MLR570 - Homework Assignment 2

Dara Varam 👺 & Lujain Khalil 🦦

Questions marked with 👺 were solved by Dara and then double-checked by Lujain. Questions marked with 🦦 were solved by Lujain and then double-checked by Dara. Also Dara was recovering from surgery while doing this. Please be nice in grading! Code for everything is available upon request.

I. Question 1 👺

Given the following code:

```python
import numpy as np
import matplotlib.pyplot as plt
from numpy.linalg import inv
# Seed for reproducibility
np.random.seed(42)
# Generate independent variable (X)
X = np.linspace(1, 10, 100)
X = X[:, np.newaxis] # Reshape X to be a 2D array (100x1)
true_beta = 3
y = true_beta * X.squeeze() + np.random.normal(0, 1, size=100)
# Add bias term (intercept) to X (i.e., X0 = 1)
X_b = np.c_[np.ones((100, 1)), X] # X_b is now (100x2)
```

1) Derive the Least Squares algorithm, starting from the objective function and leading to the closed-form solution.
2) Discuss the advantages and limitations of the Least Squares method. What is the computational complexity?
3) Implement Least Squares using the derived equations on the provided dataset. Compare your results with sklearn's implementation.

For a dataset with n samples and m features per sample, let $X \in \mathbb{R}^{n\times(m+1)}$ be the feature matrix which includes the bias terms (here, the size is $n \times (m+1)$ since we have an extra term for the biases). Let $y \in \mathbb{R}^n$ be the target vector, ϵ be the errors and finally, W be the matrix of parameters / weights that we will use to estimate the model that fits the data.

We start with the equation for a linear model, which is given by:

$$y = XW + \epsilon$$

We know that the objective function for the least-squares method is to **minimize** the sum of squared errors, and thus we minimize (over W) as follows:

$$\min_W \sum_{i=1}^{n} (y_i - X_i W)^2$$

For each sample $i \in 1, \dots, n$. Alternatively, we can write this in matrix form:

$$\min_W ||y - XW||^2 = \min_W (y - XW)^\top (y - XW)$$

To minimize this, we first expand and then differentiate (take the partial derivatives) and set to 0 such that we can get the (local) minima of the function.

$$\begin{aligned} ||y - XW||^2 &= (y - XW)^\top (y - XW) \\ = y^\top y - y^\top XW &- W^\top X^\top y + W^\top X^\top XW \\ &= y^\top y - 2W^\top X^\top y + W^\top X^\top XW \end{aligned}$$

We differentiate w.r.t W.

- Derivative of $y^\top y = 0$ since there is no W in this term.
- Derivative of $-2W^\top X^\top y = -2X^\top y$ since $-2X^\top y$ is a constant term
- Derivative of $W^\top X^\top XW = 2X^\top XW$ (thank you Matrix Cookbook).

The full thing put together:

$$\frac{\partial}{\partial W}(||y - XW||^2) = -2X^\top y + 2X^\top XW$$

We can then set this to 0 and re-arrange:

$$\begin{aligned} -2X^\top y + 2X^\top XW &= 0 \\ \implies 2X^\top y &= 2X^\top XW \\ \implies X^\top y &= X^\top XW \end{aligned}$$

The last thing to do here is to solve for W.

$$W = (X^\top X)^+ X^\top y$$

Here, $+$ represents the pseudo-inverse of $X^\top X$, assuming that it is **non-invertible**. If it is invertible, then we can just take the inverse directly instead of the pseudo-inverse.

That's it. That's the closed-form solution of the least-squares algorithm. Now let's implement it in code and compare it to sklearn's implementation:

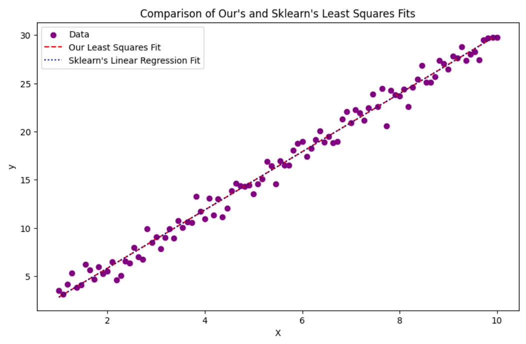

Fig. 1. Comparison of Least Squares approaches using the closed-form solutions we derived and the SKLearn implementation that comes with Python.

As you can see, the two are the exact same. The parameters W_0 and W_1 for the parameters (y-intercept and coefficient of X, respectively), are as follows, which are again, the exact same:

	W_0	W_1
SKLearn Implementation	-0.18813871	3.01532585
Our closed-form implementation	-0.18813871	3.01532585

TABLE I

Model coefficients for two implementations of the linear regressor.

Now you may be wondering why these two are the exact same. The solution is somewhere in the documentation / implementation of the LinearRegression() function of SKLearn. According to the documentation, SKLearn actually uses the closed-form solution for solving the least-squares problem. This is tpyically the case for datasets in which the number of features p is less than the number of samples n, or in cases where the dataset itself is not particularly large. For larger datasets (or more complex datasets in general), it uses the SVD decomposition, making the calculations more stable especially in the case of non-full-rank data. The library also uses specific optimizers such as LAPACK for optimization and efficiency.

For demonstrative purposes, we tried to increase the complexity of the dataset by adding more features and by increasing the number of samples. For a dataset with 4 features and some $n = 100000$, we get the following:

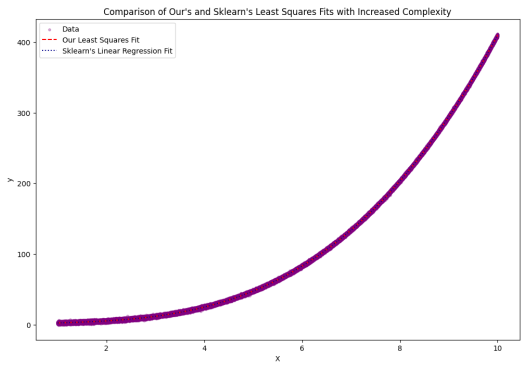

Fig. 2. Dataset with 4 features, $n = 100000$ samples. The closed-form solution and the SKLearn implementation are still the same.

	W_0	W_1	W_2	W_3
SKLearn Implementation	0.026	2.98	-1.20	0.50
Our closed-form implementation	0.026	2.98	-1.20	0.50

TABLE II

Model coefficients for two implementations of the linear regressor, now with the bigger, more complex dataset.

We thus conclude that, at least numerically for even relatively large datasets, the closed-form least-squared method is the same whether calculated manually or through using the SKLearn implementation.

Computational Complexity:

- To calculate the closed form, we use $(X^\top X)^+ X^\top y$ as mentioned before.
- The matrix multiplication of $X^\top X$ is of complexity $O(nm^2)$, where n is the number of samples and m is the number of features. This is explained by the fact that the matrix $X \in \mathbb{R}^{n\times m}$, and $X^\top \in \mathbb{R}^{m\times n}$.
- The inversion step has a complexity of $O(m^3)$ due to the process of Gaussian elimination and other transformations.
- The total complexity would, in this case, be $O(nm^2 + m^3)$.

Finally, let's discuss some advantages and disadvantages:

Advantages:

- Each coefficient (each w_i in our case) has a clear meaning that can be used to both analytically and graphically evaluate the model. We know that our w_0 corresponds to the y-intercept, and each subsequent w_i represnts the coefficient of each feature (for a unit increase in some feature x_i, there is an equivalent and proportional w_i increase in the prediction). This is highly interpretable.
- The fact that a closed-form solution exists means that it is very easy to implement. There is no need to take any numerical approximations or iteratively update values to get to it - it is a direct plug-and-solve approach.

- The least-squares method is optimal in the case where the error is normally distributed with constant variance and have no correlation with each other. In the case of the code we are given, the errors are not taken from any correlated set, and therefore OLS is optimal.

Disadvantages:

- It can be computationally expensive. The bigger the size of the matrix X (whether that be in terms of the number of samples or the number of features), the more computational power it would take to calculate the multiplication $X^\top X$ and the inversion.
- Naturally, since we are looking at squares differences (that are not scaled or weighted), then we expect the model to also be sensitive to outliers.
- $X^\top X$ can become singular in higher dimensions.
- We may be prone to overfitting as with the example shown above with 4 features. If we are approaching a case where the number of samples is close to or the same as the number of features, then we will definitely overfit. I will plot this for you to see.

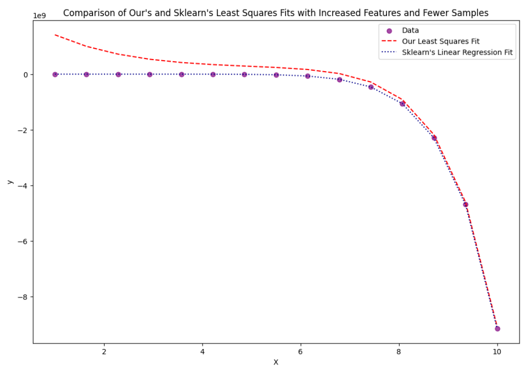

Fig. 3. Case of 15 samples and 10 features (i.e. $n = 15, m = 10$) where we can see that the number of features is approaching the number of samples.

You can clearly see in this case that our implementation performs worse than the SKLearn implementation, but it is also evident that there is some serious overfitting going on. The model's prediction line is **exactly** passing through the datapoints so the introduction of any testing data that may even be slightly off this line would not be predicted accurately.

II. Question 2

Given the following code:

```python
import numpy as np
import matplotlib.pyplot as plt
from numpy.linalg import inv
# Seed for reproducibility
np.random.seed(42)
# Generate independent variable (X)
X = np.linspace(1, 10, 100)
X = X[:, np.newaxis]  # Reshape X to be a 2D array
    (100x1)
# Generate heteroscedastic data with extremely
    increased variance
true_beta = 3
y = true_beta * X.squeeze() + np.random.normal(0, X
    .squeeze()**4, size=100)
# Add bias term (intercept) to X (i.e., X0 = 1)
X_b = np.c_[np.ones((100, 1)), X]  # X_b is now
    (100x2)
```

1) Derive the Weighted Least Squares algorithm, starting from the objective function and leading to the closed-form solution.
2) Discuss the advantages and limitations of the Weighted Least Squares method. What is the computational complexity?
3) Implement Weighted Least Squares using the derived equations on the provided dataset. Compare your results with sklearn's implementation.

Let $X \in \mathbb{R}^{n\times(m+1)}$ be the feature matrix and $y \in \mathbb{R}^n$ be the target vector, where n is the sample size and m is the number of features in our data. The extra column in X is set to 1's to account for the bias term in matrix calculations. Define the equation for a linear regression model as follows:

$$y = X\beta + \epsilon$$

where $\beta \in \mathbb{R}^{m+1}$ is the vector of coefficients and ϵ is the error term.

The objective in a standard linear regression model is to **minimize** a cost function $J(\beta)$ w.r.t β, where the cost is (usually) defined as the sum of squared residuals:

$$J(\beta) = \sum_{i=1}^{n} (y_i - x_i\beta)^2$$

Re-writing $J(\beta)$ in matrix form:

$$J(\beta) = (y - X\beta)^\top (y - X\beta)$$

This approach, known as Ordinary Least Squares (OLS), does not account for heteroscedasticity (an unnecessarily complicated way to say non-constant variance) in our data. To handle this, the Weighted Least Squares (WLS) approach introduces a diagonal weight matrix $W \in \mathbb{R}^{n\times n}$ such that each value in the diagonal of W assigns a weight to each observation in X. Observations with higher variance are given less weight, and observations with lower variance are given higher weights (weight calculations will be discussed later on in this section).

Re-defining the cost function $J(\beta)$, we get:

$$J(\beta) = (y - X\beta)^\top W (y - X\beta)$$

Our objective to minimize $J(\beta)$ w.r.t β can be written as follows:

$$\min_{\beta} J(\beta) = \min_{\beta} (y - X\beta)^\top W (y - X\beta)$$

Minimizing the objective function starts by finding the partial derivative of $J(\beta)$, and then setting it to 0. We keep in mind the following:

- $W = W^\top$, since W is a diagonal matrix
- $W + W^\top = 2W$ for the same reason
- $\frac{\partial}{\partial \beta}(u^\top A u) = u^\top (A + A^\top)\frac{\partial u}{\partial \beta}$ (thanks Matrix Cookbook)

$$\frac{\partial}{\partial \beta}(y - X\beta)^\top W (y - X\beta) = 0$$

$$\implies (y - X\beta)^\top (W + W^\top)\frac{\partial}{\partial \beta}(y - X\beta) = 0$$

$$\implies (y - X\beta)^\top (2W)(-X) = 0$$

$$\implies (y - X\beta)^\top W X = 0$$

Rearranging to find β, we get:

$$X^\top W^\top y - X^\top W^\top X\beta = 0$$

$$\implies X^\top W^\top X\beta = X^\top W^\top y$$

$$\implies \beta = (X^\top W^\top X)^{-1} X^\top W^\top y$$

$$\implies \beta = (X^\top W X)^{-1} X^\top W y$$

The closed-formed solution for a WLS regression model is $\beta = (X^\top W X)^{-1} X^\top W y$.

Before implementing this in code, we need to think of a way to define our weight matrix W as some function of our input data, $f(X) = W$. After some research, we found the following:

1) $f(X) = X^{-n}$ for any $n \in \mathbb{Z}^+$. The higher the value of X, the lower the weight assigned to it.
2) $f(X) = e^{-X}$. Similar to the above function, but the weight decreases exponentially as X increases.
3) $f(X) = \sigma^{-2}$ such that σ^2 is the estimated variance in error. There are many ways to calculate σ^2, one of them being the squared residuals $(y - \hat{y})^2$ from an initial model fit like OLS regression.

We don't see how the first two functions make sense. It assumes that smaller magnitudes of x have lower variance, and assigns weights accordingly. The third function, $f(X) = \sigma^{-2}$, is a lot more intuitive and is probably more generalizable to different fanning patterns in the data.

We will run and plot all 3 variations of the above WLS regressions, along with sklearn's implementation of both OLS and WLS. The value of n for the first function will be set to $n = 2$:

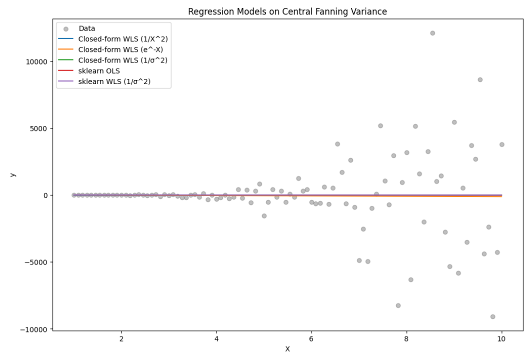

Fig. 4. Comparison of WLS approaches

We can see that all model variations perform very similarly. We can better interpret their differences by comparing the β_0 and β_1 values below:

	β_0	β_1
Closed-form WLS ($1/X^2$)	-3.959	-0.133
Closed-form WLS (e^-X)	20.951	-13.557
Closed-form WLS ($1/\sigma^2$)	5.660	-0.236
sklearn OLS	10.817	-2.192
sklearn WLS ($1/\sigma^2$)	5.660	-0.236

TABLE III
Model coefficients for various WLS implementations

We can see that sklearn's implementation of WLS is exactly the same as the closed-form solution approach. The LinearRegression() class allows for a sample_weight parameter to be passed when fitting the model. The weights are applied during the fitting process but do not explicitly form a diagonal weight matrix.

We know that the variance was set to X^4 when generating the data. For a more robust comparison, we will generate two other datasets with diagonal and backward fanning variance patterns as follows:

```python
# Set seed for reproducibility
np.random.seed(42)
# Function to generate datasets with different fanning patterns
def generate_fanning_data(pattern, n=100):
    X = np.linspace(1, 10, n)[:, np.newaxis]
    true_beta = 3
    if pattern == "diagonal":
        y = true_beta * X.squeeze() + np.random.normal(0, X.squeeze() * 2, size=n)
    elif pattern == "backward":
        y = true_beta * X.squeeze() + np.random.normal(0, (11 - X.squeeze()) * 2, size=n)
    elif pattern == "central":
        y = true_beta * X.squeeze() + np.random.normal(0, X.squeeze()**4, size=n)
    return X, y
```

After fitting the same models on the new data, we get the following:

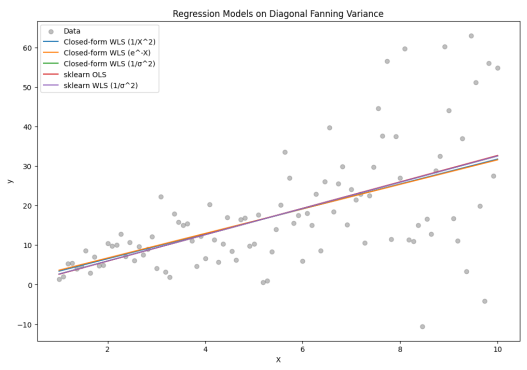

Fig. 5. Comparison of WLS approaches for diagonal fanning variance

	β_0	β_1
Closed-form WLS $(1/X^2)$	0.245	3.150
Closed-form WLS $(e^{-}X)$	0.538	3.101
Closed-form WLS $(1/\sigma^2)$	-0.656	3.319
sklearn OLS	-0.731	3.339
sklearn WLS $(1/\sigma^2)$	-0.656	3.319

TABLE IV
Model coefficients for various WLS implementations for diagonal fanning variance

With diagonal fanning variance, the differences in models are a little more obvious, but still very similar. This is probably because variance still increases as X increases. Testing with backward fanning variance:

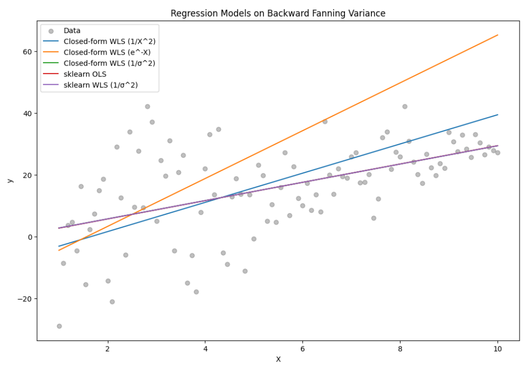

Fig. 6. Comparison of WLS approaches for backward fanning variance

	β_0	β_1
Closed-form WLS $(1/X^2)$	-7.857	4.724
Closed-form WLS $(e^{-}X)$	-12.194	7.738
Closed-form WLS $(1/\sigma^2)$	-0.199	2.955
sklearn OLS	-0.255	2.964
sklearn WLS $(1/\sigma^2)$	-0.199	2.955

TABLE V
Model coefficients for various WLS implementations for backward fanning variance

The differences here are a lot more obvious. We can see that for $f(X) = X^{-2}$ and $f(X) = e^{-X}$, the models are no longer reliable. This proves that these weight calculations are not generalizable, since in this case, variance and X values are not proportional. It is interesting, however, that standard OLS performed very similarly to WLS with $f(X) = \sigma^{-2}$. Further investigations on their differences with a variety of data would be interesting, but enough exploration for this question.

We conclude that sklearn's implementation is exactly the same as the closed-form approach, with flexibility in defining weight calculations.

Computational Complexity:

For n samples and m features, the computational complexity of $\beta = (X^\top W X)^{-1} X^\top W y$ is as follows:

- Assuming weights are calculated as a linear function of the input, computing the diagonal values of W is of complexity $O(n)$.
- The matrix multiplication of $X^\top W X$ is of complexity $O(nm^2)$, since $X^\top W \in \mathbb{R}^{m\times n}$ and $X \in \mathbb{R}^{n\times m}$.
- Inverting $X^\top W X \in \mathbb{R}^{m\times m}$ has a complexity of $O(m^3)$ due to the process of Gaussian elimination and other transformations.
- The matrix multiplication $(X^\top W X)^{-1} X^\top W$ is of complexity $O(nm^2)$, since $(X^\top W X)^{-1} \in \mathbb{R}^{m\times m}$ and $X^\top W \in \mathbb{R}^{m\times n}$.
- The total complexity would be $O(n) + O(nm^2) + O(m^3) + O(nm^2) = O(nm^2 + m^3)$.

Advantages:

- Similar to OLS, WLS is highly interpretable. Every unit increase of x causes an increase of β_1 to the target, with a starting value of β_0.
- With a suitable definition of weight calculations, WLS handles heteroscedasticity (non-constant variance) much better than OLS, giving us more efficient parameter estimations.
- Flexibility in how to define weight calculations allows us to create models that are specific to the variance patterns in our data.
- Robust to outliers due to the lower weights assigned to them.

Disadvantages:

- Depending on how weight calculations are defined, the overall complexity could increase drastically, especially in the cases where weights are computed iteratively.
- A complexity of $O(nm^2 + m^3)$ implies that WLS will be very costly for high-dimensional datasets, where $m \gg n$.
- Defining weight calculations that are not suitable to the data could cause fits that are much worse than OLS, as seen in Figure 6.

III. Question 3 👺

Given the following code:

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.kernel_approximation import RBFSampler
from sklearn.kernel_ridge import KernelRidge
```

```
# Generate a non-linear dataset
np.random.seed(42)
X = np.linspace(0, 10, 100).reshape(-1, 1) #
    Features
y = np.sin(X).ravel() + 0.1 * np.random.randn(100)
    # Non-linear target with noise
# Augment the features matrix for Least Squares (
    add bias term)
X_aug = np.hstack([np.ones((X.shape[0], 1)), X]) #
    Add a bias term (column of ones)
```

1) Derive the kernel least square, starting from the objective function and leading to the closed-form solution.
2) Discuss the advantages and limitations of the kernel least square. What is the computational complexity?
3) Implement kernel least square using the derived equations on the provided dataset. Compare your results with sklearn's implementation.

Here we go again. Let us define some terms first based on the assumption that n is the number of samples and m is the number of features. Let us take some $X \in \mathbb{R}^{n \times m}$ where X contains **both** the features and the target vector itself. We can re-write this to be the following:

$$X = \{(x_i, y_i)\}_{i=1}^{n}$$

With each $x_i \in \mathbb{R}^m$ and each $y_i \in \mathbb{R}$. Thus:

- Let $y \in \mathbb{R}^n$ be the target vector;
- Let $K \in \mathbb{R}^{n \times n}$ be the kernel matrix characterized by $K_{ij} = (x_i, x_j)$. K is any kernel function, such as RBF.
- Let $W \in \mathbb{R}^n$ be the weights in the **transformed** dimension; and
- Let λ be the regularization parameter.

Thus, we define the objective function as follows:

$$J(W) = ||y - KW||^2 + \lambda ||W||^2$$

We again want to minimize this term, so we have to take the partial derivative w.r.t W : $\frac{\partial J(W)}{\partial W}$. Let's get started on this.

$$\begin{aligned} J(\alpha) &= (y - KW)^\top (y - KW) + \lambda W^\top W \\ &= y^\top y - 2y^\top KW + W^\top K^\top KW + \lambda W^\top W \\ &\text{Since } K^\top = K (\text{symmetric matrix}); \\ \implies J(W) &= y^\top y - 2y^\top KW + W^\top KW + \lambda W^\top W \end{aligned}$$

We now take the derivative:

$$\begin{aligned} \frac{\partial J(W)}{\partial W} &= -2K^\top y + 2K^\top KW + 2\lambda W \\ &= -2Ky + 2(K + \lambda I)W \end{aligned}$$

We set this to zero to get the (local) minima:

$$\begin{aligned} -2Ky + 2(K + \lambda I)W &= 0 \\ 2Ky &= 2(K + \lambda I)W \\ Ky &= (K + \lambda I)W \\ \text{Finally, re-arrange for } W &: \\ W &= (K + \lambda I)^{+} y \end{aligned}$$

Here, we again use $+$ instead of the standard inverse to account for cases where the matrix $K + \lambda I$ is non-invertible. This is the closed-form solution. Now to compare its performance to the SKLearn implementation:

Consider the below below. Our own implementation is as follows:

```
def rbf_kernel(X1, X2, gamma):
    sq_dist = np.sum(X1**2, axis=1).reshape(-1, 1)
    + np.sum(X2**2, axis=1) - 2 * np.dot(X1, X2.T)
    return np.exp(-gamma * sq_dist)

def kernel_least_squares(X, y, gamma, lambda):
    K = rbf_kernel(X, X, gamma)
    alpha = np.linalg.inv(K + lambda * np.eye(len(K
    ))).dot(y)
    return alpha, K
```

The main idea is to define a kernel function (in our case, RBF) and an implementation whose calculation is dependent on the closed-form solution obtained earlier. Now let's see how this fairs agains the dataset we generated:

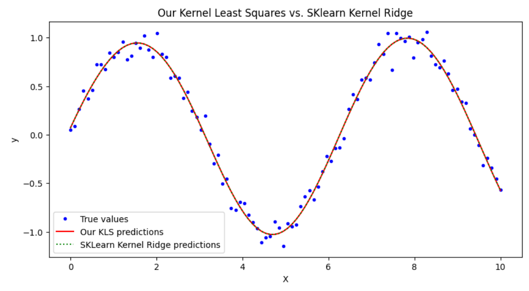

Fig. 7. The comparison of our KLS closed-form solution to the SKLearn implementation. Gamma here is 0.1 and $\lambda = 0.001$.

As you can see, the lines perfectly overlap with each other, again (as in the previous questions) suggesting that the SKLearn implementation also uses the closed-form solution. Let us look at the MSE values as confirmation of whether the two lines really are predicting the same things:

	MSE
Our KLS implementation	0.00746017154178525
SKLearn KLS implementation	0.007460171541780014

TABLE VI

Comparison between the MSE values of the two models used.

We can see that the values for the MSE match all the way down to the 14th decimal place after the 0. This suggests that they are in fact equal and the differences are due to numerical differences in calculations, which may be the case since we have different optimizers and efficiency libraries being used in the SKLearn toolkit.

Now let's look at some cases where the SKLearn implementation and our own KLS implementation may differ. We will try 2 cases in particular: Where we increase the number of samples and keep everything else the same; and when we decrease the number of samples and increase the number of features to match.

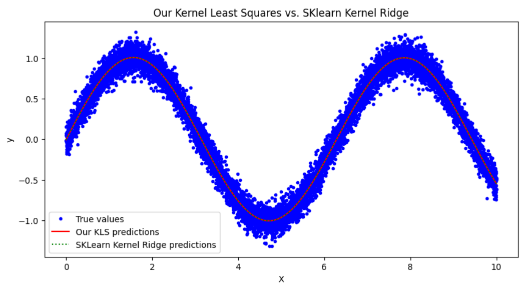

Fig. 8. Gamma $= 0.1$ and $\lambda = 0.001$, with $n = 10000$ samples. It is evident that the SKLearn implementation and our KLS implementation is still equivalent.

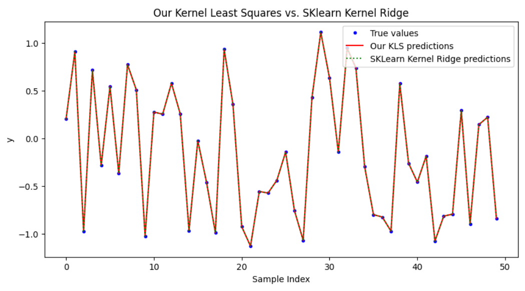

Fig. 9. Gamma $= 0.1$ and $\lambda = 0.001$, with $n = 50$ samples, and $m = 40$ features.

Let us compare the way these two look. First of all, we can see in Fig. 8 that an increased number of samples does not change the prediction line. SKLearn's implementation and the closed-form solution are exactly the same (with numerical differences in the order of 10^{-14}. Now, referring to Fig. 9, we can see that an increased number of features with less samples does not lead to a significant change in the way the two implementations look. However, we can see a significant increase in **overfitting**. The models now fit much more tightly to the datapoints, with an MSE of 4.674941×10^{-7}. This follows what happened in the earlier questions as well, with more of an aggressive overfitting here. However, to demonstrate the effect of the regularization parameter λ on this overfitting business, let us see what happens when we increase the λ to 0.1 (Fig. 10) and 1 (Fig. 11.

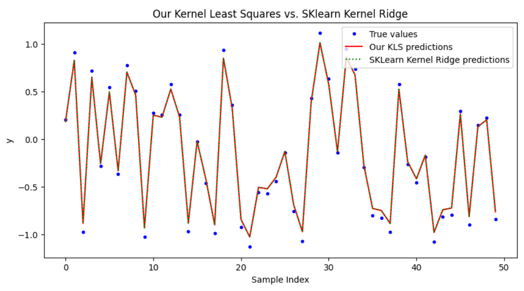

Fig. 10. Gamma $= 0.1$ and $\lambda = 0.1$, with $n = 50$ samples, and $m = 40$ features.

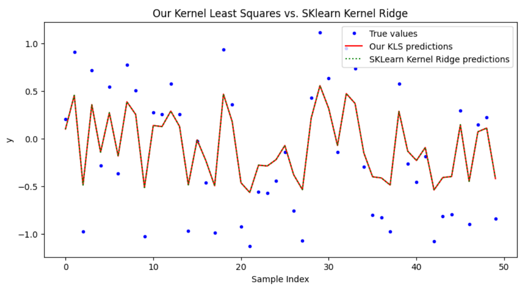

Fig. 11. Gamma $= 0.1$ and $\lambda = 1$, with $n = 50$ samples, and $m = 40$ features.

We can see that the model overfits less, but there is also a clear increase in the MSE values as well, indicating worse predictive performance. For $\lambda = 0.1$, the MSE is 0.0038713, whereas for $\lambda =$, the MSE is 0.117107. Comparatively worse and worse, obviously.

Computational Complexity: The computational complexity of our approach is as follows. We need to look at the equation:

$$W = (K + \lambda I)^{+} y$$

We first notice that the matrix $K \in \mathbb{R}^{n \times n}$. Each entry of K_{ij} has computational complexity $O(1)$ since we are applying one function to each of the entries. Since we have $n \times n$ entries, then clearly the computational complexity of generating K is $O(n^2)$.

The next item on our computational complexity journey is the inversion process of $(K + \lambda I)^{+}$. The addition operation is $O(1)$ so we just kind of disregard it. The matrix $(K + \lambda I) \in \mathbb{R}^{n \times n}$. We know from the earlier questions (and the internet) that the inversion of an $n \times n$ matrix is of computational complexity $O(n^3)$. Thus the total complexity is as follows:

$$O(n^2 + n^3) \implies O(n^3)$$

This is because the n^3 term dominates the process. Now we can look at some of the advantages and disadvantages of the KLS algorithm.

Advantages:

- Added flexibility: We have more freedom of choices when it comes to how we want to model the data.

We can change the gamma values, the λ values, the entire kernel function (RBF or otherwise), etc...

- Kernel trick: We do not need to really "go" to higher dimensions because of the kernel trick, so in terms of complexity, it is still not too complex. The inner products are computed in the same, original feature space.
- We are able to work with non-linearities a lot better now because we account for the higher dimensionality of data.

Disadvantages:

- Our weight vector W is no longer in the original feature space. Therefore, in terms of interpretation, it is more different to understand what each W_i represents when compared to the OLS or WLS methods in which each weight has a direct interpretation.
- The computational complexity is $O(n^3)$, which is, similar to previous LS methods, relatively high. This would be difficult for larger datasets for sure.
- We demonstrated how prone the model is to overfitting as was the case with other LS methods. Increasing the number of features makes the model overfit a lot.
- The model is sensitive to our choice of the regularization parameter. This could also result in overfitting or underfitting, as demonstrated in Figs. 10 and 11.

IV. Question 4 🦦

Given the following code:

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.kernel_approximation import RBFSampler
from sklearn.kernel_ridge import KernelRidge
# Generate a non-linear dataset
np.random.seed(42)
X = np.linspace(0, 10, 100).reshape(-1, 1)  # Features
y = np.sin(X).ravel() + 0.1 * np.random.randn(100)  # Non-linear target with noise
# Augment the features matrix for Least Squares (add bias term)
X_aug = np.hstack([np.ones((X.shape[0], 1)), X])  # Add a bias term (column of ones)
```

1) Derive the kernel ridge regression, starting from the objective function and leading to the closed-form solution.
2) Discuss the advantages and limitations of the kernel ridge regression. What is the computational complexity?
3) Implement kernel ridge regression using the derived equations on the provided dataset. Compare your results with sklearn's implementation.

Let $X \in \mathbb{R}^{n\times m}$ be our feature matrix and $y \in \mathbb{R}^n$ be our target vector, where n is the sample size and m is the number of features. We start by defining our kernel ridge regression objective function as follows:

$$J(\alpha) = \|y - K\alpha\|^2 + \lambda\|\alpha\|^2$$

where:

- λ is the regularization parameter,
- K is the kernel matrix, where each entry $K_{ij} = k(x_i, x_j)$ represents the similarity between samples x_i and x_j using some kernel function $k(\cdot,\cdot)$ (typically RBF).
- α is the weight vector for the **kernelized** representation of the model.

To get to our closed-form solution, we need to start by deriving the partial derivative of $J(\alpha)$ w.r.t. α, then setting it to 0. Let's start by re-writing $J(\alpha)$ in matrix form:

$$J(\alpha) = (y - K\alpha)^T(y - K\alpha) + \lambda\alpha^T\alpha$$

$$\implies J(\alpha) = y^Ty - 2y^TK\alpha + \alpha^TK^TK\alpha + \lambda\alpha^T\alpha$$

After some help from the Matrix Cookbook, we derive each term w.r.t. α:

$$\frac{\partial J(\alpha)}{\partial\alpha} = -2K^Ty + 2K^TK\alpha + 2\lambda\alpha$$

Setting to 0:

$$-2K^Ty + 2K^TK\alpha + 2\lambda\alpha = 0$$

$$\implies K^TK\alpha + \lambda\alpha = K^Ty$$

$$\implies (K + \lambda I)\alpha = y$$

$$\implies \alpha = (K + \lambda I)^{-1}y$$

Finally, our closed-form solution for kernel ridge regression is:

$$\alpha = (K + \lambda I)^{-1}y$$

Let's compare this approach with sklearn's implementation. Consider the following code:

```python
def rbf_kernel(X1, X2, gamma=0.1):
    sq_dist = np.sum(X1**2, axis=1).reshape(-1, 1) + np.sum(X2**2, axis=1) - 2 * np.dot(X1, X2.T)
    return np.exp(-gamma * sq_dist)

def kernel_ridge_regression(K, y, lambd=0.001):
    n = K.shape[0]
    return np.linalg.solve(K + lambd * np.eye(n), y)
```

We use the Radial Basis Function kernel in our implementation and calculate manually. We set $\gamma = 0.1$ and $\lambda = 0.001$ in both our closed-form implementation and sklearn's KernelRidge() class. The results are as follows:

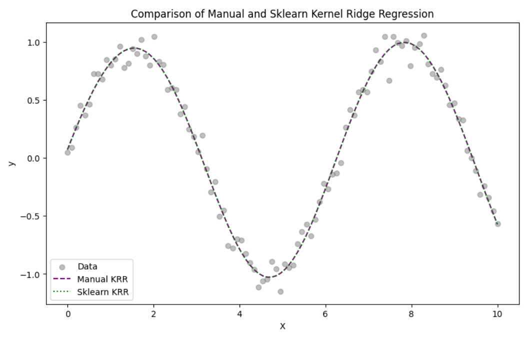

Fig. 12. Comparison of Closed-form and Sklearn Kernel Ridge Regression

They look identical, suggesting that sklearn's implementation utilizes the closed-form solution for kernel ridge regression. Let's put this comparison to the test the same way we did for kernel least squares and try to increase the number of samples in one case, then decrease the number of samples and increase the number of features in the other case:

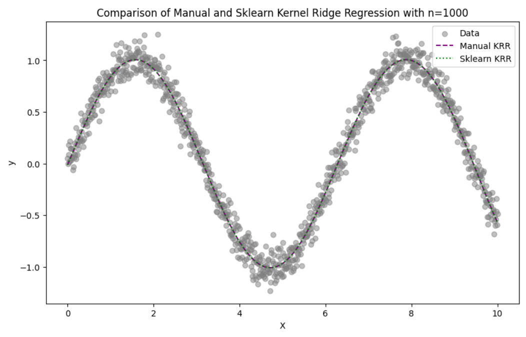

Fig. 13. Comparison of Closed-form and Sklearn Kernel Ridge Regression with $n = 1000$

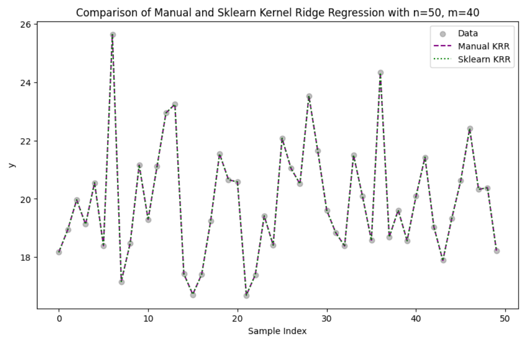

Fig. 14. Comparison of Closed-form and Sklearn Kernel Ridge Regression with $n = 50$, $m = 40$

It seems like increasing n did not have any effect on the model. When we increase the number of features, however, we can see how much both models overfit the data. This is similar to the behaviour exhibited by kernel least squares in Question 2. Comparing MSE values of all 3 cases:

Dataset	**Closed-form MSE**	**Sklearn MSE**
n=100, m=1	0.007460171542	0.007460171542
n=1000, m=1	0.009466902887	0.009466902887
n=50, m=40	1.83643808e-05	1.83643808e-05

TABLE VII

Comparison of MSE for Closed-form and Sklearn KRR across different generated datasets

The drop in MSE is obvious. If we set λ to larger values, we expect the models to not overfit the data as much. Let's test it with $\lambda = 0.1$ and $\lambda = 1$:

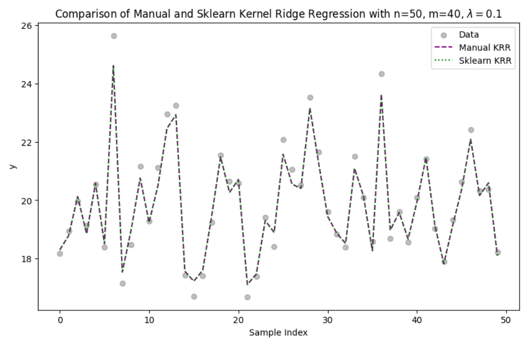

Fig. 15. Comparison of Closed-form and Sklearn Kernel Ridge Regression with $n = 50$, $m = 40$, $\lambda = 0.1$

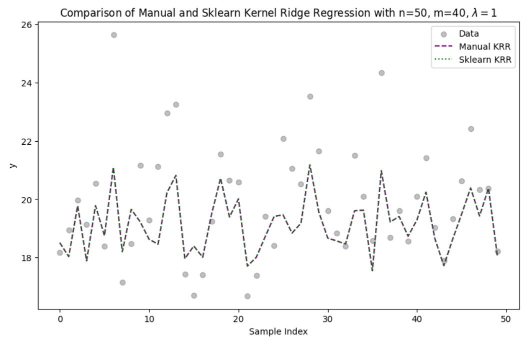

Fig. 16. Comparison of Closed-form and Sklearn Kernel Ridge Regression with $n = 50$, $m = 40$ $\lambda = 1$

We can see that the higher the value of λ, the less the model overfits the data. The MSE, however, increases significantly, as seen in the table below:

λ Value	**Closed-form MSE**	**Sklearn MSE**
$\lambda = 0.1$	0.11084723378	0.11084723378
$\lambda = 1$	2.2035324338	2.20353243378

TABLE VIII

Comparison of MSE for Closed-form and Sklearn KRR with $n = 50$, $m = 40$ for different λ values

Computational Complexity:

For n samples and m features, the computational complexity of computing $\alpha = (K + \lambda I)^{-1}y$ in Kernel Ridge Regression is as follows:

- The computation of the kernel matrix K, where $K_{ij} = k(x_i, x_j)$ has a complexity of $O(n^2 m)$, since each kernel evaluation requires a dot product, and there are $n \times n$ such evaluations. Here, $X \in \mathbb{R}^{n\times m}$, resulting in $K \in \mathbb{R}^{n\times n}$.
- $K + \lambda I$ has a complexity of $O(n^2)$, as it involves adding a scalar to each element of the diagonal of $K \in \mathbb{R}^{n\times n}$.
- $(K + \lambda I)^{-1}$ has a complexity of $O(n^3)$, since $K + \lambda I \in \mathbb{R}^{n\times n}$ due to Gaussian elimination and other transformations.
- The matrix-vector multiplication $(K + \lambda I)^{-1} y$ has a complexity of $O(n^2)$, where $(K + \lambda I)^{-1} \in \mathbb{R}^{n\times n}$ and $y \in \mathbb{R}^{n\times 1}$.
- The total complexity would be $O(n^2 m) + O(n^2) + O(n^3) + O(n^2) = O(n^3)$.

Advantages:

- Kernel ridge regression is able to capture non-linearity in the data, making it applicable to a variety of use-cases.
- The λ regularization parameter allows the model to avoid overfitting the training data, where lower λ values cause larger "underfits" if that's the correct way to word it (probably not).

Disadvantages:

- The high computational complexity of kernel ridge regression, $O(n^3)$, means it is not suitable for very large values of n. It's not exactly scalable.
- Tuning parameters like γ and λ could introduce even higher complexities when running the model.

V. Question 5 👺

Given the following code:

```
X_perceptron = np.array([[2, 3], [3, 3], [4, 5],
    [1, 0], [2, 1], [1, 1]])
y_perceptron = np.array([1, 1, 1, -1, -1, -1])
```

1) Implement the Perceptron algorithm by hand using the dataset provided. Show each step, including the weight updates, and explain the results.
2) Implement the Perceptron algorithm in Python (with the provided dataset X_.npy and y_.npy). Report the accuracy and the number of iterations required for convergence. (The number of iterations refers to how many times the algorithm checks the perceptron condition before it converges.)
3) Implement modifications to the Perceptron algorithm, such as adjustments to the learning rate or weight initialization, to reduce the number of iterations below 1000.
4) Implement the perceptron algorithm and support vector machine in python (with the provided dataset X_train.npy, X_test.npy, y_train.npy, y_test.npy). Report the accuracy on the training and test data of the perceptron and svm algorithm. Draw the decision boundary and mention why there are differences between the perceptron and svm.

Let's do this. Here is the data that we are given:

$$X_{\text{perceptron}} = \begin{bmatrix} 2 & 3 \\ 3 & 3 \\ 4 & 5 \\ 1 & 0 \\ 2 & 1 \\ 1 & 1 \end{bmatrix}$$

$$y_{\text{perceptron}} = [1, 1, 1, -1, -1, -1]$$

We apply the perceptron algorithm to find some weight w and some bias b such that the **sign** of the output matches our target for each of the 6 samples we have. Mathematically:

$$\text{if } y_i(wx_i + b) \le 0$$

Then we have misclassified. In that case, we will **update** the value of w and b in accordance to some learning rate η:

$$w = w + \eta(y_i x_i)$$
$$b = b + \eta(y_i)$$

We have a two-dimensional feature input space, and therefore our $w \in \mathbb{R}^2$. For the purposes of simplicity, let us initialize $w_0 = [0, 0]$, $b = 0$, and our learning rate $\eta = 1$.

Now let's go through each of our 6 samples.

1) Point (2, 3), $y = 1$:
   Pred. $= (1) \times \text{sign}(0 \cdot 2 + 0 \cdot 3 + 0) \le 0$
   Update needed. 👎
   $w = [0, 0] + 1 \cdot 1 \cdot [2, 3] = [2, 3]$
   $b = 0 + 1 \cdot 1 = 1$
   Updated weights: $w = [2, 3]$, $b = 1$
2) Point (3, 3), $y = 1$:
   Pred. $= (1) \times \text{sign}(2 \cdot 3 + 3 \cdot 3 + 1) > 0$
   No update needed. 👍
3) Point (4, 5), $y = 1$:
   Pred. $= (1) \times \text{sign}(2 \cdot 4 + 3 \cdot 5 + 1) > 0$
   No update needed. 👍
4) Point (1, 0), $y = -1$:
   Pred. $= (-1) \times \text{sign}(2 \cdot 1 + 3 \cdot 0 + 1) \le 0$
   Update needed. 👎
   $w = [2, 3] + 1 \cdot (-1) \cdot [1, 0] = [1, 3]$
   $b = 1 + (-1) = 0$
   Updated weights: $w = [1, 3]$, $b = 0$
5) Point (2, 1), $y = -1$:
   Pred. $= (-1) \times \text{sign}(1 \cdot 2 + 3 \cdot 1 + 0) \le 0$
   Update needed. 👎
   $w = [1, 3] + 1 \cdot (-1) \cdot [2, 1] = [-1, 2]$
   $b = 0 + (-1) = -1$
   Updated weights: $w = [-1, 2]$, $b = -1$

6) Point (1, 1), $y = -1$: Pred. $= (-1) \times \text{sign}(-1 \cdot 1 + 2 \cdot 1 - 1) \leq 0$
   Update needed. 👎
   $w = [-1, 2] + 1 \cdot (-1) \cdot [1, 1] = [-2, 1]$
   $b = -1 + (-1) = -2$
   Updated weights: $w = [-2, 1]$, $b = -2$

This is after one iteration. Let's run prediction after this to see what happens.

Point	Label y	Prediction	Correct/Incorrect
(2, 3)	1	-1	Incorrect
(3, 3)	1	-1	Incorrect
(4, 5)	1	-1	Incorrect
(1, 0)	-1	-1	Correct
(2, 1)	-1	-1	Correct
(1, 1)	-1	-1	Correct

TABLE IX
Prediction summary for each point with one iteration of the perceptron algorithm.

It is predicting all -1s with one iteration, given our $w_0 = [0, 0]$ and $b_0 = 0$ initialization. We definitely need to either change the initialization or run it for more iterations, which we will do in the subsequent sub-parts to this question.

We implement the perceptron algorithm in Python as follows:

```python
def our_perceptron_algorithm(X, y, max_iter=1000):

    weights = np.zeros(X.shape[1])
    bias = 0
    iterations = 0
    n_samples = X.shape[0]

    for _ in range(max_iter):
        errors = 0
        for i in range(n_samples):

            if y[i] * (np.dot(X[i], weights) + bias
    ) <= 0:
                weights += y[i] * X[i]
                bias += y[i]
                errors += 1
                iterations += 1
        if errors == 0:
            break

    return weights, bias, iterations
```

Based on this algorithm, we run it on the `X_.npy` and `y_.npy` to get the following:

Metric	Value
Accuracy (%)	100.00%
Iterations for convergence	29
Final weight matrix	$\begin{bmatrix} -2.80066914 \\ -7.83885357 \end{bmatrix}$
Final bias	-1

TABLE X
Perceptron Algorithm Performance Metrics

Therefore, we conclude that given our data, we converge within 29 iterations (much less than 1000) to an accuracy of 100%, with the above weights and bias. The weights here are automatically initialized to $w_0 = [0, 0]$, $b = 0$ and $\eta = 1$. However, we can modify this to pass it into the function as well:

```python
import numpy as np

def perceptron_with_init(X, y, max_iter=1000,
    learning_rate=1.0, weights_init=None, bias_init
    =0):
    weights = weights_init if weights_init is not
    None else np.zeros(X.shape[1])
    bias = bias_init
    iterations = 0
    n_samples = X.shape[0]

    for _ in range(max_iter):
        errors = 0
        for i in range(n_samples):
            if y[i] * (np.dot(X[i], weights) + bias
    ) <= 0:
                weights += learning_rate * y[i] * X
    [i]
                bias += learning_rate * y[i]
                errors += 1
                iterations += 1
        if errors == 0:
            break

    return weights, bias, iterations
```

These modifications are not really necessary since we already converge very quickly but I guess it adds more flexibility for us. Flexibility is good.

For the next part, we will use the above perceptron algorithm and an SVM algorithm that we build from scratch on the new datasets you asked us to use. We will also compare our SVM model that we create from scratch to the SVC implementation from SKLearn.

Here is the class for our SVM model:

```python
class SVM:
    def __init__(self, learning_rate=0.001,
    lambda_param=0.01, n_iters=1000):
        self.learning_rate = learning_rate
        self.lambda_param = lambda_param
        self.n_iters = n_iters
        self.weights = None
        self.bias = None

    def fit(self, X, y):
        n_samples, n_features = X.shape
        self.weights = np.zeros(n_features)
        self.bias = 0

        y_ = np.where(y <= 0, -1, 1)

        for _ in range(self.n_iters):
            for idx, x_i in enumerate(X):
                condition = y_[idx] * (np.dot(x_i,
    self.weights) - self.bias) >= 1
                if condition:
                    self.weights -= self.
    learning_rate * (2 * self.lambda_param * self.
    weights)
                else:
                    self.weights -= self.
    learning_rate * (2 * self.lambda_param * self.
    weights - np.dot(x_i, y_[idx]))
                    self.bias -= self.learning_rate
     * y_[idx]

    def predict(self, X):
        approx = np.dot(X, self.weights) - self.
    bias
        return np.sign(approx)
```

Now let's look at how each algorithm performs:

Algorithm	Train Acc. (%)	Test Acc. (%)
Perceptron	100.00	86.00
SVM (from scratch)	100.00	100.00
SVM (SKLearn)	100.00	100.00

TABLE XI
Train and test accuracies of Perceptron and SVM algorithms.

Our SVM implementation and the SKLearn SVM implementation perform well on both training and testing. The fact that the training accuracy goes to 100% for the perceptron algorithm also suggests that that converges as well, but it does not really do too well when it comes to testing data. Let's visualize the decision boundaries now.

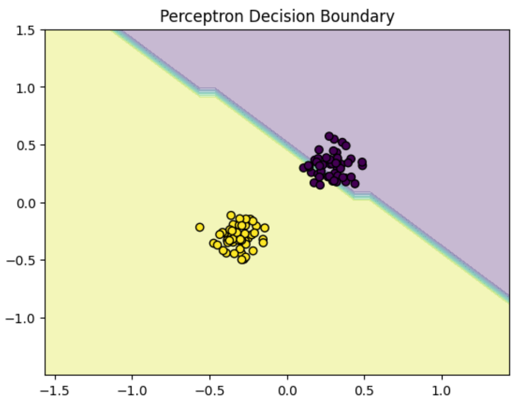

Fig. 17. Decision boundary for the perceptron algorithm, with testing data plotted as well.

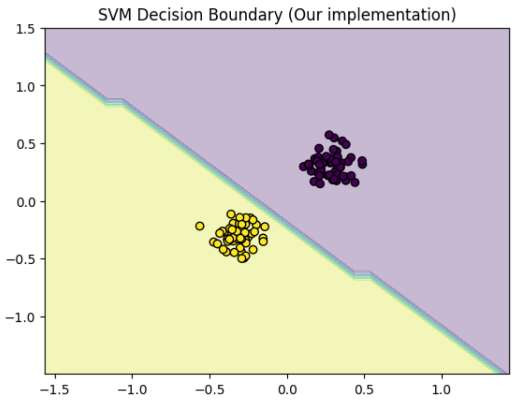

Fig. 18. Decision boundary for our SVM algorithm, with testing data plotted as well.

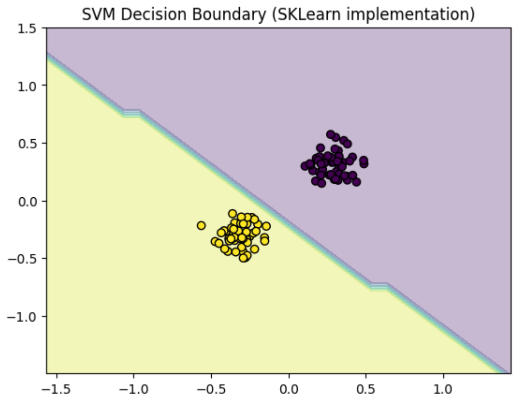

Fig. 19. Decision boundary for the SKLearn SVM algorithm, with testing data plotted as well.

As you can see in Fig. 17, one of the classes slightly overlaps with the decision boundary itself, which is why the testing accuracy is not as high. Since the perceptron algorithm does not maximize the margin between a class and the decision boundary during training, it does not leave a margin of error for testing data the same way SVM does. SVM performs much better exactly because of this maximal distance. We can see that our implementation and the SKLearn implementation result in the same decision boundary as well, which is always good. We're doing things right.

I think that's it for this question right?

VI. Question 6 🦦

Given the following code:

```
X = np.array([[2, 3], [1, 1], [4, 4], [5, 5], [6,
    7], [7, 8]])
y = np.array([1, 1, -1, -1, -1, 1])
```

1) Predict the class of the point (3, 4) using k-NN for k=1,3,5 by calculating the Euclidean distance manually and performing majority voting.
2) Implement the k-NN algorithm on the dataset above in Python using scikit-learn for k=1,3,5
3) Implement the k-NN algorithm on the iris dataset for k=1,3,5

```
iris = load_iris()
X = iris.data
y = iris.target # Split the data into training and
    testing sets
X_train, X_test, y_train, y_test = train_test_split
    (X, y, test_size=0.3, random_state=42)
```

K nearest-neighbours (k-NN) classifies a data point based on the majority class seen amongst its, you guessed it, ”K nearest-neighbours”. There are various ways to measure distance, but we will stick to Euclidian Distance for this scenario. Let $p \in \mathbb{R}^n$ and $q \in \mathbb{R}^n$ be any two data points with n-dimensions. The euclidian distance $d(p, q)$ between p and q is calculated as follows:

$$d(p,q) = \sqrt{\sum_{i=0}^{n}(p_i - q_i)^2}$$

where p_i and q_i are coordinates of the points p and q, respectively. Figure 20 plots how the data is distributed.

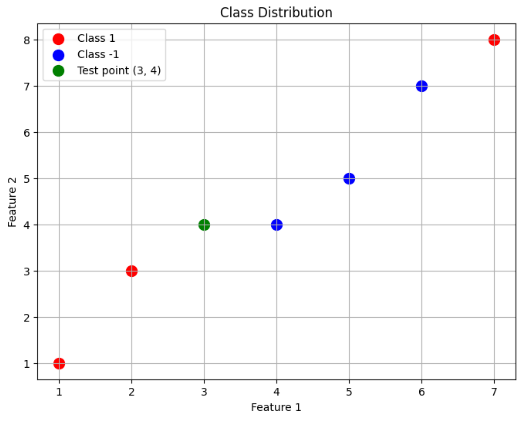

Fig. 20. Class distribution of the given data

Before calculating, we can already see that k-NN will not be too confident here, since the test point p conveniently sits right in the middle of the two classes. Other than that, the two clusters can be clearly seen, meaning k-NN will perform well.

Given the dataset X and their classes y, the steps to classify the point $p = (3,4)$ will be as follows:

1) Calculate $d(p, x_i)$ for all $x_i \in X$
2) Sort the distances from shortest to longest
3) Classify p based on majority voting for the top k-classes, where $k \in \{1,3,5\}$

Let us calculate the Euclidean distance $d(p, x_i)$ between each $x_i \in X$ and point $p = (3,4)$.

For $x_1 = (2,3)$:

$$d((3,4),(2,3)) = \sqrt{(3-2)^2 + (4-3)^2} = \sqrt{1+1} = \sqrt{2}$$

For $x_2 = (1,1)$:

$$d((3,4),(1,1)) = \sqrt{(3-1)^2 + (4-1)^2} = \sqrt{4+9} = \sqrt{13}$$

For $x_3 = (4,4)$:

$$d((3,4),(4,4)) = \sqrt{(3-4)^2 + (4-4)^2} = \sqrt{1+0} = 1$$

For $x_4 = (5,5)$:

$$d((3,4),(5,5)) = \sqrt{(3-5)^2 + (4-5)^2} = \sqrt{4+1} = \sqrt{5}$$

For $x_5 = (6,7)$:

$$d((3,4),(6,7)) = \sqrt{(3-6)^2 + (4-7)^2} = \sqrt{9+9} = \sqrt{18}$$

For $x_6 = (7,8)$:

$$d((3,4),(7,8)) = \sqrt{(3-7)^2 + (4-8)^2} = \sqrt{16+16} = \sqrt{32}$$

Sorting them out in a table, from shortest to longest distance:

x_i	$d(p, x_i)$	y_i
$(4,4)$	1.000	-1
$(2,3)$	1.414	1
$(5,5)$	2.236	-1
$(1,1)$	3.606	1
$(6,7)$	4.243	-1
$(7,8)$	5.657	1

Our majority voting predictions are as follows:

- For $k = 1$, the top 1-class is -1.
  Prediction: -1.
- For $k = 3$, the top 3-classes are $\{-1,1,-1\}$.
  Prediction: -1.
- For $k = 5$, the top 5-classes are $\{-1,1,-1,1,-1\}$.
  Prediction: -1.

As mentioned earlier, this prediction is not a very confident one. The closest data points alternate from one class to another. If we simply took k to be an even number, the model would've tied between the two classes.

Let's try this with sklearn's implementation of kNN and see how it performs:

```python
from sklearn.neighbors import KNeighborsClassifier
classifiers = {k: KNeighborsClassifier(n_neighbors=
    k) for k in [1, 3, 5]}
fits = {k: clf.fit(X, y) for k, clf in classifiers.
    items()}
pred = {k: clf.predict([point])[0] for k, clf in
    fits.items()}
```

- For $k = 1$, **Prediction: -1**.
- For $k = 3$, **Prediction: -1**.
- For $k = 5$, **Prediction: -1**.

This aligns with our manual implementation. For the sake of testing, let's set $k \in \{2,4,6\}$ and see how sklearn handles ties in majority voting:

- For $k = 2$, **Prediction: -1**.
- For $k = 4$, **Prediction: -1**.
- For $k = 6$, **Prediction: -1**.

Interesting. After some googling, it appears that sklearn handles ties by choosing the class of the closest point to its target (i.e. the prediction when $k = 1$). We can change the weights parameter from its default 'uniform' to 'distance', which just means that the further a neighbor is, the less influence it has on the prediction. Let's see:

```python
from sklearn.neighbors import KNeighborsClassifier
classifiers = {k: KNeighborsClassifier(n_neighbors=
    k, weights='distance') for k in [1, 3, 5]}
fits = {k: clf.fit(X, y) for k, clf in classifiers.
    items()}
pred = {k: clf.predict([point])[0] for k, clf in
    fits.items()}
```

- For $k = 1$, **Prediction: -1**.
- For $k = 3$, **Prediction: -1**.
- For $k = 5$, **Prediction: -1**.

Same exact thing. I guess this would be more interesting with more complex data. Convenient that we will be looking at this iris dataset now. Iris contains 4 features: sepal length, sepal width, petal length, and petal width. Let's visualize the training dataset using t-SNE:

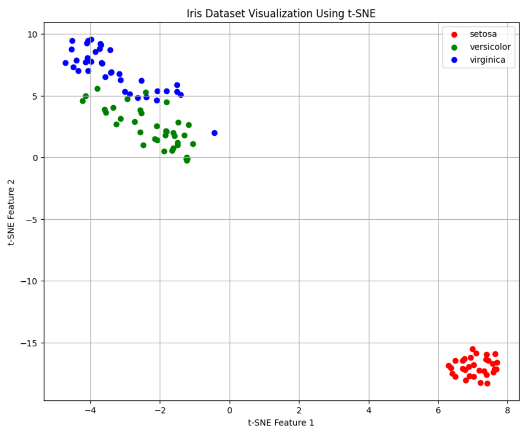

Fig. 21. Class distribution of Iris

We can see that the Setosa class should be very easy to classify. The kNN model might run into issues with differentiating between Versicolor and Virginica, but we'll see. Let's implement kNN using sklearn for $k = \{1, 3, 5\}$ and set the weights parameter to 'uniform':

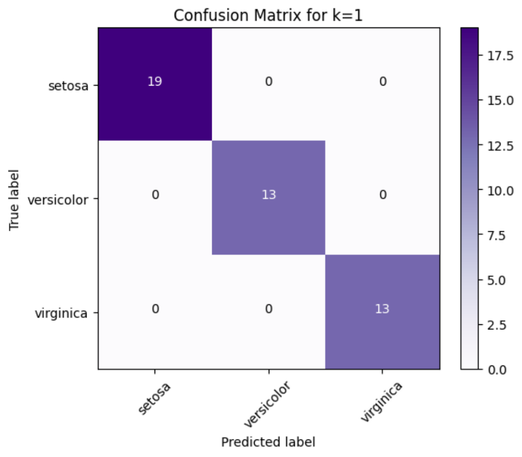

Fig. 22. Confusion Matrix for $k = 1$

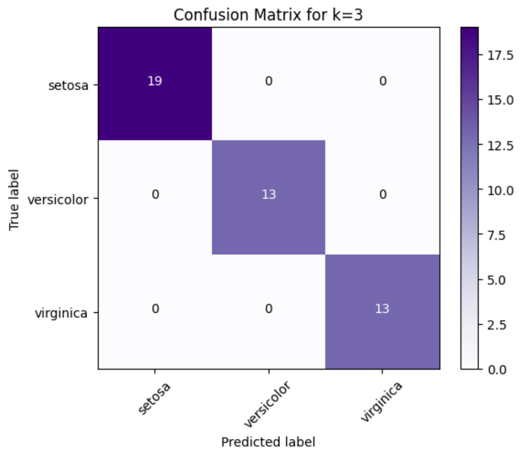

Fig. 23. Confusion Matrix for $k = 3$

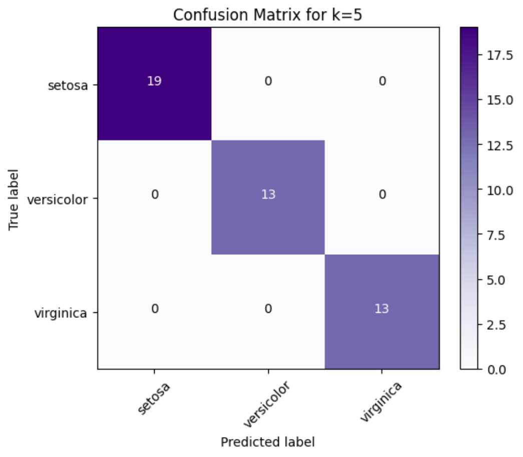

Fig. 24. Confusion Matrix for $k = 5$

Perfect accuracies across the board. Let's test setting the weights parameter to 'distance':

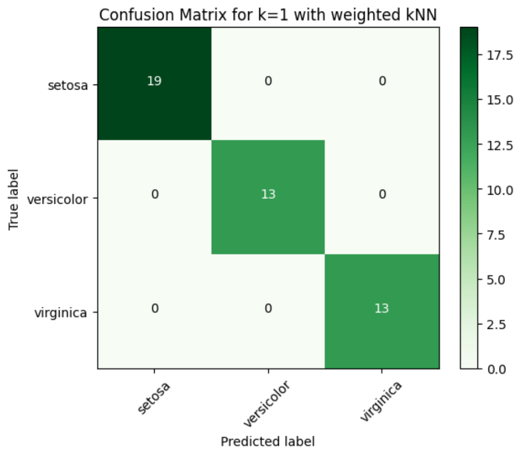

Fig. 25. Confusion Matrix for $k = 1$ with weighted kNN

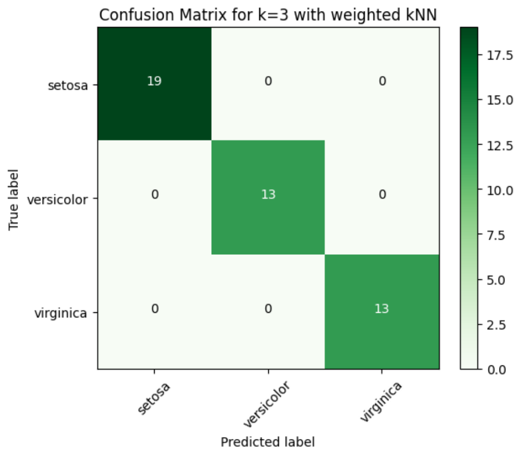

Fig. 26. Confusion Matrix for $k = 3$ with weighted kNN

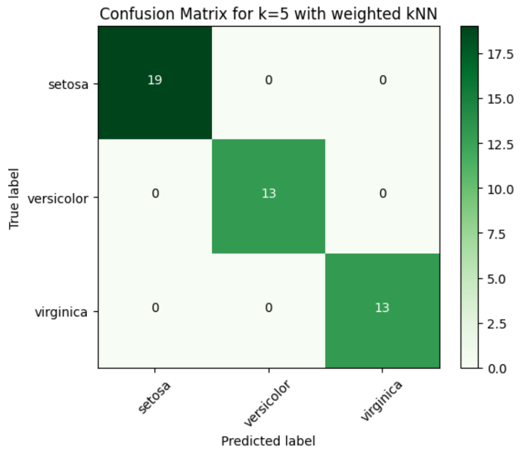

Fig. 27. Confusion Matrix for $k = 5$ with weighted kNN

Perfect accuracies, again. Maybe a more complicated dataset would actually show us differences in performance, but we'll stop here.

VII. Question 7 👺

Given the code:

```python
np.random.seed(42)
n_samples = 1000
X_car_travel = np.random.rand(n_samples, 3) * 100 # Features: traffic density, road type, and distance
y_car_travel = (X_car_travel[:, 0] * 0.5 + X_car_travel[:, 1] * 0.3 + X_car_travel[:, 2] * 1.2) +
np.random.randn(n_samples) * 5 # Simulated travel time
# Standardize the features
X_car_travel_scaled = scaler.fit_transform(X_car_travel)
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_car_travel_scaled, y_car_travel, test_size=0.2,
random_state=42)
```

1) Implement k-NN regression in Python and explore the impact of different values of k (e.g., k = 1, 3, 5, 7) on performance. In addition to using the arithmetic mean to predict the regression output, also implement the geometric and harmonic means as alternative aggregation methods. Compare the performance of the three means (MSE, R^2) across different values of k by evaluating the model on the dataset provided. Include an analysis of when one mean might outperform the others. Provide both numerical results and visualizations to support your findings.
2) (Bonus) Explore 5 different real datasets and compare the performance of the arithmetic, geometric and harmonic mean in terms of the MSE and R^2. Please provide an explanation of when does the geometric or harmonic mean provide a better performance.

Let's first look at what the data actually looks like.

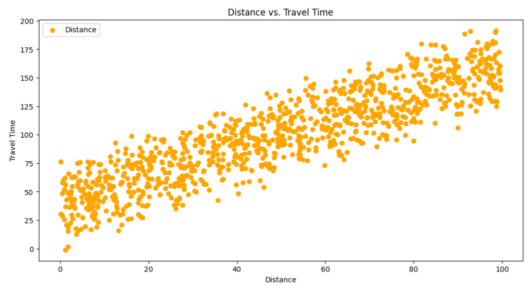

Fig. 28. Plotting travel time vs. distance.

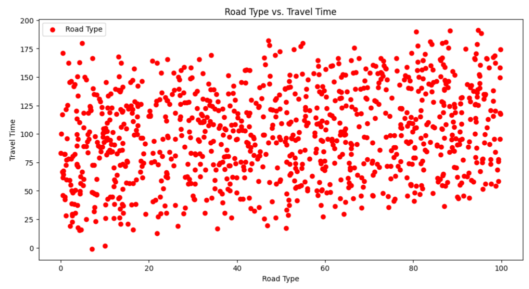

Fig. 29. Plotting travel time vs. road type.

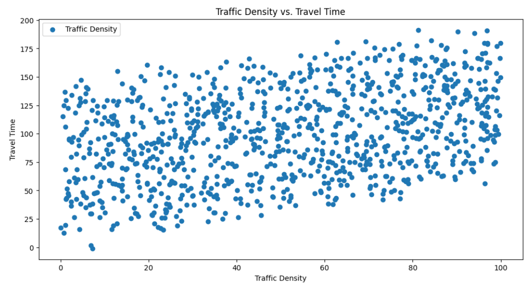

Fig. 30. Plotting travel time vs. traffic density.

Very nice. The data definitely looks like something. There seems to be a general positive correlation between all three of them, but in the case of distance, **naturally** we would expect to travel for longer periods of time if we are going further distances. The road type doesn't seem to have too much of a correlation at all, it's a little random in fact, but the traffic density also seems to have a loose positive slope.

Now let's implement KNN for this. First, let's implement something for computing the different means:

```python
def get_mean(y_values, mean_type='arithmetic'):
    if mean_type == 'arithmetic':
        return np.mean(y_values)
    elif mean_type == 'geometric':
        return gmean(y_values) if np.all(y_values > 0) else np.nan
    elif mean_type == 'harmonic':
        return hmean(y_values) if np.all(y_values > 0) else np.nan
    else:
        raise ValueError("Invalid mean type. Choose from 'arithmetic', 'geometric', or 'harmonic'.")
```

This is what we will pass into the KNN function. The KNN regressor is built as follows:

```python
def knn_regressor(X_train, y_train, X_test, k=3, mean_type='arithmetic'):
    y_pred = []
    for x_test in X_test:

        distances = np.linalg.norm(X_train - x_test, axis=1)

        nearest_neighbors = np.argsort(distances)[:k]

        y_pred.append(get_mean(y_train[nearest_neighbors], mean_type))

    return np.array(y_pred)
```

Now we can evaluate the performance depending on the mean-type that we pass in. First, for each of the mean types and each k value, we have the following MSE and R^2 values.

k	Mean Type	MSE	R^2
1	Arithmetic	69.0002	0.9574
1	Geometric	69.0002	0.9574
1	Harmonic	69.0002	0.9574
3	Arithmetic	53.9841	0.9667
3	Geometric	52.5384	0.9676
3	Harmonic	51.9673	0.9679
5	Arithmetic	47.2429	0.9708
5	Geometric	46.4082	0.9713
5	Harmonic	46.2766	0.9714
7	Arithmetic	45.1791	0.9721
7	Geometric	44.9684	0.9722
7	Harmonic	46.7364	0.9711

TABLE XII
Comparison of MSE and R^2 for Different Mean Types and Values of k

Here are some observations from the table before we move into some visualizations.

- For $k = 1$, the arithmetic, geometric and harmonic means all result in the same MSE and R^2 values, but they also perform the worst in terms of these two metrics. Higher k values generally give us lower MSE and better R^2.
- Overall, the best k value and mean-type pairing comes from $k = 7$ and the geometric mean, with an MSE of 44.9684 and an R^2 of 0.9722. This is only marginally higher than some of the other approaches though.
- We can see that overall, the geometric means performs the best terminally, which can be seen in the plots more clearly. Although all the mean types start at the same values for $k = 1$, as we increase k, the geometric mean *marginally* performs better.
- The harmonic mean, on the other hand, is the most unstable and actually performs worse than the other two as we increase k.

The below are two plots that visualize this. One uses the R^2 as the y-axis, and the other uses the MSE.

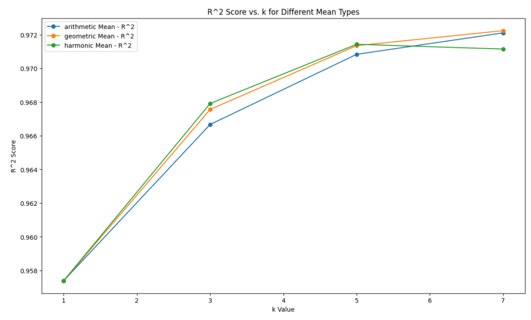

Fig. 31. R^2 values for each of the different mean types as we increase k from 1 to 7.

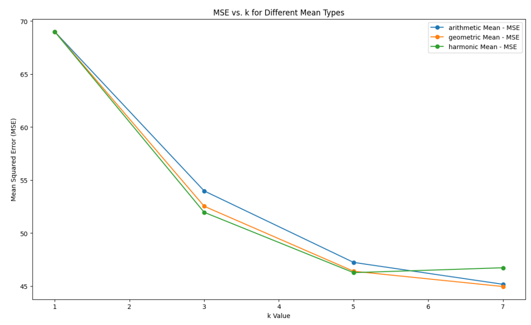

Fig. 32. MSE values for each of the different mean types as we increase k from 1 to 7.

Let's have one more visualization before we get into more analytical stuff. Here is the prediction for each of the k values when compared to a "perfect" prediction:

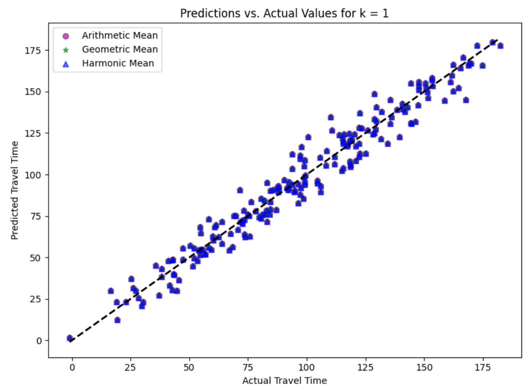

Fig. 33. Predicted vs. actual values for $k = 1$.

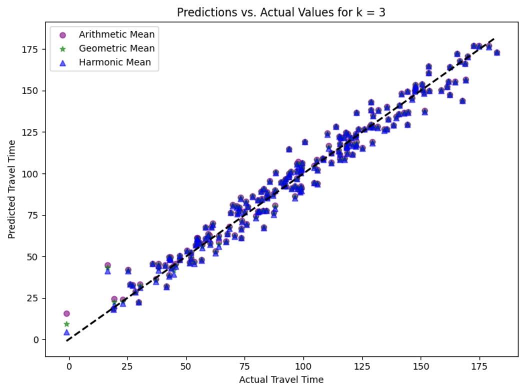

Fig. 34. Predicted vs. actual values for $k = 3$.

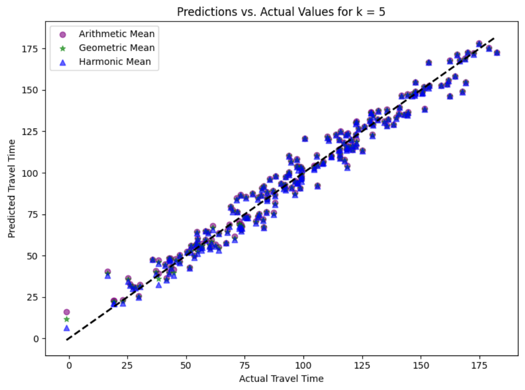

Fig. 35. Predicted vs. actual values for $k = 5$.

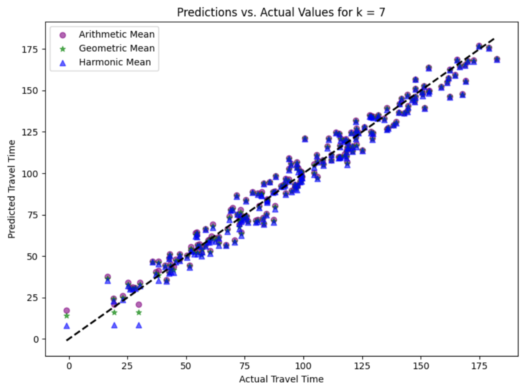

Fig. 36. Predicted vs. actual values for $k = 7$.

Let us analyze this further now. First of all, let's compare what the means each mean / represent.

- **Arithmetic mean**: Often performs well as it provides a balanced estimate by treating all nearest neighbors equally.
- **Geometric mean**: Tends to be more sensitive to outliers and is only defined for positive values; however, it might perform better when distances are small and neighbors are similar.
- **Harmonic mean**: Weighted towards smaller values and may excel when the dataset has high variability within clusters.

Considering some theory, the below are when it is recommended to use each of the different mean types.

Mean Type	**Characteristics**	**When to Use**
Arithmetic Mean	Sensitive to outliers	Symmetric data, larger k values, less outliers
Geometric Mean	Robust to high-value outliers, emphasizes smaller values	Skewed data, moderate k values
Harmonic Mean	Strong focus on small values, robust to high-value outliers	Data skewed to lower values, lower k values

TABLE XIII
SUMMARY OF CHARACTERISTICS AND WHEN TO USE EACH OF THE DIFFERENT MEAN TYPES.

For each of the mean types, we can observe the following things based on the performance on the data:

- **Arithmetic mean**: At $k = 1$, MSE = 69.0002 and $R^2 = 0.9574$. The MSE value is relatively high, meaning that with a single neighbor, outliers can very strongly affect predictions. However, with $k = 7$, the MSE drops to 45.1791 and R^2 improves to 0.9721, showing that high k values reduces the impact of outliers.
- **Geometric mean**: At $k = 3$, the MSE of the geometric mean approach is lower than the arithmetic mean approach. At $k = 7$, it slightly outperforms it. We can see that for *moderate* values, it performs well because there is a decent balance between outlier sensitivity and central tendency.
- **Harmonic mean**: For $k = 3$, the harmonic mean performs the best of all three. At the highest k value of 7, the performance actually drops, suggesting that it does not benefit from larger k values.

Shall we look at some real-world datasets now? We will look at 5 real-world datasets that are available through SKLearn:

- Ames housing dataset
- Diabetes dataset
- California housing dataset
- Wine quality dataset
- Concrete compressive strength dataset

We will begin by checking the MSE and R^2 values for each of the different types of means depending on the k values. We will continue to use the same k values as in the previous part for the sake of consistency. Although the plots are also available upon request, they really do take up a lot of the page and this is already extremely long.

The results are in!

k	Mean Type	MSE	R^2
1	Arithmetic	1.5797e+09	0.79400
1	Geometric	1.5797e+09	0.79400
1	Harmonic	1.5797e+09	0.79400
3	Arithmetic	1.6187e+09	0.78900
3	Geometric	1.5490e+09	0.79800
3	Harmonic	1.5331e+09	0.80010
5	Arithmetic	1.4687e+09	0.80850
5	Geometric	1.4569e+09	0.81010
5	Harmonic	1.4886e+09	0.80590
7	Arithmetic	1.5663e+09	0.79580
7	Geometric	1.5857e+09	0.79330
7	Harmonic	1.6402e+09	0.78620

TABLE XIV
Comparison of MSE and R^2 for Different Mean Types and Values of k for the Ames housing dataset.

k	Mean Type	MSE	R^2
1	Arithmetic	5191.2	0.02020
1	Geometric	5191.2	0.02020
1	Harmonic	5191.2	0.02020
3	Arithmetic	3364.4	0.36500
3	Geometric	3432.7	0.35210
3	Harmonic	3669.0	0.30750
5	Arithmetic	3019.1	0.43020
5	Geometric	3219.4	0.39240
5	Harmonic	3629.3	0.31500
7	Arithmetic	2987.0	0.43620
7	Geometric	3216.7	0.39290
7	Harmonic	3713.3	0.29910

TABLE XV
Comparison of MSE and R^2 for Different Mean Types and Values of k for the Diabetes dataset.

k	Mean Type	MSE	R^2
1	Arithmetic	0.65450	0.50050
1	Geometric	0.65450	0.50050
1	Harmonic	0.65450	0.50050
3	Arithmetic	0.46110	0.64810
3	Geometric	0.45840	0.65020
3	Harmonic	0.46930	0.64190
5	Arithmetic	0.43380	0.66900
5	Geometric	0.43720	0.66640
5	Harmonic	0.45710	0.65110
7	Arithmetic	0.42280	0.67740
7	Geometric	0.43140	0.67080
7	Harmonic	0.45750	0.65090

TABLE XVI
Comparison of MSE and R^2 for Different Mean Types and Values of k for the California housing dataset.

k	Mean Type	MSE	R^2
1	Arithmetic	0.56940	0.26480
1	Geometric	0.56940	0.26480
1	Harmonic	0.56940	0.26480
3	Arithmetic	0.49480	0.36110
3	Geometric	0.50190	0.35200
3	Harmonic	0.51620	0.33340
5	Arithmetic	0.47700	0.38410
5	Geometric	0.48110	0.37880
5	Harmonic	0.49500	0.36080
7	Arithmetic	0.47450	0.38740
7	Geometric	0.47750	0.38350
7	Harmonic	0.49470	0.36120

TABLE XVII
Comparison of MSE and R^2 for Different Mean Types and Values of k for the Wine Quality dataset.

k	Mean Type	MSE	R^2
1	Arithmetic	73.948	0.71300
1	Geometric	73.948	0.71300
1	Harmonic	73.948	0.71300
3	Arithmetic	69.458	0.73050
3	Geometric	71.154	0.72390
3	Harmonic	76.615	0.70270
5	Arithmetic	75.513	0.70700
5	Geometric	80.001	0.68950
5	Harmonic	89.441	0.65290
7	Arithmetic	76.509	0.70310
7	Geometric	82.320	0.68050
7	Harmonic	94.636	0.63270

TABLE XVIII
Comparison of MSE and R^2 for Different Mean Types and Values of k for the Concrete compressive strength dataset.

In all cases, we can see that the geometric, harmonic and arithmetic means are equal when $k = 1$. They diverge and follow the same sort of trends when we increase k, but the performance is significantly worse on these datasets as they are naturally more complex and harder to predict with.

VIII. Question 8 🦦

Given the following dataset:

Day	Outlook	Temp	Humidity	Wind	Tennis?
1	Sunny	Hot	High	Weak	No
2	Sunny	Hot	High	Strong	No
3	Overcast	Hot	High	Weak	Yes
4	Rain	Mild	High	Weak	Yes
5	Rain	Cool	Normal	Weak	Yes
6	Rain	Cool	Normal	Strong	No
7	Overcast	Cool	Normal	Strong	Yes
8	Sunny	Mild	High	Weak	No
9	Sunny	Cool	Normal	Weak	Yes
10	Rain	Mild	Normal	Weak	Yes
11	Sunny	Mild	Normal	Strong	Yes
12	Overcast	Mild	High	Strong	Yes
13	Overcast	Hot	Normal	Weak	Yes
14	Rain	Mild	High	Strong	No

1) Construct the decision tree using the information gain and show your detailed calculations.

2) Construct the decision tree using the gini coefficient and show your detailed calculations
3) In this question, you will implement three decision tree algorithms and compare their performance on the Yeast dataset:
   a) **ID3 (Information Gain)** – Implemented using sklearn based on **entropy** (information gain).
   b) **CART (Classification and Regression Trees)** – Implemented using sklearn based on **Gini coefficient**.
   c) **Decision Tree with Tsallis Entropy** – A custom decision tree where the split is based on **Tsallis entropy**, with a tunable parameter q. Explore different values of q and report the best performance for q in the range of [0.3,8].

```python
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
import pandas as pd
# Load the yeast dataset from OpenML
yeast_data = fetch_openml(name="yeast", version=1)
X = pd.DataFrame(yeast_data.data, columns=yeast_data.feature_names)
y = yeast_data.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

```

4) (Bonus) **Optimal q Selection**: Propose and implement a method for selecting the optimal value of q based on the dataset properties and explore it for 10 different datasets.

Let's start by defining **entropy** and **information gain** in a binary classification problem. Entropy measures the impurity of a dataset, whilst information gain measures the reduction in entropy achieved by splitting the dataset based on a specific attribute. Let p_i be the probability of class i in a dataset Y, then the entropy $H(Y)$ is defined as:

$$H(Y) = -\sum_{i=1}^{n} p_i \log_2(p_i)$$

In the context of decision trees for binary classification tasks, the entropy of a node N is defined as:

$$H(N) = -(p_0 \log_2(p_0) + p_1 \log_2(p_1))$$

The information gain $IG(S, A)$ for a node N and an attribute A is calculated as:

$$IG(N, A) = H(N) - \sum_{i \in \text{nodes}} w_i \cdot H(N_i)$$

where:

- $H(N)$ is the entropy of the parent note,
- w_i is the weight, defined as the fraction of samples in node i from the parent node,
- $H(N_i)$ is the entropy of subset nodes.

Starting at the root node of a decision tree, the information gain of every possible split is calculated. The split with the highest information gain is chosen, and then the process repeats until one of the conditions is met:

- $H(N) = 0$ for a node N
- Maximum depth is reached
- Information gain is below a specified threshold for all possible splits
- Number of samples in a node is below a specified threshold

First, we calculate the **entropy of the root node** (i.e. the full dataset). For the target variable "Tennis?" are 9 instances of "Yes" and 5 instances of "No":

$$H(N_{\text{root}}) = -\left(\frac{9}{14}\log_2(\frac{9}{14}) + \frac{5}{14}\log_2(\frac{5}{14})\right) \approx 0.940$$

Next, we calculate the information gain for splits at each attribute: Outlook, Temperature, Humidity, and Wind. Starting with **Outlook**:

- Sunny: 5 instances (3 "No", 2 "Yes")
$$H(N_{\text{Sunny}}) = -\left(\frac{3}{5}\log_2(\frac{3}{5}) + \frac{2}{5}\log_2(\frac{2}{5})\right) \approx 0.971$$
- Overcast: 4 instances (all "Yes")
$$H(N_{\text{Overcast}}) = 0$$
- Rain: 5 instances (2 "No", 3 "Yes")
$$H(N_{\text{Rain}}) = -\left(\frac{3}{5}\log_2(\frac{3}{5}) + \frac{2}{5}\log_2(\frac{2}{5})\right) \approx 0.971$$

The weighted entropy for Outlook is:

$$H(\text{Outlook}) = \frac{5}{14} \times 0.971 + \frac{4}{14} \times 0 + \frac{5}{14} \times 0.971 \approx 0.693$$

Thus, the information gain for Outlook is:

$$IG(N, \text{Outlook}) = 0.940 - 0.693 = 0.247$$

Next, **Temperature**:

- Hot: 4 instances (2 "No", 2 "Yes")
$$H(N_{\text{Hot}}) = -\left(\frac{2}{4}\log_2(\frac{2}{4}) + \frac{2}{4}\log_2(\frac{2}{4})\right) = 1$$
- Mild: 6 instances (2 "No", 4 "Yes")
$$H(N_{\text{Mild}}) = -\left(\frac{4}{6}\log_2(\frac{4}{6}) + \frac{2}{6}\log_2(\frac{2}{6})\right) \approx 0.918$$
- Cool: 4 instances (1 "No", 3 "Yes")
$$H(N_{\text{Cool}}) = -\left(\frac{3}{4}\log_2(\frac{3}{4}) + \frac{1}{4}\log_2(\frac{1}{4})\right) \approx 0.811$$

The weighted entropy for Temperature is:

$$H(\text{Temp}) = \frac{4}{14} \times 1 + \frac{6}{14} \times 0.918 + \frac{4}{14} \times 0.811 \approx 0.911$$

Thus, the information gain for Temperature is:

$$IG(N, \text{Temp}) = 0.940 - 0.911 = 0.029$$

Next, **Humidity**:

- High: 7 instances (4 ”No”, 3 ”Yes”)

$$H(N_{\text{High}}) = -\left(\frac{4}{7}\log_2(\frac{4}{7}) + \frac{3}{7}\log_2(\frac{3}{7})\right) \approx 0.985$$

- Normal: 7 instances (1 ”No”, 6 ”Yes”)

$$H(N_{\text{Normal}}) = -\left(\frac{6}{7}\log_2(\frac{6}{7}) + \frac{1}{7}\log_2(\frac{1}{7})\right) \approx 0.592$$

The weighted entropy for Humidity is:

$$H(\text{Humidity}) = \frac{7}{14} \times 0.985 + \frac{7}{14} \times 0.592 \approx 0.789$$

Thus, the information gain for Humidity is:

$$IG(N, \text{Humidity}) = 0.940 - 0.789 = 0.151$$

Lastly, **Wind**:

- Weak: 8 instances (2 ”No”, 6 ”Yes”)

$$H(N_{\text{Weak}}) = -\left(\frac{6}{8}\log_2(\frac{6}{8}) + \frac{2}{8}\log_2(\frac{2}{8})\right) \approx 0.811$$

- Strong: 6 instances (3 ”No”, 3 ”Yes”)

$$H(N_{\text{Strong}}) = -\left(\frac{3}{6}\log_2(\frac{3}{6}) + \frac{3}{6}\log_2(\frac{3}{6})\right) = 1$$

The weighted entropy for Wind is:

$$H(\text{Wind}) = \frac{8}{14} \times 0.811 + \frac{6}{14} \times 1 \approx 0.892$$

Thus, the information gain for Wind is:

$$IG(N, \text{Wind}) = 0.940 - 0.892 = 0.048$$

The information gain for each attribute from the root node is as follows:

- $IG(N, \text{Outlook}) = 0.247$
- $IG(N, \text{Temp}) = 0.029$
- $IG(N, \text{Humidity}) = 0.152$
- $IG(N, \text{Wind}) = 0.048$

We will split the root node at **Outlook**, as it results in the highest information gain of 0.247. From earlier calculations, we know that the **Overcast** node has an entropy of 0, making it our first official leaf node. Our (partial) decision tree now looks like this:

Outlook?

- Overcast: *Play Tennis*
- Sunny: ...
- Rain: ...

We now have two subtrees to continue our calculations. From this point onwards, they will be more concise to avoid clutter. Starting with the **Sunny** subtree, our updated data is as follows:

Day	Outlook	Temp	Humidity	Wind	Tennis?
1	Sunny	Hot	High	Weak	No
2	Sunny	Hot	High	Strong	No
8	Sunny	Mild	High	Weak	No
9	Sunny	Cool	Normal	Weak	Yes
11	Sunny	Mild	Normal	Strong	Yes

In this root node, there are 2 instances of ”Yes” and 3 instances of ”No”:

$$H(N) = -\left(\frac{2}{5}\log_2\frac{2}{5} + \frac{3}{5}\log_2\frac{3}{5}\right) \approx 0.971$$

Now, we calculate the information gain for each attribute: Temperature, Humidity, and Wind:

- **Temperature**:
  - $H(N_{\text{Hot}}) = 0$
  - $H(N_{\text{Mild}}) = 1$
  - $H(N_{\text{Cool}}) = 0$
  - Weighted entropy $H(\text{Temperature}) = 0.4$
  - $IG(N, \text{Temperature}) = 0.971 - 0.4 = 0.571$
- **Humidity**:
  - $H(N_{\text{High}}) = 0$
  - $H(N_{\text{Normal}}) = 0$
  - Weighted entropy $H(\text{Humidity}) = 0$
  - $IG(N, \text{Humidity}) = 0.971 - 0 = 0.971$
- **Wind**:
  - $H(N_{\text{Weak}}) = 0.918$
  - $H(N_{\text{Strong}}) = 1$
  - Weighted entropy $H(\text{Wind}) = 0.951$
  - $IG(N, \text{Wind}) = 0.971 - 0.951 = 0.02$

We will split the root node at **Humidity**, as it results in the highest information gain of 0.971. We can also see that the entropies for both Humidity conditions are 0, giving us our next set of leaf nodes. Our updated (partial) decision tree now looks like this:

Outlook?

- Overcast: *Play Tennis*
- Sunny: **Humidity?**
  - High: *Dont play Tennis*
  - Normal: *Play Tennis*
- Rain: ...

Going back to our **Rain** subtree, our updated data is as follows:

Day	Outlook	Temp	Humidity	Wind	Tennis?
4	Rain	Mild	High	Weak	Yes
5	Rain	Cool	Normal	Weak	Yes
6	Rain	Cool	Normal	Strong	No
10	Rain	Mild	Normal	Weak	Yes
14	Rain	Mild	High	Strong	No

In this root node, there are 3 instances of ”Yes” and 2 instances of ”No”:

$$H(N) = -\left(\frac{3}{5}\log_2\frac{3}{5} + \frac{2}{5}\log_2\frac{2}{5}\right) \approx 0.971$$

Now, we calculate the information gain for each attribute: Temperature, Humidity, and Wind:

- **Temperature**:
  - $H(N_{\text{Mild}}) = 0.918$
  - $H(N_{\text{Cool}}) = 1$
  - Weighted entropy $H(\text{Temperature}) = 0.951$
  - $IG(N, \text{Temperature}) = 0.971 - 0.951 = 0.02$
- **Humidity**:
  - $H(N_{\text{High}}) = 1$
  - $H(N_{\text{Normal}}) = 0.918$
  - Weighted entropy $H(\text{Humidity}) = 0.951$
  - $IG(N, \text{Humidity}) = 0.971 - 0.951 = 0.02$
- **Wind**:
  - $H(N_{\text{Weak}}) = 0$
  - $H(N_{\text{Strong}}) = 0$
  - Weighted entropy $H(\text{Wind}) = 0$
  - $IG(N, \text{Wind}) = 0.971 - 0 = 0.971$

We will split the root node at **Wind**, as it results in the highest information gain of 0.971. We can also see that the entropies for both Wind conditions are 0, giving us our final set of leaf nodes. Our updated (full) decision tree now looks like this:

Outlook?

- Overcast: *Play Tennis*
- Sunny: **Humidity?**
  - High: *Dont play Tennis*
  - Normal: *Play Tennis*
- Rain: **Wind?**
  - Weak: *Play Tennis*
  - Strong: *Dont play Tennis*

Finally, we can visualize the decision tree we built:

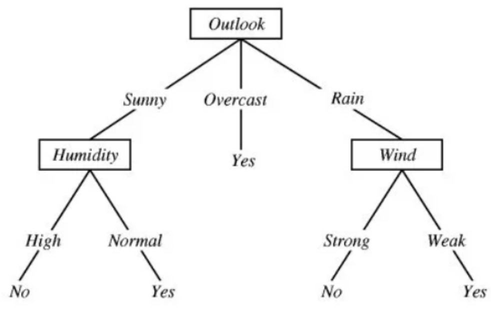

Fig. 37. Decision Tree using Entropy and Information Gain

Let's compare it with sklearn's DecisionTreeClassifier() implementation:

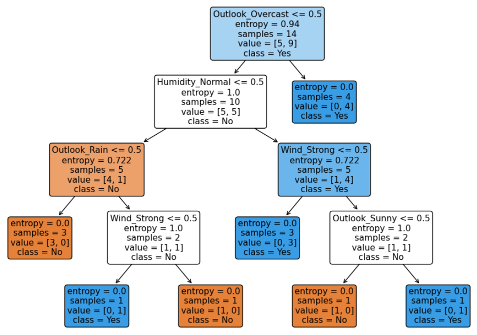

Fig. 38. Entropy implementation using sklearn

We can see that rather than splitting each condition into it's own node, the "best" condition of an attribute was used for a binary split. Because of this, sklearn's decision tree has a much larger depth than the one we built.

Now we're going to do that all over again, but using **gini coefficient** instead of entropy and information gain. Gini coefficient is another way to measure impurity in a dataset. For a node N with c classes, gini coefficient is defined as:

$$G(N) = 1 - \sum_{i=1}^{c} {p_i}^2$$

where p_i is the probability of class i in the node. In the context of binary classification, we can define gini coefficient as follows:

$$G(N) = 1 - ({p_0}^2 + {p_1}^2)$$

The same logic applies here, where the weighted gini coefficient of each possible attribute split is calculated. The only difference, however, is that there is no "gain" to be calculated. Simply, the attribute with the lowest gini coefficient is chosen. Let's start with the root node. We calculate the gini coefficient for splits at each attribute: Outlook, Temperature, Humidity, and Wind. Starting with **Outlook**:

- Sunny: 5 instances (3 "No", 2 "Yes")

$$G(N_{\text{Sunny}}) = 1 - \left((\frac{3}{5})^2 + (\frac{2}{5})^2 \right) \approx 0.48$$

- Overcast: 4 instances (all "Yes")

$$G(N_{\text{Overcast}}) = 0$$

- Rain: 5 instances (2 "No", 3 "Yes")

$$G(N_{\text{Rain}}) = 1 - \left((\frac{2}{5})^2 + (\frac{3}{5})^2 \right) \approx 0.48$$

The weighted gini coefficient for Outlook is:

$$G(\text{Outlook}) = \frac{5}{14} \times 0.48 + \frac{4}{14} \times 0 + \frac{5}{14} \times 0.48 \approx 0.343$$

Next, **Temperature**:

- Hot: 4 instances (2 ”No”, 2 ”Yes”)

$$G(N_{\text{Hot}}) = 1 - \left((\frac{2}{4})^2 + (\frac{2}{4})^2 \right) \approx 0.5$$

- Mild: 6 instances (2 ”No”, 4 ”Yes”)

$$G(N_{\text{Mild}}) = 1 - \left((\frac{2}{6})^2 + (\frac{4}{6})^2 \right) \approx 0.444$$

- Cool: 4 instances (1 ”No”, 3 ”Yes”)

$$G(N_{\text{Cool}}) = 1 - \left((\frac{1}{4})^2 + (\frac{3}{4})^2 \right) \approx 0.375$$

The weighted gini coefficient for Temperature is:

$$G(\text{Temp}) = \frac{4}{14} \times 0.5 + \frac{6}{14} \times 0.444 + \frac{4}{14} \times 0.375 \approx 0.440$$

Next, **Humidity**:

- High: 7 instances (4 ”No”, 3 ”Yes”)

$$G(N_{\text{High}}) = 1 - \left((\frac{4}{7})^2 + (\frac{3}{7})^2 \right) \approx 0.49$$

- Normal: 7 instances (1 ”No”, 6 ”Yes”)

$$G(N_{\text{Normal}}) = 1 - \left((\frac{1}{7})^2 + (\frac{6}{7})^2 \right) \approx 0.245$$

The weighted gini coefficient for Humidity is:

$$G(\text{Humidity}) = \frac{7}{14} \times 0.49 + \frac{7}{14} \times 0.245 \approx 0.429$$

Lastly, **Wind**:

- Weak: 8 instances (2 ”No”, 6 ”Yes”)

$$G(N_{\text{Weak}}) = 1 - \left((\frac{2}{8})^2 + (\frac{6}{8})^2 \right) \approx 0.375$$

- Strong: 6 instances (3 ”No”, 3 ”Yes”)

$$G(N_{\text{Strong}}) = 1 - \left((\frac{3}{6})^2 + (\frac{3}{6})^2 \right) \approx 0.5$$

The weighted gini coefficient for Wind is:

$$G(\text{Wind}) = \frac{8}{14} \times 0.375 + \frac{6}{14} \times 0.5 \approx 0.429$$

The gini coefficient for each attribute from the root node is as follows:

- $G(\text{Outlook}) = 0.343$
- $G(\text{Temp}) = 0.440$
- $G(\text{Humidity}) = 0.367$
- $G(\text{Wind}) = 0.429$

We will split the root node at **Outlook**, as it results in the lowest gini coefficient of 0.343. From earlier calculations, we know that the **Overcast** node has a gini coefficient of 0, making it our first official leaf node. Our (partial) decision tree now looks like this:

Outlook?

- Overcast: *Play Tennis*
- Sunny: ...
- Rain: ...

We now have two subtrees to continue our calculations. From this point onwards, they will be more concise to avoid clutter. Starting with the **Sunny** subtree, we calculate the gini coefficient for each attribute as follows:

- **Temperature**:
  - $G(N_{\text{Hot}}) = 0$
  - $G(N_{\text{Mild}}) = 0.5$
  - $G(N_{\text{Cool}}) = 0$
  - Weighted gini $H(\text{Temperature}) = 0.2$
- **Humidity**:
  - $G(N_{\text{High}}) = 0$
  - $G(N_{\text{Normal}}) = 0$
  - Weighted gini $H(\text{Humidity}) = 0$
- **Wind**:
  - $G(N_{\text{Weak}}) = 0.444$
  - $G(N_{\text{Strong}}) = 0.5$
  - Weighted gini $H(\text{Wind}) = 0.467$

We will split the root node at **Humidity**, as it results in the lowest gini coefficient of 0. We can also see that the gini coefficients for both Humidity conditions are 0, giving us our next set of leaf nodes. Our updated (partial) decision tree now looks like this:

Outlook?

- Overcast: *Play Tennis*
- Sunny: **Humidity?**
  - High: *Dont play Tennis*
  - Normal: *Play Tennis*
- Rain: ...

Going back to our **Rain** subtree, we calculate the gini coefficient for each attribute as follows:

- **Temperature**:
  - $G(N_{\text{Mild}}) = 0.444$
  - $G(N_{\text{Cool}}) = 0.5$
  - Weighted gini $H(\text{Temperature}) = 0.467$
- **Humidity**:
  - $G(N_{\text{High}}) = 0.5$
  - $G(N_{\text{Normal}}) = 0.444$
  - Weighted gini $H(\text{Humidity}) = 0.467$
- **Wind**:
  - $G(N_{\text{Weak}}) = 0$
  - $G(N_{\text{Strong}}) = 0$
  - Weighted gini $H(\text{Wind}) = 0$

We will split the root node at **Wind**, as it results in the lowest gini coefficient of 0. We can also see that the gini coefficients for both Wind conditions are 0, giving us our final set of leaf nodes. Our updated (final) decision tree now looks like this:

Outlook?

- Overcast: *Play Tennis*
- Sunny: **Humidity?**
  - High: *Dont play Tennis*
  - Normal: *Play Tennis*
- Rain: **Wind?**
  - Weak: *Play Tennis*
  - Strong: *Dont play Tennis*

Exactly the same as Figure 37 when entropy was used. Let's compare with sklearn's implementation:

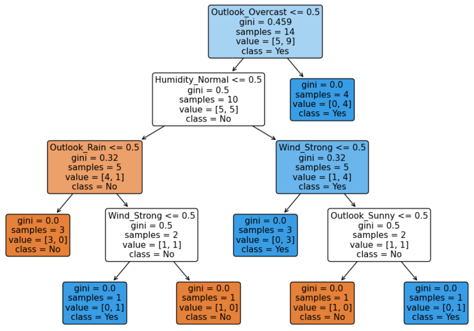

Fig. 39. Gini Coefficient implementation using sklearn

When we think about comparing entropy and gini coefficient, the most obvious difference is the range. Entropy ranges from 0 to 1, with 0 indicating total purity and 1 indicating total impurity. Gini coefficient, however, only ranges from 0 to 0.5, with 0.5 representing total impurity. When plotting the equations as a function of p_1, we get the following:

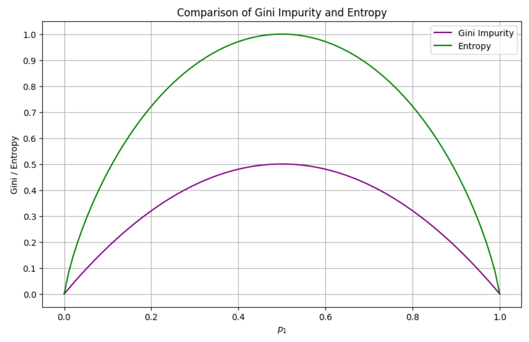

Fig. 40. Comparison of Entropy and Gini Coefficient

But on what basis would someone choose one over the other? After some more googling, we found that despite gini coefficient being less interpretable, it is often quicker to compute do to the absence of logarithmic functions, unlike entropy. For much larger datasets, the difference in computational complexity is significant enough to choose gini coefficient over entropy.

Moving on. Let's implement sklearn on the yeast dataset with the following code:

```python
# ID3 (Information Gain)
id3_tree = DecisionTreeClassifier(criterion='
    entropy', random_state=42)
id3_tree.fit(X_train, y_train)
id3_predictions = id3_tree.predict(X_test)
# CART (Gini Coefficient)
cart_tree = DecisionTreeClassifier(criterion='gini'
    , random_state=42)
cart_tree.fit(X_train, y_train)
cart_predictions = cart_tree.predict(X_test)
# Accuracies
id3_accuracy = accuracy_score(y_test,
    id3_predictions)
cart_accuracy = accuracy_score(y_test,
    cart_predictions)
```

The decision trees are too big to plot, so here's a table with the accuracies instead:

Model	Accuracy
ID3 (Information Gain)	0.4910
CART (Gini Coefficient)	0.5179

TABLE XIX
COMPARING PERFORMANCE OF ID3 AND CART ON THE YEAST DATASET

Before we start implementing Tsallis Entropy, let's define it:

$$S_q(p) = \frac{1 - \sum p_i^q}{q - 1}$$

where q is a tunable parameter and p_i in the probability of class i. As q approaches 2, it converges to the standard gini coefficient. If it approaches 1, it converges to our standard entropy defined earlier (known as Shannon entropy).

To start things off, we run 20 experiments using values of $q \in [3, 8]$ and evaluate their accuracies:

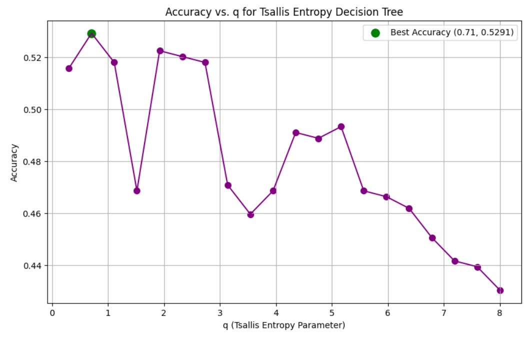

Fig. 41. Accuracy vs q for Tsallis Entropy

For this set of data, the best accuracy achieved was 52.91% at a q-value of 0.71. We can observe a general trend where accuracy decreases as q increases. There is, however, some randomness to the pattern. To account for this in the hope that a smoother curve will be achieved, we will average the accuracies over 10 runs of the same experiment, but with random data splits:

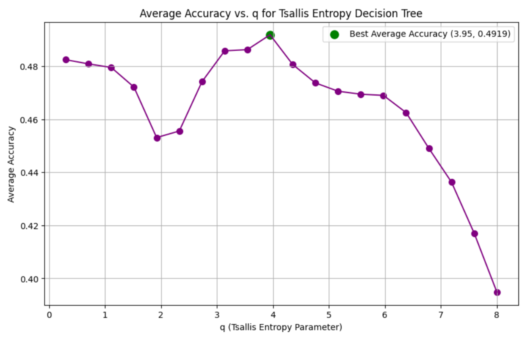

Fig. 42. Average Accuracy vs q for Tsallis Entropy across 10 runs

Interesting. The curve is now much smoother, with a peak accuracy of 49.19% for a q value of 3.95. Accuracy dips around $q = 2$, which is when Tsallis Entropy becomes equivalent to Gini Coefficient. Exploring this on multiple datasets would require better computational resources (like an A100 that is functional), so this is the best we could do to explore q on our local computers