# AI in Genomics

## Contents

# 1 Biopython for Genomics

## 1.1 Intro

Let's begin with an example: Suppose we want to predict whether a particular sequence of DNA has a binding site for a transcription factor (TF) of your interest or not. Using the traditional approach, we would use a positional weight matrix (PWF) to scan the sequence and identify the potential motifs that are overrepresented. Using an ML-based approach, we would give an ML model plenty of DNA sequences until the ML model learns the mathematical relationship between the features from those DNA sequences that either have or don't have binding sites (labels) based on experimental results.

### 1.1.1 Sequencing techniques

- First-generation DNA sequencing: Sanger sequencing
- Second-generation DNA sequencing: Illumina sequencing by synthesis (SBS) technology. The typical size of generated fragments are in the range of 50-300 bases.
- Third-generation DNA sequencing: Illumina Pacific Bioscience single-molecule sequencing real-time (SMRT) and Oxford Nanopore Technologies nanopore sequencing. The typical size of generated fragments are in the range of 15000 bases.

### 1.1.2 Cloud computing for genomics data analysis

- Amazon Web Services (AWS)
- Azure
- Google Cloud Platform (GCP)

## 1.2 Biopython

### 1.2.1 Seq object

It essentially combines a Python string with biological methods such as DNA, RNA, or protein:

```
# Create Seq object
from Bio.Seq import Seq

my_seq = Seq("AGTAGGACAGAT")

# Print Seq object
print(my_seq)
```

```
## AGTAGGACAGAT
```

```
# Return the complement of the Seq object
print(my_seq.complement())
```

```
## TCATCCTGTCTA
```

### 1.2.2 SeqRecord object

This object differs from the Seq object in that it holds a sequence (as a Seq object) with additional information such as identifier, name, and description.

```
# Create SeqRecord object
from Bio.SeqRecord import SeqRecord

my_seqrecord = SeqRecord(
    Seq("AGTAGGACAGAT"),
    id="ENSG00000121966",
```

```
    name="Gene1",
    description="A sample gene",
)

# Print SeqRecord object
print(my_seqrecord)
```

```
## ID: ENSG00000121966
## Name: Gene1
## Description: A sample gene
## Number of features: 0
## Seq('AGTAGGACAGAT')
```

#### 1.2.3 SeqIO object

The SeqIO object in Biopython provides the standard sequence input/output interface. It supports several file formats as input and output including FASTA, FASTQ, and GenBank (GB).

### 1.3 Use case – Sequence analysis of Covid-19

First let's import (parse) the FASTA file:

```
# Parse file
from Bio import SeqIO

with open("Data/covid19.fasta") as file:
    for record in SeqIO.parse(file, "fasta"):
        print(f'Sequence information: \n{record}')
        print(f'Sequence length: {len(record)}')
```

```
## Sequence information:
## ID: NC_045512.2
## Name: NC_045512.2
## Description: NC_045512.2 Severe acute respiratory syndrome coronavirus 2 isolate Wuhan-Hu-1, complet
## Number of features: 0
## Seq('ATTAAAGGTTTATACCTTCCCAGGTAACAAACCAACCAACTTTCGATCTCTTGT...AAA')
## Sequence length: 29903
```

#### 1.3.1 Calculate GC content

Now let's calculate its GC content. GC content is one of the important features of a DNA sequence as it is an important predictor of gene function and species ecology. GC content is calculated by counting the number of Gs and Cs in the sequence and dividing that by the total sequence length.

```
from Bio import SeqIO
from Bio.SeqUtils import gc_fraction

with open("Data/covid19.fasta") as file:
    for record in SeqIO.parse(file, "fasta"):
        gc_content = gc_fraction(record)
        print(f'GC content: {round(gc_content, 2)}')
```

```
## GC content: 0.38
```

#### 1.3.2 Calculate nucleotide content

Nucleotide content such as the percentages of A, T, C, and G are useful for sequence characterization purposes.

```
from Bio import SeqIO

with open("Data/covid19.fasta") as file:
    for record in SeqIO.parse(file, "fasta"):
        seq_record = record.seq
        seq_length = len(record.seq)
        print(f'{round(seq_record.count("T") / seq_length * 100, 1)}% T')
        print(f'{round(seq_record.count("A") / seq_length * 100, 1)}% A')
        print(f'{round(seq_record.count("C") / seq_length * 100, 1)}% C')
        print(f'{round(seq_record.count("G") / seq_length * 100, 1)}% G')
```

```
## 32.1% T
## 29.9% A
## 18.4% C
## 19.6% G
```

### 1.3.3 Calculate dinucleotide content

It's also valuable to count the dinucleotides (AT, AC, GT, and so on).

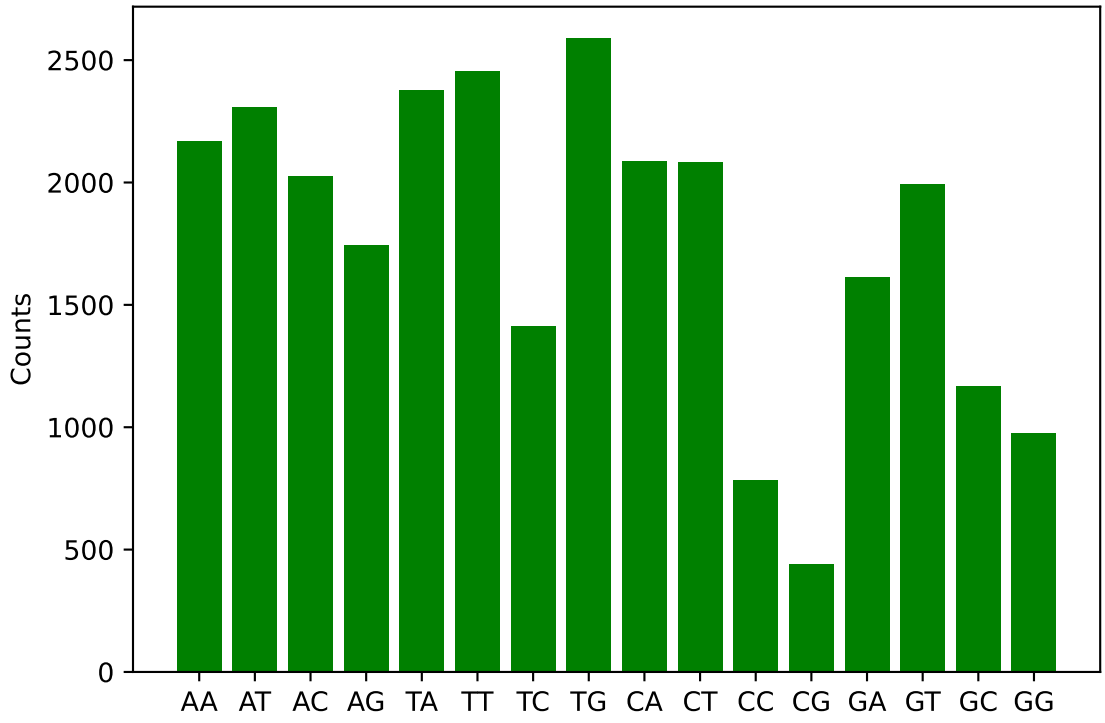
```
from Bio import SeqIO

nucl = ['A', 'T', 'C', 'G']
di_nucl_dict = {}
with open("Data/covid19.fasta") as file:
    for record in SeqIO.parse(file, "fasta"):
        for n1 in nucl:
            for n2 in nucl:
                di = str(n1) + str(n2)
                di_nucl_dict[di] = record.seq.count(di)

print(di_nucl_dict)
```

```
## {'AA': 2169, 'AT': 2308, 'AC': 2023, 'AG': 1742, 'TA': 2377, 'TT': 2454, 'TC': 1413, 'TG': 2589, 'CA
```

```
# Generate a dinucleotide plot
import matplotlib.pyplot as plt

di = [k for k, v in di_nucl_dict.items()]
counts = [v for k, v in di_nucl_dict.items()]
plt.bar(di,counts, color = "green")
```

```
## <BarContainer object of 16 artists>
```

```
plt.ylabel("Counts")
plt.show()
```




### 1.3.4 Save features

Let's save all the features that we have extracted so far into a file and get it ready for modeling.

```
# Extract all features
from Bio import SeqIO
from Bio.SeqUtils import gc_fraction
import pandas as pd

nucl = ['A', 'T', 'C', 'G']
features = {}

with open('Data/covid19.fasta') as file:
    for record in SeqIO.parse(file, "fasta"):
        for n1 in nucl:
            for n2 in nucl:
                di = str(n1) + str(n2)
                features[di] = record.seq.count(di)
            A_count = record.seq.count('A')
            features['A_count'] = round(A_count / len(record) * 100, 2)
            C_count = record.seq.count('C')
            features['C_count'] = round(C_count / len(record) * 100, 2)
            G_count = record.seq.count('G')
            features['G_count'] = round(G_count / len(record) * 100, 2)
            T_count = record.seq.count('T')
            features['T_count'] = round(T_count / len(record) * 100, 2)
```

```
            features['GC_content'] = round(gc_fraction(record), 2)
            features['Size'] = len(record)

# Create a dataframe
features_df = pd.DataFrame.from_dict([features])
features_df['virus'] = "Covid19"

# Save features into a sile
features_df.to_csv("Output/covid19_features.csv", index = None)

# Check the output file
features = pd.read_csv("Output/covid19_features.csv")
features
```

```
##      AA    AT    AC    AG  A_count  ...    GA    GT    GC   GG    virus
## 0  2169  2308  2023  1742    29.94  ...  1612  1990  1168  973  Covid19
##
## [1 rows x 23 columns]
```

### 1.4 Motif finder

A motif is a pattern in a nucleotide or amino acid sequence that has a specific structure. Sequence motifs play a key role in gene expression regulating both transcriptional and post-transcriptional levels.

```
from Bio import motifs
from Bio.Seq import Seq

# Create a DNA motif Seq object
my_motif = [Seq("ACGT"), Seq("TCGA"), Seq("CGGC")]

# Convert Seq object to motif object
motifs = motifs.create(my_motif)
print(motifs)
```

```
## ACGT
## TCGA
## CGGC
```

```
print(motifs.counts)
```

```
##         0      1      2      3
## A:   1.00   0.00   0.00   1.00
## C:   1.00   2.00   0.00   1.00
## G:   0.00   1.00   3.00   0.00
## T:   1.00   0.00   0.00   1.00
```

Now create a logo from the motifs:

```
motifs.weblogo('Output/my_motif.png')
```

## 2 Machine Learning for Genomics

### 2.1 The basic workflow of ML in genomics

1. Data collection
2. Preprocessing

3. Exploratory data analysis (EDA) and visualization
4. Feature extraction and selection
5. Train-test splitting
6. Model training
7. Model evaluation
8. Model interpretation
9. Model deployment
10. Model monitoring

- For model evaluation, one may use many metrics such as MSE, MAE, RMSE, and R-Squared.
- Some model-agnostic interpretability methods include: Local-Interpretable Modelagnostic Explanations (LIME), SHapley Additive ExPlanations (SHAP), and Explanation Summary (ExSUM).

## 2.2 Use case – Disease prediction

- Goal: Mapping the relationships between individual patients' sample gene expression values (features) and the target variable (Normal versus Tumor).
- Task: Classification
- Model: Logistic regression
- Data structure: Each row of the data represents a patient sample that consists of gene expressions.

### 2.2.1 Data collection

We will use the gene expression data of lung cancer samples and we will try to predict normal versus tumor outcomes.

```
import pandas as pd

lung1 = pd.read_csv("Data/Lung/GSE87340.csv.zip")
lung2 = pd.read_csv("Data/Lung/GSE60052.csv.zip")
lung3_1 = pd.read_csv("Data/Lung/GSE40419_1.csv.zip")
lung3_2 = pd.read_csv("Data/Lung/GSE40419_2.csv.zip")
lung4 = pd.read_csv("Data/Lung/GSE37764.csv.zip")
lung_1_4 = pd.concat([lung1, lung2, lung3_1, lung3_2, lung4])

# Check data
lung_1_4.iloc[:,0:10].head()
```

```
##            ID    class  ...  ENSG00000000971  ENSG00000001036
## 0  SRR4296063   Normal  ...        13.178872        11.469473
## 1  SRR4296064    Tumor  ...        13.208972        11.510862
## 2  SRR4296065   Normal  ...        14.038661        11.651766
## 3  SRR4296066    Tumor  ...        13.170466        11.546855
## 4  SRR4296067   Normal  ...        13.538341        11.733252
##
## [5 rows x 10 columns]
```

### 2.2.2 Data preprocessing

#### 2.2.2.1 Dealing with missing data

Remove it or impute it.

```
# Check the amount of missing data in each column
lung_1_4.isna().sum()

# Check the amount of missing data in all columns
```

```
## ID                 0
```

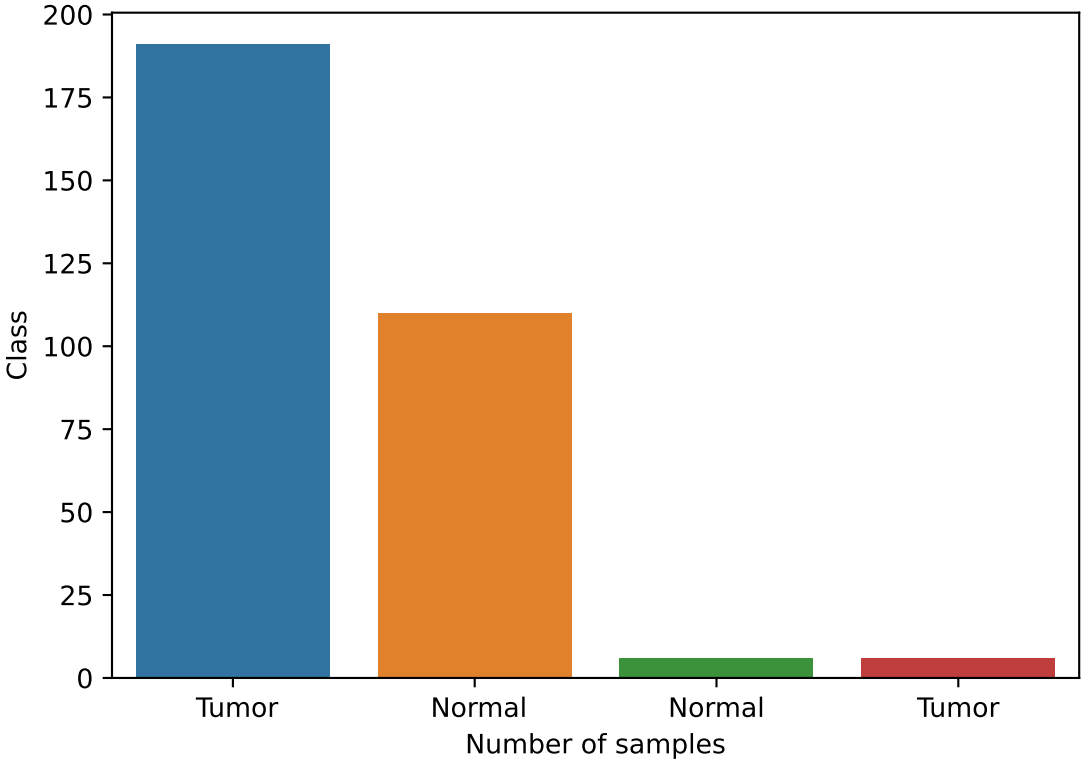
```
## class              0
## ENSG00000000003    0
## ENSG00000000005    0
## ENSG00000000419    0
##                   ..
## ENSG00000285990    0
## ENSG00000285991    0
## ENSG00000285992    0
## ENSG00000285993    0
## ENSG00000285994    0
## Length: 58737, dtype: int64
```

```
lung_1_4.isna().sum().sum()
```

```
## 0
```

**2.2.2.2 EDA** Let's first start by plotting the distribution of samples corresponding to each lung cancer type. We first create a DataFrame of the class column, then calculate the number of rows corresponding to each class, and then reset the index to make it easy for plotting.

```
df = lung_1_4['class'].value_counts().reset_index()

# Visualize the classes on a bar plot
import seaborn as sns
import matplotlib.pyplot as plt

sns.barplot(x = "class", y = "count", data = df)
plt.xlabel("Number of samples")
plt.ylabel("Class")
```



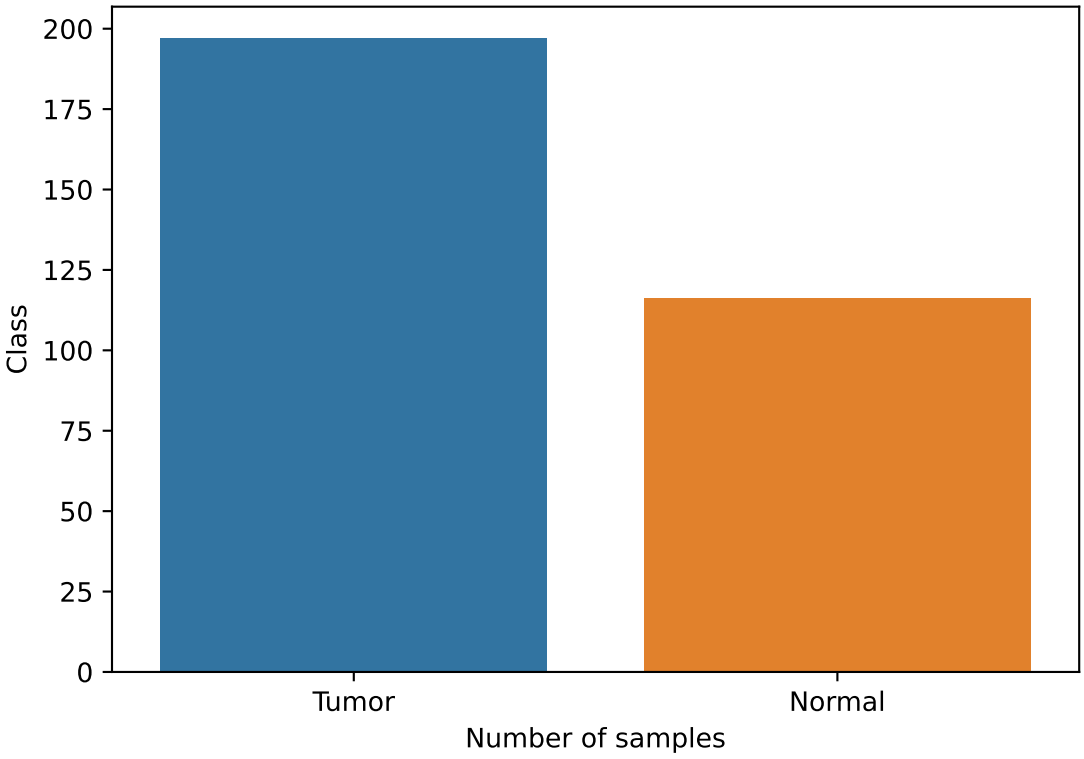

We have a problem now. As you can see, there are two types of samples, both of which are classified as Normal and the same for Tumor. Let's look at the different classes closely and see what's going on:

```
set(lung_1_4['class'])
```

```
## {' Normal', 'Normal', 'Tumor', ' Tumor'}
```

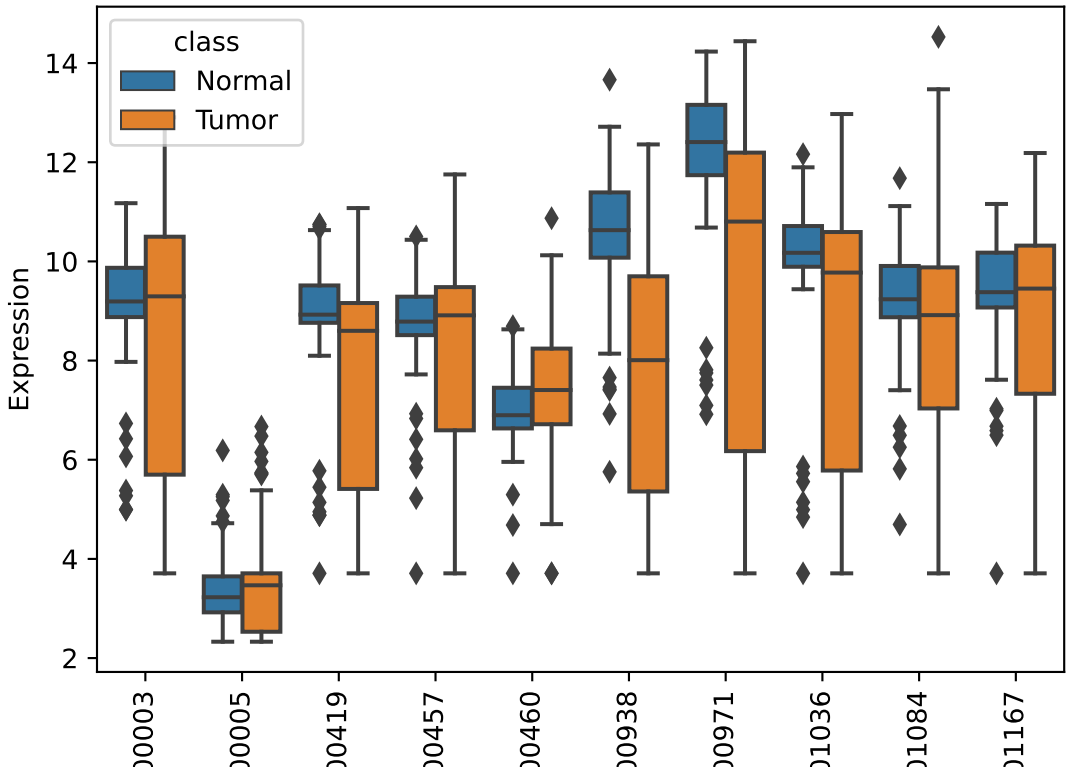
If you look closely, we notice that there is an extra space in front of the first and second classes. Let's rename those right away using the following replace method:

```
lung_1_4['class'] = lung_1_4['class'].replace(' Normal', 'Normal')
lung_1_4['class'] = lung_1_4['class'].replace(' Tumor', 'Tumor')
df = lung_1_4['class'].value_counts().reset_index()

# Replot
sns.barplot(x = "class", y = "count", data = df)
plt.xlabel("Number of samples")
plt.ylabel("Class")
```




**2.2.2.3 Data transformation** Any systematic differences between samples must be corrected before proceeding to the next step. First, we will restrict our dataset to the first 10 columns since it is challenging to visualize all the columns at once in a single boxplot. Then, we convert the data from wide format to long format using the melt method in Pandas:

```
lung_1_4_m = pd.melt(lung_1_4.iloc[:,1:12], id_vars = "class")

# Look at the distribution of expression across selected samples
ax = sns.boxplot(x = "variable" , y = "value", data = lung_1_4_m, hue = "class")
ax.set_xticklabels(ax.get_xticklabels(), rotation = 90)
plt.xlabel("Genes")
plt.ylabel("Expression")
```




Each sample has a somewhat similar distribution of gene expression values except for the first few samples (compare the medians). In addition, the expression values are already normalized and there is no need to normalize this further. So, let's proceed without normalizing these samples.

### 2.2.3 Train-test splitting

In this case, we will split the train and test datasets in the ratio of 75:25.

```
# Drop the ID and class columns in the dataset, and convert it to a NumPy ndarray
x_data = lung_1_4.drop(['class', 'ID'], axis = 1).values

# Similarly, we will create a NumPy ndarray for the labels
y_data = lung_1_4['class'].values

# Convert the categorical data in the type column to numbers using the ordinal encoding method
classes = lung_1_4['class'].unique().tolist()

import numpy as np
func = lambda x: classes.index(x)
y_data = np.asarray([func(i) for i in y_data], dtype = "float32")
print(y_data[1:10])
```

```
## [1. 0. 1. 0. 1. 0. 1. 0. 1.]
```

Here, 0 represents the Normal class, while 1 represents the Tumor class. Now, we are ready to split the data into training and testing.

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(x_data, y_data, random_state = 42, test_size = 0.25

print(f'Training X shape: {X_train.shape}')
```

```
## Training X shape: (234, 58735)
```

```
print(f'Training Y shape: {y_train.shape}')
```

```
## Training Y shape: (234,)
```

```
print(f'Test X shape: {X_test.shape}')
```

```
## Test X shape: (79, 58735)
```

```
print(f'Test Y shape: {y_test.shape}')
```

```
## Test Y shape: (79,)
```

### 2.2.4 Model training

```
model_lung1 = LogisticRegression()
model_lung1.fit(X_train, y_train)
```

```
## LogisticRegression()
##
## /Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages/sklearn/linear_model/
## STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.
##
## Increase the number of iterations (max_iter) or scale the data as shown in:
##     https://scikit-learn.org/stable/modules/preprocessing.html
## Please also refer to the documentation for alternative solver options:
##     https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
##   n_iter_i = _check_optimize_result(
```

### 2.2.5 Model evaluation

Now that model has been trained, let's run the model on one sample of the test data.

```
pred = model_lung1.predict(X_test[12].reshape(1, -1))
pred
```

```
## array([1.], dtype=float32)
```

Do predictions for all samples in the test data:

```
all_pred_lung = model_lung1.predict(X_test)
```

Let's calculate the accuracy score:

```
model_lung1.score(X_test, y_test)
```

```
## 0.9620253164556962
```

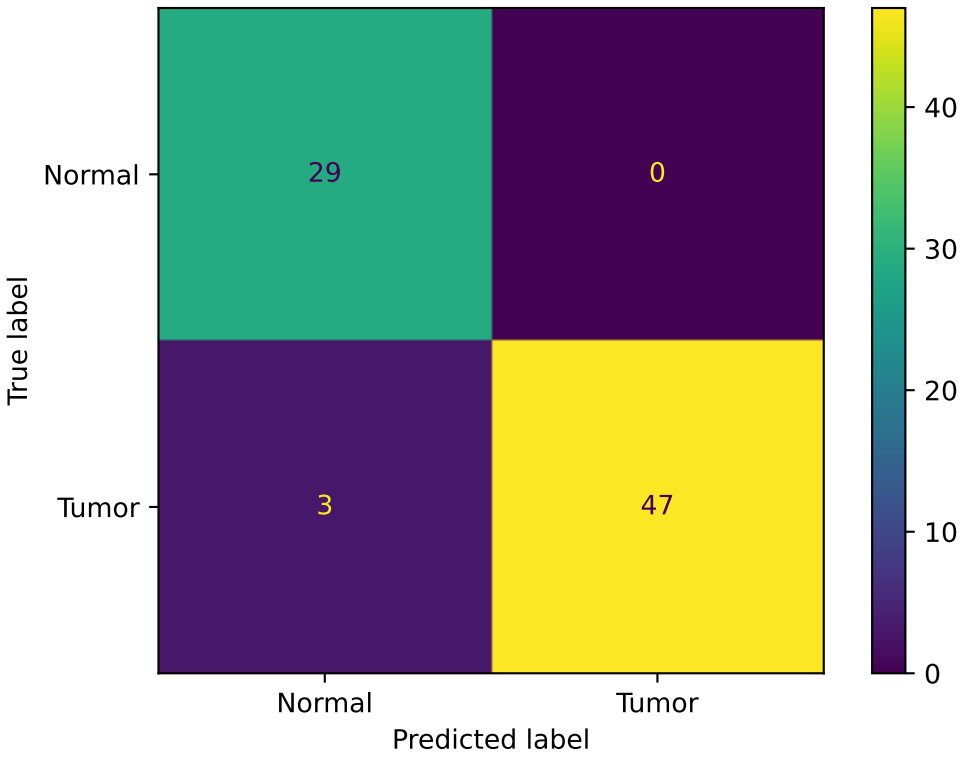
Let's run a confusion matrix:

```
from sklearn.metrics import confusion_matrix ,ConfusionMatrixDisplay, classification_report
```

```
cm = confusion_matrix(y_test, all_pred_lung)
disp = ConfusionMatrixDisplay(confusion_matrix = cm, display_labels = ["Normal", 'Tumor'])
disp.plot()
```

```
## <sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay object at 0x11c70f850>
```

```
plt.show()
```




Please note that the cost of misclassifying a sample is high for false negative samples compared to false positive ones because we don't want to miss any patient that has a tumor.

Now, let's get the classification report:

```
print(classification_report(y_test, all_pred_lung))
```

```
##               precision    recall  f1-score   support
##
##          0.0       0.91      1.00      0.95        29
##          1.0       1.00      0.94      0.97        50
##
##     accuracy                           0.96        79
##    macro avg       0.95      0.97      0.96        79
## weighted avg       0.97      0.96      0.96        79
```

# 3 Deep Learning in Genomics

## 3.1 Application of DNNs in genomics

### 3.1.1 Gene expression prediction

DNNs, with their ability to map the input-output data, are suited for gene expression-based disease classification. For example, CNNs were successfully used to classify Alzheimer's disease.

### 3.1.2 SNP prediction

SNPs are commonly used to detect disease-causing genes in humans, predict a person's response to drugs or their susceptibility to developing the disease, and classifying complex diseases using Genomics SNP data.

### 3.1.3 Protein structure predictions

It involves modeling the relationship between the amino acids of a protein and its corresponding 3D structure. By 2020, AphaFold's performance is impressive, and it is now considered the go-to model for predicting protein structure.

### 3.1.4 Regulatory genomics

Regulatory genomics is the study of gene regulatory elements such as promoters, enhancers, silencers, insulators, and so on. They play an important role in gene regulation and hence functionally characterizing them is very important. In addition to these gene regulatory elements, identifying sequence motifs in DNA and RNA regulatory regions is key since they represent target sites of a particular regulatory protein, such as the transcription factor (TF).

### 3.1.5 Gene regulatory networks (GRNs)

GRNs are defined as networks that are inferred by gene expression data. GRNs are an exciting area of functional genomics and represent causal relationships between the regulators and the target genes. GRNs are important to understand the causal map of network interactions, molecular marker detection, hub gene detection, and so on.

### 3.1.6 Single-cell RNA sequencing (scRNA-Seq)

scRNA-Seq enables gene expression measurements in individual cells, thereby enabling cell-type clustering. Despite its huge success, biological inference remains the major limitation because of the sparse nature of the generated data. In addition, there is a large volume of dropout events in the data.

# 4 CNNs in Genomics

## 4.1 Why CNNs

**Problems with FCNNs:** 1. There are too many parameters to learn (e.g., for a 32 x 32 color image, we would have 32 x 32 x 3 = 3,072 parameters). 2. The 2D or 3D image is converted to a 1D flattened vector, and so the spatial relationship of the different pixels is completely lost.

CNNs are currently being used in genomics tasks where local patterns are very important to the outcome—for example, the detection of conserved motifs to identify blocks of genes in a DNA sequence or binding sites of a protein such as a transcription factor (TF).