```
Start coding or generate with AI.
```

Task

Deploy the trained model from the file "/content/WineQT.csv" in a production environment using Flask or FastAPI.

Load the data

Subtask:

Load the dataset into a pandas DataFrame.

Reasoning: Load the dataset into a pandas DataFrame and display the first few rows.

```
import pandas as pd

df = pd.read_csv('/content/WineQT.csv')
display(df.head())
```

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.9968	3.20	0.68	9.8
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.9970	3.26	0.65	9.8
3	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.9980	3.16	0.58	9.8
4	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4

Prepare the data

Subtask:

Preprocess the data by handling missing values, encoding categorical features, and scaling numerical features.

Reasoning: Identify and handle missing values.

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

```
fixed acidity           0
volatile acidity        0
citric acid             0
residual sugar          0
chlorides               0
free sulfur dioxide     0
total sulfur dioxide    0
density                 0
pH                      0
```

```
sulphates               0
alcohol                 0
quality                 0
Id                      0
dtype: int64
```

Reasoning: There are no missing values, so the next step is to identify and encode categorical features. Based on the column names and the head of the dataframe, there are no apparent categorical features that need encoding. All columns seem to be numerical. The next step is to scale the numerical features.

```
from sklearn.preprocessing import StandardScaler

# Exclude the 'quality' and 'Id' columns from scaling
numerical_features = df.select_dtypes(include=['float64', 'int64']).columns.tolist()
features_to_scale = [feature for feature in numerical_features if feature not in ['quality', 'Id']]

scaler = StandardScaler()
df_scaled = df.copy()
df_scaled[features_to_scale] = scaler.fit_transform(df_scaled[features_to_scale])

df_processed = df_scaled
display(df_processed.head())
```

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates
0	-0.521580	0.939332	-1.365027	-0.466421	-0.231395	-0.450467	-0.363610	0.555854	1.270695	-0.573658
1	-0.292593	1.941813	-1.365027	0.050060	0.234247	0.915920	0.643477	0.036165	-0.708928	0.130881
2	-0.292593	1.273492	-1.161568	-0.171289	0.107253	-0.060071	0.246745	0.140103	-0.325775	-0.045254
3	1.653789	-1.399789	1.483400	-0.466421	-0.252560	0.135127	0.429852	0.659792	-0.964363	-0.456235
4	-0.521580	0.939332	-1.365027	-0.466421	-0.231395	-0.450467	-0.363610	0.555854	1.270695	-0.573658

Train a model

Subtask:

Train a machine learning model on the prepared data.

Reasoning: Separate features and target, split data into training and testing sets, import and instantiate a RandomForestClassifier, and train the model.

```
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier

X = df_processed.drop('quality', axis=1)
y = df_processed['quality']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

model = RandomForestClassifier(random_state=42)
model.fit(X_train, y_train)
```

```
▾    RandomForestClassifier
RandomForestClassifier(random_state=42)
```

Save the model

Subtask:

Save the trained model to a file for later use.

Reasoning: Save the trained model to a file using joblib.

```
import joblib

joblib.dump(model, 'trained_model.joblib')
```

```
['trained_model.joblib']
```

Create a web application

Subtask:

Build a web application using Flask or FastAPI to serve the model.

Reasoning: Create a Python file named `app.py` and write the basic Flask application structure, including necessary imports and initialization.

```
%%writefile app.py
from flask import Flask, request, jsonify
import joblib
import pandas as pd

app = Flask(__name__)

# Load the trained model
# model = joblib.load('trained_model.joblib')

# Define a route for predictions
# @app.route('/predict', methods=['POST'])
# def predict():
#     try:
#         data = request.get_json()
#         # Assuming the input data is a dictionary matching the model's expected features
#         input_df = pd.DataFrame([data])
#         prediction = model.predict(input_df)
#         return jsonify({'prediction': prediction.tolist()})
#     except Exception as e:
#         return jsonify({'error': str(e)})

# if __name__ == '__main__':
#     app.run(debug=True)
```

```
Writing app.py
```

Reasoning: The file `app.py` has been created. Now, load the trained model and define the prediction route within the Flask application in `app.py`.

```
%%writefile app.py
from flask import Flask, request, jsonify
import joblib
import pandas as pd

app = Flask(__name__)

# Load the trained model
model = joblib.load('trained_model.joblib')

# Define a route for predictions
@app.route('/predict', methods=['POST'])
def predict():
    try:
        data = request.get_json()
        # Assuming the input data is a dictionary matching the model's expected features
        input_df = pd.DataFrame([data])
        # Ensure the order of columns matches the training data
        input_df = input_df[X_train.columns]
        prediction = model.predict(input_df)
        return jsonify({'prediction': prediction.tolist()})
    except Exception as e:
        return jsonify({'error': str(e)})

if __name__ == '__main__':
    # Running on 0.0.0.0 makes it accessible externally if needed
    app.run(host='0.0.0.0', debug=True)
```

```
Overwriting app.py
```

Load the model

Subtask:

Load the saved model within the web application.

Reasoning: Verify that the `app.py` file exists and contains the necessary code to load the trained model.

```
with open('app.py', 'r') as f:
    app_content = f.read()
    print(app_content)

import os
if os.path.exists('trained_model.joblib'):
    print("trained_model.joblib exists.")
else:
    print("trained_model.joblib does not exist.")
```

```
from flask import Flask, request, jsonify
import joblib
import pandas as pd

app = Flask(__name__)
```

```
# Load the trained model
model = joblib.load('trained_model.joblib')

# Define a route for predictions
@app.route('/predict', methods=['POST'])
def predict():
    try:
        data = request.get_json()
        # Assuming the input data is a dictionary matching the model's expected features
        input_df = pd.DataFrame([data])
        # Ensure the order of columns matches the training data
        input_df = input_df[X_train.columns]
        prediction = model.predict(input_df)
        return jsonify({'prediction': prediction.tolist()})
    except Exception as e:
        return jsonify({'error': str(e)})

if __name__ == '__main__':
    # Running on 0.0.0.0 makes it accessible externally if needed
    app.run(host='0.0.0.0', debug=True)

trained_model.joblib exists.
```

Make predictions

Subtask:

Create an API endpoint in the web application to accept input data and return predictions from the loaded model.

Reasoning: Review the app.py file and verify that the /predict endpoint is correctly defined and configured to handle POST requests, extract data from incoming JSON, convert it to a pandas DataFrame with correct column order, use the model for prediction, and return a JSON response.

```
with open('app.py', 'r') as f:
    app_content = f.read()
    print(app_content)
```

```
from flask import Flask, request, jsonify
import joblib
import pandas as pd

app = Flask(__name__)

# Load the trained model
model = joblib.load('trained_model.joblib')

# Define a route for predictions
@app.route('/predict', methods=['POST'])
def predict():
    try:
        data = request.get_json()
        # Assuming the input data is a dictionary matching the model's expected features
        input_df = pd.DataFrame([data])
        # Ensure the order of columns matches the training data
        input_df = input_df[X_train.columns]
        prediction = model.predict(input_df)
        return jsonify({'prediction': prediction.tolist()})
    except Exception as e:
        return jsonify({'error': str(e)})

if __name__ == '__main__':
    # Running on 0.0.0.0 makes it accessible externally if needed
    app.run(host='0.0.0.0', debug=True)
```

Test the application

Subtask:

Test the web application locally to ensure it works correctly.

Reasoning: Run the Flask application in the background and send a POST request with a sample payload to test the `/predict` endpoint. Then, print the response and terminate the Flask application process.

```
import subprocess
import time
import requests
import json

# Run the Flask app in the background
process = subprocess.Popen(['python', 'app.py'])

# Give the server a moment to start
time.sleep(5)

# Construct a sample payload (using the first row of X_test)
sample_payload = X_test.iloc[0].drop('Id').to_dict()

# Send a POST request to the /predict endpoint
url = 'http://127.0.0.1:5000/predict'  # Default Flask host and port
try:
    response = requests.post(url, json=sample_payload)
    response.raise_for_status() # Raise an exception for bad status codes
    print("Response from /predict endpoint:")
    print(response.json())
except requests.exceptions.RequestException as e:
    print(f"Error during request: {e}")
finally:
    # Terminate the Flask application process
    process.terminate()
    print("\nFlask application terminated.")
```

```
Response from /predict endpoint:
{'error': "name 'X_train' is not defined"}

Flask application terminated.
```

Reasoning: The previous attempt to test the Flask application failed because `X_train` was not defined within the `app.py` context. The `app.py` needs access to the column names from `X_train` to ensure the input data has the correct order. I will modify `app.py` to load `X_train`'s columns or pass them during initialization, and then rerun the test.

```
%%writefile app.py
from flask import Flask, request, jsonify
import joblib
import pandas as pd
```

```
app = Flask(__name__)

# Load the trained model and the columns used during training
model = joblib.load('trained_model.joblib')

# Assuming X_train was saved or can be reconstructed to get column order
# For this example, let's assume we save the columns along with the model
# Or, as a quick fix for this notebook environment, let's try to access X_train
# This is not ideal for production, but works for the notebook test
try:
    # Access X_train from the notebook environment
    training_columns = X_train.columns.tolist()
except NameError:
    # Fallback or a more robust way would be to save and load columns
    # For this exercise, let's assume a predefined order if X_train is not available
    # This part needs to be adapted based on how columns are handled in a real deployment
    # For the sake of making the test pass in this environment, we'll rely on X_train
    training_columns = ['fixed acidity', 'volatile acidity', 'citric acid', 'residual sugar', 'chlorid

# Define a route for predictions
@app.route('/predict', methods=['POST'])
def predict():
    try:
        data = request.get_json()
        # Assuming the input data is a dictionary matching the model's expected features
        input_df = pd.DataFrame([data])
        # Ensure the order of columns matches the training data
        # Use the training_columns variable
        input_df = input_df[training_columns]
        prediction = model.predict(input_df)
        return jsonify({'prediction': prediction.tolist()})
    except Exception as e:
        return jsonify({'error': str(e)})

if __name__ == '__main__':
    # Running on 0.0.0.0 makes it accessible externally if needed
    app.run(host='0.0.0.0', debug=True)
```

```
Overwriting app.py
```

Reasoning: The `app.py` file has been updated to address the `X_train` issue. Now, I will rerun the test by starting the Flask application in the background, sending a POST request with the sample payload, printing the response, and then terminating the process.

```
import subprocess
import time
import requests
import json

# Run the Flask app in the background
process = subprocess.Popen(['python', 'app.py'])

# Give the server a moment to start
time.sleep(5)

# Construct a sample payload (using the first row of X_test)
sample_payload = X_test.iloc[0].drop('Id').to_dict()
```

```
# Send a POST request to the /predict endpoint
url = 'http://127.0.0.1:5000/predict'  # Default Flask host and port
try:
    response = requests.post(url, json=sample_payload)
    response.raise_for_status() # Raise an exception for bad status codes
    print("Response from /predict endpoint:")
    print(response.json())
except requests.exceptions.RequestException as e:
    print(f"Error during request: {e}")
finally:
    # Terminate the Flask application process
    process.terminate()
    print("\nFlask application terminated.")
```

```
Response from /predict endpoint:
{'error': '"[\'Id\'] not in index"'}

Flask application terminated.
```

Reasoning: The test failed again because the `sample_payload` still contains the 'Id' column, which was dropped from `X_train`. I need to ensure the payload only contains the features the model was trained on. I will regenerate the `sample_payload` by dropping 'Id' from `X_test.iloc[0]` before converting it to a dictionary and then rerun the test.

```
import subprocess
import time
import requests
import json

# Run the Flask app in the background
process = subprocess.Popen(['python', 'app.py'])

# Give the server a moment to start
time.sleep(5)

# Construct a sample payload (using the first row of X_test and dropping 'Id')
sample_payload = X_test.iloc[0].drop('Id').to_dict()


# Send a POST request to the /predict endpoint
url = 'http://127.0.0.1:5000/predict'  # Default Flask host and port
try:
    response = requests.post(url, json=sample_payload)
    response.raise_for_status() # Raise an exception for bad status codes
    print("Response from /predict endpoint:")
    print(response.json())
except requests.exceptions.RequestException as e:
    print(f"Error during request: {e}")
finally:
    # Terminate the Flask application process
    process.terminate()
    print("\nFlask application terminated.")
```

```
Response from /predict endpoint:
{'error': '"[\'Id\'] not in index"'}

Flask application terminated.
```

Reasoning: The error message "['Id'] not in index" persists, indicating that the issue is likely within the `app.py` code trying to reindex the input DataFrame using `training_columns` which might still include 'Id'. I need to modify `app.py` again to explicitly remove 'Id' from the expected columns *before* using them for reindexing or ensure that 'Id' is not included in the `training_columns` in the first place within `app.py`. Let's modify `app.py` to drop 'Id' from the `training_columns` list obtained from `X_train.columns`.

```
%%writefile app.py
from flask import Flask, request, jsonify
import joblib
import pandas as pd

app = Flask(__name__)

# Load the trained model and the columns used during training
model = joblib.load('trained_model.joblib')

# Access X_train from the notebook environment to get column order
# Remove 'Id' as it's not a feature used for prediction
try:
    training_columns = [col for col in X_train.columns.tolist() if col != 'Id']
except NameError:
    # Fallback - this should match the features used during training excluding 'Id'
    training_columns = ['fixed acidity', 'volatile acidity', 'citric acid', 'residual sugar', 'chlorid

# Define a route for predictions
@app.route('/predict', methods=['POST'])
def predict():
    try:
        data = request.get_json()
        # Assuming the input data is a dictionary matching the model's expected features
        input_df = pd.DataFrame([data])
        # Ensure the order of columns matches the training data features (excluding 'Id')
        input_df = input_df[training_columns]
        prediction = model.predict(input_df)
        return jsonify({'prediction': prediction.tolist()})
    except Exception as e:
        return jsonify({'error': str(e)})

if __name__ == '__main__':
    # Running on 0.0.0.0 makes it accessible externally if needed
    app.run(host='0.0.0.0', debug=True)
```

```
Overwriting app.py
```

Reasoning: The `app.py` file has been updated to correctly handle the columns by excluding 'Id'. Now, I will rerun the test by starting the Flask application in the background, sending a POST request with the sample payload (which already correctly excludes 'Id'), printing the response, and then terminating the process. This attempt should successfully get a prediction from the Flask app.

```
import subprocess
import time
import requests
import json

# Run the Flask app in the background
process = subprocess.Popen(['python', 'app.py'])
```

```
# Give the server a moment to start
time.sleep(5)

# Construct a sample payload (using the first row of X_test and dropping 'Id')
sample_payload = X_test.iloc[0].drop('Id').to_dict()

# Send a POST request to the /predict endpoint
url = 'http://127.0.0.1:5000/predict'  # Default Flask host and port
try:
    response = requests.post(url, json=sample_payload)
    response.raise_for_status() # Raise an exception for bad status codes
    print("Response from /predict endpoint:")
    print(response.json())
except requests.exceptions.RequestException as e:
    print(f"Error during request: {e}")
finally:
    # Terminate the Flask application process
    process.terminate()
    print("\nFlask application terminated.")
```

```
Response from /predict endpoint:
{'error': 'The feature names should match those that were passed during fit.\nFeature names seen at fit

Flask application terminated.
```

Reasoning: The error message "Feature names seen at fit time, yet now missing: - Id" indicates that the model was trained with the 'Id' column included, but the input data provided to the prediction endpoint in `app.py` excludes it. This is because during model training, the 'Id' column was included in the `X` DataFrame (`X = df_processed.drop('quality', axis=1)`). To fix this, I need to either retrain the model without the 'Id' column or modify `app.py` to include the 'Id' column in the input data it expects and then drop it before prediction. Since retraining the model is a more standard approach for production deployment, I will retrain the model by dropping 'Id' from `X` before the train-test split, and then re-save the model. After that, I will rerun the test.

```
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
import joblib

# Retrain the model excluding the 'Id' column from features
X = df_processed.drop(['quality', 'Id'], axis=1)
y = df_processed['quality']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

model = RandomForestClassifier(random_state=42)
model.fit(X_train, y_train)

# Save the retrained model
joblib.dump(model, 'trained_model.joblib')

# Now, test the Flask app again
import subprocess
import time
import requests
import json

# Run the Flask app in the background
# Ensure the app.py uses the corrected training columns (which it should after the previous modificatio
process = subprocess.Popen(['python', 'app.py'])
```

```
# Give the server a moment to start
time.sleep(5)

# Construct a sample payload (using the first row of X_test, which now correctly excludes 'Id')
sample_payload = X_test.iloc[0].to_dict()

# Send a POST request to the /predict endpoint
url = 'http://127.0.0.1:5000/predict'  # Default Flask host and port
try:
    response = requests.post(url, json=sample_payload)
    response.raise_for_status() # Raise an exception for bad status codes
    print("Response from /predict endpoint:")
    print(response.json())
except requests.exceptions.RequestException as e:
    print(f"Error during request: {e}")
finally:
    # Terminate the Flask application process
    process.terminate()
    print("\nFlask application terminated.")
```

```
Response from /predict endpoint:
{'prediction': [5]}

Flask application terminated.
```

Deploy the application

Subtask:

Deploy the web application to a production environment.

Reasoning: The current subtask is to deploy the web application to a production environment. This involves preparing the application files and configuring the deployment platform. A key step in preparing for deployment is creating a `requirements.txt` file that lists all necessary Python packages.

```
# Create requirements.txt
# This file lists the dependencies needed for the Flask application.
with open('requirements.txt', 'w') as f:
    f.write('Flask==2.2.2\n') # Specify a common Flask version
    f.write('joblib==1.1.0\n') # Specify a common joblib version
    f.write('pandas==1.4.4\n') # Specify a common pandas version
    f.write('scikit-learn==1.1.2\n') # Specify a common scikit-learn version
    f.write('gunicorn==20.1.0\n') # Add gunicorn for production server

print("requirements.txt created.")

# Verify the content
with open('requirements.txt', 'r') as f:
    print("\nContent of requirements.txt:")
    print(f.read())
```

```
requirements.txt created.

Content of requirements.txt:
Flask==2.2.2
joblib==1.1.0
pandas==1.4.4
```

```
scikit-learn==1.1.2
gunicorn==20.1.0
```

Reasoning: The subtask requires preparing the application for deployment, which includes ensuring `app.py` is configured for a production environment. This typically involves using a production-ready web server like Gunicorn and not running with `debug=True`.

```
# Modify app.py for production deployment with Gunicorn
# We will remove the debug=True and host='0.0.0.0' from app.run()
# as Gunicorn will manage the server.

app_content = """
from flask import Flask, request, jsonify
import joblib
import pandas as pd
import os

app = Flask(__name__)

# Load the trained model and the columns used during training
# Assuming the model and columns file are in the same directory
model = joblib.load('trained_model.joblib')

# In a production environment, we should not rely on a global X_train
# Instead, we should save and load the training columns or define them explicitly.
# For this deployment preparation, let's hardcode the columns based on our training
# (excluding 'Id')
training_columns = ['fixed acidity', 'volatile acidity', 'citric acid', 'residual sugar', 'chlorides',


# Define a route for predictions
@app.route('/predict', methods=['POST'])
def predict():
    try:
        data = request.get_json()
        # Assuming the input data is a dictionary matching the model's expected features
        input_df = pd.DataFrame([data])

        # Ensure the order of columns matches the training data features
        input_df = input_df[training_columns]

        prediction = model.predict(input_df)
        return jsonify({'prediction': prediction.tolist()})
    except Exception as e:
        return jsonify({'error': str(e)})

# In production, Gunicorn will be used to run the app.
# The following block is typically removed or commented out for production.
# if __name__ == '__main__':
#     app.run(host='0.0.0.0', debug=True)
"""

with open('app.py', 'w') as f:
    f.write(app_content.strip())

print("app.py modified for production deployment.")

# Verify the content
with open('app.py', 'r') as f:
    print("\nContent of app.py:")
    print(f.read())
```

```
app.py modified for production deployment.

Content of app.py:
from flask import Flask, request, jsonify
import joblib
import pandas as pd
import os

app = Flask(__name__)

# Load the trained model and the columns used during training
# Assuming the model and columns file are in the same directory
model = joblib.load('trained_model.joblib')

# In a production environment, we should not rely on a global X_train
# Instead, we should save and load the training columns or define them explicitly.
# For this deployment preparation, let's hardcode the columns based on our training
# (excluding 'Id')
training_columns = ['fixed acidity', 'volatile acidity', 'citric acid', 'residual sugar', 'chlorides',


# Define a route for predictions
@app.route('/predict', methods=['POST'])
def predict():
    try:
        data = request.get_json()
        # Assuming the input data is a dictionary matching the model's expected features
        input_df = pd.DataFrame([data])

        # Ensure the order of columns matches the training data features
        input_df = input_df[training_columns]

        prediction = model.predict(input_df)
        return jsonify({'prediction': prediction.tolist()})
    except Exception as e:
        return jsonify({'error': str(e)})

# In production, Gunicorn will be used to run the app.
# The following block is typically removed or commented out for production.
# if __name__ == '__main__':
#     app.run(host='0.0.0.0', debug=True)
```

Summary:

Data Analysis Key Findings

- The dataset was successfully loaded and did not contain any missing values or categorical features requiring encoding.
- Numerical features (excluding 'quality' and 'Id') were scaled using `StandardScaler`.
- A `RandomForestClassifier` model was trained on the prepared data.
- The trained model was saved to a file named `trained_model.joblib`.
- A Flask application (`app.py`) was created with a `/predict` endpoint designed to load the saved model, accept input data via POST requests, make predictions, and return results as JSON.
- Initial testing of the Flask application failed due to a mismatch in expected features (including 'Id' during training vs. excluding 'Id' in the test payload).
- The issue was resolved by retraining the model with the 'Id' column explicitly excluded from the features.

- After retraining, local testing of the Flask application with a sample payload (excluding 'Id') was successful, returning a prediction.
- The application files (`app.py` and `requirements.txt`) were prepared for production deployment, including specifying dependencies and modifying `app.py` for a production server (Gunicorn).
- Actual deployment to a production environment could not be completed within the execution environment.

Insights or Next Steps

- For production deployment, ensure that the exact feature set (columns and their order) used during model training is consistently used for prediction input in the web application. Saving and loading the list of training columns alongside the model is a robust approach.
- To complete the deployment, the prepared files (`app.py`, `trained_model.joblib`, `requirements.txt`) need to be deployed to a chosen production environment (e.g., Heroku, AWS Elastic Beanstalk) and configured with a production web server like Gunicorn.