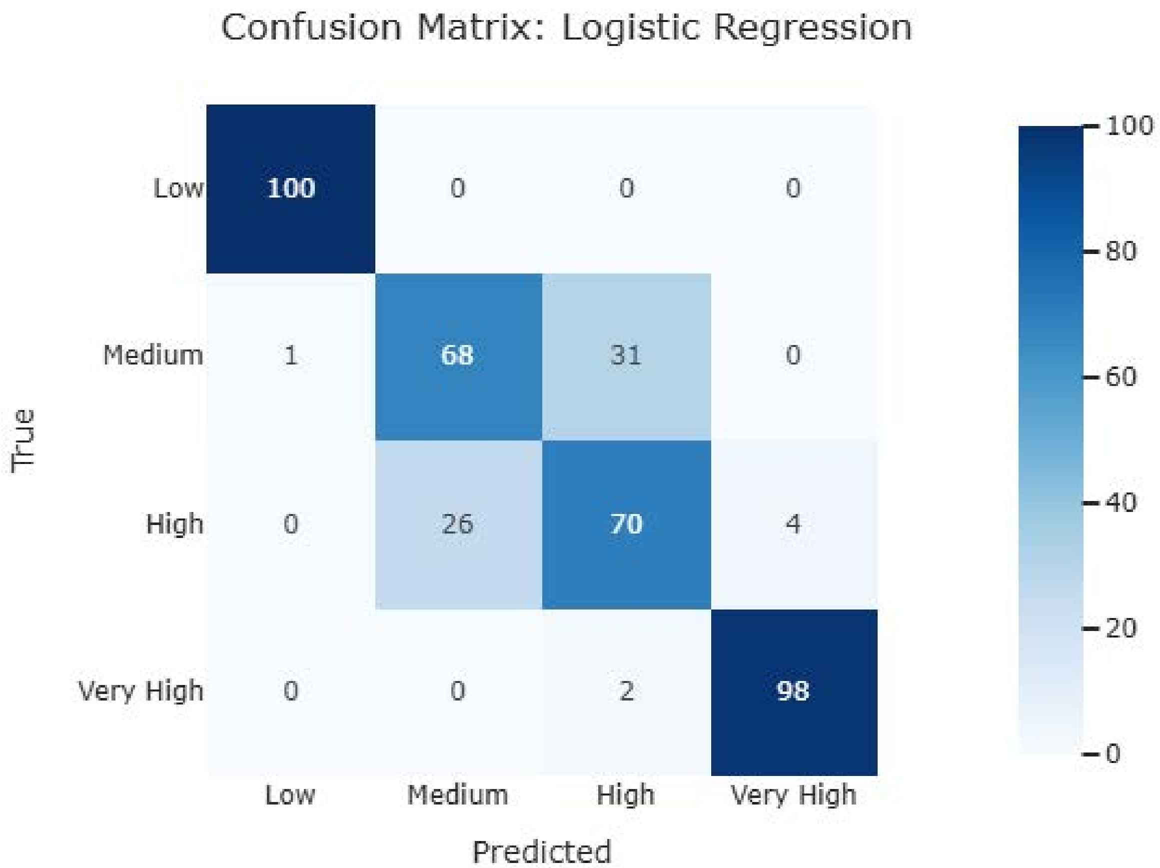
   - F1-scores generally follow the trends of precision and recall.
   - Higher F1-scores for classes 0 and 3 indicate better overall performance for these price ranges.
   - Lower F1-scores for classes 1 and 2 highlight the model's relatively weaker performance in these categories.

Model Performance and Generalization: The model demonstrates reasonably good overall performance, with an accuracy of 84.00% on the testing set. The slight drop in accuracy from training to testing (89.38% to 84.00%) indicates some degree of overfitting but it's not concerning. The model seems to generalize fairly well to unseen data.

Model Insights and Use: The model excels at predicting 'Low Price' and 'Very High Price' mobile phones, achieving high precision, recall, and F1-scores for these classes. The model has more difficulty accurately predicting 'Medium Price' and 'High Price' phones, as indicated by lower precision, recall, and F1-scores. Despite some weaknesses, the model can be useful for providing a preliminary price range prediction for mobile phones. It can assist in market analysis, product categorization, and potentially even pricing strategies.

In [ ]:

```
# Plotting the Confusion Matrix
def plot_confusion_matrix(y_test: np.ndarray, y_preds: np.ndarray, model_name: str, plot
_name: str) -> None:
    """Plot confusion matrix."""

    cm = confusion_matrix(y_test, y_preds)

    fig = px.imshow(
        cm,
        text_auto=True,  # Display values on the heatmap
        labels=dict(x="Predicted", y="True"),  # Set axis labels
        x=['Low', 'Medium', 'High', 'Very High'],  # Update x-axis labels
        y=['Low', 'Medium', 'High', 'Very High'],  # Update y-axis labels
        color_continuous_scale="Blues"  # Customize the color scale
    )

    fig.update_layout(title=f"Confusion Matrix: {model_name}")  # Set plot title
    fig_to_html(fig, f"{plot_name}")
    fig.show()  # Display plot

plot_confusion_matrix(y_test.to_numpy(), y_preds, "Logistic Regression", "confusion_matri
x_log_reg.html")
```

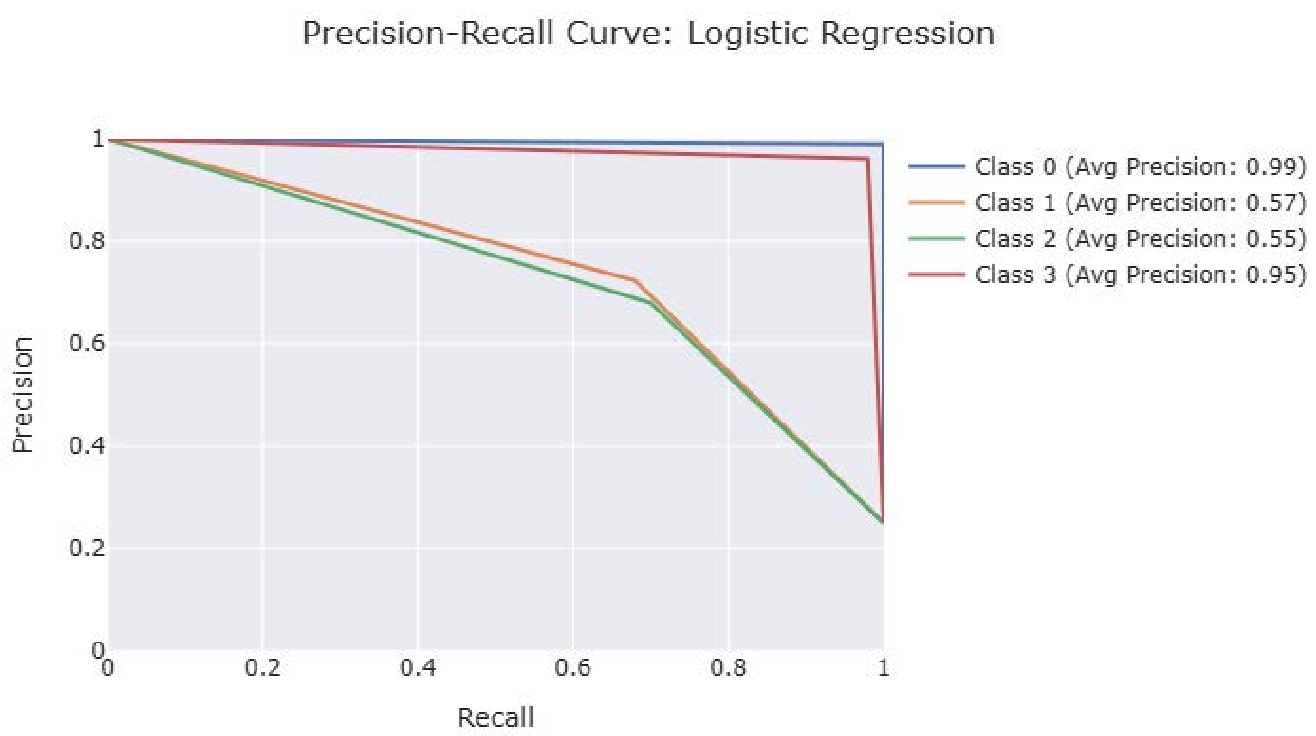


The above confusion matrix justifies the classification report.

In [ ]:

```
# Plotting Precision-Recall Curve
def plot_precision_recall_curve(y_test: np.ndarray, y_preds: np.ndarray, model_name: str
, plot_name: str) -> None:
    """Plot precision-recall curve."""

    import plotly.graph_objects as go
    from sklearn.metrics import precision_recall_curve, average_precision_score
    from sklearn.preprocessing import label_binarize

    # Assuming you have 'y_test' (true labels) and 'y_preds' (predicted labels)

    # 1. Binarize the labels
```

```
    n_classes = len(ds['price_range'].unique())  # Get the number of classes
    y_test_bin = label_binarize(y_test, classes=range(n_classes))
    y_preds_bin = label_binarize(y_preds, classes=range(n_classes))

    # 2. Create the Plotly figure
    fig = go.Figure()

    # 3. Calculate and plot precision-recall curves for each class
    for i in range(n_classes):
        precision, recall, _ = precision_recall_curve(y_test_bin[:, i], y_preds_bin[:, i
])
        avg_precision = average_precision_score(y_test_bin[:, i], y_preds_bin[:, i])

        fig.add_trace(go.Scatter(
            x=recall,
            y=precision,
            mode='lines',
            name=f"Class {i} (Avg Precision: {avg_precision:.2f})"
        ))

    # 4. Update layout for better visualization
    fig.update_layout(
        title=f"Precision-Recall Curve: {model_name}",
        xaxis_title="Recall",
        yaxis_title="Precision",
        xaxis_range=[0, 1],
        yaxis_range=[0, 1],
        showlegend=True
    )

    fig_to_html(fig, f"{plot_name}")

    fig.show()  # Display plot

plot_precision_recall_curve(y_test.to_numpy(), y_preds, "Logistic Regression", "pr_curve_
log_reg.html")
```



Overall Interpretation of PR Curve: **The Precision-Recall curve reinforces the observations from the classification report and accuracy metrics. The model demonstrates strong performance in identifying 'Low**

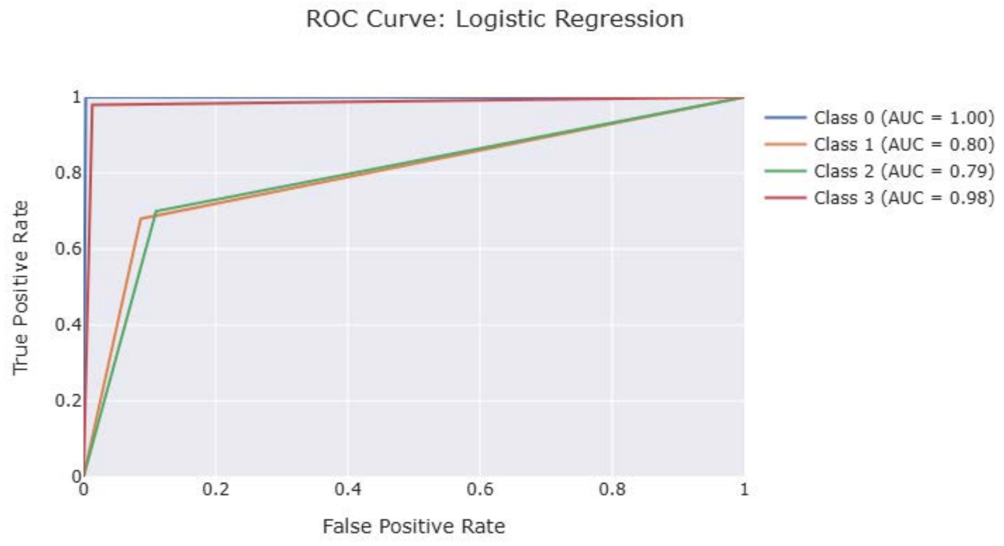
Price' and 'Very High Price' phones, but it struggles with the 'Medium Price' and 'High Price' categories. This could be due to overlapping features or less clear distinctions between these price ranges in the dataset.

Insights:

- **The model might be most useful in scenarios where correctly identifying 'Low Price' and 'Very High Price' phones is critical, even if it means some misclassification of 'Medium Price' and 'High Price' phones.**
- **If accurate prediction of all price ranges is equally important, further investigation and model improvement may be necessary, focusing on improving the performance for the 'Medium Price' and 'High Price' categories.**

In [ ]:

```
# Plotting ROC Curve
import plotly.graph_objects as go
from sklearn.metrics import roc_curve, auc, roc_auc_score
from sklearn.preprocessing import label_binarize

def plot_roc_curve(y_test: np.ndarray, y_preds: np.ndarray, model_name: str, plot_name:
str) -> None:
    """Plots the ROC curve."""

    # 1. Binarize the labels.
    n_classes = len(ds['price_range'].unique())  # Get the number of classes
    y_test_bin = label_binarize(y_test, classes=range(n_classes))
    y_preds_bin = label_binarize(y_preds, classes=range(n_classes))

    # 2. Create the figure.
    fig = go.Figure()

    # 3. Calculate the fpr and tpr.
    for i in range(n_classes):
        fpr, tpr, _ = roc_curve(y_test_bin[:, i], y_preds_bin[:, i])
        roc_auc = auc(fpr, tpr)

        fig.add_trace(go.Scatter(
            x=fpr,
            y=tpr,
            mode='lines',
            name=f"Class {i} (AUC = {roc_auc:.2f})"
        ))

    # 4. Update the plot.
    fig.update_layout(
        title=f"ROC Curve: {model_name}",
        xaxis_title="False Positive Rate",
        yaxis_title="True Positive Rate",
        xaxis_range=[0, 1],
        yaxis_range=[0, 1],
        showlegend=True
    )

    fig_to_html(fig, f"{plot_name}")

    fig.show()  # Display

plot_roc_curve(y_test.to_numpy(), y_preds, "Logistic Regression", "roc_curve_log_reg.html
")
```



Overall Interpretation: The ROC curve further supports the findings from the precision-recall curve and other metrics. The model demonstrates outstanding performance in identifying 'Low Price' and 'Very High Price' phones, achieving high true positive rates with low false positive rates. However, it faces challenges in discriminating between 'Medium Price' and 'High Price' phones, as indicated by the lower and more curved ROC curves for these classes.

Insights:

- The model is highly reliable for scenarios where correctly identifying 'Low Price' and 'Very High Price' phones is crucial, even if it means some misclassification of 'Medium Price' and 'High Price' phones.
- If accurate prediction of all price ranges is equally important, further investigation and model improvement may be necessary, focusing on improving the discrimination ability for the 'Medium Price' and 'High Price' categories.

8.1.3 Logging Model

In [ ]:

```
# Logging Experiment
from datetime import datetime
experiment_name = "mob_price_pred_log_reg"
run_name = "run_"+str(datetime.now().strftime("%d-%m-%y_%H:%M:%S"))

run_metrics = {"train_acc": train_acc, "test_acc": test_acc}

artifact_paths = {"mob_scatter_plot": "/content/plotly_html/mobile_phone_scatter_plot.htm
l", "battery_power_vs_price_range": "/content/plotly_html/battery_power_vs_price_range.ht
ml", "ram_vs_price_range": "/content/plotly_html/ram_vs_price_range.html", "3g_4g_availab
ility_by_price_range": "/content/plotly_html/3g_4g_availability_by_price_range.html",
    "confusion_matrix": "/content/plotly_html/confusion_matrix_log_reg.html", "pr_curve":
"/content/plotly_html/pr_curve_log_reg.html", "roc_curve": "/content/plotly_html/roc_curv
e_log_reg.html",
                  }

run_params = {"penalty": grid_search.best_params_["logisticregression__penalty"], "C": g
rid_search.best_params_["logisticregression__C"]}

create_experiment(experiment_name, run_name, run_metrics, grid_search.best_estimator_, m
odel_name="log_reg", artifact_paths=artifact_paths, run_params=run_params, tag_dict={"tag
1": "Logistic Regression", "tag2": "Mobile Phone Price Prediction"})
```

```
Initialized MLflow to track repo "pranay.makxenia/ML_Projects"

Repository pranay.makxenia/ML_Projects initialized!

2024/11/28 14:12:37 WARNING mlflow.models.model: Model logged without a signature and inp
ut example. Please set `input_example` parameter when logging the model to auto infer the
```

```
model signature.
```

```
 View run run_28-11-24_14:12:27 at: https://dagshub.com/pranay.makxenia/ML_Projects.mlflow/#/experiments/13/runs/d87ee03b963f4205a86fd31ee4817bff
 View experiment at: https://dagshub.com/pranay.makxenia/ML_Projects.mlflow/#/experiments/13
Run - run_28-11-24_14:12:27 is logged to Experiment - mob_price_pred_log_reg
```

8.2 K-Nearest Neighbors Classifier

8.2.1 Model Training

In [23]:

```python
np.random.seed(42)

# Create a pipeline
pipe = make_pipeline(StandardScaler(), KNeighborsClassifier())

# Create a parameter grid
param_grid = {
    'kneighborsclassifier__n_neighbors': [3, 5, 7, 9, 11, 13, 15],   # Number of neighbors
    'kneighborsclassifier__weights': ['uniform', 'distance']  # Weighting scheme
}

# Create a GridSearchCV object
grid_search = GridSearchCV(pipe, param_grid, cv=5, scoring="accuracy")

# Fit the model
grid_search.fit(X_train, y_train)
```

Out[23]:

```
▸ GridSearchCV i ?
▸ best_estimator_: Pipeline
    ▸ StandardScaler ?
    ▸ KNeighborsClassifier ?
```

In [24]:

```python
# Best estimator
grid_search.best_estimator_
```

Out[24]:

```
▸ Pipeline i ?
  ▸ StandardScaler ?
  ▸ KNeighborsClassifier ?
```

In [25]:

```python
# Best score
grid_search.best_score_
```

Out[25]:

```
0.579375
```

8.2.2 Model Evaluation

In [26]:

```
# Train Set Score (Accuracy)
train_acc = grid_search.score(X_train, y_train)
print(f"Training Accuracy: {train_acc*100:.2f}%")

# Test Set Score (Accuracy)
test_acc = grid_search.score(X_test, y_test)
print(f"Testing Accuracy: {test_acc*100:.2f}%")
```

```
Training Accuracy: 100.00%
Testing Accuracy: 57.50%
```

In [27]:

```
# Making predictions on y_test
np.random.seed(42)
y_preds = grid_search.best_estimator_.predict(X_test)
```

In [28]:

```
# Making predictions on test set
pred_price = grid_search.best_estimator_.predict(pd.DataFrame(X_test.iloc[15].to_numpy()
, index=X_test.columns).T)
pred_probs = grid_search.best_estimator_.predict_proba(pd.DataFrame(X_test.iloc[15].to_n
umpy(), index=X_test.columns).T)
true_price = y_test.iloc[15]

classes_ = np.array(["Low Cost", "Medium Cost", "High Cost", "Very High Cost"])

print("Price Range Prediction for KNN Model:")
print("\tTest Set:")
print(f"""\t\tPredicted Price Range: {pred_price[0]}({classes_[pred_price[0]]}) | True Pr
ice Range: {true_price} ({classes_[true_price]})""")
print(f"""\t\tModel's Confidence on Prediction: {np.max(pred_probs):.2%}""")
ds.loc[X_test.iloc[15].name]
```

```
Price Range Prediction for KNN Model:
 Test Set:
  Predicted Price Range: 3(Very High Cost) | True Price Range: 3 (Very High Cost)
  Model's Confidence on Prediction: 67.91%
```

Out[28]:

	1985
battery_power	1829.0
blue	1.0
clock_speed	2.1
dual_sim	0.0
fc	8.0
four_g	0.0
int_memory	59.0
m_dep	0.1
mobile_wt	91.0
n_cores	5.0
pc	15.0
px_height	1457.0
px_width	1919.0
ram	3142.0
sc_h	16.0
sc_w	6.0

	1985
talk_time	5.0
three_g	1.0
touch_screen	1.0
wifi	1.0
price_range	3.0

dtype: float64

In [ ]:

```
# Classification Report
print(f"K-Nearest Neighbors Classification Report:\n\n{classification_report(y_test, y_preds)}")
```

```
K-Nearest Neighbors Classification Report:

              precision    recall  f1-score   support

           0       0.79      0.68      0.73       100
           1       0.41      0.45      0.43       100
           2       0.44      0.47      0.45       100
           3       0.72      0.70      0.71       100

    accuracy                           0.57       400
   macro avg       0.59      0.57      0.58       400
weighted avg       0.59      0.57      0.58       400
```

Okay, let's analyze the performance metrics of your K-Nearest Neighbors model:

Metrics Analysis:

1. **Training Accuracy (100.00%):** This metric represents the model's accuracy on the training data. A 100% training accuracy suggests that the model has perfectly memorized the training data. While this might seem impressive, it often indicates overfitting, where the model has learned the training data too well and may not generalize well to unseen data.
2. **Testing Accuracy (57.50%):** This metric represents the model's accuracy on the testing data, which is unseen data during training. A significantly lower testing accuracy (57.50%) compared to the training accuracy (100.00%) confirms the overfitting concern. The model's performance drops considerably when applied to new, unseen data.
3. **Precision:** Precision measures the proportion of correctly predicted positive instances (for a specific class) out of all instances predicted as positive for that class.

    - Precision for class 0 (Low Price) is relatively high (0.79), meaning that when the model predicts a phone as 'Low Price,' it is correct about 79% of the time.
    - Precision for class 3 (Very High Price) is also relatively good (0.72,) indicating decent accuracy in predicting 'Very High Price' phones.
    - Precision for classes 1 (Medium Price) and 2 (High Price) are lower (0.41 and 0.44 respectively), suggesting that the model has more difficulty accurately identifying these price ranges.
4. **Recall:** Recall measures the proportion of correctly predicted positive instances (for a specific class) out of all actual positive instances for that class.

    - Recall for class 0 (Low Price) is 0.68, indicating that the model correctly identifies about 68% of 'Low Price' phones.
    - Recall for classes 1, 2, and 3 is around 0.45, 0.47, and 0.70 respectively, indicating a moderate ability to capture phones belonging to these price ranges.
5. **F1-score:** The F1-score is the harmonic mean of precision and recall, providing a balanced measure of both metrics.

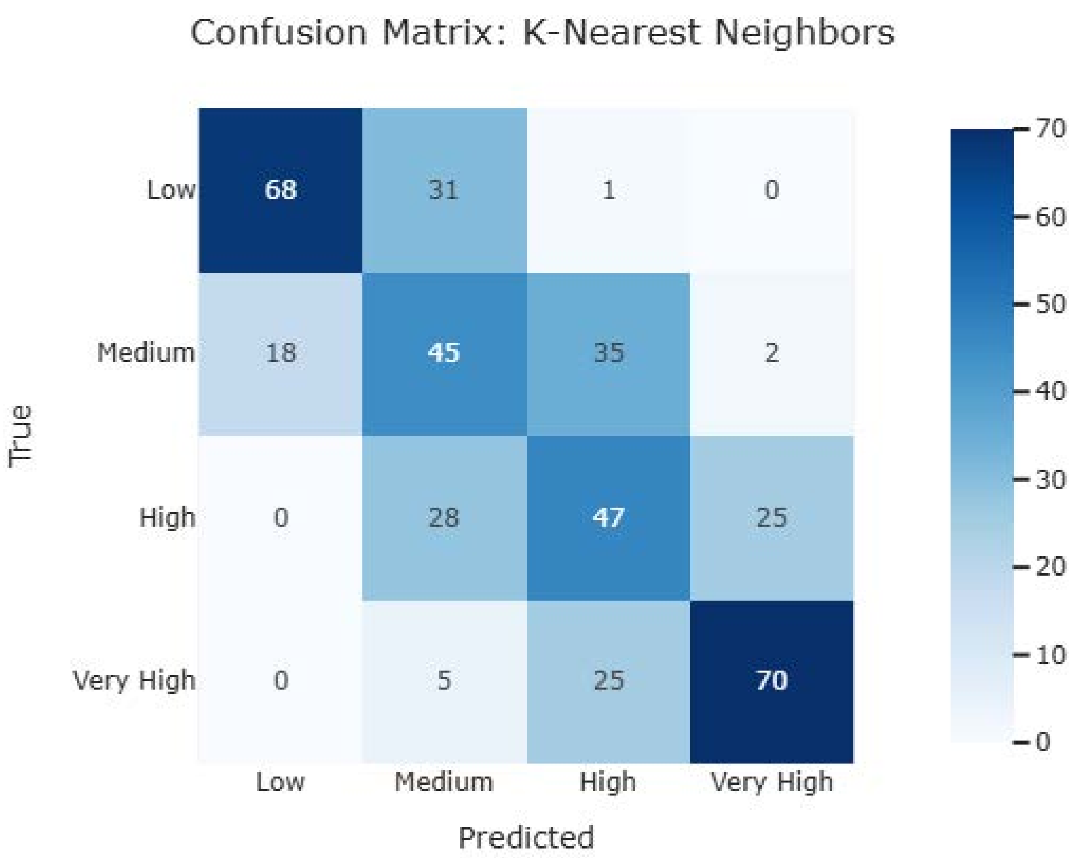
    - F1-scores generally follow the trends of precision and recall.
    - Class 0 has a relatively higher F1-score (0.73), while classes 1, 2, and 3 have lower F1-scores, reflecting the model's overall performance on each price range.

Model Performance and Generalization: The model demonstrates poor overall performance, with a testing accuracy of only 57.50%. This is significantly lower than the training accuracy, highlighting the overfitting issue. The large discrepancy between training and testing accuracy indicates that the model has not generalized well to unseen data. It has memorized the training data but fails to apply the learned patterns to new instances effectively.

Model Insights and Use: The model shows some ability to predict 'Low Price' and 'Very High Price' phones, although with limited accuracy. The model suffers from severe overfitting, resulting in poor generalization to unseen data. It has difficulty accurately predicting 'Medium Price' and 'High Price' phones. In its current state, the model is not reliable for predicting mobile phone price ranges. Its poor generalization makes it unsuitable for practical applications.

In [ ]:

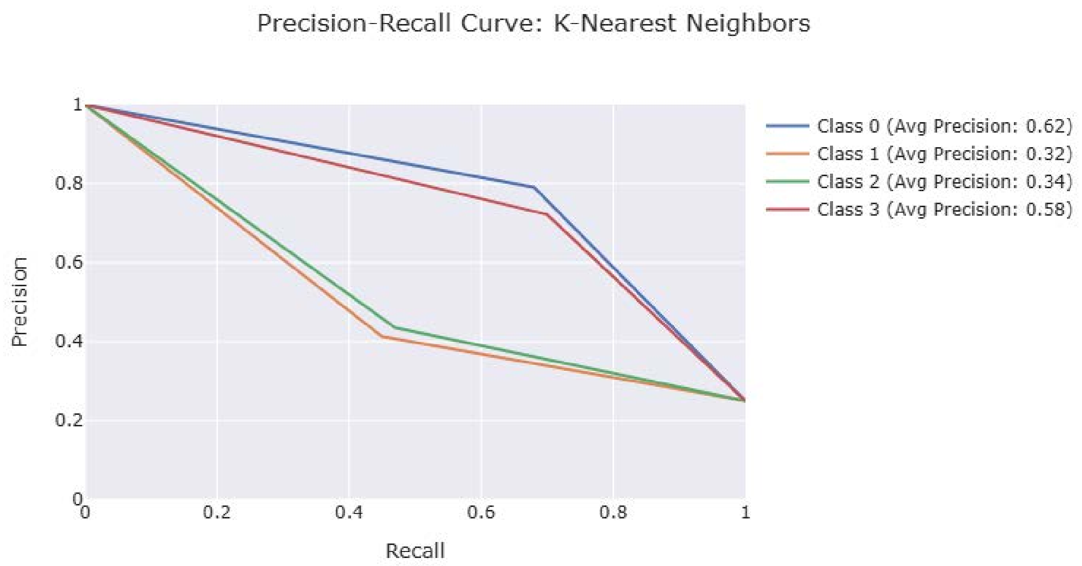
```
# Plotting the Confusion Matrix
plot_confusion_matrix(y_test.to_numpy(), y_preds, "K-Nearest Neighbors", "confusion_matri
x_knn.html")
```



The above confusion matrix justifies the classification report on K-Nearest Neighbour classifier.

In [ ]:

```
# Plotting Precision-Recall Curve
plot_precision_recall_curve(y_test.to_numpy(), y_preds, "K-Nearest Neighbors", "pr_curve_
knn.html")
```



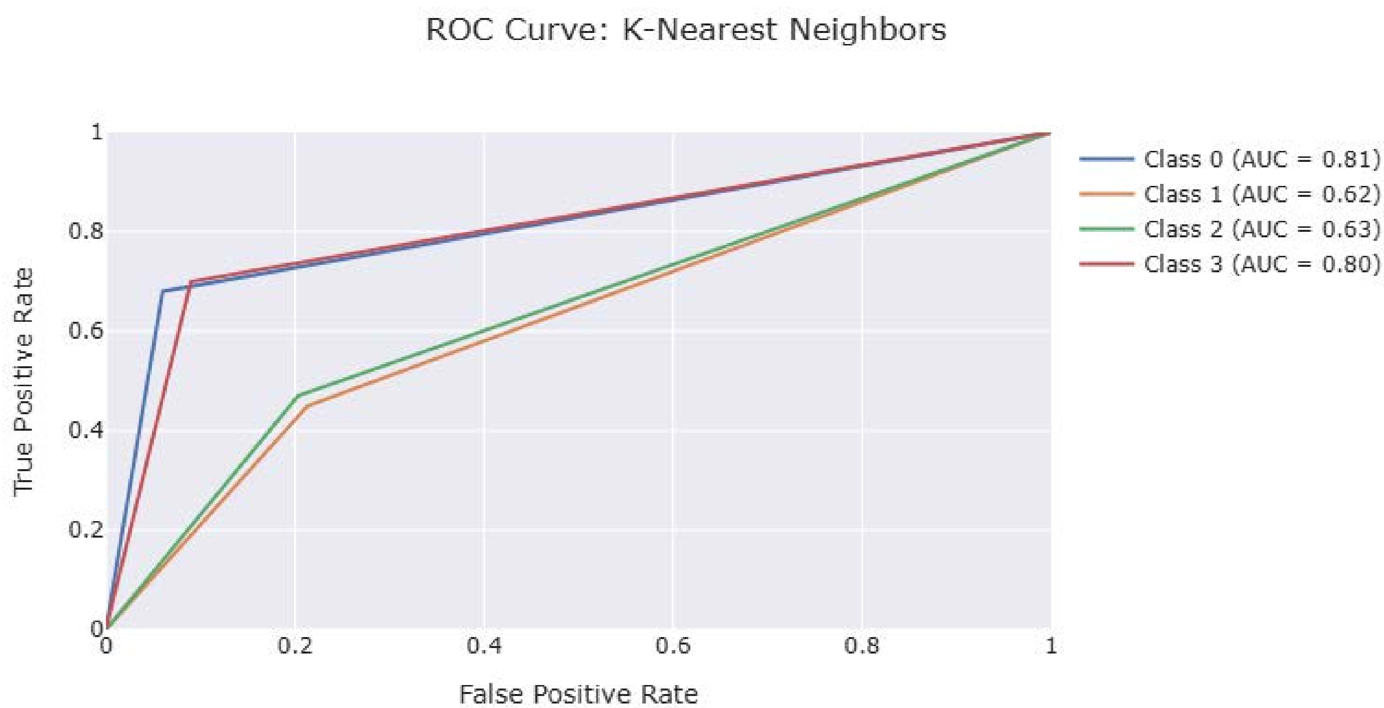
Overall Interpretation: The Precision-Recall curve reflects the observations from the classification report and accuracy metrics. The K-Nearest Neighbors model exhibits suboptimal performance, especially for the 'Medium Price' and 'High Price' categories. The curves for these classes are lower and more curved, indicating a significant trade-off between precision and recall.

Insights:

- The model might be somewhat useful in scenarios where correctly identifying 'Low Price' and 'Very High Price' phones is more important than achieving high accuracy across all price ranges. However, the overall performance is not ideal, particularly for 'Medium Price' and 'High Price' phones.
- Further investigation and model improvement are necessary to address the limitations and improve the precision and recall for all price categories. This might involve feature engineering, hyperparameter tuning, data augmentation, or exploring alternative algorithms.

In [ ]:

```
# Plotting ROC Curve
plot_roc_curve(y_test.to_numpy(), y_preds, "K-Nearest Neighbors", "roc_curve_knn.html")
```



Overall Interpretation: The ROC curve reinforces the findings from the precision-recall curve and other metrics. The K-Nearest Neighbors model exhibits suboptimal performance, especially for the 'Medium Price' and 'High Price' categories. The curves for these classes are lower and further away from the top-left corner, indicating a less effective discrimination ability. The model struggles to distinguish between these price ranges effectively.

Insights:

- The model's performance is relatively better for 'Low Price' and 'Very High Price' phones, but it struggles with 'Medium Price' and 'High Price' phones.
- The lower AUC scores for classes 1 and 2 suggest that the model has difficulty accurately classifying these price ranges.
- Further investigation and model improvement are necessary to address the limitations and improve the overall performance, particularly for the 'Medium Price' and 'High Price' categories. This might involve feature engineering, hyperparameter tuning, data augmentation, or exploring alternative algorithms.

8.2.3 Logging Model

In [ ]:

```
# Logging Experiment
from datetime import datetime
experiment_name = "mob_price_pred_knn"
run_name = "run_"+str(datetime.now().strftime("%d-%m-%y_%H:%M:%S"))

run_metrics = {"train_acc": train_acc, "test_acc": test_acc}

artifact_paths = {"mob_scatter_plot": "/content/plotly_html/mobile_phone_scatter_plot.html", "battery_power_vs_price_range": "/content/plotly_html/battery_power_vs_price_range.html", "ram_vs_price_range": "/content/plotly_html/ram_vs_price_range.html", "3g_4g_availability_by_price_range": "/content/plotly_html/3g_4g_availability_by_price_range.html",
    "confusion_matrix": "/content/plotly_html/confusion_matrix_knn.html", "pr_curve": "/content/plotly_html/pr_curve_knn.html", "roc_curve": "/content/plotly_html/roc_curve_knn.html",
                  }

run_params = {"n_neighbors": grid_search.best_params_["kneighborsclassifier__n_neighbors"], "weights": grid_search.best_params_["kneighborsclassifier__weights"]}

create_experiment(experiment_name, run_name, run_metrics, grid_search.best_estimator_, model_name="knn", artifact_paths=artifact_paths, run_params=run_params, tag_dict={"tag1": "KNN", "tag2": "Mobile Phone Price Prediction"})
```

```
Initialized MLflow to track repo "pranay.makxenia/ML_Projects"

Repository pranay.makxenia/ML_Projects initialized!
```

```
2024/11/28 14:20:01 INFO mlflow.tracking.fluent: Experiment with name 'mob_price_pred_knn' does not exist. Creating a new experiment.
2024/11/28 14:20:11 WARNING mlflow.models.model: Model logged without a signature and input example. Please set `input_example` parameter when logging the model to auto infer the model signature.
```

```
 View run run_28-11-24_14:20:00 at: https://dagshub.com/pranay.makxenia/ML_Projects.mlflow/#/experiments/14/runs/cf729e801b3c498bb5c6e6d38fb73c1f
 View experiment at: https://dagshub.com/pranay.makxenia/ML_Projects.mlflow/#/experiments/14
Run - run_28-11-24_14:20:00 is logged to Experiment - mob_price_pred_knn
```

8.3 Random Forest Classifier

8.3.1 Model Training

In [29]:

```
np.random.seed(42)

# Create a pipeline
pipe = make_pipeline(StandardScaler(), RandomForestClassifier(n_jobs=-1))

# Create a parameter grid
param_grid = {
    'randomforestclassifier__n_estimators': [100, 150, 250, 300],
    'randomforestclassifier__max_features': [10, 19, 'sqrt', 'log2'],
    'randomforestclassifier__max_depth': [None, 5, 10, 20],
    'randomforestclassifier__min_samples_split': [2, 5, 10],
    'randomforestclassifier__min_samples_leaf': [1, 2, 4]
}

# Create a GridSearchCV object
grid_search = GridSearchCV(pipe, param_grid, cv=5, scoring="accuracy")

# Fit the model
grid_search.fit(X_train, y_train)
```

```
/usr/local/lib/python3.10/dist-packages/numpy/ma/core.py:2820: RuntimeWarning:

invalid value encountered in cast
```

Out[29]:

```
▸ GridSearchCV i ?
▸ best_estimator_: Pipeline
    ▸ StandardScaler ?
    ▸ RandomForestClassifier ?
```

In [30]:

```
# Best estimator
grid_search.best_estimator_
```

Out[30]:

```
▸ Pipeline i ?
  ▸ StandardScaler ?
  ▸ RandomForestClassifier ?
```

In [31]:

```
# Best score
grid_search.best_score_
```

Out[31]:

```
0.8868750000000001
```

8.3.2 Model Evaluation

In [32]:

```
# Train Set Score (Accuracy)
train_acc = grid_search.score(X_train, y_train)
print(f"Training Accuracy: {train_acc*100:.2f}%")
```

```python
# Test Set Score (Accuracy)
test_acc = grid_search.score(X_test, y_test)
print(f"Testing Accuracy: {test_acc*100:.2f}%")
```

```
Training Accuracy: 97.81%
Testing Accuracy: 91.00%
```

In [33]:

```python
# Making predictions on y_test
np.random.seed(42)
y_preds = grid_search.best_estimator_.predict(X_test)
```

In [34]:

```python
# Making predictions on test set
pred_price = grid_search.best_estimator_.predict(pd.DataFrame(X_test.iloc[15].to_numpy()
, index=X_test.columns).T)
pred_probs = grid_search.best_estimator_.predict_proba(pd.DataFrame(X_test.iloc[15].to_n
umpy(), index=X_test.columns).T)
true_price = y_test.iloc[15]

classes_ = np.array(["Low Cost", "Medium Cost", "High Cost", "Very High Cost"])

print("Price Range Prediction for Random Forest Classifier Model:")
print("\tTest Set:")
print(f"""\t\tPredicted Price Range: {pred_price[0]}({classes_[pred_price[0]]}) | True Pr
ice Range: {true_price} ({classes_[true_price]})""")
print(f"""\t\tModel's Confidence on Prediction: {np.max(pred_probs):.2%}""")
ds.loc[X_test.iloc[15].name]
```

```
Price Range Prediction for Random Forest Classifier Model:
 Test Set:
  Predicted Price Range: 3(Very High Cost) | True Price Range: 3 (Very High Cost)
  Model's Confidence on Prediction: 99.20%
```

Out[34]:

	1985
battery_power	1829.0
blue	1.0
clock_speed	2.1
dual_sim	0.0
fc	8.0
four_g	0.0
int_memory	59.0
m_dep	0.1
mobile_wt	91.0
n_cores	5.0
pc	15.0
px_height	1457.0
px_width	1919.0
ram	3142.0
sc_h	16.0
sc_w	6.0
talk_time	5.0
three_g	1.0
touch_screen	1.0
wifi	1.0

price_range	3.0 1985

dtype: float64

In [ ]:

```
# Classification Report
print(f"Random Forest Classifier Classification Report:\n\n{classification_report(y_test, y_preds)}")
```

```
Random Forest Classifier Classification Report:

              precision    recall  f1-score   support

           0       0.96      0.95      0.95       100
           1       0.86      0.87      0.87       100
           2       0.86      0.87      0.87       100
           3       0.96      0.95      0.95       100

    accuracy                           0.91       400
   macro avg       0.91      0.91      0.91       400
weighted avg       0.91      0.91      0.91       400
```

Okay, let's analyze the performance metrics of your Random Forest Classifier model:

Metrics Analysis:

1. **Training Accuracy (97.81%): This metric represents the model's accuracy on the training data. It indicates that the model correctly predicted the price range for approximately 97.81% of the mobile phones in the training set. This high accuracy suggests that the model has learned the training data very well. However, it's essential to consider the testing accuracy to assess if the model is overfitting.**
2. **Testing Accuracy (91.00%): This metric represents the model's accuracy on the testing data, which is unseen data during training. It indicates that the model correctly predicted the price range for approximately 91.00% of the mobile phones in the testing set. This high testing accuracy, compared to the training accuracy, suggests that the model generalizes well to new, unseen data and is not significantly overfitting.**
3. **Precision: Precision measures the proportion of correctly predicted positive instances (for a specific class) out of all instances predicted as positive for that class.**
   - **Precision for classes 0 (Low Price) and 3 (Very High Price) is very high (0.96), meaning that when the model predicts a phone as 'Low Price' or 'Very High Price,' it is correct about 96% of the time.**
   - **Precision for classes 1 (Medium Price) and 2 (High Price) is also relatively good (0.86), indicating decent accuracy in predicting these price ranges.**
4. **Recall: Recall measures the proportion of correctly predicted positive instances (for a specific class) out of all actual positive instances for that class.**
   - **Recall for classes 0 (Low Price) and 3 (Very High Price) is 0.95, indicating that the model correctly identifies about 95% of 'Low Price' and 'Very High Price' phones.**
   - **Recall for classes 1 (Medium Price) and 2 (High Price) is slightly higher (0.87), suggesting a good ability to capture phones belonging to these price ranges.**
5. **F1-score: The F1-score is the harmonic mean of precision and recall, providing a balanced measure of both metrics.**
   - **F1-scores generally follow the trends of precision and recall.**
   - **Classes 0 and 3 have high F1-scores (0.95), while classes 1 and 2 have slightly lower but still good F1-scores (0.87), reflecting the model's overall performance on each price range.**

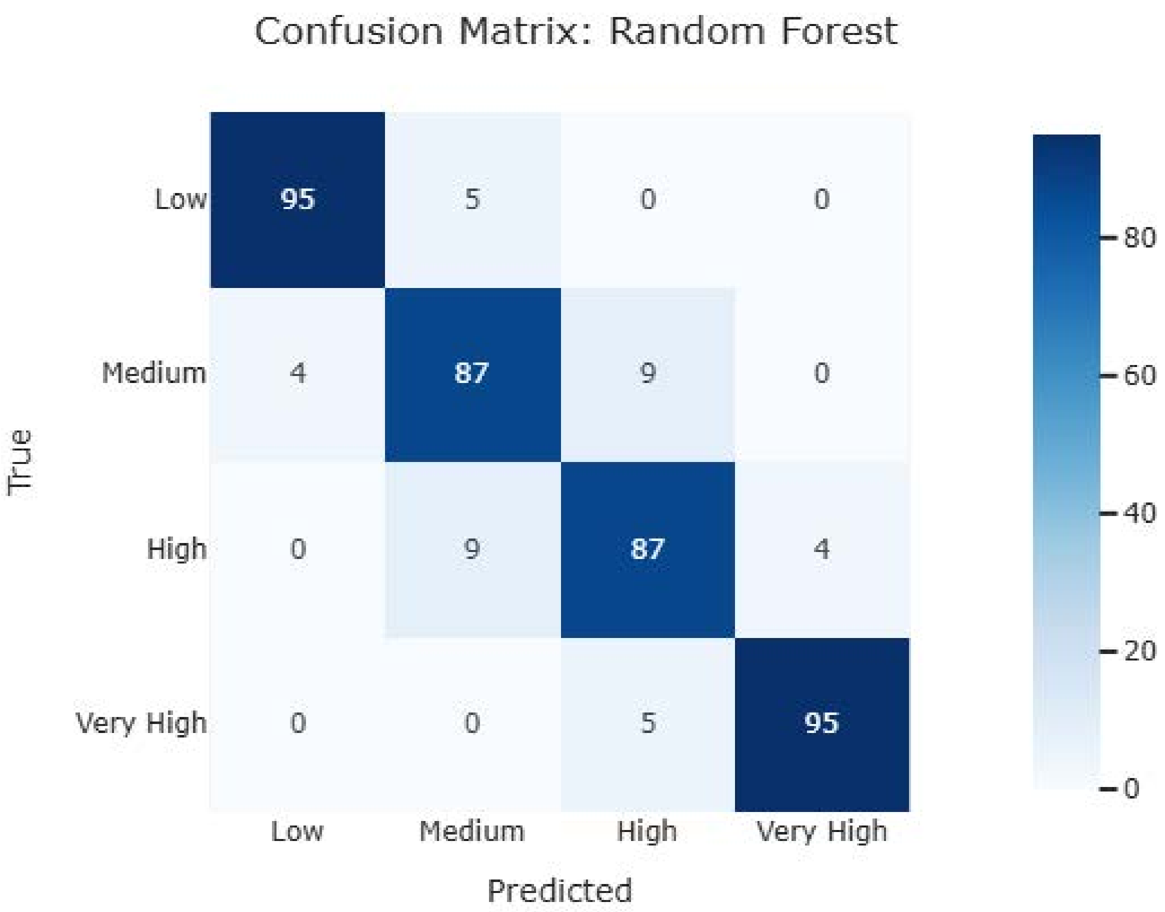
Model Performance and Generalization: The model demonstrates excellent overall performance, with a high testing accuracy of 91.00%. This indicates that the model is able to predict mobile phone price ranges accurately. The relatively small difference between training and testing accuracy suggests that the model generalizes well to unseen data and is not significantly overfitting. It has learned the underlying patterns in the data and can apply them effectively to new instances.

Model Insights and Use: The model excels at predicting all price ranges, achieving high precision, recall, and F1-scores for all classes. It demonstrates strong discrimination ability and generalization capabilities. While the

model performs very well, there is still room for potential improvement, particularly for classes 1 and 2, where the precision and recall are slightly lower than for classes 0 and 3. The Random Forest Classifier model is a highly effective and reliable tool for predicting mobile phone price ranges. It can be used for market analysis, product categorization, pricing strategies, and other applications where accurate price range prediction is crucial.

In [ ]:

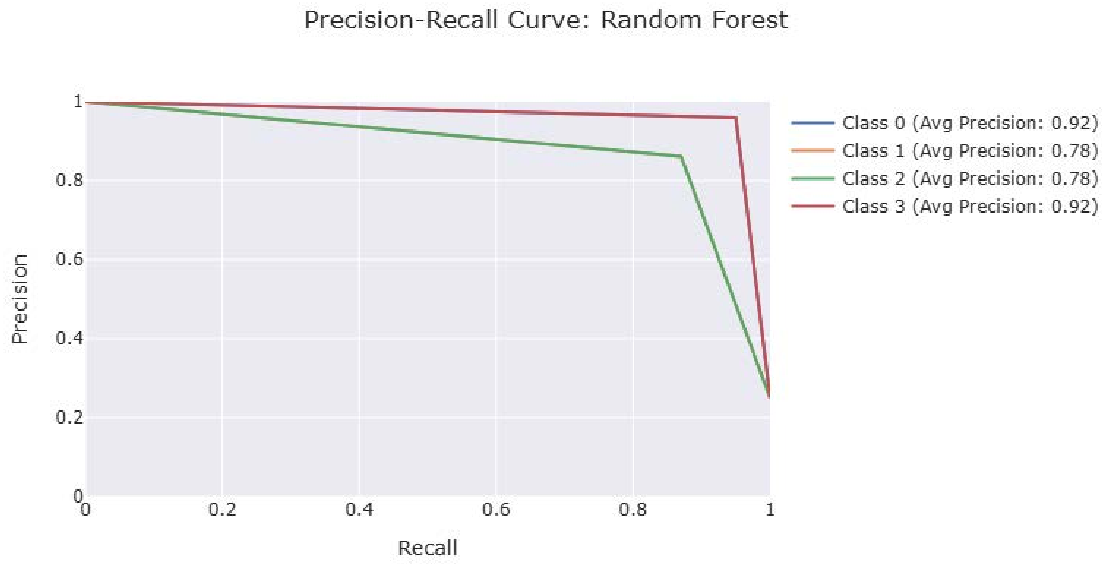
```
# Plotting the Confusion Matrix
plot_confusion_matrix(y_test.to_numpy(), y_preds, "Random Forest", "confusion_matrix_rf.h
tml")
```



The above confusion matrix justifies the classification report on Random Forest Classifier.

In [ ]:

```
# Plotting Precision-Recall Curve
plot_precision_recall_curve(y_test.to_numpy(), y_preds, "Random Forest", "pr_curve_rf.htm
l")
```



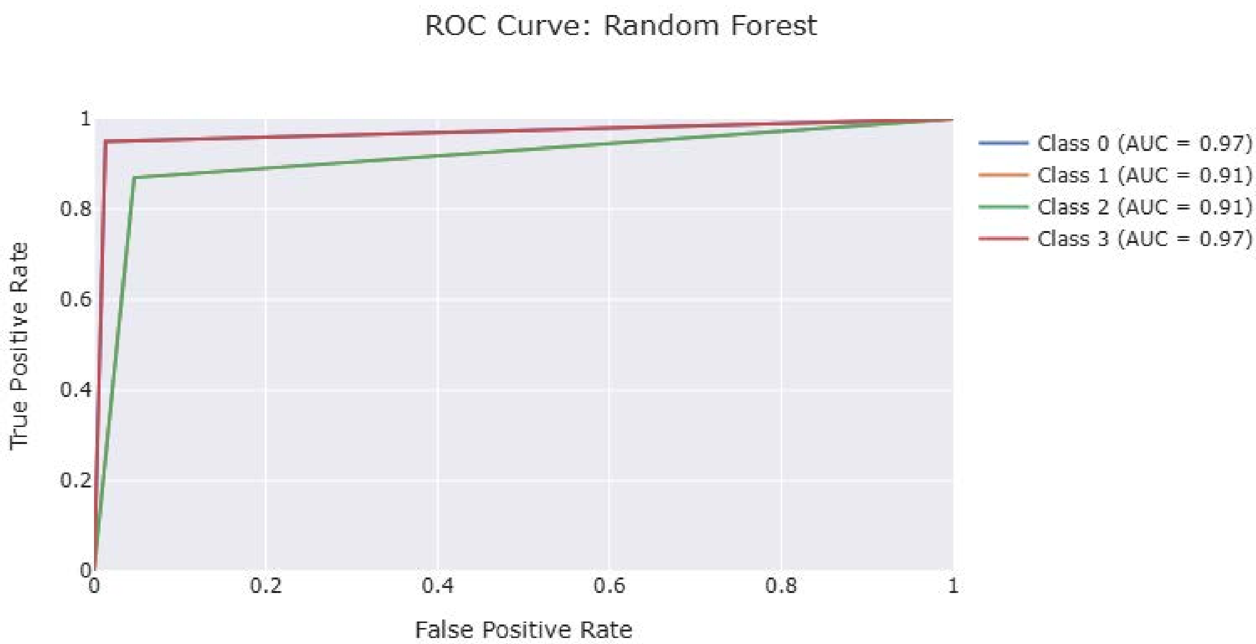
Overall Interpretation: The Precision-Recall curve demonstrates that the Random Forest Classifier performs very well across all price ranges, achieving high precision and recall. The curves for all classes are generally high and stay close to the top-right corner, indicating a good balance between precision and recall.

Insights:

- The Random Forest Classifier is a highly effective model for predicting mobile phone price ranges, achieving excellent performance for all classes.
- The model demonstrates a good ability to distinguish between different price ranges, minimizing both false positives and false negatives.
- The high average precision scores for all classes further support the model's strong performance.

In [ ]:

```
# Plotting ROC Curve
plot_roc_curve(y_test.to_numpy(), y_preds, "Random Forest", "roc_curve_rf.html")
```



Overall Interpretation: The ROC curve analysis demonstrates that the Random Forest Classifier performs exceptionally well across all price ranges, achieving high true positive rates with low false positive rates. The curves for all classes are generally high and close to the top-left corner, indicating a strong ability to discriminate between different price ranges.

Insights:

- **The Random Forest Classifier is a highly effective model for predicting mobile phone price ranges, achieving excellent performance for all classes.**
- **The model demonstrates a strong ability to distinguish between different price ranges, minimizing both false positives and false negatives.**
- **The high AUC scores for all classes further support the model's strong performance.**

8.3.3 Logging Model

In [ ]:

```
# Logging Experiment
from datetime import datetime
experiment_name = "mob_price_pred_random_forest"
run_name = "run_"+str(datetime.now().strftime("%d-%m-%y_%H:%M:%S"))

run_metrics = {"train_acc": train_acc, "test_acc": test_acc}

artifact_paths = {"mob_scatter_plot": "/content/plotly_html/mobile_phone_scatter_plot.html", "battery_power_vs_price_range": "/content/plotly_html/battery_power_vs_price_range.html", "ram_vs_price_range": "/content/plotly_html/ram_vs_price_range.html",
    "confusion_matrix": "/content/plotly_html/confusion_matrix_rf.html", "pr_curve": "/content/plotly_html/pr_curve_rf.html", "roc_curve": "/content/plotly_html/roc_curve_rf.html",
                  }

run_params = {
    "n_estimators": grid_search.best_params_["randomforestclassifier__n_estimators"],
    "max_features": grid_search.best_params_["randomforestclassifier__max_features"],
    "max_depth": grid_search.best_params_["randomforestclassifier__max_depth"],
    "min_samples_split": grid_search.best_params_["randomforestclassifier__min_samples_split"],
    "min_samples_leaf": grid_search.best_params_["randomforestclassifier__min_samples_leaf"]
}

create_experiment(experiment_name, run_name, run_metrics, grid_search.best_estimator_, model_name="random_forest", artifact_paths=artifact_paths, run_params=run_params, tag_dict={"tag1": "Random Forest Classifier", "tag2": "Mobile Phone Price Prediction"})
```

```
Initialized MLflow to track repo "pranay.makxenia/ML_Projects"

Repository pranay.makxenia/ML_Projects initialized!
```

```
2024/11/28 15:17:35 INFO mlflow.tracking.fluent: Experiment with name 'mob_price_pred_random_forest' does not exist. Creating a new experiment.
2024/11/28 15:17:47 WARNING mlflow.models.model: Model logged without a signature and input example. Please set `input_example` parameter when logging the model to auto infer the model signature.
```

```
 View run run_28-11-24_15:17:34 at: https://dagshub.com/pranay.makxenia/ML_Projects.mlflow/#/experiments/15/runs/237207ce11a8487898c3665851dd28af
 View experiment at: https://dagshub.com/pranay.makxenia/ML_Projects.mlflow/#/experiments/15
Run - run_28-11-24_15:17:34 is logged to Experiment - mob_price_pred_random_forest
```

9. Conclusion

After carefully analyzing the above trained models, we can clearly see that `Random Forest Classifier` **is the best model followed by** `Logistic Regression Classifier`**. We will still deploy all the three models, for comparison, on a streamlit app.**

Next

Next we will create a streamlit app to deploy the models for predicting mobile phone price.

In [ ]:

```
from google.colab import files
import shutil

def zip_and_download_folder(folder_path, zip_filename):
  shutil.make_archive(zip_filename, 'zip', folder_path)
  files.download(zip_filename + '.zip')

zip_and_download_folder('/content/plotly_html', 'plotly_html')
```

In [ ]:

```

```