Q2:

In this scenario where the target function is a line that separates points into two classes (positive and negative) on the plane, both decision tree and nearest-neighbor learning have their strengths and weaknesses.

For this specific problem, I'd lean towards using decision tree learning. Decision trees are effective for problems where there's a clear decision boundary, especially when it aligns with axes (like a line in the plane). They partition the feature space into regions and can create splits perpendicular to the axes, which makes them well-suited for problems with linear decision boundaries.

Nearest-neighbor learning, on the other hand, might not perform as well in this scenario because its decision-making relies heavily on the entire dataset. While it can capture intricate patterns in the data, it might struggle to generalize well for this linearly separable problem. Nearest neighbors could potentially create a more convoluted decision boundary that doesn't align with the simple linear separation of the classes.

So, given the simplicity of the target function—a straight line—and the need for a clear decision boundary, a decision tree seems like the more suitable choice between the two algorithms.

Q3:

L1 and L2 regularization are techniques used to prevent overfitting in machine learning models by adding a penalty term to the cost function.

L1 regularization (Lasso) adds the absolute values of the coefficients of the features to the cost function. It tends to force some coefficients to become exactly zero, effectively performing feature selection and resulting in sparse models. This happens because L1 regularization encourages sparsity by eliminating certain less important features, leaving only the most relevant ones in the model.

L2 regularization (Ridge) adds the squared magnitudes of the coefficients of the features to the cost function. It doesn’t typically force coefficients to be exactly zero, instead, it shrinks the coefficients towards zero, reducing their impact but not eliminating them entirely. This tends to produce models with all features contributing to some extent to the predictions.

The bias-variance trade-off refers to the balance between a model's ability to represent the underlying patterns in the data (low bias) and its sensitivity to noise or fluctuations in the training data (variance).

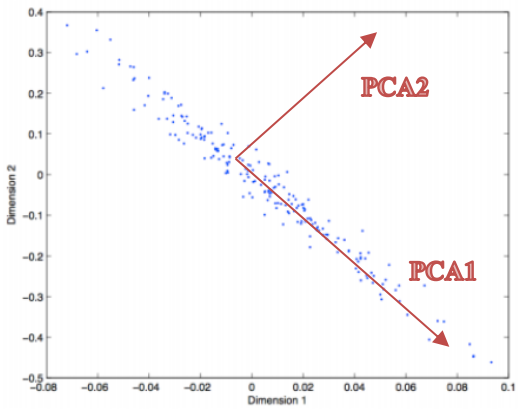
Example 1: Consider a linear regression model that fits data points with a straight line. It has low variance (less sensitive to changes in the training set) but high bias (may oversimplify complex patterns present in the data).

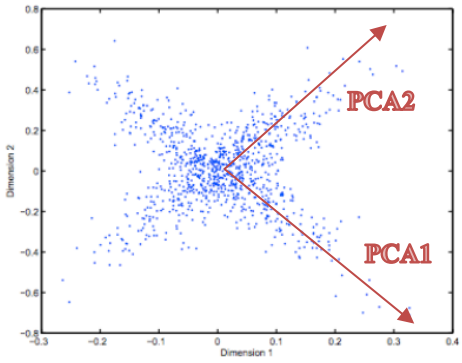
Example 2: In contrast, a high-degree polynomial regression model fits the training data extremely well, capturing intricate patterns. This model has low bias (can represent complex relationships) but high variance (tends to overfit and generalize poorly to new data).

The curse of dimensionality refers to the problems and limitations that arise when dealing with high-dimensional data spaces. As the number of dimensions (features) increases, the amount of data needed to effectively cover that space grows exponentially.

For example, in nearest-neighbor-based algorithms, as the number of dimensions increases, the distance between points becomes less meaningful. In high-dimensional spaces, most points are far away from each other, leading to increased computational complexity and difficulties in finding nearest neighbors accurately. This can result in decreased performance or increased errors in algorithms relying on distance metrics, impacting their effectiveness in high-dimensional spaces.

Q4:





Q5:

K-means aims to partition data into k clusters by minimizing the within-cluster variance. It tends to work well when clusters are relatively spherical, non-overlapping, and equally sized due to its optimization for minimizing variance.

Hierarchical clustering methods, especially with single, complete, or average linkages, might not perform as effectively at k=2 because they tend to produce clusters with varying shapes and sizes. They are sensitive to the linkage criterion and can create clusters that might not be distinct or equally separated.

The EM algorithm, while effective for probabilistic clustering with Gaussian Mixture Models, might not inherently create distinct clusters at k=2 as its convergence depends on initializations and the underlying distribution of the data.

Q6:

Decision Trees can be adapted for regression tasks by modifying the splitting criterion and the way predictions are made at the leaf nodes. Instead of using entropy or Gini impurity for classification, regression trees use metrics like mean squared error (MSE) or mean absolute error (MAE) to determine the best split at each node.

Regarding the expected value at any leaf in a Decision Tree with squared error as the error function:

The objective is to minimize the squared error within each leaf node.

At each node, the decision tree algorithm tries to partition the data to minimize the squared error of the target variable within each partition.

When the tree is constructed, at each leaf node, the predicted value is taken as the average (mean) of the target values of the data points in that leaf node.

This means that the expected value at any leaf in a regression tree built with squared error as the error function will indeed be the mean of the target values of the data points within that leaf node.

The MSE is 10.416078431372549

Q7:

Let's design the perceptron:

A will have a weight of 1.

B will have a weight of -1 (to implement the negation).

The bias term will be -0.5.

The output of the perceptron will be 1 if A=1 and B=0, and 0 otherwise, which matches the truth table for A∧¬B.

For the XOR function (A⊕B), which cannot be implemented by a single-layer perceptron, a two-layer network of perceptrons can be used.

Here's a two-layer network to implement A⊕B:

First layer: A∧¬B and B∧¬A

Second layer: OR operation on outputs of the first layer

To create the first layer, we need two perceptrons:

For A∧¬B, use the weights and bias as described earlier for the A∧¬B function.

For B∧¬A, the weights and bias will be:

A with a weight of -1, B with a weight of 1, and a bias of -0.5.

The output of these two perceptrons will represent A∧¬B and B∧¬A respectively.

The second layer will combine these outputs using an OR operation. We can implement this using a single perceptron with inputs from the outputs of the first layer:

Both outputs from the first layer are inputs to this second-layer perceptron.

Assign a weight of 1 to both inputs, with a bias of -0.5.

This two-layer network will correctly implement the XOR (A⊕B) function using perceptrons.