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
- Logistic regression in cancer genomics
- ML workflow with biological data
- Deep-learning-based variant calling via DeepVariant
- Accelerated Genomics workflows with Parabricks

Unsupervised Learning

Principal component analysis (PCA) and K-means clustering

⚙ Prerequisites

- BioNT Applied Machine Learning for Biological Data
  - Module 1: Python Numpy and Pandas

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

✍ Time

2 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

Note

ML use-case

- Drug sensitivity scores: 50 drugs and 25 patients
- Unsupervised learning (PCA and clustering) analysis will
  1. Transform the drug sensitivity data (high-dimensional) into a dataset (lower-dimensional) that capture the most significant variance and patterns
  2. Group patients into distinct strata based on similarities in their overall drug sensitivity patterns

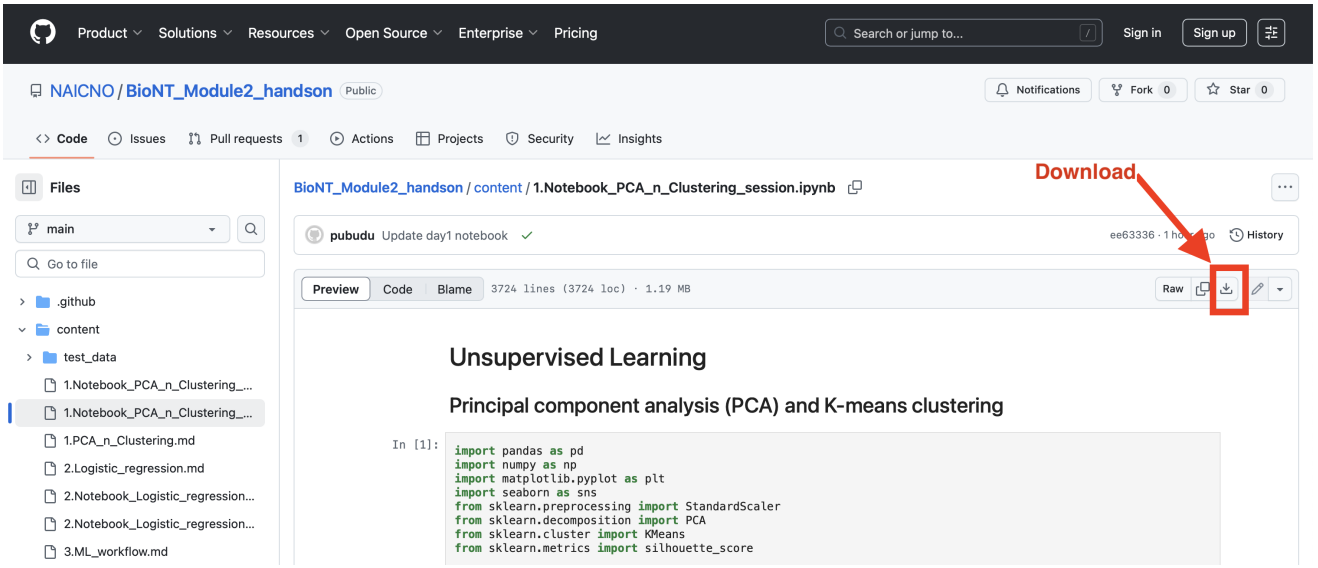
Dataset

- 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

download test dataset

Notebook

- Download the notebook



Classification

Logistic regression

⚙ Prerequisites

- BioNT Applied Machine Learning for Biological Data

- Module 1: Python Numpy and Pandas

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

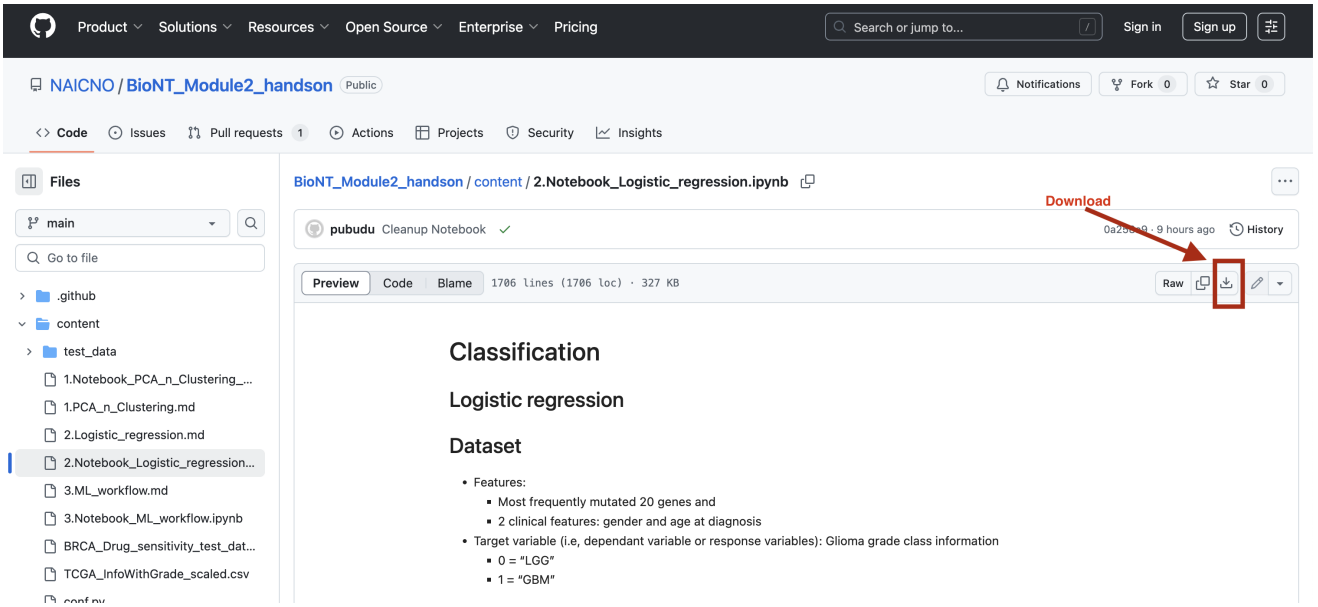
- Download dataset: TCGA_InfoWithGrade_scaled.csv
- Features:
  - Most frequently mutated 20 genes and
  - 2 clinical features: gender and age at diagnosis
- Target variable (i.e, dependant variable or response variables): Glioma grade class information
  - 0 = “LGG”
  - 1 = “GBM”

Source

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

Notebook

- Download the notebook



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.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

Notebook

- Download the notebook

DeepVariant: Deep-learning tool

✍ Time

- Lecture: 20 minutes
- Exercise: 10 mins
- hands-on: 30 mins

❗ Objectives

- We will introduce two such tools used in variant calling and functional effect prediction
    - DeepVariant: Live session
    - AlphaMissense (Additional notes, not intended to cover in live session)

Variant calling

- Variant calling is the process of identifying of variants from sequence data

- Compare the sequence data from an individual to a reference genome to identify differences

A reference genome sequence (green) is used for comparing individual human genome sequences (purple) to find genomic variants. Specifically, the side-by-side comparison of a newly generated human genome sequence with a reference human genome sequence allows for the detection of genomic variants in the former. Although only one generated human genome sequence is shown for the purposes of simplification, in reality, each person actually has two genome sequences (one inherited from each parent), both of which would be compared to the reference genome sequence.

Reference: https://www.genome.gov/about-genomics/educational-resources/fact-sheets/human-genomic-variation

Source

Data preprocessing steps for variant calling

- Will be discussed in detail on Day-5 sessions

Main input for variant calling

- Alignment file used can be think of as a dataset representing sequence reads that are aligned to a reference genome - i.e., sequence reads are pileup along the reference genome

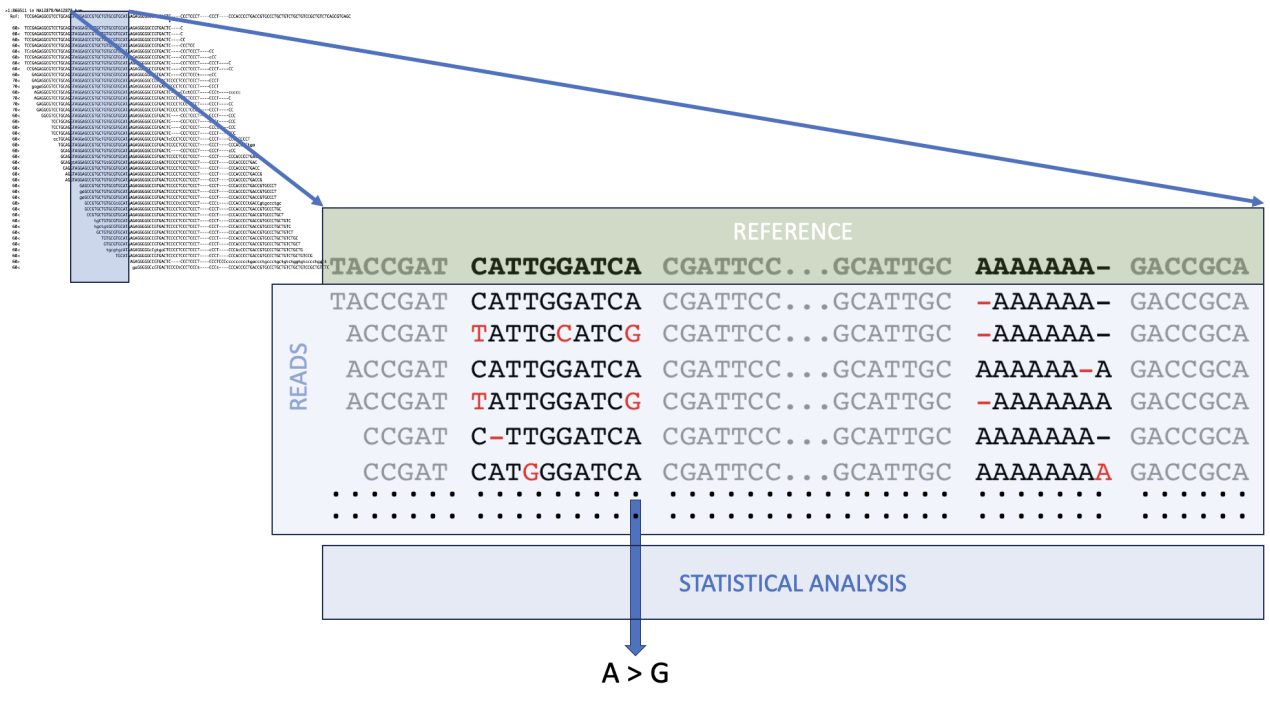
```
>1:866511 in NA12878/NA12878.bam
  Ref:  TCCGAGAGGCGTCCTGCAGGTAGGAGCCGTGCTGTGCGTGCATAAGAGGGGGCCGTGACTC----CCCTCCCT----CCCT----CCCACCCCTGACCGTGCCCTGCTGTCTGCTGTCCGCTGTCTCAGCGTGAGC
                                                                                *
  60>  TCCGAGAGGCGTCCTGCAGGTAGGAGCCGTGCTGTGCGTGCATAAGAGGGGGCCGTGACTC----C
  60<  TCCGAGAGGCGTCCTGCAGGTAGGAGCCGTGCTGTGCGTGCATAAGAGGGGGCCGTGACTC----C
  60>  TCCGAGAGGCGTCCTGCAGGTAGGAGCCGTGCTGTGCGTGCATAAGAGGGGGCCGTGACTC----CC
  60<  TCCGAGAGGCGTCCTGCAGGTAGGAGCCGTGCTGTGCGTGCATAAGAGGGGGCCGTGACTC----CCCTCC
  60<  TCcGAGAGGCGTCCTGCAGGTAGGAGCCGTGCTGTGCGTGCATAAGAGGGGGCCGTGACTC----CCCTCCCT----CC
  60>  TCCGAGAGGCGTCCTGCAGGTAGGAGCCGTGCTGTGCGTGCATAAGAGGGGGCCGTGACTC----CCCTCCCT----cCC
  60<  TCCGAGAGGCGTCCTGCAGGTAGGAGCCGTGCTGTGCGTGCATAAGAGGGGGCCGTGACTC----CCCTCCCT----CCCT----C
  60<   CCGAGAGGCGTCCTGCAGGTAGGAGCCGTGCTGTGCGTGCATAAGAGGGGGCCGTGACTC----CCCTCCCT----CCCT----CC
  60>     GAGAGGCGTCCTGCAGGTAGGAGCCGTGCTGTGCGTGCATAAGAGGGGGCCGTGACTC----CCCTCCCt----cCC
  70<     GAGAGGCGTCCTGCAGGTAGGAGCCGTGCTGTGCGTGCATAAGAGGGGGCCGTGACTCCCCTCCCTCCCT----CCCT
  70<     gagaGGCGTCCTGCAGGTAGGAGCCGTGCTGTGCGTGCATAAGAGGGGGCCGTGACTCCCCTCCCTCCCT----CCCT
  60>      AGAGGCGTCCTGCAGGTAGGAGCCGTGCTGTGCGTGCATAAGAGGGGGCCGTGACTC----CCctCCCT----CCCt----ccccc
  70<      AGAGGCGTCCTGCAGGTAGGAGCCGTGCTGTGCGTGCATAAGAGGGGGCCGTGACTCCCCTCCCTCCCT----CCCT----C
  70>       GAGGCGTCCTGCAGGTAGGAGCCGTGCTGTGCGTGCATAAGAGGGGGCCGTGACTCCCCTCCCTCCCT----CCCT----CC
  70<       GAGGCGTCCTGCAGGTAGGAGCCGTGCTGTGCGTGCATAAGAGGGGGCCGTGACTCCCCTCCCTCCCT----CCCT----CC
  60<         GGCGTCCTGCAGGTAGGAGCCGTGCTGTGCGTGCATAAGAGGGGGCCGTGACTC----CCCTCCCT----CCCT----CCC
  60>             TCCTGCAGGTAGGAGCCGTGCTGTGCGTGCATAAGAGGGGGCCGTGACTC----CCCTCCCT----CCCt----CCC
  60>             TCCTGCAGGTAGGAGCCGTGCTGTGCGTGCATAAGAGGGGGCCGTGACTC----CCCTCCCT----CCCt----CCC
  60<             TCCTGCAGGTAGGAGCCGTGCTGTGCGTGCATAAGAGGGGGCCGTGACTC----CCCTCCCT----CCCT----CCC
  60<              ccTGCAGGTAGGaGCCGTGcTGTGCGTGCATAAGAGGGGGCCGTGACTcCCCTCCCTCCCT----CCCT----CCCACCCCT
  60>                TGCAGGTAGGAGCCGTGCTGTGCGTGCATAAGAGGGGGCCGTGACTCCCCTCCCTCCCT----CCCT----CCCACCCCtga
  60>                 GCAGGTAGGAGCCGTGCTGTGCGTGCATAAGAGGGGGCCGTGACTC----CCCTCCCT----CCCT----cCC
  60<                 GCAGGTAGGAGCCGTGCTGTGCGTGCATAAGAGGGGGCCGTGACTCCCCTCCCTCCCT----CCCT----CCCACCCCTGAC
  60<                 GCAGgtAGGAGCCGTGCTGtGCGTGCATAAGAGGGGGCCGtGACTCCCCTCCCTCCCT----CCCT----CCCACCCCTGAC
  60<                  CAGGTAGGAGCCGTGCTGTGCGTGCATAAGAGGGGGCCGTGACTCCCCTCCCTCCCT----CCCT----CCCACCCCTGACC
  60<                   AGGTAGGAGCCGTGCTGTGCGTGCATAAGAGGGGGCCGTGACTCCCCTCCCTCCCT----CCCT----CCCACCCCTGACCG
  60<                   AGGTAGGAGCCGTGCTGTGCGTGCATAAGAGGGGGCCGTGACTCCCCTCCCTCCCT----CCCT----CCCACCCCTGACCG
  60<                         GAGCCGTGCTGTGCGTGCATAAGAGGGGGCCGTGACTCCCCTCCCTCCCT----CCCT----CCCACCCCTGACCGTGCCCT
  60<                         gaGCCGTGCTGTGCGTGCATAAGAGGGGGCCGTGACTCCCCTCCCTCCCT----CCCT----CCCACCCCTGACCGTGCCCT
  60<                         gaGCCGTGCTGTGCGTGCATAAGAGGGGGCCGTGACTCCCCTCCCTCCCT----CCCT----CCCACCCCTGACCGTGCCCT
  60>                           GCCGTGCTGTGCGtGCATAAGAGGGGGCCGTGACTCCCCtCCCTCCCT----CCCt----CCCACCCCtGACCgtgccctgc
  60<                           GCCGTGCTGTGCGTGCATAAGAGGGGGCCGTGACTCCCCTCCCTCCCT----CCCT----CCCACCCCTGACCGTGCCCTGC
  60<                            CCGTGCTGTGCGTGCATAAGAGGGGGCCGTGACTCCCCTCCCTCCCT----CCCT----CCCACCCCTGACCGTGCCCTGCT
  60<                               tgCTGTGCGTGCATAAGAGGGGGCCGTGACTCCCCTCCCTCCCT----CCCT----CCCACCCCTGACCGTGCCCTGCTGTC
  60<                               tgctgtGCGTGCATAAGAGGGGGCCGTGACTCCCCTCCCTCCCT----CCCT----CCCACCCCTGACCGTGCCCTGCTGTC
  60<                                GCTGTGCGTGCATAAGAGGGGGCCGTGACTCCCCTCCCTCCCT----CCCT----CCCaCCCCTGACCGTGCCCTGCTGTCT
  60<                                  TGTGCGTGCATAAGAGGGGGCCGTGACTCCCCTCCCTCCCT----CCCT----CCCACCCCTGACCGTGCCCTGCTGTCTGC
  60<                                   GTGCGTGCATAAGAGGGGGCCGTGACTCCCCTCCCTCCCT----CCCT----CCCACCCCTGACCGTGCCCTGCTGTCTGCT
  60<                                    tgcgtgcATAAGAGGGGGcCgtgaCTCCCCTCCCTCCCT----cCCT----CCCAcCCCTGACCGTGCCCTGCTGTCTGCTG
  60<                                        TGCATAAGAGGGGGCCGTGACTCCCCTCCCTCCCT----CCCT----CCCACCCCTGACCGTGCCCTGCTGTCTGCTGTCCG
  60>                                              AGAGGGGGCCGTGACTC----CCCTCCCT----CCCTCCCccccccccctgaccctgccctgctgtctggtgtcccctggct
  60<                                               gaGGGGGCcGTGACTCCCCtCCCTCCCt----CCCt----CCCACCCCTGACCGTGCCCTGCTGTCTGCTGTCCGCTGTCTC
```

Source

Standard variant calling tools

Standard variant calling tools are based on statistical models and various QC parameters

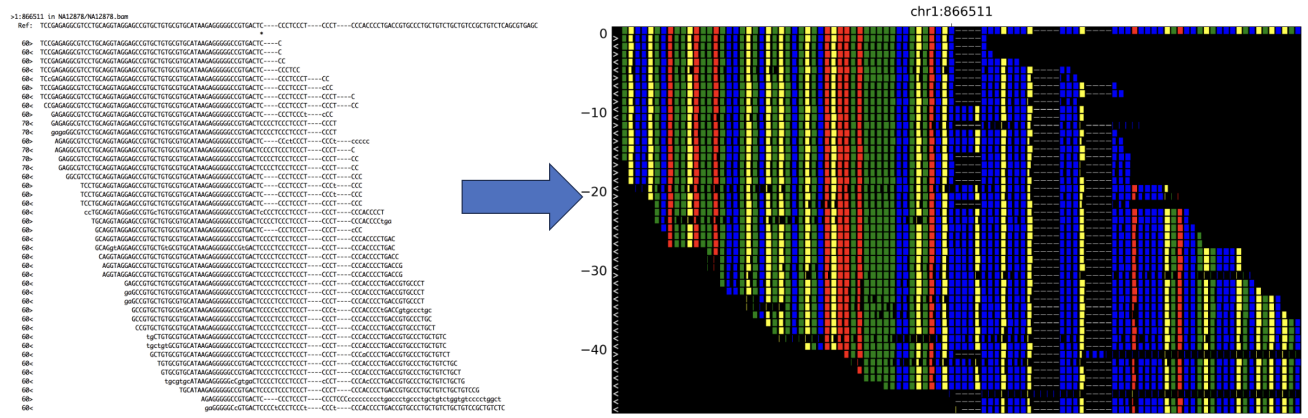
- These tools first analyze alignment files to detect read-positions that differ from reference
- Apply statistical methods combining various information (nucleotide and QC parameters) of these read-positions to identify genomic variants



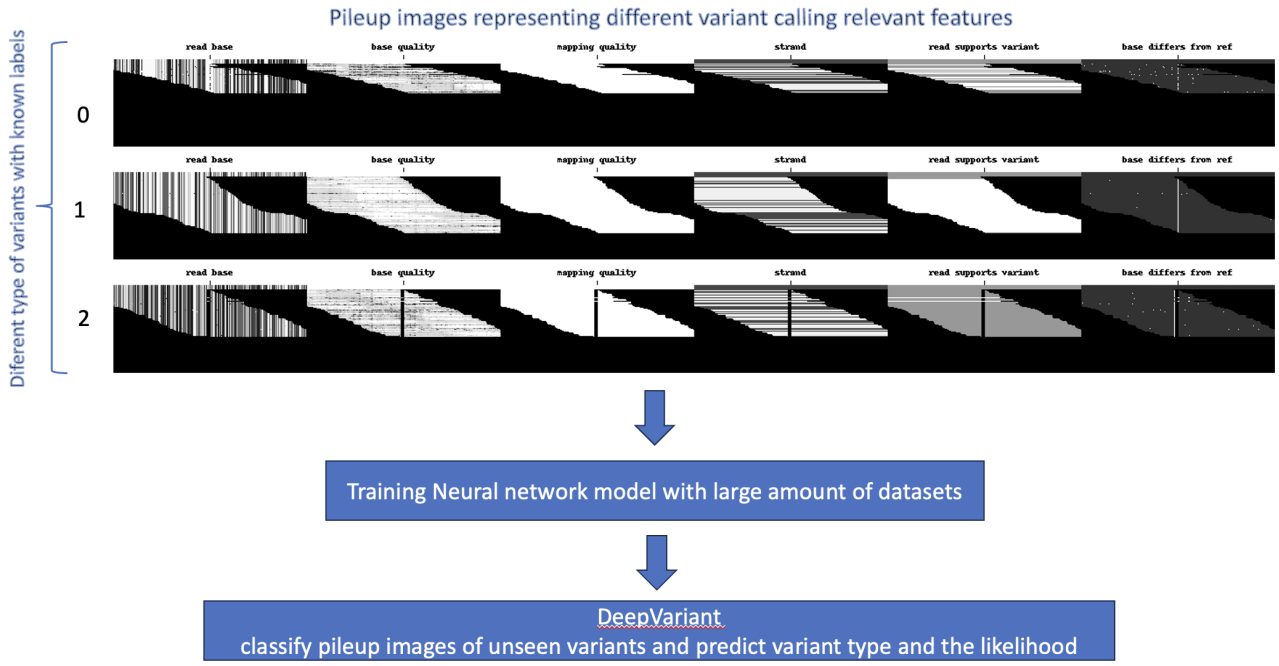
```
#Run Haplotype Caller
$ gatk HaplotypeCaller \
     --java-options -Xmx30g \
     --input ${INPUT_BAM} \
     --output ${OUT_VCF} \
     --reference ${REFERENCE} \
     --native-pair-hmm-threads ${CPU}
```

Deep learning based variant caller - DeepVariant

- Various visualization techniques have been used to validate regions of alignment files with variants (e.g., Pileup-images)



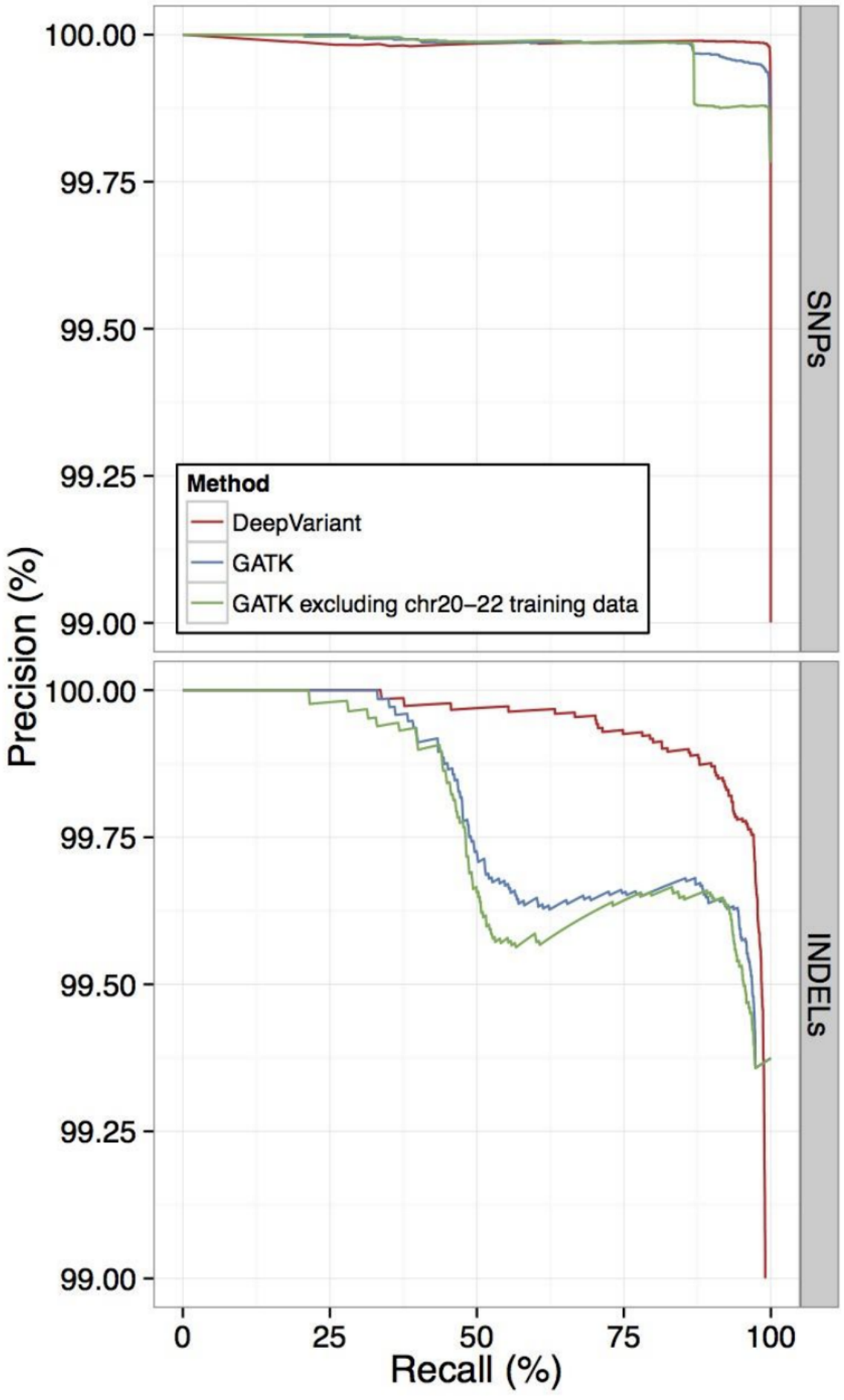
- DeepVariant leverages this concept of pileup-images to visualize not only the bases, but other features that are important for variant calling
  - DeepVariant generates sets of images for candidate variant positions representing range of features
  - Stack of pileup images each representing
    - Base calling quality
    - Mapping quality
    - Metadata on where position is reference or not
    - etc…
- Availability of these images transform variant calling a image classification problem
- DeepVariant use deep-learning model to classify these images and predict variants with high precision



DeepVariant vs traditional variant callers

- DeepVariant showed higher Precision and sensitivity scores compared traditional callers (Ref: Original DeepVariant paper and Independent studies)

✔ Precision vs Recall plot



- High accuracy of DeepVariant compared to traditional callers:

- DeepVariant won 2020 PrecisionFDA Truth Challenge V2 for all Benchmark Regions across Multiple sequencing Technologies
- DeepVariant - best SNP Performance in 2016 PrecisionFDA Truth Challenge
- DeepVariant makes a great difference especially for low coverage samples
- References are linked in DeepVariant GitHub repo

✍ Exercise

- Why DeepVariants (deep-learning based) could outperform traditional variant callers?

DeepVariant model training and evaluation

- This training dataset consist of 100s of millions of samples from multiple genomes, sequencers, and preparation methods
- This help minimize the bias in the model towards a specific sequencing platform or technology

✔ DeepVariant training data

Ref: DeepVariant training data

- Model is evaluated using unseen data from [precisionFDA Truth Challenge] (https://precision.fda.gov/challenges/truth/results

❗ Note

Hands-on: DeepVariant run

- Log into VM following instructions given in previous session

Run inside the VM

```
# Move to home directory
cd $HOME

# Check your current working directory (you'll see e.g., /home/biont*)
pwd

# Run docker interactive mode
docker run \
-it \
--rm \
--gpus all \
-v /data:/data \
-v $PWD:$PWD \
-w $PWD \
nvcr.io/nvidia/clara/clara-parabricks:4.3.0-1 bash
```

Now you are inside the docker container

```
# Set path variable (i.e, copy following lines)

FASTA="/data/ngs/ref/Homo_sapiens_assembly38.fasta"
KNOWN_SITES="/data/ngs/ref/Homo_sapiens_assembly38.known_indels.vcf.gz"
BAM=/data/ngs/BAM/dw_sample.bam

# Run DV command & generate deepvariant.vcf output file (i.e, copy following
lines)
pbrun deepvariant --ref ${FASTA} \
--in-bam ${BAM} \
--num-gpus 1 \
--logfile dv.log \
--out-variants deepvariant.vcf

## You can exit the docker with `exit` command
```

- Inspect the DeepVariant output `deepvariant.vcf`

AlphaMissense

▶ AlphaMissense notes:

Unsupervised Learning

Principal component analysis (PCA) and K-means clustering

```
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

```
! curl
https://naicno.github.io/BioNT_Module2_handson/_downloads/d10b2868b0538b6d8e941c3b4de06a4
-o BRCA_Drug_sensitivity_test_data.csv
! ls -lh BRCA_Drug_sensitivity_test_data.csv
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed

  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
```

```
100 18108  100 18108    0     0   113k      0 --:--:-- --:--:-- --:--:--  113k
100 18108  100 18108    0     0   113k      0 --:--:-- --:--:-- --:--:--  113k
```

```
-rw-r--r-- 1 runner docker 18K Jun 10 13:52 BRCA_Drug_sensitivity_test_data.csv
```

```
data = pd.read_csv('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
```

Dimensionality of the datasets

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.drop(columns=["Patient
ID"]).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('float64')]
```

```
# Basic statistics for drug sensitivity values
data.describe(include='all').T.sort_values('mean', ascending=False).head(10)
```

```
data.describe(include='all').T.sort_values('mean', ascending=False).tail(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);
```

Variance plots

```
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 (Standardization)

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);
```

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)

## `fit()` in PCA:
# Calculates the principal components (eigenvectors) and their explained variances
# Determines the transformation matrix based on your data
# Stores these components internally in the PCA object
# Does not actually transform your data

## `transform()` in PCA:
# Projects your data onto the principal components using the previously calculated
transformation matrix
# Actually reduces the dimensionality of your data
# Example: pca.transform(X_test) projects test data onto the same components learned
from training data

## `fit_transform()` in PCA:
# Combines both operations: calculates principal components and then projects your data
# Equivalent to calling fit() followed by transform()
# More efficient than calling them separately
# Example: pca.fit_transform(X_train) learns components from training data and
immediately projects it
```

```
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 50 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
- Why focus on maximizing variance?
  - By capturing most of the variation in the first few uncorrelated PCs, PCA allows you to summarize a large set of potentially correlated variables with a smaller number of representative variables
  - This process can be viewed as identifying directions in the original feature space along which the data vary the most, or finding low-dimensional linear surfaces that are closest to the observations

    - This is valuable for dimension reduction and compression, especially when dealing with many attributes that might be redundant
- By transforming the original correlated variables into a set of uncorrelated principal components, PCA effectively removes redundancy in the data
    - PCA transforms the original correlated variables into a new set of uncorrelated variables (the principal components), ordered by how much variance they explain. This allows you to potentially use a smaller subset of these new, uncorrelated components to represent the data while retaining most of the original variability
- **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

```
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))

drug_data_normalised.plot(kind="box", title="Drug Sensitivity Boxplot", ax=axes[0])
axes[0].set_xticklabels(axes[0].get_xticklabels(), rotation=90)  # Rotate x-labels for
axes[0]
pc_df.plot(kind="box", title="PCA Components Boxplot", ax=axes[1])
plt.tight_layout()
```

Note:

- PCA successfully achieved its goal of decorrelating our dataset
- Decorrelation:
  - The variance originally shared between features (correlated features) in the original dataset has been reorganized and captured along these new, independent PC axes
- Correlated variables inherently contain overlapping information (shared variation)
- When variables are correlated, it means their values tend to change together to some degree
  - This implies that they are capturing some of the same underlying patterns or variance in the data
  - Dealing with a large set of correlated variables suggests there might be redundancy
  - The correlation structure can be thought of as arising from shared underlying sources or latent variables.
- PCA transforms these into uncorrelated components
  - The process of Principal Component Analysis explicitly finds a new set of variables - principal components (PCs), which are linear combinations of the original features
  - A key property of these principal components is that they are uncorrelated with each other
  - The second principal component is defined as having maximal variance out of all linear combinations that are uncorrelated with the first principal component, and this extends to subsequent components
- Decorrelation ensures each PC captures a distinct aspect of variability:
  - Because each subsequent principal component is constrained to be uncorrelated with the ones already found, it is forced to capture variation in a direction that is distinct from the directions represented by the earlier components
  - The first PC captures the most variance. The second PC captures the most variance that is not captured by the first PC (due to the uncorrelated constraint), and so on.
  - This ensures that each PC adds new, non-overlapping information about the data's variability, allowing the first few components to collectively explain a "sizable amount of the variation" and provide a lower-dimensional representation that retains "as much of the information as possible"

```
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:

- 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 principal component (PC):**
  - Each PC captures a specific amount of variance (this is also known as variance explained by each PC)
  - For a specific PC, say the `m-th` one, its variance explained is essentially the variance of the scores across `m-th` PC, when the data points are projected principal components
- **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 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.axvline(x=10, color='r', linestyle='--')
plt.axhline(y=0.90, color='r', linestyle='dotted', label='90% explained variance
threshold')
plt.legend(loc='best')
plt.tight_layout()
plt.show()
```

```
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 Explained Variance')
plt.grid()
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)
# 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 + 1}")  # +1 because
index starts at 0
```

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

Scree Plot

- 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)

Logic Behind Using a Scree Plot:

1. Understanding Eigenvalues

- Eigenvalues are the foundation of scree plots. In PCA, eigenvalues represent the amount of variance captured by each principal component

- Eigenvalue associated with a specific eigenvector is equal to the variance of the corresponding principal component
- Specifically:
  - Each eigenvalue corresponds to one principal component
  - The magnitude of an eigenvalue indicates how much variance that component explains
  - Eigenvalues are measured in the same units as the original data's variance (When PCA is performed on the standardized data, eigenvalues are unitless because standardization removes units)
  - The sum of all eigenvalues equals the total variance in the dataset

3. The Scree Plot Displays Eigenvalues

- A scree plot displays the **eigenvalues** (not directly the proportion of variance explained) on the Y-axis for each principal component on the X-axis
- The eigenvalues are plotted in descending order, creating the characteristic "scree" or slope appearance.

4. Relationship to Proportion of Variance Explained (PVE):

- **PVE** for the m-th principal component = (Eigenvalue of m-th component) / (Sum of all eigenvalues)
- Together, all principal components explain 100% of the variance (sum of all PVEs = 1)

5. Identifying the "Elbow"

- The visual logic involves examining the scree plot and looking for a point where the eigenvalues drop off noticeably
- This sharp drop-off is often referred to as an "elbow" in the plot
- Since eigenvalues directly represent variance amounts, a steep decline indicates that subsequent components capture significantly less variance.

6. Interpreting the Elbow
   - The idea is that the components before the elbow have **large eigenvalues** and therefore capture a "sizable amount of the variation," representing the key dimensions along which the data vary the most
   - Components after the elbow have **small eigenvalues** and explain significantly less additional variance, making them less "interesting" or necessary for understanding the main patterns in the data.
7. By choosing the number of components up to the elbow, you aim to retain the minimum number of components required to explain a substantial portion of the total variance while achieving dimensionality reduction.

Important Considerations:

- This method of choosing the number of components by inspecting the scree plot is **"ad hoc" and "inherently subjective"**
- There isn't a single, universally accepted objective rule based solely on the scree plot
- The decision often depends on the specific application and the dataset
- In practice, analysts might look at the first few components for interesting patterns and continue examining subsequent ones until no further insights are gained
- Alternative methods include using cumulative proportion of variance explained (e.g., retaining components that explain 80-90% of total variance) or cross-validation approaches

```
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()
```

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 are
  - the weights that transform original, potentially correlated variables into a new set of principal components
  - i.e., 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**

Loading and correlation:

- Loadings are not correlations themselves - they are the coefficients (weights) in the linear combination that defines the principal component
- In PCA (with standardized data), loadings are proportional to correlations between original variables and PC scores

Positive Loading (e.g., +0.7):

- This indicates a positive correlation
- When the score of the principal component increases, the value of the original feature also tends to increase. Conversely, when the PC score decreases, the feature's value tends to decrease.

Negative Loading (e.g., -0.6):

- This indicates a negative correlation
- When the score of the principal component increases, the value of the original feature tends to decrease. Conversely, when the PC score decreases, the feature's value tends to increase

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])]
)

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

Logic - Minimizing Variation:

- The core logic of K-means is to find a partition that minimizes the within-cluster variation.
- The algorithm aims to partition the observations into K clusters such that the total within-cluster variation, summed over all K clusters, is minimized.
- The objective being minimized is the sum of squared distances between each point and the centroid of its assigned cluster

Inertia (within-cluster sum-of-squares)

- Inertia is a mathematical measure that quantifies how tightly grouped the data points are within their assigned clusters

- Definition: Inertia is the total within-cluster sum of squares. It is a measure of how internally coherent the clusters are
- K-means clustering uses an iterative optimization strategy to minimize inertia
- 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 using inertia only raises some limitations
  - It measures how similar each point is to its own cluster (cohesion)
  - but do not evaluate how different it is from other clusters (separation)
- Silhouette Score is a measure of how similar an object is to its own cluster compared to other clusters
- Measures how similar a data point is to its own cluster (cohesion) compared to other clusters (separation)
- While Inertia is the objective function that the K-means algorithm directly tries to minimize during its iterative process, the Silhouette Score is a measure used to evaluate the quality of the clustering result after the algorithm has finished. It is not part of the K-means algorithm's iterative assignment and update steps.

Is each data point placed in the right cluster?

- How close a point is to other points in its assigned cluster (inertia)
- How close that same point is to points in the nearest neighboring cluster (Silhouette Score)

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()
```

```
# 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)

# Fit K-means with the optimal number of clusters
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'])

print("Descriptive Statistics for Each Cluster:")
cluster_stats
```

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

Overall Observations:

- PC1 and PC2 seem to be more effective at differentiating the clusters based on their mean values than PC3

PC1 (Captures the most variance in your original data):

- Separation (mean):
  - PC1 strongly differentiates all three clusters based on their mean values
  - Cluster 0 has a positive mean (2.423), indicating samples here score high on the trait(s) PC1 represents
  - Cluster 1 has a moderately negative mean (-1.347)
  - Cluster 2 has a very strongly negative mean (-8.970), indicating samples in this cluster score very low on trait(s) PC1 represents
- Spread (std):
  - Cluster 0 has an intermediate spread (std = 2.670)
  - Cluster 1 is the most compact (homogeneous) along PC1 (std = 2.241) relative to Cluster 0 and 2
  - Cluster 2 is the most spread out (heterogeneous) along PC1 (std = 3.366)

PC2 (Captures the second most variance):

- Separation (mean):
  - PC2 is also effective at differentiating the clusters, particularly separating Cluster 1 from Clusters 0 and 2.
  - Cluster 1 has a strongly negative mean (-3.703).
  - Cluster 0 (1.315) and Cluster 2 (2.066) both have positive mean PC2 scores, with Cluster 2's mean being slightly higher.
- Spread (std):
  - Cluster 0 is the most compact along PC2 (std = 1.812).
  - Cluster 1 has an intermediate spread (std = 2.310).
  - Cluster 2 is the most spread out along PC2 (std = 2.686).

Cluster 0:

- Patients in this cluster strongly exhibit the characteristics represented by the positive direction of PC1 and moderately by the positive direction of PC2
- They are quite consistent in their PC2 (and PC3) values

Cluster 1:

- Patients are strongly characterized by the negative direction of PC2 and moderately by the negative direction of PC1
- They are quite consistent in their PC1 values (relatively)

Cluster 2:

- Patients are strongly characterized by the negative direction of PC1 and strongly by the strongly positive direction of PC2
- They are quite spread-out in their PC1 and PC2 values

Classification

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 necessary libraries
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
```

```
# Download the test dataset
! curl
https://naicno.github.io/BioNT_Module2_handson/_downloads/041231c291c25343976f8b70db54f23
-o TCGA_InfoWithGrade_scaled.csv
! ls -lh TCGA_InfoWithGrade_scaled.csv
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed

  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
```

```
 29 53487   29 15640    0     0   100k      0 --:--:-- --:--:-- --:--:--  100k
100 53487  100 53487    0     0   342k      0 --:--:-- --:--:-- --:--:--  341k
```

```
-rw-r--r-- 1 runner docker 53K Jun 10 13:52 TCGA_InfoWithGrade_scaled.csv
```

```
# Create a DataFrame from the CSV file
gliomas = pd.read_csv('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
```

```
# Inspect the first few rows of the DataFrame
gliomas.head()
```

```
# Count the number of samples in each grade (Count observations in target variable)
gliomas.groupby('Grade').size()
```

Visualize data distributions

```
# 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()
```

- Scaling binary features would destroy their natural interpretability - coefficients would no longer represent the log-odds change from “absent” (0) to “present” (1)

Binary features encoded as 0/1 have a natural, meaningful interpretation where the coefficient represents the change in log-odds when the feature goes from absent (0) to present (1). If you scale these features (e.g., to have mean 0 and standard deviation 1), a “0” might become -0.85 and a “1” might become +1.23, making the coefficient interpretation much less intuitive. You’d lose the direct interpretation of “how much does the presence of this feature change the odds?” Additionally, binary features are already on a bounded, comparable scale (0 to 1), unlike continuous features that might range from 0 to 100,000, so scaling for convergence purposes is typically unnecessary.

```
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

```
# Initialize the Logistic Regression model
lr = LogisticRegression()
# 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

Additional notes:

- Raw model output of logistic regression is equal to the log-odds
- Why does this matter?
  - Improves interpretability: A one-unit increase in a feature increases the log-odds by the corresponding coefficient
- Homework: Explore mathematics relationship between “raw model output” and “log odds”

```
# Plot Raw model output histogram
sns.histplot(lr.decision_function(X_train))
plt.title('Distribution of Raw model output')
plt.xlabel('Raw model output')
plt.ylabel('Count')
```

The `decision_function()` returns the raw linear combination: $\beta_0 + \beta_1x_1 + \beta_2x_2 + ... + \beta_nx_n$. Since this is just a weighted sum of features, it has no mathematical bounds and can theoretically range from -∞ to +∞. The sigmoid function is then applied to transform these logits into probabilities between 0 and 1.

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

```
# Access prediction probabilities
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)
```

- `predict_proba()` returns a two-column array where Column 0 = P(class=0) = P(LGG) and Column 1 = P(class=1) = P(GBM) for each test patient. These probabilities sum to 1.0 for each patient
- In medical contexts, probability outputs are crucial because they provide confidence levels (e.g., “85% chance of GBM” vs “51% chance of GBM”) that help clinicians make more informed treatment decisions and determine when additional testing might be needed.

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

- If we’re looking at the probability of class 0:
  - Probability near 0.0 = low chance of being class 0 = likely class 1

    - Probability near 1.0 = high chance of being class 0 = confident class 0 prediction
- A U-shaped probability distribution is actually desirable in logistic regression, indicating that the model can confidently distinguish between classes. Most predictions are either very likely class 0 (near 0.0) or very likely class 1 (near 1.0), with few ambiguous cases in between.
- Looking at the histogram, there are noticeably more blue bars (class 0 predictions) near probability 0.0 than orange bars (class 1 predictions) near probability 1.0, indicating the model predicts more instances as belonging to class 0.
- The model makes very confident predictions - most probabilities cluster at the extremes (0.0 and 1.0) with very few uncertain predictions in the middle ranges (0.3-0.7).

Predict the Glioma type of the new dataset

```
# Predict classes for the test set
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 classes

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

```
# Get the list of features
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 and their corresponding features
# `.coef_.` attribute of the fitted LogisticRegression model (`lr`) contains the
coefficients for each feature
coefficients = pd.DataFrame(lr.coef_.T, index=feature_list, columns=['Coefficient'])
# Sort the coefficients
coefficients.sort_values(by='Coefficient', ascending=False)
```

```
# Plot coefficients for each feature
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())
```

- In logistic regression,
  - positive coefficients (like GRIN2A ~+1.0) increase the log-odds and therefore the probability of the positive class when the feature value increases
  - Negative coefficients (like IDH1 ~-3.0) decrease the log-odds and probability of the positive class
- The direction of the bar indicates whether the feature pushes predictions toward or away from the positive class.

Evaluation of the model performance

- Confusion matrix

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()
```

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

```
# Download the test dataset
! curl
https://naicno.github.io/BioNT_Module2_handson/_downloads/d072ac9ebbb200e56a8125b33b88765
-o TCGA_InfoWithGrade.csv
! ls -lh TCGA_InfoWithGrade.csv
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed

  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
```

```
  0 86432    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
100 86432  100 86432    0     0   462k      0 --:--:-- --:--:-- --:--:--  461k
```

```
-rw-r--r-- 1 runner docker 85K Jun 10 13:52 TCGA_InfoWithGrade.csv
```

```
# Read the dataset to get the glioma dataframe
gliomas = pd.read_csv('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
```

```
# Inspect the first few rows of the DataFrame
gliomas.head()
```

```
# Quick summary statistics of the DataFrame
gliomas.describe()
```

Analyze the Target Variable

```
# Check data types of the target variable 'Grade'
gliomas["Grade"].dtype
```

```
# Get the counts of each grade in the 'Grade' column
gliomas["Grade"].value_counts()
```

```
# Get the fractions of each grade in the 'Grade' column
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

- Note that np.nan values are inserted into the original dataset specifically to demonstrate techniques that help handel these datasets

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

Delete rows with missing values in target variable

```
# Remove rows with missing values in the 'Grade' column
gliomas.dropna(subset=["Grade"], axis=0, inplace=True)
```

```
# % 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
```

```
# 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()
```

```
# Drop the 'Race' column
gliomas.drop(columns=["Race"], inplace=True)
```

When a category within a feature has very few samples (e.g., “Race”), a model might learn patterns from these specific few instances that are not generalizable to the wider population or new data. It essentially “memorizes” these rare cases and their associated outcomes, which can lead to poor performance on unseen data.

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 and create kf object
kf = KFold(n_splits=5, shuffle=True, random_state=1111)

# Create splits
## The `split` method of the KFold object will yield indices for training and
validation sets
splits = kf.split(gliomas.drop("Grade", axis=1))

# Print the indices of the training and validation sets for each fold
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"]


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": [],
}

# Initialize scaler
scaler = StandardScaler()
fold = 1

for train_index, val_index in splits:
    # Setup the training and validation data
    X_train, y_train = X.iloc[train_index], y.iloc[train_index]
    X_val, y_val = X.iloc[val_index], y.iloc[val_index]

    # Create copies to avoid modifying original data
    X_train_scaled = X_train.copy()
    X_val_scaled = X_val.copy()

    # Scale only the Age_at_diagnosis column
    X_train_scaled['Age_at_diagnosis'] =
scaler.fit_transform(X_train[['Age_at_diagnosis']]).flatten()
    X_val_scaled['Age_at_diagnosis'] =
scaler.transform(X_val[['Age_at_diagnosis']]).flatten()

    # Initialize the Logistic Regression model with cross-validation
    lr_cv = LogisticRegression()

    # Use the SCALED data for training and prediction
    lr_cv.fit(X_train_scaled, y_train)  # ← Now using scaled data
    predictions = lr_cv.predict(X_val_scaled)  # ← Now using scaled data

    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
```

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

Additional notes (regarding parameters used in following cell):

- `C` (Regularization Strength):
  - Controls how much to penalize complex models
  - Smaller values = more regularization (simpler model)
- Loss function (penalty):
  - `l2`: Ridge regularization (shrinks coefficients toward zero)
  - `l1`: Lasso regularization (can set coefficients exactly to zero, good for feature selection)
- solver:
  - `liblinear`: Good for small datasets, works with both L1 and L2
  - `saga`: Good for large datasets, works with both L1 and L2
  - `ElasticNet`: Combines `L1` and `L2` benefits (address multicollinearity handle groups of correlated genes)
  -
- `max_iter:`
  - Maximum number of iterations for the algorithm to converge
  - Increase if you get convergence warnings

```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
```

```
# 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()
```

Unsupervised Learning

Q & A session: Principal component analysis (PCA) and K-means clustering

Instructor note

- Participants were given 45-60 minutes to
  - go through the Jupter notebook and
  - select correct answers for the questions
- Instructor narrate the answers and reasoning after the self-study time
  - time: 30 minutes

Dimensionality of the datasets

❓ Questions

Question 1:

You have a dataset with 50 columns (drug compounds) and 25 rows (patients). Is this dataset considered high-dimensional?

- A) No, because 50 columns is not a large number
- B) No, because the dataset is too small overall
- C) Yes, because there are more features (50) than samples (25)
- D) Yes, because 25 patients is insufficient for any analysis

Question 2:

Why is your dataset with 50 drug compounds and 25 patients considered high-dimensional?

- A) Because 50 is the threshold for high-dimensionality
- B) Because the number of features (50) significantly exceeds the number of samples (25)
- C) Because drug compound data is inherently high-dimensional
- D) Because 25 patients represent too many medical conditions

Question 3:

In your dataset, the ratio of features to samples is 2:1 (50 features to 25 samples). This high-dimensional characteristic is most likely to cause:

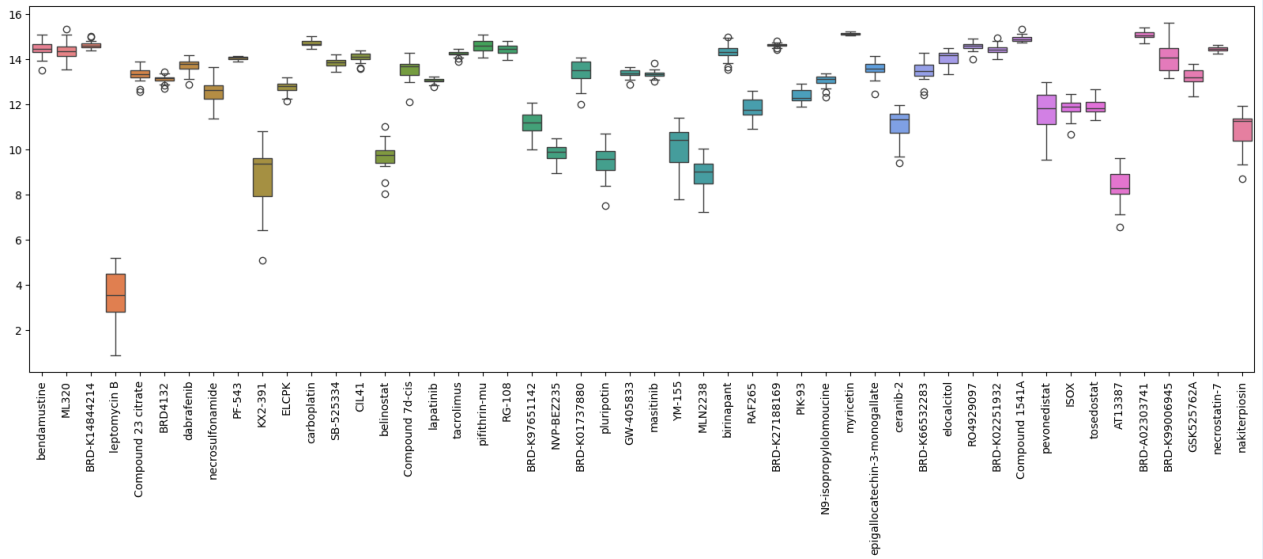
- A) Faster model training due to more information per patient
- B) Better generalization because of the rich feature space
- C) The “curse of dimensionality” and potential overfitting issues
- D) Automatic feature selection by machine learning algorithms

Standardization

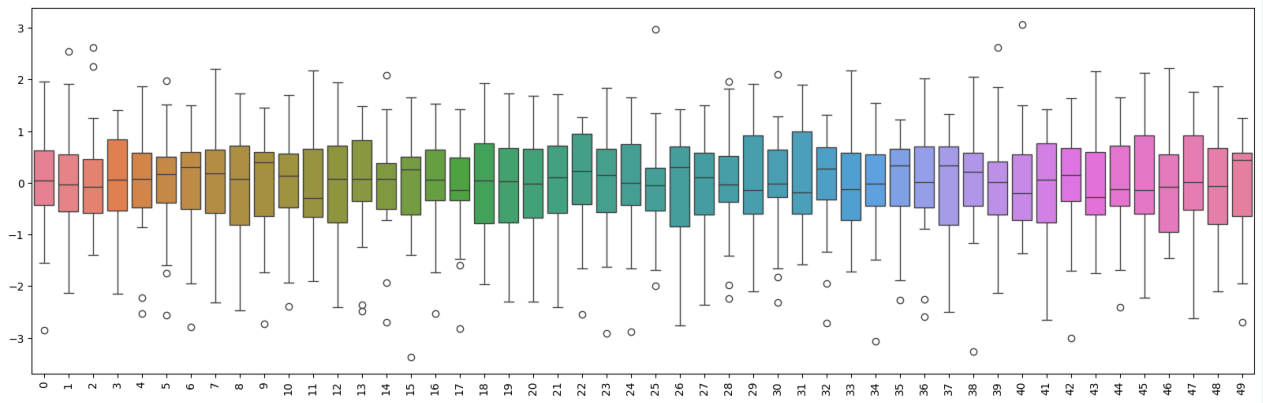
❓ Questions

Before (A) / After standardization plots (B):

A.



B.



Question 1: Scale Uniformity After Standardization:

Comparing the original drug sensitivity scores (A) with the standardized versions (B), what is the most important change for PCA analysis?

- A) The standardized data has fewer outliers than the original data
- B) All drug compounds now have similar scales (roughly -3 to +3) instead of vastly different ranges
- C) The standardized data shows stronger correlations between compounds
- D) The standardized data has reduced the total number of features

Question 2: PCA Component Interpretation:

After standardization, all drug compounds now have approximately the same variance. How will this affect your PCA results compared to using the original unstandardized data?

- A) PC1 will still be dominated by the originally high-variance compounds like leptoporin B
- B) PCA components will now reflect actual biological/chemical relationships rather than just scale differences
- C) PCA will find fewer meaningful components due to the uniform scaling
- D) The explained variance percentages will be identical to the unstandardized analysis

Question 3: Practical PCA Decision:

You're about to perform PCA for drug discovery research. Based on your standardization comparison, which approach would give you more interpretable principal components for identifying drug response patterns?

- A) Use original data because it preserves the natural measurement scales of each drug
- B) Use standardized data because it allows PCA to focus on underlying biological patterns rather than measurement scale artifacts
- C) It doesn't matter - PCA will automatically handle scale differences

- D) Use original data because standardization removes important variance information

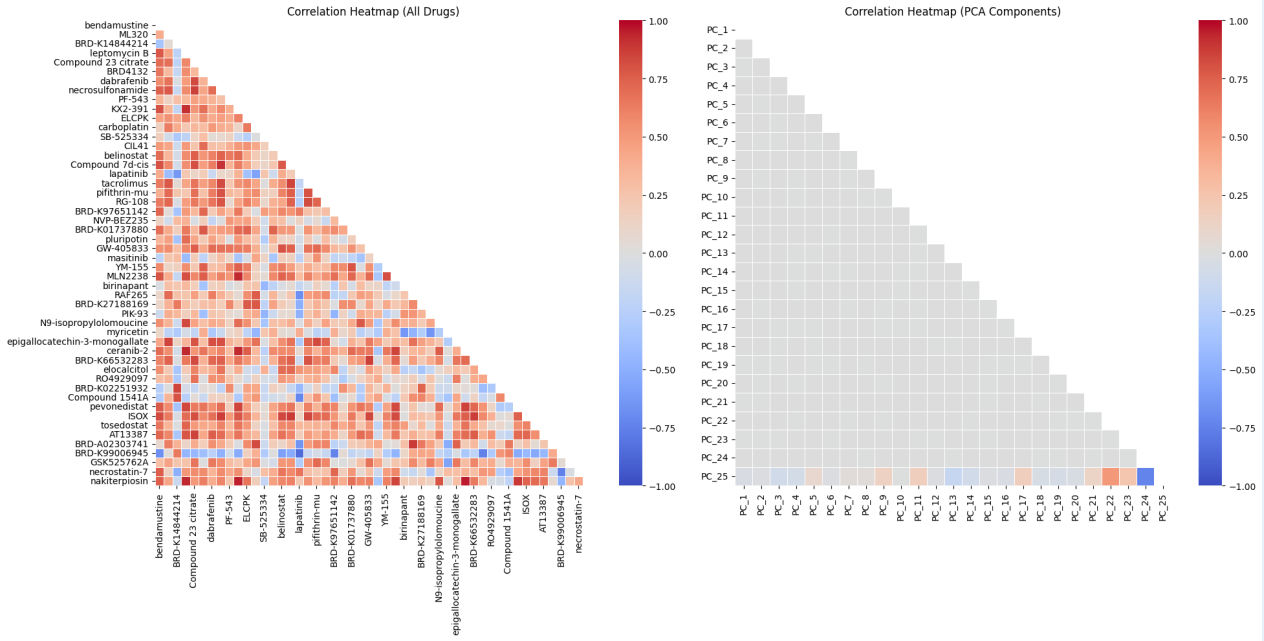
Question 4: Variance Landscape:

Looking at your standardized data plot where all compounds now have similar variance, what does this mean for PCA's search for "maximum variance directions"?

- A) PCA will no longer work because there's no variance to capture
- B) PCA will now find directions based on correlations and biological patterns rather than being biased by high-variance features
- C) PCA will randomly select components since all variances are equal
- D) PCA will focus only on outlier detection

Apply PCA transformation

❓ Questions



Question 1:

Comparing the two heatmaps, what is the most significant difference between the correlation patterns of the original drug compounds (left) and the PCA components (right)?

- A) The PCA components show stronger correlations than the original compounds
- B) The original compounds are uncorrelated while PCA components are highly correlated
- C) The original compounds show various correlation patterns, while PCA components are essentially uncorrelated (orthogonal)
- D) Both heatmaps show identical correlation structures

Question 2:

In your PCA components heatmap (right), most correlations appear to be near zero (gray/white). This pattern demonstrates which fundamental property of PCA?

- A) PCA randomly shuffles the original correlations
- B) PCA creates principal components that are orthogonal (perpendicular) to each other, eliminating correlation
- C) PCA amplifies the strongest correlations from the original data
- D) PCA preserves all original correlation relationships between features

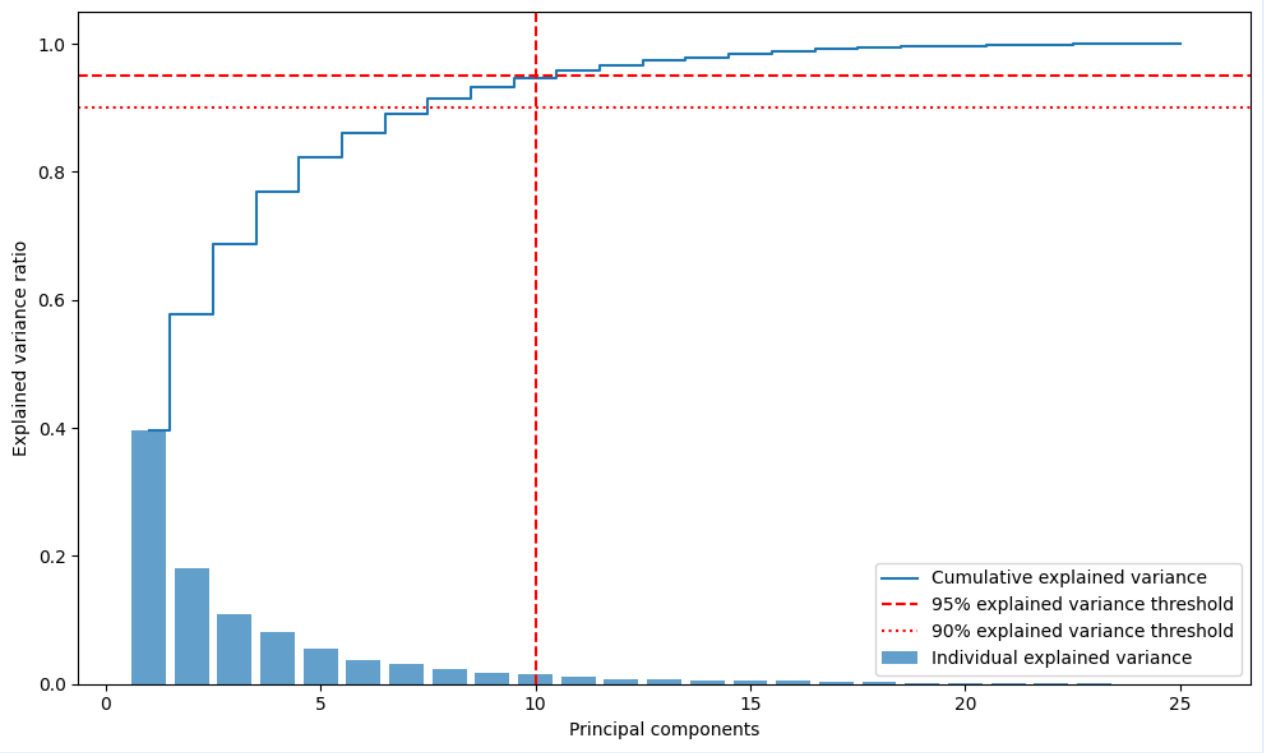
Question 3:

The original drug compounds heatmap shows clusters of correlated compounds (red blocks), while the PCA heatmap is predominantly gray. What does this tell you about what PCA has accomplished?

- A) PCA has lost important information about drug relationships
- B) PCA has transformed the correlated original features into a new set of uncorrelated components that still capture the data's variance
- C) PCA has created random noise instead of meaningful components
- D) PCA has made the data analysis more complicated

Explained variance ratios

❓ Questions



Question 1:

Your plot shows that 10 principal components explain 95% of the total variance (red dashed line). For a machine learning application, which approach would be most appropriate?

- A) Always use all 25 components to avoid losing any information
- B) Use exactly 10 components because they reach the 95% threshold
- C) Use only PC1 since it explains the most variance (40%)
- D) Choose the number of components based on your specific analysis goals and computational constraints

Question 2:

Examining the individual explained variance bars (blue), there's a sharp drop after PC1 and PC2, then the contributions become much smaller. This pattern suggests:

- A) The first two components capture the main data structure, while next ffew components may represent minor patterns and last few components represent noise
- B) Only PC1 and PC2 are mathematically valid
- C) Components 3-25 contain no useful information
- D) This indicates an error in the PCA calculation

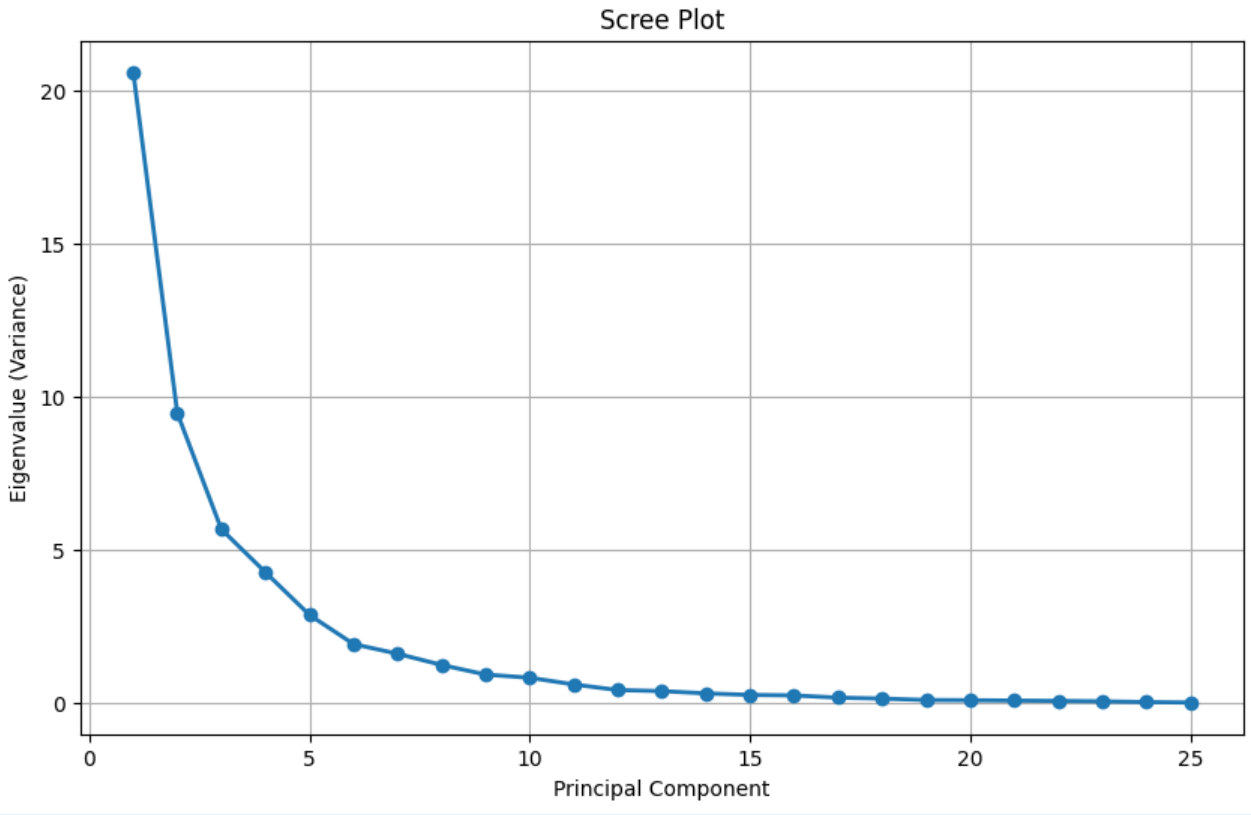
Question 3:

In the context of your drug sensitivity dataset, what does PC1's 40% explained variance represent biologically?

- A) 40% of the drugs are important for the analysis
- B) The first principal component captures a major pattern of drug response variation across patients that accounts for 40% of the total variability
- C) 40% of the patients respond similarly to drugs
- D) 60% of the data is noise and should be discarded

Generate scree plot

❓ Questions



Question 1:

In your scree plot, the eigenvalues drop sharply from PC1 (~21) to PC2 (~9) to PC3 (~6), then gradually flatten out after PC6-7. What is the logic behind choosing the number of components at the “elbow” point where the curve starts to flatten?

- A) Components after the elbow contain no mathematical information
- B) Components before the elbow capture major data patterns, while those after the elbow likely represent noise or minor variations
- C) The elbow point is randomly determined and has no statistical meaning
- D) You should always choose exactly at the steepest drop point

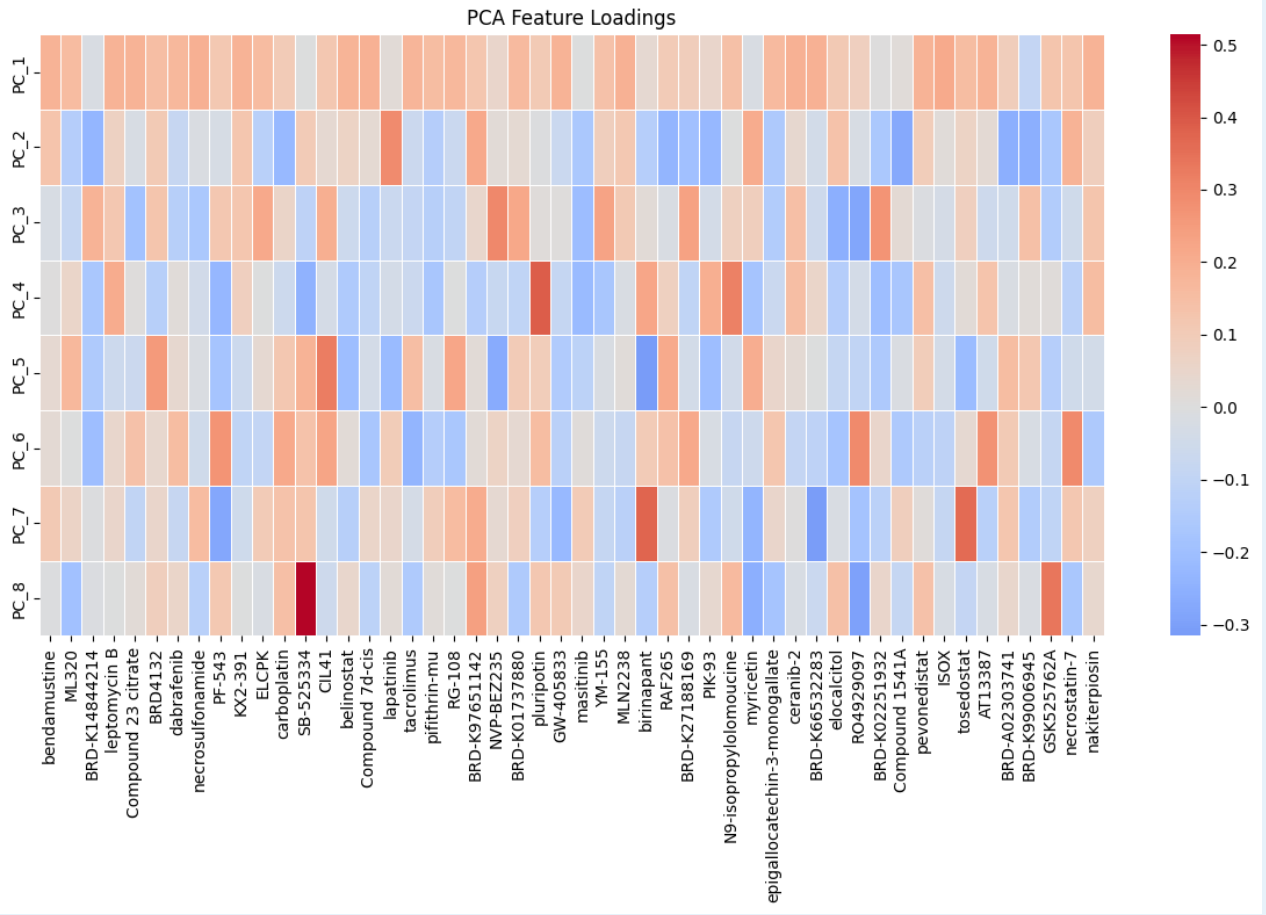
Question 2:

Looking at your scree plot, why do we typically avoid including principal components from the flat region (PC8 onwards, where eigenvalues are close to 0)?

- A) These components are mathematically incorrect and will cause errors
- B) These components represent very small amounts of variance and may capture noise rather than meaningful signal
- C) These components take too much computational time to calculate
- D) These components are always highly correlated with the first few components

Interpretation and Analysis

❓ Questions



Question 1:

In this PCA loadings heatmap, some drug compounds show dark red or dark blue colors while others appear white/gray. What do these color intensities tell you about each compound's contribution to the principal components?

- A) Dark colors indicate drugs that are more effective therapeutically
- B) Dark red/blue indicate high absolute loading values, meaning these compounds strongly contribute to defining that principal component
- C) White/gray areas indicate missing data for those drug compounds
- D) Color intensity represents the correlation between different drugs

Question 2:

Looking at PC1 (top row), you can see both red (positive) and blue (negative) loadings for different drug compounds. What does this pattern of positive and negative loadings indicate?

- A) Positive loadings are "good" drugs and negative loadings are "bad" drugs
- B) This indicates an error in the PCA calculation since all loadings should be positive
- C) Compounds with positive loadings move in the same direction as PC1, while those with negative loadings move in the opposite direction
- D) Positive and negative loadings cancel each other out, making PC1 meaningless

Question 3:

To understand what PC1 represents biologically in your drug sensitivity study, which compounds should you focus on for interpretation?

- A) Only the compounds with positive loadings (red colors)
- B) Only the compounds with negative loadings (blue colors)
- C) The compounds with the highest absolute loadings (darker shades of red and blue), regardless of sign
- D) The compounds with near-zero loadings (white/gray) because they're most stable

Question 4:

If you wanted to name or characterize what biological pathway PC1 represents, how would you use the loading information?

- A) Look up the biological functions of compounds with the highest absolute loadings in PC1
- B) Only consider the single compound with the highest positive loading
- C) Average all the loading values to get a general interpretation
- D) Focus on compounds that appear in multiple principal components

Classification

Q & A session: Logistic regression

Instructor note

- Participants were given 30 minutes to
  - go through the Jupter notebook and
  - select correct answers for the questions
- Instructor narrate the answers and reasoning after the self-study time
  - time: 30 minutes

Understanding biological context (ML-use case)

❓ Questions

Question 1:

What type of machine learning problem is glioma grading/classification as described in this tutorial?

- A) Unsupervised clustering problem
- B) Binary classification problem
- C) Multi-class classification problem
- D) Regression problem

Question 2:

In this study, how is the target variable encoded?

- A) LGG = 1, GBM = 0
- B) LGG = 0, GBM = 1
- C) Both are encoded as 1
- D) Text labels are used without numeric encoding Answer: - B) LGG = 0, GBM = 1 Explanation: The target variable encoding is explicitly stated as 0 = “LGG” and 1 = “GBM”.

Question 3:

What types of features are being used in this logistic regression model?

- A) Only genetic mutation data
- B) Only clinical features
- C) 20 most frequently mutated genes plus 3 clinical features
- D) All available genetic and clinical data

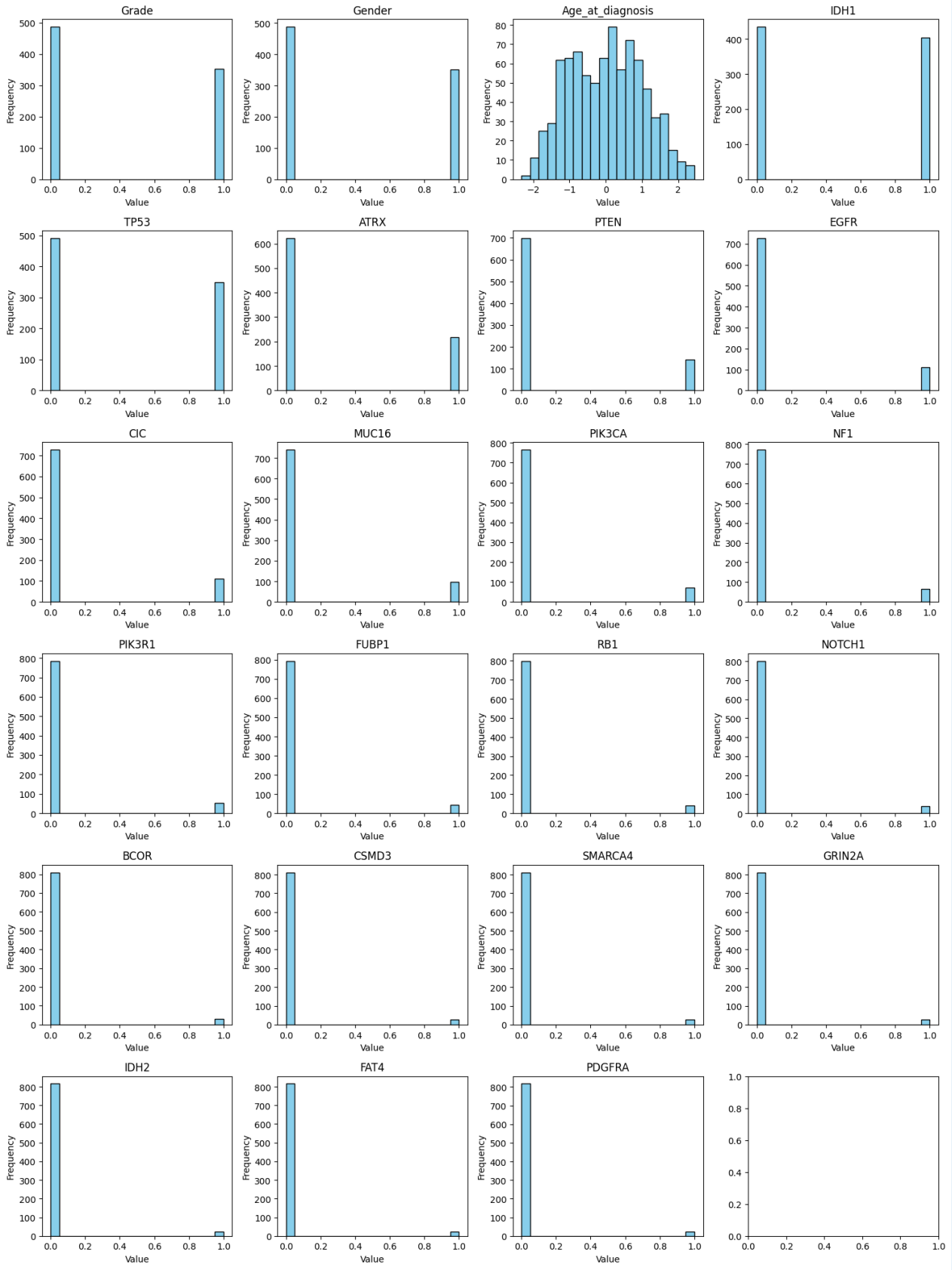
Question 4:

Why is accurate glioma grading/classification clinically important?

- A) It determines the research funding allocation
- B) Different grades require different treatment approaches and have different prognoses
- C) It’s only important for statistical purposes
- D) It helps organize hospital records

Visualize data distributions

❓ Questions



Question 1:

Binary features (keeping them as 0/1) while scaling Age_at_diagnosis. What is the primary reason why scaling binary features is generally NOT recommended in logistic regression?

- A) Binary features are too simple to benefit from scaling
- B) Scaling binary features would destroy their natural interpretability - coefficients would no longer represent the change from “absent” (0) to “present” (1)
- C) Binary features automatically have equal variance, so scaling is unnecessary

- D) Logistic regression algorithms cannot handle scaled binary features

Question 2:

Given the heavy imbalance in most binary features (with most values being 0 and small number of observations with 1s'), what potential issue might this create during logistic regression training?

- A) The model will converge faster due to the simplicity of the data
- B) The model may have difficulty learning meaningful patterns from rare events (1s) and might be biased toward predicting the majority class
- C) The imbalanced features will automatically be weighted equally by the algorithm

Question 3:

When interpreting logistic regression coefficients for the heavily imbalanced binary features shown in these plots, what should you be particularly cautious about?

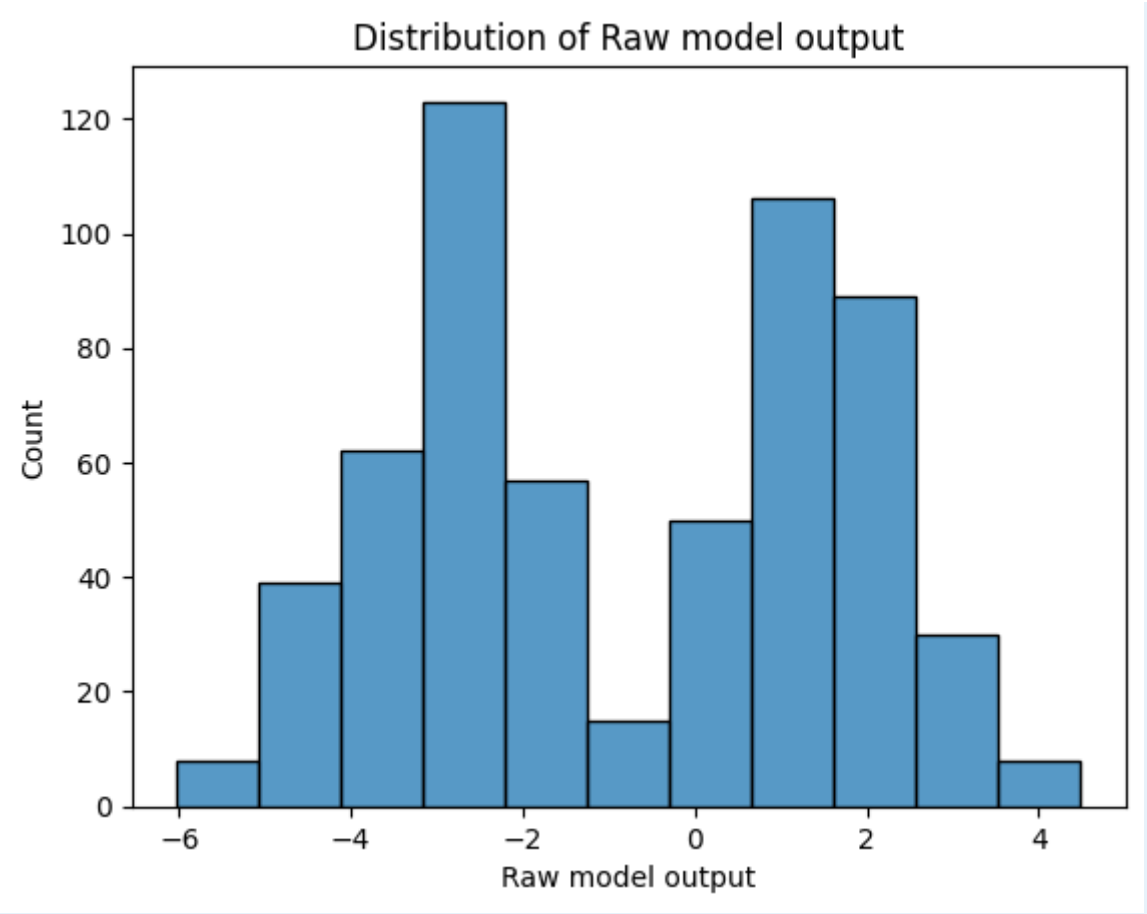
- A) Coefficients for rare events may be unstable and can lead to poor generalization
- B) Coefficients will be automatically adjusted by the algorithm to account for imbalance
- C) Imbalanced features always produce more reliable coefficient estimates
- D) The scaling of Age_at_diagnosis will make other coefficients uninterpretable

Split original dataset

❓ Questions

Question 1:

In the `train_test_split` code, `test_size=0.3` means 30% of data goes to testing. For a medical dataset like glioma classification, what is the primary consideration when choosing this split ratio?



- A) Larger test sets always give better model performance
- B) Balancing reliable performance evaluation with sufficient training data, especially important given limited medical data availability
- C) Test size should always be exactly 30% regardless of dataset characteristics
- D) Smaller test sets are always preferred to maximize training data

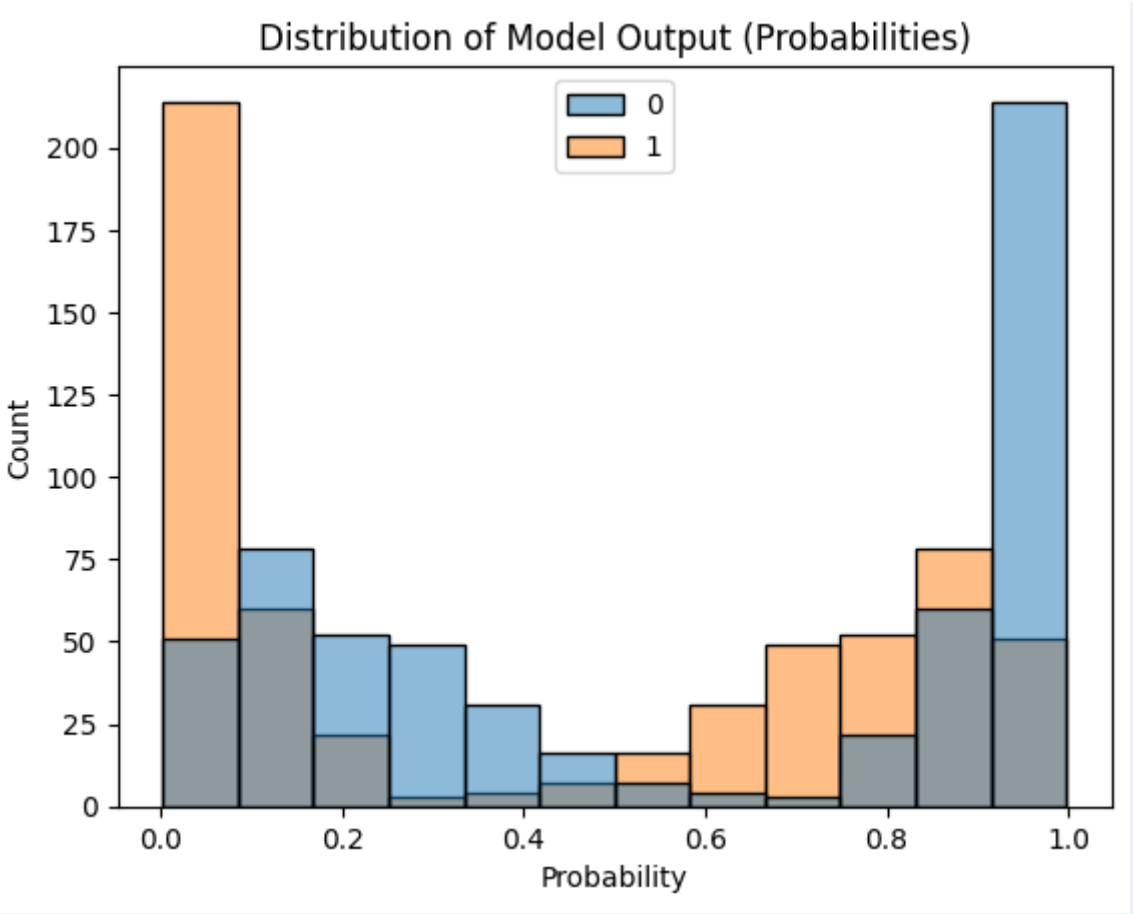
Question 2:

In the `train_test_split` code, why is the `stratify=gliomas["Grade"]` parameter crucial in this glioma classification problem?

- A) It randomly shuffles the data for better performance
- B) It ensures both training and test sets have proportional representation of LGG and GBM cases
- C) It sorts the data by grade for easier processing
- D) It removes outliers from the dataset

The Model Output (Probability)

❓ Questions



Question 1:

The histogram shows a distinctive U-shaped distribution of predicted probabilities, with many predictions clustered near 0.0 and 1.0, and fewer predictions in the middle range (0.3-0.7). What does this pattern indicate about the model's behavior?

- A) The model is making unreliable predictions
- B) The model is well-calibrated and confident in most of its predictions, clearly separating the two classes
- C) The model has failed to converge properly during training
- D) The sigmoid transformation is not working correctly

Question 2: What does lr.predict_proba(X_test) output, and why are probability predictions particularly valuable in medical diagnosis like glioma classification?

- A) It outputs only the predicted class labels (0 for LGG, 1 for GBM)
- B) It outputs probability estimates for each class (P(LGG) and P(GBM)) for each patient, allowing clinicians to assess prediction confidence
- C) It outputs the raw coefficient values for each gene and clinical feature
- D) It outputs the training accuracy of the model

Predict test-datasaets

❓ Questions

Question 1:

What is the key difference between `lr.predict(X_test)` and lr.predict_proba(X_test) in this glioma classification model?

- A) `predict()` gives probabilities for each class, while `predict_proba()` gives final class labels
- B) `predict()` gives final class decisions (0 for LGG, 1 for GBM) by applying a 0.5 threshold to probabilities, while `predict_proba()` gives the actual probability values
- C) `predict()` is more accurate than `predict_proba()`
- D) There is no difference between the two methods

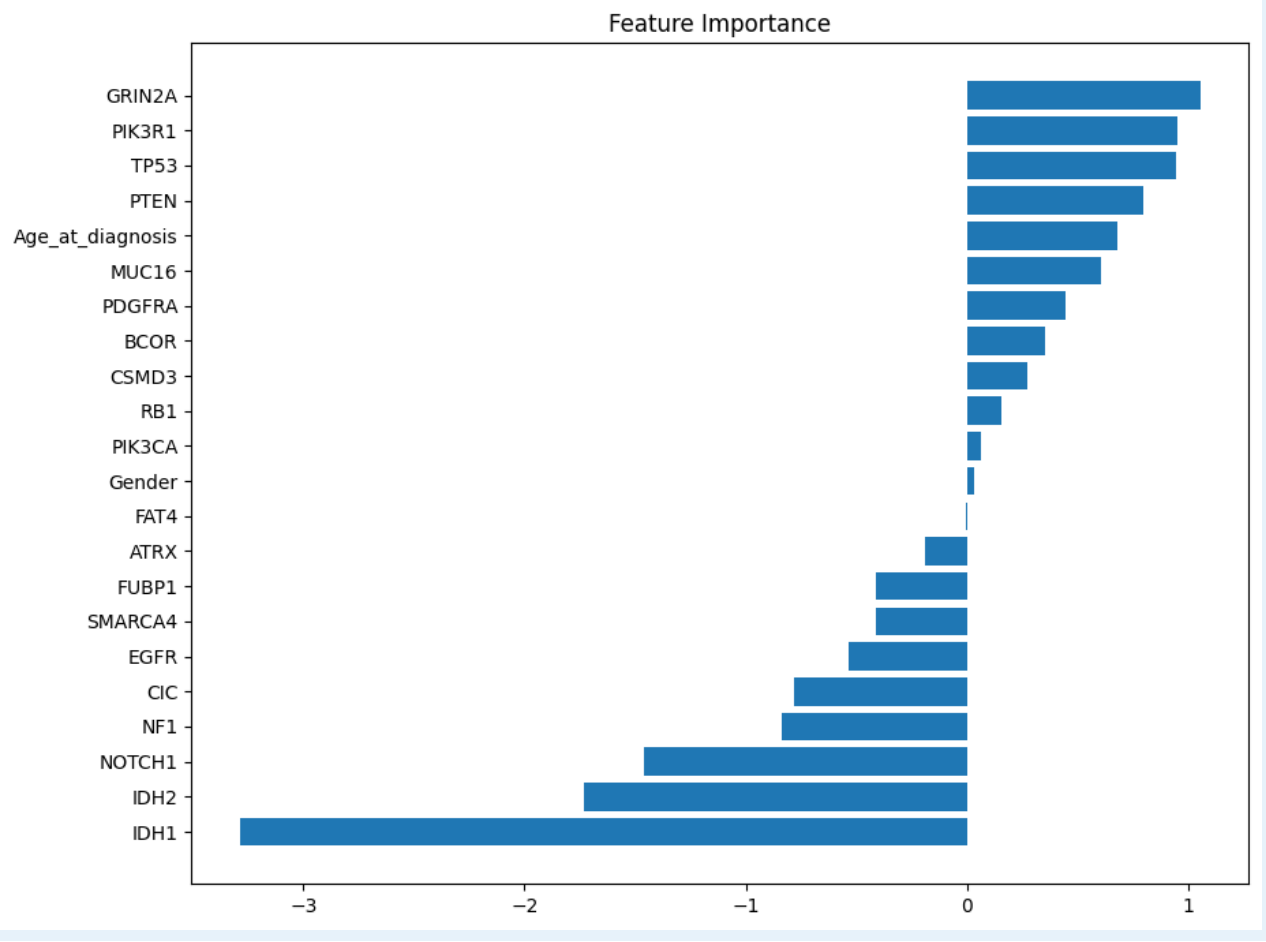
Examine and understand the importance of features in predicting classes

❓ Questions

Question 1 (Coefficient Sign Interpretation):

In this glioma classification model, what does a positive coefficient in `lr.coef_` indicate for a specific gene or clinical feature?

- A) The feature has no effect on glioma classification
- B) The feature increases the likelihood of predicting GBM (class 1) when present or higher in value
- C) The feature increases the likelihood of predicting LGG (class 0) when present or higher in value
- D) The feature should be removed from the model



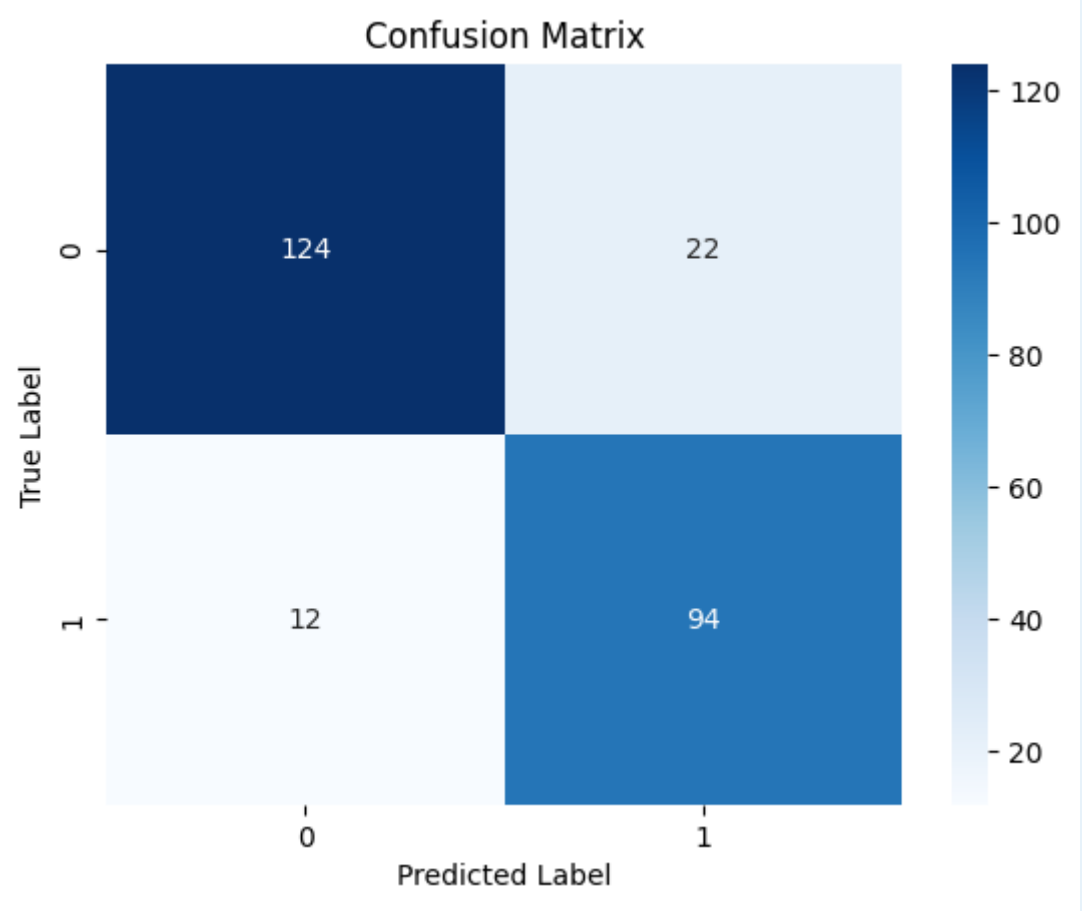
Question 2 (Coefficient Sign Interpretation):

Looking at the feature importance plot, what can you conclude about the features GRIN2A (rightmost bar) and IDH1 (leftmost bar) in terms of their effect on the predicted outcome?

- A) GRIN2A decreases the probability of the positive class, while IDH1 increases it
- B) GRIN2A increases the probability of the positive class, while IDH1 decreases it
- C) Both features have the same effect but different magnitudes
- D) The sign of the coefficient doesn't matter, only the magnitude

Evaluation of the model performance

❓ Questions



Question 1:

Based on the confusion matrix shown, what are the True Positives, False Positives, True Negatives, and False Negatives for this glioma classification model?

- A) TP=124, FP=22, TN=94, FN=12
- B) TP=94, FP=22, TN=124, FN=12
- C) TP=22, FP=94, TN=12, FN=124
- D) TP=94, FP=12, TN=124, FN=22

Complete Machine Learning Workflow

Machine Learning Workflow

Instructor note

- Participants were given 60 minutes to
  - go through the Jupter notebook and
  - select correct answers for the questions
- Instructor narrate the answers and reasoning after the self-study time
  - time: 45 minutes

Data exploration

❓ Questions

Question 1:

When dealing with gene mutation features where >95% of samples are wild-type (0), what is the most important consideration?

- A) Apply standard scaling (z-score normalization) to make features comparable
- B) Use log transformation to reduce skewness in the distribution
- C) Consider the impact on model training - rare events may be difficult to learn
- D) Convert to categorical variables using one-hot encoding

Question 2:

Your dataset contains age (continuous, 20-80 range), gender (binary 0/1), and gene mutations (binary 0/1). What normalization strategy is most appropriate?

- A) Apply Min-Max scaling to all features to get 0-1 range
- B) Apply Z-score normalization to all features for zero mean, unit variance
- C) Scale only the continuous features (age), leave binary features unchanged
- D) Apply log transformation to all features to handle skewness

Question 3:

Why is feature scaling for `Age_at_diagnosis` is required?

- A) To convert the `Age_at_diagnosis` distribution into a perfect Gaussian (normal) distribution.
- B) To reduce the number of unique values in `Age_at_diagnosis`, thereby simplifying the model.

- C) To ensure that `Age_at_diagnosis` values are transformed to be either 0 or 1, matching the gene features.
- D) To prevent `Age_at_diagnosis` from disproportionately influencing the model's parameter estimation due to its larger numerical range compared to the binary (0/1) features.

Question 4:

In this glioma classification dataset, what should be your primary concern regarding the rare gene mutations?

- A) The computational cost will be too high with so many zero values
- B) Rare mutations might be the most clinically important but hardest to detect
- C) Binary features don't need any preprocessing in machine learning
- D) The dataset is too small and needs data augmentation

Question 5:

You're analyzing a dataset with 10,000 patients where a particular gene mutation occurs in only 50 patients (0.5% prevalence). When should you be most concerned about this feature imbalance?

- A) Always - any feature with <5% prevalence will bias the logistic regression model
- B) Never - logistic regression inherently handles sparse features well
- C) Only when the 50 mutation carriers don't provide sufficient statistical power to reliably estimate the gene's effect
- D) Only when the mutation is randomly distributed and not associated with the outcome

Question 6:

In disease genomics, why might completely removing very rare genetic variants (occurring in <1% of samples) be problematic from a biological perspective?

- A) Rare variants always have larger effect sizes than common variants
- B) Some rare variants may be clinically actionable, even if statistically underpowered in current sample
- C) Removing sparse features will always improve model generalization
- D) Feature selection should only be based on statistical criteria, not biological knowledge

Missing data handling

❓ Questions

Question 1:

Your target variable (Grade) has missing values in 0.119% of samples. What is the most appropriate approach?

- A) Impute the missing grades using the mode (most frequent class)
- B) Use a sophisticated imputation method like KNN to predict missing grades
- C) Remove these samples entirely from both training and testing datasets
- D) Replace missing grades with a third category “Unknown” and make it a 3-class problem

Question 2:

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

- A) This code drops columns that have more than 5% missing values using a 95% completeness threshold (Drop ATRX_xNA - 25% missing and IDH1_xNA - 80% missing, while Keeping All other features (0.119% or 0% missing)
- B) Only Drop IDH1_xNA
- C) Drop all columns with missing values
- D) Keep columns with 0% missing values

Question 3:

When should this column-dropping step be performed in your ML pipeline?

- A) Before removing samples (rows) with missing target variables
- B) After removing samples with missing target variables but before train/test split
- C) After train/test split but before feature scaling
- D) After model training to remove unimportant features

Train-test and Standerdisation

❓ Questions

```
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"],
)

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']])
```

Question 1:

Why do we use `fit_transform()` on training data but only `transform()` on test data?

- A) `fit_transform()` is faster than `transform()` for larger datasets
- B) To prevent data leakage by ensuring scaling parameters come only from training data
- C) `transform()` automatically applies different scaling to test data for better performance
- D) It's a coding convention but doesn't impact model performance

Question 2:

What would happen if you calculated scaling parameters (mean and standard deviation) using the entire dataset before splitting?

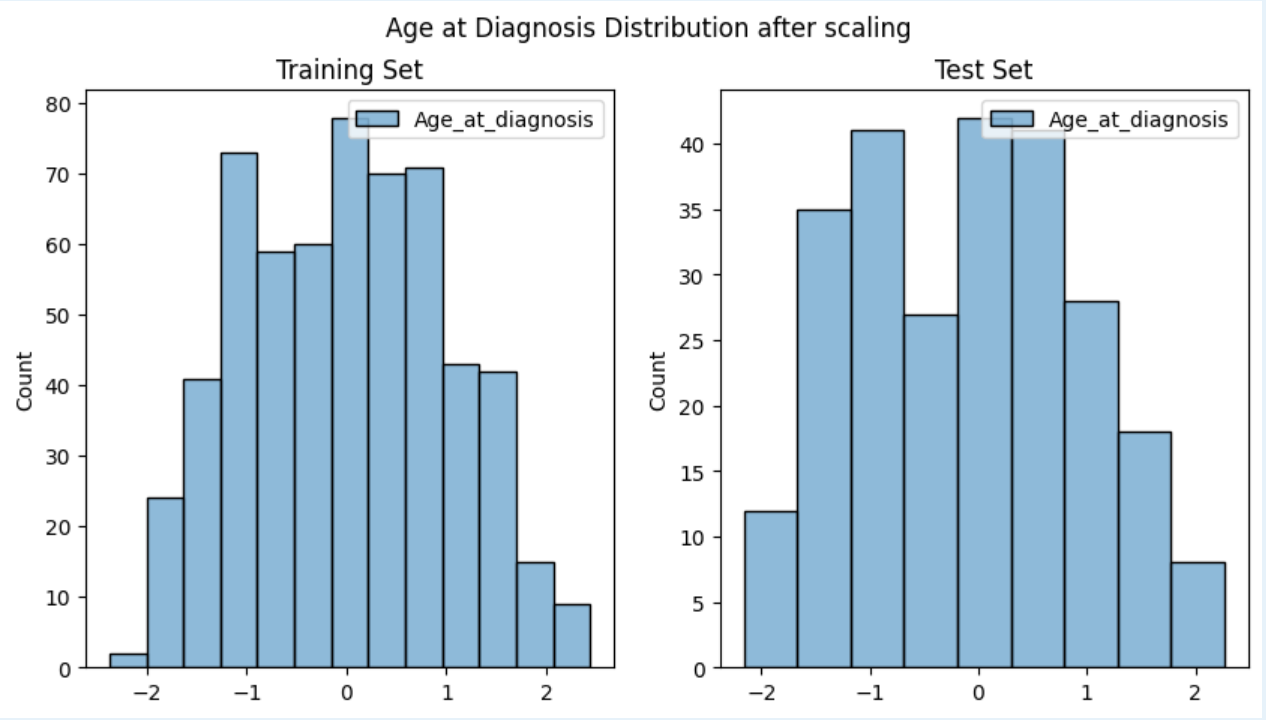
- A) The model would perform better due to more stable scaling parameters
- B) It would create data leakage because test set statistics influence training preprocessing
- C) Nothing significant - the difference in scaling parameters would be minimal
- D) The model would be more generalizable to new data

Question 3:

If a new patient has age = 85 years (outside the training age range of 20-80), what should happen during prediction?

- A) Reject the prediction because the age is out of range
- B) Retrain the scaler including this new data point
- C) Apply the same training scaler transformation, even if it results in an extreme scaled value
- D) Use a different scaling method specifically for this outlier

Question 4:



What is the most important reason for a machine learning practitioner to perform such a visual check after splitting the data?

- A) To ensure that no data points were lost during the train_test_split operation.
- B) To confirm that the Age_at_diagnosis feature has been transformed to a normal distribution in both sets.
- C) To verify that the feature distributions are reasonably similar between the training and test sets, which helps ensure that the test set provides a fair and representative evaluation of the model's performance.
- D) To decide if the Age_at_diagnosis feature should be used as the target variable instead of "Grade".

Corss-validation

❓ Questions

Question 1:

You split your dataset into 70% train/30% test, getting a test precison of 87% (recall 85%). What's the main limitation of this single performance estimate?

- A) precison of 87% (recall 85%) is too low for medical applications
- B) The estimate could vary with different random splits of the same data
- C) 30% test size is too large and wastes training data
- D) precison and recall are the wrong metric for binary classification problems

Question 2:

Say that Your single test set has 252 samples (30% of 840). If you want to evaluate model performance on rare glioma subtypes that represent 5% of cases, how many samples would you have?

- A) About 42 samples - sufficient for reliable performance estimation
- B) About 13 samples - too few for meaningful statistical conclusions
- C) About 126 samples - more than adequate for analysis
- D) The number doesn't matter if the model is well-trained

Question 3:

A hospital wants to deploy your glioma classifier but asks: "How confident are you that this precison of 87% (recall 85%) will hold for our patient population?" With only a single holdout test, what's your most honest answer?

- A) "Very confident - precison of 87% (recall 85%) are true values since we used proper train/test split"
- B) "Moderately confident - the precison and recall could realistically range from 80-94% based on this single test"
- C) "Cannot provide confidence bounds - need cross-validation or multiple test sets for reliability estimates"
- D) "Completely confident - precison and recall doesn't vary between hospitals"

Corss-validation: Code

❓ Questions

Question 1:

(A)

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

# Initialize scaler
scaler = StandardScaler()
fold = 1

lr_cv = LogisticRegression()
scaler = StandardScaler()

for train_index, val_index in splits:
    ...
    # Initialize the Logistic Regression model with cross-validation

    X_train_scaled['Age_at_diagnosis'] =
scaler.fit_transform(X_train[['Age_at_diagnosis']]).flatten()
    X_val_scaled['Age_at_diagnosis'] =
scaler.transform(X_val[['Age_at_diagnosis']]).flatten()

    # Use the SCALED data for training and prediction
    lr_cv.fit(X_train_scaled, y_train)  # ← Now using scaled data
    predictions = lr_cv.predict(X_val_scale- D)  # ← Now using scaled data
```

(B)

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

# Initialize scaler
scaler = StandardScaler()
fold = 1

for train_index, val_index in splits:
    ...
    # Initialize the Logistic Regression model with cross-validation
    lr_cv = LogisticRegression()

    X_train_scaled['Age_at_diagnosis'] =
scaler.fit_transform(X_train[['Age_at_diagnosis']]).flatten()
    X_val_scaled['Age_at_diagnosis'] =
scaler.transform(X_val[['Age_at_diagnosis']]).flatten()

    # Use the SCALED data for training and prediction
    lr_cv.fit(X_train_scaled, y_train)  # ← Now using scaled data
    predictions = lr_cv.predict(X_val_scale- D)  # ← Now using scaled data
```

In this A code-block, the same `lr_cv` object is used across all 5 folds. What potential issue could this create?

- A) Each fold builds upon the previous fold's learned parameters, creating data leakage
- B) The model's internal state gets reset automatically, so there's no issue
- C) Memory usage increases exponentially with each fold
- D) The model converges faster in later folds due to better initialization

Question 2:

The code declares `scaler = StandardScaler()` before the loop and reuses it (above A & B code-blocks). What happens when you call `fit_transform()` on the same scaler object multiple times?

- A) It accumulates statistics across all folds, causing data leakage
- B) It overwrites previous statistics with new fold's statistics - no leakage
- C) It averages statistics across folds for more stable scaling
- D) It causes an error because you can only fit once

Question 3:

This code uses KFold instead of StratifiedKFold. In a medical dataset where GBM (aggressive cancer) represents 40% of cases, what could go wrong?

- A) Some folds might have 60% GBM while others have 20%, creating inconsistent evaluation conditions
- B) The total number of samples evaluated will be different across folds
- C) Cross-validation will take significantly longer to complete
- D) The precision and recall calculations will become invalid

Hyperparameter tuning

❓ Questions

Question 1:

You train a logistic regression model for glioma classification using default scikit-learn parameters and achieve 78% F1-score. After hyperparameter tuning with GridSearchCV, you achieve 85% F1-score. What does this improvement primarily demonstrate?

- A) Default parameters are intentionally set to poor values to encourage tuning
- B) Machine learning algorithms need parameter optimization to match specific dataset characteristics
- C) Hyperparameter tuning always guarantees at least 7% improvement in any metric
- D) The original 78% score was due to a coding error in the implementation

Question 2:

```python
# original grid in the notebook
param_grid = [
    # For l2 penalty
    {
        'penalty': ['l2'],
        'C': [0.001, 0.01, 0.1, 1, 10, 100],
        'solver': ['lbfgs', 'liblinear', 'newton-cg', 'sag', 'saga'],
        'max_iter': [1000, 5000],  # Higher iterations for sparse data
        'class_weight': [None, 'balanced']  # Handle class imbalance if present
    },
    # For l1 penalty (best for sparse features)
    {
        'penalty': ['l1'],
        'C': [0.0001, 0.001, 0.01, 0.1, 1, 10],  # Extended lower range
        'solver': ['liblinear', 'saga'],
        'max_iter': [1000, 5000],
        'class_weight': [None, 'balanced']
    },
    # For elasticnet penalty
    {
        'penalty': ['elasticnet'],
        'l1_ratio': [0.1, 0.3, 0.5, 0.7, 0.9],  # Finer granularity
        'solver': ['saga'],
        'C': [0.001, 0.01, 0.1, 1, 10],  # Extended range
        'max_iter': [1000, 5000],
        'class_weight': [None, 'balanced']
    }
]

# Alternative: Focused grid for very sparse genomics data
sparse_focused_grid = [
    # Emphasize L1 and ElasticNet for feature selection
    {
        'penalty': ['l1'],
        'C': [0.001, 0.01, 0.1, 1],  # Focus on stronger regularization
        'solver': ['liblinear'],
        'max_iter': [5000]
    },
    {
        'penalty': ['elasticnet'],
        'l1_ratio': [0.5, 0.7, 0.9],  # Favor L1 component
        'solver': ['saga'],
        'C': [0.01, 0.1, 1],
        'max_iter': [5000]
    }
]
```

The sparse_focused_grid excludes L2 regularization and only includes L1 and ElasticNet penalties. Why is this choice particularly effective for sparse genomics data?

- A) L2 regularization is computationally too expensive for high-dimensional sparse data
- B) L1 and ElasticNet can set coefficients to exactly zero, performing automatic feature selection on irrelevant sparse features (while addressing multicollinearity)
- C) L2 regularization requires balanced features and cannot handle any level of sparsity
- D) L1 and ElasticNet converge faster than L2 when most features are sparse

Python dependencies

- Download requirements file `requirements.txt`