For Analysing missing data:

```
import pandas as pd

# Load the dataset from your local path
file_path = r"D:\ACS related\ACS - WIL\STAGE- I (Week1_3)\Dataset_ATS_v2.csv"
df = pd.read_csv(file_path)

# Display basic info about the dataset to identify missing values
print(df.info())
print(df.isnull().sum())
```

- Load the dataset from your specified path.
- Show the structure of the dataset.
- Display the count of missing values in each column.

```
# Checking for unique values in categorical columns to identify inconsistencies
categorical_columns = ['gender', 'Dependents', 'PhoneService', 'MultipleLines', 'InternetService',
'Contract', 'Churn']
unique_values = {col: df[col].unique() for col in categorical_columns}

# Display the unique values for each categorical column
unique_values
```

Analysis of Categorical Values:

- **`gender`**: Only contains `'Female'` and `'Male'` → **No inconsistencies.**
- **`Dependents`**: Contains `'Yes'` and `'No'` → **No inconsistencies.**
- **`PhoneService`**: Contains `'Yes'` and `'No'` → **No inconsistencies.**
- **`MultipleLines`**: Contains `'Yes'` and `'No'` → **No inconsistencies.**
- **`InternetService`**: Contains `'DSL'` and `'Fiber optic'` → **Possible inconsistency:**
  - `'Fiber optic'` could be standardized (e.g., change to `'Fiber Optic'` for consistency).
- **`Contract`**: Contains `'Month-to-month'`, `'One year'`, `'Two year'` → **No inconsistencies.**
- **`Churn`**: Contains `'Yes'` and `'No'` → **No inconsistencies.**

Next Step:

I have standardized `'Fiber optic'` to `'Fiber Optic'` for consistency.

```
# Standardizing 'Fiber optic' to 'Fiber Optic'
df['InternetService'] = df['InternetService'].replace({'Fiber optic': 'Fiber Optic'})

# Verify the changes
df['InternetService'].unique()
```

Result

```
array(['DSL', 'Fiber Optic'], dtype=object)
```

The **`InternetService`** column has been standardized by changing `'Fiber optic'` to `'Fiber Optic'`. Now, the unique values in this column are **'DSL'** and **'Fiber Optic'**, ensuring consistency.

 # Applying One-Hot Encoding to categorical columns

```
df_encoded = pd.get_dummies(df, columns=['gender', 'Dependents', 'PhoneService',
                            'MultipleLines', 'InternetService', 'Contract', 'Churn'], drop_first=True)

# Display the encoded dataset
tools.display_dataframe_to_user(name="One-Hot Encoded Dataset", dataframe=df_encoded)
```

The dataset has been successfully transformed using **One-Hot Encoding**. Categorical variables are now represented as separate binary columns.

RE-CHECK Below codes:

```
from sklearn.model_selection import train_test_split

import pandas as pd
from sklearn.model_selection import train_test_split

# Ensure the dataset is loaded (if not already)
file_path = r"D:\ACS related\ACS - WIL\STAGE- I (Week1_3)\Dataset_ATS_v2.csv"
df = pd.read_csv(file_path)

# Apply One-Hot Encoding to categorical columns
df_encoded = pd.get_dummies(df, columns=['gender', 'Dependents', 'PhoneService',
                            'MultipleLines', 'InternetService', 'Contract', 'Churn'], drop_first=True)

# Splitting features and target variable
X = df_encoded.drop(columns=['Churn_1'])  # Features
y = df_encoded['Churn_1']  # Target variable

# Splitting the dataset into 80% training and 20% testing
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y)

# Display the training and testing set sizes
train_test_sizes = {
    "X_train Shape": X_train.shape,
    "X_test Shape": X_test.shape,
    "y_train Shape": y_train.shape,
    "y_test Shape": y_test.shape
```

```
}

# Convert train_test_sizes dictionary to a DataFrame for display
train_test_sizes_df = pd.DataFrame(list(train_test_sizes.items()), columns=["Dataset", "Shape"])

# Display the training and testing datasets properly
print("Training Dataset Sample:")
print(X_train.join(y_train).head())

print("\nTesting Dataset Sample:")
print(X_test.join(y_test).head())

print("\nDataset Split Information:")
print(train_test_sizes_df)train_test_sizes
```

 # Appropriate scaling techniques are applied to normalise the data, enhancing model performance.

Applying SMOTE for the imbalanced data:

```
>>>
>>> from imblearn.over_sampling import SMOTE
>>> import pandas as pd
>>>
>>> # Dummy imbalanced dataset
>>> X = pd.DataFrame({
...     'feature1': [1, 2, 3, 4, 5, 6],
...         'feature2': [10, 20, 30, 40, 50, 60]
...         })
>>> y = pd.Series([0, 0, 0, 0, 1, 1])  # Imbalanced
>>>
>>> # Apply SMOTE with lower k_neighbors
>>> sm = SMOTE(random_state=42, k_neighbors=1)
>>> X_res, y_res = sm.fit_resample(X, y)
>>>
>>> # Print class balance after SMOTE
>>> print(pd.Series(y_res).value_counts())
0    4
1    4
Name: count, dtype: int64
>>> from imblearn.over_sampling import SMOTE
>>> import pandas as pd
>>>
>>> # Dummy imbalanced dataset
>>> X = pd.DataFrame({
...     'feature1': [1, 2, 3, 4, 5, 6],
...         'feature2': [10, 20, 30, 40, 50, 60]
...         })
>>> y = pd.Series([0, 0, 0, 0, 1, 1])  # Imbalanced
>>>
>>> # Fix: Set k_neighbors=1 because we only have 2 minority samples
>>> sm = SMOTE(random_state=42, k_neighbors=1)
>>> X_res, y_res = sm.fit_resample(X, y)
>>>
>>> # Now this will work because y_res is defined
>>> print
<built-in function print>
>>>
>>>
```

- `SMOTE` from `imblearn` to handle class imbalance.
- `pandas` for data manipulation.

- `X` is your **feature matrix** with two features (`feature1` and `feature2`).
- `y` is your **target label** (0 = majority class, 1 = minority class).
- You have **class imbalance**: 4 samples of class `0` and 2 samples of class `1`.

- `SMOTE` creates synthetic samples of the **minority class (1)** to balance the dataset.
- `k_neighbors=1` is used because you only have **2 samples in the minority class**.

- Normally `k=5` by default, but that would fail with only 2 minority examples.

After resampling:

- Now both class `0` and class `1` have **equal samples (4 each)**.

Normalisation:

```
# Re-running with the newly uploaded file
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, classification_report

# Load data
df = pd.read_csv(r"D:\ACS related\ACS - WIL\STAGE -II (Week4_8)\Week 7 Data Preparation and
Modeling\final\Resampled_Training_Data.csv")

# Separate features and target
X = df.drop("Churn_1", axis=1)
y = df["Churn_1"]

# Define numerical columns
numerical_cols = ['tenure', 'MonthlyCharges']

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

# Normalize
scaler = MinMaxScaler()
X_train_norm = X_train.copy()
X_test_norm = X_test.copy()
X_train_norm[numerical_cols] = scaler.fit_transform(X_train[numerical_cols])
X_test_norm[numerical_cols] = scaler.transform(X_test[numerical_cols])

# Models
models = {
   "Logistic Regression": LogisticRegression(max_iter=1000),
   "Random Forest": RandomForestClassifier(),
   "SVM": SVC()
}

# Evaluate
results = []
for model_name, model in models.items():
```

```
    model.fit(X_train_norm, y_train)
    y_pred = model.predict(X_test_norm)
    acc = accuracy_score(y_test, y_pred)
    report = classification_report(y_test, y_pred, output_dict=True)
    results.append({
        "Model": model_name,
        "Accuracy": acc,
        "Report": report
    })

# Create DataFrame and print
results_df = pd.DataFrame(results)
print(results_df[["Model", "Accuracy"]])
```