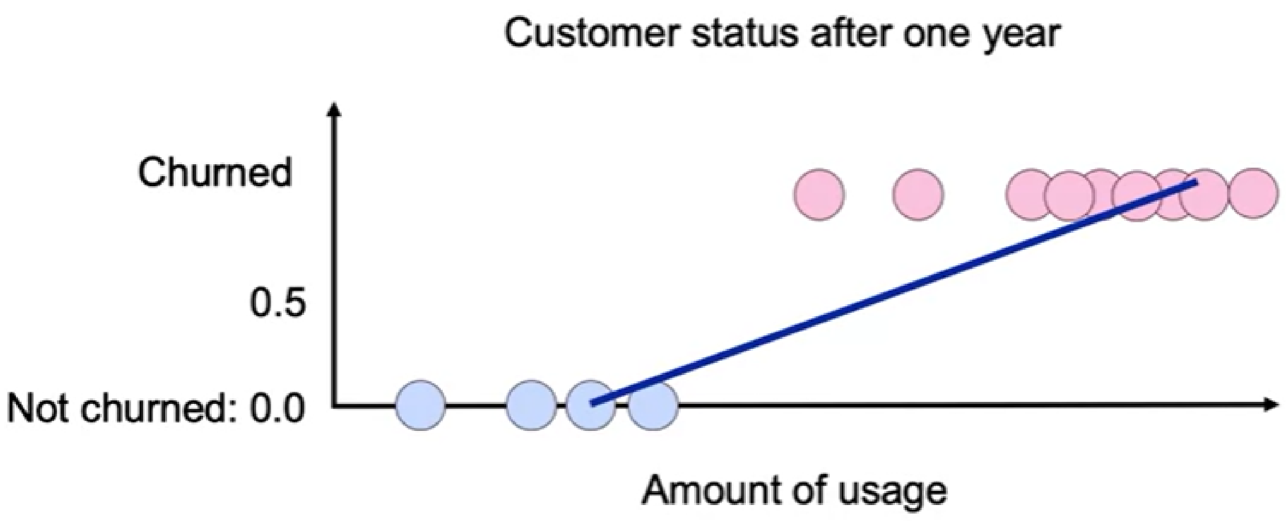
Course III - Supervised ML (Classification)

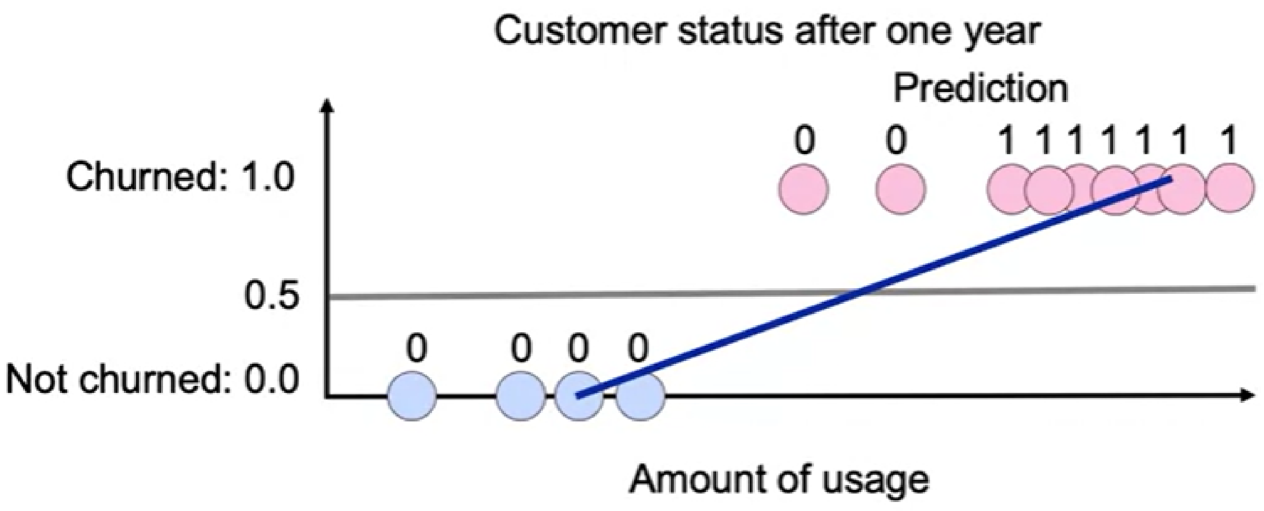
3.1 Introduction to Classification and Logistic Regression

- Classification vs. Regression:**
  - Supervised learning is divided into regression and classification.
  - In regression, the outcome is a continuous number, while in classification, the outcome is categorical (i.e., classes or labels).
  - Examples of regression problems include house prices and box office revenues, while examples of classification problems include fraud detection and customer churn prediction.
- **Understanding Classification:**
  - Classification involves predicting the class or category of a new example based on past examples.
  - Features of past examples are quantified and represented in a feature space.
  - Known labels belonging to each example are used for training the model.
  - Similarity between past and new examples is measured using machine learning algorithms to determine the most similar class.
- **Commonly Used Classification Models:**
  - Logistic Regression: Extends linear regression for classification problems.
  - K-Nearest Neighbors (KNN): Categorizes based on the similarity of nearest neighbors in the feature space.
  - Support Vector Machines (SVM): Leverages the kernel trick to create complex decision boundaries.
  - Neural Networks: Combines linear and nonlinear steps to learn complex decision boundaries.
  - Decision Trees: Use intermediate decision boundaries to create complex final decision boundaries.
  - Random Forests, Boosting, and Ensemble Methods: Utilize multiple classifiers to reduce variance and bias.
- **Introduction to Logistic Regression:**
  - Logistic regression is introduced as a classification algorithm used to predict binary outcomes.
  - It addresses the limitations of linear regression when applied to classification tasks, especially when dealing with skewed data distributions.
- **Example of Customer Churn Prediction:**
  - The example of customer churn prediction in a telecom company is used to illustrate the binary classification problem.
  - Usage in minutes is considered as the feature, and the outcome is whether the customer churned (1) or not (0).
- **Treating Classification as Regression:**
  - Binary classification can be treated as a regression problem by encoding the classes as 1 and 0.
  - A line of best fit (regression line) is used to separate the two classes based on the feature values.



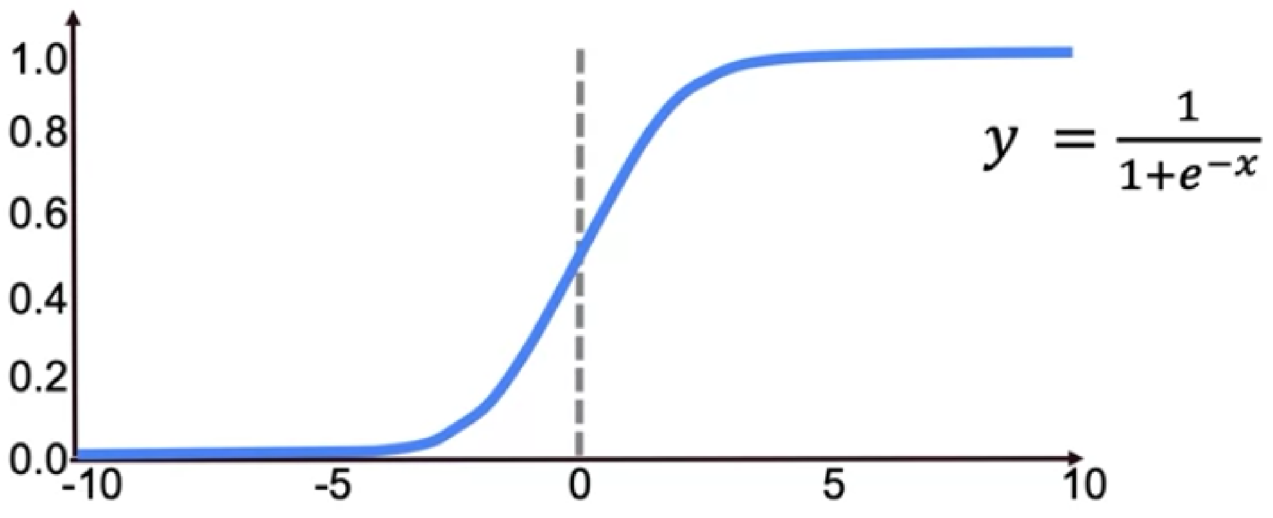
$$y_\beta(x) = \beta_0 + \beta_1 x + \varepsilon$$

- **Limitations of Linear Regression for Classification:**
  - Linear regression may lead to skewed predictions, especially when the decision boundary between classes is not linear.
  - In cases where the regression line is heavily slanted, the 0.5 cutoff for classification may result in incorrect predictions.

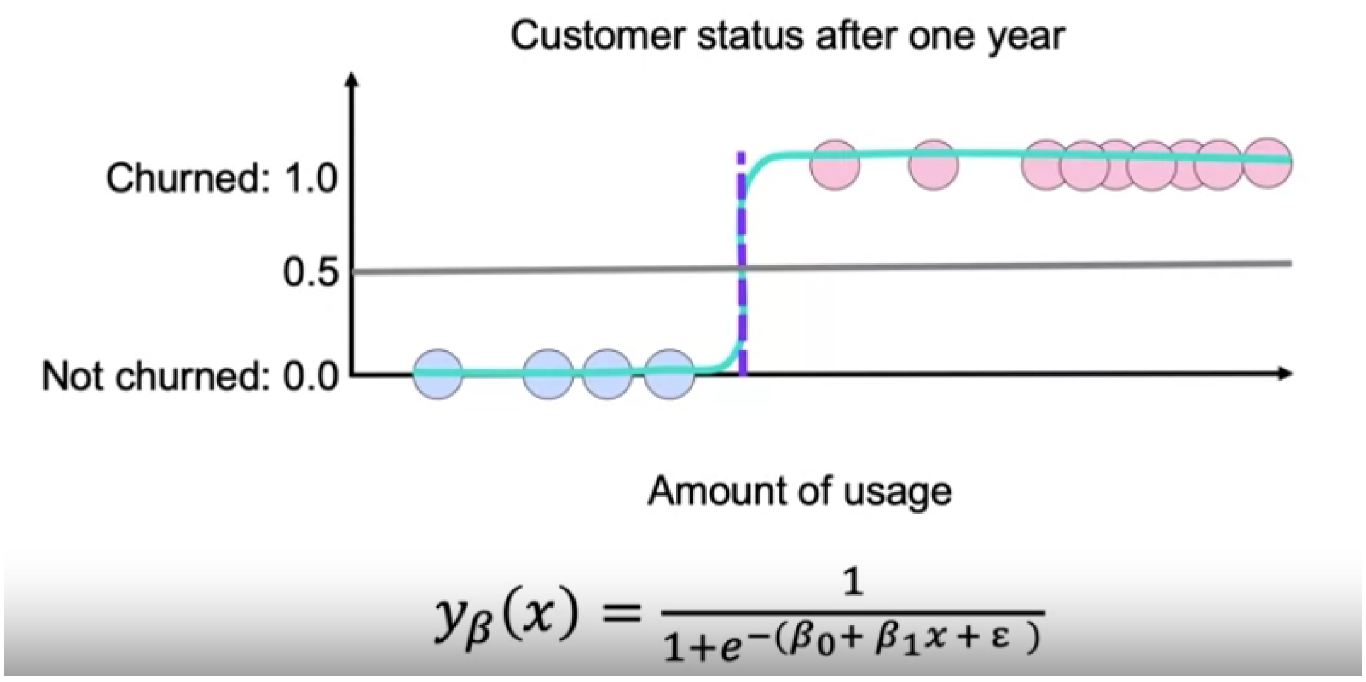


If model result > 0.5: predict churned
If model result < 0.5: predict not churned

- **Introduction to the Sigmoid Function:**
  - The Sigmoid function, represented as $\frac{1}{1+e^{-x}}$, is introduced.
  - It transforms the output of the linear regression function into values between 0 and 1.
  - This function ensures that the output of logistic regression is bounded and interpretable as probabilities.



- **Logistic Regression as a Classification Algorithm:**
    - Despite the name, logistic regression is a classification algorithm, not a regression algorithm.
    - The logistic function ensures that the output of logistic regression represents the probability of a sample belonging to a certain class.
- **Decision Boundary and Classification:**
    - The decision boundary is the threshold at which the predicted probability switches from one class to another (e.g., from churned to not churned).
    - Logistic regression correctly classifies samples on both sides of the decision boundary based on their predicted probabilities.



- **Logistic Function (Sigmoid Function):**
  The logistic function, also known as the sigmoid function, transforms the linear equation into a function of probabilities bounded between 0 and 1.

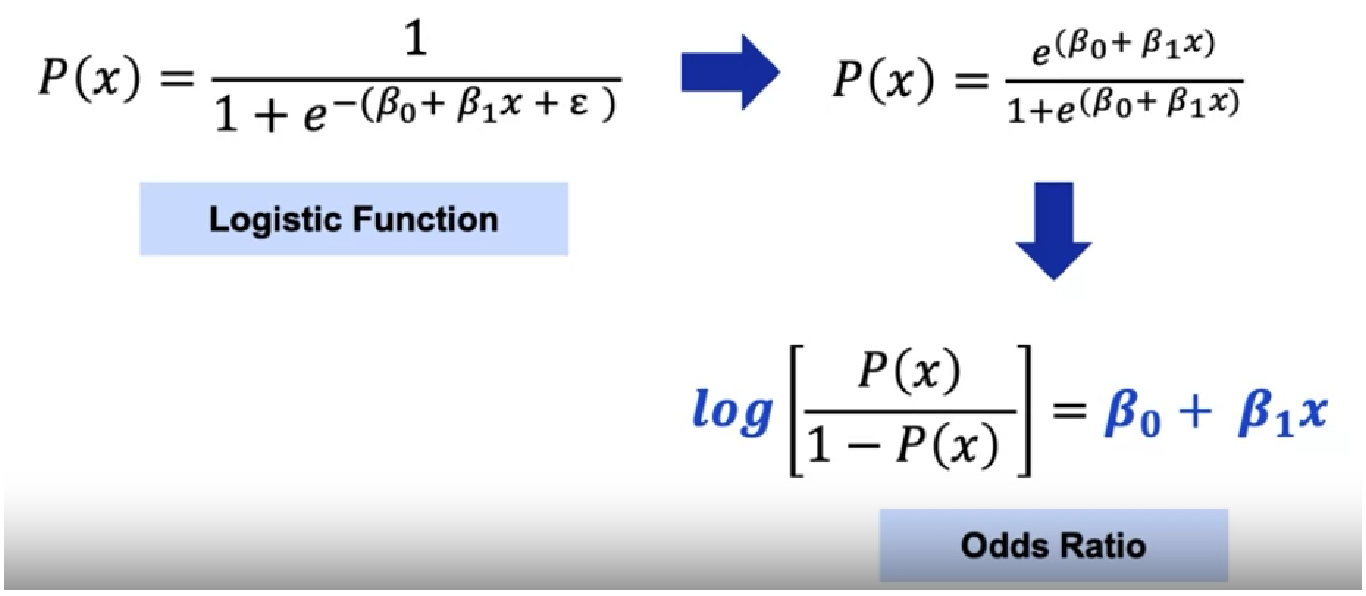
  $p(x) = \frac{1}{1+e^{-z}}$

  where $p(x)$ represents the probability of the sample belonging to a certain class, and z is the linear function of input features.
- **Odds Ratio:**
  The odds ratio is introduced to interpret the output of logistic regression. It represents the ratio of the probability of an event occurring to the probability of it not occurring.

  $\text{Odds Ratio} = \frac{p(x)}{1-p(x)}$

- **Log-Odds (Logit):**
  The log-odds, also known as the logit function, is a linear function of the input features, providing interpretability in terms of the log-odds of belonging to a certain class.

  $$\text{logit}(p(x)) = \log\left(\frac{p(x)}{1-p(x)}\right)$$



$$P(x) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x + \varepsilon)}} \Rightarrow P(x) = \frac{e^{(\beta_0 + \beta_1 x)}}{1+e^{(\beta_0 + \beta_1 x)}}$$

Logistic Function

$$\log\left[\frac{P(x)}{1 - P(x)}\right] = \beta_0 + \beta_1 x$$

Odds Ratio

These equations demonstrate how logistic regression transforms the linear relationship between input features and output into probabilities, allowing for meaningful interpretation of the model coefficients in terms of odds and log-odds.

- **Extension to Multiple Features:**
  - In cases with multiple features, the decision boundary becomes a hyperplane separating the classes in the feature space.
  - Logistic regression can handle higher-dimensional data by extending the linear decision boundary to hyperplanes.

Overall, the Sigmoid function plays a crucial role in logistic regression by transforming the output of linear regression into probabilities, enabling the algorithm to make binary classifications based on these probabilities. The decision boundary separates the classes, and logistic regression can be extended to handle more complex classification tasks with multiple features.

In a multi-class classification scenario, binary classifiers like logistic regression can be used with a technique called one-versus-all (OvA) or one-versus-rest (OvR) to handle multiple classes. Here's how it works:

1. **One-Versus-All Technique:**
   - With the one-versus-all approach, each class is treated as the positive class, while all other classes are combined into the negative class.
   - For example, if there are three classes (not churned, canceled, and left for a competitor), one logistic regression model is trained to distinguish not churned from all other classes.
2. **Training Multiple Binary Classifiers:**
   - Separate logistic regression models are trained for each class against the rest of the classes.
   - For instance, one model is trained to distinguish not churned from churned and canceled, another to distinguish canceled from not churned and left for a competitor, and so on.
3. **Decision Boundaries for Each Class:**

    - Each binary logistic regression model defines a decision boundary between the specific class and the rest.
    - These decision boundaries separate the instances most likely to belong to the class from those belonging to other classes.
4. **Predicting Class Labels:**
    - When a new instance needs to be classified, each binary classifier produces a probability score for its corresponding class.
    - The class with the highest estimated probability is chosen as the predicted class label for the instance.
    - In this way, the instance is assigned to the class with the highest probability among all classes.
5. **Visualizing Decision Boundaries:**
    - The decision boundaries generated by the binary logistic regression models determine the regions of the feature space associated with each class.
    - Each class is represented by a distinct region separated by the decision boundaries.

Overall, the one-versus-all approach allows binary classifiers like logistic regression to be extended to multi-class classification problems by training multiple classifiers, each focusing on distinguishing one class from the rest. The final predicted class label is determined by selecting the class with the highest probability score among all classes.

To fit a logistic regression model using scikit-learn, we follow these steps:

1. **Import Logistic Regression Model:**

    ```
    from sklearn.linear_model import LogisticRegression
    ```

2. **Instantiate the Model:**

    ```
    LR = LogisticRegression(penalty='l2', C=1.0)
    ```

    - Here, we instantiate the logistic regression model, specifying the regularization type ('l2' for L2 regularization) and the regularization strength parameter `C`, where higher values of `C` indicate less penalty.
3. **Fit the Model:**

    ```
    LR.fit(X_train, y_train)
    ```

    - We fit the model using the training data (`X_train` for features and `y_train` for labels).
4. **Make Predictions:**

    ```
    predictions = LR.predict(X_test)
    ```

    - After fitting the model, we can use it to make predictions on new data (`X_test`), obtaining the predicted class labels.
5. **Access Coefficients:**

    ```
    coefficients = LR.coef_
    ```

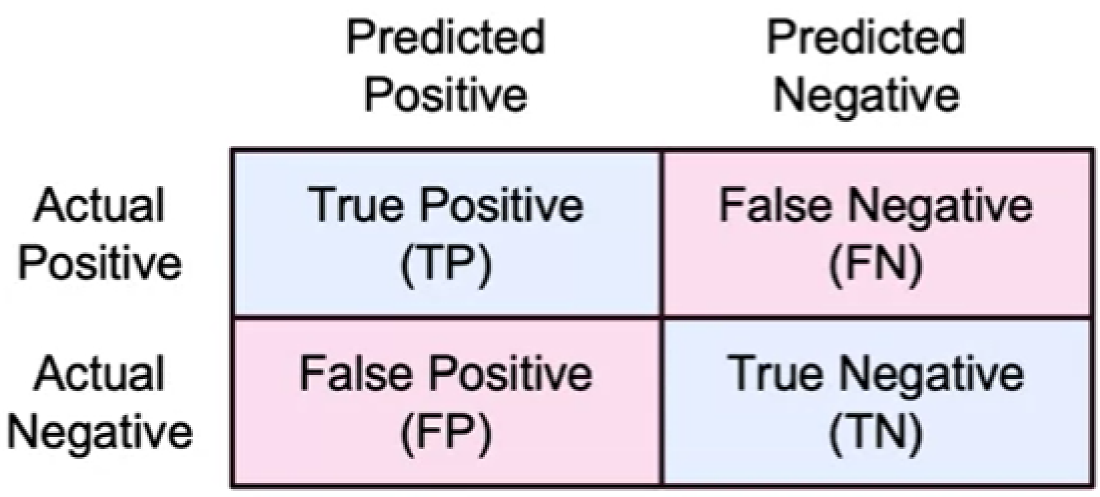
- We can access the coefficients of the logistic regression model using the `coef_` attribute. These coefficients represent the weights assigned to each feature in the prediction.

6. **Additional Considerations:**
    - For statistical inference and obtaining p-values for coefficients, the statsmodels package may be more suitable.
    - Scikit-learn offers convenient cross-validation methods for parameter tuning, such as `GridSearchCV`, which helps find the best hyperparameters for the model.

Applications of logistic regression extend beyond customer churn prediction and include:

- Predicting customer spending likelihood
- Forecasting customer engagement
- Detecting fraudulent transactions in e-commerce
- Assessing loan default risk in finance

3.2 Classification Error Metrics



	Predicted Positive	Predicted Negative
Actual Positive	True Positive (TP)	False Negative (FN)
Actual Negative	False Positive (FP)	True Negative (TN)

1. **Accuracy:**
    - Accuracy measures the overall correctness of the classification by calculating the ratio of correctly predicted samples to the total number of samples.

$$\text{Accuracy} = \frac{\text{True Positives} + \text{True Negatives}}{\text{Total Samples}}$$

2. **Recall (Sensitivity):**
    - Recall measures the ability of the model to correctly identify all actual positive instances. It calculates the ratio of true positives to the total actual positives.

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

3. **Precision:**
    - Precision measures the accuracy of positive predictions made by the model. It calculates the ratio of true positives to the total predicted positives.

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

4. **Specificity:**
    - Specificity measures the ability of the model to correctly identify all actual negative instances. It calculates the ratio of true negatives to the total actual negatives.

$$\text{Specificity} = \frac{\text{True Negatives}}{\text{True Negatives} + \text{False Positives}}$$

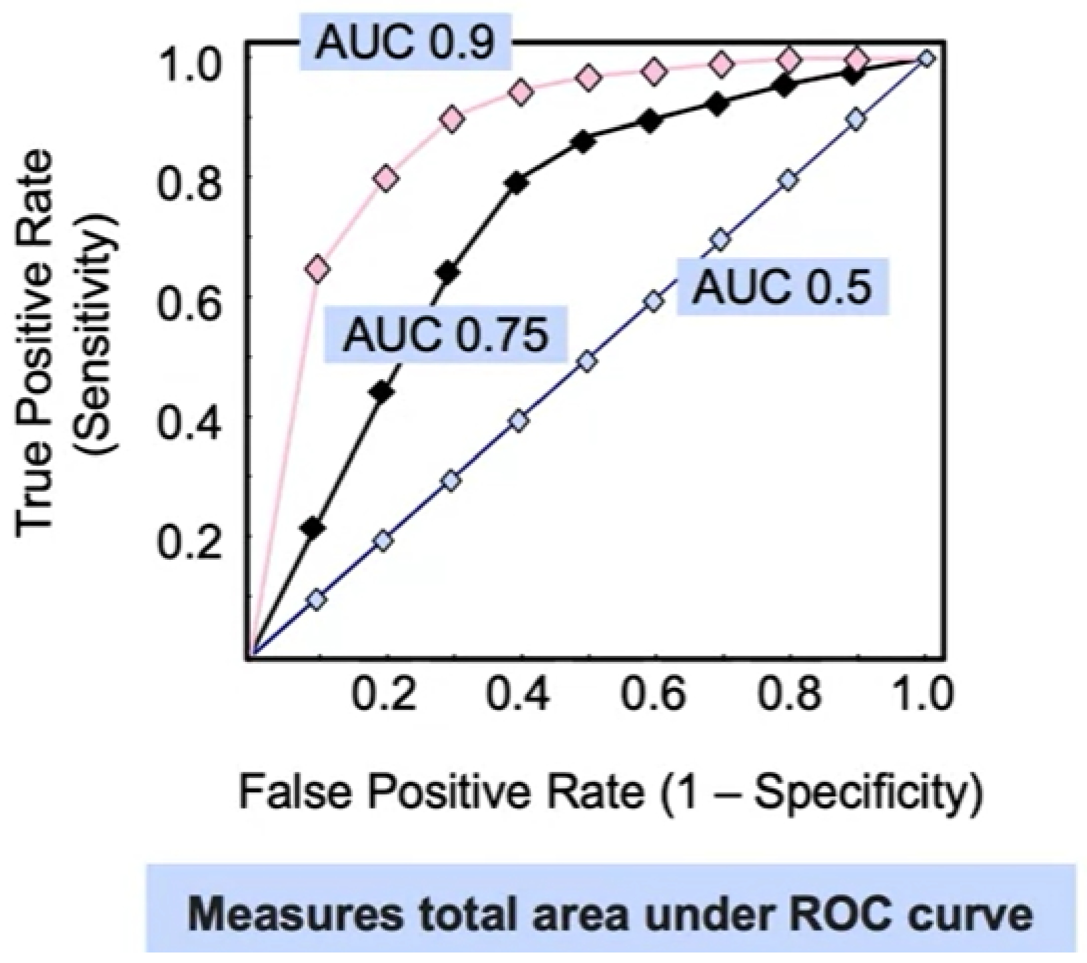
5. **F1 Score:**
    - The F1 score is the harmonic mean of precision and recall. It provides a balance between precision and recall and is useful when there is an uneven class distribution.

    $$\text{F1 Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

    These metrics are essential for evaluating classification models, especially in scenarios where the data is imbalanced. By understanding these metrics, we can assess the performance of our models and make informed decisions about model selection and optimization.

The Receiver Operating Characteristic (ROC) curve is a graphical representation of the performance of a classification model across various thresholds.

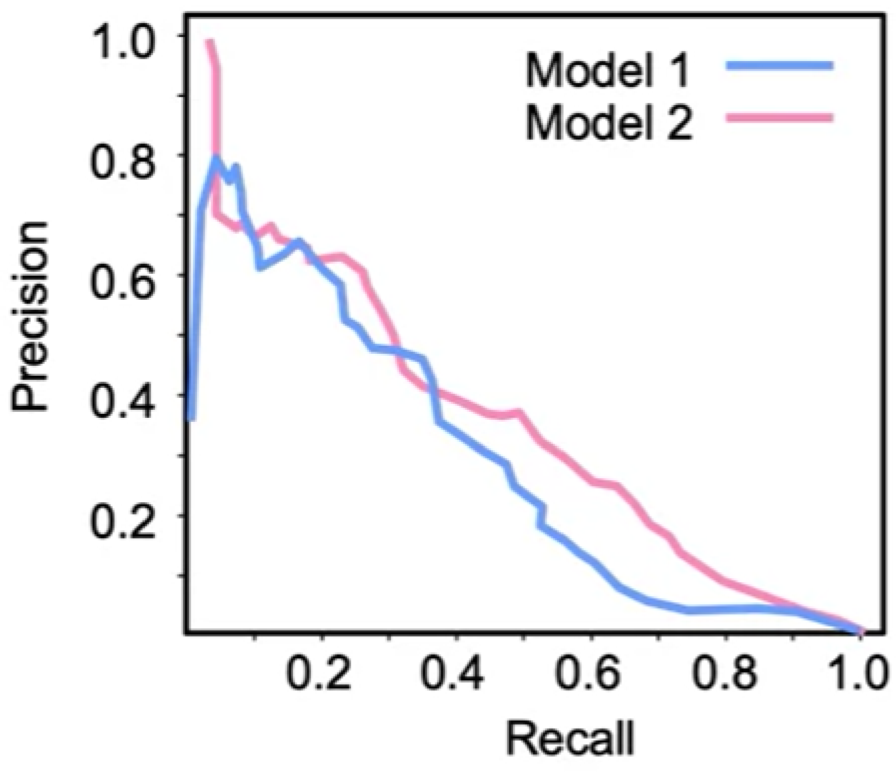
1. **ROC Curve:**
    - The ROC curve plots the sensitivity (true positive rate) against the false positive rate (1 - specificity) for different threshold values.
    - Sensitivity indicates the model's ability to correctly identify all actual positive instances.
    - False positive rate represents the proportion of actual negative instances incorrectly predicted as positive.



2. **Threshold Consideration:**
    - The curve considers different probability thresholds for classification, rather than the traditional 0.5 threshold used in binary classification.
    - By adjusting the threshold, the trade-off between true positive rate and false positive rate can be controlled.
3. **Interpretation:**

- The diagonal of the ROC matrix represents random guessing, with an area under the curve (AUC) of 0.5.
- A perfect classifier would have an ROC curve that reaches the top-left corner, with an AUC close to 1.
- The closer the AUC is to 1, the better the model's ability to distinguish between positive and negative classes.

4. **Balanced Metric:**
    - ROC AUC is a balanced metric that considers both true positive rate and false positive rate, unlike accuracy which can be misleading with imbalanced datasets.
5. **Application:**
    - ROC curves are useful for evaluating classifiers with balanced class distributions.
6. **Precision-Recall Curve:**
    - Precision-recall curves provide another perspective on classifier performance, especially in situations with imbalanced classes.
    - They plot precision (positive predictive value) against recall (sensitivity) for different threshold values.



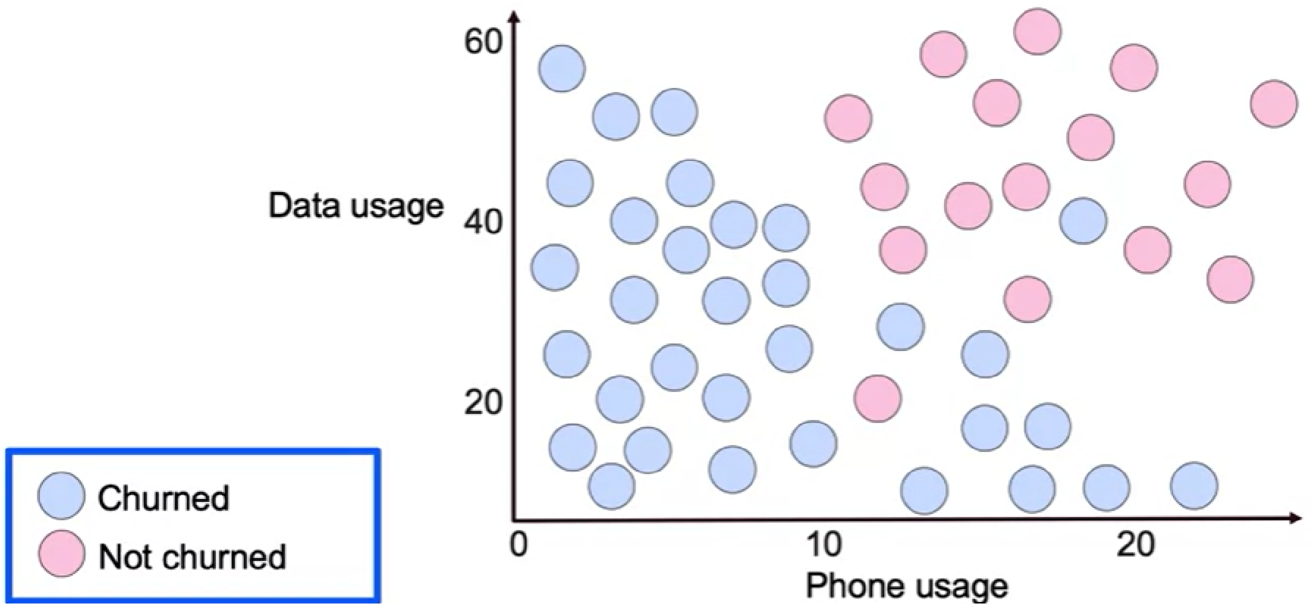
Measures trade-off between precision and recall

7. **Choosing the Right Approach:**
    - The choice between ROC curve and precision-recall curve depends on the class distribution and the relative importance of false positives and false negatives in the specific application.
    - Business outcomes and the costs associated with different types of errors should be considered when selecting the appropriate evaluation metric and threshold.

Understanding and analyzing these curves help in making informed decisions about classifier performance and selecting the optimal threshold for classification tasks, considering the specific context and requirements of the problem at hand.

3.3 K Nearest Neighbors

Example Scenario: We discuss a scenario where we aim to predict telecom customer churn based on historical data containing information about customer usage in minutes and data usage in gigabytes. By plotting this data on a scatter plot with churn outcomes indicated by color (blue for churned and red for not churned), we visualize how KNN can be used to predict the churn likelihood for a new customer.



K Nearest Neighbors Algorithm: The KNN algorithm predicts the label of a new data point by considering the labels of its K nearest neighbors in the feature space. In the case of binary classification, the label is determined by a majority vote among the K neighbors.

Choosing K: The choice of K, the number of nearest neighbors to consider, is a critical hyperparameter in KNN. We explored how different values of K can influence the prediction and how using an odd value for K helps avoid tie situations.

Implementation Considerations: It's important to scale the features before applying KNN to ensure that all features contribute equally to the distance calculation. Additionally, tie-breaking strategies can be employed when determining the predicted label in case of conflicts among the nearest neighbors.

Hyperparameter Tuning: The selection of the optimal value for K is crucial and often involves tuning through techniques like cross-validation to find the K that yields the best performance on unseen data.

To get started with our KNN model, there are several key considerations:

1. **Determination of K**: The choice of the number of nearest neighbors, K, is crucial and needs to be carefully selected. This is a hyperparameter that must be tuned based on the problem at hand and the performance metrics chosen for evaluation.
2. **Distance Measurement**: It's essential to decide on the distance metric used to calculate the similarity or distance between data points. Common distance metrics include Euclidean distance, Manhattan distance, Minkowski distance, and more. The choice of distance metric can significantly impact the performance of the KNN algorithm.
3. **Weighting of Neighbors**: In some cases, it may be beneficial to assign different weights to neighbors based on their distance from the data point being predicted. Closer neighbors may be given higher weights, indicating that they have more influence on the prediction. This can help improve the accuracy of the model, especially when dealing with unevenly distributed data.
4. **Decision Boundary**: The decision boundary of the KNN model depends on the chosen value of K. Smaller values of K result in more complex decision boundaries that can lead

to overfitting, while larger values of K lead to smoother decision boundaries that may suffer from underfitting. Finding the right balance between bias and variance is crucial.

5. **Choosing the Optimal K**: Since K is a hyperparameter, it needs to be tuned using techniques like cross-validation or the elbow method. The optimal value of K is typically selected based on the error rate or performance metrics such as accuracy, precision, recall, F1 score, etc., on a hold-out validation set.
6. **Business Objectives and Error Metrics**: It's important to align the choice of K and other parameters with the business objectives and the desired error metric. Whether the focus is on maximizing recall, precision, F1 score, or another metric depends on the specific requirements and constraints of the problem.

Overall, understanding these considerations and making informed decisions about K, distance measurement, and weighting is essential for building an effective KNN model tailored to the specific classification task at hand.

In K Nearest Neighbors (KNN) algorithm, computing the distance between two given points is crucial as it serves as the similarity measure, where closer points are assumed to be more similar. There are several distance measures commonly used:

1. **Euclidean Distance (L2 distance)** : This is the most popular distance measure, where the distance between two points in Euclidean space is calculated using the square root of the sum of squared differences of each feature dimension. It's visually intuitive and represents the straight-line distance between two points.
2. **Manhattan Distance (L1 distance)**: Also known as the Manhattan distance, it measures the sum of absolute differences between corresponding coordinates of points. It's called Manhattan distance because it's akin to measuring the distance a person would walk along city blocks in Manhattan.

After selecting a distance measure, it's essential to consider the scale of variables because KNN heavily relies on distance. If variables are not scaled appropriately, features with larger magnitudes might dominate the distance calculation, leading to biased predictions.

Feature Scaling: Feature scaling is the process of normalizing the range of independent variables or features of the data. It ensures that each feature contributes equally to the distance calculation, preventing bias towards features with larger scales. Common scaling methods include:

- **Min-Max Scaling**: Rescales features to a range between 0 and 1 by subtracting the minimum value and dividing by the range (maximum - minimum).
- **Standardization (Z-score normalization)**: Transforms features to have a mean of 0 and a standard deviation of 1 by subtracting the mean and dividing by the standard deviation.

Properly scaled features ensure that KNN considers each feature equally when determining the nearest neighbors and making predictions.

KNN can be extended to handle multiple classes by choosing the class with the majority vote among its nearest neighbors. The choice of K for multiple classes is crucial, and it's often recommended to choose a K value that ensures one class always has a majority vote, such as a multiple of the number of classes plus one.

Additionally, KNN can be adapted for regression tasks, where it predicts continuous values. In regression, the predicted value is typically the mean value of the target variable among its nearest neighbors. The choice of K in regression determines the level of smoothing applied to the predictions.

Pros:

1. **Simple to Implement:** KNN is straightforward to understand and implement. It doesn't involve complex mathematical computations or the estimation of parameters.
2. **Adapts Well to New Data:** As new data is introduced, the model adapts by recalculating the nearest neighbors. This makes it suitable for scenarios where the data distribution changes over time.
3. **Easy to Interpret:** KNN provides intuitive results that are easy to interpret, making it valuable for decision-makers in business applications. It's transparent and allows stakeholders to understand the reasoning behind predictions easily.
4. **Useful for Customer Profiling:** KNN can be powerful in identifying customer profiles similar to the most lucrative ones. This can help in predicting the likelihood of new customers being lucrative based on their similarities to existing profitable customers.

Cons:

1. **Slow Prediction:** Predicting with KNN can be slow, especially for large datasets, as it requires calculating distances to numerous data points.
2. **Lack of Model Representation:** KNN does not provide an explicit model representation like logistic regression. It doesn't offer insights into the relationship between features and the target variable.
3. **High Memory Requirement:** KNN requires storing the entire training dataset in memory, which can be memory-intensive for large datasets.
4. **Sensitive to High-Dimensional Data:** KNN may struggle with high-dimensional data, as the concept of distance becomes less meaningful in high-dimensional spaces. As the number of features increases, the distance between points tends to become more uniform, leading to less discriminative power.

Comparison between KNN and Linear Regression:

1. **Model Fitting Speed:**
   - Linear Regression: Relatively slow as it involves a search for the best parameters.
   - K Nearest Neighbors: Fast, as it only requires storing the data without parameter estimation.
2. **Memory Efficiency:**
   - Linear Regression: Very memory efficient as it only needs to remember the coefficients.
   - K Nearest Neighbors: Memory intensive as it needs to remember all data points in the training set.
3. **Prediction Speed:**
   - Linear Regression: Fast, as it involves a simple linear combination of vectors.
   - K Nearest Neighbors: Slower, as it requires computing distances to multiple data points.

Syntax for creating a KNN classification model:

1. Import the K Nearest Neighbors classifier from sklearn:

   ```
   from sklearn.neighbors import KNeighborsClassifier
   ```

2. Create an instance of the KNeighborsClassifier class and set hyperparameters:

```
KNN = KNeighborsClassifier(n_neighbors=3)
```

3. Fit the model to the training data:

```
KNN.fit(X_train, y_train)
```

4. Predict on the test set:

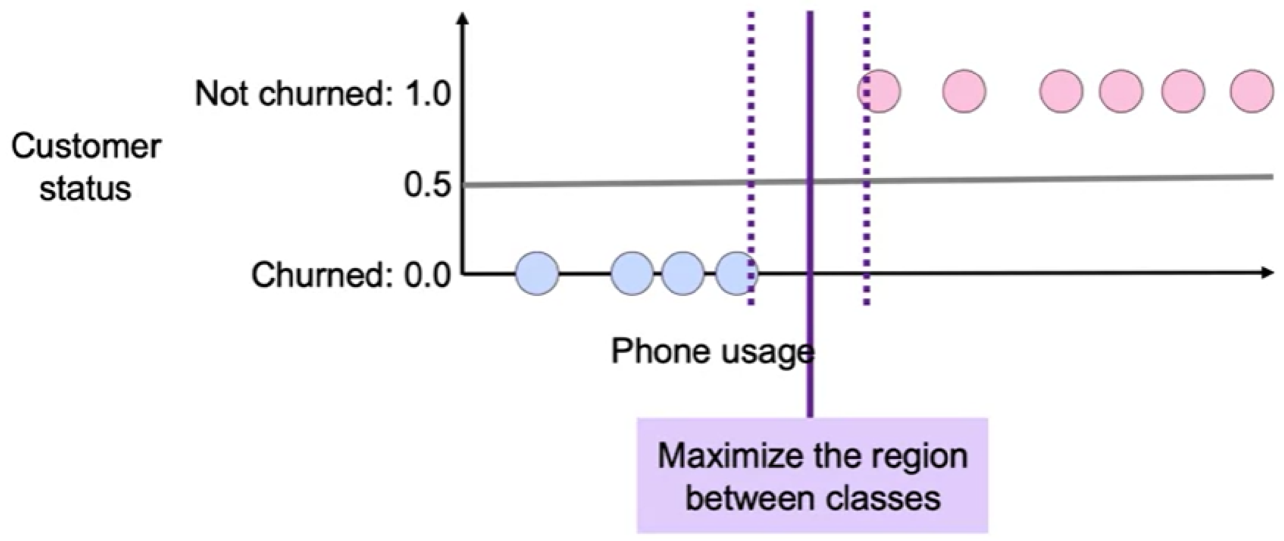
```
y_predict = KNN.predict(X_test)
```

- Ensure to choose appropriate hyperparameters like `n_neighbors` (k) based on the problem and data.
- Other parameters like weights (uniform or distance-based) and distance metric (Euclidean or Manhattan) can be customized as needed.
- For regression tasks, replace `KNeighborsClassifier` with `KNeighborsRegressor`, and ensure the target variable is continuous.

3.4 Support Vector Machines

Let's build intuition behind SVM starting from what we learned about logistic regression. Logistic regression aimed to transform the loss function to weigh far-away points less, thus avoiding being skewed by outliers. Now, let's extend this to understand SVM:

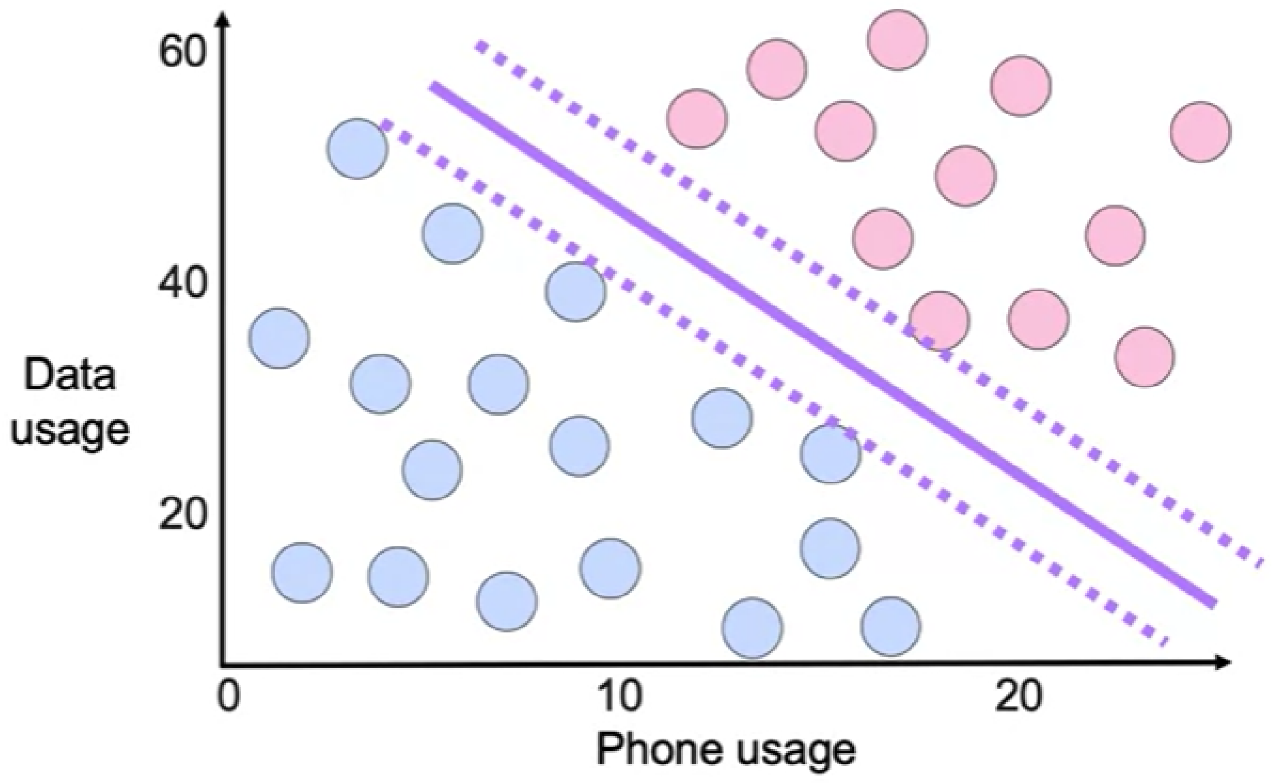
1. **Visualizing Decision Boundary:**
   - Consider a one-dimensional example with phone usage.
   - Attempt to find the optimal point to split the data and create a decision boundary.
2. **Selecting Decision Boundary:**
   - Initially, arbitrary decision boundaries may result in misclassifications.
   - By exploring different positions for the boundary, aiming for no misclassifications, we seek to optimize its placement.
3. **Introduction of Margin:**
   - Introduce a dotted line representing the distance from the nearest point to the decision boundary.
   - Aim to maximize the region between the two classes, ensuring a margin of separation.
4. **Maximizing Margin:**
   - Set the boundary equidistant from the closest points of each class to maximize the margin between them.



The ultimate goal of the SVM algorithm is to draw the decision boundary in a way that maximizes the margin between classes. This ensures robust classification performance.

To deepen our understanding of why selecting a decision boundary with equidistant measures between the blue and red classes is preferable, let's consider the implications for new customers.

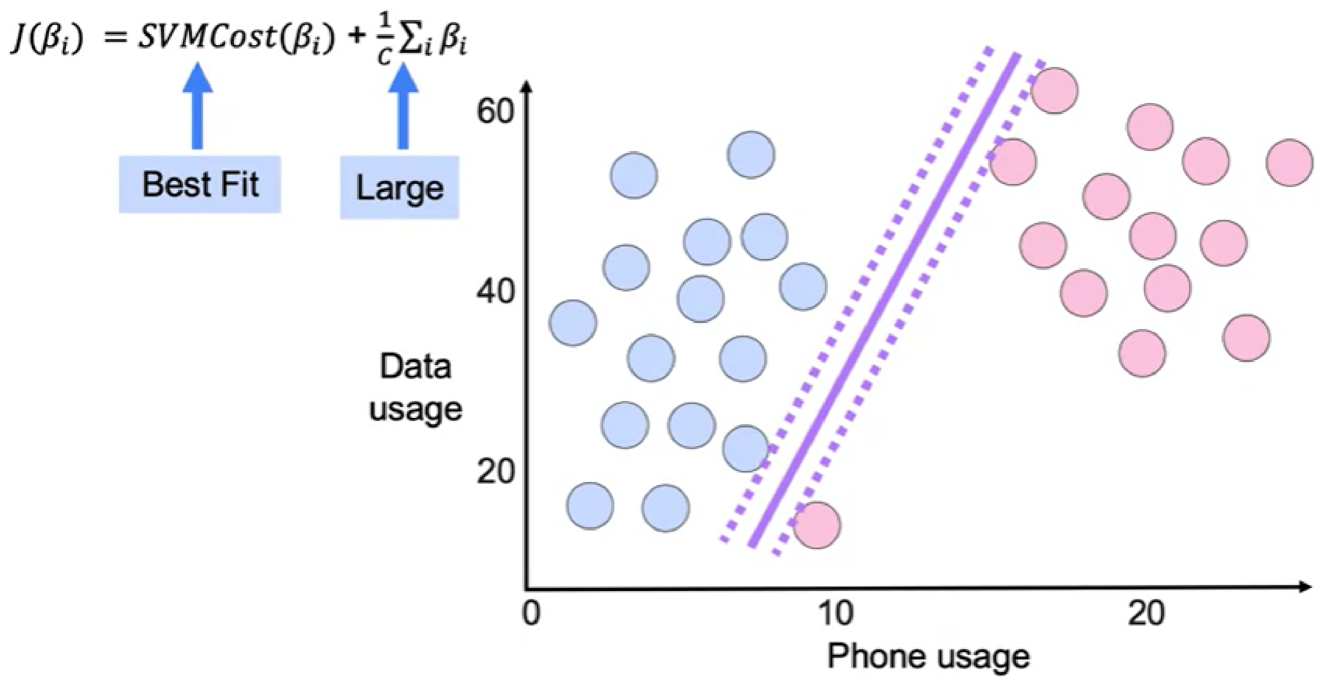
1. **Avoiding Sensitivity to Boundary Position:**
   - If the boundary is too close to the blue dots, a new customer slightly to the right of the boundary might be misclassified as red, even if it's very close to blue.
   - Similarly, a customer very close to blue but on the other side of the boundary might be misclassified as blue.
2. **Support Vectors:**
   - The blue and red samples that define the margin (the dotted lines) are called support vectors.
   - These support vectors play a crucial role in determining the optimal decision boundary for the SVM.



Extending this concept to higher dimensions, such as two-dimensional feature space with data usage and phone usage, we aim to find the best linear decision boundary again. Let's examine different boundary placements:

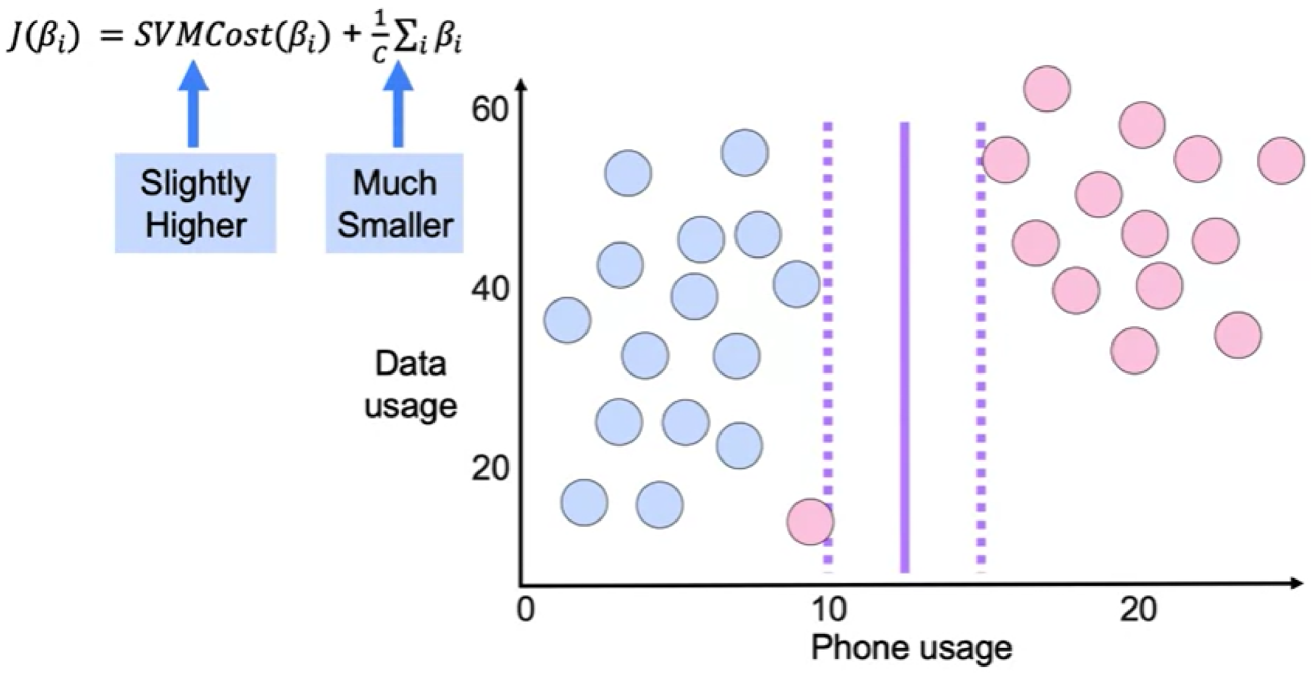
1. **Ineffective Boundaries:**
    - Boundaries resulting in many misclassifications are ineffective.
    - Tight boundaries close to one class may lead to sensitivity issues similar to those observed in the one-dimensional example.
2. **Optimizing Boundary Placement:**
    - The goal is to maximize the distance between the two classes by finding the hyperplane that best separates them.
    - Using support vectors, we aim to find the distance that's furthest from each of the two groups.

In the context of Support Vector Machines (SVMs), the cost function used to optimize the solution is based on hinge loss. Unlike logistic regression, where the cost function is smoothed out and rarely reaches zero due to the probabilistic nature of predictions, SVMs utilize hinge loss to penalize misclassifications more heavily.



Let's break down how hinge loss works:

1. **Penalty for Correct Classification within Margin:**
    - If a point lies within the margin and is correctly classified, it incurs no error term in the cost function.
    - The loss value for such points is zero.
2. **Penalty for Points Outside Margin:**
    - Points to the left of the margin (on the wrong side of the decision boundary) start incurring penalties.
    - As we move away from the decision boundary towards the margin, the loss increases linearly.
    - The loss is set to one at the margin and further increases linearly for points outside the margin.
3. **Regularization in SVMs:**
    - To prevent overfitting and find a balance between minimizing misclassifications and keeping the decision boundary simple, SVMs employ regularization.
    - Regularization is achieved by adding a regularization term to the cost function, similar to techniques like ridge, elastic net, or lasso in linear regression.
    - This additional term penalizes complex decision boundaries, encouraging simpler models.

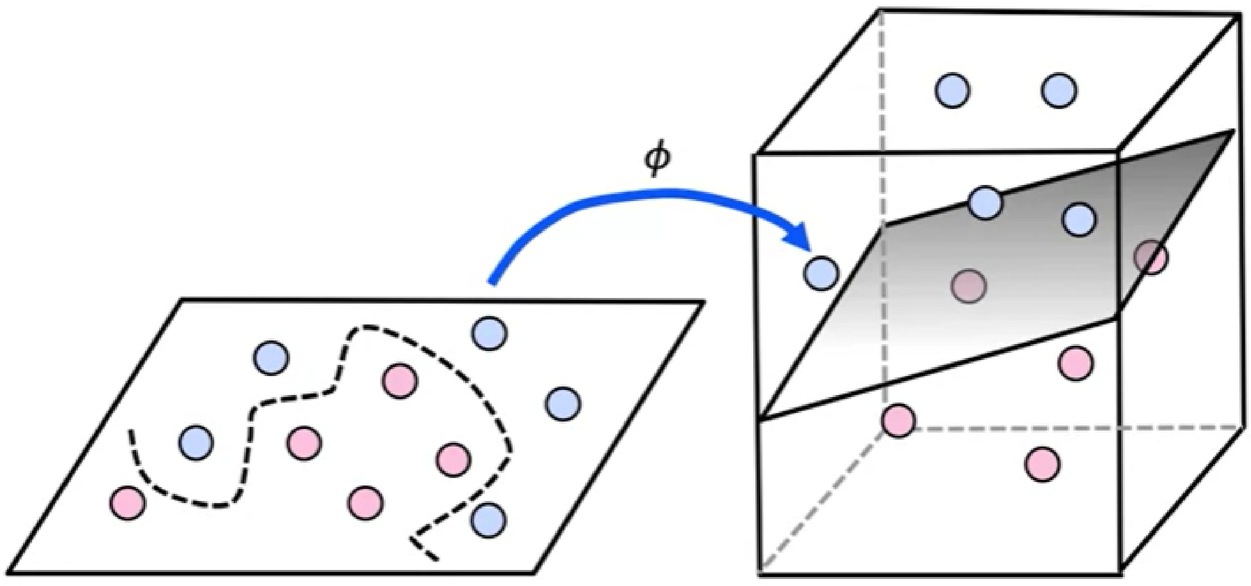


4. **Impact of Noisy Data on Decision Boundary:**
    - In cases where the dataset is noisy or contains outliers, optimizing the boundary to correctly classify every single point may lead to overfitting.
    - The goal of SVMs is not to perfectly classify every point but to find a boundary that separates the classes while allowing for some misclassifications.
    - By introducing regularization, SVMs strike a balance between fitting the data and generalizing to unseen examples.

3.5 SVM Kernels

The key concept behind using kernels in SVMs is to transform the original feature space into a higher-dimensional space where the data may become separable by a linear boundary. This transformation allows for more complex decision boundaries in the original feature space, which may appear non-linear.

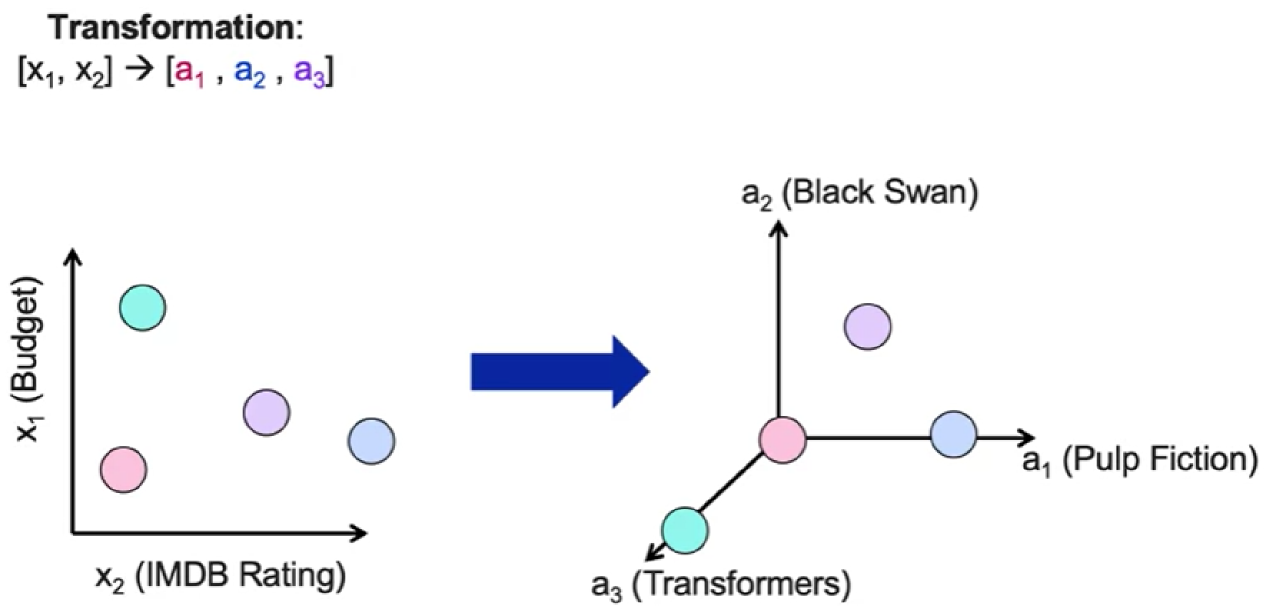
While the idea may seem abstract, the essence is that complex curves or decision boundaries in lower-dimensional space may correspond to simpler linear boundaries in higher-dimensional space. By leveraging kernels, SVMs can effectively learn these non-linear decision boundaries by finding the optimal hyperplane in the transformed feature space.



1. **Creating Non-linear Boundaries with Feature Engineering:**
    - We discussed how to create non-linear decision boundaries by generating additional features from existing ones, similar to polynomial feature expansion. These additional

features represent higher dimensions in the feature space.

2. **Utilizing Similarity Functions and Radial Basis Functions:**
   - Another approach involves defining similarity functions, such as Gaussian functions, to measure the similarity between data points and reference points (e.g., movies in a dataset). Radial basis functions (RBFs) are used to calculate these similarities.
3. **Mapping to Higher-dimensional Space:**
   - By applying these similarity functions, we can map the original data points from lower-dimensional space to higher-dimensional space, where they may become linearly separable. Each data point gets mapped to a new set of coordinates based on its similarity to reference points.
4. **Implementation with Sklearn:**
   - We demonstrated how to implement SVMs with kernel functions using Scikit-learn (`SVC` class). Different kernel functions, such as radial basis function (RBF), can be specified along with hyperparameters like `gamma` and `C` to control model complexity and regularization.
5. **Tuning Hyperparameters:**
   - Hyperparameters like `gamma` and `C` can significantly affect model performance and generalization. Techniques like grid search cross-validation (`GridSearchCV`) can be used to find the optimal values for these hyperparameters.
6. **Computational Efficiency with Kernel Trick:**
   - While the kernel trick allows SVMs to operate in higher-dimensional spaces without explicitly computing the transformations, it can still lead to computational bottlenecks, especially with large datasets.



By leveraging the kernel trick, SVMs can effectively handle non-linear classification tasks by transforming the feature space to higher dimensions, where data points may become separable by a linear boundary. This technique offers a powerful tool for building complex yet interpretable models for various machine learning tasks.

In this section, we discussed the challenges of using support vector machines (SVMs) with radial basis function (RBF) kernels, particularly in scenarios with large datasets where training time can be a significant bottleneck. To address this issue, we explored the concept of kernel approximation, which allows us to achieve good enough results with computational ease and reduced training time.

1. **Approximating Kernels for Efficiency:**
   - Kernel approximation methods like Nystroem and RBF sampler are used to create a dataset in higher-dimensional space without performing the classification step. These

    methods aim to reduce the computational complexity associated with directly applying the kernel function to every data point.

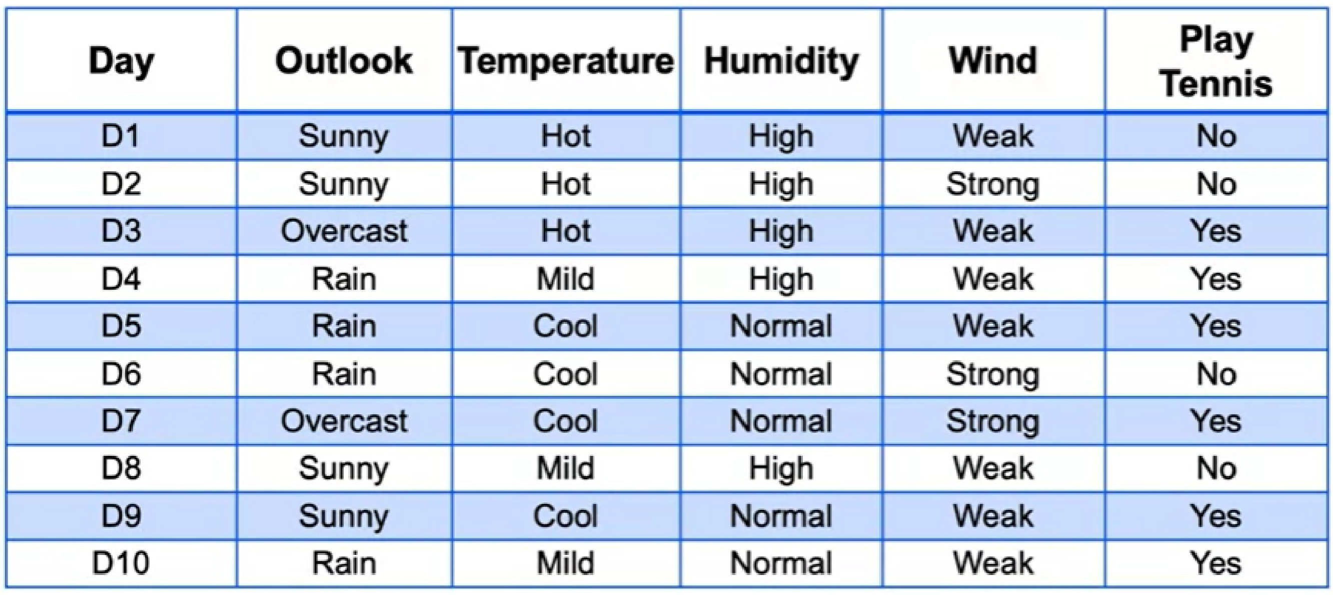
2. **Choosing between Kernel Approximation and Regular SVC:**
    - The decision to use kernel approximation or regular SVC with kernels depends on the size of the dataset and the number of features:
        - **Many Features, Small Dataset:** For datasets with a large number of features (>10,000) and a small dataset, simple linear classifiers like logistic regression or LinearSVC are preferred due to the complexity already present in the feature space.
        - **Few Features, Medium Dataset:** When the dataset has fewer features (<100) but a medium-sized dataset (~10,000 rows), using SVC with sophisticated kernels like RBF is suitable. Mapping to higher dimensions can enhance model performance without significant computational overhead.
        - **Very Few Features, Large Dataset:** In scenarios with very few features (<100) but a large dataset, additional feature engineering or kernel approximation methods like Nystroem or RBF sampler are recommended to handle the large volume of data efficiently.

By understanding the trade-offs between computational efficiency and model performance, practitioners can make informed decisions about the choice of SVMs with kernels or kernel approximation methods based on the characteristics of their dataset and computational resources available.

3.6 Decision Trees

Introduction to Decision Trees:

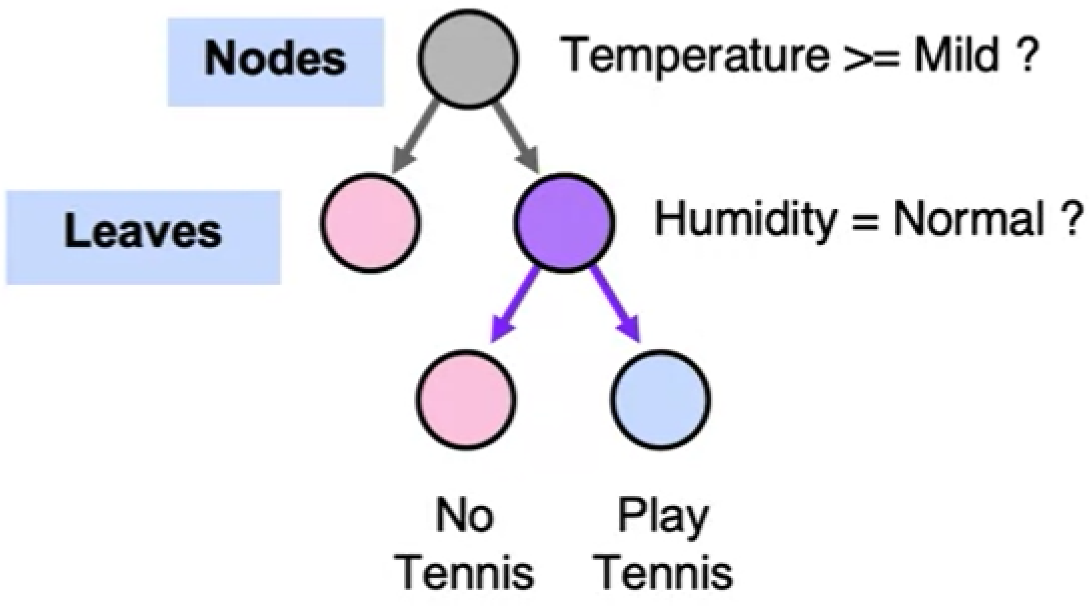
- Decision trees are a predictive modeling tool used for both classification and regression tasks.
- They partition the dataset based on features to make predictions about the target variable.
- Example application: Predicting customer turnout at a tennis facility.



Day	Outlook	Temperature	Humidity	Wind	Play Tennis
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Strong	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
D10	Rain	Mild	Normal	Weak	Yes

Splitting the Dataset:

- Decision trees iteratively split the dataset into subsets based on specific features.
- Splits are represented by nodes in the tree, with questions asked at each node to determine data division.

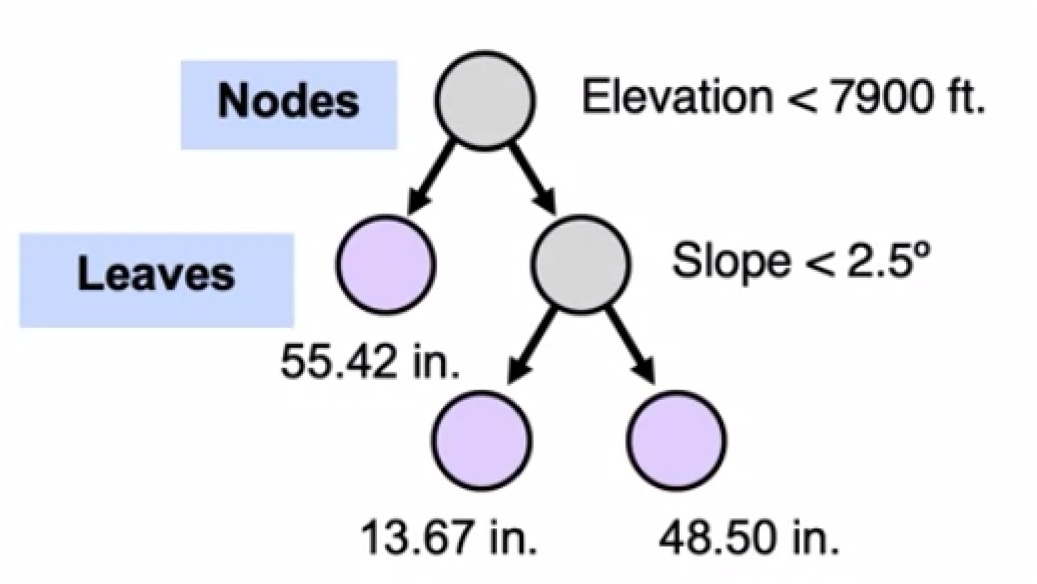


Classification vs. Regression Trees:

- Decision trees can be used for:
  - Classification: Predicting class labels.
  - Regression: Predicting continuous values.

Building Regression Trees:

- Nodes ask yes or no questions based on features.
- Leaves contain the average value of the target variable within each subset.



Tree Depth and Overfitting:

- Depth of a decision tree determines the number of splits and model complexity.
- Increasing depth can lead to overfitting, capturing noise instead of underlying patterns.

Balancing Tree Depth:

- Finding the right depth balance is crucial to prevent overfitting and ensure optimal performance on unseen data.
- Strategies for determining appropriate depth will be discussed.

Decision Tree Classification:

1. **Selecting Features and Splitting:**
   - Decision trees start by selecting a feature and asking a true/false question to split the dataset into subsets.

- Continuously split using available features to create further subsets.

2. **Stopping Criteria:**
   - One approach is to stop splitting when the leaves are pure, meaning each leaf contains only one class.
   - Another approach is to set a maximum depth and prune the tree.
   - Alternatively, stopping can be based on predefined performance metrics, such as classification accuracy.
3. **Avoiding Overfitting:**
   - Allowing the tree to split until leaves are pure may lead to overfitting, where the model performs well on the training set but poorly on unseen data.
   - Pruning at a maximum depth or stopping based on performance metrics helps avoid overfitting.
4. **Finding the Right Splits:**
   - Evaluate all possible splits and use greedy search to find the best split.
   - Information gain is used to measure the quality of a split, determined by how much uncertainty is reduced after the split.
5. **Measuring Information Gain:**
   - **Classification Error:**
     - Calculate the classification error for parent and child nodes:
       - $\text{Error}(t) = 1 - \text{Accuracy}(t)$
       - Where $\text{Accuracy}(t)$ is the proportion of correct classifications at node t.
     - Compute the weighted average of the child nodes' errors to determine information gain:
       - $\text{Information Gain} = \text{Error}_{\text{parent}} - \sum_{i=1}^{n} \frac{N_i}{N} \times \text{Error}(i)$
       - Where $\text{Error}_{\text{parent}}$ is the error at the parent node, N_i is the number of instances in the i^{th} child node, N is the total number of instances, and n is the number of child nodes.

This process helps in constructing decision trees for classification tasks, ensuring effective feature selection, splitting, and stopping criteria to achieve accurate predictions while avoiding overfitting.

Entropy as an Error Metric in Decision Trees:

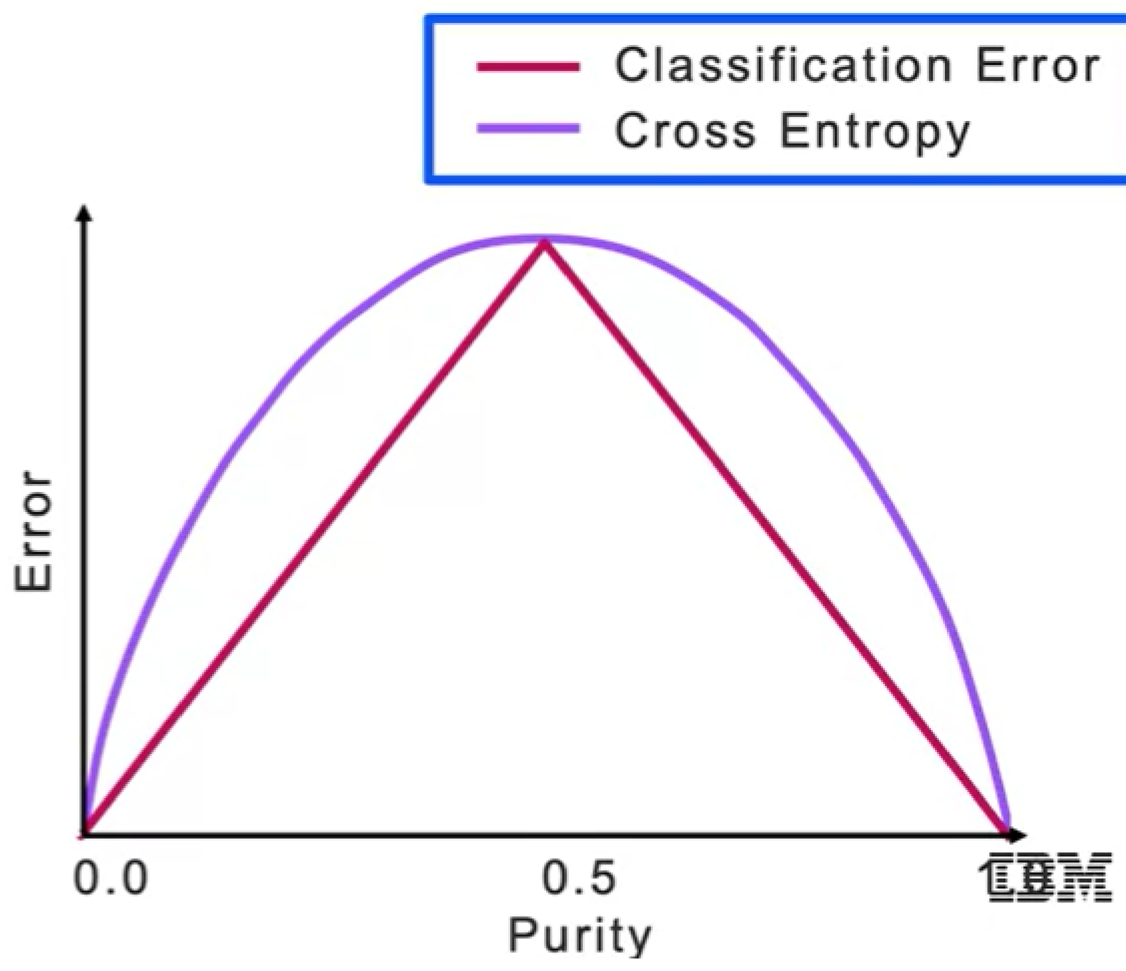
1. **Introduction to Entropy:**
   - Entropy is a measure of impurity or disorder in a dataset.
   - In decision trees, entropy is used as an error metric to find the best split.
2. **Entropy Calculation:**
   - The entropy of node t is calculated using the formula: $H(t) = -\sum_i P(i|t) \cdot \log_2(P(i|t))$
   - Where $P(i|t)$ is the probability of class i given node t.
   - The entropy represents the sum of the negative probabilities of each class, weighted by their likelihood.
3. **Example Calculation:**
   - Initial entropy is calculated for the top node using the probabilities of each class within that node.
   - Entropy is then calculated for each child node after splitting.
   - Weighted average of child node entropies is computed to determine information gain.
4. **Information Gain:**
   - Information gain is the reduction in entropy after a split.

- It allows for further splitting down the tree, leading to the possibility of achieving homogeneous nodes.

5. **Comparison with Classification Error:**
    - Unlike classification error, entropy allows for information gain, enabling further splitting and potentially reaching homogeneous nodes.
    - This flexibility improves the decision tree's ability to capture underlying patterns in the data.

Using entropy as an error metric in decision trees enables more effective splitting, ultimately leading to better classification performance and the potential to achieve fully homogeneous nodes in the tree structure.

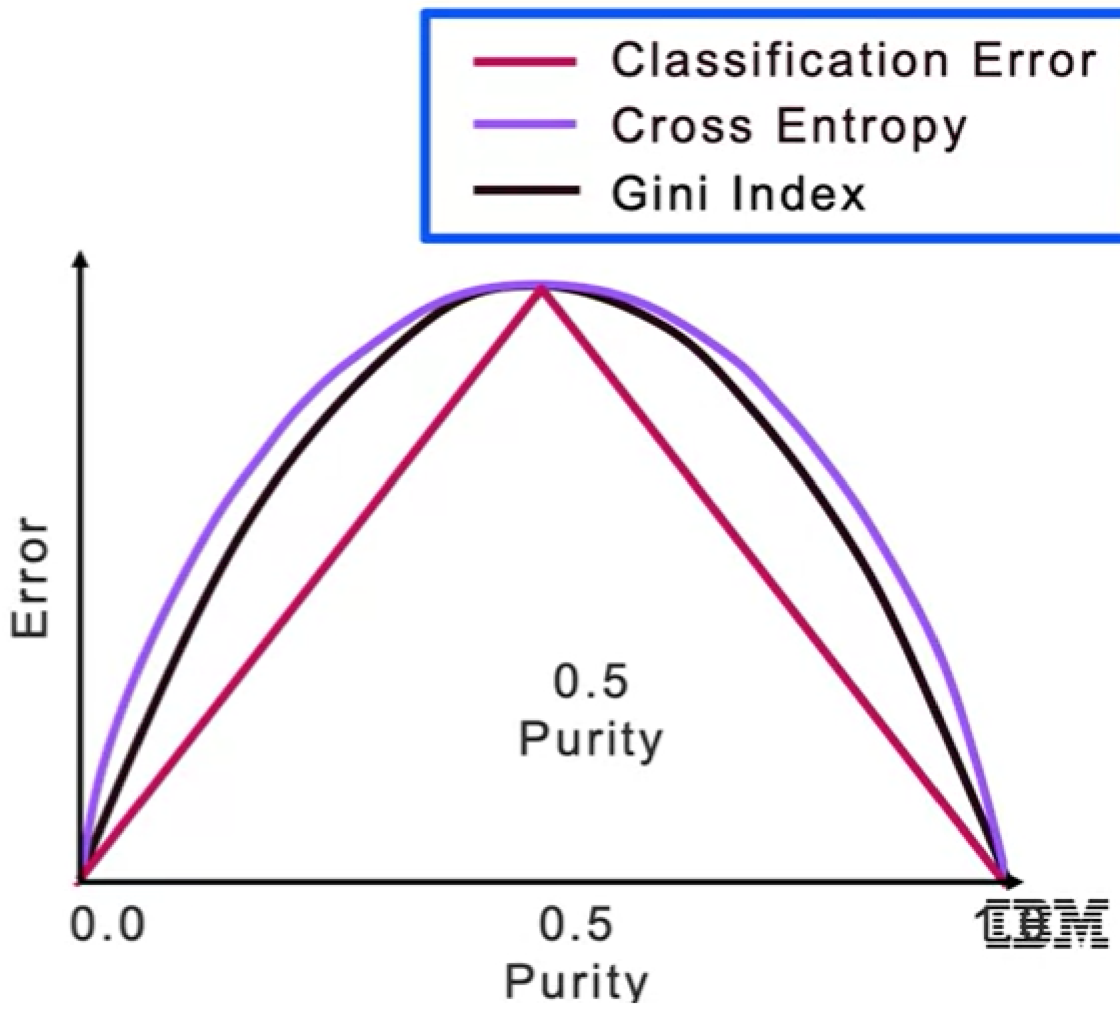
Comparison of Classification Error and Entropy:



1. **Classification Error Function:**
    - **X-axis:** Purity of the node (proportion of instances belonging to the majority class).
    - **Y-axis:** Classification error (error rate if all instances are classified as the majority class).
    - The relationship forms a straight line.
    - Maximum error (50%) occurs at the 50/50 split.
    - Error decreases linearly as purity increases.
2. **Entropy Function:**
    - **X-axis:** Purity of the node.
    - **Y-axis:** Entropy (measure of impurity or disorder).
    - The relationship forms a curved function.
    - Maximum entropy occurs at the 50/50 split.
    - Curvature allows for further splitting beyond the 50/50 split.
    - Weighted average ensures smaller entropy values after splitting, indicating information gain.
3. **Explanation:**

- With classification error, the weighted average of child nodes always lies on a straight line between the two child nodes.
- Curvature in entropy function allows the weighted average to fall below the parent node, ensuring information gain and the possibility of further splitting.
- The curvature above the straight line enables the tree to continue splitting, reaching homogeneous nodes.

4. **Comparison with Gini Index:**
    - Gini Index also exhibits a bulge similar to entropy, ensuring the possibility of perfect splits.
    - Gini Index is simpler to compute than entropy as it does not involve logarithms.
    - Both entropy and Gini Index provide a balance between splitting for information gain and avoiding overfitting.



5. **Avoiding Overfitting:**
    - While perfect splits are possible, allowing the tree to reach homogeneous nodes at every leaf leads to overfitting.
    - Balancing bias and variance is essential to avoid overfitting and achieve optimal model performance.

Understanding the differences between classification error, entropy, and other metrics like the Gini Index helps in selecting the appropriate error metric for decision tree construction, ensuring effective splitting and avoiding overfitting.

Pruning Decision Trees to Address Overfitting:

1. **High Variance and Overfitting:**
    - Decision trees tend to exhibit high variance, leading to overfitting.
    - They capture noise and intricacies of the training data, failing to generalize well to unseen data.
2. **Pruning Techniques:**

- **Max Depth:** Limit the maximum depth of the tree, allowing only a certain number of splits.
    - Pruned trees have a predetermined depth, preventing excessive branching and overfitting.
- **Classification Error Threshold:** Stop splitting nodes if the classification error falls below a certain threshold.
    - If the leaf node is correctly classified for a high percentage of samples, further splitting may not be necessary.
- **Information Gain Threshold:** Require a minimum level of information gain to continue splitting.
    - If the information gain from a split is below a specified threshold, the split is not performed.
- **Minimum Number of Samples:** Set a minimum number of samples required in each leaf node.
    - If a leaf node contains fewer samples than the threshold, further splitting is halted.

3. **Strengths of Decision Trees:**
    - **Interpretability:** Decision trees are easy to interpret and visualize, making them suitable for explaining predictions.
        - Even large trees can be interpreted by focusing on major decision points (roots).
    - **Ease of Use:** Decision trees can handle various types of data and automatically convert features into binary features.
        - No scaling is required, as the ordering of values remains the same regardless of scaling.
4. **Practical Implementation with scikit-learn:**
    - Import the `DecisionTreeClassifier` class from `sklearn.tree`.
    - Create an instance of the class and set hyperparameters like splitting criterion (`criterion`), max depth (`max_depth`), and maximum features to consider (`max_features`).
    - Fit the model to the training data using the `fit()` method.
    - Make predictions on the holdout set using the `predict()` method.
    - Tune hyperparameters using cross-validation to optimize model performance.
    - For regression tasks, use the `DecisionTreeRegressor` class with similar hyperparameters, but with a continuous outcome variable.

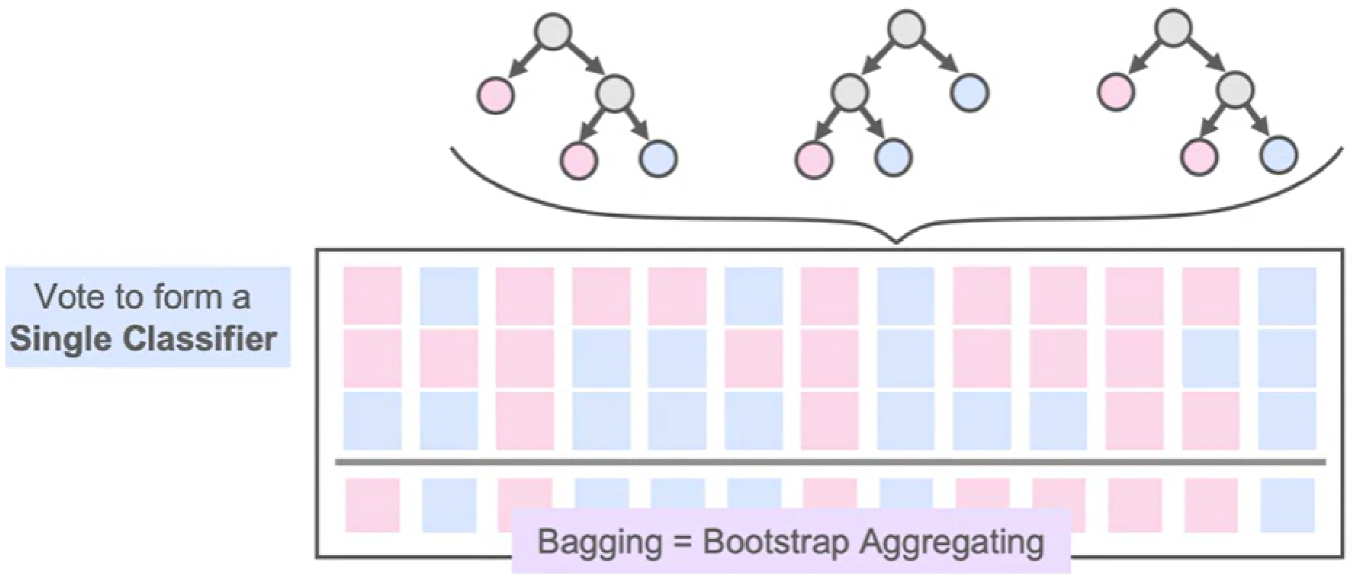
3.7 Ensemble Based Methods

Bootstrap Aggregation (Bagging):

1. **Voting Mechanism:**
    - Each decision tree trained on a bootstrap sample contributes a vote to the final decision.
    - For a given instance in the dataset, predictions from all trees are considered.
    - The majority class among the predictions becomes the final prediction for that instance.
2. **Meta Classification:**
    - The process of determining the majority class based on the aggregated predictions is called meta classification.
    - Meta classification combines the outputs of multiple classifiers to make a final decision.

3. **Example:**
    - Suppose we have three decision trees.
    - For a specific instance, Tree 1 predicts "Red," Tree 2 predicts "Red," and Tree 3 predicts "Blue."
    - By majority voting, the final prediction for that instance is "Red" since two out of three trees predict "Red."
4. **Bagging (Bootstrap Aggregating):**
    - Bagging is short for bootstrap aggregating, which encompasses the entire process described above.
    - Bootstrap sampling involves creating multiple random samples (with replacement) from the original dataset.
    - Decision trees are built on each bootstrap sample independently.
    - Aggregating involves combining the predictions of all decision trees to make a final prediction.
5. **Workflow:**
    - Bootstrap: Generate multiple bootstrap samples from the original dataset.
    - Decision Tree Training: Build a decision tree for each bootstrap sample.
    - Voting: Combine predictions from all decision trees using majority voting.
    - Meta Classification: Determine the final prediction based on the majority vote.

Bagging enables the creation of an ensemble model that leverages multiple decision trees to improve predictive accuracy and generalization. By aggregating the predictions of individual trees through majority voting, bagging reduces the variance and enhances the robustness of the model.



Determining the Number of Trees in Bagging:

1. **Hyperparameter Tuning:**
    - The number of trees in bagging is a hyperparameter that needs to be tuned.
    - More trees generally lead to less overfitting in bagging.
    - However, there's a point of diminishing returns, usually around 50 trees.
2. **Similarities with Decision Trees:**
    - Bagging trees, like basic decision trees, are easy to interpret and implement.
    - They can handle different data types without requiring preprocessing.
3. **Differences from Decision Trees:**
    - Reduced Variability: Bagging reduces variance, thereby decreasing the chances of overfitting.

- Efficiency: Bagging is computationally efficient, especially for growing trees in parallel, compared to boosting methods.

4. **Parallel Tree Growth:**
    - Bagging allows for growing trees in parallel, where each tree is independent of others as it's specific to its own dataset.
    - This parallelization contributes to the computational efficiency of bagging.

Code Implementation for Bagging:

1. **Importing Bagging Classifier:**
    - Import the `BaggingClassifier` class from `sklearn.ensemble`.
2. **Setting Hyperparameters:**
    - Define the number of estimators (trees) as the main hyperparameter.
    - Typically set the number of estimators to optimize model performance.
3. **Fitting and Predicting:**
    - Instantiate the `BaggingClassifier` object with the desired hyperparameters.
    - Fit the model to the training set using the `fit()` method.
    - Make predictions on the test set using the `predict()` method.
4. **Parameter Tuning with Cross-Validation:**
    - Tune hyperparameters, including the number of estimators, using cross-validation.
    - Follow standard machine learning optimization steps for parameter tuning.
5. **Regression Variant:**
    - For regression tasks, use `BaggingRegressor` instead of `BaggingClassifier`.

Bagging offers reduced variability and improved computational efficiency compared to basic decision trees, making it a valuable technique in ensemble learning.

Role of Random Forests:

1. **Variance Reduction in Bagging:**
    - In bagging, if we have n independent trees, each with variance Sigma squared, the bagged variance is reduced to Sigma squared divided by n.
    - However, due to sampling with replacement, trees in bagging are highly correlated, leading to less reduction in variance.
2. **Introducing More Randomness:**
    - To ensure trees in ensemble methods like random forests are significantly different and decorrelated, we introduce more randomness.
    - Random forests restrict the number of features trees are allowed to be built from, selecting a random subset of both rows and columns.
    - The resulting algorithm is called random forest, which combines bootstrapping and aggregating with random subsets of rows and columns.

Determining the Number of Trees for Random Forest:

- Similar to bagging, the number of trees in random forest is determined by observing when the out-of-bag error tends to plateau.
- Any additional trees beyond this point won't significantly improve results.

Implementing Random Forest in Practice:

- Import the `RandomForestClassifier` class from `sklearn.ensemble`.
- Set hyperparameters such as the number of estimators (trees).

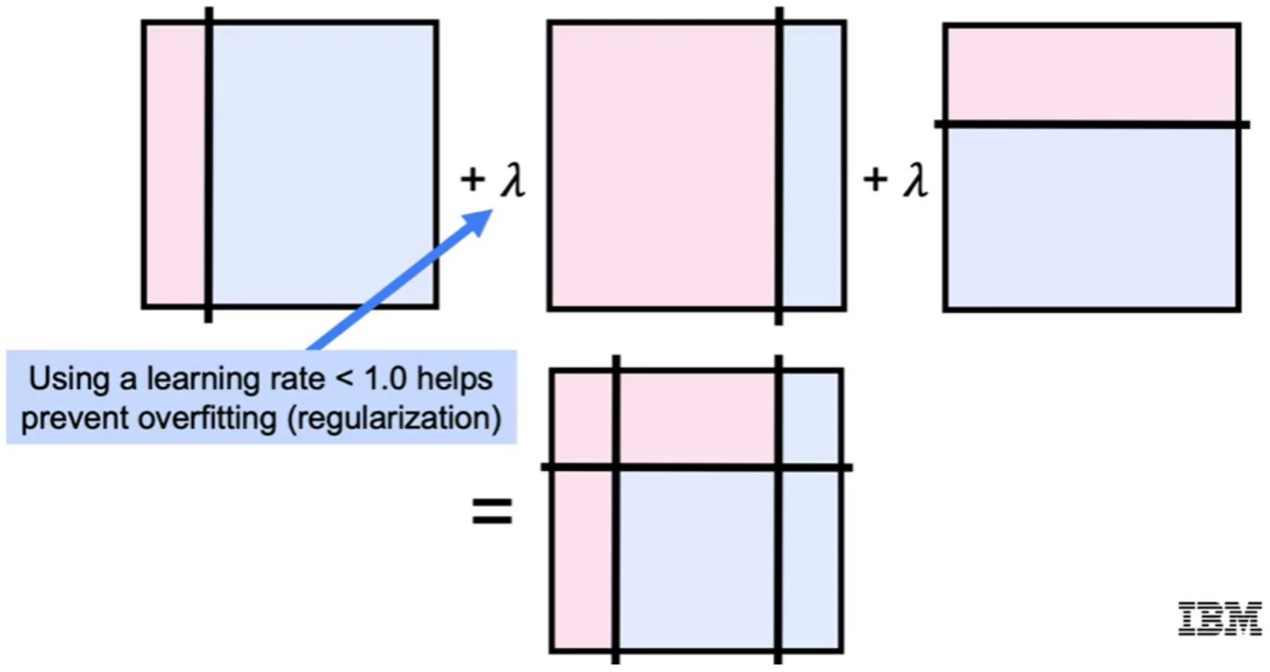
- Fit the model to the training set and make predictions on the test set.
- Use cross-validation to tune hyperparameters for better out-of-sample fits.
- For regression tasks, use `RandomForestRegressor` instead of `RandomForestClassifier`.

Further Variance Reduction with Extra Trees:

- In cases where random forest does not reduce variance enough, even more randomness can be introduced.
- Extra Trees classifiers randomly select the splits in decision trees, instead of using a greedy search.
- These extra random trees are called Extra Trees classifiers.

Boosting:

In boosting, the process involves iteratively creating weak learners, typically decision stumps, and combining their predictions to form a strong classifier. Here's an overview of how boosting works:


$+ \lambda$
$+ \lambda$
Using a learning rate < 1.0 helps prevent overfitting (regularization)
=
IBM

1. **Initial Weak Learner:**
    - Start with an initial weak learner, such as a decision stump, which divides the dataset into two regions based on a single feature.
2. **Weighted Training:**
    - Train the weak learner on the dataset, where each sample is assigned a weight. Initially, all weights are equal.
3. **Error Analysis:**
    - Evaluate the weak learner's performance and identify misclassified samples. These misclassified samples are given higher weights for the next iteration.
4. **Subsequent Weak Learners:**
    - Train additional weak learners, each focusing more on correcting the mistakes of the previous learners. The weights of the samples are adjusted at each iteration to prioritize the misclassified samples.
5. **Combining Predictions:**
    - Combine the predictions of all weak learners using a weighted sum or voting mechanism to form the final prediction.
6. **Learning Rate:**
    - Introduce a learning rate hyperparameter to control the contribution of each weak learner to the final prediction. A smaller learning rate means slower correction of

errors and potentially less overfitting.

7. **Regularization:**
    - Regularization techniques, such as limiting the number of iterations or pruning weak learners, can be applied to prevent overfitting.

Boosting aims to iteratively improve the model's performance by focusing on correcting the errors made by previous weak learners. By combining multiple weak learners, each contributing its own decision boundary, boosting can create a complex final decision boundary that effectively captures the underlying patterns in the data.

In boosting, different loss functions are used to determine the margin for each point in the dataset, which is then used to calculate the error. Here's an overview of the loss functions commonly used in boosting:

1. **Zero-One Loss:**
    - The zero-one loss function returns a value of 1 for incorrectly classified points and ignores correctly classified ones. However, it is not used in practice due to its non-differentiability and difficulty in optimization.
2. **Exponential Loss (AdaBoost):**
    - AdaBoost, or adaptive boosting, uses an exponential loss function. This loss function heavily penalizes misclassified points with large margins, making AdaBoost sensitive to outliers.
3. **Log Likelihood Loss (Gradient Boosting):**
    - Gradient boosting algorithms, such as Scikit-Learn's gradient boosting, typically use a log likelihood loss function. This loss function assigns reduced values for large margins of misclassified points, making gradient boosting more robust to outliers compared to AdaBoost.

Comparing bagging and boosting:

- **Base Learners:**
    - In bagging, base learners (typically decision trees) are independent from each other and are often full trees.
    - In boosting, base learners are weak learners (e.g., decision stumps) created successively, with each learner building on the mistakes of previous learners.
- **Data Usage:**
    - Bagging trains each classifier on bootstrapped samples, while boosting considers the entire dataset for training each classifier. Boosting also takes into account the residuals from previous models when building each successive learner.
- **Weighting:**
    - Bagging assigns equal weights to all samples for each classifier's decision, leading to an equal vote. In boosting, previous mistakes are weighted more heavily at each iteration to correct them, resulting in different weights for each weak learner.
- **Risk of Overfitting:**
    - Bagging does not typically lead to overfitting, as each classifier is trained independently.
    - Boosting can lead to overfitting if the number of trees is too high, as each successive tree aims to correct the errors made by previous trees.

To mitigate overfitting in boosting, hyperparameters such as the learning rate and the number of trees should be optimized using techniques like cross-validation. Additionally, parameters like subsample and max_features can be used to add randomness and reduce model complexity, respectively.

Overall, the choice between bagging and boosting depends on the specific requirements of the problem and the characteristics of the dataset. Both methods offer effective ensemble learning techniques for improving model performance.

Here's the Python syntax for implementing gradient boosting and AdaBoost classifiers using scikit-learn:

For Gradient Boosting:

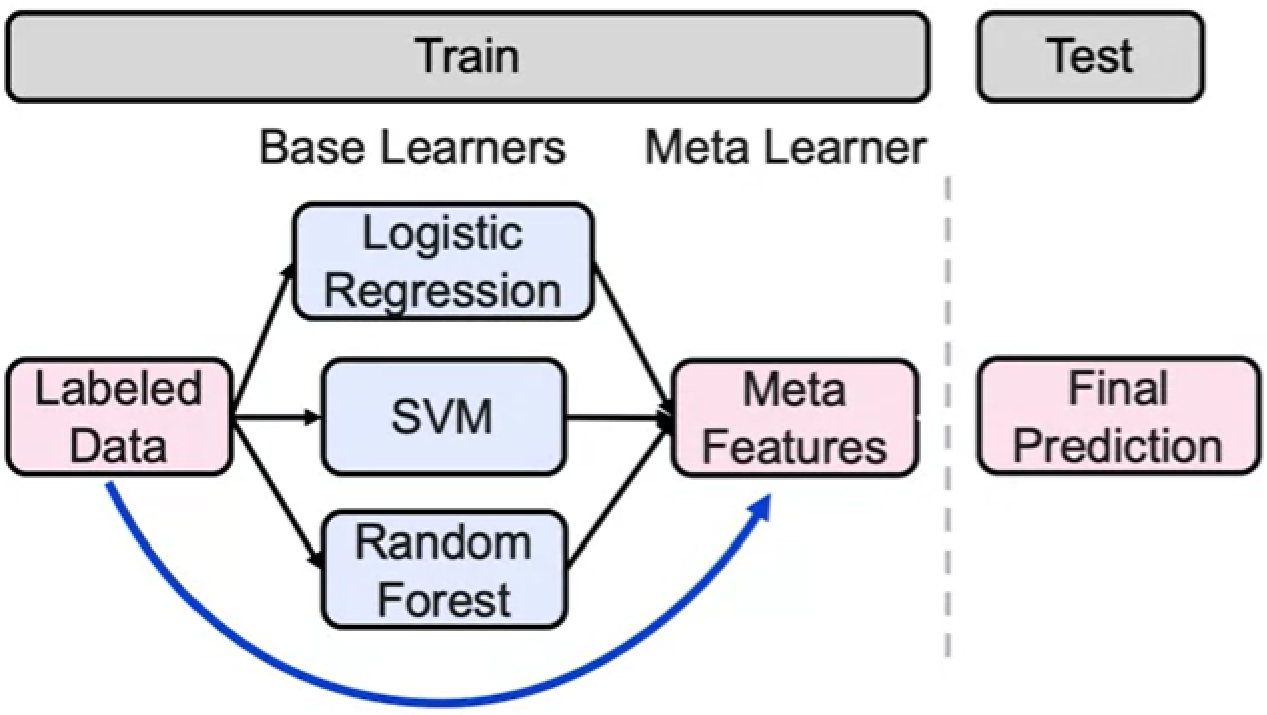
```
from sklearn.ensemble import GradientBoostingClassifier

# Initialize the GradientBoostingClassifier with hyperparameters
GBC = GradientBoostingClassifier(learning_rate=0.1, max_features=1,
subsample=0.5, n_estimators=200)

# Fit the model on the training set
GBC.fit(X_train, y_train)

# Predict on the test set
y_pred = GBC.predict(X_test)

# Tune hyperparameters using cross-validation

# For regression, use GradientBoostingRegressor
```

For AdaBoost:

```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier

# Initialize the AdaBoostClassifier with hyperparameters
# Optionally, you can specify the base estimator and its parameters
base_estimator = DecisionTreeClassifier(max_depth=1)
ABC = AdaBoostClassifier(base_estimator=base_estimator, learning_rate=0.1,
n_estimators=200)

# Fit the model on the training set
ABC.fit(X_train, y_train)

# Predict on the test set
y_pred = ABC.predict(X_test)

# Tune hyperparameters using cross-validation

# For regression, use AdaBoostRegressor
```

In both cases, you can tune the hyperparameters such as learning rate, number of estimators, and others using cross-validation to optimize the model's performance. Additionally, you can use different base estimators and specify their parameters to customize the boosting algorithm according to your needs.

Stacking:



Stacking is an ensemble learning technique that combines multiple base models with a meta-learner to improve predictive performance. Here's a summary of how stacking works and how to implement it in Python:

1. **Base Learners**: The base learners can be any machine learning algorithm. In the example provided, logistic regression, support vector machines (SVM), and random forests are used as base learners. There's no restriction on the choice of base learners, and even ensemble methods like random forests can be included.
2. **Training Base Learners**: Train each base learner on the training data and obtain their predictions or scores on the validation set. These predictions serve as new features, known as meta-features.
3. **Meta-Learner**: A meta-learner is then trained on the meta-features to make the final predictions. The meta-learner can be any machine learning model, such as logistic regression, SVM, or even another ensemble method like random forests.
4. **Aggregation Step**: The final step involves aggregating the predictions from the base learners using the meta-learner. This aggregation can be done using methods like majority voting or weighted voting, similar to other ensemble methods.
5. **Hyperparameter Tuning**: Just like other ensemble methods, hyperparameters of both base learners and the meta-learner need to be optimized using cross-validation. It's crucial to have hold-out sets not only for the final classification method but also for the base learners to properly learn their parameters.

Here's how to implement stacking using scikit-learn:

```
from sklearn.ensemble import StackingClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC

# Define base learners
base_learners = [
    ('lr', LogisticRegression()),
    ('svm', SVC()),
    ('rf', RandomForestClassifier())
]

# Initialize the StackingClassifier with base learners and meta-learner
```

```python
stacked_classifier = StackingClassifier(
    estimators=base_learners,
    final_estimator=LogisticRegression(),  # Choose any final estimator
    stack_method='auto',  # Method for stacking, auto selects the best method
    cv=5  # Number of folds for cross-validation
)

# Fit the StackingClassifier on the training set
stacked_classifier.fit(X_train, y_train)

# Predict on the test set
y_pred = stacked_classifier.predict(X_test)

# Tune hyperparameters using cross-validation

# For regression, use StackingRegressor
```

In the example above, `LogisticRegression`, `SVC`, and `RandomForestClassifier` are chosen as base learners. The `final_estimator` parameter specifies the meta-learner, which in this case is another `LogisticRegression` model. Hyperparameters such as the number of folds for cross-validation (`cv`) and the stacking method (`stack_method`) can also be specified.

3.8 Model Interpretability

Importance of Model Interpretability:

- Understanding machine learning model outcomes is crucial, especially as models become more complex.
- Interpretability helps understand model structures, important features, and how features influence predictions.
- Understanding model workings can provide valuable insights, particularly in sensitive domains like healthcare and finance.
- Interpretability builds trust in the model's correctness and aids in debugging and fixing issues that may arise in production.

Self-Interpretable Models:

- These models have simple and intuitive structures that are easily understood by humans without additional explanation methods.
- They are preferred in high-risk domains because humans can comprehend the model's logic and make similar predictions.
- **Linear Models**: Simple linear models are easy to understand as they involve a linear combination of features. For example, a linear regression model predicting house prices based on features like the number of rooms and high school ranking.
- **Tree Models (Decision Trees)**: Decision trees mimic human reasoning by creating IF-THEN-ELSE rules. They are easy to interpret, but large trees can become complex and prone to overfitting.
- **K-Nearest Neighbor (KNN) Model**: KNN predicts based on similarity to nearby instances. It's interpretable if the feature space is small, but can become complex with a large feature space.

Non-Self-Interpretable Models:

- These models have complex structures and are often described as black-box models.

- They are used because they can achieve state-of-the-art performance in specific problems like natural language translation, image recognition, and traffic patterns.
- **Random Forest**: Random forests are ensembles of decision trees. Each tree generates a prediction, and the final prediction is based on a voting process. It's difficult to understand each tree's contribution to the final prediction.
- **Non-linear Support Vector Machines (SVM)**: SVMs create complex decision boundaries in high-dimensional spaces, making them difficult to interpret.
- **Deep Neural Networks**: DNNs consist of multiple layers of interconnected neurons, making them highly complex and challenging to interpret.

Interpretation Methods:

- **Intrinsic Methods**: Applied to self-interpretable models, such as linear models and decision trees, to simplify the model structure using techniques like regularization or pruning.
- **Post-hoc Methods**: Used to interpret non-self-interpretable models after they have been trained. These methods aim to provide explanations for model predictions by analyzing model behavior and identifying important features.

Model-Agnostic Explanation Methods:

- These methods provide a unified approach to explain various machine learning models, making the interpretability process more manageable and consistent.
- They produce explanations that have the same formats and presentations across different models.

Permutation Feature Importance:

- This method identifies the most salient features of a model by measuring how much the model's performance decreases when the values of a feature are randomly shuffled.
- The importance of a feature is calculated based on the difference in prediction errors before and after permutation.
- Feature importance can be visualized using bar charts or box plots, ranking features by their importance.

Partial Dependency Plot (PDP):

- PDP is used to illustrate the relationship between a feature and the model's outcome.
- It shows how the model's prediction changes as the value of a particular feature changes, while keeping other features unchanged.
- By visualizing the marginal effects of a feature using a line chart, we can observe the relationship between the feature and the model outcome.
- PDP helps understand whether the relationship between the feature and the outcome is linear, non-linear, or monotonic.

Surrogate Models:

- Surrogate models are simpler models trained to approximate the behavior of complex black-box models, making them more interpretable.
- These models provide insights into how inputs are mapped to outputs in black-box models.

Global Surrogate Models:

- Global surrogate models approximate the black-box model's behavior across the entire dataset.
- Steps to build a global surrogate model:
  - Select a dataset X as input.
  - Use the black-box model to make predictions Y using dataset X.
  - Train a simple surrogate model using X and Y.
  - Measure the difference between the predictions of the surrogate model (Y') and the black-box model (Y).

Local Surrogate Models (LIME):

- LIME is a method for building local surrogate models to explain specific instances.
- Steps to build a local surrogate model using LIME:
  - Select a representative data instance.
  - Generate an artificial dataset by perturbing the features around the selected instance.
  - Weight the instances based on their similarity to the seed instance.
  - Train a surrogate model using the weighted dataset.
- LIME provides insights into how the black-box model makes predictions on individual instances.

Interpreting LIME Results:

- LIME produces visualizations that show the contribution of each feature to the model's prediction for a specific instance.
- Features contributing positively are labeled in green, while those contributing negatively are labeled in red.
- By analyzing the LIME outcome, we can construct a narrative explaining why the black-box model made a particular prediction for the instance.

3.9 Modeling Unbalanced Classes

Challenges of Unbalanced Classes:

- Classifiers optimized for accuracy may perform poorly on underrepresented classes in highly unbalanced datasets.
- Metrics other than accuracy are needed to effectively evaluate model performance on unbalanced data.

Solutions to Unbalanced Datasets:

- **Downsampling**: Taking a random subset of the majority class to match the size of the minority class, creating a balanced dataset.
- **Upsampling**: Duplicating instances of the minority class to match the size of the majority class, creating a balanced dataset.
- **Resampling**: Combining upsampling and downsampling to achieve a balanced dataset by adjusting the sizes of both classes.

These strategies aim to balance the class distribution in the dataset before fitting the model, thereby improving the model's ability to learn from both classes equally.

Steps and Considerations:

1. **Train-Test Split**:
   - Perform a stratified train-test split to maintain class balance in both sets.

2. **Sampling**:
    - **Downsampling**:
        - Reduce the majority class to match the size of the minority class by randomly selecting a subset of majority class instances.
        - Increases recall but decreases precision.
    - **Upsampling**:
        - Increase the minority class to match the size of the majority class by duplicating minority class instances.
        - Mitigates excessive weight on the minority class, but may lead to overfitting.
        - Increases recall but may slightly decrease precision.
    - **Resampling**:
        - A combination of upsampling and downsampling to achieve a desired balance between the classes.
3. **Effect on Metrics**:
    - Upsampling increases recall but may decrease precision, while downsampling increases precision but may decrease recall.
    - The choice between upsampling and downsampling depends on the specific trade-offs and goals of the analysis.
4. **Cross-Validation**:
    - Use cross-validation to evaluate the performance of different sampling techniques.
    - Assess metrics such as ROC curves to determine the effectiveness of each sampling method.
5. **Model Selection**:
    - Choose the best model based on evaluation criteria such as accuracy or area under the ROC curve (AUC).
    - Consider the business objectives and the importance of precision and recall in the context of the application.
6. **Threshold Selection**:
    - Depending on business objectives, select an appropriate threshold value for the classifier based on the ROC curve.
    - Balance between precision and recall according to the specific needs of the problem.

Sampling Techniques:

- **Random Oversampling**: Duplicating instances from the minority class to balance class distribution.
- **Synthetic Oversampling**: Generating synthetic instances for the minority class using techniques like SMOTE (Synthetic Minority Over-sampling Technique).
- **Under Sampling**: Reducing the number of instances from the majority class to balance class distribution.
- **Combination Sampling**: Utilizing a mix of oversampling and undersampling techniques.

Ensemble Methods:

- Leveraging oversampling or undersampling techniques within ensemble methods to ensure balance between classes.

Class Weight Adjustment:

- Many models allow weighted observations to adjust for class imbalance.
- Setting the `class_weight` hyperparameter to "balanced" distributes weights to each class equally.

Stratified Sampling:

- Ensuring class balance consistency in both train and test sets.
- Options available in Python include using the `stratify` argument in `train_test_split`, `StratifiedShuffleSplit`, `StratifiedKFold`, or `RepeatedStratifiedKFold`.

Random Oversampling:

- Involves randomly duplicating instances from the minority class with replacement.
- Best suited for categorical data where the distance between features may not have interpretive significance.

Synthetic Oversampling:

- Involves creating new samples for the minority class that do not exist in the original dataset.
- Uses the concept of K nearest neighbors (KNN) to generate synthetic samples.
- Steps:
    - Start with a point from the minority class.
    - Choose one of its K nearest neighbors.
    - Generate a new point randomly along the line connecting the two points.
    - Repeat this process for each neighbor.
- Two main approaches: SMOTE and ADASYN.

SMOTE (Synthetic Minority Over-sampling Technique):

- Can be divided into subsets:
    - Regular SMOTE: Connects minority class points to any neighbors, even those of other classes, and generates new points.
    - Borderline SMOTE: Classifies points as outliers, safe, or in-danger based on their nearest neighbors.
        - Borderline 1 SMOTE: Connects minority in-danger points only to minority points.
        - Borderline 2 SMOTE: Connects minority in-danger points to nearby points regardless of class.
    - SVM SMOTE: Uses a support vector machine classifier to find support vectors and generate samples.

ADASYN (Adaptive Synthetic Sampling):

- Similar to SMOTE but generates more samples in areas where the nearest neighbor rule is not respected.
- The number of samples generated for each point is proportional to the number of samples not from the same class in its neighborhood.

Undersampling Techniques:

1. **NearMiss Algorithm**:
    - NearMiss selects positive samples (majority class) based on their proximity to negative samples (minority class).
    - NearMiss-1 selects positive samples for which the average distance to the N closest negative samples is the smallest.
    - NearMiss-2 selects positive samples for which the average distance to the farthest negative samples is the smallest.

    - NearMiss-3 is a two-step algorithm: first, it finds the K-nearest neighbors of each negative sample, then selects positive samples based on the largest average distance to these neighbors.
2. **Tomek Links**:
    - A Tomek link exists if two samples from different classes are each other's nearest neighbors.
    - Removing Tomek links helps in creating more distinct classes.
    - Removal can involve either removing the majority class alone or both classes.
3. **Edited Nearest Neighbors (ENN)**:
    - Uses K-nearest neighbors with K equal to 1.
    - If a point in the majority class is misclassified, it is removed.

These undersampling techniques aim to reduce the size of the majority class to balance the dataset and create more distinct classes. However, they can still be affected by outliers or noise in the data.

Mixing Oversampling and Undersampling:

- SMOTE can be used for oversampling, while Tomek links or Edited Nearest Neighbors can be used for downsampling the majority class.
- Balanced bagging involves continuously downsampling each bootstrap sample to ensure balance before training decision trees.

Steps to Keep in Mind:

- Always perform the train-test split before any oversampling or undersampling to avoid data leakage.
- Choose sensible evaluation metrics such as AUC, precision-recall curve, F1 score, or Cohen's Kappa, as accuracy can be misleading with unbalanced classes.

Evaluation Metrics:

- AUC provides a trade-off between true positive rate and false positive rate and helps in selecting the optimal threshold for classification.
- Precision-recall curve and F1 score balance precision and recall, which are crucial for unbalanced classes.
- Cohen's Kappa measures agreement between models and is useful for assessing model performance.

Choosing between Oversampling and Undersampling:

- Undersampling may lead to higher recall for the minority class but lower precision, while oversampling maintains precision but may have lower recall.