Applied Machine Learning for Biological Data Hands-on sessions

⚙ Prerequisites

- NumPy and Pandas fundamentals for handling biological datasets

Who is the hands-on sessions for?

About the hands-on sessions

Overall schedule

PCA and clustering in cancer genomics	100 minutes
Logistic regression in cancer genomics	60 minutes
ML workflow with biological data	100 minutes
Deep-learning-based variant calling via DeepVariant	70 minutes
Accelerated Genomics workflows with Parabricks	100 minutes

Unsupervised Learning

Principal component analysis (PCA) and K-means clustering

⚙ Prerequisites

- BioNT Applied Machine Learning for Biological Data
  - Module 1: Python NumPy and Pandas
  - Module 2: Unsupervised Learning: Clustering (K-Means Clustering, Hierarchical clustering, Clustering evaluation metrics)

Participants should gain skills introduced in above mentioned Lessons or equivalent skills.

✍ Time

1 hours and 30 minutes

Objectives

Objectives

- Demonstrate the use of unsupervised learning for drug sensitivity analysis.
- Example workflow of PCA and K-means clustering with test dataset (drug sensitivity patterns across patients) for patient stratification

Dataset

- Imputed Drug Sensitivities:
  - This data was imputed for TCGA-BRCA patients based on a model trained on cancer cell line gene expression and corresponding in vitro drug response measurements
- Source: Cancer drug sensitivity prediction from routine histology images
- Link to download test dataset*

Jupyter notebook

- We have made a jupyter notebook with the exercises, learners are expected to download this to their computer and open it in jupyter.
- Link to download Jupyter notebook

Workflow

1. Exploratory Data Analysis
   - Calculate basic statistics (mean, median, variance)
   - Visualize data distributions
   - Create correlation heatmap
2. PCA Implementation
   - Determine appropriate number of components
   - Apply PCA transformation
   - Calculate explained variance ratios
   - Generate scree plot
3. Visualization of PCA Results
   - Create biplot of first two principal components
   - Plot samples in PC space
   - Generate loadings plot
4. Interpretation and Analysis
   - Analyze principal component loadings
   - Identify drug contributions to each PC
5. Cluster patients in PC space
   - Correlate PC scores with metadata

Classification

Logistic regression

⚙ Prerequisites

- BioNT Applied Machine Learning for Biological Data
  - Module 1: Python Numpy and Pandas
  - Module 2: Classification: Logistic regression; Tree-based methods; Matrices for classification evaluation

Participants should gain skills introduced in above mentioned Lessons or equivalent skills.

✍ Time

1 hours

❗ Objectives

Objectives

- Demonstrate the use of classification for cancer dataset
- Example Logistic regression analysis with Glioma test dataset for Glioma sub-type classification

❗ Note

ML use-case

- Gliomas - most common primary tumors of the brain
- Glioma categories
  - Low grade gliomas (LGG) - Slower growing gliomas
  - Glioblastoma Multiforme (GBM) - Most aggressive gliomas type
- Glioma classification (grading) depend on the histological/imaging criteria, but clinical and molecular/mutation factors are also very crucial for accurately diagnose glioma patients.
- Logistic regression based analysis tries to use most frequently mutated 20 genes and 3 clinical features to classify/ grade gliomas

Dataset

- Download dataset: TCGA_InfoWithGrade_scaled.csv
- Features:
  - Most frequently mutated 20 genes and
  - 3 clinical features: gender, age at diagnosis, race
- Target variable (i.e, dependant variable or response variables): Glioma grade class information
  - 0 = “LGG”
  - 1 = “GBM”

Source

- UCI Machine Learning Repository - Glioma Grading Clinical and Mutation Features

Complete Machine Learning Workflow

Machine Learning Workflow

⚙ Prerequisites

- BioNT Applied Machine Learning for Biological Data
  - Module 1: Python Numpy and Pandas
  - Module 2: Classification: Logistic regression; Tree-based methods; Matrices for classification evaluation

Participants should gain skills introduced in above mentioned Lessons or equivalent skills.

✍ Time

2 hours

Objectives

- Demonstrate the use of complete classification workflow for cancer dataset (expand on previous hands-on session)
- Example workflow of Logistic regression with Glioma test dataset for Glioma sub-type classification

ML use-case

- ML use-case as described in Classification hands-on session
- Example Logistic regression workflow tries to use most frequently mutated 20 genes and 3 clinical features to classify/ grade gliomas
- Demonstrate following key techniques
  - Data exploration and handing missing data
  - Scaling
  - Cross-validation
  - Hyper-parameter tuning with GridSearch

Dataset

- Download dataset: TCGA_InfoWithGrade_scaled.csv
- Dataset as described in Classification hands-on session
  - Features:
    - Most frequently mutated 20 genes and
    - 3 clinical features: gender, age at diagnosis, race

    - Target variable (i.e, dependant variable or response variables): Glioma grade class information
        - 0 = “LGG”
        - 1 = “GBM”
- Several Additional columns and rows with null values are spiked into the original dataset for demonstration purpose

Source

- UCI Machine Learning Repository - Glioma Grading Clinical and Mutation Features

PCA and Clustering

PCA and Clustering Analysis of Drug Sensitivity Data

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
```

Data Preparation

- Clean the dataset by handling missing values
- Scale/normalize the data
- Check for outliers
- Separate metadata from drug sensitivity values

```
data = pd.read_csv('test_data/pca_clustering/BRCA_Drug_sensitivity_test_data.csv')
data.head()
```

```
print(f"Dataset shape: {data.shape}")
print(f"Patient IDs: {data['Patient ID'].nunique()} unique values")
print(f"Features: {data.shape[1]-1} drug sensitivity measurements")
```

```
Dataset shape: (25, 51)
Patient IDs: 25 unique values
Features: 50 drug sensitivity measurements
```

Notes: High-dimensional data

- High-dimensional data refers to datasets where the number of features or attributes (dimensions, denoted as p) is significantly large
- In such datasets, each observation can be thought of as residing in a high-dimensional space
- The definition of “high-dimensional” data can vary depending on the context, the field of study, and the specific analysis being performed.

Dataset with 51 columns (drug compounds) and 25 rows (patients):

- Contains more features (50 drug sensitivity scores) than samples (25 patients)
- *Difficulty in Visualization:* Identifying patterns visually becomes impossible
- *Sparsity:* With 50 features but only 25 samples, data points are scattered across a vast 50-dimensional space
- *Distance metrics become less meaningful:* In high dimensions, the difference between the nearest and farthest neighbors becomes less significant - nearly all points appear similar-distances from each other
- *Overfitting risk:* model has many potential combinations of features to consider, but limited examples to learn from, it has a large capacity to fit even noise in the limited training examples leading to overfitting and poor performance

Clean the dataset by handling missing values

```
# Check for missing values
data.isnull().sum().sum()
```

```
# Check data types
print("\nData types:")
print("Datatypes of first 10 columns:", data.dtypes[:10])
print("Different datatypes in the dataframe:", data.dtypes.unique())
```

```
Data types:
Datatypes of first 10 columns: Patient ID              object
bendamustine           float64
ML320                  float64
BRD-K14844214          float64
leptomycin B           float64
Compound 23 citrate    float64
BRD4132                float64
dabrafenib             float64
necrosulfonamide       float64
PF-543                 float64
dtype: object
Different datatypes in the dataframe: [dtype('O') dtype('float64')]
```

```
# Basic statistics for drug sensitivity values
data.iloc[2:,:].describe().T.sort_values('mean', ascending=False).head(10)
```

```
# Step 2: Exploratory Data Analysis
def perform_eda(data):
    print("\n=== Exploratory Data Analysis ===")
    # Separate metadata from drug sensitivity values
    drug_data = data.drop('Patient ID', axis=1)

    # Distribution of drug sensitivity values
    plt.figure(figsize=(15, 6))
    plt.subplot(1, 2, 1)
    sns.histplot(drug_data.values.flatten(), kde=True)
    plt.title('Distribution of Drug Sensitivity Values')
    plt.xlabel('Sensitivity Value')

    # Boxplot of drug sensitivity values (first 10 drugs)
    plt.subplot(1, 2, 2)
    sns.boxplot(data=drug_data.iloc[:, :10])
    plt.title('Boxplot of First 10 Drugs')
    plt.xticks(rotation=90)
    plt.tight_layout()

    return drug_data
```

```
drug_data = perform_eda(data)
```

```
=== Exploratory Data Analysis ===
```

```
drug_data.head()
```

```
plt.figure(figsize=(20, 6))
sns.boxplot(data=drug_data)
plt.xticks(rotation=90)
```

```
variances = drug_data.var().sort_values(ascending=False)
plt.figure(figsize=(20, 6))
sns.barplot(x=variances.index, y=variances.values)
plt.xticks(rotation=90)
plt.title('Variance Across Columns')
plt.ylabel('Variance')
plt.show()
```

Normalize the data

PCA is sensitive to the variances of the original features, therefore data normalization before applying PCA is crucial

- PCA works by identifying the directions (principal components) that capture the maximum variance in the data.
- Variables with larger variances can dominate the principal components
  - This means the principal components might primarily reflect the variability of the features with the largest ranges
  - principal components do not capture the underlying relationships in the data
- Drugs with inherently higher variability in their sensitivity scores would disproportionately influence the principal components, potentially masking the contributions and relationships involving drugs with lower score variability.

PCA looks for directions of maximum variance, standardization ensures that each feature contributes more equally to the determination of the principal components

```
# Create the scaler and standardize the data
scaler = StandardScaler()
drug_data = scaler.fit_transform(drug_data)
```

```
print("Type of drug_data:", type(drug_data))
print("Shape of drug_data:", drug_data.shape)
print("First 5 rows and columns of drug_data:\n", drug_data[:5, :5])
```

```
Type of drug_data: <class 'numpy.ndarray'>
Shape of drug_data: (25, 50)
First 5 rows and columns of drug_data:
 [[ 1.11745915 -0.14919196 -0.01756464  1.22165355  0.70067562]
 [-0.4261314  -0.02638244  2.24640919  0.06536914 -0.66625469]
 [ 0.62984536  0.7962644  -0.40535119 -0.00447747  0.3918278 ]
 [-0.3123208  -1.78569382 -1.18197636 -1.3745291  -0.26480228]
 [ 0.62347222 -0.0529662  -0.5338052   0.83604629 -0.09453399]]
```

```
plt.figure(figsize=(20, 6))
sns.boxplot(data=drug_data)
plt.xticks(rotation=90)
```

```
drug_data.var(axis=0)
```

```
variances = drug_data.var(axis=0)#.sort_values(ascending=False)
plt.figure(figsize=(20, 6))
sns.barplot(x=range(0,len(variances)), y=variances)
plt.xticks(ticks=range(0, len(variances)), rotation=90)
plt.title('Variance Across Columns')
plt.ylabel('Variance')
plt.show()
```

PCA Implementation

- Apply PCA transformation
- Calculate explained variance

Apply PCA transformation

```
# Create the PCA instance and fit and transform the data with pca
pca = PCA()
pc = pca.fit_transform(drug_data)
```

```
print("Type of pc:", type(pc))
print("Shape of pc:", pc.shape)
print("First 5 rows and columns of pc:\n", pc[:5, :5])
```

```
Type of pc: <class 'numpy.ndarray'>
Shape of pc: (25, 25)
First 5 rows and columns of pc:
 [[ 3.64790724  3.78552267  1.33659718 -0.20864174 -2.17346721]
 [ 0.32125071 -5.9898321   4.18994535 -0.20697045  0.17194817]
 [ 3.83231009  1.28508967 -0.47749898 -2.19393142  3.98217509]
 [-5.63359232  3.69098317 -3.47828106 -0.8034397   0.44613498]
 [ 0.17402269  1.51797501  0.16606953  2.91234111  0.42116796]]
```

Note:

- Original `drug_data` had 25 rows (observations), so `pc` also had 25 rows representing patients
- We didn't specify the number of components (`n_components`) when creating the PCA object (`pca = PCA()`), scikit-learn defaults to calculating `min(n_samples, n_features)` components. Original dataset had 50 features, PCA still only computes 25 components because the maximum number of meaningful principal components is limited by the number of samples (you can't find more independent directions of variance than you have data points).

The `fit_transform` method did two things:

1. `fit`: It analyzed drug_data to find the 25 principal component directions (axes) based on the variance and correlations between your original features.
2. `transform`: It then projected your original 25 samples from their original feature space (with >=25 dimensions) onto this new coordinate system defined by the 25 principal components

```
pc_df = pd.DataFrame(pc, columns=[f'PC_{i}' for i in range(1, pc.shape[1]+1)])
pc_df.head()
```

PCA context

- PCA aims to summarize a large set of correlated variables with a smaller number of representative variables
- The goal is to find a low-dimensional representation of the data that retains as much of the original variation as possible
- **The first principal component** is defined as the linear combination of the original features that has the largest sample variance
- **Subsequent principal components** are then found such that they have the maximal variance out of all linear combinations that are uncorrelated with the preceding principal components
    - For example, the second principal component must be uncorrelated with the first, the third with the first two, and so on
- By transforming the original correlated variables into a set of uncorrelated principal components, PCA effectively removes redundancy in the data
    - Correlated variables inherently contain overlapping information. Decorrelation ensures that each principal component captures a distinct aspect of the data's variability

```
drug_data_normalised = pd.DataFrame(drug_data, columns=data.columns[1:])
```

```
# Correlation heatmap
# plt.figure(figsize=(12, 10))

fig, axes = plt.subplots(1, 2, figsize=(20, 10))

mask = np.triu(np.ones_like(drug_data_normalised.corr(), dtype=bool))
sns.heatmap(drug_data_normalised.corr(), mask=mask, annot=False, cmap='coolwarm',
            linewidths=0.5, vmin=-1, vmax=1, ax=axes[0])
axes[0].set_title('Correlation Heatmap (All Drugs)')

mask = np.triu(np.ones_like(pc_df.corr(), dtype=bool))
sns.heatmap(pc_df.corr(), mask=mask, annot=False, cmap='coolwarm',
            linewidths=0.5, vmin=-1, vmax=1, ax=axes[1])
axes[1].set_title('Correlation Heatmap (PCA Components)')


plt.tight_layout()

plt.show()
plt.close()
```

```
fig, axes = plt.subplots(2, 1, figsize=(20, 10))

pc_df.plot(kind="box", title="PCA Components Boxplot", ax=axes[0])
drug_data_normalised.plot(kind="box", title="Drug Sensitivity Boxplot", ax=axes[1])
plt.xticks(rotation=90)
plt.tight_layout()
```

Note:

- PCA successfully achieved its goal of decorrelating our dataset
- Decorrelation:
  - The variance originally shared (correlation) between features in the original dataset has been reorganized and captured along these new, independent PC axes
- Why focus on maximizing variance during decorrelation
  - High-variance directions contain more signal, while low-variance directions often represent noise
  - Allows PCA to identify the most “important” directions (data that account for the greatest spread)
- These uncorrelated components that capture maximum variance provide a more efficient and interpretable representation of the data compared to the original correlated features

The first principal component is defined as the linear combination of features that has the largest variance, subject to the constraint that the coefficients in the linear combination is one. Subsequent principal components are found by maximizing variance among linear combinations uncorrelated with previous components

```
# pc_df.head()
```

```
# # Calculate cumulative variance = cumulative proportion of variance explained by the
principal components
# variances = pc_df.var(axis=0)
# cumulative_variance = variances.cumsum() / variances.sum()

# # Plot cumulative variance
# plt.figure(figsize=(10, 6))
# plt.plot(range(1, len(cumulative_variance) + 1), cumulative_variance, marker='o',
linestyle='--')
# plt.title('Cumulative Variance Explained by Principal Components')
# plt.xlabel('Number of Principal Components')
# plt.ylabel('Cumulative Variance Explained')
# plt.grid()
# plt.show()
```

Explained variance calculation

- PCA seeks a low-dimensional representation of a dataset that captures as much as possible of the variation in the original data
- **Variance Explained by a PC:**
  - The variance explained by the `nth` principal component is the variance of its scores
- **Total Variance:**
  - The total variance present in the original data (assuming variables are centered to have mean zero) is the sum of the variances of the original features
- **Relationship between PC Variance and Total Variance:**
  - A key property is that the ***sum of the variances of all principal components equals the total variance of the original data***
  - This means maximizing the variance of the `n` principal components is equivalent to minimizing the reconstruction error when approximating the data with those `n` components
- **Proportion of Variance Explained (PVE) | Explained Variance Ratio:**
  - The PVE of the `mth` principal component is calculated as the variance of the `mth` principal component scores divided by the total variance in the original data
  - The PVEs for all principal components (up to min(n-1, p)) sum to one
- **Cumulative PVE:**
  - The cumulative PVE of the principal components is simply the sum of the PVEs for those components

```
## Access the explained variance directly
explained_variance = pca.explained_variance_
print("Explained variance by component:", explained_variance)
```

```
Explained variance by component: [2.06118885e+01 9.46481592e+00 5.66619038e+00
4.25914905e+00
 2.86483250e+00 1.91158440e+00 1.59439385e+00 1.22821102e+00
 9.13563089e-01 8.09987389e-01 5.90796907e-01 4.07254006e-01
 3.73077915e-01 2.98129789e-01 2.48097781e-01 2.31592814e-01
 1.58434105e-01 1.29071221e-01 8.11724160e-02 7.48451289e-02
 6.29952247e-02 5.23843184e-02 3.73057774e-02 1.35597766e-02
 3.99827721e-32]
```

```
explained_variance_ratio = pca.explained_variance_ratio_
print("Explained variance ratio by component:", explained_variance_ratio)
```

```
Explained variance ratio by component: [3.95748260e-01 1.81724466e-01 1.08790855e-01
8.17756618e-02
 5.50047841e-02 3.67024205e-02 3.06123619e-02 2.35816516e-02
 1.75404113e-02 1.55517579e-02 1.13433006e-02 7.81927692e-03
 7.16309597e-03 5.72409195e-03 4.76347739e-03 4.44658203e-03
 3.04193482e-03 2.47816745e-03 1.55851039e-03 1.43702647e-03
 1.20950831e-03 1.00577891e-03 7.16270926e-04 2.60347711e-04
 7.67669224e-34]
```

```
# Option 3: Calculate the cumulative explained variance
cumulative_explained_variance = np.cumsum(explained_variance_ratio)
print("Cumulative explained variance:", cumulative_explained_variance)
```

```
Cumulative explained variance: [0.39574826 0.57747273 0.68626358 0.76803924 0.82304403
0.85974645
 0.89035881 0.91394046 0.93148087 0.94703263 0.95837593 0.96619521
 0.9733583  0.9790824  0.98384587 0.98829246 0.99133439 0.99381256
 0.99537107 0.99680809 0.9980176  0.99902338 0.99973965 1.
 1.        ]
```

Visualize the explained variance

```
# Visualize the explained variance
plt.figure(figsize=(10, 6))
plt.bar(range(1, len(explained_variance_ratio) + 1), explained_variance_ratio,
alpha=0.7, label='Individual explained variance')
plt.step(range(1, len(cumulative_explained_variance) + 1),
cumulative_explained_variance, where='mid', label='Cumulative explained variance')
plt.ylabel('Explained variance ratio')
plt.xlabel('Principal components')
plt.axhline(y=0.95, color='r', linestyle='--', label='95% explained variance
threshold')
plt.axhline(y=0.90, color='r', linestyle='dotted', label='90% explained variance
threshold')
plt.legend(loc='best')
plt.tight_layout()
plt.show()
```

Determine optimal number of components

- Variance Threshold
- Scree Plot (Elbow Method)

Variance Threshold

- Find number of components that explain at least 95% (or 99%) of variance
- Selects components that collectively explain a predefined percentage (e.g., 95%) of total variance
- Ensures you keep enough information while reducing dimensions
- Provides a clear cutoff criterion that doesn't require subjective judgment (Automated selection)

```
n_components_95 = np.argmax(cumulative_explained_variance >= 0.95) + 1
# np.argmax(): This function returns the index of the first occurrence of the maximum
value in the array.

print(f"Number of components for 95% variance: {n_components_95}")
```

```
Number of components for 95% variance: 11
```

Scree Plot (Elbow Method)

- Helps identify the point where additional components yield diminishing returns (Visual identification)
  - Balances model complexity against information retention
- The "elbow" often marks where principal components transition from capturing signal to capturing noise (Noise reduction)
- Visually reveals the relative importance of components, making the decision process more transparent

```
plt.figure(figsize=(10, 6))
plt.plot(range(1, len(cumulative_explained_variance) + 1),
cumulative_explained_variance, marker='o', linestyle='--')
plt.title('Cumulative Variance Explained by Principal Components')
plt.xlabel('Number of Principal Components')
plt.ylabel('Cumulative Variance Explained')
plt.grid()
plt.show()
```

```
plt.figure(figsize=(10, 6))
plt.plot(range(1, len(explained_variance) + 1), explained_variance, 'o-', linewidth=2)
plt.title('Scree Plot')
plt.xlabel('Principal Component')
plt.ylabel('Eigenvalue (Variance)')
plt.grid(True)
plt.show()
```

- The eigenvalue of a component equals the variance of the data points when projected onto that component
  - Eigenvalue as a number that tells you “how much variance” is associated with a specific eigenvector (which represents a principal component direction)
  - When you project your data onto that eigenvector’s direction, the variance you calculate for those projected points will be exactly the eigenvalue

Automated Selection with PCA

- Instead of manually determining the number of components, you can initialize PCA with a variance threshold:

```
# Automatically select components to explain 90% of variance
optimal_pca = PCA(n_components=0.90)  # Keep enough components to explain 90% of
variance
pc_auto = optimal_pca.fit_transform(drug_data)
print(f"Number of components selected: {optimal_pca.n_components_}")

optimal_pca_df = pd.DataFrame(
    data=pc_auto,
    columns=[f'PC{i+1}' for i in range(pc_auto.shape[1])]
)
optimal_pca_df['Patient ID'] = data["Patient ID"]

print(f"Optimal PCA DataFrame: \n{optimal_pca_df.head()}")
```

```
Number of components selected: 8
Optimal PCA DataFrame:
        PC1       PC2       PC3       PC4       PC5       PC6       PC7  \
0  3.647907  3.785523  1.336597 -0.208642 -2.173467 -0.159574 -0.201224
1  0.321251 -5.989832  4.189945 -0.206970  0.171948  0.960831 -0.520194
2  3.832310  1.285090 -0.477499 -2.193931  3.982175 -0.967499 -0.351455
3 -5.633592  3.690983 -3.478281 -0.803440  0.446135  0.610747 -2.052404
4  0.174023  1.517975  0.166070  2.912341  0.421168 -0.887175 -1.127990

        PC8    Patient ID
0 -0.175144  TCGA-D8-A1JU
1  1.106246  TCGA-AC-A3QQ
2  0.389307  TCGA-C8-A12Q
3  0.316239  TCGA-AR-A1AY
4 -1.452414  TCGA-A8-A0A2
```

Feature Loadings

- Feature Loadings = Contribution of each original feature to each principal component
- Shows which original features most strongly influence each principal component - **Feature influence**
- Helps interpret what each principal component represents - **Component interpretation**
- Identifies which original features are most important for your dataset's structure - **Feature selection**
- Reveals relationships between features in your high-dimensional space - **Dimensionality insights**

High absolute values (positive or negative) indicate strong influence on that component. A loading of 0 means no influence

```
loadings = optimal_pca.components_

# Get feature names (assuming you have them in a list or as column names)
feature_names = data.iloc[:, 1:].columns  # Or your list of feature names

# Create a DataFrame with the loadings
loadings_df = pd.DataFrame(
    loadings,
    columns=feature_names,
    index=[f'PC_{i+1}' for i in range(loadings.shape[0])]
)

# loading_df = pd.DataFrame(
#     loadings.T,
#     columns=[f'PC{i+1}' for i in range(loadings.shape[0])],
#     index=drug_data_normalised.columns
# )
loadings_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 8 entries, PC_1 to PC_8
Data columns (total 50 columns):
 #   Column                          Non-Null Count  Dtype
---  ------                          --------------  -----
 0   bendamustine                    8 non-null      float64
 1   ML320                           8 non-null      float64
 2   BRD-K14844214                   8 non-null      float64
 3   leptomycin B                    8 non-null      float64
 4   Compound 23 citrate             8 non-null      float64
 5   BRD4132                         8 non-null      float64
 6   dabrafenib                      8 non-null      float64
 7   necrosulfonamide                8 non-null      float64
 8   PF-543                          8 non-null      float64
 9   KX2-391                         8 non-null      float64
 10  ELCPK                           8 non-null      float64
 11  carboplatin                     8 non-null      float64
 12  SB-525334                       8 non-null      float64
 13  CIL41                           8 non-null      float64
 14  belinostat                      8 non-null      float64
 15  Compound 7d-cis                 8 non-null      float64
 16  lapatinib                       8 non-null      float64
 17  tacrolimus                      8 non-null      float64
 18  pifithrin-mu                    8 non-null      float64
 19  RG-108                          8 non-null      float64
 20  BRD-K97651142                   8 non-null      float64
 21  NVP-BEZ235                      8 non-null      float64
 22  BRD-K01737880                   8 non-null      float64
 23  pluripotin                      8 non-null      float64
 24  GW-405833                       8 non-null      float64
 25  masitinib                       8 non-null      float64
 26  YM-155                          8 non-null      float64
 27  MLN2238                         8 non-null      float64
 28  birinapant                      8 non-null      float64
 29  RAF265                          8 non-null      float64
 30  BRD-K27188169                   8 non-null      float64
 31  PIK-93                          8 non-null      float64
 32  N9-isopropylolomoucine          8 non-null      float64
 33  myricetin                       8 non-null      float64
 34  epigallocatechin-3-monogallate  8 non-null      float64
 35  ceranib-2                       8 non-null      float64
 36  BRD-K66532283                   8 non-null      float64
 37  elocalcitol                     8 non-null      float64
 38  RO4929097                       8 non-null      float64
 39  BRD-K02251932                   8 non-null      float64
 40  Compound 1541A                  8 non-null      float64
 41  pevonedistat                    8 non-null      float64
 42  ISOX                            8 non-null      float64
 43  tosedostat                      8 non-null      float64
 44  AT13387                         8 non-null      float64
 45  BRD-A02303741                   8 non-null      float64
 46  BRD-K99006945                   8 non-null      float64
 47  GSK525762A                      8 non-null      float64
 48  necrostatin-7                   8 non-null      float64
 49  nakiterpiosin                   8 non-null      float64
dtypes: float64(50)
memory usage: 3.2+ KB
```

```
print("Contribution of each original feature to each principal component")
loadings_df
```

```
Contribution of each original feature to each principal component
```

Visualization - Feature Loadings

```
# Create a heatmap
plt.figure(figsize=(12, 8))
heatmap = sns.heatmap(
    loadings_df,
    cmap='coolwarm',
    center=0,
    annot=False,
    fmt=".2f",
    linewidths=.5
)
plt.title('PCA Feature Loadings')
plt.tight_layout()
plt.show()
```

```
# Select top n contributing features for each component
def get_top_features(loadings_df, n=10):
    top_features = pd.DataFrame()

    for pc in loadings_df.index:
        pc_loadings = loadings_df.loc[pc].abs().sort_values(ascending=False)
        top_features[pc] = pc_loadings.index[:n]

    return top_features

top_features = get_top_features(loadings_df)
print("Top contributing features for each principal component:")
top_features
```

```
Top contributing features for each principal component:
```

K-means Cluster analysis

- K-means seeks to group observations so that those within each cluster are more closely related to one another than objects assigned to different clusters
- “Good” clustering is one for which the within-cluster variation is as small as possible
- K-means aims to minimize the total *within-cluster variation* summed over all K cluster
- *Within-cluster variation:*
  - Describes how dispersed the data points are within their clusters
  - In K-means specifically, the *within-cluster sum-of-squares (Inertia)* is the standard way to quantify Within-cluster variation, but other clustering algorithms might use different mathematical formulations to measure variation

Inertia (within-cluster sum-of-squares)

- Inertia is a mathematical measure that quantifies how tightly grouped the data points are within their assigned clusters
- K-means clustering uses an iterative optimization strategy to minimize inertia

Variation of Increasing with Number of Clusters

- Inertia always decreases or stays the same, never increases
- The rate of decrease typically follows a curve that looks like this:
  - Rapid decrease initially (adding the first few clusters)
  - Gradually diminishing returns as K continues to increase
  - Eventually, minimal improvements with additional clusters

Why This Happens?

- With more clusters, each data point can be assigned to a centroid that's closer to it
- When K=1: Maximum inertia (all points compared to global mean)
- When K=n (number of data points): Zero inertia (each point is its own cluster)
- The first few clusters capture the major structure in the data, while additional clusters only capture finer details (*Diminishing returns*)

The Elbow Method

This behavior forms the basis of the popular "elbow method" for determining the optimal number of clusters:

1. Plot inertia against K (for K=1,2,3…)
2. Look for the "elbow point" where the curve bends sharply
3. This point represents where adding more clusters stops providing significant reduction in inertia

Important Considerations:

1. Inertia will always decrease as K increases, even if you're adding meaningless clusters
2. This is why we look for the elbow point rather than simply minimizing inertia

Silhouette Score

- Evaluating clustering quality that addresses some limitations of using inertia alone
- It measures how well-separated clusters are by considering both:
  - How similar each point is to its own cluster (cohesion)
  - How different it is from other clusters (separation)

Is each data point placed in the right cluster?

- How close a point is to other points in its assigned cluster
- How close that same point is to points in the nearest neighboring cluster

Interpretation of Silhouette Score

- The silhouette score ranges from -1 to 1:
  - Close to +1: The point is well matched to its own cluster and far from neighboring clusters
  - Close to 0: The point lies near the boundary between two clusters
  - Close to -1: The point might be assigned to the wrong cluster
- The overall silhouette score of a cluster is simply the average of all individual silhouette scores.

Finding Optimal K Using Silhouette

Unlike inertia which always decreases as K increases, the silhouette score typically:

- Increases as K approaches the natural number of clusters
- Reaches a maximum at or near the optimal K
- Decreases as K becomes too large
- This makes it particularly useful for determining the optimal number of clusters - you simply choose the K value that maximizes the average silhouette score.

```
# 1. Determine optimal number of clusters using the Elbow method

inertia = []
silhouette_scores = []
k_range = range(2, 11)  # Testing from 2 to 10 clusters

# Exclude the Patient ID column for clustering
pca_data = optimal_pca_df.drop('Patient ID', axis=1)

for k in k_range:
    kmeans = KMeans(n_clusters=k, random_state=42, n_init=10)
    kmeans.fit(pca_data)
    inertia.append(kmeans.inertia_)

    # Calculate silhouette score (only valid for k >= 2)
    labels = kmeans.labels_
    silhouette_scores.append(silhouette_score(pca_data, labels))
    print(f"Silhouette score for {k} clusters: {silhouette_scores[-1]}")
```

```
Silhouette score for 2 clusters: 0.315860568817066
Silhouette score for 3 clusters: 0.23399486593926483
Silhouette score for 4 clusters: 0.1923227376052971
Silhouette score for 5 clusters: 0.21280378841022102
Silhouette score for 6 clusters: 0.2215781206218201
Silhouette score for 7 clusters: 0.18825389524527888
Silhouette score for 8 clusters: 0.16503544411475235
Silhouette score for 9 clusters: 0.13979820154212966
Silhouette score for 10 clusters: 0.13772690406314397
```

```
# Plot the Elbow curve
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(k_range, inertia, 'o-')
plt.xlabel('Number of clusters (k)')
plt.ylabel('Inertia')
plt.title('Elbow Method for Optimal k')
plt.grid(True)

# Plot silhouette scores
plt.subplot(1, 2, 2)
plt.plot(k_range, silhouette_scores, 'o-')
plt.xlabel('Number of clusters (k)')
plt.ylabel('Silhouette Score')
plt.title('Silhouette Scores for Different k')
plt.grid(True)
plt.tight_layout()
plt.show()
```

K-means seeks to group observations so that those within each cluster are more closely related to one another than objects assigned to different clusters “good” clustering is one for which the within-cluster variation is as small as possible K-means aims to minimize the total within-cluster variation summed over all K cluster

Distortion is the sum of the squares of distances between each data point and its assigned cluster center Distortion generally has an inverse relationship with the number of clusters i.e., As the number of clusters increases, distortion tends to decrease

```
# 2. Apply K-means with the optimal k
optimal_k = 3  # optimal k from the plots
kmeans = KMeans(n_clusters=optimal_k, random_state=42, n_init=10)
#clusters = kmeans.fit_predict(pca_data)

clusters = kmeans.fit(pca_data)

print(f"Number of clusters: {optimal_k}")
print(f"Cluster centers:\n{kmeans.cluster_centers_}")
print(f"Cluster labels for each patient sample: {clusters.labels_}, shape:
{clusters.labels_.shape}")
```

```
Number of clusters: 3
Cluster centers:
[[ 2.42255496  1.31481132  0.22658619  0.31864774  0.11053081 -0.05490954
  -0.12686288 -0.10959923]
 [-1.34671408 -3.70301852 -0.57098639 -0.42079775 -0.54593247 -0.09185375
   0.23802588  0.04913763]
 [-8.97044194  2.06631993  0.19937064 -0.61137725  0.72118837  0.48887309
   0.0789207   0.43334165]]
Cluster labels for each patient sample: [0 1 0 2 0 1 0 0 0 0 0 1 0 0 0 2 0 1 0 1 0 2 1
1 0], shape: (25,)
```

```python
# 3. Add cluster labels to the dataframe
optimal_pca_df['Cluster'] = clusters.labels_
print("First 5 rows of PCA DataFrame with cluster labels:")
optimal_pca_df.head()
```

```
First 5 rows of PCA DataFrame with cluster labels:
```

```python
# Basic cluster analysis
print("Cluster distribution:")
print(optimal_pca_df['Cluster'].value_counts())
```

```
Cluster distribution:
Cluster
0    15
1     7
2     3
Name: count, dtype: int64
```

```python
optimal_pca_df.groupby(["Cluster", "Patient
ID"]).size().reset_index(name='Counts').drop(columns='Counts')
```

```python
# Calculate mean of each feature for each cluster
cluster_means = optimal_pca_df.drop(columns="Patient ID").groupby('Cluster').mean()
print("\nCluster centers in PCA space:")
print(cluster_means)
```

```
Cluster centers in PCA space:
               PC1       PC2       PC3       PC4       PC5       PC6       PC7  \
Cluster
0         2.422555  1.314811  0.226586  0.318648  0.110531 -0.054910 -0.126863
1        -1.346714 -3.703019 -0.570986 -0.420798 -0.545932 -0.091854  0.238026
2        -8.970442  2.066320  0.199371 -0.611377  0.721188  0.488873  0.078921

               PC8
Cluster
0       -0.109599
1        0.049138
2        0.433342
```

```
# 4. Visualize clusters using first two principal components
plt.figure(figsize=(10, 8))
for cluster in range(optimal_k):
    cluster_points = optimal_pca_df[optimal_pca_df['Cluster'] == cluster]
    plt.scatter(
        cluster_points['PC1'],
        cluster_points['PC2'],
        label=f'Cluster {cluster}',
        alpha=0.7,
        s=80
    )
    for i, row in cluster_points.iterrows():
        plt.text(row['PC1'], row['PC2'], str(row['Patient ID']), fontsize=8, alpha=0.7)

centers = kmeans.cluster_centers_
plt.scatter(
    centers[:, 0],
    centers[:, 1],
    s=200,
    marker='.',
    c='red',
    label='Centroids'
)


plt.legend(title='Cluster', fontsize=10, title_fontsize=12, loc='lower left')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.title('Clusters Visualized on First Two Principal Components')
plt.grid(True)
plt.tight_layout()
plt.show()
```

```
optimal_pca_df.head()
```

```
features_to_analyze = optimal_pca_df.columns.drop(['Patient ID', 'Cluster'])
features_to_analyze
```

```
cluster_stats = optimal_pca_df.groupby('Cluster')[['PC1', 'PC2', 'PC3']].agg(['mean',
'std', 'min', 'max'])

# --- 2. Display the results ---
print("Descriptive Statistics for Each Cluster:")
cluster_stats

# # Optional: Display mean values separately for clarity
# print("\nMean Feature Values for Each Cluster:")
# print(cluster_stats['mean'])
```

```
Descriptive Statistics for Each Cluster:
```

```
optimal_pca.components_.shape
```

Logistic regression

Dataset

- Features:
  - Most frequently mutated 20 genes and
  - 2 clinical features: gender and age at diagnosis
- Target variable (i.e, dependant variable or response variables): Glioma grade class information
  - 0 = “LGG”
  - 1 = “GBM”

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, precision_score, recall_score, roc_curve,
auc
```

```
gliomas = pd.read_csv('test_data/logistic_reg/TCGA_InfoWithGrade_scaled.csv')
gliomas.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 839 entries, 0 to 838
Data columns (total 23 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   Grade             839 non-null    int64
 1   Gender            839 non-null    int64
 2   Age_at_diagnosis  839 non-null    float64
 3   IDH1              839 non-null    int64
 4   TP53              839 non-null    int64
 5   ATRX              839 non-null    int64
 6   PTEN              839 non-null    int64
 7   EGFR              839 non-null    int64
 8   CIC               839 non-null    int64
 9   MUC16             839 non-null    int64
 10  PIK3CA            839 non-null    int64
 11  NF1               839 non-null    int64
 12  PIK3R1            839 non-null    int64
 13  FUBP1             839 non-null    int64
 14  RB1               839 non-null    int64
 15  NOTCH1            839 non-null    int64
 16  BCOR              839 non-null    int64
 17  CSMD3             839 non-null    int64
 18  SMARCA4           839 non-null    int64
 19  GRIN2A            839 non-null    int64
 20  IDH2              839 non-null    int64
 21  FAT4              839 non-null    int64
 22  PDGFRA            839 non-null    int64
dtypes: float64(1), int64(22)
memory usage: 150.9 KB
```

```
gliomas.head()
```

```
gliomas.groupby('Grade').size()
```

```
# Plot the distribution of data in all columns
fig, axs = plt.subplots(nrows=6, ncols=4, figsize=(15, 20))
axs = axs.flatten()

# Iterate over each column and plot the distribution
for i, column in enumerate(gliomas.columns):
    axs[i].hist(gliomas[column], bins=20, color='skyblue', edgecolor='black')
    axs[i].set_title(column)
    axs[i].set_xlabel('Value')
    axs[i].set_ylabel('Frequency')
plt.tight_layout()
plt.show()
```

```
gliomas.describe().T
```

Split original dataset

- `train_test_split` function from scikit-learn split a dataset into random train and test subsets
- Default behaviour,
  - Shuffles the data before splitting
  - Does not inherently preserve the distribution of the original dataset when splitting to train and test subsets
  - The distribution in train and test subsets the depends on the randomness of the split
- Stratified sampling in `train_test_split` function ensures that the distribution of classes (e.g., LGG and GBM distribution) in original dataset is the same in split-datasets

```
X_train, X_test, y_train, y_test = train_test_split(
    gliomas.drop("Grade", axis=1),
    gliomas["Grade"],
    test_size=0.3,
    random_state=42,
    stratify=gliomas["Grade"],
)
```

```
print("X_train", X_train.shape)
print("X_test", X_test.shape)
print("y_train", y_train.shape)
print("y_test", y_test.shape)
```

```
X_train (587, 22)
X_test (252, 22)
y_train (587,)
y_test (252,)
```

Train a logistic regression model

```
lr = LogisticRegression(solver='lbfgs', max_iter=100)

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

Logistic regression model

- Think of logistic regression as having three distinct layers that work together
  - The Raw Model (Linear Predictor)
  - The Model Output (Probability)
  - The Decision Boundary

The Raw Model (Linear Predictor)

$z = β0 + β1x1 + β2x2 + β3*x3$

Where:

- z is the raw model output
  - Unbounded score that could be any real number from -∞ to +∞
- $β0$ is the intercept or bias term
- $β1, β2, β3$ are the coefficients or weights associated with features $x1, x2, x3$ respectively
- $x1, x2, x3$ are the values of the features

```
sns.histplot(lr.decision_function(X_train))
plt.title('Distribution of Raw model output')
```

The Model Output (Probability)

- Raw score then gets transformed through the sigmoid function
- Sigmoid function: Converts our unbounded raw score into a probability between 0 and 1

```
probabilities = lr.predict_proba(X_test)
print("Probabilities for first 5 test samples:")
print(probabilities[:5])
print("\nNote: Column 0 = P(class=0), Column 1 = P(class=1)")
```

```
Probabilities for first 5 test samples:
[[0.2877258  0.7122742 ]
 [0.30536037 0.69463963]
 [0.86905511 0.13094489]
 [0.24988542 0.75011458]
 [0.91616818 0.08383182]]

Note: Column 0 = P(class=0), Column 1 = P(class=1)
```

```
sns.histplot(lr.predict_proba(X_train))
plt.title('Distribution of Model Output (Probabilities)')
```

```
# Verify the relationship: probability = sigmoid(raw_output)

raw_output = lr.decision_function(X_test)  # All test samples

## `LogisticRegression` model is parameterized to directly model the probability of
class 1, so sigmoid(raw_output) gives P(class=1), not P(class=0)
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

sns.scatterplot(y=sigmoid(raw_output), x=probabilities[:, 1])
plt.xlabel('P(class=1)')
plt.ylabel('Sigmoid(raw_output)')
plt.title('P(class=1) vs Sigmoid(raw_output)')
#plt.plot([0, 1], [0, 1], 'k--', lw=2)
plt.xlim(0, 1)
plt.ylim(0, 1)
plt.show()
```

The Decision Boundary

- Decision Boundary helps make actual yes/no predictions
- Done by setting a threshold on our probability outputs
- Default threshold
  - Raw output = 0 (linear predictor equals zero)
  - Probability = 0.5 (after sigmoid transformation)
- Threshold creates what we call the decision boundary that separates classes
  - If we have two features decision boundary is a line or
  - in higher dimensions, decision boundary is a hyperplane

```
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt

# Reduce to 2D
pca = PCA(n_components=2)
X_train_2d = pca.fit_transform(X_train)
X_test_2d = pca.transform(X_test)

# Train on 2D data
lr_2d = LogisticRegression(solver='lbfgs', max_iter=100)
lr_2d.fit(X_train_2d, y_train)

predicted_classes_2d = lr_2d.predict(X_test_2d)
Z_prob = lr_2d.predict_proba(X_test_2d)[:, 1]

print(Z_prob[:10], X_test_2d[:10,:], predicted_classes_2d[:10])
```

```
[0.60805337 0.50597094 0.47229217 0.6781526  0.0568336  0.81048772
 0.75363624 0.04432821 0.80684454 0.058396  ] [[ 0.74515325 -0.18734103]
 [ 0.56837262 -0.30335269]
 [-0.33389392  0.81342899]
 [ 0.91672279 -0.15814884]
 [-1.81564144  0.51476147]
 [ 0.80692301  0.60144003]
 [ 1.16169866 -0.1718711 ]
 [-1.91648207  0.42722421]
 [ 1.28223986 -0.06832215]
 [-1.47115881  0.06883081]] [1 1 0 1 0 1 1 0 1 0]
```

```
pred_2ds = pd.DataFrame(X_test_2d, columns=['PC1', 'PC2'])
pred_2ds['Predicted_class'] = predicted_classes_2d
pred_2ds['Probability'] = Z_prob
pred_2ds['True'] = y_test.values
pred_2ds.head()
```

```
pred_2ds.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 252 entries, 0 to 251
Data columns (total 5 columns):
 #   Column           Non-Null Count  Dtype  
---  ------           --------------  -----  
 0   PC1              252 non-null    float64
 1   PC2              252 non-null    float64
 2   Predicted_class  252 non-null    int64  
 3   Probability      252 non-null    float64
 4   True             252 non-null    int64  
dtypes: float64(3), int64(2)
memory usage: 10.0 KB
```

```
plt.figure(figsize=(10, 8))

for pred_class in lr_2d.classes_:
    cluster_points = pred_2ds[pred_2ds['Predicted_class'] == pred_class]
    plt.scatter(
        cluster_points['PC1'],
        cluster_points['PC2'],
        label=f'Predictions: {pred_class}',
        alpha=0.7,
        s=80
    )

    # for i, row in cluster_points.iterrows():
    #     plt.text(row['PC1'], row['PC2'], str(round(row['Probability'], 2)),
fontsize=8, alpha=0.7)

plt.xlabel('PC1')
plt.ylabel('PC2')
plt.legend()
plt.show()
```

```
plt.figure(figsize=(10, 8))

# Plot all points with color based on probability
scatter = plt.scatter(
    pred_2ds['PC1'],
    pred_2ds['PC2'],
    c=pred_2ds['Probability'],  # Color based on probability
    cmap='RdBu_r',  # Red for high probability, Blue for low
    s=80,
    alpha=0.8,
    edgecolors='black',
    linewidth=0.5
)

# Add colorbar
plt.colorbar(scatter, label='Probability of Class 1')
```

Predict the Glioma type of the new dataset

```
# lr = LogisticRegression(solver='lbfgs', max_iter=100)

# # Fit the model
# lr.fit(X_train, y_train)
predicted_classes = lr.predict(X_test)
print("Predicted classes for first 10 samples:", predicted_classes[:10])
```

```
Predicted classes for first 10 samples: [1 1 0 1 0 1 1 0 1 0]
```

Examine and understand the importance of features in predicting glioma type

Coefficients of the model

- Magnitude of the coefficients indicates the relative importance of each feature on predicting positive class (in this case 1 - GBM)
- Larger coefficients imply a stronger influence on the predicted probability
- Interpretation of coefficients assumes that the features are independent of each other

```
feature_list = gliomas.columns[1:]
print("Number of Features", len(feature_list))
print("Features", feature_list)
```

```
Number of Features 22
Features Index(['Gender', 'Age_at_diagnosis', 'IDH1', 'TP53', 'ATRX', 'PTEN', 'EGFR',
       'CIC', 'MUC16', 'PIK3CA', 'NF1', 'PIK3R1', 'FUBP1', 'RB1', 'NOTCH1',
       'BCOR', 'CSMD3', 'SMARCA4', 'GRIN2A', 'IDH2', 'FAT4', 'PDGFRA'],
      dtype='object')
```

```
# Create a DataFrame of the coefficients an their corresponding features
coefficients = pd.DataFrame(lr.coef_.T, index=feature_list, columns=['Coefficient'])
coefficients.sort_values(by='Coefficient', ascending=False)
```

```
plt.figure(figsize=(10, 8))
plt.title('Feature Importance')

plt.barh('index', 'Coefficient', align='center',
data=coefficients.sort_values(by='Coefficient', ascending=True).reset_index())
```

Evaluation of the model performance

- Confusion matrix
- Performance evaluation matrices

Confusion matrix

```
# Compute confusion matrix
cm = confusion_matrix(y_test, predicted_classes)

# Plot confusion matrix as heatmap
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix')
plt.show()
```

Performance evaluation matrices

Precision (Positive Predictive Value)

- Precision focuses on the correctness of the positive predictions made by the model
- Signifies the model's ability to avoid misclassifying negative cases as positive
- Higher precision indicates fewer misclassifications (i.e., Higher precision means a smaller proportion of negative cases incorrectly identified as positive)

Precision is percentage of correctly predicted GBM patients out of all predicted GBM patients

```
print("Confusion matrix:\n", cm)
TP = cm[1, 1]
FP = cm[0, 1]

print("True positive count (correct GBM predictions):", TP)
print("False positive count (incorrect GBM predictions):", FP)

precision = TP / (TP + FP)

print("Precision of predicting positive class - GBM:", np.round(precision, 3))
```

```
Confusion matrix:
 [[124  22]
 [ 12  94]]
True positive count (correct GBM predictions): 94
False positive count (incorrect GBM predictions): 22
Precision of predicting positive class - GBM: 0.81
```

```
# Compute precision score
lr_precision = precision_score(y_test, predicted_classes, average=None)
print(pd.DataFrame({"category": lr.classes_, "precision": np.round(lr_precision, 3)}))
```

```
   category  precision
0         0      0.912
1         1      0.810
```

Recall | Sensitivity | True positive rate

This is the **percentage of actual positive cases that the test correctly identifies as positive**. In other words, it measures how good the test is at catching what it's supposed to catch.

Fraction of correctly predicted GBM patients out of actual GBM patients

- Recall
  - Measures the ability of correctly predicting positive cases from the actual positive cases
  - A higher true positive rate signifies a more effective model in correctly identifying positive cases
  - Recall = Fraction of correctly predicted positive observations to the all observations in actual class

```
print("Confusion matrix:\n", cm)
TP = cm[1, 1]
FN = cm[1, 0]
TPR = TP / (TP + FN)
print("True positive count (correct GBM predictions):", TP)
print("False negative count (missed GBM predictions or GBMs predicted as LGGs):", FN)
print("True Positive Rate", np.round(TPR, 3))
```

```
Confusion matrix:
 [[124  22]
 [ 12  94]]
True positive count (correct GBM predictions): 94
False negative count (missed GBM predictions or GBMs predicted as LGGs): 12
True Positive Rate 0.887
```

```
lr_recall = recall_score(y_test, predicted_classes, average=None)
print(pd.DataFrame({"category": lr.classes_, "recall": np.round(lr_recall, 3)}))
```

```
   category  recall
0         0   0.849
1         1   0.887
```

Probability of model predictions

- Model calculates the probability of an observation belonging to a class (i.e., probability of predicting an observation as positive class)
- Default probability used for the classification is 0.5

Probability threshold

- When the threshold is set low, more items are classified as positive, potentially increasing both the number of true positives (TPR) and the number of false positives (FPR)
- As the threshold increases, the classifier becomes more conservative, usually increasing the FPR at a slower rate than the TPR.

```
test_probabilities = lr.predict_proba(X_test)
# Create a DataFrame with observed and predicted values
lr_res = pd.DataFrame({"obs": y_test, "pred_1": test_probabilities[:, 1]})

# plt.figure(figsize=(8, 6)) # not necessary
#sns.boxplot(x="obs", y="pred_1", showfliers=False, data=lr_res, hue="obs")
sns.stripplot(x="obs", y="pred_1", data=lr_res, color="black", alpha=0.3)
plt.axhline(y=0.5, color="red", linestyle="--")

plt.xlabel("Treue Class")
plt.ylabel("Predicted Probability of class 1")
```

Receiver Operating Characteristic

An ROC curve is a graph that shows how good a classification model is at distinguishing between two classes.

- The ROC curve plots the TPR (recall) on the y-axis against the FPR on the x-axis for all possible threshold values.
- False Positive Rate (FPR)

    - This is the percentage of actual negative cases that your test incorrectly identifies as positive. It's measuring how often your test gives false alarms.
    - FPR captures the rate of the model producing a positive prediction when the actual class is negative
    - FPR = proportion of cases that are incorrectly predicted as positive out of negatives

      ```
      FPR = FP / (TN + FP)
      ```
- An ideal classifier would have a point in the upper left corner of the plot, corresponding to a TPR of 1 (perfect recall) and an FPR of 0 (no false alarms).
- A classifier with no discriminative power would have points along the diagonal line from the bottom left to the top right, known as the line of no discrimination. Here, the TPR and FPR are equal, which is similar to random guessing

```
# predicted probabilities (test_probabilities)
y_pred_prob = test_probabilities[:,1]#lr.predict_proba(X_test)[:, 1]

# Compute ROC curve and ROC area
fpr, tpr, thresholds = roc_curve(y_test, y_pred_prob)
roc_auc = auc(fpr, tpr)

# Define example thresholds (adjust as needed)
thresholds_to_show = []
for i in range(1, len(thresholds), 10):
    thresholds_to_show.append(thresholds[i]) # Example thresholds


# Plot ROC curve
plt.figure()
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (area = %0.2f)' %
roc_auc)
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic')

# Loop through thresholds and mark points on ROC curve
for threshold in thresholds_to_show:
  # Find the index of the threshold in the thresholds array
  thresh_idx = np.where(thresholds == threshold)[0]
  # Extract corresponding FPR and TPR values
  fpr_at_thresh = fpr[thresh_idx]
  tpr_at_thresh = tpr[thresh_idx]
  # Plot a marker at the point on the ROC curve
  plt.plot(fpr_at_thresh, tpr_at_thresh, 'o', markersize=8, color='red',
           label=f'Threshold = {threshold:.2f}')

j_statistic = tpr - fpr

# Find the index of the threshold with the maximum J statistic
best_thresh_idx = np.argmax(j_statistic)

# Extract the best threshold, TPR, and FPR values
best_threshold = thresholds[best_thresh_idx]
best_tpr = tpr[best_thresh_idx]
best_fpr = fpr[best_thresh_idx]

thresholds_to_show.append(thresholds[best_thresh_idx])
plt.plot(best_fpr, best_tpr, 'x', markersize=8, color='green', label=f'Best Threshold =
{best_threshold:.2f}')

plt.legend(loc="lower right")
plt.show()
```

```
## Apply custom threshold and make predictions

(y_pred_prob >= best_threshold).astype(int)
```

```
# Compute confusion matrix
fig, ax = plt.subplots(1,2, figsize=(12, 4) )
cm_cust_thresh = confusion_matrix(y_test, (y_pred_prob >= best_threshold).astype(int))

# Plot confusion matrix as heatmap
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', ax=ax[0])
ax[0].set_xlabel('Predicted Label')
ax[0].set_ylabel('True Label')
ax[0].set_title('Confusion Matrix - Default Threshold')

# Plot ROC curve
# Plot confusion matrix as heatmap
sns.heatmap(cm_cust_thresh, annot=True, fmt='d', cmap='Blues', ax=ax[1])
ax[1].set_xlabel('Predicted Label')
ax[1].set_ylabel('True Label')
ax[1].set_title('Confusion Matrix - Custom Threshold')
plt.show()
```

Complete ML workflow

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.metrics import precision_score, recall_score, f1_score, roc_curve, auc,
confusion_matrix, classification_report
from sklearn.model_selection import KFold
```

Exploratory analysis

1. Understand the Data Structure and Summary
    1. Load and Inspect
    2. Descriptive Statistics
2. Analyze the Target Variable
    1. Check Data Type (Ensure your target variable is appropriately represented)
    2. Determine the frequency of each class in your binary target variable
3. Analyze features
    1. Visualize the distributions of features
4. Identify and Handle Missing Values

Understand the Data Structure and Summary

```
gliomas = pd.read_csv('test_data/logistic_reg/TCGA_InfoWithGrade.csv')
gliomas.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 840 entries, 0 to 839
Data columns (total 26 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   Grade             839 non-null    float64
 1   Gender            840 non-null    float64
 2   Age_at_diagnosis  839 non-null    float64
 3   Race              839 non-null    float64
 4   IDH1              839 non-null    float64
 5   TP53              839 non-null    float64
 6   ATRX              839 non-null    float64
 7   PTEN              839 non-null    float64
 8   EGFR              840 non-null    float64
 9   CIC               839 non-null    float64
 10  MUC16             839 non-null    float64
 11  PIK3CA            839 non-null    float64
 12  NF1               839 non-null    float64
 13  PIK3R1            840 non-null    float64
 14  FUBP1             839 non-null    float64
 15  RB1               839 non-null    float64
 16  NOTCH1            840 non-null    float64
 17  BCOR              839 non-null    float64
 18  CSMD3             839 non-null    float64
 19  SMARCA4           839 non-null    float64
 20  GRIN2A            839 non-null    float64
 21  IDH2              840 non-null    float64
 22  FAT4              839 non-null    float64
 23  PDGFRA            840 non-null    float64
 24  ATRX_xNA          630 non-null    float64
 25  IDH1_xNA          168 non-null    float64
dtypes: float64(26)
memory usage: 170.8 KB
```

```
gliomas.head()
```

```
gliomas.describe().T
```

Analyze the Target Variable

```
gliomas["Grade"].dtype
```

```
gliomas["Grade"].value_counts()
```

```
gliomas["Grade"].value_counts(normalize=True)
```

Analyze features

```
# Plot the distribution of data in all columns
fig, axs = plt.subplots(nrows=5, ncols=5, figsize=(15, 20))
axs = axs.flatten()

# Iterate over each column and plot the distribution
for i, column in enumerate(gliomas.drop(columns=["Grade"]).columns):
    axs[i].hist(gliomas[column], bins=20, color='skyblue', edgecolor='black')
    axs[i].set_title(column)
    axs[i].set_xlabel('Value')
    axs[i].set_ylabel('Frequency')

plt.tight_layout()
plt.show()
```

Identify and Handle Missing Values

```
# % of missing values in each column
gliomas.isna().sum()/len(gliomas)*100
```

Keep columns with at least 95% non-missing values

```
threshold = int(0.95 * len(gliomas))
# Keep columns with at least 95% non-missing values
gliomas.dropna(thresh=threshold, axis=1, inplace=True)
```

```
# % of missing values in each column
gliomas.isna().sum()/len(gliomas)*100
```

Delete rows with missing values in target variable

```
gliomas.dropna(subset=["Grade"], axis=0, inplace=True)
```

```
# % of missing values in each column
gliomas.isna().sum()/len(gliomas)*100
```

```
# Plot the distribution of data in all columns
fig, axs = plt.subplots(nrows=5, ncols=5, figsize=(15, 20))
axs = axs.flatten()

# Iterate over each column and plot the distribution
for i, column in enumerate(gliomas.drop(columns=["Grade"]).columns):
    axs[i].hist(gliomas[column], bins=20, color='skyblue', edgecolor='black')
    axs[i].set_title(column)
    axs[i].set_xlabel('Value')
    axs[i].set_ylabel('Frequency')

plt.tight_layout()
plt.show()
```

```
gliomas.drop(columns=["Race"], inplace=True)
```

Split original dataset

```
X_train, X_test, y_train, y_test = train_test_split(
    gliomas.drop("Grade", axis=1),
    gliomas["Grade"],
    test_size=0.3,
    random_state=42,
    stratify=gliomas["Grade"],
)
print("X_train:", X_train.shape, "X_test:", X_test.shape, "y_train:", y_train.shape,
"y_test:", y_test.shape)
```

```
X_train: (587, 22) X_test: (252, 22) y_train: (587,) y_test: (252,)
```

```
# Visulize the distribution of the target variable in the training and training set

fig, axes = plt.subplots(1, 2, figsize=(10, 5))

sns.histplot(X_train[['Age_at_diagnosis']], ax=axes[0])
axes[0].set_title('Training Set')

sns.histplot(X_test[['Age_at_diagnosis']], ax=axes[1])
axes[1].set_title('Test Set')

plt.tight_layout()
plt.show()
```

```
# Fit and transform the 'age' column
## Apply simple transformation using the StandardScaler `scaler = StandardScaler()`
## directly on the 'Age_at_diagnosis' column in train and test datasets
scaler = StandardScaler()
X_train['Age_at_diagnosis'] = scaler.fit_transform(X_train[['Age_at_diagnosis']])
X_test['Age_at_diagnosis'] = scaler.transform(X_test[['Age_at_diagnosis']])
```

```
# Visulize the distribution of the target variable in the training and training set

fig, axes = plt.subplots(1, 2, figsize=(10, 5))

sns.histplot(X_train[['Age_at_diagnosis']], ax=axes[0])
axes[0].set_title('Training Set')

sns.histplot(X_test[['Age_at_diagnosis']], ax=axes[1])
axes[1].set_title('Test Set')

plt.suptitle('Age at Diagnosis Distribution after scaling')
plt.show()
```

Logistic regression model

```
lr = LogisticRegression()

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

```
# Predict the Glioma type of the new dataset
lr.predict(X_test)
```

```
# Examine and understand the importance of features in predicting glioma type
feature_list = gliomas.columns[1:]
print("Number of Features", len(feature_list))
print("Features", feature_list)
```

```
Number of Features 22
Features Index(['Gender', 'Age_at_diagnosis', 'IDH1', 'TP53', 'ATRX', 'PTEN', 'EGFR',
       'CIC', 'MUC16', 'PIK3CA', 'NF1', 'PIK3R1', 'FUBP1', 'RB1', 'NOTCH1',
       'BCOR', 'CSMD3', 'SMARCA4', 'GRIN2A', 'IDH2', 'FAT4', 'PDGFRA'],
      dtype='object')
```

```
# Create a DataFrame of the coefficients an their corresponding features
coefficients = pd.DataFrame(lr.coef_.T, index=feature_list, columns=['Coefficient'])
coefficients.sort_values(by='Coefficient', ascending=False)
```

Evaluate model performance

```
# Predict on test set
y_pred = lr.predict(X_test)

# Compute confusion matrix
cm = confusion_matrix(y_test, y_pred)

# Plot confusion matrix as heatmap
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix')
plt.show()
```

```
# Generate classification report
report = classification_report(y_test, y_pred)
print(report)
```

```
              precision    recall  f1-score   support

         0.0       0.91      0.85      0.88       146
         1.0       0.81      0.89      0.85       106

    accuracy                           0.87       252
   macro avg       0.86      0.87      0.86       252
weighted avg       0.87      0.87      0.87       252
```

The problems with single test dataset (holdout sets) in model validation

- Using different random seeds can lead to different results even when using the same model and dataset
- The variability in results makes it difficult to accurately assess the model's true performance and generalizability
- Cross-validation is proposed as the gold-standard solution to overcome the limitations of holdout sets in model validation

Cross-validation

- Cross-validation is a method that involves running a single model on various training/validation combinations to get more confident final metrics

```
# Use KFold
kf = KFold(n_splits=5, shuffle=True, random_state=1111)

# Create splits
splits = kf.split(gliomas.drop("Grade", axis=1))

# Print the number of indices
for split, k in zip(splits, range(1, 6)):
    print("Fold %d" % k)
    train_index, val_index = split
    print("Number of training indices: %s; First 10: %s" % (len(train_index),
train_index[:10]))
    print("Number of validation indices: %s; First 10: %s" % (len(val_index),
val_index[:10]))
```

```
Fold 1
Number of training indices: 671; First 10: [ 2  3  5  6  8  9 10 11 12 13]
Number of validation indices: 168; First 10: [ 0  1  4  7 23 25 32 33 34 38]
Fold 2
Number of training indices: 671; First 10: [ 0  1  3  4  5  6  7  8  9 10]
Number of validation indices: 168; First 10: [ 2 12 16 24 35 39 45 49 57 61]
Fold 3
Number of training indices: 671; First 10: [ 0  1  2  3  4  6  7  8  9 10]
Number of validation indices: 168; First 10: [ 5 13 26 27 29 36 37 41 46 47]
Fold 4
Number of training indices: 671; First 10: [ 0  1  2  4  5  7  9 12 13 14]
Number of validation indices: 168; First 10: [ 3  6  8 10 11 19 22 31 40 43]
Fold 5
Number of training indices: 672; First 10: [ 0  1  2  3  4  5  6  7  8 10]
Number of validation indices: 167; First 10: [ 9 14 15 17 18 20 21 28 30 42]
```

```
X = gliomas.drop("Grade", axis=1)
y = gliomas["Grade"]

lr_cv = LogisticRegression(max_iter=1000, solver="lbfgs")
# `max_iter` Maximum number of iterations for solvers to converge: Default: 100
# `solver` Algorithm to use in the optimization problem: Default: 'lbfgs'
## 'lbfgs' is an optimization algorithm that is particularly effective for large
datasets

kf = KFold(n_splits=5, shuffle=True, random_state=1111)
splits = kf.split(gliomas.drop("Grade", axis=1))

kf_cv_scores = {
    "Fold": [],
    "Precision_Class_0": [],
    "Precision_Class_1": [],
    "Recall_Class_0": [],
    "Recall_Class_1": [],
    "F1_Class_0": [],
    "F1_Class_1": [],
}

fold = 1
for train_index, val_index in splits:
    # Setup the training and validation data using .iloc for integer-based indexing
    X_train, y_train = X.iloc[train_index], y.iloc[train_index]
    X_val, y_val = X.iloc[val_index], y.iloc[val_index]
    # Fit the logistic regression model
    lr_cv.fit(X_train, y_train)
    # Make predictions, and print the accuracy
    predictions = lr_cv.predict(X_val)

    precision = precision_score(y_val, predictions, average=None)
    recall = recall_score(y_val, predictions, average=None)
    f1 = f1_score(y_val, predictions, average=None)

    kf_cv_scores["Fold"].append(fold)
    kf_cv_scores["Precision_Class_0"].append(round(float(precision[0]),3))
    kf_cv_scores["Recall_Class_0"].append(round(float(recall[0]),3))
    kf_cv_scores["Precision_Class_1"].append(round(float(precision[1]),3))
    kf_cv_scores["Recall_Class_1"].append(round(float(recall[1]),3))
    kf_cv_scores["F1_Class_0"].append(round(float(f1[0]),3))
    kf_cv_scores["F1_Class_1"].append(round(float(f1[1]),3))
    fold += 1

kf_cv_scores
```

```
kf_cv_scores_df = pd.DataFrame(kf_cv_scores)
kf_cv_scores_df
```

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

# Line plot showing variation across folds
ax1.plot(kf_cv_scores['Fold'], kf_cv_scores['Precision_Class_0'], '-o',
label='Precision_Class_0')
ax1.plot(kf_cv_scores['Fold'], kf_cv_scores['Precision_Class_1'], '-*',
label='Precision_Class_1')

ax1.set_xlabel('Fold')
ax1.set_xticks(kf_cv_scores['Fold'])
ax1.set_ylabel('Precision Score')
ax1.set_ylim(0, 1)
ax1.legend(loc='lower right')
ax1.grid(True, alpha=0.3)

ax2.plot(kf_cv_scores['Fold'], kf_cv_scores['Recall_Class_0'], '-s',
label='Recall_Class_0')
ax2.plot(kf_cv_scores['Fold'], kf_cv_scores['Recall_Class_1'], '-*',
label='Recall_Class_1')

ax2.set_xlabel('Fold')
ax2.set_xticks(kf_cv_scores['Fold'])
ax2.set_ylabel('Recall Score')
ax2.set_ylim(0, 1)
ax2.legend(loc='lower right')
ax2.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```

```
cv_summary = kf_cv_scores_df.aggregate(
    {
        "Precision_Class_0": ["min", "max", "mean", "std"],
        "Precision_Class_1": ["min", "max", "mean", "std"],
        "Recall_Class_0": ["min", "max", "mean", "std"],
        "Recall_Class_1": ["min", "max", "mean", "std"],
        "F1_Class_0": ["min", "max", "mean", "std"],
        "F1_Class_1": ["min", "max", "mean", "std"],
    }
)
cv_summary
```

```
cv_summary.T
```

```
cv_summary.T[["std","min", "mean", "max"]].plot(kind="bar", figsize=(12, 5),
title="Cross Validation Summary Statistics")
plt.xticks(rotation=45)
plt.ylabel("Score")
plt.xlabel("Metric")
plt.legend(bbox_to_anchor=(1.01, 1), loc='upper left')
```

Interpretation guidelines:

1. Mean Performance:
   - Higher mean = better overall performance
   - Compare to baseline or business requirements
2. Standard Deviation:
   - Low SD = consistent performance across folds (good generalization)
   - High SD = variable performance (potential overfitting or data issues)
3. Range and Min/Max:
   - Small range = consistent across different data subsets
   - Large range = sensitive to specific data characteristics

Hyperparameter Tuning

Hyperparameters:

- Set before training begins
- Configure model architecture/behavior
- Not learned; require manual tuning
- Core hyperparameters in `LogisticRegression`:
  - Penalty (loss): 'l1', 'l2', 'elasticnet', None (Default: 'l2')
  - Regularization strength: smaller values mean stronger regularization (Default: 1.0)
  - Solver (Algorithm to use for optimization): `newton-cg`, `lbfgs`, `liblinear`, `sag`, `saga` (Default: `lbfgs`)
- Core Hyperparameters can
  - Directly influence model's learning process and capabilities
  - Control model complexity, learning behavior, and prevention of overfitting
  - Changes significantly impact accuracy, generalization, and prediction quality

```python
# Get the features and target variable
X_train, X_test, y_train, y_test = train_test_split(
    gliomas.drop("Grade", axis=1),
    gliomas["Grade"],
    test_size=0.3,
    random_state=42,
    stratify=gliomas["Grade"],
)

# Create the base model
lr = LogisticRegression(random_state=42, max_iter=10000)

# Define the parameter grid
# Define a VALID parameter grid
param_grid = [
    # For l2 penalty (works with all solvers)
    {
        'penalty': ['l2'],
        'C': [0.001, 0.01, 0.1, 1, 10, 100],
        'solver': ['lbfgs', 'liblinear', 'newton-cg', 'sag', 'saga']
    },
    # For l1 penalty (only works with liblinear and saga)
    {
        'penalty': ['l1'],
        'C': [0.001, 0.01, 0.1, 1, 10, 100],
        'solver': ['liblinear', 'saga']
    },
    # For elasticnet penalty (only works with saga)
    # Note: l1_ratio must be between 0 and 1
    {
        'penalty': ['elasticnet'],
        'l1_ratio': [0.1, 0.5, 0.9],
        'solver': ['saga'],
        'C': [0.1, 1, 10]

    }
]

# Create GridSearchCV object
grid_search = GridSearchCV(
    estimator=lr,
    param_grid=param_grid,
    cv=5,  # 5-fold cross-validation
    scoring='f1_macro',  # Assessment metric
    verbose=1,  # Print progress
    n_jobs=-1  # Use all CPU cores
)

# Fit the grid search
grid_search.fit(X_train, y_train)

# Access results for different metrics
print(f"Best parameters (based on f1):", grid_search.best_params_)
print(f"Best f1 score:", grid_search.best_score_)

# Make predictions with the best model
best_model = grid_search.best_estimator_
y_pred = best_model.predict(X_test)

# Evaluate the best model
print("\nTest set accuracy:", best_model.score(X_test, y_test))
print("\nClassification Report:")
print(classification_report(y_test, y_pred))
```

```
Fitting 5 folds for each of 51 candidates, totalling 255 fits
```

```
Best parameters (based on f1): {'C': 1, 'penalty': 'l1', 'solver': 'liblinear'}
Best f1 score: 0.8708974463864253

Test set accuracy: 0.8690476190476191

Classification Report:
              precision    recall  f1-score   support

         0.0       0.92      0.85      0.88       146
         1.0       0.81      0.90      0.85       106

    accuracy                           0.87       252
   macro avg       0.87      0.87      0.87       252
weighted avg       0.87      0.87      0.87       252
```

```
# After fitting
results_df = pd.DataFrame(grid_search.cv_results_)
results_df.head().T
```

```
# Compute confusion matrix
cm = confusion_matrix(y_test, y_pred)

# Plot confusion matrix as heatmap
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix')
plt.show()
```

```
# predicted probabilities (test_probabilities)
y_pred_prob = best_model.predict_proba(X_test)[:,1] #lr.predict_proba(X_test)[:, 1]

# Compute ROC curve and ROC area
fpr, tpr, thresholds = roc_curve(y_test, y_pred_prob)
roc_auc = auc(fpr, tpr)

# Plot ROC curve
plt.figure()
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (area = %0.2f)' %
roc_auc)
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic')

j_statistic = tpr - fpr

# Find the index of the threshold with the maximum J statistic
best_thresh_idx = np.argmax(j_statistic)

# Extract the best threshold, TPR, and FPR values
best_threshold = thresholds[best_thresh_idx]
best_tpr = tpr[best_thresh_idx]
best_fpr = fpr[best_thresh_idx]

plt.plot(best_fpr, best_tpr, 'x', markersize=8, color='green', label=f'Best Threshold =
{best_threshold:.2f}')
plt.legend(loc="lower right")
plt.show()
```

```
## Apply custom threshold and make predictions

(y_pred_prob >= best_threshold).astype(int)
```

```
# Compute confusion matrix
fig, ax = plt.subplots(1,2, figsize=(12, 4) )
cm_cust_thresh = confusion_matrix(y_test, (y_pred_prob >= best_threshold).astype(int))

# Plot confusion matrix as heatmap
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', ax=ax[0])
ax[0].set_xlabel('Predicted Label')
ax[0].set_ylabel('True Label')
ax[0].set_title('Confusion Matrix - Default Threshold')

# Plot ROC curve
# Plot confusion matrix as heatmap
sns.heatmap(cm_cust_thresh, annot=True, fmt='d', cmap='Blues', ax=ax[1])
ax[1].set_xlabel('Predicted Label')
ax[1].set_ylabel('True Label')
ax[1].set_title('Confusion Matrix - Custom Threshold')
plt.show()
```

Quick Reference

Python dependencies

- Download requirements file `requirements.txt`