Table of Contents

Data Archive Book

Welcome to the Data Archive Book.

Preface

Remember the Data Archive is non-linear — use internal links for navigation. If you need a specific page see the **Table of Contents** to jump to a topic.

Accessing Gen Ai Generated Content

To assess whether the content generated by a Generative AI is truthful and faithful, several methods and frameworks can be employed. Truthfulness refers to whether the generated content is factually correct, while faithfulness refers to whether it accurately reflects the input data or prompt.

interpretability

1. Frameworks for Truthfulness and Faithfulness

- Subject Matter Expert (SME) Reviews: One of the most reliable methods for verifying truthfulness and faithfulness is through SME validation. SMEs can manually check the content to ensure it aligns with domain-specific knowledge and is factually accurate.
- Knowledge Graph and External Data: Generative AI models can be linked to external sources of truth, such as knowledge graphs, databases, or other verified resources. This allows the system to cross-check facts and improve the truthfulness of the content.
- Retrieval-Augmented Generation (RAG): This framework involves retrieving relevant information from trusted sources before generating content. It helps ensure that the AI is providing up-to-date, reliable, and contextually accurate responses.
- Evaluation Metrics: Some metrics can be used to evaluate faithfulness:
    - Factual Consistency Metrics: Tools such as BERTScore or FactCC can compare generated text with reference text or factual databases to check for consistency.
    - Human Evaluation: In certain contexts, human evaluators rate the content on aspects of truthfulness and faithfulness. This can be part of quality assurance processes.
- Cross-Referencing Data: AI-generated content should be cross-referenced with existing, credible sources to confirm its accuracy. For example, if the AI makes a historical claim or provides statistical data, those facts should be verifiable through known data repositories.
- Fact-Checking Tools: Using automated fact-checking tools or models trained to detect false information can provide another layer of defence against untruthful content.

Accuracy

Definition

- Accuracy Score is the proportion of correct predictions out of all predictions made. In other words, it is the percentage of correct predictions.
- Accuracy can have issues with Imbalanced Datasetswhere there is more of one class than another.

Formula

- The formula for accuracy is: $\text{Accuracy} = \frac{TN+TP}{\text{Total}}$ In the context of Classification problems, particularly binary classification, TN and TP are components of the confusion matrix:
- TP (True Positive): The number of instances that are correctly predicted as the positive class. For example, if the model predicts a positive outcome and it is indeed positive, it counts as a true positive.
- TN (True Negative): The number of instances that are correctly predicted as the negative class. For example, if the model predicts a negative outcome and it is indeed negative, it counts as a true negative.

The Confusion Matrix also includes:

- FP (False Positive): The number of instances that are incorrectly predicted as the positive class. This is also known as a "Type I error."
- FN (False Negative): The number of instances that are incorrectly predicted as the negative class. This is also known as a "Type II error."

These metrics are used to evaluate the performance of a classification model, providing insights into not just accuracy but also precision, recall, and other performance measures.

Exploring Accuracy in Python

To explore accuracy in Python, you can use libraries such as `scikit-learn`, which provides the `accuracy_score` function. This function compares the predicted labels with the true labels and calculates the accuracy.

Example Usage

```
from sklearn.metrics import accuracy_score

# Assuming pred and y_test are defined

accuracy = accuracy_score(y_test, pred)

print("Prediction accuracy: {:.2f}%".format(accuracy  100.0))
```

- Make sure to replace `pred` and `y_test` with your actual prediction and test data variables.

Activation Function

Activation functions play a role in Neural network by introducing non-linearity, allowing models to learn from complex patterns and relationships in the data.

How do we choose the right Activation Function

Key Uses of Activation Functions:

1. Non-linearity: Without activation functions, neural networks would behave as linear models, unable to capture complex, non-linear patterns in the data
2. Data transformation: Activation functions modify input signals from one layer to another, helping the model focus on important information while ignoring irrelevant data,
3. Backpropagation: They enable gradient-based optimization by making the network differentiable, essential for efficient learning.

Purpose of Typical Activation Functions

Linear: Outputs a continuous value, suitable for regression.

ReLU (Rectified Linear Unit):

- Purpose: ReLU is used to introduce non-linearity by turning neurons "on" or "off." It outputs the input directly if it is positive; otherwise, it outputs zero. This helps in efficiently training deep networks by mitigating the vanishing gradient problem.
- Function: $f(x) = \max(0, x)$

Sigmoid:

- Purpose: Sigmoid is used primarily in Binary Classification tasks. It squashes input values to a range between 0 and 1, making it suitable for representing probabilities.
- Function: $f(x) = \frac{1}{1+e^{-x}}$

  Tanh:

- Purpose: Tanh is similar to the sigmoid function but outputs values in the range of -1 to 1. This zero-centered output can be beneficial for optimization in certain scenarios.
- Function: $f(x) = \tanh(x)$

Softmax:

- Purpose: Softmax is used in multi-class classification tasks. It converts a vector of raw scores (logits) into a probability distribution, where each value is between 0 and 1, and the sum of all values is 1. This allows the outputs to be interpreted as probabilities, with larger inputs corresponding to larger output probabilities.
- Application: In both softmax regression and neural networks with softmax outputs, a vector $\mathbf{z}$ is generated by a linear function and then passed through the softmax function to produce a probability distribution. This enables the selection of one output as the predicted category.

The softmax function converts a vector of raw scores (logits) into a probability distribution. The formula for the softmax function for a vector $\mathbf{z} = [z_1, z_2, \ldots, z_N]$ is given by:

$\sigma(\mathbf{z})_i = \frac{e^{zi}}{\sum_{j=1}^{N} e^{z_j}}$

This ensures that the output values are between 0 and 1 and that they sum to 1, making them interpretability|interpretable as probabilities.

Active Learning

Think captchas for training.

To help the Supervised Learning models when they are less confident.

Reducing labelling time or need for it.

Ada Boosting

Resources: LINK

Overview:

Ada Boosting short for Adaptive Boosting, is a specific type of Boosting algorithm that focuses on improving the accuracy of predictions by combining multiple weak learners into a strong learner. It is particularly known for its simplicity and effectiveness in classification tasks.

How AdaBoost Works:

1. **Base Learners**: In AdaBoost, the base learners are typically low-depth trees, also known as stumps. These are simple models that perform slightly better than random guessing.
2. **Sequential Training**: AdaBoost trains these stumps sequentially. Each stump is trained to correct the errors made by the previous stumps. This sequential approach ensures that each new model focuses on the data points that were misclassified by earlier models.
3. **Weighting**: After each stump is trained, AdaBoost assigns a weight to it based on its accuracy. More accurate stumps receive higher weights, giving them more influence in the final prediction.
4. **Error Focus**: The algorithm increases the weights of the misclassified data points, making them more prominent in the training of the next stump. This ensures that subsequent models pay more attention to the difficult-to-classify instances.
5. **Final Prediction**: The final prediction is a weighted sum of the predictions from all the stumps. The stumps with higher accuracy have more say in the final classification.

Further Understanding

Creating a Forest with AdaBoost:

To create a forest using AdaBoost, you start with a Decision Tree or Random Forests approach, but instead of using full-sized trees, you use stumps.

These stumps are trained sequentially, with each one focusing on the errors of the previous stumps.

The final prediction is a weighted sum of the predictions from all the stumps, where more accurate stumps have more influence on the final outcome.

Key Differences from Random Forests:

- **Tree Depth**: In Random Forests, full-sized trees are used, and each tree gets an equal say in the final prediction. In contrast, AdaBoost uses low-depth trees (stumps) and assigns different weights to each stump based on its accuracy.
- **Order and Sequence**: In AdaBoost, the order of the stumps is important because errors are passed on in sequence. In Random Forests, trees are built independently and simultaneously.

Advantages of AdaBoost:

- **Increased Accuracy**: By focusing on the errors of previous models, AdaBoost can significantly improve the accuracy of predictions.
- **Simplicity**: AdaBoost is relatively simple to implement and understand compared to other ensemble methods.
- **Flexibility**: It can be applied to various types of base models and is not limited to a specific algorithm.

Challenges of AdaBoost:

- **Sensitivity to Noisy Data**: AdaBoost can be sensitive to noisy data and outliers, as it focuses heavily on correcting errors.
- **Complexity**: While simpler than some other boosting methods, AdaBoost can still be computationally intensive due to its sequential nature.

Adjusted R Squared

Adjusted R-squared is a Regression Metricsor assessing the quality of a regression model, especially when multiple predictors are involved. It helps ensure that the model remains parsimonious while still providing a good fit to the data.

When evaluating a regression model, if you notice a large difference between R squared and adjusted R², it indicates that the additional predictors may not be improving the model's performance.

In such cases, it may be beneficial to drop those extra variables to simplify the model without sacrificing predictive power.

Key features:

1. **Penalty for Number of Predictors**:
    - Adjusted R² adjusts the R² value by penalizing the addition of unnecessary predictors. This means that if you add a variable that does not improve the model significantly, the adjusted R² will decrease, reflecting that the model may be overfitting.
2. **Comparison with R-squared**:
    - Adjusted R² is always less than or equal to R². While R² can artificially inflate with the addition of more predictors (even if they are not useful), adjusted R² provides a more reliable assessment of model fit by considering the number of predictors relative to the number of observations.
3. **Interpretation**:
    - Like R², adjusted R² values range from 0 to 1, where values closer to 1 indicate a better fit. However, a significant difference between R² and adjusted R² suggests that the model may be penalized for including extra variables that do not contribute meaningfully to the prediction.
4. **Formula**: $R^2_{adj.} = 1 - (1 - R^2) \cdot \frac{n-1}{n-p-1}$
    - Where:
        - R^2 = R-squared value
        - n = number of observations
        - p = number of predictors (independent variables)

Anomaly Detection With Statistical Methods

Basic:

- Z-Normalisation|Z-Score
- Interquartile Range (IQR) Detection
- Percentile Detection

Advanced:

- Gaussian Model
- Isolated Forest

Grubbs' Test

Context:

Grubbs' test is a hypothesis test designed to detect a single outlier in a normally distributed dataset. It tests the largest deviation from the mean relative to the standard deviation. This test is iterative and removes one outlier at a time.

Purpose:

To determine whether the most extreme data point (either smallest or largest) is a statistical outlier.

Steps:

- Compute the test statistic:

$$G = \frac{\max(|X - \mu|)}{\sigma}$$

  where:
  - X: Data points
  - μ: Mean of the dataset
  - σ: Standard deviation of the dataset.
- Compare G to a critical value:
  - The critical value depends on the sample size n and significance level α (e.g., 0.05).
  - If G exceeds the critical value, the data point is considered an outlier.

Limitations:

- Assumes data follows a normal distribution.
- Inefficient for detecting multiple outliers simultaneously.

Histogram-Based Outlier Detection (HBOS)

Context:

HBOS is a non-parametric method that detects anomalies by analyzing the distribution of individual features independently. It relies on histograms, which estimate feature density.

Purpose:

To identify outliers as data points falling in bins with low frequencies or densities.

Steps:

- Create histograms for each feature:
  - Divide each feature's range into bins.
  - Count the frequency of data points in each bin.
- Calculate scores for each data point:
  - Outliers are points in bins with significantly lower densities compared to others.

Advantages:

- Does not assume a specific data distribution.
- Scales well to large datasets.

Limitations:

- Assumes feature independence (not ideal for multivariate data).
- Sensitive to bin size selection.

One-Class SVM

One-Class Support Vector Machine is a variation of the SVM algorithm used for anomaly detection. It learns a decision boundary around the normal data points.

Steps:

- Train the model on the normal data points.
- The model attempts to find a hyperplane that separates the normal data from the origin.
- Points that fall outside this boundary are classified as anomalies.

Auc

AUC (Area Under the Curve) is a metric for binary classification problems, representing the area under the ROC (Receiver Operating Characteristic)

Key Concepts

Represents the area under the ROC curve.

AUC values range from 0 to 1, where 1 indicates perfect classification and 0.5 suggests no discriminative power (equivalent to random guessing).

Roc and Auc Score

The `roc_auc_score` is a function from the `sklearn.metrics` module in Python that computes the Area Under the Receiver Operating Characteristic Curve (ROC AUC) from prediction scores. It is a widely used metric for evaluating the performance of binary classification models.

Key Points about `roc_auc_score` :

- **Purpose**: It quantifies the overall ability of the model to discriminate between the positive and negative classes across all possible classification thresholds.
- **Range**: The score ranges from 0 to 1, where:
  - 1 indicates perfect discrimination (the model perfectly distinguishes between the positive and negative classes).
  - 0.5 suggests no discriminative power (equivalent to random guessing).

    - Values below 0.5 indicate a model that performs worse than random guessing.
- **Input**: The function takes the true binary labels and the predicted probabilities (or decision function scores) as inputs.
- **Output**: It returns a single scalar value representing the AUC.

Example Code

```
from sklearn.metrics import roc_auc_score


# Actual and predicted values
y_act = [1, 0, 1, 1, 0]
y_pred = [1, 1, 0, 1, 0]


# Compute AUC
auc = roc_auc_score(y_act, y_pred)
print(f'AUC: {auc}')
```

Backpropagation

[!Summary]

Backpropagation is an essential algorithm in the training of neural networks and iteratively correcting its mistakes. It involves a process of calculating the gradient of the loss function $L(\theta)$ concerning each weight in the network, allowing the system to update its weights via Gradient Descent.

This process helps minimize the difference between predicted outputs and actual target values. Mathematically, the chain rule of calculus is employed to propagate errors backward through the network.

Each layer in the network computes a partial derivative that is used to adjust the weights. This iterative approach continues until a convergence criterion is met, typically when the change in loss falls below a threshold.

The backpropagation algorithm is critical in Supervised Learning, where labeled data is used to train models to recognize patterns.

[!Breakdown]

Key Components:

- **Algorithm**: Gradient Descent
- **Mathematical Foundation**: Chain Rule for derivatives
- **Metrics**: Loss function (e.g., Mean Squared Error, Cross-Entropy)

[!important]

- Gradient descent uses $\nabla L(\theta) = \frac{\partial L}{\partial \theta}$ to iteratively minimize the loss.
- Backpropagation optimizes deep learning models by adjusting weights based on error gradients.

[!attention]

- The method is computationally expensive for deep networks due to the need to compute gradients for each layer.
- Vanishing/exploding gradients in deep layers can prevent proper weight updates.

[!Example]

A feed-forward neural network trained on image classification data uses backpropagation to minimize cross-entropy loss. The gradient of the loss is calculated layer by layer, adjusting weights through an optimization algorithm like Adam.

[!Follow up questions]

- How does backpropagation compare with other optimization algorithms such as Newton's method or evolutionary strategies?
- What role does Regularisation play in addressing overfitting when using backpropagation in deep neural networks?

> [!Related Topics]
>
> - Gradient Descent Optimizers (Adam Optimizer, RMSprop)
> - vanishing and exploding gradients problem

Backpropgation

Backpropgation is used to calc the gradient of the loss function with respect to the model parameters, when there are a lot of parameters - i.e in a neural network. Simple example of backpropgation

Simple example of computation graph Computation graph - calcing derivatives? Use sympy to calculate derivatives for the loss function.

> The steps in backprop
>
> Now that you have worked through several nodes, we can write down the basic method:\ working right to left, for each node:
>
> - calculate the local derivative(s) of the node
> - using the chain rule, combine with the derivative of the cost with respect to the node to the right.

The 'local derivative(s)' are the derivative(s) of the output of the current node with respect to all inputs or parameters.

Example of using sympy to calculate derivatives for the loss function. Use `diff` , `subs`

```
from sympy import symbols, diff
```

Sympy

Bagging

Overview:

Bagging, short for Bootstrap Aggregating, is an Model Ensemble technique designed to improve the stability and accuracy of machine learning algorithms.

It works by training multiple instances of the same learning algorithm on different subsets of the training data and then combining their predictions.

How Bagging Works:

1. **Bootstrap Sampling**: Bagging involves creating multiple subsets of the training data by sampling with replacement. This means that each subset, or "bootstrap sample," is drawn randomly from the original dataset, and some data points may appear multiple times in a subset while others may not appear at all.

2. **Parallel Training**: Each bootstrap sample is used to train a separate instance of the same base learning algorithm. These models are trained independently and in parallel, which makes bagging computationally efficient.
3. **Combining Predictions**: Once all models are trained, their predictions are combined to produce a final output. For regression tasks, this is typically done by averaging the predictions. For classification tasks, majority voting is used to determine the final class label.

Key Concepts of Bagging:

- **Reduction of Overfitting**: By averaging the predictions of multiple models, bagging reduces the variance and helps prevent overfitting, especially in high-variance models like decision trees.
- **Diversity**: The use of different subsets of data for each model introduces diversity among the models, which is crucial for the success of ensemble methods.
- **Parallelization**: Since each model is trained independently, bagging can be easily parallelized, making it scalable and efficient for large datasets.

Example of Bagging:

Random Forest: A well-known example of a bagging technique is the Random Forests algorithm.

It uses decision trees as base models and combines their predictions to improve accuracy and robustness.

Each tree in a random forest is trained on a different bootstrap sample of the data, and the final prediction is made by averaging the outputs (for regression) or majority voting (for classification).

Further Understanding

Advantages of Bagging:

- **Increased Accuracy**: By combining multiple models, bagging often achieves higher accuracy than individual models.
- **Robustness**: Bagging is less sensitive to overfitting, especially when using high-variance models like decision trees.
- **Flexibility**: It can be applied to various types of base models and is not limited to a specific algorithm.

Challenges of Bagging:

- **Complexity**: While bagging reduces overfitting, it can increase the complexity of the model, making it harder to interpret.
- **Computational Cost**: Training multiple models can be computationally intensive, although this can be mitigated by parallel processing.

Bias And Variance

Related to Overfitting

Ways to Reduce Bias and Variance:

- Regularisation

What is Bias in Machine Learning?

Bias occurs when a model produces consistently unfair or inaccurate results, usually caused during training due to design choices.

What Does High Bias Mean for a Machine Learning Model?

High bias refers to a situation where a model has a strong and often simplistic assumption about the underlying data, leading to underfitting.

It is biased to the data.

What is the Variance of a Machine Learning Model?

Variance measures how much a model's predictions change when trained on different subsets of the training data. It indicates how much the model overfits the training data.

What is the Difference Between Bias and Variance in Machine Learning?

- Bias: The error that occurs when the model cannot learn the true relationship between input and output variables.
- Variance: The error that arises when the model is too sensitive to the training data and does not generalize well to new data.

Explain the Bias-Variance Trade-off in the Context of Model Complexity:

The bias-variance trade-off describes the relationship between model complexity and performance.

- High bias (underfitting) occurs when a model is too simple, leading to poor performance on both training and test data.
- High variance (overfitting) happens when a model is overly complex, performing well on training data but poorly on unseen data.

Boosting

Boosting is a type of Model Ensemble in machine learning that focuses on improving the accuracy of predictions by building a sequence of models. Each subsequent model focuses on correcting the errors made by the previous ones.

It combines Weak Learners (models that are slightly better than random guessing) to create a strong learner.

Key Concepts of Boosting:

1. Sequential Learning: Boosting involves training models sequentially. Each new model is trained to correct the errors made by the previous models. This means that the models are not independent of each other; instead, each model is built on the mistakes of the previous ones.
2. Focus on Misclassified Data: As models are trained in sequence, more emphasis is placed on the data points that were misclassified by earlier models. This helps the ensemble model to gradually improve its performance by focusing on the difficult-to-classify instances.
3. Weak Learners: Boosting combines multiple weak learners, which are models that perform slightly better than random guessing. By combining these weak learners, boosting creates a strong learner that has improved accuracy.
4. Examples of Boosting Algorithms: Some well-known boosting algorithms include Ada boosting, Gradient Boosting, and XGBoost. Each of these algorithms has its own approach to boosting, but they all share the core principle of sequentially improving model performance.

Advantages of Boosting:

- Increased Accuracy: By focusing on the errors of previous models, boosting can significantly improve the accuracy of predictions.
- Flexibility: Boosting can be applied to various types of base models and is not limited to a specific algorithm.
- Robustness: Boosting can handle complex datasets and is effective in reducing bias and variance.

Challenges of Boosting:

- Complexity: Boosting models can be more complex and computationally intensive than single models.
- Interpretability: The final model may be harder to interpret compared to simpler models like decision trees.

Classification

Classification is a type of Supervised Learning in machine learning, where the algorithm learns from labeled data to predict which category or class a new, unlabeled data point belongs to. The goal is to assign the correct label to input data based on patterns learned from the training set.

Examples of Classifiers

Classifier: A model used for classification tasks, predicting discrete labels or categories. For example, determining whether an email is spam or not, or identifying the species of a flower based on its features. This contrasts with a Regressor (Regression), which predicts continuous values.

Naive Bayes

Decision Tree

Support Vector Machines

K-nearest neighbours

Neural network

Model Ensemble

Choosing a Classifier Algorithm

1. Data Characteristics: Some algorithms work better on structured data, while others perform better on unstructured data.
2. Problem Complexity: Simple classifiers for straightforward problems, complex models for intricate tasks.
3. Model Performance: Consider accuracy and speed requirements.
4. Model Interpretability: Some models, like decision trees, are easier to interpret, while others, like neural networks, can be more challenging.
5. Model Scalability: Large datasets need scalable models like SVM or Naive Bayes.
6. Model Flexibility: Algorithms like KNN are flexible when the data distribution is unknown.

Use Cases of Classification

7. Object Recognition: Classifying objects in images (e.g., identifying a cat or a dog).
8. Spam Filtering: Classifying emails as either spam or legitimate.
9. Medical Diagnosis: Using patient symptoms and test results to classify diseases.

Clustering

Clustering involves grouping a set of data points into subsets or clusters based on inherent patterns or similarities. It is an Unsupervised Learningtechnique used for tasks like customer segmentation and standardised/Outliers|anomalies detection. The primary goal of clustering is to organize data by grouping similar items.

Applications

- Customer Segmentation: Group customers with similar purchasing behavior or demographics for targeted marketing.
- Image Segmentation: Group pixels in an image based on color or texture to identify objects or regions.
- Anomaly Detection: Identify clusters of normal behavior to detect anomalies that deviate significantly from these clusters.

Methods

- K-means
- DBScan
- Hierarchical Clustering
- Gaussian Mixture Models

Interpretability

Feature Scaling: Essential for bringing features to the same scale, as clusters may appear distorted without it.

```
from sklearn.preprocessing import scale
from sklearn.preprocessing import MinMaxScaler
```

Use clustering to find Correlation between features. Utilize a Dendrograms to visualize the relationship between features.

Confusion Matrix

A Confusion Matrix is a table used to evaluate the performance of a Classification model. It provides a detailed breakdown of the model's predictions across different classes, showing the number of true positives, true negatives, false positives, and false negatives.

Purpose

- The confusion matrix helps identify where the classifier is making errors, indicating where it is "confused" in its predictions.

Structure

		Predicted	
		Negative **(N)** **-**	Positive **(P)** **+**
Actual	Negative **-**	True Negative **(TN)**	**False Positive (FP) Type I Error**
	Positive **+**	**False Negative (FN) Type II Error**	True Positive **(TP)**

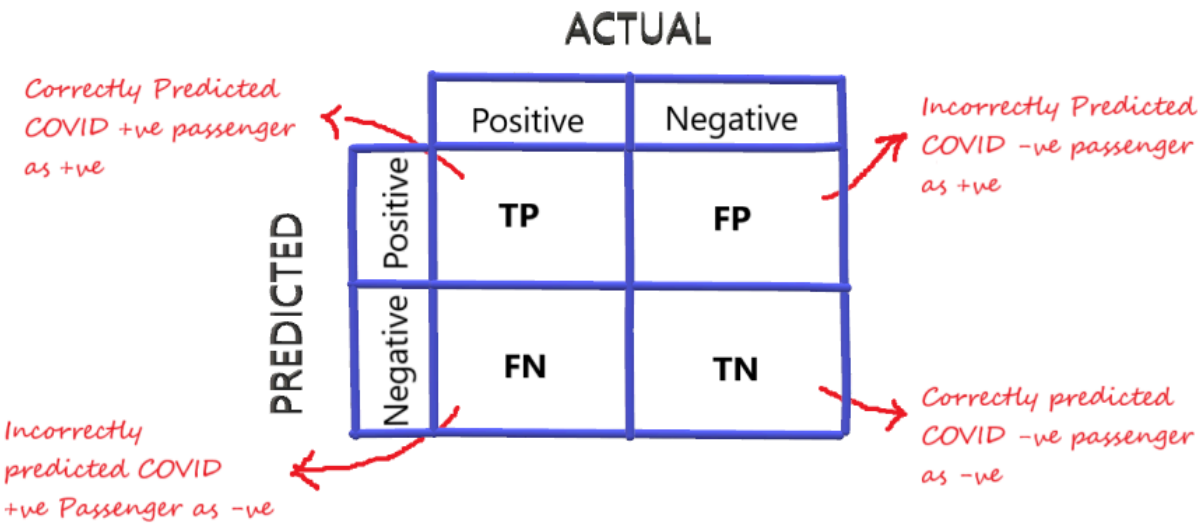
Structure

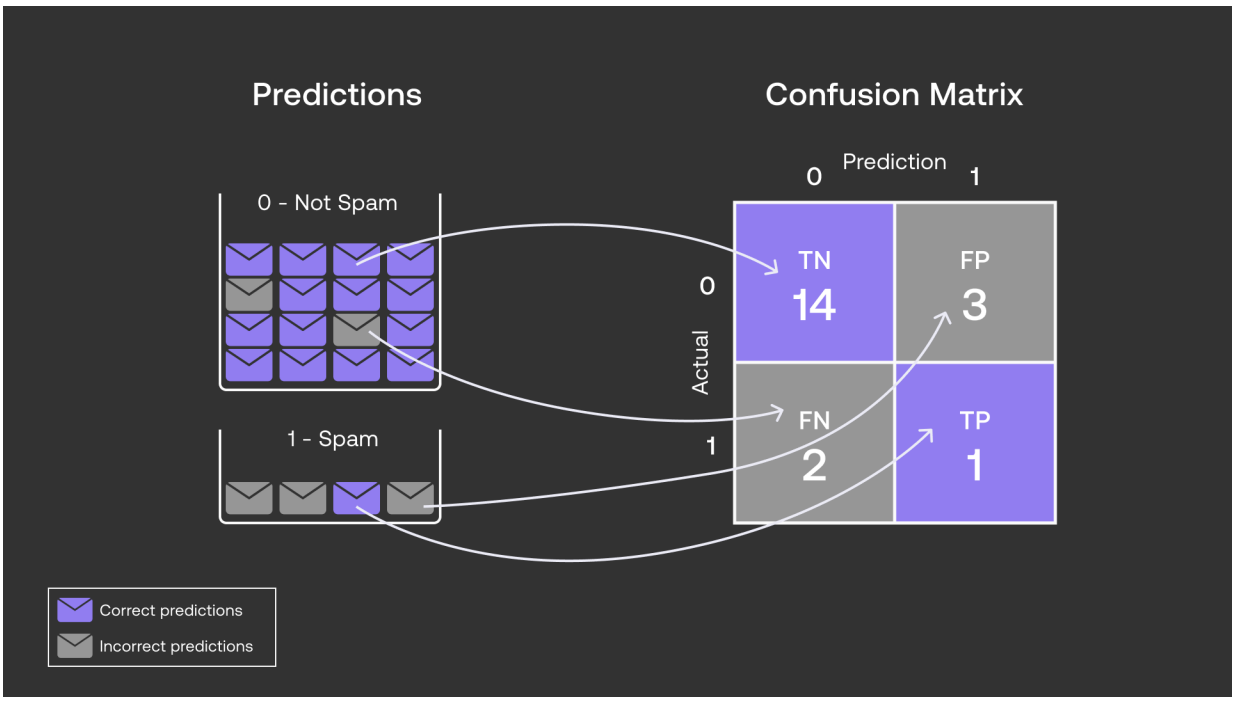
- True Positives (TP): Correctly predicted positive instances.
- False Positives (FP): Incorrectly predicted positive instances (Type 1 error).
- True Negatives (TN): Correctly predicted negative instances.
- False Negatives (FN): Incorrectly predicted negative instances (Type 2 error).

Metrics

- Accuracy: The overall percentage of correct predictions. In this case, the accuracy is 78.3%.
- Precision: The ratio of true positives to all positive predictions (including both TPs and FPs). In this case, the precision for class 0 is 85.7% and the precision for class 1 is 66.4%.
- Recall: The ratio of true positives to all actual positive cases (including both TPs and FNs). In this case, the recall for class 0 is 80.6% and the recall for class 1 is 74.1%.
- F1 Score: A harmonic average of precision and recall. In this case, the F1-score for class 0 is 83.0% and the F1-score for class 1 is 70.0%.
- Specificity
- Recall

Further Examples





Example Code

```
from sklearn.metrics import confusion_matrix


# Assuming y_train and y_train_pred are your true and predicted labels
conf_matrix = confusion_matrix(y_train, y_train_pred)
print(conf_matrix)
```

Example Output:

```
array([[377, 63],
       [ 91, 180]], dtype=int64)
```

Cost Sensitive Analysis

Cost-sensitive analysis in machine learning refers to the practice of incorporating the costs associated with different types of errors into the model training and evaluation process.

This approach is particularly useful in scenarios where the consequences of false positives and false negatives are not equal, and it is important to minimize the overall cost rather than just the error rate.

In cost-sensitive analysis, the objective is to find the best Model Parameters that minimize the overall cost, rather than just focusing on minimizing the error rate. This involves considering the costs associated with different types of classification errors, as defined in the cost matrix, and adjusting the model accordingly.

To achieve this, you can:

1. Optimize Model Parameters: Use techniques like grid search or random search to find the optimal hyperparameters that minimize the total cost. This involves evaluating different combinations of parameters and selecting the ones that result in the lowest cost.
2. Incorporate Cost Information: Modify the learning algorithm to directly incorporate cost information, such as using cost-sensitive decision trees or adjusting class weights in algorithms that support it.
3. Adjust Decision Thresholds: Fine-tune the decision thresholds to balance the trade-off between false positives and false negatives, aiming to minimize the total cost.

Key Concepts:

Cost Matrix: A cost matrix is used to define the costs associated with different types of classification errors. For binary classification, it typically includes:

- True Positive Cost (C(TP)): The cost when a positive instance is correctly classified.
- False Positive Cost (C(FP)): The cost when a negative instance is incorrectly classified as positive.
- True Negative Cost (C(TN)): The cost when a negative instance is correctly classified.
- False Negative Cost (C(FN)): The cost when a positive instance is incorrectly classified as negative.

Objective: The goal of cost-sensitive analysis is to minimize the total cost, which is calculated based on the cost matrix and the confusion matrix of the model's predictions.

Approaches to Cost-Sensitive Learning:

1. Cost-Sensitive Algorithms: Modify existing algorithms to incorporate cost information directly into the learning process. For example, decision trees can be adapted to consider misclassification costs when splitting nodes.
2. Reweighting Instances: Adjust the weights of instances in the training data based on their associated costs. This can be done by assigning higher weights to instances that are more costly to misclassify.
3. Threshold Adjustment: Modify the decision threshold of a classifier to account for different costs. For example, in a binary classifier, the threshold for classifying an instance as positive can be adjusted to minimize the expected cost.

Cross Entropy

Cross entropy is a Loss function used in Classification tasks, particularly for categorical data. The cross entropy loss function is particularly effective for multi-class classification problems, where the goal is to assign an input to one of several categories.

Cross entropy measures confidence.

Cross entropy works by measuring the (difference/loss) dissimilarity between two probability distributions: the true distribution (actual class labels) and the predicted distribution (model's output probabilities).

Fit of Predictions:

- A low cross entropy loss means the predicted probabilities are close to the true labels (e.g., assigning high probability to the correct class).
- A high loss indicates significant divergence, meaning the model's predictions are inaccurate or uncertain.

By minimizing cross entropy, the model learns to produce probability distributions that closely match the true class distributions, thereby improving its classification accuracy.

1. Probability Distributions: In a classification task, the model outputs a probability distribution over the possible classes for each input. For example, in a three-class problem, the model might output probabilities like [0.7, 0.2, 0.1] for classes A, B, and C, respectively.
2. True Labels: The true class label is represented as a one-hot encoded vector. If the true class is A, the vector would be [1, 0, 0].
3. Cross Entropy Calculation calculates the loss by comparing the predicted probabilities with the true labels. The formula for cross entropy loss L for a single instance is:

   $$L = -\sum_{i=1}^{N} y_i \log(p_i)$$

   where:

   - N is the number of classes.
   - y_i is the true label (1 if the class is the true class, 0 otherwise).
   - p_i is the predicted probability for class i.
4. Interpretation: The cross entropy loss increases as the predicted probability diverges from the actual label. If the model assigns a high probability to the correct class, the loss is low. Conversely, if the model assigns a low probability to the correct class, the loss is high.
5. Optimization: During training, the model's parameters are adjusted to minimize the cross entropy loss across all training examples. This process helps the model improve its predictions over time.

Where is it used

Cross entropy is widely used in classification for several reasons:

Probabilistic Modeling:

```
- It directly aligns with the goals of probabilistic classifiers, as it measures how well the predicted probability distri
```

Focus on Confidence:

```
- Encourages the model to assign higher probabilities to the correct classes, improving not just accuracy but also confide
```

Optimization Efficiency:

```
- Cross entropy is smooth and convex for logistic regression-like models, enabling efficient gradient-based optimization.
```

Multi-Class Support:

```
- Works seamlessly in multi-class scenarios where the true labels are one-hot encoded and predictions are probability dist
```

Implementation

In ML_Tools see:

- Cross_Entropy_Single.py
- Cross_Entropy.py
- Cross_Entropy_Net.py

Cross Validation

Cross-validation is a statistical technique used in machine learning to assess how well a model will generalize to an independent dataset. It is a crucial step in the model-building process because it helps ensure that the model is not overfitting or underfitting the training data.

- Cross-validation is a technique used in machine learning and statistics to evaluate the performance (Model Optimisation) of a predictive model.
- It provides a robust evaluation by splitting the training data into smaller chunks and training the model multiple times.
- K-Fold Cross-Validation: Involves dividing the dataset into (k) equal-sized subsets (called "folds") and using each fold as a validation set once, while the remaining (k-1) folds are used for training.
- The model's performance is averaged across all (k) folds to provide a more robust estimate of its generalization performance.

Common Variations

- K-Fold Cross-Validation: The most common method, where the data is split into (k) folds and the model is trained (k) times, each time using a different fold as the validation set.
- Stratified K-Fold: Ensures each fold has a similar proportion of class labels, important for imbalanced datasets.
- Repeated K-Fold: Repeats the process multiple times with different random splits for more robust results.
- Leave-One-Out Cross-Validation (LOOCV): Each data point is used once as a test set while the rest serve as the training set.

How Cross-Validation Fits into Building a Machine Learning Model

1. Model Evaluation: Used to evaluate the performance of different models or algorithms to choose the best one.
2. Hyperparameter Tuning: Provides a reliable performance metric for each set of hyperparameters.
3. Model Validation: Ensures consistent performance across different subsets of data.
4. Bias and variance tradeoff: Helps in understanding the tradeoff between bias and variance, guiding the choice of model complexity.

Advantages:

- Reduced Bias: Offers a more reliable performance estimate compared to using a single validation set.
- Efficient Data Use: All data is used for both training and validation.
- Prevents Overfitting: By evaluating on multiple folds, it can detect if the model is overfitting to the training data.

Choosing (k)

- Common values: 5 or 10
- Higher (k) leads to more accurate estimates but increases computation time.
- Consider dataset size and complexity when choosing (k).

Code Implementation

In ML_Tools see:

- KFold_Cross_Validation.py

Cross-Validation Strategy in Time Series

All notebooks use cross-validation based on `TimeSeriesSplit` to ensure proper evaluation of performance with no Data Leakage. This method ensures that training and test data are split while maintaining the chronological order of the data.

Data Pipeline To Data Products

The journey from Data Pipeline to Data Product involves transforming raw data into valuable insights or applications that can be used to drive business decisions. This process typically includes several stages, each with its own set of tasks and objectives.

Read more on Data Orchestration Trends: The Shift From Data Pipelines to Data Products.

Workflow

1. **Define Objectives**:

    - Understand the business goals and what insights or products are needed.
2. **Design the Pipeline**:
    - Plan the architecture and select appropriate tools for each stage of the pipeline.
3. **Implement and Test**:
    - Build the pipeline, ensuring data flows smoothly from ingestion to product delivery.
    - Test for accuracy, performance, and reliability.
4. **Deploy and Monitor**:
    - Deploy the pipeline in a production environment.
    - Continuously monitor for performance and make adjustments as needed.
5. **Iterate and Improve**:
    - Gather feedback and refine the pipeline and products to better meet business needs.

Example

Imagine a retail company wants to create a recommendation system for its online store:

1. **Data Ingestion**: Collect customer browsing and purchase data from the website.
2. **Data Processing**: Clean and transform the data to identify patterns in customer behavior.
3. **Data Storage**: Store the processed data in a data warehouse for easy access.
4. **Data Analysis**: Use machine learning algorithms to analyze the data and generate recommendations.
5. **Data Visualization**: Create dashboards to visualize customer trends and recommendation performance.
6. **Data Products**: Deploy the recommendation system on the website to enhance customer experience.

Dbscan

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is a Clustering algorithm that groups together data points based on density. It is particularly useful when K-means doesn't work well, such as in datasets with complex shapes or when there are outliers.

- **Used when K-means doesn't work**: DBSCAN handles datasets with irregular cluster shapes and is not sensitive to outliers like K-means.
- **When you have nesting of clusters**: It can identify clusters of varying shapes and sizes without needing to predefine the number of clusters, unlike K-means.
- **Groups core points to make clusters**: DBSCAN identifies core points, which have many nearby points, and groups them together.
- **Can identify standardised/Outliers**: It detects noise points (outliers) that don't belong to any cluster.

Python Example:

```
from sklearn.cluster import DBSCAN
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt

# Create sample data
X, _ = make_blobs(n_samples=300, centers=4, cluster_std=0.60, random_state=0)

# Apply DBSCAN
dbscan = DBSCAN(eps=0.3, min_samples=5)
clusters = dbscan.fit_predict(X)

# Plot results
plt.scatter(X[:, 0], X[:, 1], c=clusters, cmap='plasma')
plt.show()
```

This will cluster the data and visualize it, highlighting core points and marking outliers as separate clusters.

🌐 Sources

1. hex.tech - When and Why To Choose Density-Based Methods
2. newhorizons.com - DBSCAN vs. K-Means: A Guide in Python

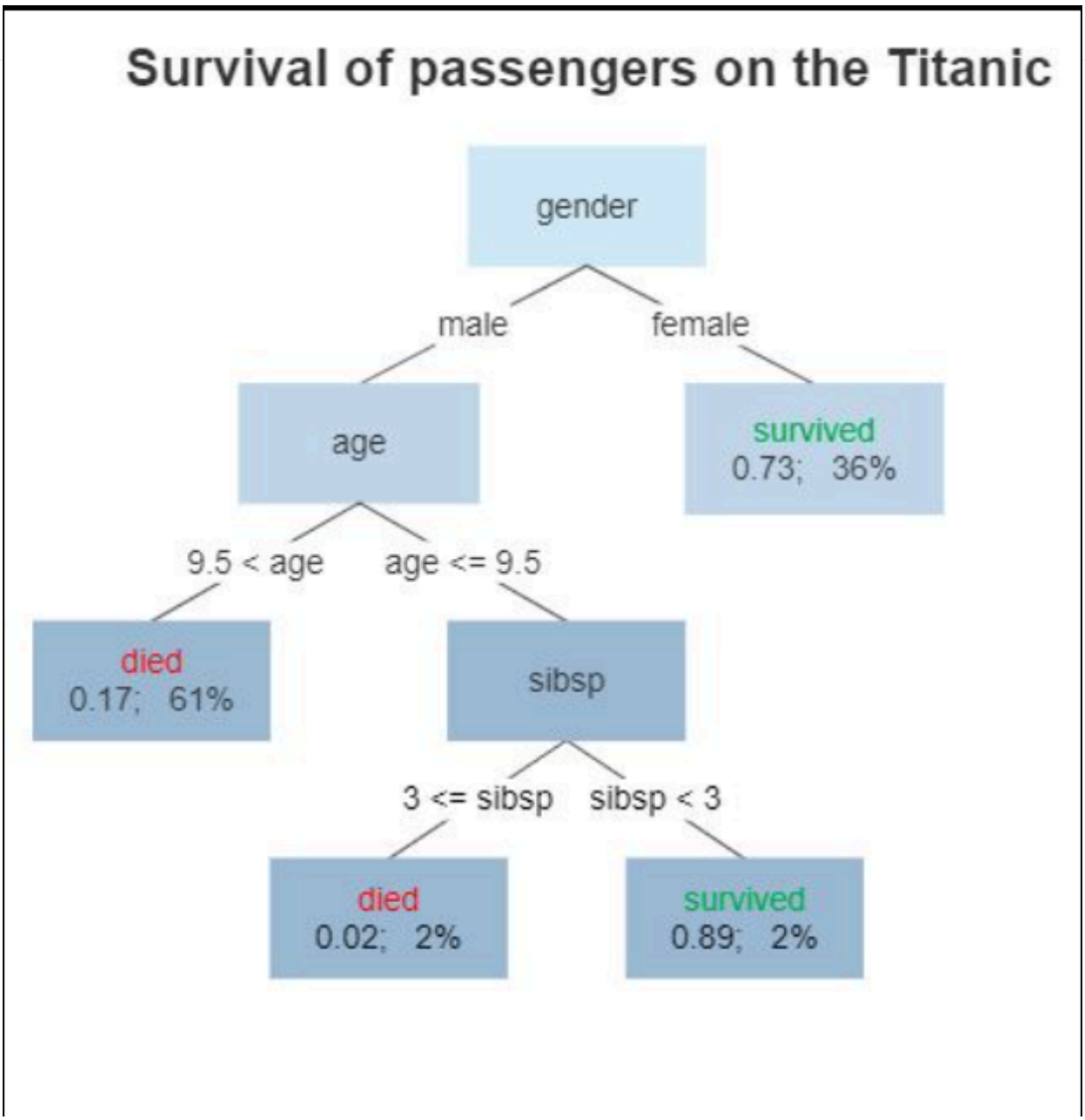
Decision Tree

A Decision Tree is a type of Supervised Learning algorithm used to predict a target variable based on input features. It involves splitting data into subsets to create a tree-like model.

Decision Tree Structure are a flowchart-like model where each internal node represents a decision based on a feature, branches represent outcomes, and leaf nodes represent final predictions.

Splits data recursively based on feature importance, forming a tree-like structure.

The decision tree algorithm calculates the Gini impurity for each possible split and selects the one with the lowest impurity. Use to make predictions on new data, the algorithm traverses the decision tree from the root node to a leaf node, following decision rules based on input features. Once it reaches a leaf node, it assigns the corresponding class label or prediction.



Key Concepts

1. Objective: Predict a target variable using input features.
2. Splitting: Identify the best feature to split the data into subsets, aiming for homogeneous groups.
3. Impurity Calculation: Use metrics like Gini Impurity or Cross Entropy (Gini Impurity vs Cross Entropy)to evaluate splits. Choose the split that minimizes impurity.
4. Purity: A node is pure if it perfectly classifies the data, requiring no further splits.
5. Leaf Node Output: Assigns the most common class label or average value in the node.
6. Overfitting: Can occur if the tree is too complex. Mitigate with pruning and limiting tree depth.
7. Cross Validation: Refine the model to better generalize to new data.

Splitting Process

The splitting process in a Decision Tree involves dividing the dataset into subsets to create a tree-like structure. This process is crucial for building an effective model that can predict target variables accurately.

Splitting Criteria:

- The algorithm evaluates various features to determine the best split at each node.
- It selects the feature and split point that minimize impurity in the resulting child nodes.
- The split that most effectively reduces impurity is chosen, ensuring that each subset is as homogeneous as possible.

Building Process

1. **Initial Splitting**:
    - Begin at the root node and select the best feature to split the data. This selection is based on impurity measures such as Gini impurity or entropy for classification tasks, and variance reduction for regression tasks.
    - The goal is to create subsets that are as homogeneous as possible with respect to the target variable.
2. **Recursive Partitioning**:
    - After the initial split, each subset becomes a child node.
    - The algorithm recursively applies the splitting process to each child node.
    - Continue splitting until stopping criteria are met, such as reaching a maximum tree depth, having a minimum number of samples per node, or achieving insufficient improvement in purity.
3. **Leaf Nodes**:
    - The process continues until reaching leaf nodes, which have no further splits.
    - At each leaf node, assign a class label (for classification) or predict a continuous value (for regression) based on the majority class or average value of the samples in that node.

Refinement

Pruning:

- Pre-pruning: Stop tree growth early based on criteria like maximum depth or minimum impurity improvement.
- Post-pruning: Allow the tree to grow fully, then prune back based on performance metrics.

Hyperparameter

Can use GridSeachCv to pick the best paramaters.

Parameter	Purpose	Effect	Example
`criterion`	Splitting criteria	Impacts decision logic.	`criterion='gini'` or `criterion='entropy'`
`max_depth`	Maximum tree depth	Prevents overfitting.	`max_depth=5` limits the tree depth to 5.
`min_samples_split`	Min samples to split a node	Limits tree growth.	`min_samples_split=10` requires at least 10 samples to split a node.
`min_samples_leaf`	Min samples at leaf node	Reduces overfitting.	`min_samples_leaf=5` ensures every leaf has at least 5 samples.
`max_features`	Features considered for splitting	Adds randomness.	`max_features='sqrt'` or `max_features=3`.
`max_leaf_nodes`	Max leaf nodes allowed	Reduces overfitting.	`max_leaf_nodes=20` caps the tree at 20 leaves.
`class_weight`	Adjusts for imbalanced data	Improves fairness.	`class_weight='balanced'` or `class_weight={0:1, 1:2}`.
`ccp_alpha`	Pruning parameter	Simplifies tree.	`ccp_alpha=0.01` prunes weak splits based on complexity.

Advantages and Disadvantages of Decision Trees

Advantages:

- Simple and interpretability|interpretable model.
- Minimal data preparation required.
- Transparent decision-making process.

Disadvantages:

- Prone to overfitting, especially with complex datasets.
- Sensitive to small changes in data.
- Can become complex with many features.

Decision Tree

Deep Learning Frameworks

Watch Overview Video

TensorFlow

Focus: TensorFlow is a comprehensive open-source platform for machine learning. It provides a flexible and comprehensive ecosystem of tools, libraries, and community resources that lets researchers push the state-of-the-art in ML, and developers easily build and deploy ML-powered applications.

Integration: TensorFlow can implement a wide range of machine learning algorithms, including those available in Sci-kit Learn, making it versatile for various applications.

Modularity: Its modular architecture allows users to deploy computation across a variety of platforms (CPUs, GPUs, TPUs), and from desktops to clusters of servers to mobile and edge devices.

Parallelization: TensorFlow is optimized for high-performance numerical computation, making it suitable for large-scale machine learning tasks that require parallel processing.

Use Cases: TensorFlow is widely used in both academic research and industry for tasks such as image and speech recognition, natural language processing, and more.

Sci-kit Learn

Focus: Sci-kit Learn is a simple and efficient tool for data mining and data analysis, built on NumPy, SciPy, and matplotlib. It is primarily used for traditional machine learning techniques such as classification, regression, clustering, and dimensionality reduction.

Limitations: While excellent for classical machine learning tasks, Sci-kit Learn is not designed for deep learning or neural network architectures, which require more specialized frameworks like TensorFlow or PyTorch.

Keras

API Level: Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano. It allows for easy and fast prototyping through user-friendly, modular, and extensible code.

Integration: Keras is tightly integrated with TensorFlow 2.0, providing a simplified interface for building and training deep learning models.

Purpose: Designed to enable fast experimentation, Keras is ideal for beginners and researchers who need to quickly prototype and test new ideas.

Performance: While Keras simplifies model building, it may not be as performant as lower-level frameworks like TensorFlow when it comes to fine-tuning and optimizing models for production.

Deep Learning

> [!Summary] Deep learning is a subset of machine learning that uses neural networks to process large-scale data for tasks like image and speech recognition, natural language processing, and recommendation systems.
>
> A neural network consists of layers of nodes where each node performs weighted sums of its inputs, applies activation functions like ReLU or sigmoid, and produces an output.
>
> Backpropagation is the primary algorithm for training neural networks by minimizing error through Gradient Descent. Regularization techniques, such as dropout, prevent overfitting.
>
> Popular frameworks like PyTorch and TensorFlow facilitate deep learning model development.

Questions:

- What is the role of gradient-based optimization in training deep learning models.
- Explain different gradient descent algorithms, their advantages, and limitations.

Areas of Deep Learning:

- LLM
- Neural network|Neural Network

> [!Follow up questions]
>
> - How does the choice of activation function affect the performance of deep learning models across different tasks?
> - What are the trade-offs between different gradient descent algorithms (e.g., Stochastic Gradient Descent|SGD vs. Adam) in training neural networks? See Optimisation techniques.
>
> [!Related Topics]
>
> - Transfer Learning: Applying pre-trained models to new tasks.

Dimensionality Reduction

Dimensionality reduction is a step in the Preprocessing phase of machine learning that helps simplify models, enhance interpretability, and improve computational efficiency.

Its a technique used to reduce the number of input variables (features) in a dataset while retaining as much information as possible. This process is essential for several reasons:

1. **Improves Model Performance**: Reducing the number of features can help improve the performance of machine learning models by minimizing overfitting and reducing noise.
2. **Enhances Visualization**: It allows for easier Data Visualisation of high-dimensional data by projecting it into lower dimensions (e.g., 2D or 3D).
3. **Reduces Computational Cost**: Fewer features mean less computational power and time required for training models.

Common Techniques

- **Principal Component Analysis (Principal Component Analysis)**: A statistical method that transforms the data into a new coordinate system, where the greatest variance by any projection lies on the first coordinate (principal component/orthogonal components), the second greatest variance on the second coordinate, and so on.
- **t-Distributed Stochastic Neighbor Embedding (t-SNE)**: A technique particularly well-suited for visualizing high-dimensional data by reducing it to two or three dimensions while preserving the local structure of the data. t-SNE is a non-linear technique used for visualization and dimensionality reduction by preserving pairwise similarities between data points, making it suitable for exploring high-dimensional data.
- Linear Discriminant Analysis method used for both classification and dimensionality reduction, which finds a linear combination of features that best separates two or more classes.

Explain the curse of dimensionality

Dropout

Dropout is a Regularisation technique used in Neural network training to prevent overfitting. It works by randomly dropping units (neurons) during training, which helps the network to not rely too heavily on any single neuron.

Purpose

- The main goal of dropout is to improve the generalization of the model by reducing over-reliance on specific neurons. This encourages the network to learn more robust features that are useful in different contexts.

How It Works

- During each training iteration, a subset of neurons is randomly selected and ignored (dropped out). This means their contribution to the activation of downstream neurons is temporarily removed on the forward pass, and any weight updates are not applied to the neuron on the backward pass.

Implementation

```
from tensorflow.keras.layers import Dropout
# Add a dropout layer with a dropout rate of 0.5
# The dropout rate (e.g., 0.5) specifies the fraction of neurons to drop during training. A rate of 0.5 means that half of
Dropout(0.5)
```

Embeddings For Oov Words

Can you find words in a Vector Embedding|word embedding that where not used to creates the embedding?

Yes, but with important caveats. If a word is not in the spaCy model's vocabulary with a vector, then:

✅ What you can do

Option 1: Filter out words without vectors (what you're doing now)

This is the cleanest option:

```
if token.has_vector:
    embeddings.append(token.vector)
    valid_words.append(word)
```

Option 2: Fallback to character-level embeddings (optional)

If you're using `en_core_web_lg`, spaCy sometimes provides approximate vectors for out-of-vocabulary (OOV) words using subword features. But with `en_core_web_md`, OOV words truly lack vector meaning.

Option 3: Use a different embedding model

Use FastText or transformer-based models (e.g., Sentence Transformers), which can produce embeddings for OOV words based on subword information or context.

Example with FastText (using gensim):

```
from gensim.models import KeyedVectors

model = KeyedVectors.load_word2vec_format("cc.en.300.vec")  # or download from FastText
embedding = model.get_vector("unseenword")  # FastText will synthesize it
```

💡 Summary

Approach	Handles OOV?	Notes
spaCy `en_core_web_md`	❌	Skips words without vectors (recommended)
spaCy `en_core_web_lg`	⚠️ Sometimes	May infer vectors using subword info
FastText / GloVe	✅	Good for unseen words
Sentence Transformers (BERT)	✅	Contextualized, ideal for phrases/sentences

NLP #ml_process #ml_optimisation

Encoding Categorical Variables

Overview

Categorical variables need to be converted into numerical representations to be used in models, particularly in Regression analysis. This process is essential for transforming categorical results into a format that algorithms can interpret.

Label Encoding

This method assigns a unique integer to each category in the variable.

```
from sklearn.preprocessing import LabelEncoder


label_encoder = LabelEncoder()
var1_cat = df['var1']  # Replace df with your DataFrame
var1_encoded = label_encoder.fit_transform(var1_cat)
```

For example, if `df[col]` contains the categories `['apple', 'banana', 'orange']`, the `LabelEncoder` would transform them into `[0, 1, 2]`.

However, keep in mind that this encoding can imply an order or hierarchy in the data, which might not be intended. In some cases, you might want to use `OneHotEncoder` instead, which creates a binary vector for each category.{}

Given a term in the df you can transform it without needing to look up its value.

```
company="google"
company_n = LabelEncoder().transform([company])
```

One-Hot Encoding

This technique creates a binary column for each category, allowing the model to treat each category as a separate feature.

```
from sklearn.preprocessing import OneHotEncoder


binary_encoder = OneHotEncoder(categories='auto')
var1_1hot = binary_encoder.fit_transform(var1_encoded.reshape(-1, 1))
var1_1hot_mat = var1_1hot.toarray()
var1_DF = pd.DataFrame(var1_1hot_mat, columns=['cat1', 'cat2', 'cat3'])  # Adjust column names as needed
var1_DF.head()
```

Understanding OneHotEncoder:

The `OneHotEncoder` from `sklearn.preprocessing` is used to convert categorical integer values into a format that can be provided to machine learning algorithms to do a better job in prediction. It creates a binary column for each category and returns a sparse matrix or dense array.

Converting All Categorical Variables to Dummies

To convert all categorical variables in a DataFrame to dummy variables, you can use the following loop:

```
for col in df.columns:
    if df[col].dtype == 'object':
        dummies = pd.get_dummies(df[col], drop_first=False)
        dummies = dummies.add_prefix(f'{col}_')
        df.drop(col, axis=1, inplace=True)
        df = df.join(dummies)
```

Dummy Variable Trap: When using one-hot encoding, it's important to avoid the **dummy variable trap**, which occurs when one category can be perfectly predicted from the others. To prevent this, you can drop one of the dummy variables, as one column is sufficient to represent a binary choice (0 or 1).

Alternative Encoding Method

Another way to encode categorical variables is by mapping them directly to integers:

```
dataset['var1'] = dataset['var1'].map({'A': 0, 'B': 1, 'C': 2}).astype(int)
```

Related Topics

- **Regression**: Understanding how regression models utilize encoded variables.
- **Feature Engineering**: Techniques to enhance model performance through better feature representation.

Overview

- Categorical variables need to be converted into numerical representations for use in models. This is essential for transforming categorical data into a format that algorithms can interpret.

Methods

- Label Encoding: Assigns a unique integer to each category.
- One-Hot Encoding: Creates a binary column for each category, allowing the model to treat each category as a separate feature.

Example Code

```
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
import pandas as pd


# Label Encoding
label_encoder = LabelEncoder()
var1_encoded = label_encoder.fit_transform(df['var1'])


# One-Hot Encoding
binary_encoder = OneHotEncoder(categories='auto')
var1_1hot = binary_encoder.fit_transform(var1_encoded.reshape(-1, 1))
var1_1hot_mat = var1_1hot.toarray()
var1_DF = pd.DataFrame(var1_1hot_mat, columns=['cat1', 'cat2', 'cat3'])
```

Evaluating Language Models

The LMSYS Chatbot Arena is a platform where various large language models (LLMs), including versions of GPT and other prominent models like LLaMA or Claude, are compared through side-by-side interactions. The performance of these models is evaluated using human feedback based on the quality of their generated responses.

The arena employs several techniques to rank and compare models:

1. **Elo Rating System**: Adapted from chess, this system rates models based on their relative performance in head-to-head competitions. When one model's response is preferred over another's, the winning model gains points while the losing model loses points. The rating difference helps determine the strength of models in future predictions. The system adjusts ratings gradually to avoid bias towards more recent results, ensuring stability over time.
2. Bradley-Terry Model: This model goes beyond simple win-loss records by taking into account the difficulty of the task and the models' relative strengths. It helps fine-tune the ranking, especially when one model consistently performs better against tougher tasks.

In addition to these ranking systems, users can directly compare LLMs by giving them Prompting to handle, such as writing articles, answering questions, or performing translations. Human voters then decide which model's output is better, or they can declare a tie if neither response is satisfactory.

These methods ensure continuous improvement of the rankings, providing a transparent and evolving leaderboard of the best generative models, including GPT versions

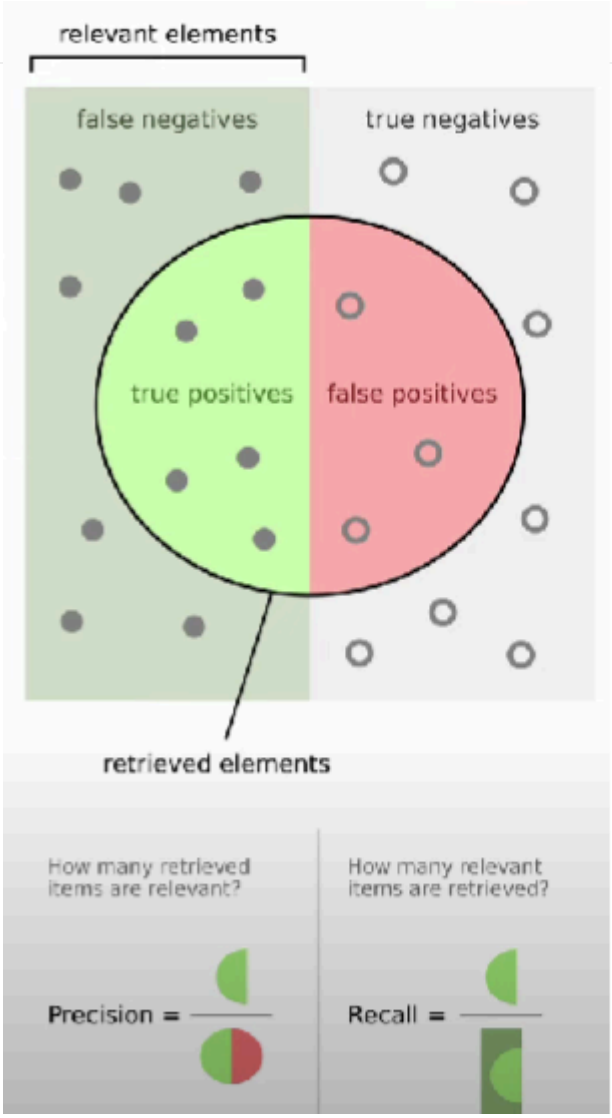
https://openlm.ai/chatbot-arena/

https://www.analyticsvidhya.com/blog/2024/05/from-gpt-4-to-llama-3-lmsys-chatbot-arena-ranks-top-llms/

Evaluation Metrics

Description

Confusion Matrix Accuracy Precision Recall Precision or Recall F1 Score Recall Specificity



Resources:

Link to good website describing these

In ML_Tools see: Evaluation_Metrics.py

Types of predictions

Types of predictions in evaluating models. Also see Why Type 1 and Type 2 matter

True Positive (TP):

- This occurs when the model correctly predicts the positive class. For example, if the model predicts that an email is spam and it actually is spam, that's a true positive.

False Positive (FP):

- Also known as a "Type I error," this occurs when the model incorrectly predicts the positive class. For example, if the model predicts that an email is spam but it is not, that's a false positive.

True Negative (TN):

- This occurs when the model correctly predicts the negative class. For example, if the model predicts that an email is not spam and it actually is not spam, that's a true negative.

False Negative (FN):

- Also known as a "Type II error," this occurs when the model incorrectly predicts the negative class. For example, if the model predicts that an email is not spam but it actually is spam, that's a false negative.

Evaluation metrics in practice

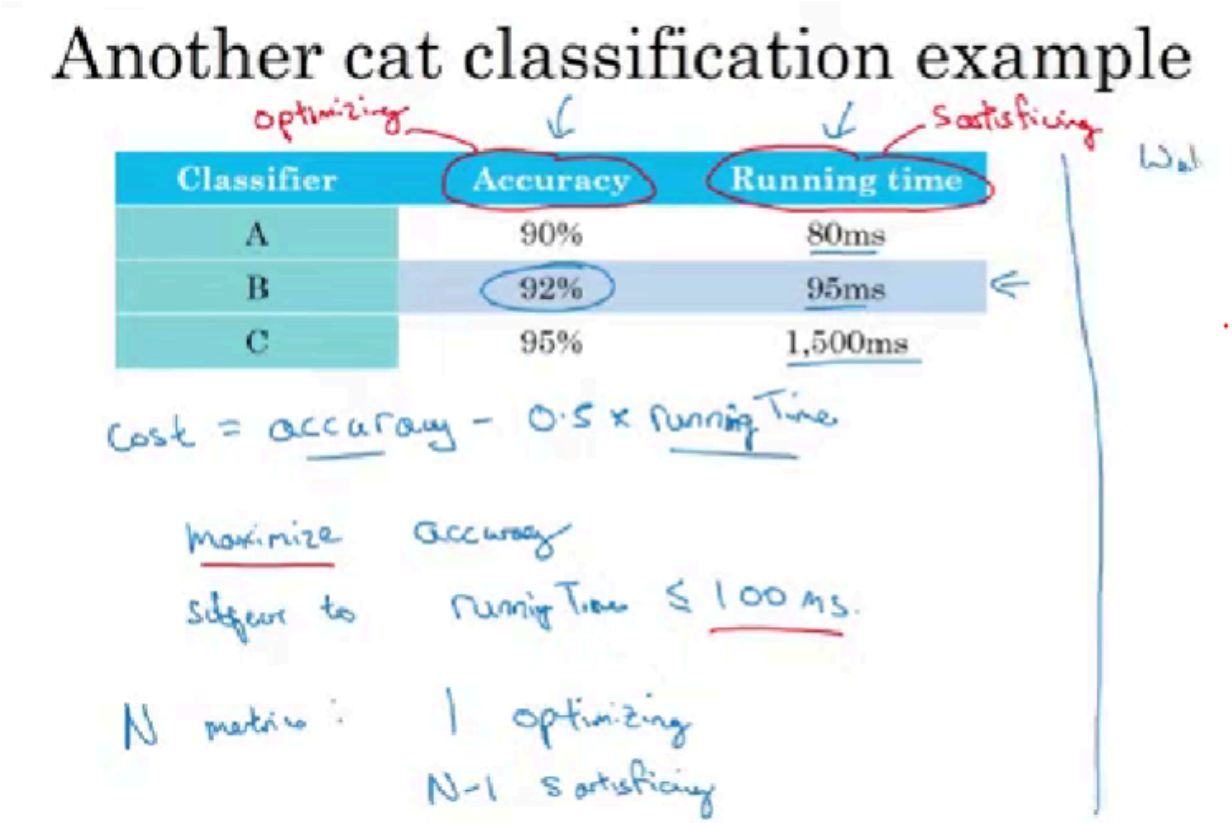
Having many evaluation metrics is hard to understand and optimise. Sometimes it is best to combine into one.

Use a single number i.e. accuracy or F1 Score .

This speeds up development of ml projects.

In order to use metrics to evaluate a model we can:

- Can combine multiple metrics a formula, i.e. weighted average.
- If there is a metrics we are happy that the model passes a given level then we can have it "Satisfying". So the for the given metric it just needs to pass a given level.
- For metrics we are interested in we have it "Optimising", the one we want to be the best.
- Setup: Pick N-1 satisfying and 1 optimising.



Faiss

FAISS (Facebook AI Similarity Search) is a library developed by Facebook AI Research that enables efficient similarity search and clustering of dense vectors. It is especially well-suited for applications involving high-dimensional vector data, such as NLP

Related terms:

- Vector Embedding

Overview

FAISS is optimized for:

- **Fast retrieval** from large collections of vectors (millions or more).
- **Approximate nearest neighbor (ANN)** search, which trades off accuracy for speed.
- **Exact search**, depending on the chosen index type.
- **GPU acceleration** for very large-scale search tasks.

Core Concept

At its core, FAISS takes a large number of **high-dimensional vectors** (e.g., sentence or document embeddings), and enables fast **similarity search** to retrieve the most similar vectors to a given Querying|query.

For example, in an NLP Memory|context:

- Documents or notes are embedded into vector space using a model like SBERT.
- These embeddings are stored in a FAISS index.
- Given a query, its embedding is computed, and FAISS returns the nearest neighbors (i.e., most semantically similar notes).

Index Types

FAISS offers different types of indices depending on use case:

- `IndexFlatL2` : exact search using L2 (Euclidean) distance.
- `IndexIVFFlat` : approximate search using inverted files.
- `IndexHNSW` : Hierarchical Navigable Small World graph-based index (good for high recall).
- `IndexPQ` : product quantization for memory-efficient indexing.

Feature Engineering

Its the term given to the iterative process of building good features for a better model. Its the process that makes relevant features (using formulas and relations between others).

We use it when we have a refined and optimised model.

What does it involve

- Create new features from existing ones (e.g., ratios, interactions).
- Transform features to better capture non-linear relationships.
- Dimensionality Reduction if necessary.

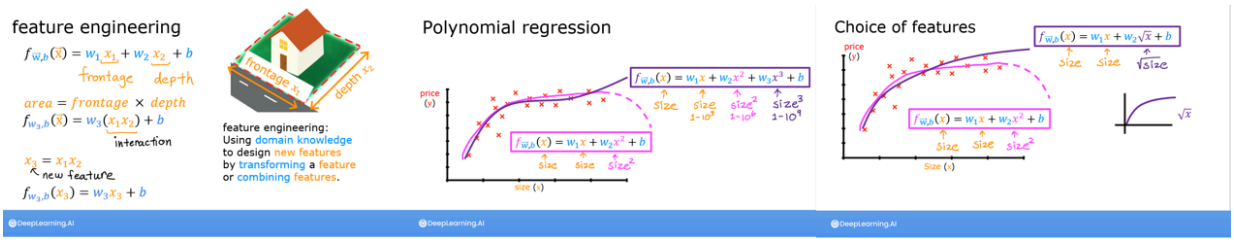
The main techniques of feature engineering:

- are selection (picking subset),
- learning (picking the best),
- extraction and combination(combining).

Example: Predicting house prices. Raw features might be square footage, number of bedrooms, and location. Feature engineering could involve: Combining square footage and bedrooms into a "living space" feature.

Example:

- Decompose datetime information into separate features for date and time to capture their respective predictive powers.



Feature Importance

Feature importance refers to techniques that assign scores to input features (predictors) in a machine learning model to indicate their relative impact on the model's predictions.

Feature importance is typically assessed after Model Building. It involves analyzing the trained model to determine the impact of each feature on the predictions.

Feature importance helps in:

- improving model interpretability,
- identifying key predictors,
- and possibly performing Feature Selection to reduce dimensionality, and refining performance

The outcome is a ranking or scoring of features based on their importance.

By understanding which features contribute the most to the predictions, you can focus on the most relevant information in your data and potentially reduce model complexity without sacrificing performance.

Types of Feature Importance Methods

1. Model-Specific Methods:
    - Tree-based models: Models like Random Forests, Gradient Boosted Trees, and Decision Trees have built-in mechanisms for calculating feature importance. They do so based on the decrease in impurity (e.g., Gini Impurity in classification tasks or variance in regression tasks) or based on the reduction in error when the feature is used for splitting.
    - Linear models: In models like linear regression or logistic regression, feature importance can be derived from the absolute values of the model coefficients, assuming features are standardized.
2. Model-Agnostic Methods:
    - Permutation importance: This method measures the importance of a feature by randomly shuffling its values and observing the impact on the model's performance. The larger the decrease in performance, the more important the feature is.
    - SHapley Additive exPlanations

- Local Interpretable Model-agnostic Explanations

Code snippets for conducting Feature Importance

SHapley Additive exPlanations

Local Interpretable Model-agnostic Explanations

Tree-based algorithms like Random Forests or XGBoost automatically calculate feature importance.

In Python, for example, after training a Random Forest model, you can access the feature importance scores using:

```
from sklearn.ensemble import RandomForestClassifier


# Train a RandomForest model
model = RandomForestClassifier()
model.fit(X_train, y_train)


# Get feature importance scores
importances = model.feature_importances_
```

This method uses the decrease in node impurity as a measure of feature importance.

Feature Selection

Purpose: The primary goal of feature selection is to identify and retain the most relevant features for model training while removing irrelevant or redundant ones. This helps in simplifying the model, reducing overfitting, and improving computational efficiency.

Process: Feature selection is typically performed before model training. It involves evaluating features based on certain criteria or algorithms to decide which features to keep or discard.

Through an iterative process, feature selection experiments with different methods, adjusts parameters, and evaluates model performance until an optimal set of features is found.

See Feature Selection vs Feature Importance

Methods

The choice of feature selection method depends on factors like the size of your dataset, the number of features, and the complexity of your model. It's often a balance between computational cost and performance improvement.

- Filter Methods: Select features based on statistical properties, independent of any machine learning algorithm. (Separate stage to training)
- Wrapper Methods: Involve training multiple models with different subsets of features and selecting the subset that yields the best performance. (Separate stage to training)
- Embedded Methods: Perform feature selection as part of the model training process. (Part of training)

After selecting features, it's essential to evaluate your model's performance (Model Evaluation) with the chosen subset. Sometimes, feature selection can inadvertently remove important information.

Detecting Noisy or Redundant Features

- Correlation Analysis: Use a Heatmap or Clustering. Features with low correlation to the target or high correlation with other features may be candidates for removal.
- Dimensionality Reduction Techniques: Techniques like Principal Component Analysis or Singular Value Decomposition (SVD) can transform the features into a lower-dimensional space while preserving as much variance as possible. Features with low contribution to the principal components can be considered for removal.
- Visualizations: Plotting pairwise scatter plots or Heatmap of feature Correlation can provide visual insights into redundant features. Clusters of highly correlated features or scatter plots showing no discernible pattern with the target variable can indicate noisy or redundant features.

Investigating Features

- Variance Thresholding: Check the Variance & Distributions of each feature. Features with very low variance (close to zero) contribute little information and may be considered noisy. Removing such features can help simplify the model without sacrificing much predictive power.
- Univariate Feature Selection: Use statistical tests like chi-square for categorical variables or ANOVA for numerical variables to assess the relationship between each feature and the target variable. Features with low test scores or high p-values may be less relevant and can be pruned.

Feed Forward Neural Network

A **Feedforward Neural Network (FFNN)** is the simplest type of Neural network. In this model, connections between neurons do not form a cycle, allowing data to flow in one direction—from the input layer, through the hidden layers, to the output layer—without any loops or backward connections. This straightforward design is primarily used for supervised learning tasks.

Structure

- Information flows in one direction: input → hidden layers → output.
- During **forward propagation**, input data is passed through the network, processed by each layer, and an output is produced.
- Unlike recurrent neural networks (RNNs), FFNNs do not share information or weights between layers, meaning the model does not maintain memory of past inputs.

Learning

- FFNNs learn by adjusting weights and biases during training to minimize the Loss function.

Limitations

- **Shallow vs. Deep Networks:** Simple feedforward networks with only a few hidden layers (shallow networks) may struggle to learn complex, hierarchical representations of data. Deeper networks (deep feedforward networks) with many layers can model more complex patterns but require more data and computational resources to train.
- **Overfitting:** FFNNs can overfit on the training data, especially if they have many parameters and not enough regularization (e.g., dropout, Ridge|L2 regularization).

- **No Temporal Understanding:** Unlike Recurrent Neural Networks or transformers, FFNNs cannot model sequential dependencies in data. They are better suited for static, non-sequential tasks.

Forward Propagation

[!Summary]

Forward propagation is the process by which input data moves through a neural network, layer by layer, to produce an output. During this process, each layer's weights and biases are applied to the input data, and an activation function is used to transform the data at each layer.

Mathematically, for each layer, the input x is transformed into an output y through the equation $y = f(Wx + b)$, where W represents the weights, b is the bias, and f is the activation function (e.g., ReLU, sigmoid). The output from one layer becomes the input to the next, and this continues until the final layer produces the predicted output.

This process does not involve learning; it only computes the prediction based on current weights.

[!Breakdown]

Key Components:

- Input data: Initial values fed into the network.
- Weights (W) and biases (b): Parameters adjusted during training.
- Activation function: Non-linear transformation, e.g., ReLU or sigmoid.
- Output: Prediction made by the network.

[!important]

- Forward propagation calculates predictions by applying current model parameters to inputs.
- It is the first step before backpropagation, where the error is used to adjust weights.

[!attention]

- Forward propagation does not involve learning or updating weights.
- The accuracy of forward propagation depends entirely on the current values of weights and biases.

[!Example]

In a simple neural network with one hidden layer, forward propagation can be described as:

$$z_1 = W_1 x + b_1$$

$$a_1 = \text{ReLU}(z_1)$$

$$z_2 = W_2 a_1 + b_2$$

$$y = \text{sigmoid}(z_2)$$

Here, x is the input, and y is the output prediction.

[!Follow up questions]

- How does the choice of activation function impact the forward propagation process?
- In deep networks, how can vanishing and exploding gradients problem during forward propagation affect training?

[!Related Topics]

- Backpropagation in neural networks
- Activation functions in deep learning

Gaussian Mixture Models

Gaussian Mixture Models (GMMs) represent data as a mixture of multiple Gaussian distributions, with each cluster corresponding to a different Gaussian component. GMMs are more effective than K-means because they consider the distributions of the data rather than relying solely on distance metrics.

Soft Clustering technique.

In ML_Tools see: Gaussian_Mixture_Model_Implementation.py

Kmeans vs GMM

GMMs can have difference Covariance Structures

Key Concepts

- **Gaussian Components**: Each Gaussian distribution is characterized by its mean and Covariance.
- **Likelihood**: The likelihood of a data point belonging to a cluster is given by the formula:

  $$P(X|C_k) = \pi_k \cdot \mathcal{N}(X|\mu_k, \Sigma_k)$$

  where $P(X|C_k)$ is the probability of data point X given cluster C_k, π_k is the prior probability of cluster C_k, and $\mathcal{N}$ is the Gaussian distribution.
- **Expectation-Maximization (EM) Algorithm**: GMMs utilize the EM algorithm to iteratively optimize the parameters of the Gaussian components.

Advantages of GMMs

- **Complex Data Distributions**: GMMs can capture complex data distributions, unlike K-means, which only considers distance metrics.
- **Probabilistic Framework**: GMMs provide a probabilistic framework for clustering, allowing for soft assignments of data points to clusters.
- **Modeling Elliptical Clusters**: The use of covariance matrices enables GMMs to model elliptical clusters, enhancing clustering performance.

Applications

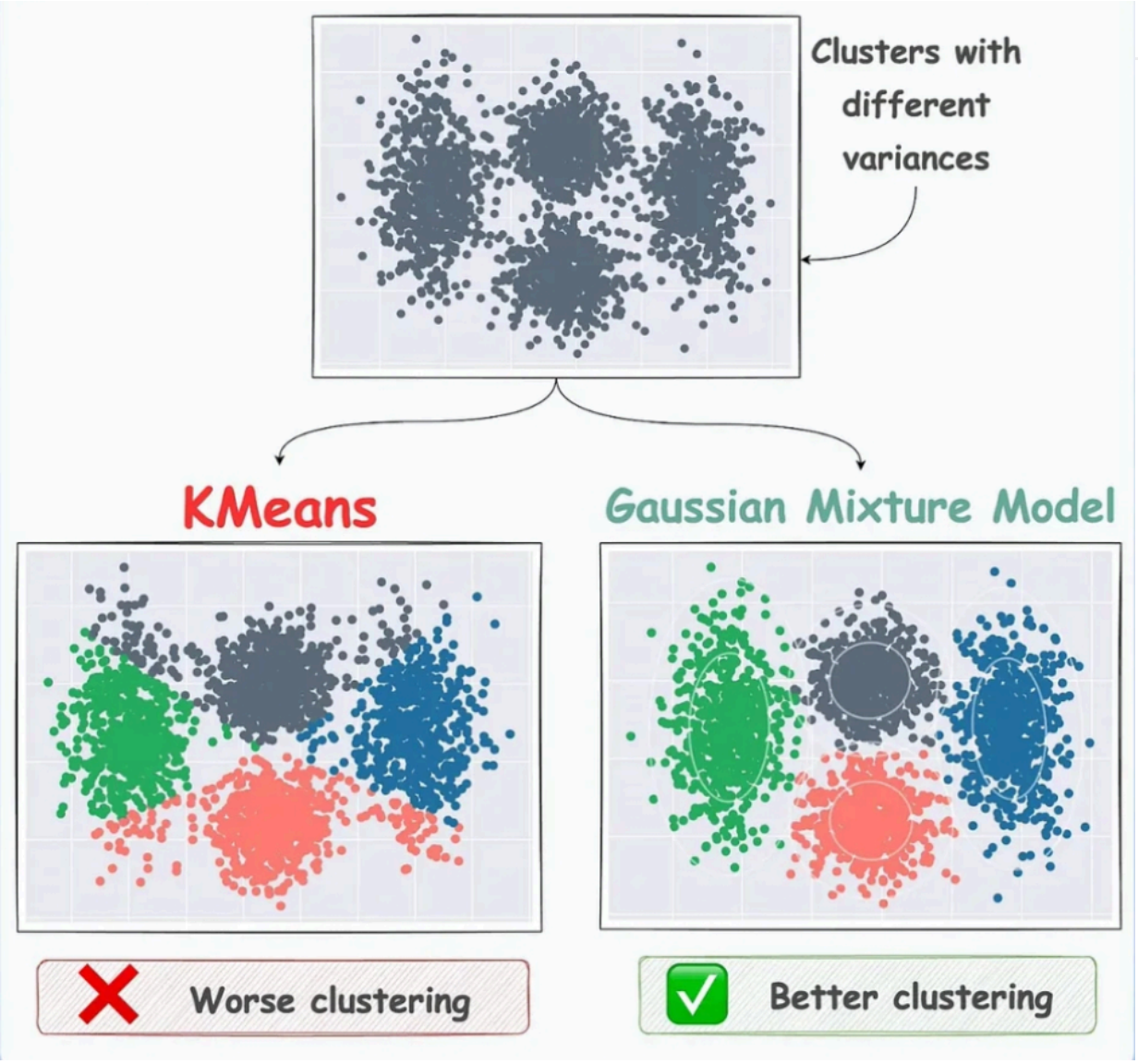
- **Anomaly Detection**: GMMs are widely used in various applications, including anomaly detection.

Important Considerations

- **Covariance Types**: The choice of covariance types (full, tied, diagonal, spherical) can significantly impact the performance of GMMs.

Follow-up Questions

- How do GMMs compare to other clustering algorithms in terms of scalability and computational efficiency?
- What are the implications of choosing different covariance types in GMMs?



Gradient Boosting Regressor

https://scikit-learn.org/1.5/modules/generated/sklearn.ensemble.GradientBoostingRegressor.html#sklearn.ensemble.GradientBoostingRegressor

Boosting

The `GradientBoostingRegressor` from the `sklearn.ensemble` module is a model used for regression tasks. It builds an Model Ensemble of Decision Tree in a sequential manner, where each tree tries to correct the errors made by the previous ones. Here's a breakdown of the key parameters:

1. **loss**: Specifies the loss function to optimize. Default is `'squared_error'`, which is the least-squares loss function. Other options like `'absolute_error'` can be used for robustness against outliers.
2. **learning_rate**: Controls the contribution of each tree to the final prediction. A smaller value (e.g., 0.01) makes the model learn more slowly, but it can lead to better generalization. Default is 0.1.
3. **n_estimators**: The number of boosting stages (i.e., trees). More trees can improve performance but also increase the risk of overfitting. Default is 100.

4. **subsample**: The fraction of samples to be used for fitting each tree. Setting this to a value less than 1.0 can help reduce overfitting, at the cost of a slight increase in bias. Default is 1.0 (use all samples).
5. **criterion**: The function used to measure the quality of a split. `'friedman_mse'` is the default, which is an improved version of mean squared error for decision trees. Other options include `'mse'` and `'mae'`.
6. **max_depth**: The maximum depth of the individual trees. This parameter controls the complexity of each tree. Default is 3, which typically works well for most tasks.
7. **min_samples_split**: The minimum number of samples required to split an internal node. Default is 2, meaning any node can be split as long as there are at least 2 samples.
8. **min_samples_leaf**: The minimum number of samples required to be at a leaf node. This helps control overfitting by requiring more data points at each leaf. Default is 1.
9. **alpha**: The quantile used for the loss function in cases of robust regression. This is useful when dealing with data that includes outliers. Default is 0.9.
10. **validation_fraction**: The fraction of training data to set aside for validation to monitor performance during training. Default is 0.1.
11. **n_iter_no_change**: The number of iterations with no improvement on the validation score to wait before stopping the training early. Default is `None`, meaning no early stopping.
12. **ccp_alpha**: Complexity parameter used for pruning the trees. A larger value leads to more pruning (simplifying the model), which can help prevent overfitting.

Gradient Boosting

Gradient Boosting is a technique used for building predictive models Model Building, particularly in tasks like regression and classification. It combines the concepts of Boosting and Gradient Descent to create strong models by sequentially combining multiple Weak Learners (Decision Tree.

Key Idea: Instead of fitting a single strong model, Gradient Boosting builds multiple weak learners sequentially. Each new model focuses on correcting the mistakes made by the previous ones by fitting to the residuals (differences between observed and predicted values).

Gradient Boosting builds an ensemble of Weak Learners (usually Decision Tree) sequentially. Each new model focuses on the errors of the previous ones, aiming to minimize the residual errors.

Final Prediction: The final prediction is made by aggregating the predictions of all the weak models, usually through a weighted sum.

High Performance: Known for its high performance and efficiency in terms of speed and memory usage.

Watch Video Explanation

Model Ensemble

Key Components

- Weak Learners: Typically decision trees used in the ensemble.
- Loss Function: Measures how well the model fits the data.
- learning rate: Controls the contribution of each weak learner to the final model.

Examples

- LightGBM
- XGBoost
- CatBoost

Benefits

- Predictive Accuracy: Often outperforms other Machine Learning Algorithms.
- Feature Handling: Effectively manages heterogeneous features and automatically selects relevant ones.
- Overfitting: Less prone to overfitting compared to other complex models.

Gradient Descent

Gradient descent is an Optimisation function used to minimize errors in a model by adjusting its parameters iteratively. It works by moving in the direction of the steepest decrease of the Loss function.

Uses the difference quotient.

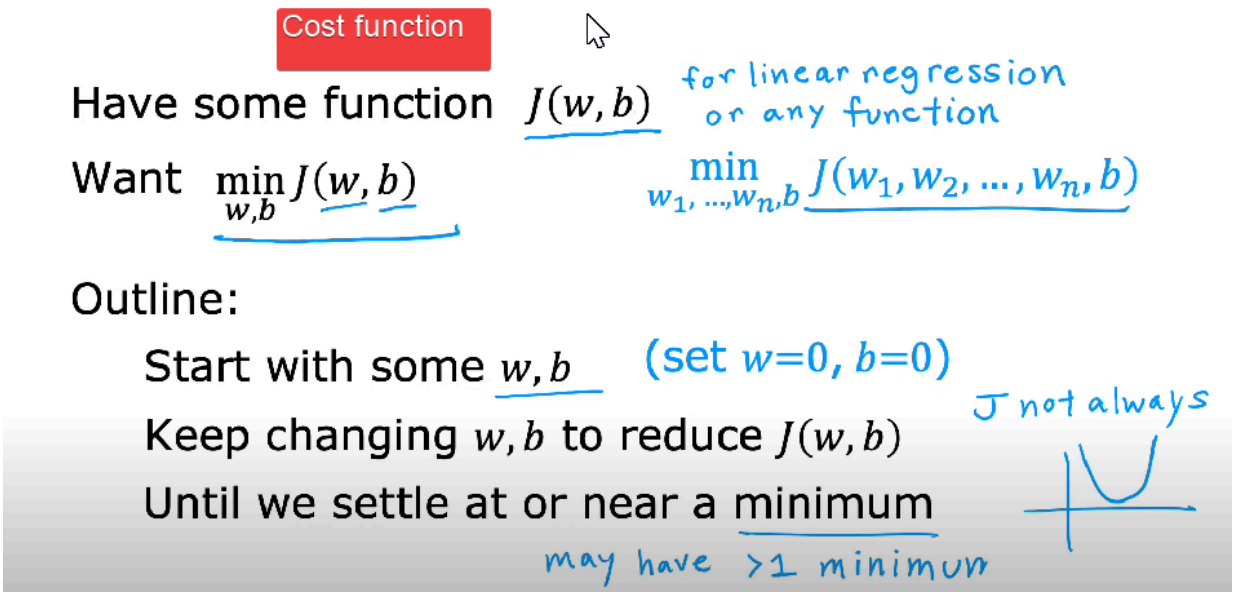
The step size is important between derivatives (small then slow) (if large then might miss minimum).

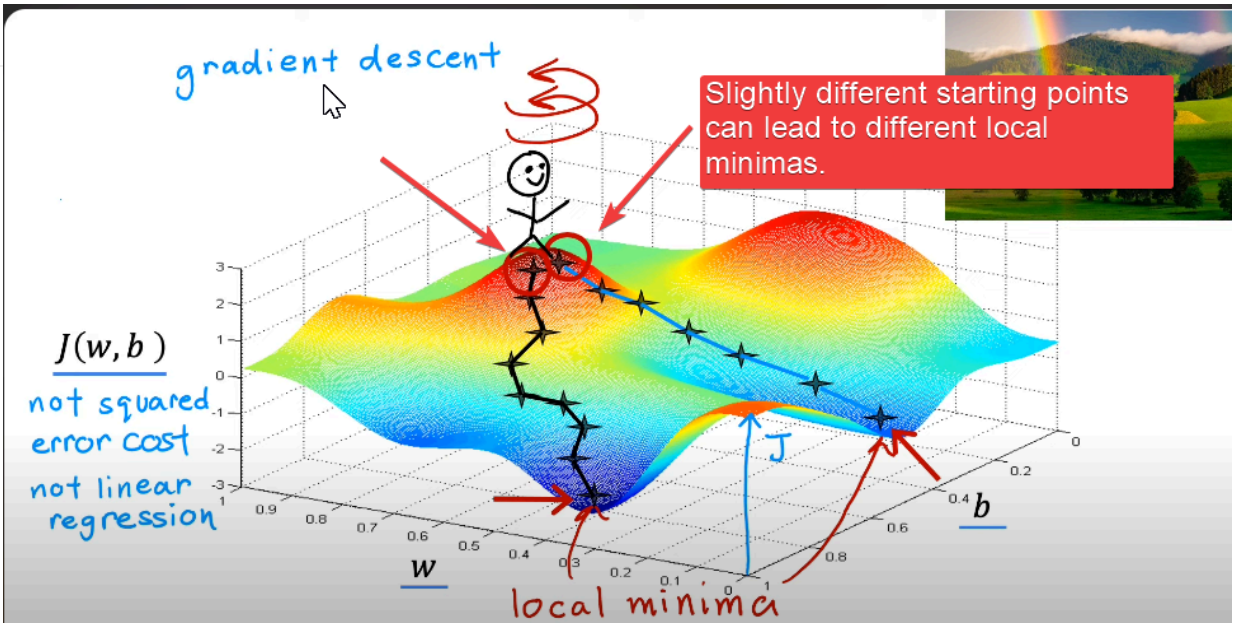
With Stochastic method we can don't need to the entire data set again, we can just add the new information to get improvement.

Gradient descent uses the entire data set.

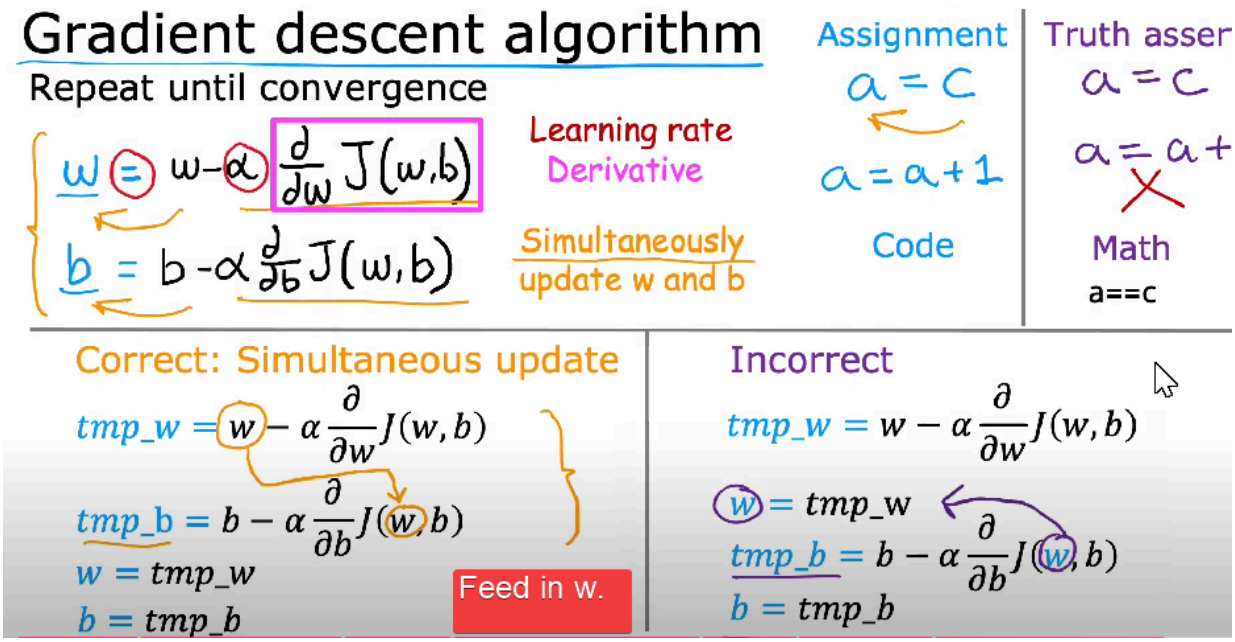
Used to find the min/max of Cost Function.

Given any point on the cost function surfaces. Then ask, "In what direction should I go to make the biggest change downhill or up hill, i.e. gradient descent"





How do you implement Gradient descent? You update the (direction) parameter by the small step by the learning rate.



Stochastic Gradient Descent

Stochastic uses random entries to get derivative instead of the full dataset Why do we use Stochastic Gradient Descent?;; To find the derivative of discrete data so we can determine a straight line with the Least Square Error (LSE). What is Stochastic Gradient Descent?;; updates the model parameters based on the gradient of a single randomly chosen data point.

Batch gradient descent

What is Batch gradient descent?;; computes the gradient of the entire dataset,

Mini-batch gradient descent

Stochastic Mini-batched descent is the fastest way (groups then does randomly). What is Mini-batch gradient descent?;; Is a compromise of Batch gradient descent and Stochastic Gradient Descent.

What is the difference between batch gradient descent and stochastic gradient descent?;; Batch gradient descent computes the gradient of the cost function using the entire training dataset in each iteration, while stochastic gradient descent updates the model's parameters based on the gradient of the cost function with respect to one training example at a time. Mini-batch gradient descent is a compromise, using a subset of the training data in each iteration.

Gradient Descent

Gradient descent is commonly used in:

- **Deep Learning**: Frameworks like TensorFlow and PyTorch use variations of gradient descent for training.
- **Custom Implementations**: If you write logistic regression from scratch, gradient descent is a straightforward optimization method.

How Gradient Descent Works

Gradient Descent, a common Optimisation techniques, iteratively updates the Model Parameters by computing the gradient of the loss function with respect to the parameters. The update formula is:

$$\theta = \theta - \alpha \nabla_\theta \text{Cost}(\theta)$$

Where:

- θ are the parameters (intercept and coefficients).
- α is the learning rate (step size for updates).
- $\nabla_\theta \text{Cost}(\theta)$ is the gradient of the cost function with respect to the parameters θ.

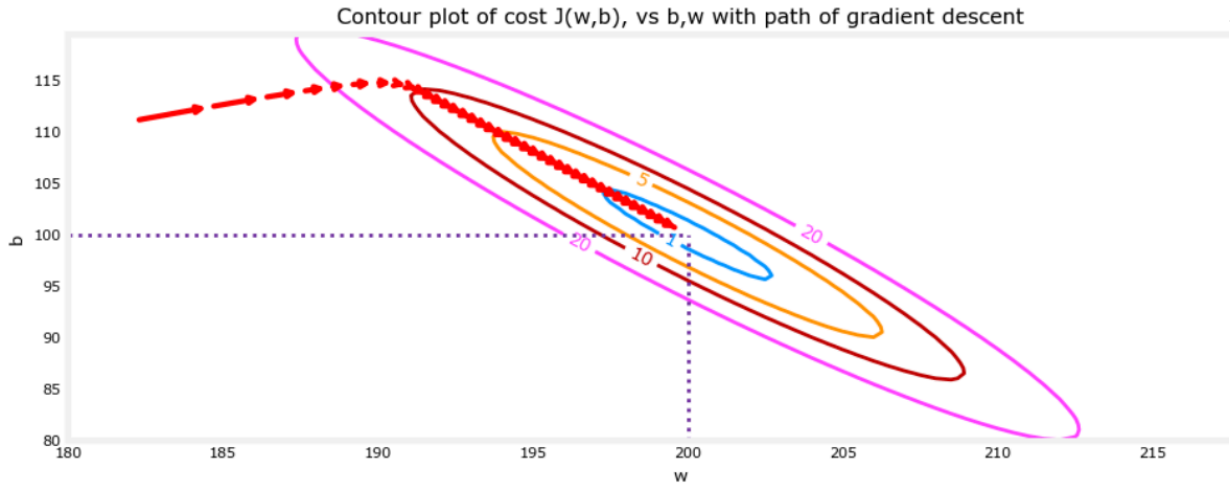
Process:

1. Calculate the gradient of the loss function.
2. Adjust the parameters in the direction of the negative gradient (to reduce loss).
3. Repeat until either:
    - The loss function converges (minimal change between updates), or
    - The maximum number of iterations is reached.

Gradient Descent

Cost Function value versus number of iterations of Gradient Descent should decrease

Can use contour plots to show Gradient Descent# moving towards minima.



Graph Theory Community

In graph theory, a community (also known as a cluster or module) is a group of nodes that are more densely connected to each other than to the rest of the network.

graph_analysis #clustering

https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.community.louvain.louvain_communities.html

Intuition

Communities often represent:

- Functional units in biological networks (e.g., protein complexes)
- Groups of friends or followers in social networks
- Topical clusters in knowledge graphs or citation networks

They capture meso-scale structure—between the local (node/edge) and global (graph-level) scale.

Formal Definition

There is no single universal definition, but communities typically exhibit:

- High intra-community density: lots of edges within the group
- Low inter-community density: few edges connecting to other groups

Mathematically, a common goal is to maximize modularity, a measure that quantifies the density of links inside communities compared to links between them.

Community Detection Algorithms

Some widely used algorithms:

Algorithm	Description
Louvain	Fast and widely used; optimizes modularity
Girvan–Newman	Based on removing high-betweenness edges
Label Propagation	Propagates labels through the network
Leiden	Improved version of Louvain for better quality and performance

Interpretability

Links

1. Interpretability Importance
2. https://christophm.github.io/interpretable-ml-book/index.html

Interpretability

Interpretability in machine learning (ML) is about understanding the reasoning behind a model's predictions. It involves making the model's decision-making process comprehensible to humans, which is crucial for trust, debugging, and ensuring fairness and reliability.

Importance of Interpretability

- **Trust**: Stakeholders are more likely to trust models they understand.
- **Debugging**: Easier to identify and fix issues in interpretable models.
- **Bias Detection**: Helps identify biases in data and model predictions.
- **Social Acceptance**: Models that can explain their decisions are more socially acceptable.
- **Fairness and Reliability**: Ensures models are fair and reliable, especially in high-impact areas.

Levels of Interpretability

- **Global, Holistic Model Interpretability**: Involves comprehending the entire model at once, including feature importance and interactions. This level of interpretability is challenging, especially for models with many parameters.
- **Global Model Interpretability on a Modular Level**: Focuses on understanding parts of the model (e.g., weights in linear models or splits in decision trees). While individual parameters may be interpretable, their interdependence complicates interpretation.
- **Local Interpretability for a Single Prediction**: Allows for detailed examination of why a model made a specific prediction for an individual instance. This can provide clearer insights as local predictions may exhibit simpler relationships than the global model.

Challenges in Achieving Interpretability

- Effective interpretation requires **context**; for instance, understanding the significance of weights in linear models is often conditional on other feature values.
- **Trade-offs**: Users must weigh the need for predictions against the need for understanding the rationale, particularly in contexts where decisions have significant consequences.
- **Human Learning**: Interpretability supports human curiosity, facilitating updates to mental models based on new information.

- **Safety and Bias Detection**: Essential for high-risk applications (e.g., self-driving cars) and for identifying biases in decision-making.
- **Social Acceptance**: Machines that explain their decisions tend to be more accepted.

Properties of Explanations

These properties provide a framework for evaluating and comparing explanation methods in interpretable machine learning, ensuring they are effective and useful for understanding model predictions.

Properties of Explanation Methods

1. **Expressive Power**: Refers to the types of explanations generated (e.g., rules, decision trees, natural language).
2. **Translucency**: Measures the extent to which an explanation method examines the model's internal parameters. High translucency allows for more informative explanations, while low translucency enhances portability.
3. **Portability**: Indicates the range of models compatible with the explanation method. Methods that treat models as black boxes (e.g., surrogate models) are more portable.
4. **Algorithmic Complexity**: Reflects the computational demands of generating explanations, which is crucial when processing time is a concern.

Properties of Explanations

1. **Accuracy**: Assesses how well the explanation predicts unseen data. High accuracy is vital if the explanation is used in place of the model.
2. **Fidelity**: Evaluates how closely the explanation matches the black box model's predictions. High fidelity is essential; otherwise, the explanation is ineffective.
3. **Consistency**: Measures how similar explanations are across models trained on the same task. High consistency is desirable when models rely on similar relationships.
4. **Stability**: Examines how consistent explanations are for similar instances. High stability is preferred to avoid erratic changes due to minor variations in input features.
5. **Comprehensibility**: Assesses how easily humans understand the explanations. This property is challenging to define but is critical for effective communication of model behavior.
6. **Certainty**: Indicates whether the explanation reflects the model's confidence in its predictions, adding value by clarifying prediction reliability.
7. **Degree of Importance**: Evaluates how well the explanation identifies the importance of features involved in a decision.
8. **Novelty**: Addresses whether the instance to be explained lies outside the training data distribution, affecting prediction accuracy.
9. **Representativeness**: Measures how many instances an explanation covers, ranging from individual predictions to broader model interpretations.

Understanding an Explanation

Here are the key takeaways on human-friendly explanations in interpretable machine learning:

Need comprehensibility and accuracy in explanations to enhance user understanding and trust in machine learning models.

Importance of Human-Friendly Explanations

1. **Preference for Short Explanations**: Humans favor concise explanations (1-2 causes) that contrast current situations with hypothetical scenarios where the event did not occur.
2. **Nature of Explanations**: An explanation answers "why" questions, focusing primarily on everyday situations rather than general scientific queries.

Characteristics of Good Explanations

1. **Contrastive Nature**: Good explanations highlight differences between predicted outcomes, aiding comprehension. For instance, explaining why a loan was rejected by comparing it to a hypothetical accepted application is more effective.
2. **Selective Focus**: People tend to prefer explanations that identify one or two key causes rather than exhaustive lists. This selective approach aligns with the "Rashomon Effect," where multiple valid explanations can exist for the same event.
3. **Social Context**: Explanations are influenced by the social context and audience. Tailoring explanations to the audience’s knowledge level enhances understanding.
4. **Emphasis on Abnormal Causes**: Humans focus on rare or abnormal causes to explain events. Including these in explanations can significantly enhance clarity.
5. **Truthfulness vs. Selectivity**: While truthfulness is important, selectivity often takes precedence. A concise, selective explanation is more impactful than a comprehensive but complex one.
6. **Consistency with Prior Beliefs**: Explanations that align with the explainee's existing beliefs are more readily accepted, highlighting the challenge of integrating complex model behaviors that contradict common intuitions.
7. **Generality**: Good explanations should be generalizable, but abnormal causes can sometimes provide more compelling insights.

Implications for Interpretable Machine Learning

- **Design Considerations**: Create explanations that are short, contrastive, and tailored to the audience’s background.
- **Methodology**: Incorporate techniques that can produce contrastive explanations while maintaining accuracy and fidelity to the model's predictions.
- **Audience Awareness**: Understanding the audience's social context and prior beliefs is crucial for effective communication of model outcomes.
- Understand how the model makes predictions.
- Use techniques like feature importance scores or LIME to explain individual predictions.
- **How can we design machine learning models that are both accurate and interpretable?** While deep learning models often achieve high accuracy, their complexity can make them difficult to interpret. This raises questions about how to balance accuracy and interpretability. Exploring techniques for visualizing and

understanding the internal representations learned by deep networks, or developing inherently interpretable models that still achieve high performance, could lead to greater trust and adoption of machine learning in critical applications like healthcare and finance.

Isolated Forest

Isolation Forest (iForest) is an Model Ensemble-based method used for anomaly detection. It operates by isolating data points using a series of random binary splits.

The key idea is that standardised/Outliers|anomalies, being rare and different, are easier to isolate and thus require fewer splits.

Mathematically, the isolation of a point is captured by the path length in a decision tree, where shorter paths indicate anomalies. The algorithm constructs multiple isolation trees, and the anomaly score of a point is determined by the average path length across all trees.

Isolation Forest is highly efficient for large datasets and is particularly useful when the assumption is that anomalies are rare and distinct from normal instances.

Steps:

- Randomly select a feature and a split value between the maximum and minimum values of that feature.
- Repeat this process to create a tree structure.
- Anomalies are isolated faster than normal points, leading to shorter path lengths in the tree.
- The average path length across multiple trees is used to compute an anomaly score.

Key Components:

- **Isolation Trees (iTrees)**: Binary trees where the goal is to isolate observations based on randomly chosen features and split values.
- **Anomaly Score**: Calculated based on the average path length across all isolation trees.
- **Path Length**: Anomalies tend to have shorter path lengths as they are easier to isolate.
- **Random Splitting**: Random feature selection and splitting result in the separation of instances, with fewer splits isolating anomalies.

Important

- Anomalies are identified based on shorter average path lengths in the isolation forest, indicating that fewer splits are needed to isolate them.
- The method scales well with large datasets because it relies on randomly generated trees, avoiding complex distance or density computations.
- Isolation Forest assumes that anomalies are few and distinct; it may perform poorly when anomalies are not easily distinguishable.
- The method is sensitive to the Hyperparameter such as the number of trees and sample size.

Follow up questions

- How does the isolation forest compare to density-based methods like DBSCAN in terms of detecting complex anomalies? Anomaly Detection with Clustering
- What impact does the choice of sample size have on the performance and accuracy of isolation forests in high-dimensional data?

Related Topics

- Random Forests for classification and regression
- One-Class Support Vector Machines|SVM for anomaly detection

K Means

K-means clustering is an Unsupervised Learning algorithm that partitions data into (k) clusters. Each data point is assigned to the cluster with the nearest centroid.

The algorithm partitions a dataset into k clusters by assigning data points to the closest cluster mean. The means are updated iteratively until convergence is achieved.

In ML_Tools see: K_Means.py

Key Features

- Unsupervised Learning: K-means organizes unlabeled data into meaningful groups without prior knowledge of the categories.
- Hyperparameter k: The number of clusters must be specified beforehand. The optimal number of clusters can be determined using WCSS and elbow method.

Algorithm Process:

1. Randomly choose k initial centroids.
2. Assign each data point to the nearest centroid.
3. Recalculate the centroids based on the current cluster assignments.
4. Repeat steps 2 and 3 until convergence (i.e., centroids no longer change significantly).

Visualization: Scatterplots can be used to visualize clusters and their centroids.

Adaptability: K-means can be updated with new data and allows for comparison of changes in centroids over time.

The initial centroids can effect the end results.

To correct this the algo is run multiple times with varying starting positions.

The centroids are updated after each iteration.

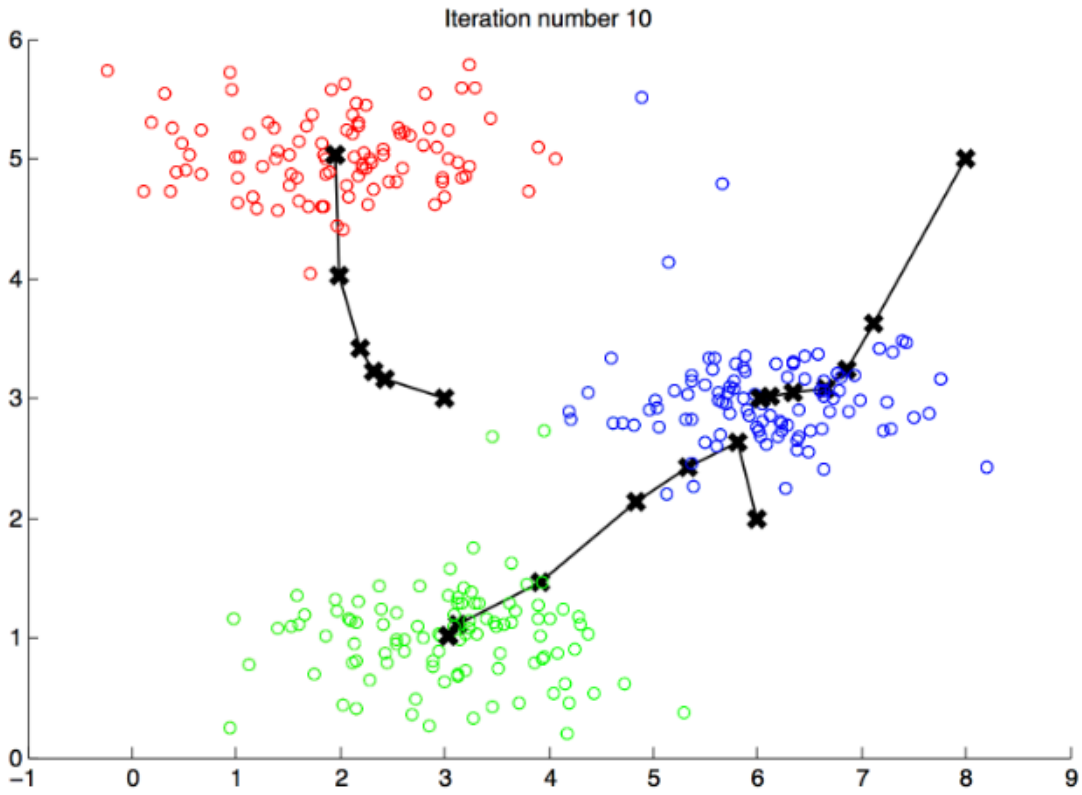


Figure 1: The expected output.

Limitations

- Sensitivity to Initialization: The algorithm is sensitive to the initial placement of centroids, which can affect the final clustering outcome.
- Predefined Number of Clusters: The number of clusters k must be specified in advance, which may not always be straightforward.

Resources

- Statquest Video on K-means

K Nearest Neighbours

K-nearest Neighbors is a non-parametric method used for both classification and regression tasks. It classifies a sample by a majority vote of its neighbors, assigning the sample to the class most common among its (k) nearest neighbors, where (k) is a small positive integer.

How It Works

- **Classification:** When a new data point needs classification, KNN identifies its (k) nearest neighbors in the training data based on feature similarity. The class label most common among these neighbors is assigned to the new data point.

- **Regression:** For regression tasks, KNN predicts the average value of the (k) nearest neighbors.

Applications

- Recommender systems
- Pattern recognition

Key Points

- **Non-parametric:** KNN does not make any assumptions about the underlying data distribution.
- **Supervised Learning:** Despite the note's mention of unsupervised learning, KNN is actually a supervised learning algorithm because it requires labeled training data.

Use Cases

- KNN is useful for tasks where the decision boundary is irregular and not easily captured by parametric models (parametric vs non-parametric models). It is simple to implement and understand but can be computationally expensive with large datasets.

Understanding K-nearest Neighbors

KNN is a straightforward algorithm that relies on the proximity of data points to make predictions. It is particularly effective in scenarios where the relationship between features and the target variable is complex and non-linear.

- **Choice of (k):** The value of (k) is crucial. A small (k) can be sensitive to noise, while a large (k) can smooth out the decision boundary too much.
- **Distance Metric:** The choice of distance metric (e.g., Euclidean, Manhattan) affects how neighbors are determined and can impact the algorithm's performance.

KNN is best suited for smaller datasets due to its computational intensity, as it requires calculating the distance between the new data point and all existing data points.

K-nearest neighbours

- Classifies a new data point based on the majority class of its k nearest neighbors.
- Simple but computationally expensive for large datasets.

Learning Rate

Description

The learning rate is a Hyperparameter in machine learning that determines the step size at which a model's parameters are updated during training. It plays a significant role in the optimization process, particularly in algorithms like Gradient Descent which are used to minimize the Loss function.

Key Points about Learning Rate:

1. **Parameter Updates**:
    - During training, the model's parameters (such as weights and biases in neural networks) are adjusted iteratively to minimize the loss function.

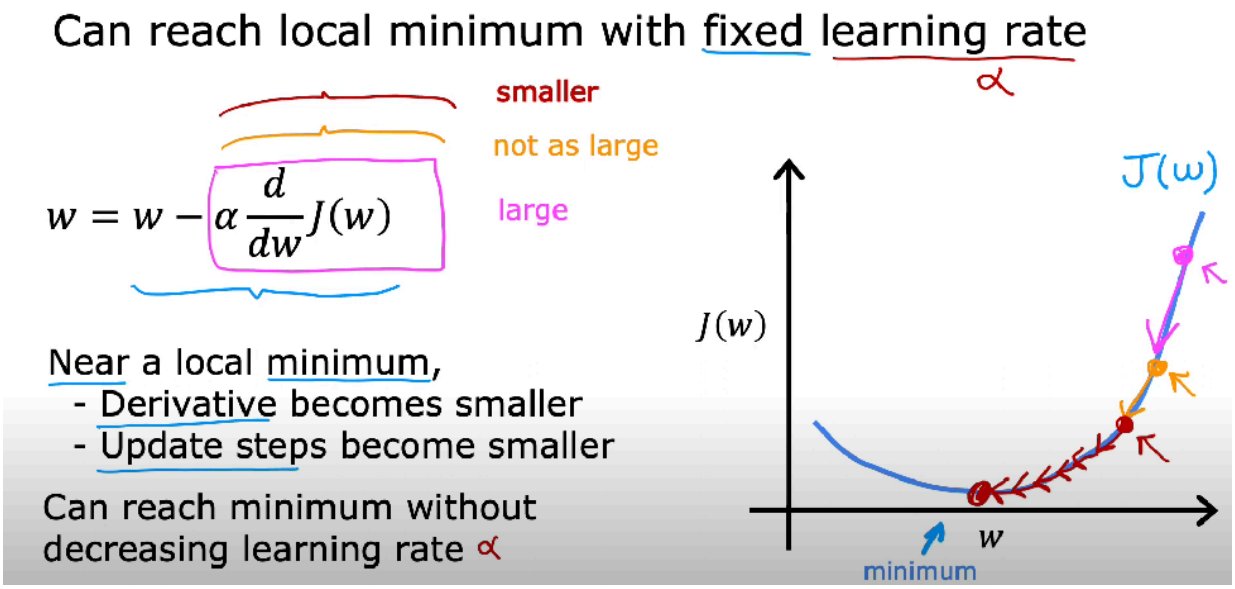
- The learning rate controls how much the parameters are changed in response to the estimated error each time the model weights are updated.

2. **Impact on Training**/ Convergence
   - A high learning rate can lead to faster convergence but risks overshooting the optimal solution, potentially causing the model to diverge.
   - A low learning rate ensures more stable and precise convergence but may result in slow training and can get stuck in local minima. A lower learning rate makes the model more robust but requires more iterations to converge.
   -
3. **Tuning**:
   - The learning rate is a hyperparameter that needs careful tuning. It can be adjusted manually or through automated hyperparameter optimization techniques like standardised/Optuna.
   - The optimal learning rate depends on various factors, including the dataset, model complexity, and the specific optimization algorithm used.
4. **Practical Considerations**:
   - It's common to start with a moderate learning rate and adjust based on the model's performance during training.
   - Techniques like learning rate schedules or adaptive learning rate methods (e.g., Adam Optimizer) can dynamically adjust the learning rate during training to improve convergence.

This impacts the efficiency of Gradient Descent

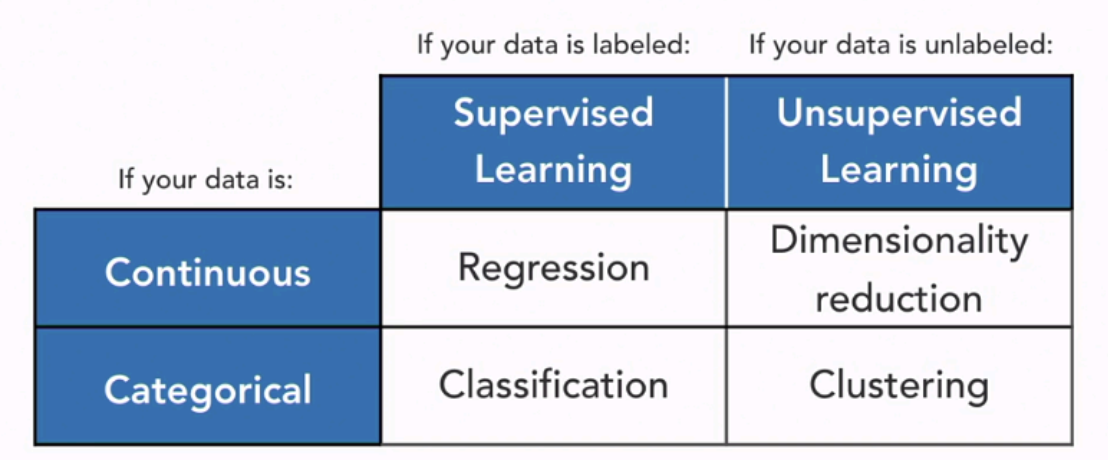
Effects occur if too small (takes long), or too large (over shoots missing the minima).

What happens if you are at a local minima? Then no change.



Learning Styles

What does the data look like continuous or categorical?



Unsupervised Learning Regression Classification

Unsupervised Learning Dimensionality Reduction Clustering

Lightgbm

LightGBM is a gradient boosting framework that is designed for speed and efficiency. It is particularly well-suited for handling large datasets and high-dimensional data.

- **Tree Growth**: Splits the tree leaf-wise, which can lead to faster convergence compared to level-wise growth.
- **Learning Rate**: Similar to Gradient Descent, LightGBM uses a learning rate to control the contribution of each tree.
- **DART**: A variant of LightGBM known for its performance.
- **Parameter Definition**: Requires parameters to be defined in a dictionary for model configuration.

Watch Video Explanation

Key Parameters

- **Learning Rate**: Controls the step size at each iteration while moving toward a minimum of the loss function.
- **Number of Leaves**: Determines the complexity of the tree model.

Advantages

- **Speed**: Renowned for its speed, often outperforming other gradient boosting implementations.
- **Memory Usage**: Optimizes memory usage, enabling efficient handling of large datasets.
- **Leaf-Wise Growth**: Grows trees leaf-wise, leading to faster convergence.
- **Parallel and GPU Learning**: Supports parallel and GPU learning for further speedup.

Use Cases

- **Large Datasets**: Ideal for applications where speed is crucial.
- **High-Dimensional Data**: Efficient when dealing with high-dimensional data and categorical features.

Linear Regression

Description

Linear regression assumes linearity between the input features and the target variable. Assumes that the relationship between the independent variable(s) and the dependent variable is linear.

During the training phase, the algorithm adjusts the slope (m) and the intercept (b) of the line to minimize the Loss function.

The linear regression model is represented as:

$$y = b_0 + b_1x_1 + b_2x_2 + \ldots + b_nx_n$$

- y is the dependent variable (the variable we want to predict).
- $x_1, x_2, \ldots, x_n$ are the independent variables (features or predictors).
- $b_0, b_1, b_2, \ldots, b_n$ are the coefficients (weights) associated with each independent variable.
- b_0 is the intercept term.

You evaluate the performance of your model by comparing its predictions to the actual values in a separate test dataset. Common metrics for evaluating regression models are:

- Mean Squared Error (MSE)
- Root Mean Squared Error (RMSE)
- R squared.

The goal of linear regression is to find the values of coefficients $b_0, b_1, b_2, \ldots, b_n$ that minimize the sum of squared errors (SSE), also known as the residual sum of squares (RSS) or (MSE - mean square error).

Mathematically, SSE is a Loss function given by:

$$SSE = \sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

where N is the number of observations, y_i is the actual value of the dependent variable for observation i, and $\hat{y}_i$ is the predicted value based on the linear regression model.

The formula for the Regression Sum of Squares (SSR) in the context of linear regression is:

$$SSR = \sum_{i=1}^{N}(\hat{y}_i - \bar{y})^2$$

Where:

- $\hat{y}_i$ is the predicted value of the dependent variable for observation i based on the linear regression model.
- $\bar{y}$ is the mean of the observed values of the dependent variable.
- N is the total number of observations.

SSR measures the amount of variability in the dependent variable that is explained by the independent variables in the model. It reflects how well the regression model captures the relationship between the independent and dependent variables.

Total Sum of Squares (SST) represents the total variability in the dependent variable y. The relationship between SST, SSR, and SSE is given by:

$SST = SSR + SSE$

This equation reflects the decomposition of total variability into explained variability (SSR) and unexplained variability (SSE) due to errors.

Ordinary Least Squares

The Ordinary Least Squares method is used to minimize SSE. It achieves this by finding the values of that minimize the sum of squared differences between the observed and predicted values. The formulas for are derived by setting partial derivatives of SSE with respect to each coefficient to zero.

OLS is an analytical method

Gradient Descent

It iteratively updates coefficients to minimize error.

Gradient descent is an optimization algorithm used to minimize the cost function in linear regression by iteratively adjusting the model parameters (coefficients). Here's how it works with linear regression:

1. **Initialize Parameters**: Start with initial guesses for the coefficients (weights), typically set to zero or small random values.
2. **Compute Predictions**: Use the current coefficients to make predictions for the dependent variable $\hat{y}$ using the linear regression model: $\hat{y} = b_0 + b_1x_1 + b_2x_2 + \ldots + b_nx_n$
3. **Calculate the Cost Function**:Compute the loss function, SSE.
4. **Compute the Gradient**: Calculate the gradient of SSE function with respect to each coefficient. The gradient is a vector of partial derivatives indicating the direction and rate of change of the cost function:

   $\frac{\partial J}{\partial b_j} = -\frac{2}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)x_{ij}$ Here, x_{ij} is the value of the j-th feature for the i-th observation.
5. **Update the Coefficients**: Adjust the coefficients in the opposite direction of the gradient to reduce the cost function. This is done using a learning rateα, which controls the size of the steps taken: $b_j = b_j - \alpha\frac{\partial J}{\partial b_j}$
6. **Iterate** & Converge Repeat steps 2 to 5 until the cost function converges to a minimum or a predefined number of iterations is reached. The algorithm converges when the changes in the cost function or the coefficients become very small, indicating that the minimum has been reached.

Model Evaluation

- R squared
- Adjusted R squared takes into account the number of variables.
- F-statistic overall significance of model (lower worse).
- Feature Selection Use P>|t| column to decide whether to keep variable less that 0.05
- p-values in linear regression in sklearn

```
# Add a constant. Essentially, we are adding a new column (equal in leng
x = sm.add_constant(x1)
# Fit the model, according to the OLS (ordinary least squares) method wi
results = sm.OLS(y,x).fit()
# Print a nice summary of the regression. That's one of the strong point
results.summary()
```

OLS Regression Results

Dep. Variable:	GPA	R-squared:	0.406
Model:	OLS	Adj. R-squared:	0.399
Method:	Least Squares	F-statistic:	56.05
Date:	Wed, 17 Jan 2024	Prob (F-statistic):	7.20e-11
Time:	14:38:34	Log-Likelihood:	12.672
No. Observations:	84	AIC:	-21.34
Df Residuals:	82	BIC:	-16.48
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	0.2750	0.409	0.673	0.503	-0.538	1.088
SAT	0.0017	0.000	7.487	0.000	0.001	0.002

Omnibus:	12.839	Durbin-Watson:	0.950
Prob(Omnibus):	0.002	Jarque-Bera (JB):	16.155
Skew:	-0.722	Prob(JB):	0.000310
Kurtosis:	4.590	Cond. No.	3.29e+04

```
# Print a nice summary of the regression.
results.summary()
```

[9]

OLS Regression Results

Dep. Variable:	GPA	R-squared:	0.407
Model:	OLS	Adj. R-squared:	0.392
Method:	Least Squares	F-statistic:	27.76
Date:	Wed, 17 Jan 2024	Prob (F-statistic):	6.58e-10
Time:	15:08:30	Log-Likelihood:	12.720
No. Observations:	84	AIC:	-19.44
Df Residuals:	81	BIC:	-12.15
Df Model:	2		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	0.2960	0.417	0.710	0.480	-0.533	1.125
SAT	0.0017	0.000	7.432	0.000	0.001	0.002
Rand 1,2,3	-0.0083	0.027	-0.304	0.762	-0.062	0.046

Omnibus:	12.992	Durbin-Watson:	0.948
Prob(Omnibus):	0.002	Jarque-Bera (JB):	16.364
Skew:	-0.731	Prob(JB):	0.000280
Kurtosis:	4.594	Cond. No.	3.33e+04

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 3.33e+04. This might indicate that there are strong multicollinearity or other numerical problems.

Impact of Extra Variables on Intercept:

When additional variables are introduced, it can impact the intercept (b_0) in the linear regression model. The intercept is the value of y when all independent variables ($x_1, x_2, \ldots, x_n$) are zero. The presence of extra variables can affect the baseline value of the dependent variable.

Logistic Regression

Logistic regression models the log-odds of the probability as a linear function of the input features.

It models the probability of an input belonging to a particular class using a logistic (sigmoid) function.

The model establishes a decision boundary (threshold) in the feature space.

Logistic regression is best suited for cases where the decision boundary is approximately linear in the feature space.

Logistic Regression can be used for Binary Classificationtasks.

Related Notes:

- Logistic Regression Statsmodel Summary table
- Logistic Regression does not predict probabilities
- Interpreting logistic regression model parameters
- Model Evaluation
- To get Model Parameters use Maximum Likelihood Estimation

In ML_Tools, see:

- Regression_Logistic_Metrics.ipynb

Key Concepts of Logistic Regression

Logistic Function (Sigmoid Function)

Logistic regression models the probability that an input belongs to a particular class using the logistic (sigmoid) function. This function maps any real-valued input into the range (0,1), representing the probability of belonging to the positive class (usually class 1).

The sigmoid function is defined as:

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

where

$$z = \mathbf{w} \cdot \mathbf{x} + b$$

Thus, the logistic regression model is given by:

$$P(y = 1 \mid \mathbf{x}) = \sigma(z)$$

Log odds: Transforming from continuous to 0-1

Logistic regression is based on the log-odds (logit) transformation, which expresses probability in terms of odds:

$$\text{Odds} = \frac{P(y=1|\mathbf{x})}{1-P(y=1|\mathbf{x})}$$

Taking the natural logarithm of both sides gives the logit function:

$$\log\left(\frac{P(y=1|\mathbf{x})}{1-P(y=1|\mathbf{x})}\right) = \mathbf{w} \cdot \mathbf{x} + b$$

This equation shows that logistic regression models the log-odds of the probability as a linear function of the input features.

Decision Boundary

- Similar to Support Vector Machines, logistic regression defines a decision boundary that separates the two classes.
- The logistic function determines the probability of a data point belonging to a specific class. If this probability exceeds a given threshold (typically 0.5), the model assigns the point to the positive class; otherwise, it is classified as negative.

Binary Classification

- Logistic regression is primarily used for binary classification tasks, where the target variable has only two possible values (e.g., "0" and "1").
- It can handle multiple independent variables (features) and assigns probabilities to the target classes based on the feature values.
- Examples include:

No Residuals

- Unlike Linear Regression, logistic regression does not compute standard residuals.
- Instead, Model Evaluation is performed by comparing predicted probabilities with actual class labels using metrics such as accuracy, precision, recall, and the Confusion Matrix.

Also see:

Related terms:

- Cost function for logistic regression
- Gradient computation in logistic regression
- Regularized logistic regression
- Cost function for regularized logistic regression

Logistic regression can be extended to handle non-linear decision boundaries through:

- Polynomial features to capture more complex relationships.
- Regularization techniques to improve generalization.

Explaining logistic regression

Loss Function

Loss functions are used in training machine learning models. Also known as a cost function, error function, or objective function. Serves as a metric for model evaluation.

Purpose: Measure predictive accuracy: Measures the difference between predicted and actual values. That is they measure how well a model's predictions match the actual target values by quantifying the error between the predicted output and the true output.

Goal: To be minimized: The primary goal during model training is to minimize this loss, improving accuracy of predictions on unseen data.

Used during training to adjust model parameters and during evaluation to assess model performance.

Examples

- **Mean Squared Error (MSE)**: Commonly used for regression tasks.
- **Cross Entropy**: Often used for classification tasks.

Resources

- Video Explanation
- Loss versus Cost function

Lstm

What is LSTM

LSTM (Long Short-Term Memory) networks are a specialized type of Recurrent Neural Network (RNN) designed to overcome the vanishing and exploding gradients problem that affects traditional Recurrent Neural Networks.

LSTMs address this challenge through their unique architecture.

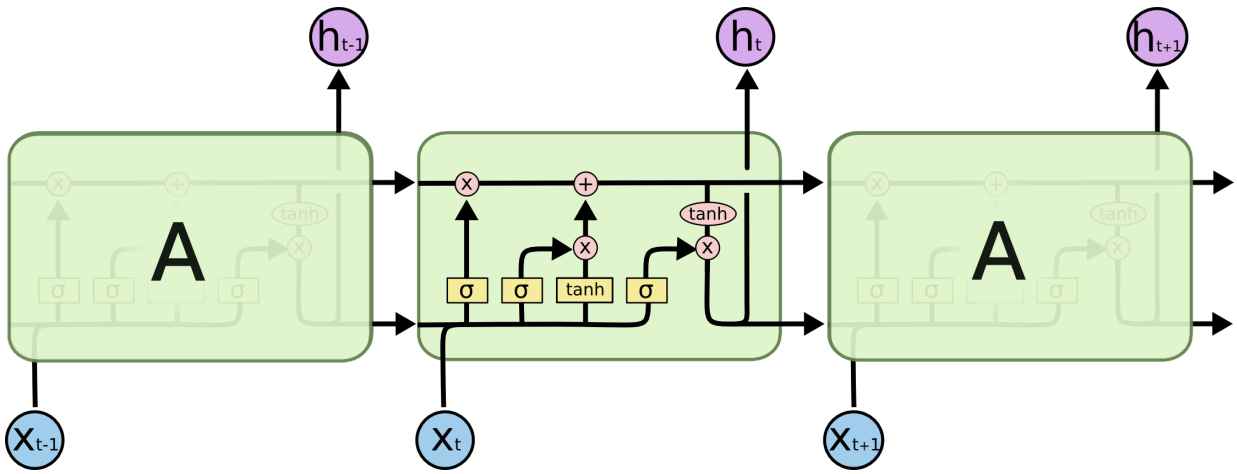
Used for tasks that require the retention of information over time, and problems involving sequential data.

The key strength of LSTMs is their ability to manage long-term dependencies using their gating mechanisms.

Key Components of LSTM Networks:

Resources: Understanding LSTM Networks

x_t input, h_t output, cell state C_t, conveyer belt



Memory Cell:

- The core of an LSTM network is the memory cell, which maintains information over long time intervals. This cell helps store, forget, or pass on information from previous time steps.

Gates:

- Input Gate: Controls how much of the input should be allowed into the memory cell.
- Forget Gate: Determines which information should be discarded from the memory cell.

- Output Gate: Controls what part of the cell's memory should be output as the hidden state for the current time step.

These gates are regulated by sigmoid activation, which output values between 0 and 1, acting like a filter to determine the amount of information that should pass through. This gate mechanism allows the LSTM network to maintain a balance between retaining relevant data and discarding unnecessary information over time.

Why is LSTM less favourable over using transformers

[!Summary]

Long Short-Term Memory (LSTM) networks, a type of Recurrent Neural Network (RNN), are less favorable than Transformer for many modern tasks, especially in Natural Language Processing (NLP). LSTMs process sequences of data one step at a time, making them inherently sequential and difficult to parallelize. Transformers, on the other hand, leverage a self-attention mechanism that allows them to process entire sequences simultaneously, leading to faster training and the ability to capture long-range dependencies more effectively.

Mathematically, LSTM's sequential nature leads to slower computations, while the Transformer's attention mechanism computes relationships between all tokens in a sequence, allowing better scalability and performance for tasks like translation, summarization, and language modeling.

[!Breakdown]

Key Components:

- Sequential Processing in LSTM: Each time step relies on the previous one, creating a bottleneck for long sequences.
- Self-Attention Mechanism in Transformers: Allows simultaneous processing of all elements in a sequence.
- Parallelization: Transformers leverage parallel computing more effectively due to non-sequential data processing.
- Positional Encoding: Used by Transformers to retain the order of the sequence, overcoming the need for explicit recurrence.

[!important]

- LSTMs are slower in training due to their sequential nature, as calculations depend on previous states.
- Transformers efficiently handle long-range dependencies using self-attention, which calculates the relationships between tokens directly without needing previous time steps.

[!attention]

- LSTMs suffer from vanishing/exploding gradient issues, especially in long sequences, limiting their effectiveness for long-term dependencies.
- Transformers require more data and computational power to train, which can be a limitation in resource-constrained environments.

[!Example]

In a language translation task, LSTMs process words sequentially, making them less efficient in handling long sentences. In contrast, a Transformer can analyze the entire sentence at once, using self-attention to determine relationships between all words, leading to faster and more accurate translations.

[!Follow up questions]

> - How does the attention mechanism in Transformers improve the model's ability to capture long-range dependencies compared to LSTM's cell structure?
> - In what cases might LSTM still be a better option over Transformers, despite their limitations?
>
> [!Related Topics]
>
> - Attention mechanism in deep learning
> - BERT (Bidirectional Encoder Representations from Transformers)
> - GRU (Gated Recurrent Unit) as an alternative to LSTM

Example Workflow in Python using Keras:

In this example, we define a simple LSTM model in Keras for a time series forecasting task. The model processes sequences with 1000 time steps, and the LSTM layer has 50 units, followed by a fully connected (Dense) layer for the final prediction.

```
import numpy as np
from keras.models import Sequential
from keras.layers import LSTM, Dense

# Sample data: time series with 1000 timesteps and 1 feature
time_steps = 1000
features = 1
X_train = np.random.rand(1000, time_steps, features)
y_train = np.random.rand(1000)

# Define LSTM model
model = Sequential()
model.add(LSTM(50, activation='tanh', return_sequences=False, input_shape=(time_steps, features)))
model.add(Dense(1))  # Output layer for regression tasks

model.compile(optimizer='adam', loss='mse')

# Train the model
model.fit(X_train, y_train, epochs=10, batch_size=64)
```

notes

LSTM How to implement LSTM with PyTorch? https://lightning.ai/lightning-ai/studios/statquest-long-short-term-memory-lstm-with-pytorch-lightning?view=public&section=all without lightning - there is a script there

LSTM

Machine Learning Algorithms

Machine learning Algorithms are used to automate tasks, extract insights, and make more informed decisions.

Choosing the right algorithm for a specific problem involves understanding the task, the characteristics of the data, and the strengths and limitations of different algorithms.

Supervised Learning

Common Classification algorithms include:

- Logistic Regression
- Support Vector Machines
- Naive Bayes
- Decision Tree
- Random Forests

Common Regression algorithms include:

- Linear Regression
- Support Vector Regression
- Random Forest Regression

Unsupervised Learning

Common Clustering algorithms include:

- K-means
- Gaussian Mixture Models
- Clustering
- Dimensionality Reduction

Common Dimensionality Reduction algorithms include:

- Principal Component Analysis
- Manifold Learning

Strengths and Limitations of Machine Learning Algorithms

Strengths:

Automation: Machine learning algorithms can automate complex tasks, freeing up human resources for other activities.

Adaptability: Machine learning algorithms can adapt to changing data patterns, making them suitable for dynamic environments.

Scalability: Machine learning algorithms can handle large datasets efficiently, making them applicable to big data problems.

Knowledge Discovery: Machine learning algorithms can help discover hidden patterns and relationships in data, leading to new insights and knowledge.

Limitations:

Data Dependence: The performance of machine learning algorithms heavily depends on the Data Quality and quantity of the training data.

Overfitting occurs when the model learns the training data too well and fails to generalise to new, unseen data.

Bias and variance: Machine learning algorithms can be biased, reflecting the biases present in the training data.

Interpretability: Some machine learning algorithms, especially deep learning models, can be complex and difficult to interpret, making it challenging to understand the reasoning behind their predictions.

Markov Decision Processes

Markov Decision Process (Markov Decision Processes|MDP) is a formal framework for decision-making where outcomes depend solely on the current state (Markov property). \

architecture

Markov Decision Processes (Markov Decision Processes|MDPs): The mathematical framework for modelling decision-making, characterized by states, actions, transition probabilities, and rewards. Your understanding of probability theory and stochastic processes will be crucial here.

Model Building

The Model Building phase follows the Preprocessing phase, where data is organized and prepared for analysis. This phase focuses on selecting and setting up the appropriate machine learning models to solve the problem at hand.

Key Steps

Types of Models:

- Choose a model to apply based on the problem requirements and data characteristics.
- Explore different Machine Learning Algorithms to find the best fit for your data.
- Consider the tradeoffs between parametric vs nonparametric models.

Setting Up a Model:

- Divide the data into Train-Dev-Test Sets to ensure robust evaluation and tuning.
- Optimize Model Parameters and configurations for best performance.

Model Selection:

- Evaluate the appropriateness of models in the Model Selection phase.

Model Deployment

Deploying a machine learning model involves moving it from a development environment to a production environment where it can make predictions on new data.

Steps for Model Deployment

Model Exporting

- Use tools like `joblib` or `pickle` to serialize the model.

```
import joblib
joblib.dump(model, 'linear_regression_model.pkl')
```

Deployment Options

- Application Integration: Embed the model into an application for real-time predictions.
- API Deployment: Use frameworks like Flask or FastAPI to create an API endpoint for the model.
- Automated Workflows: Integrate the model into automated data processing pipelines.

Tools and Platforms

- Sklearn Pipeline: Streamline the deployment process by integrating Preprocessing and model steps.
- Gradio: Create user-friendly interfaces for model interaction.
- Streamlit.io

Considerations

- Scalability: Ensure the deployment solution can handle the expected load.
- Model Observability: Implement monitoring to track model performance and detect issues.

Deploying using PyCaret

AWS: When deploying model on AWS S3, environment variables must be configured using the command-line interface. To configure AWS environment variables, type `aws configure` in terminal. The following information is required which can be generated using the Identity and Access Management (IAM) portal of your amazon console account:

- AWS Access Key ID
- AWS Secret Key Access
- Default Region Name (can be seen under Global settings on your AWS console)
- Default output format (must be left blank)

GCP: To deploy a model on Google Cloud Platform ('gcp'), the project must be created using the command-line or GCP console. Once the project is created, you must create a service account and download the service account key as a JSON file to set environment variables in your local environment. Learn more about it: https://cloud.google.com/docs/authentication/production

Azure: To deploy a model on Microsoft Azure ('azure'), environment variables for the connection string must be set in your local environment. Go to settings of storage account on Azure portal to access the connection string required. AZURE_STORAGE_CONNECTION_STRING (required as environment variable) Learn more about it: https://docs.microsoft.com/en-us/azure/storage/blobs/storage-quickstart-blobs-python?toc=%2Fpython%2Fazure%2FTOC.json

Model Ensemble

Ensemble models in machine learning are techniques that combine the predictions of multiple individual models to improve overall performance. Ensemble methods can achieve better accuracy and robustness than any single model alone.

Key Concepts of Ensemble Models:

1. **Diversity**: The strength of ensemble models lies in the diversity of the base models. Different models may capture different patterns or errors in the data, and combining them can lead to more accurate predictions.
2. **Combination**: Ensemble methods aggregate the predictions of individual models using techniques like averaging, voting, or weighted sums to produce a final prediction.

Main Ensemble Techniques:

- Bagging
- Boosting
- Stacking
- Isolated Forest

In ML_Tools see: Comparing_Ensembles.py

Further Understanding

Analogy:

- Ensemble methods can be likened to consulting multiple doctors for a diagnosis. Each doctor (model) may have a different opinion, but by considering all opinions, the final diagnosis (prediction) is more accurate than relying on a single doctor's opinion.

Advantages of Ensemble Models:

- **Increased Accuracy**: By combining multiple models, ensemble methods often achieve higher accuracy than individual models.
- **Robustness**: They are less sensitive to overfitting, especially when using techniques like bagging.
- **Flexibility**: Ensemble methods can be applied to various types of base models and are not limited to a specific algorithm.

Challenges:

- **Complexity**: Ensemble models can be more complex and computationally intensive than single models.
- **Interpretability**: The final model may be harder to interpret compared to simpler models like decision trees.

Model Evaluation

Assess the model's performance using various metrics to ensure it meets the desired accuracy and reliability.

Appropriate evaluation metrics are used based on the problem type (classification vs. regression), to assess how well the model predicts.

The aim is to improve accuracy but also to generalize and avoid biases and Overfitting.

- **Performance Assessment**: Models are evaluated on a testing set using metrics relevant to the problem type.
- **Generalization and Bias**: Evaluation includes assessing how well the model generalizes to new data and identifying any biases.

For categorical classifiers: Evaluation Metrics: Use metrics such as accuracy, precision, recall, F1-score, and confusion matrix to evaluate performance.

For regression tasks: Regression Metrics: Metrics like Mean Absolute Error (MAE), Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and R-squared (R^2) are used.

Cross Validation is a technique used to assess the performance of a model by splitting the data into multiple subsets for training and testing to assesses performance and generalization. It helps detect Overfitting, provides reliable performance estimates.

Feature Importance: After training, analyze which features have the most significant impact on the model's predictions.

Model Observability

Monitor the model's performance over time (in production). Similar to Model Validation.

In the context of machine learning (ML), Observability refers to the ability to monitor, understand, and diagnose the performance and behaviour of ML models in production.

It encompasses the processes, tools, and techniques that help practitioners ensure models are functioning as expected and identify when they deviate from desired outcomes.

Master Observability Datadog

Key aspects of observability in machine learning include:

Observability is a process in ML, and is usually achieved through logging, metrics collection, real-time monitoring, and advanced diagnostic tools integrated into the ML pipeline.

1. Monitoring Model Performance (monitoring metrics):
    - Tracking key metrics such as Accuracy, Precision,Recall,F1 Score,ROC (Receiver Operating Characteristic)d other relevant KPIs over time to identify performance degradation or improvements.
    - Monitoring Performance Drift in model inputs (features) and outputs (predictions) to detect when the model no longer performs well due to changes in data distribution (data drift) or changes in relationships between variables (Performance Drift).
2. Error and Isolated Forest:
    - Identifying when predictions are out of the expected range or when the model behaves abnormally, such as high error rates on specific subsets of data or excessive latency in prediction generation.
3. Interpretability:
    - Ensuring that the internal workings of the model (e.g., feature importance, decision pathways) are visible, interpretable, and explainable to humans. This allows for easier debugging and accountability, especially in critical applications such as finance, healthcare, or autonomous systems.
4. data lineage and Provenance:
    - Tracking the data sources, transformations, and processes that influence the model's input data. This provides visibility into how data flows through the pipeline and helps in reproducing results or addressing data-related issues.
5. Pipeline Monitoring:
    - Observing the entire ML pipeline from data ingestion and preprocessing to model training, validation, and deployment. This includes identifying bottlenecks, delays, and system failures that may affect the model's ability to make predictions in real-time.
6. Alerts and Automation:

- Setting up automated alerts when certain thresholds are breached, such as a sudden drop in accuracy or an increase in response time. This allows for prompt interventions, whether retraining the model, adjusting the pipeline, or tuning hyperparameters.

Why Observability Matters in Machine Learning:

- Ensures Reliability: Observability provides insights into how models behave under different conditions, ensuring that they remain reliable and consistent in their performance.
- Prevents Model Drift (Performance Drift): With observability, teams can detect model drift early, enabling them to retrain or recalibrate the model before performance deteriorates.
- Improves Accountability: Particularly in high-stakes applications, having observability in place allows organizations to understand and justify the model's decisions.
- Supports Continuous Monitoring: Observability is critical in ML systems that operate continuously in production, ensuring they are making accurate and meaningful predictions over time.

Monitor the model's performance over time. If the data distribution changes (concept drift), or the model's accuracy declines, retraining or updating the model may be necessary.

Related to:

- Data Observability
- Model Validation

Model Parameters Tuning

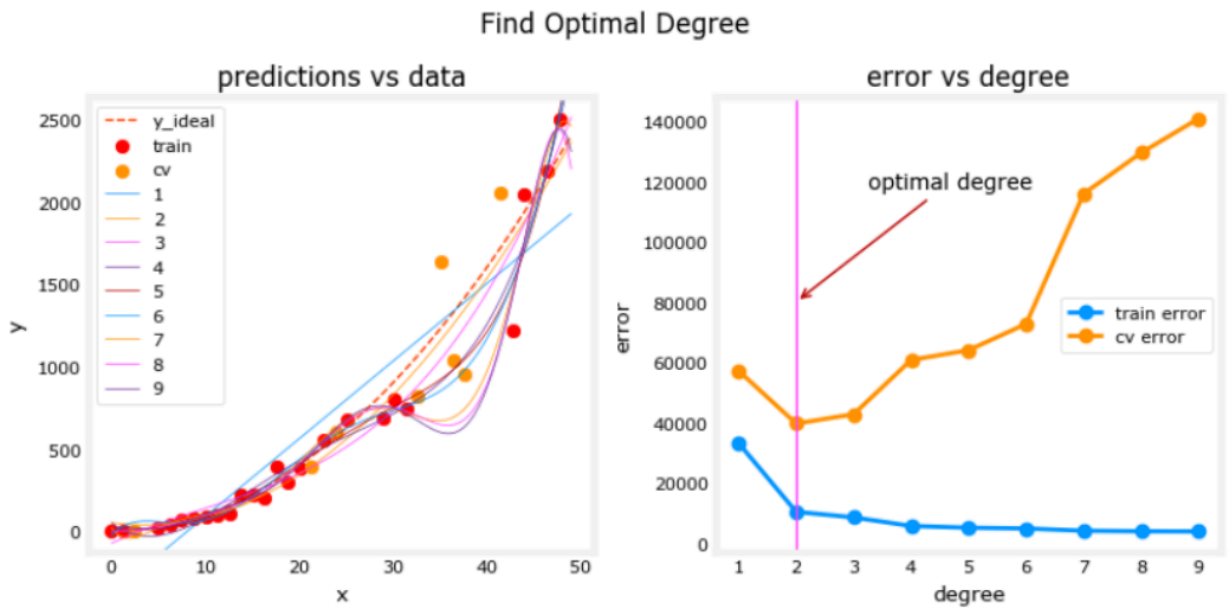
To find optimal Model Parameters.

Finding Optimal Model Parameters

1. Parameter Space Exploration:
    - It's useful to visualize slices of the parameter space by selecting two parameters at a time. This helps in understanding how different parameter values affect the model's performance.
2. Cost Function
    - The cost function is used to find the minimum error in predictions. It measures the difference between predicted and actual values, and the goal is to minimize this function to improve model accuracy.
3. Optimisation function
    - Ideal parameters are found using optimization functions, which adjust the model parameters to minimize the loss function. Common optimization algorithms include Gradient Descent, Adam Optimizer, and Stochastic Gradient Descent.
4. Data Splitting:
    - Split the data into training and cross-validation sets to evaluate model performance. Plot the parameter of interest on the x-axis and accuracy on the y-axis to visualize performance.

Optimisation techniques

Example



To find optimal model parameters, graph the parameter against error of the model.

On the left plot, the solid lines represent the predictions from these models. A polynomial model with degree 1 produces a straight line that intersects very few data points, while the maximum degree hews very closely to every data point.

On the right:

```
- the error on the trained data (blue) decreases as the model complexity increases as expected
- the error of the cross-validation data decreases initially as the model starts to conform to the data, but then increase
```

Model Selection

Model selection is an integral part of building a Machine Learning Operations to ensure that the best performing model is chosen for a given task, avoiding issues like overfitting or underfitting.

This is a crucial step because the model's ability to generalize to unseen data depends on selecting the right one.

Model selection typically involves the following steps:

1. Define candidate models: These can be models of different types (e.g., decision trees, support vector machines, neural networks) or the same model type but with varying hyperparameters.
2. Train each model: Train all the candidate models on the training set using different algorithms or parameter settings.
3. Evaluate performance (Model Evaluation): Use a validation set or cross-validation to evaluate the performance of each model. Common evaluation metrics include accuracy, precision, recall, F1 score, and mean squared error, depending on the type of problem (classification or regression).
4. Select the best model: Based on the evaluation metrics, choose the model that performs best on the validation set. The aim is to balance bias and variance to achieve good generalization to unseen data.

5. Test on unseen data: Finally, test the selected model on a test set to ensure that it generalizes well and has not been overfitted to the validation data.

Common approaches for model selection include:

- GridSeachCv and Random Search for hyperparameter tuning.
- Cross Validation to ensure robustness by evaluating model performance on different subsets of the data.
- Bayesian Optimization, which can be used to efficiently search the hyperparameter space.
- Choose the best-performing model based on Evaluation Metrics and optimization results.
- Cross Validation: Evaluate the model more robustly by splitting the training data into smaller chunks and training the model multiple times.
- Model Interpretability: Utilize tools to understand and interpret the model's predictions, ensuring transparency and trustworthiness.

Multi Head Attention

Summary

Aggregates different perspectives

This approach allows the model to attend to different parts of the input sequence simultaneously, capturing various aspects of the context more effectively.

Like a hydra, it focuses on different aspects of the context. Getting a finer understanding.

Multi-head attention captures more context by dividing the input processing into multiple independent attention heads. Each head focuses on different parts of the input and captures diverse types of relationships, both local and global. This parallelism allows the model to learn multiple perspectives simultaneously, enriching its understanding of the input sequence and improving performance on complex tasks like language modeling, machine translation, and more.

Related to the Attention mechanism.

Multi-Head Attention

In **multi-head attention**, the idea is to split the input into multiple subsets of attention heads. Instead of computing a single attention score for each token pair, multiple attention "heads" are used, with each head attending to different parts of the input. This provides several benefits:

a) Diverse Attention Patterns

Each head in multi-head attention can focus on different aspects of the input sequence, allowing the model to capture multiple relationships between tokens. For instance:

- One head may focus on syntactic relationships (like word order or structure).
- Another head may focus on semantic relationships (like the meaning or context of words).

By having multiple attention heads, the model can learn to capture **different types of context simultaneously**.

b) Different Projection Spaces

Each attention head has its own set of parameters, which project the input into a different subspace (i.e., they use different weight matrices to transform queries, keys, and values). This allows each head to learn different relationships between tokens in various representational subspaces, thus increasing the model's capacity to understand complex dependencies.

For example, one head might focus on short-range dependencies (like adjacent words), while another head could capture long-range dependencies (like relationships between words at opposite ends of the sentence).

c) Improved Expressiveness

By combining the outputs of multiple heads, the model gains a richer representation of the context. Each attention head contributes unique insights about how tokens in the sequence relate to each other, and by concatenating these different perspectives, the overall attention mechanism becomes more expressive.

Multi-Head Attention Process

The process for multi-head attention involves the following steps:

1. **Linear Transformations**: The input vectors (representing words or tokens) are linearly transformed into **queries (Q)**, **keys (K)**, and **values (V)** for each head. These transformations are different for each head, allowing each head to capture different relationships in the data.
2. **Attention Calculation for Each Head**: For each head, scaled dot-product attention is calculated independently. The attention mechanism computes scores by comparing queries and keys, and these scores are used to weigh the values. This results in a unique context vector for each head.
3. **Concatenation**: The outputs from all the attention heads are concatenated into a single vector. This step combines the different perspectives learned by each head.
4. **Final Linear Transformation**: After concatenation, a final linear transformation is applied to combine the information from all heads into a single vector that can be used in the next layer of the model.

How Multi-Head Attention Captures More Context

Multi-head attention captures more context than single-head attention for several reasons:

- **Multiple Focus Areas**: By using multiple heads, the model can simultaneously focus on different parts of the sequence. Some heads might attend to local dependencies, while others might capture more distant relationships. This gives the model a broader understanding of the entire sequence.
- **Handling Ambiguity**: In natural language, the meaning of words can depend heavily on context. Multi-head attention allows the model to disambiguate meanings by attending to different context clues in parallel. For instance, the word "bank" in "I went to the bank" can mean different things, and different heads can capture clues from the surrounding words to determine whether "bank" refers to a financial institution or a riverbank.
- **Diverse Representations**: Each head transforms the input into different representational subspaces, meaning the model learns diverse representations of the same input. This diversity enhances the model's ability to generalize and capture complex relationships in the data.

Application Example: Language Translation

Consider translating a sentence from one language to another. The multi-head attention mechanism in the Transformer model helps capture different linguistic structures:

- One head might focus on aligning subject-verb pairs between the two languages.
- Another head might capture longer dependencies, like how nouns and pronouns refer to each other across a long sentence.
- A third head might capture grammatical structure differences between the source and target languages.

By aggregating these different perspectives, multi-head attention ensures that the model understands both the local and global context, leading to better translation quality.

Naive Bayes

Can values for X,y be categroical ? Encoding Categorical Variables

BernoulliNB()

Why Naive Bayes?;;Order doesn't matter, features are independent. Treated it as a Bag of words. Which simplifies the above equation.

Want to use this in classifiers for ML Want to understand: Multinomial Naive bayes classifer There is also: Gaussian Naive Bayes

Issues

To avoid having 0 probability sometimes they add counts α to do this.

Links:

https://youtu.be/PPeaRc-r1OI?t=169

Formula

$P(A|B) = P(A) \times \frac{P(B|A)}{P(B)}$ Think of the line as "given".

Examples

Example 1

Example: Picnic Day

You are planning a picnic today, but the morning is cloudy

- Oh no! 50% of all rainy days start off cloudy!
- But cloudy mornings are common (about 40% of days start cloudy)
- And this is usually a dry month (only 3 of 30 days tend to be rainy, or 10%)

What is the chance of rain during the day?

We will use Rain to mean rain during the day, and Cloud to mean cloudy morning.

The chance of Rain given Cloud is written P(Rain|Cloud)

So let's put that in the formula:

$$P(\text{Rain}|\text{Cloud}) = \frac{P(\text{Rain})\,P(\text{Cloud}|\text{Rain})}{P(\text{Cloud})}$$

- P(Rain) is Probability of Rain = 10%
- P(Cloud|Rain) is Probability of Cloud, given that Rain happens = 50%
- P(Cloud) is Probability of Cloud = 40%

$$P(\text{Rain}|\text{Cloud}) = \frac{0.1 \times 0.5}{0.4} = .125$$

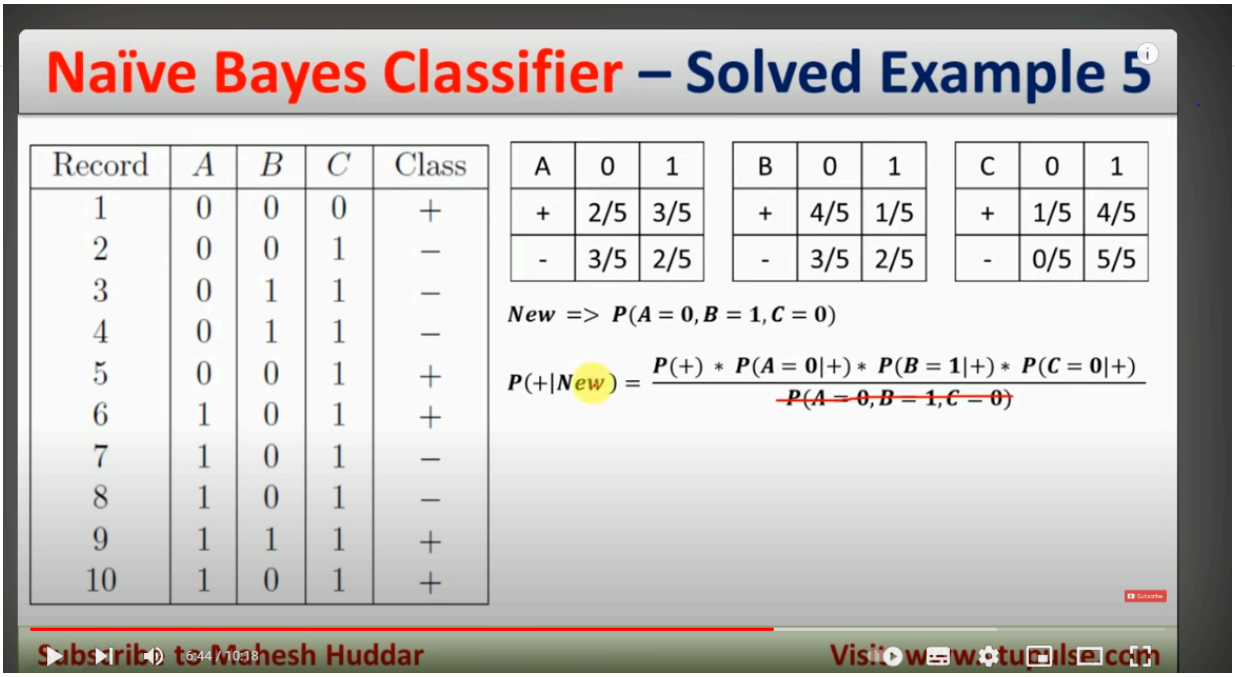
Or a 12.5% chance of rain. Not too bad, let's have a picnic!

Example 2

In the formula above P(A) is P(+), P(B)=P(NEW)

P(B|A) = P(A=0|+)... P(C=0|+)

P(A=0,B=1,C=0) is the same for both + and - class so remove.



Example Car accidents

What's the probability of car having an accident given that driver is driving in summer, there is no rain, it's a night and it's an urban area?

Mock data:

Season	Weather	Daytime	Area	Did Accident Occur?
Summer	No-Raining	Night	Urban	No
Summer	No-Raining	Day	Urban	No
Summer	Raining	Night	Rural	No
Summer	Raining	Night	Urban	Yes
Summer	Raining	Day	Urban	No
Summer	Raining	Night	Rural	No
Winter	Raining	Night	Urban	Yes
Winter	Raining	Night	Urban	Yes
Winter	Raining	Night	Rural	Yes
Winter	No-Raining	Night	Rural	No
Winter	No-Raining	Night	Urban	No
Winter	No-Raining	Day	Urban	Yes
Spring	No-Raining	Night	Rural	Yes
Spring	No-Raining	Day	Rural	Yes
Spring	Raining	Night	Urban	No
Spring	Raining	Day	No	No
Spring	No-Raining	Night	Urban	No
Autumn	Raining	Night	Urban	Yes
Autumn	Raining	Day	Rural	Yes
Autumn	No-Raining	Night	Urban	No
Autumn	No-Raining	Day	Rural	No
Autumn	No-Raining	Day	Urban	No
Autumn	Raining	Day	Yes	No
Autumn	Raining	Night	Yes	No
Autumn	No-Raining	Night	No	No

To handle data like this it is possible to calculate frequencies for each case:

0. Accident probability

$P(Accident) = \frac{9}{25} = 0.36$

$P(No - Accident) = \frac{16}{25} = 0.64$

1. Season probability

Frequency table:

Season	Accident	No Accident	
Spring	2/9	3/16	5/25
Summer	1/9	5/16	6/25
Autumn	2/9	6/16	8/25
Winter	4/9	2/16	6/25
	9/25	16/25	

Probabilities based on table:

$P(Spring) = \frac{5}{25} = 0.20$

$P(Summer) = \frac{6}{25} = 0.24$

$P(Autumn) = \frac{8}{25} = 0.32$

$P(Winter) = \frac{6}{25} = 0.24$

$P(Spring|Accident) = \frac{2}{9} = 0.22$

$P(Summer|Accident) = \frac{1}{9} = 0.11$

$P(Autumn|Accident) = \frac{2}{9} = 0.22$

$P(Winter|Accident) = \frac{4}{9} = 0.44$

2. Weather probability

Frequency table:

	Accident	No Accident	
Raining	6/9	7/16	13/25
No-Raining	3/9	9/16	12/25
	9/25	16/25	

Probabilities based on table:

$P(Raining) = \frac{13}{25} = 0.52$

$P(No - Raining) = \frac{12}{25} = 0.48$

$P(Raining|Accident) = \frac{6}{9} = 0.667$

$P(No - Raining|Accident) = \frac{12}{25} = 0.333$

3. Daytime probability

Frequency table:

	Accident	No Accident	
Day	3/9	6/16	9/25
Night	6/9	10/16	16/25
	9/25	16/25	

Probabilities based on table:

$P(Day) = \frac{9}{25} = 0.36$

$P(Night) = \frac{16}{25} = 0.64$

$P(Day|Accident) = \frac{3}{9} = 0.333$

$P(Night|Accident) = \frac{6}{9} = 0.667$

4. Area probability

Frequency table:

	Accident	No Accident	
Urban Area	5/9	8/16	13/25
Rural Area	4/9	8/16	12/25
	9/25	16/25	

Probabilities based on table:

$P(Urban) = \frac{13}{25} = 0.52$

$P(Rural) = \frac{12}{25} = 0.48$

$P(Urban|Accident) = \frac{5}{9} = 0.556$

$P(Rural|Accident) = \frac{4}{9} = 0.444$

Assemble:

Calculating probablity of car accident occuring in summer, when there is no rain and during night, in urban area.

Where B equals to:

- Season: Summer
- Weather: No-Raining
- Daytime: Night
- Area: Urban

Where A equals to:

- Accident

Using Naive Bayes:

$$P(A|B) = P(Accident|Season = Summer, Weather = No - Raining, Daytime = Night, Area = Urban)$$

$$P(A|B) = \frac{P(Summer|Accident)P(No-Raining|Accident)P(Night|Accident)P(Urban|Accident)P(Accident)}{P(Summer)P(No-Raining)P(Night)P(Urban)}$$

$$P(A|B) = \frac{\frac{1}{9}\frac{6}{9}\frac{6}{9}\frac{5}{9}\frac{9}{25}}{\frac{6}{25}\frac{12}{25}\frac{10}{25}\frac{13}{25}} = \frac{0.111 \cdot 0.667 \cdot 0.667 \cdot 0.556 \cdot 0.36}{0.24 \cdot 0.48 \cdot 0.64 \cdot 0.52} = \frac{0.0099}{0.038} = 0.26$$

$$P(A) = P(Accident)$$

$$P(B) = P(Summer)P(No - Raining)P(Night)P(Urban)$$

$$P(B|A) = P(Summer|Accident)P(No - Raining|Accident)P(Night|Accident)P(Urban|Accident)$$

What is the Bayes theorem?;; The formula is P(A|B) = P(B|A) * P(A) / P(B).

What are the main advantages of Naive Bayes, and when is it commonly used?;; simplicity, quick implementation, and scalability, used in text classification.

**When using Naive Bayes with numerical variables, what condition is assumed on the data?;; Naive Bayes assumes that numerical variables follow a normal distribution.

How does Naive Bayes perform with categorical variables? makes no assumptions about the data distribution.

What is Naive Bayes, and why is it called "naive"?;; Algo which uses Bayes theorem, used for classification problems. It is "naive" because it assumes that predictor variables are independent, which may not be the case in reality. The algorithm calculates the probability of an item belonging to each possible class and chooses the class with the highest probability as the output.

Naive Bayes Naive Bayes classifiers are based on Bayes' theorem and assume that the features are conditionally independent given the class label.

Naive Bayes

- A probabilistic classifier based on Bayes' theorem.
- Simple and fast, especially effective for text classification.

Named Entity Recognition

Named Entity Recognition (NER) is a subtask of NLP|Natural Language Processing (NLP) that involves identifying and classifying key entities in text into predefined categories such as names, organizations, locations.

The process typically employs algorithms like Conditional Random Fields (CRFs) or deep learning models such as Bi-directional LSTM (Long Short-Term Memory) networks.

Mathematically, NER can be framed as a sequence labeling problem where the goal is to assign a label y_i to each token x_i in a sentence. The model learns from annotated datasets, optimizing parameters to maximize the likelihood $P(y|x)$ using techniques like backpropagation.

NER has significant implications in information extraction, search engines, and automated customer support systems.

Important

- NER transforms unstructured text into structured data for analysis.
- The choice of model significantly impacts the accuracy of entity recognition.

Example

An example of NER is identifying "Apple Inc." as an organization in the sentence: "Apple Inc. released a new product."

Follow up questions

- How does the choice of training data affect the performance of NER models
- What are the challenges of NER in multilingual contexts
- Why is named entity recognition (NER) a challenging task
- In NER how would you handle ambiguous entities

Related Topics

- Text classification in NLP
- Information extraction techniques

Neural Network

A Neural network|Neural Network is a computational model inspired by biological neural networks in the human brain. It consists of layers of interconnected nodes (neurons) that process and transmit information. Neural networks are fundamental to Deep Learning.

Resources:

- Keras Guide

Also see:

- Types of Neural Networks
- Neural network in Practice

Key Components

The number of starting nodes depends on the input parameter, similar for output. The width and depth of the net are called Hyperparameter.

Neurons (Nodes):

- The basic units of a neural network. Each neuron receives input, processes it, and passes it to the next layer. A neuron's output is typically computed using a mathematical function known as the activation function.

Layers:

- Neural networks are structured in layers of neurons:
  - Input Layer: Receives the raw input data (features) that are fed into the network.
  - Hidden Layers: Process the inputs received from the previous layer. There can be multiple hidden layers, making a neural network "deep." These layers transform the data to learn complex relationships and patterns.
  - Output Layer: Produces the final prediction or result, such as a class label in classification tasks or a continuous value in regression.

Weights and Biases: Model Parameters

- There are weights and biases at each layer. The shape of the weights is determined by the number of units in the previous layer and the number of units in the current layer.
- Each connection between neurons has a weight that determines how much influence one neuron has on another. Weights are adjusted during the learning process to minimize prediction errors.
- Biases allow the network to shift the output of the activation function and help it better fit the data.

Training: See Fitting weights and biases of a neural network

Optimization: See Optimisation techniques

- The optimization process (often gradient descent) updates the network's weights to minimize the loss function, ensuring the model improves with training and generalizes well to new, unseen data.

Inputs:

- We need Normalisation of values (inputs) here for feeding the network to have balanced weights at the nodes.

Context

Example of Neural Network:

A neural network can be used for a task like image classification. For instance:

- The input layer receives the pixel values of the image.
- Hidden layers transform these pixel values through a series of mathematical operations, learning important features such as edges, shapes, and textures.
- The output layer classifies the image into one of the predefined categories (e.g., "cat" or "dog").

Pros:

- Flexibility: Can model complex, non-linear relationships.
- Adaptability: Can be applied to a wide range of problems like image recognition, speech processing, and game playing.
- Automatic Feature Extraction: Neural networks, especially CNNs, can automatically learn important features from raw data without manual intervention.

Cons:

- Data-hungry: Neural networks typically require large datasets to perform well.
- Computationally Intensive: Training deep networks can require substantial computational resources.

- Black Box Nature: The internal decision-making process is often difficult to interpret, although research into interpretability is addressing this.

Optimisation Function

Optimization functions adjust the Model Parameters to minimize the Loss function, which measures how well the model performs. This is a fundamental step in training machine learning models.

General Optimization Process:

The Optimisation function (e.g., LBFGS, Newton-CG) iteratively updates the Model Parameters by:

1. Calculating the gradient of the loss function with respect to the parameters.
2. Updating the parameters in the direction of the negative gradient (as described in Gradient Descent).

This process is repeated until:

- The cost function converges (i.e., the change in the loss function becomes negligible), or
- The maximum number of iterations is reached.

See Optimisation techniques.

Outliers

Outliers are data points that differ significantly from other observations in the dataset. They can skew and mislead the training of machine learning models, especially those sensitive to the scale of data, such as Linear Regression. They can sway the generality of the model, skewing predictions and increasing the standard deviation.

Related Concepts:

- Handling outliers in similar to Handling Missing Data
- Methods for Handling Outliers
- Anomaly Detection

Why Removing Outliers May Improve Regression but Harm Classification

Impact on Regression Model:

Regression models, particularly linear regression, are sensitive to outliers because they attempt to minimize the sum of squared errors. By removing outliers, the model can better capture the underlying trend of the data, leading to improved performance metrics such as R-squared and reduced mean squared error.

Impact on Classification Models

- Class Boundary Distortion: Classification models, such as decision trees or support vector machines, rely on the distribution of data points to define class boundaries. Outliers can provide valuable information about the variability within classes.
- Loss of Information: Removing outliers may lead to the loss of important data points that could help in distinguishing between classes, potentially resulting in a less accurate model. For example, an outlier might represent a rare but important class that the model needs to learn from.

Overfitting

[!Summary]

Overfitting in machine learning occurs when a model captures not only the underlying patterns in the training data but also the noise, leading to poor performance on unseen data, and is unable to generalise.

Mathematically, overfitting results in a model with low bias but high variance, meaning it adapts too closely to the training data and fails to generalize well.

Key methods to address overfitting include Regularisation (such as L_1 and L_2 regularization), Cross Validation, and simpler models.

In statistical terms, it indicates a model with high complexity and too many parameters relative to the amount of training data, which results in $f(x)$ poorly representing the population distribution.

[!Breakdown]

Key Components:

- Regularization (Lasso: L_1, Ridge: L_2) to penalize model complexity.
- Cross Validation to ensure model generalization.
- Early stopping in training to avoid learning noise.
- Simplification of models to prevent fitting irrelevant patterns.

[!important]

- Overfitting indicates high variance in the model's performance, which can be identified by a significant drop in accuracy between training and test datasets.
- Regularization adds penalty terms to the cost function, reducing model complexity and mitigating overfitting.

[!attention]

- Overfitting is more common in models with high-dimensional datasets.
- Excessive model tuning (hyperparameter optimization) may inadvertently increase overfitting.

[!Follow up questions]

- How does the choice of regularization type (e.g., L_1 vs. L_2) affect model generalization in overfitting scenarios?
- What role does the size of the training dataset play in mitigating overfitting?

[!Related Topics]

- Cross Validation techniques (e.g., k-fold, Leave-One-Out cross-validation)
- Bias and varianceradeoff in machine learning models

Performance Drift

Not Data Drift

TL;DR. Data drift is a change in the input data. Concept drift is a change in input-output relationships. Both often happen simultaneously.

Performance drift refers to the gradual decline in a machine learning model's accuracy or effectiveness over time as the underlying data distribution changes.

This phenomenon occurs when the real-world data that the model is applied to differs from the data it was trained on. Mathematically, this is often represented by a shift in the joint distribution $P(X, Y)$ of the features X and target variable Y.

Performance drift can occur due to *concept drift* (when the relationship between inputs and outputs changes) or *covariate shift* (when the distribution of the inputs changes). The model's prediction error increases, leading to suboptimal decisions or predictions.

Key Components:

- **Concept drift**: Changes in the relationship between inputs and outputs, $P(Y|X)$.
- **Covariate shift**: Change in the input data distribution, $P(X)$.
- **Model monitoring** Data Observability|monitoring: Continuous assessment of a model's accuracy over time to detect drift.
- **Retraining**: Updating the model with new data to restore performance.

Important

- Performance drift results from data distribution shifts, leading to increased prediction errors.
- Monitoring and retraining are key strategies to address performance drift in real-world applications.
- A lack of continuous monitoring can result in undetected model performance degradation.
- Overfitting a model to the original data without considering future data can accelerate drift.

Example In a credit scoring model, performance drift may occur if consumer spending habits change due to an economic recession. The model trained on pre-recession data will perform poorly on post-recession data as the input patterns ($P(X)$) and the relationship between inputs and outputs ($P(Y|X)$) shift.

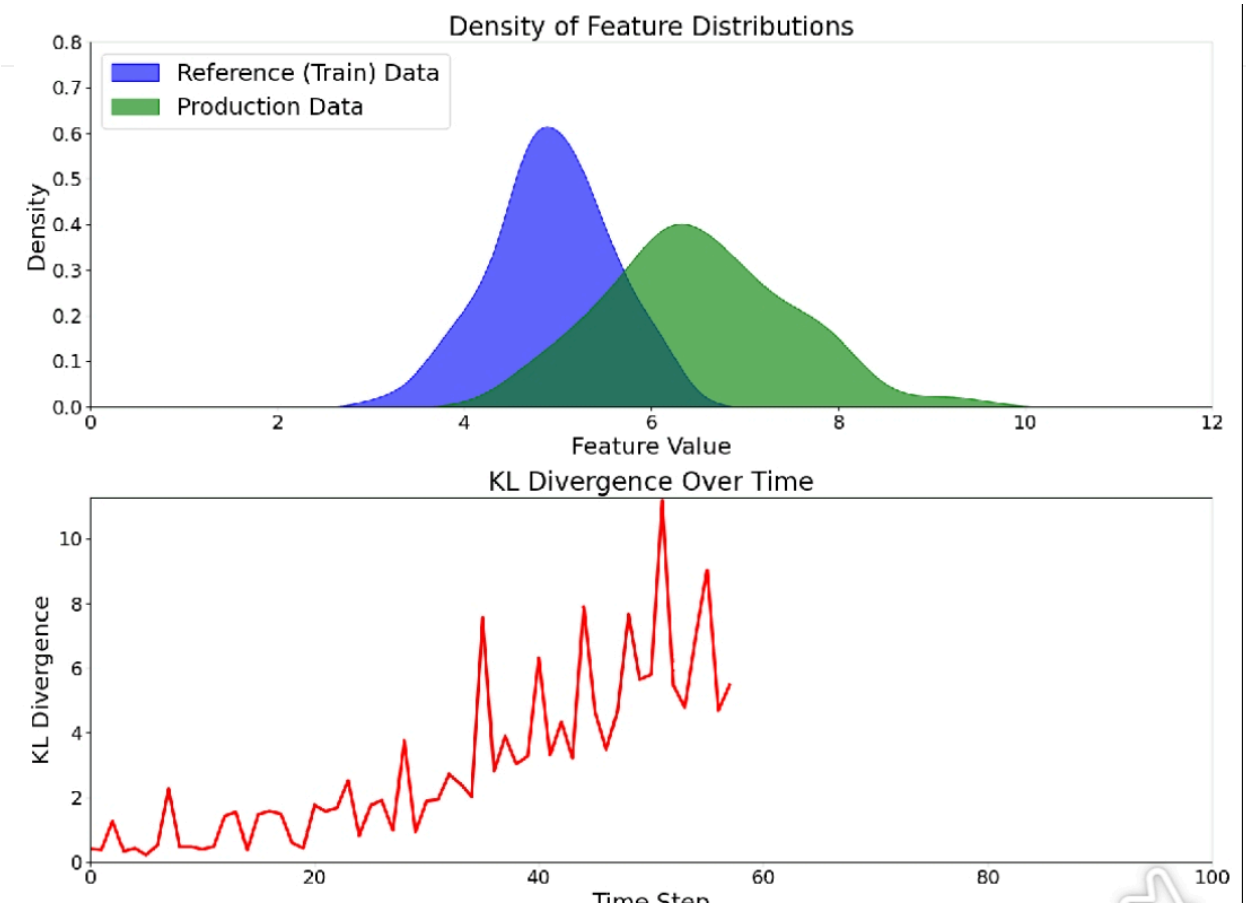
Questions

- How can adaptive learning techniques help mitigate the effects of performance drift?
- What statistical methods can be used to detect early signs of concept drift in production models?

Related Topics

- Model retraining strategies

Images



Precision Or Recall

Precision and Recall are two fundamental metrics used to evaluate the performance of a Classification model, particularly in binary classification tasks. They are related through a trade-off: improving one often comes at the expense of the other.

Key Differences:

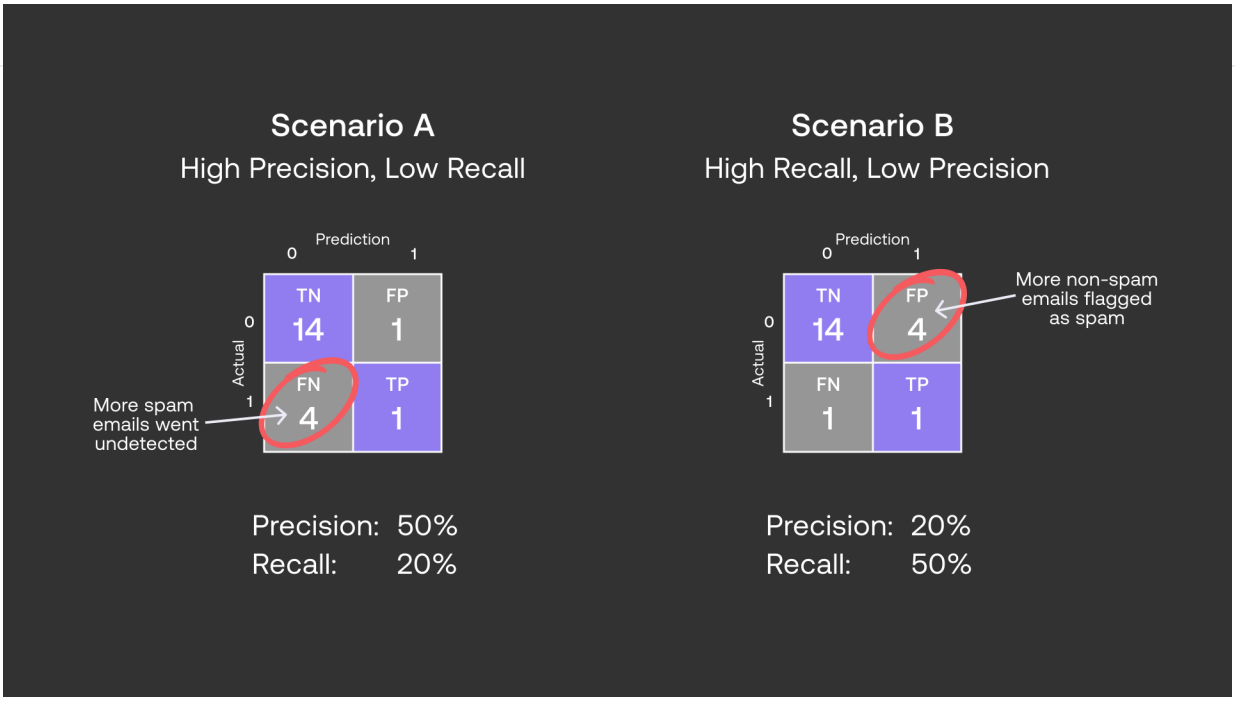
- Precision focuses on the quality of positive predictions.
- Recall focuses on the ability to identify all relevant positive instances

Trade-off:

- It is challenging to optimize both precision and recall simultaneously. Improving precision by reducing false positives may lead to an increase in false negatives, thereby reducing recall, and vice versa.
- However, it's important to balance recall with precision. A model with high recall might also have a higher rate of false positives (non-spam emails incorrectly marked as spam), which can lead to important emails being missed.

Task Dependency Example:

- The choice between prioritizing precision or recall is task-dependent. In a spam classification task, it might be more important to avoid moving important emails to the spam folder (high precision) than to occasionally allow spam emails into the inbox (lower recall). Thus, precision is prioritized over recall in this context.



Precision

Precision Score is a metric used to evaluate the Accuracy of a Classification model, specifically focusing on the positive class.

How many retrieved items are relevant?

This metric indicates the accuracy of positive predictions. The formula for precision is:

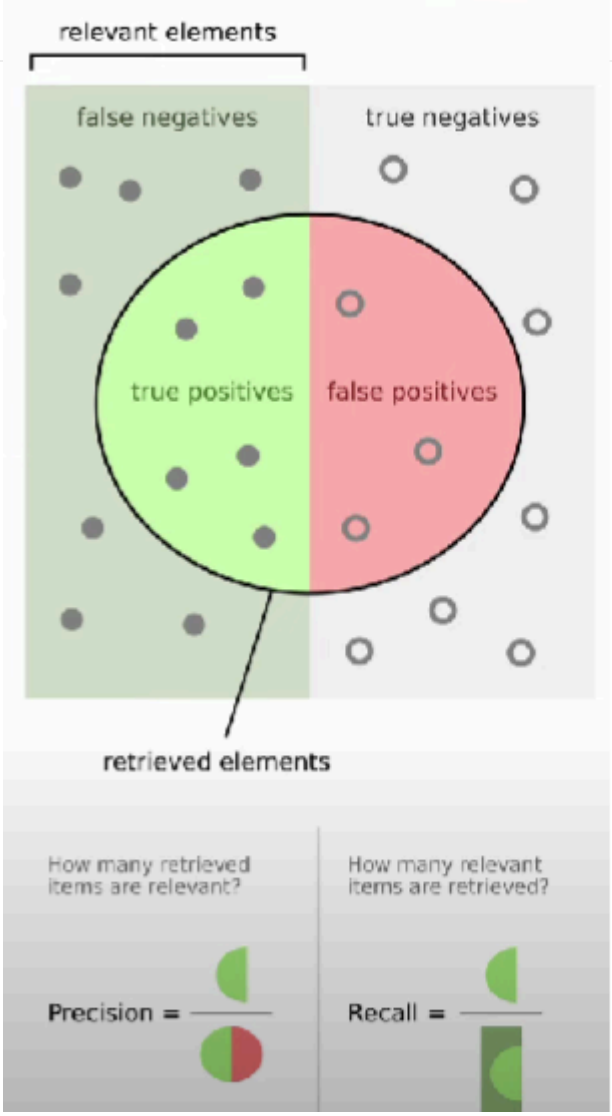
$$\text{Precision} = \frac{TP}{TP+FP}$$

where:

- **TP (True Positives):** The number of correctly predicted positive instances.
- **FP (False Positives):** The number of instances incorrectly predicted as positive.

Importance:

- Precision is crucial in scenarios where the **cost of false positives is high**, such as in spam detection or medical diagnosis. It helps in understanding how many of the predicted positive instances are actually positive.



Related Concepts

- Classification Report & Precision or Recall

Preprocessing

Data Preprocessing

Data Preprocessing refers to the overall process of cleaning and transforming raw data into a format that is suitable for analysis and modelling. This includes a variety of tasks, such as:

Useful:

- Data Cleansing
- Data Transformation

Others:

- Data Collection
- Data Reduction

End goal:

- EDA

Feature Preprocessing

Feature preprocessing refers to the process of transforming raw data into a clean data set for learning models, after Data Preprocessing. This step is crucial for improving model performance and ensuring accurate predictions

1. **Feature Scaling**: Normalizing or standardizing features to ensure they are on a similar scale. Normalization and Scaling: Adjusting the range of features, often using techniques like min-max scaling or z-score normalization, to ensure that all features contribute equally to the model.
2. Feature Selection: Identifying and retaining the most relevant features that contribute to the predictive power of the model, often using statistical tests or model-based approaches.
3. Dimensionality Reduction: Reducing the number of features while preserving important information, using techniques like Principal Component Analysis (PCA).
4. Feature Engineering: Creating new features from existing data to improve model performance, often based on domain knowledge.

Q Learning

Q-learning is a value-based, model-free Reinforcement learning algorithm where the agent learns the optimal policy by updating Q-values based on the rewards received. It is particularly useful in discrete environments like grids.

Uses a Q-Table which is populated by Q-values which are the maximum expected future reward for the given state and action. We improve the Q-Table in an iterative approach

Resources:

- Q-Learning Explained - Reinforcement Learning Tutorial

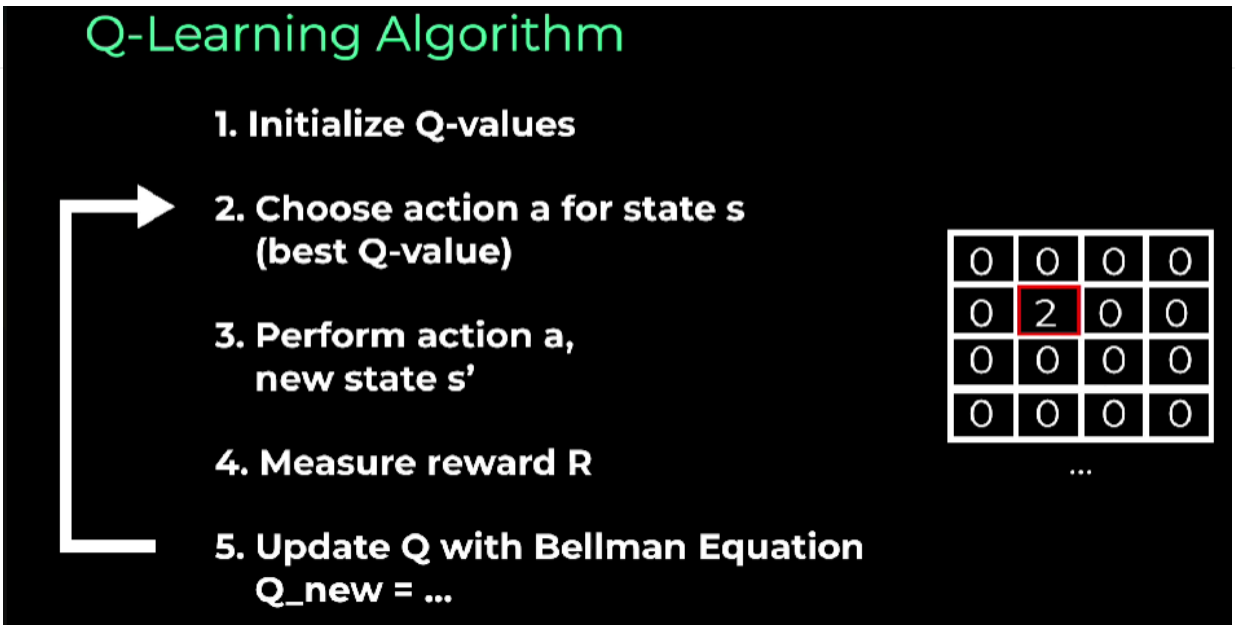
Q-learning update rule:

The left hand side gets updated (Bellman Equations)

$$Q_{new}(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t) \right]$$

Explanation:

- $Q(s_t, a_t)$: The Q-value of the current state s_t and action a_t.
- α: The learning rate, determining how much new information overrides old information.
- r_{t+1}: The reward received after taking action a_t from state s_t.
- γ: The discount factor, balancing immediate and future rewards.
- $\max_{a'} Q(s_{t+1}, a')$: The maximum Q-value for the next state s_{t+1} across all possible actions a'.



Notes:

- Q-learning is well-suited for environments where the state and action spaces are discrete and manageable in size.
- The algorithm is designed to converge to the optimal policy, even in non-deterministic environments, as long as each state-action pair is sufficiently explored.
- Exploration vs. Exploitation

Random Forests

A random forest is an Model Ensemble of Decision Trees. Take many decision trees decisions to get better result.

What is the Random Forest method;; an ensemble learning method based on constructing multiple decision trees during training and combining their predictions through averaging. Random Forests are flexibility, robustness, and ability to handle high-dimensional data, as well as their resistance to overfitting.

What is an issue with Random Forests;; susceptible to overfitting, especially when dealing with noisy or high-dimensional data. Proper tuning of hyperparameters like the number of trees and maximum depth is crucial to mitigate this.

Random forests combine multiple decision trees to improve accuracy and generalisation.

What is Random Forest, and how does it work?;; Random Forest is an method that can perform regression, classification, dimensionality reduction, and handle missing values. It builds multiple decision trees and combines their outputs. Each tree is grown using a subset of the data and features, and the final output is determined by aggregating the predictions of individual trees.

Remember that for a Random Forest, we randomly choose a subset of the features AND randomly choose a subset of the training examples to train each individual tree.

if n is the number of features, we will randomly select $\sqrt{n}$ of these features to train each individual tree.

- Note that you can modify this by setting the `max_features` parameter.

You can also speed up your training jobs with another parameter, `n_jobs`.

- Since the fitting of each tree is independent of each other, it is possible fit more than one tree in parallel.

- So setting `n_jobs` higher will increase how many CPU cores it will use. Note that the numbers very close to the maximum cores of your CPU may impact on the overall performance of your PC and even lead to freezes.
- Changing this parameter does not impact on the final result but can reduce the training time.

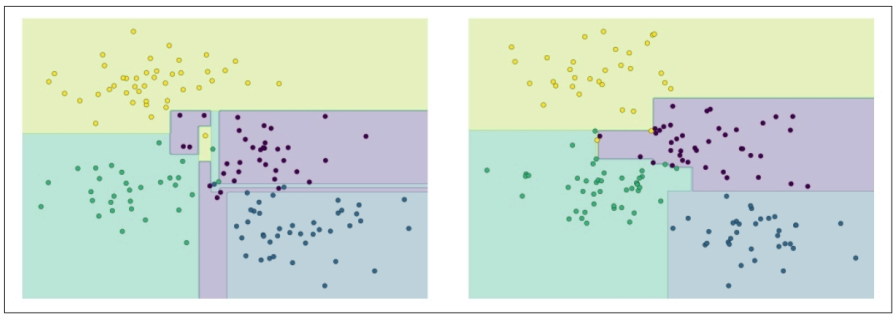
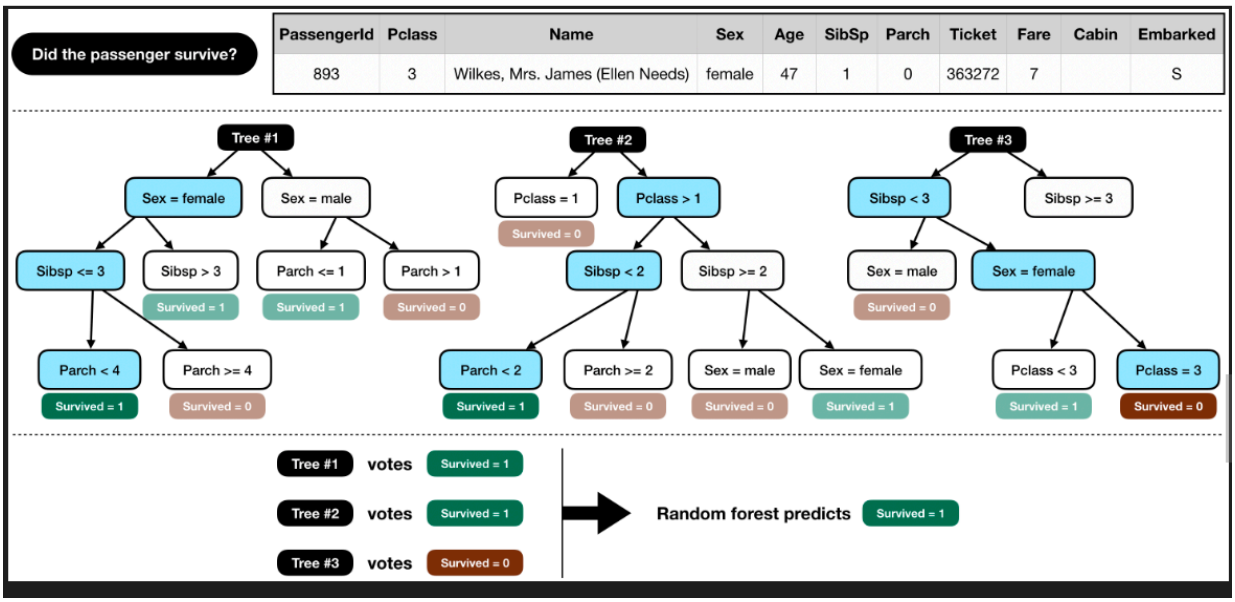


Figure 5-72. An example of two randomized decision trees

It is clear that in some places, the two trees produce consistent results (e.g., in the four corners), while in other places, the two trees give very different classifications (e.g., in the regions between any two clusters). The key observation is that the inconsistencies tend to happen where the classification is less certain, and thus by using information from *both* of these trees, we might come up with a better result!

If you are running this notebook live, the following function will allow you to interactively display the fits of trees trained on a random subset of the data (Figure 5-73):



What is an issue with Random Forests;; susceptible to overfitting, especially when dealing with noisy or high-dimensional data. Proper tuning of hyperparameters like the number of trees and maximum depth is crucial to mitigate this.

Decision Tree are not the best - need to make flexible for new data. THey work well with the data set they are defined on.

How to proceed with random forest: (build tree's randomly) i.e solve the issue with decision trees. Processis called Bagging 1) randomly select a dataset (bootstrap) 2) randomly select two (or multiple) features for each branch and proceed like in decision tree.

variety makes trees better.

To make a prediction , run data through trees in forest, and get prediction, conclude with majority prediciton.

How to know if random forest is good ?

Use data that was not in boot strap data set - measure the accuracy based on these classiifcations.

Refine the random forest by qweaking the Hyperparameter of number of features used per step.

Recall

Recall Score is a Evaluation Metrics used to evaluate the performance of a Classification model, focusing on the model's ability to **identify all relevant instances of the positive class**.

It answers the question: **How many relevant items are retrieved?**

High recall means that the model is effective at identifying most of the actual spam emails. This is useful in environments where missing a spam email could lead to security risks such as in corporate email systems.

The formula for recall is:

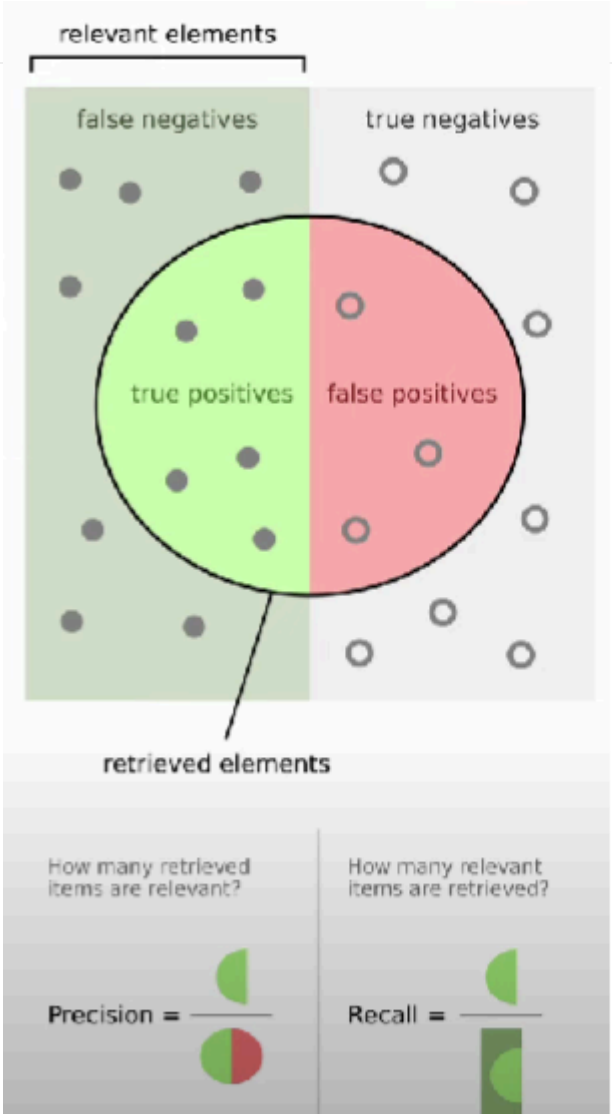
$$\text{Recall} = \frac{TP}{TP+FN}$$

Importance

- Recall is crucial in scenarios where **missing a positive instance is costly**, such as in disease screening or fraud detection.
- It helps in understanding how well the model captures all the actual positive instances.

Related Concepts

- **Sensitivity** (also known as recall or the true positive rate) measures the proportion of actual positives that are correctly identified by the model. It indicates how well the model is at identifying positive instances.



Recommender Systems

Crab on Python

A recommender system, or recommendation system, is a type of information filtering system that aims to predict the preferences or interests of users by analyzing their behavior and the behavior of similar users or items. These systems are widely used in various applications, such as e-commerce, streaming services, social media, and content platforms, to provide personalized recommendations to users.

Key Components of Recommender Systems:

1. **User Data**: Information about users, such as their preferences, ratings, purchase history, and interactions with items.
2. **Item Data**: Information about the items being recommended, which can include attributes, descriptions, and metadata.
3. **Recommendation Algorithms**: The methods used to generate recommendations. Common approaches include:

- **Collaborative Filtering**: This technique relies on the behavior and preferences of similar users. It can be user-based (finding similar users) or item-based (finding similar items).
- **Content-Based Filtering**: This approach recommends items based on the features of the items and the preferences of the user. For example, if a user likes action movies, the system will recommend other action movies based on their attributes.
- **Hybrid Methods**: Combining collaborative and content-based filtering to improve recommendation accuracy and overcome limitations of each method.

4. **Evaluation Metrics**: Metrics used to assess the performance of the recommender system, such as precision, recall, F1 score, and mean average precision.

Applications of Recommender Systems:

- **E-commerce**:
- **Streaming Services**:
- **Social Media**:
- **News and Content Platforms**:

Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are a type of neural network designed to process sequential data by maintaining a memory of previous inputs through hidden states. This makes them suitable for tasks where the order of data is needed, such as:

- Time Series prediction,
- speech recognition,
- and NLP|natural language processing (NLP).

RNNs have loops in their architecture, allowing information to persist across sequence steps. However, they face challenges with long sequences due to vanishing and exploding gradients problem. To address these issues, variants like Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) have been developed.

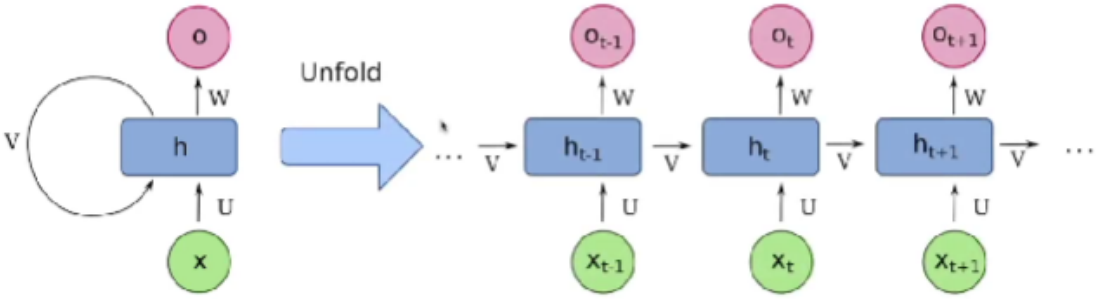
Resources:

Video link https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks

Key Concepts of RNNs

Sequential Data Handling

- RNNs maintain a hidden state that acts as memory, enabling them to model temporal dependencies. This is essential for tasks where the current output depends on both current and previous inputs.
- At each time step, RNNs process an input, combine it with the previous hidden state, and produce an output along with an updated hidden state.
- The hidden state carries forward information influenced by all previous inputs, theoretically allowing RNNs to remember long-term dependencies.



Backpropagation Through Time (BPTT)

- RNNs are trained using BPTT, a variant of backpropagation. The network unrolls over time, treating each time step as a layer in a deep network.
- Gradients are computed for each time step, and weights are updated based on cumulative error across the sequence. This allows learning of long-term dependencies but can lead to vanishing and exploding gradients in long sequences.

Variants of RNNs

- **LSTM**: Introduces gates (input, forget, output) to control information flow, addressing vanishing gradients and improving long-sequence handling.
- **GRU**: A simpler variant of LSTM with fewer parameters, offering efficiency and ease of training while maintaining performance on sequence tasks.

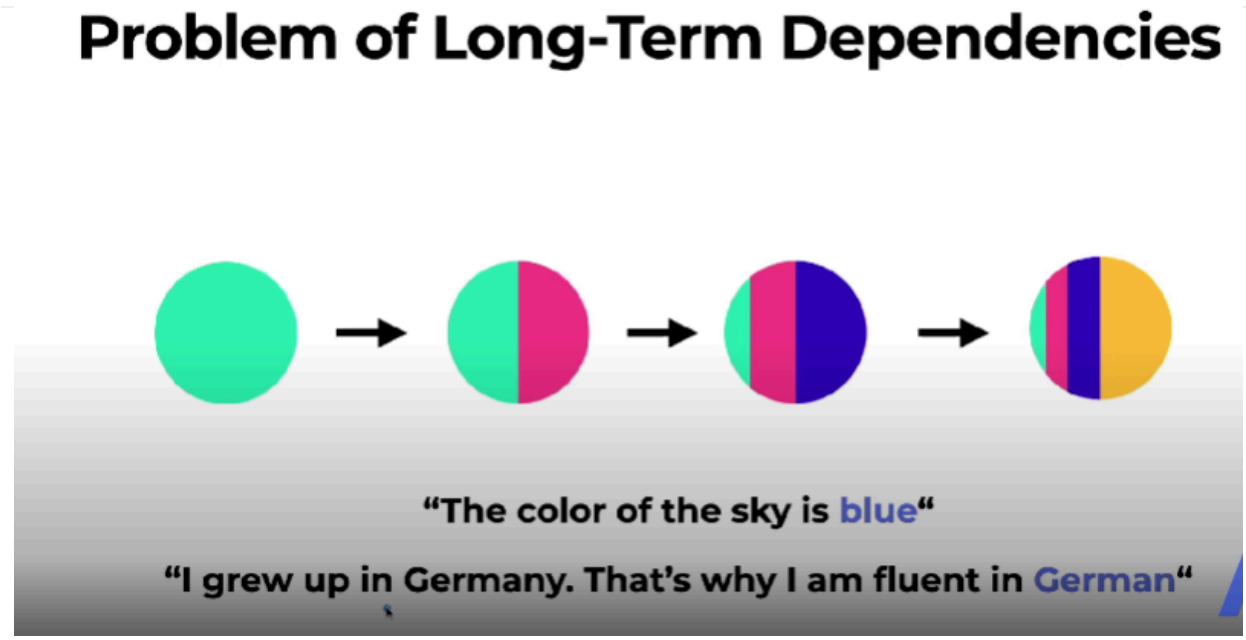
Example Code (RNN in Python with PyTorch)

See RNN_Pytorch.py

Problem of Long Term Dependencies

The more time steps we include the less data we keep from the past.

Solutions: LSTM and GRU: use gates: but are costly in computation.



Other areas

Use of RNNs in energy sector

RNNS and Transformer|Transformers

Why have Transformer|Transformer's have replaced traditional RNN.

Inductive Reasoning, Memories and Attention.

How to address the limitations of vanilla recurrent networks.

Issues:

- RNNs are not inductive: They memorize sequences extremely well, but don't generalise well.
- They couple their representation size to the amount of computation per step.

Regression Metrics

These metrics provide various ways to evaluate the performance of regression models.

Evaluating Regression Models

These metric provide:

1. **Comprehensive Evaluation**: Each metric provides a different perspective on model performance. For example, while MSE and RMSE give insights into the average error magnitude, MAE provides a straightforward average error measure, and R-squared indicates how well the model explains the variance in the data.
2. **Sensitivity to standardised/Outliers**: Metrics like MSE and RMSE are sensitive to outliers due to the squaring of errors, which can be useful if you want to emphasize larger errors. In contrast, MAE and Median Absolute Error are more robust to outliers.

3. **Interpretability**: RMSE is in the same units as the target variable, making it easier to interpret in the context of the data. This can be particularly useful for stakeholders who need to understand the model's performance in practical terms.
4. **Model Comparison**: These metrics allow you to compare different models or configurations to determine which one performs best on your data.
5. **Variance Explanation**: R-squared and Explained Variance Score provide insights into how much of the variability in the target variable is captured by the model, which is crucial for understanding the model's effectiveness.

Common Regression Metrics

Mean Absolute Error (MAE):

- Definition: MAE measures the average absolute differences between predicted and actual values.
- Interpretation: Lower values indicate better model performance, as it reflects fewer errors in predictions.
- Formula: $\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$
- Where:
  - n = number of observations
  - y_i = actual value
  - $\hat{y}_i$ = predicted value

Mean Squared Error (MSE):

- Definition: MSE calculates the average of the squares of the errors (the differences between predicted and actual values).
- Interpretation: Like MAE, lower values are better. However, MSE is more sensitive to outliers due to the squaring of errors, which can disproportionately affect the metric. Greater error values are exaggerated.
- Formula: $\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$
- Where:
  - n = number of observations
  - y_i = actual value
  - $\hat{y}_i$ = predicted value

Root Mean Squared Error (RMSE):

- Definition: RMSE is the square root of MSE, providing an error metric in the same units as the target variable.
- Interpretation: Lower RMSE values indicate better model performance, and it also emphasizes larger errors due to the squaring process. Easier to interpreted (interpretability), back to the same scale as the input.
- Formula: $\text{RMSE} = \sqrt{\text{MSE}} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$

R squared

Adjusted R squared

Median Absolute Error:

- Definition: This metric measures the median of the absolute errors between predicted and actual values.
- Interpretation: It provides a robust measure of prediction accuracy, especially in the presence of standardised/Outliers.
- Formula: $\text{MedAE} = \text{median}(|y_i - \hat{y}_i|)$

Explained Variance Score:

- Definition: This metric measures the proportion of variance in the target variable that is predictable from the features.
- Interpretation: Higher values indicate that the model explains a greater proportion of the Variance in the target variable.
- Formula: $\text{Explained Variance} = 1 - \frac{\text{Var}(y-\hat{y})}{\text{Var}(y)}$
- Where:
  - $\text{Var}(y)$ = variance of the actual values
  - $\text{Var}(y - \hat{y})$ = variance of the prediction errors

Implementation

See Regression_Metrics.py

Regression

[!Summary]

Regression analysis is a statistical method used to predict a continuous variable based on one or more predictor variables. The most common form, Linear Regression, assumes a linear relationship between the dependent variable y and independent variables x_1, x_2, x_n. The goal is to minimize the residual sum of squares (RSS) between observed and predicted values. Other forms, such as Logistic Regression, handle classification problems.

Regression models can incorporate techniques like regularization (L_1, L_2) to improve performance and prevent overfitting, especially with high-dimensional data. Advanced methods like Polynomial Regression address non-linearity, while generalized linear models (GLMs) extend regression to non-normal response variables.

Regressor: This is a type of model used for regression tasks, where the goal is to predict continuous values. For example, a regressor might be used to predict the price of a house based on its features, or to forecast future sales figures.

[!Breakdown]

Key Components:

- Linear Regression: Predicts $y = \beta_0 + \beta_1 x_1 + \beta_n x_n + \epsilon$, where ϵ is the error term.
- Regularization: Adds L_1 (Lasso) or L_2 (Ridge) penalty to prevent overfitting in high-dimensional data.
- Feature transformation: Polynomial Regression and logarithmic transformations adjust for non-linearity in data.
- Regression is a type of Supervised Learning.

[!important]

- R^2 is a key metric, showing how much of the variance in y is explained by x.
- Multicollinearity can inflate variances of coefficient estimates, harming model reliability.

[!attention]

- Regression assumes linearity, so improper application to non-linear data can lead to biased predictions.
- Overfitting can occur with too many predictors, especially in small datasets.

[!Example]

In predicting insurance claims, a linear regression model could take input variables like age and driving history to estimate the expected claim amount. A transformation, such as logarithmic scaling, could address any non-linear patterns between variables.

[!Follow up questions]

- How can Polynomial Regression improve predictions in non-linear datasets?
- What are the benefits of combining Linear Regression with Feature Engineering for complex datasets?

Regularisation Of Tree Based Models

Tree models, such as Random Forests and Gradient Boosting, can also be regularized, although they don't use L1 or L2 regularization directly. Instead, they are regularized through hyperparameters like max depth, min samples split, and learning rate to control the complexity of the trees.

In tree-based models, regularization is not applied in the same way as it is in linear models (i.e., using L1 or L2 penalties).

In tree models, Regularisation is done by controlling the growth of the trees using hyperparameters like

- `max_depth` ,
- `min_samples_split` ,
- `min_samples_leaf` .

These hyperparameters restrict the growth of the tree, preventing it from becoming too complex.

For Model Ensemble methods like Random Forests and Gradient Boosting, additional regularization techniques like

- subsampling,
- bootstrap sampling,
- and learning rate control

to help prevent overfitting. These techniques effectively restrict the model complexity, leading to better generalization .

Below are the common regularization techniques used in tree models such as Decision Tree, Random Forests.

Regularization in Different Tree Models

- Decision Trees: Prone to overfitting when not regularized, since they tend to grow large and complex trees. Regularization through pruning, limiting tree depth, and controlling minimum samples per split or leaf is critical.
- Random Forests: Regularization is mainly achieved through the use of multiple decision trees, random feature selection (`max_features`), and bootstrapping (`bootstrap`). Each tree learns a different part of the data, which reduces overfitting.
- Gradient Boosting Models (GBMs): Regularized by tuning the `learning_rate` , `subsample` , and controlling the tree depth and other tree-based hyperparameters like `min_samples_split` . The slower learning process with a smaller learning rate combined with these hyperparameters helps prevent overfitting.

Regularization Techniques for Tree Models

- Limiting Tree Depth:
  - Max Depth (`max_depth`): This parameter restricts the maximum depth of the tree. Trees that are too deep can model complex patterns, but they are prone to overfitting. By limiting the depth, you constrain the tree's ability to learn highly specific patterns in the training data.
  - Example: In scikit-learn, setting `max_depth` for a Decision Tree or Random Forest.

```
from sklearn.tree import DecisionTreeClassifier

# Limit tree depth to regularize the model
model = DecisionTreeClassifier(max_depth=5)
model.fit(X_train, y_train)
```

- Minimum Samples for Splitting:
  - Min Samples Split (`min_samples_split`): This parameter specifies the minimum number of samples required to split an internal node. Increasing this value makes the tree more conservative and prevents it from splitting when there are too few samples, thus controlling its complexity.
  - This helps avoid creating splits based on noise, which could lead to overfitting.

```
model = DecisionTreeClassifier(min_samples_split=10)
model.fit(X_train, y_train)
```

- Minimum Samples per Leaf:
  - Min Samples Leaf (`min_samples_leaf`): This parameter sets the minimum number of samples a node must have after a split to be a leaf. Higher values result in fewer splits, producing simpler trees that are less likely to overfit.
  - This also encourages broader splits that require more data, reducing the sensitivity to outliers.

```
model = DecisionTreeClassifier(min_samples_leaf=4)
model.fit(X_train, y_train)
```

- Max Number of Features:
  - Max Features (`max_features`): This controls the number of features to consider when looking for the best split. Reducing the number of features makes the model less likely to overfit, as it limits the search space for splits. For Random Forests, this also introduces randomness that can improve generalization.

```
from sklearn.ensemble import RandomForestClassifier

model = RandomForestClassifier(max_features='sqrt')  # Uses sqrt of total features
model.fit(X_train, y_train)
```

- Max Leaf Nodes:
  - Max Leaf Nodes (`max_leaf_nodes`): This parameter limits the total number of leaf nodes the tree can have. Fewer leaf nodes result in simpler trees that are less likely to overfit the training data.

```
model = DecisionTreeClassifier(max_leaf_nodes=10)
model.fit(X_train, y_train)
```

- Subsampling (for Ensemble Methods like Random Forests and Gradient Boosting):
  - Bootstrap Sampling (`bootstrap`): For Random Forests, regularization is achieved through bootstrapping (random sampling with replacement) during training. This introduces variability and helps prevent overfitting.

- Subsample (`subsample`): For Gradient Boosting, the `subsample` parameter controls the fraction of the training data used for fitting each individual tree. A value less than 1 introduces randomness and reduces the chance of overfitting, similar to how dropout works in neural networks.

```
from sklearn.ensemble import GradientBoostingClassifier


model = GradientBoostingClassifier(subsample=0.8)  # Use 80% of data for each tree
model.fit(X_train, y_train)
```

- Learning Rate (for Gradient Boosting Models):
  - Learning Rate (`learning_rate`): This parameter controls how much each tree contributes to the overall model in Gradient Boosting. A lower learning rate slows down the learning process, requiring more trees but helping to avoid overfitting by making small adjustments at each step.

```
model = GradientBoostingClassifier(learning_rate=0.1)
model.fit(X_train, y_train)
```

- Pruning:
  - For Decision Trees, pruning is a post-processing regularization technique where branches that contribute little to the overall performance of the model are removed. This prevents the tree from learning noise in the data.
  - In scikit-learn, Cost Complexity Pruning (`ccp_alpha`) is used for pruning. A larger value of `ccp_alpha` leads to more aggressive pruning, simplifying the tree.

```
model = DecisionTreeClassifier(ccp_alpha=0.01)
model.fit(X_train, y_train)
```

Regularisation

Regularization is a technique in machine learning that reduces the risk of overfitting by adding a penalty to the Loss function during model training. This penalty term restricts the magnitude of the model's parameters, thereby controlling the complexity of the model. It is especially useful in linear models but can also be applied to more complex models like neural networks.

Key Concepts

- L_1 **Regularization (Lasso):** Adds the absolute value of the coefficients to the loss function, encouraging sparsity by driving some coefficients to zero, effectively selecting a subset of features.
- L_2 **Regularization (Ridge):** Adds the square of the coefficients to the loss function, shrinking them toward zero. It encourages smaller coefficients but does not push them exactly to zero, helping reduce overfitting by penalizing large weights.
- **Elastic Net:** Combines both Lasso and Ridge regularization.

Benefits

- **Prevents Overfitting:** Regularization adds a penalty term to the loss function to avoid overfitting.

- **Feature Sparsity:** L_1 encourages feature sparsity, while L_2 reduces coefficient magnitudes.
- **Enhanced Generalization:** Dropout enhances generalization by preventing unit co-adaptation in neural networks.

Considerations

- **Underfitting Risk:** Over-penalizing parameters can lead to underfitting, where the model becomes too simplistic.
- **Tuning λ:** Choosing the right penalty term (i.e., λ) is crucial for balancing bias and variance.

When and why not to us regularisation

Questions

- How does the balance between L_1 and L_2 regularization impact model performance in large feature spaces?
- What are the best practices for tuning the λ parameter in regularization? Model Parameters Tuning.

Example

Consider a linear regression model with L_2 regularization (Ridge). The loss function would be:

$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{j=1}^{n} \theta_j^2$

Here, λ controls the strength of the regularization. Higher λ values shrink the coefficients more.

Related Topics

- Feature Selection: L1 regularization can zero out irrelevant features, improving model interpretability and reducing computational costs.
- Model Selection techniques for high-dimensional data.

Applications

Regularization is widely used in linear models but is also applied in other machine learning models, particularly those prone to overfitting:

- Neural network
- Regularisation of Tree based models

Implementation

In ML_Tools see: Regularisation.py

Roc (Receiver Operating Characteristic)

ROC (Receiver Operating Characteristic) is a graphical representation of a classifier's performance across different thresholds, showing the trade-off between sensitivity (true positive rate) and specificity (1 - false positive rate).

A graphical plot that illustrates the diagnostic ability of a binary classifier system as its discrimination threshold is varied.

It plots the true positive rate (TPR) against the false positive rate (FPR) at different threshold settings.

In ML_Tools see: ROC_Curve.py

Why Use Predicted Probabilities?

In ROC analysis, predicted probabilities (`y_probs`) are used instead of predicted classes (`y_pred`) because the ROC curve evaluates the model's performance across different threshold levels. Probabilities allow you to adjust the threshold to see how it affects sensitivity and specificity.

Threshold Level

The threshold level is the probability value above which an instance is classified as the positive class. Adjusting the threshold affects Recall and Specificity

- Lower Threshold: Increases sensitivity but may decrease specificity.
- Higher Threshold: Increases specificity but may decrease sensitivity.

Example Code

```
from sklearn.metrics import roc_curve, RocCurveDisplay
import matplotlib.pyplot as plt

# Actual and predicted values
y_act = [1, 0, 1, 1, 0]
y_pred = [1, 1, 0, 1, 0]

# Compute ROC curve
fpr, tpr, thresholds = roc_curve(y_act, y_pred)

# Display ROC curve
roc_display = RocCurveDisplay(fpr=fpr, tpr=tpr).plot()
plt.show()
```

Logistic Regression Example

```
from sklearn.linear_model import LogisticRegression

# Train a logistic regression model
logreg = LogisticRegression()
logreg.fit(X_train, y_train)

# Predict probabilities for the positive class
y_probs = logreg.predict_proba(X_test)[:, 1]

# Compute ROC curve and AUC
fpr, tpr, thresholds = roc_curve(y_test, y_probs)
roc_auc = roc_auc_score(y_test, y_probs)

# Plot ROC curve
plt.figure(figsize=(10, 6))
plt.plot(fpr, tpr, color='blue', lw=2, label=f'ROC Curve (AUC = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='red', linestyle='--', lw=2, label='Random Guessing')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.xlabel('False Positive Rate (FPR)')
plt.ylabel('True Positive Rate (TPR)')
plt.legend()
plt.show()
```

Specificity

Specificity, also known as the true negative rate, measures the proportion of actual negatives that are correctly identified by the model. It indicates how well the model is at identifying negative instances. Formula:

$\text{Specificity} = \frac{TN}{TN+FP}$ Importance

- Specificity is crucial in scenarios where it is important to correctly identify negative instances, such as in medical testing where a false positive could lead to unnecessary treatment.

Support Vector Machines

Support Vector Machines (SVM) are a type of supervised learning algorithm primarily used for classification tasks, though they can also be adapted for regression. The main idea is to find an optimal hyperplane that divides data into different classes by maximizing the margin between them. The support vectors are the data points closest to the hyperplane, influencing its position and orientation.

Key Features

- Hyperplane: Finds a hyperplane that maximizes the margin between classes.
- High-Dimensional Spaces: Robust in high-dimensional spaces, such as image and text classification.

Advantages

- Highly effective for high-dimensional data (datasets with many features).

- Useful for classification tasks where a clear margin of separation exists between classes.

Disadvantages

- Can be computationally expensive for large datasets and sensitive to the choice of hyperparameters.
- Performance is highly dependent on the Kernelling choice, requiring careful tuning.

How SVM Works

In ML_Tools see: SVM_Example.py

1. Initial Space: Start in the low-dimensional space, where the data may not be linearly separable.
2. Kernel Function: Use a Kernelling function to move the data into a higher dimension where separation is clearer.
3. Hyperplane Placement: Place hyperplanes (decision boundaries) between the data clusters to classify the data.

Margins

- Outliers and Soft Margins: SVM allows for some miscalculations or errors in classification to handle outliers. This is part of the Bias and variance tradeoff, where the model is allowed to make a few mistakes to improve generalization.
- Soft Margins: Allow some data points to be within the margin or even on the wrong side of the hyperplane, enabling SVM to handle imperfect separations.

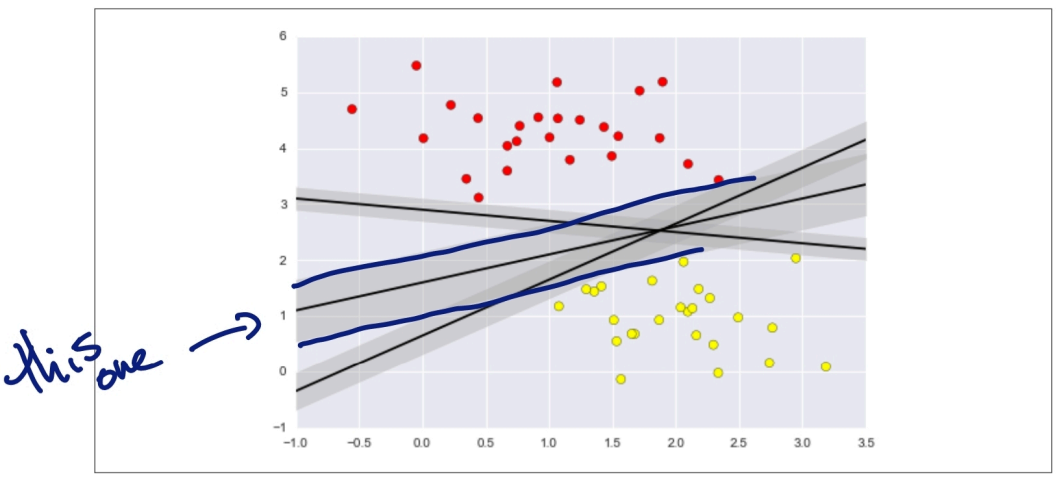


Figure 5-55. Visualization of "margins" within discriminative classifiers

In support vector machines, the line that maximizes this margin is the one we will choose as the optimal model. Support vector machines are an example of such a *maximum margin* estimator.

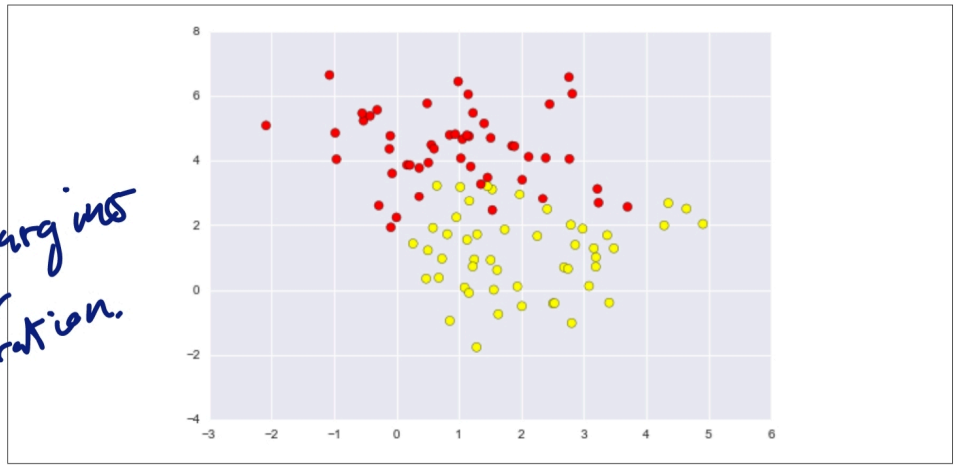


Figure 5-62. Data with some level of overlap

Tensorflow

Open sourced by Google Based on a dataflow graph

Text summarization with TensorFlow

Open-source library for numerical computation and large-scale Machine Learning, focusing on static dataflow graphs.

Get same code use of tensorflow example:

Basic example is

Handwritten Digit Classification

Pytorch vs Tensorflow

Transfer Learning

Transfer learning is a technique in machine learning that leverages knowledge gained from one setting (source domain) to improve performance on a different but related setting (target domain).

The core idea is to train a model on a large dataset in the source domain, learning rich feature representations that capture general patterns and relationships in the data.== These learned representations can then be transferred to the target domain, where they can be fine-tuned with a smaller dataset to achieve good performance on the target task.

Transfer learning makes sense when:

- It makes sense to do when there is lots of examples for basic layers and training, and there are few of the specialised data set.
- Task A and B have the same input x.
- You have a lot more data for Task A than Task B.
- Low level features from A could be helpful for learning B.
- When labelled data is scarce.

We can use pretrained models (i.e. from Hugging Face, Keras applications, PyTorch pretrained models, model zoo).

Examples of Transfer Learning

- **Image Recognition:** A model trained on a large dataset of labelled images (e.g., ImageNet) can learn features like edges, shapes, and textures that are useful for recognising a wide variety of objects. These features can then be transferred to a different image recognition task with a smaller dataset, such as classifying medical images or identifying different species of plants.
    - Pretraining - training on image recognition. Fine-training - retraining on radiology.
- **Natural Language Processing:** A language model trained on a massive text corpus can learn word embeddings that capture semantic relationships between words. These embeddings can then be transferred to tasks like sentiment analysis or machine translation, where they can improve performance, even with limited labelled data in the target language.

Types of Transfer Learning

- **Unsupervised Pretraining for Supervised Learning:** The sources describe how unsupervised pretraining with models like denoising autoencoders can be used to learn useful representations that can be transferred to supervised learning tasks.
- **Cross-Domain Transfer Learning:** This involves transferring knowledge between domains with different input distributions but the same task.
- **Performance Drift:** This is a form of transfer learning where the data distribution changes gradually over time. The model needs to adapt to these changes to maintain good performance.

Benefits of Transfer Learning

- **Improved Generalisation:** By leveraging knowledge from a larger dataset, transfer learning can help models generalise better to new data, especially when the target dataset is small.
- **Reduced Data Requirements:** Transfer learning can significantly reduce the amount of labelled data needed to train a model in the target domain. This is particularly beneficial for tasks where labelled data is expensive or time-consuming to obtain.
- **Faster Training:** Fine-tuning a pretrained model on a smaller dataset is typically faster than training a model from scratch.

Follow up questions

- Why might fine-tuning a pre-trained model like GPT yield better results than training from scratch

Practical Implementation

In ML_Tools see: transfer_learning.py

Links

- Video Overview
- Deep Learning Video

Transformer

A transformer in machine learning (ML) refers to a deep learning model architecture designed to process sequential data, such as natural language processing (NLP). It was introduced in the paper "standardised/Attention Is All You Need" and has since become a cornerstone in NLP tasks.

Transformers excel at handling sequence-based data and are particularly known for their self-attention mechanisms Attention mechanism, which allow them to process long-range dependencies in data.

Key Concepts of a Transformer

1. **Architecture Overview**:
    - A transformer model consists of an encoder and a decoder, although some models use only the **encoder** (like BERT only consists of encoders) or only the **decoder** (like GPT3). Each of these components is made up of layers that include mechanisms for attention and Feed Forward Neural Network.
    - Encoder learns the context, decoder does the task.
2. **Self-Attention Mechanism**:
    - The core innovation of transformers is the self-attention mechanism, which allows the model to weigh the importance of different words in a sentence relative to each other. This is crucial for understanding context and relationships in language. See [Attention mechanism].
    - **Scaled Dot-Product Attention**: For each word in a sentence, the model computes attention scores with every other word. These scores are used to create a weighted representation of the input, emphasizing relevant words and de-emphasizing less relevant ones.
3. Multi-head attention
    - Instead of having a single attention mechanism, transformers use multiple attention heads. Each head learns different aspects of the relationships between words, allowing the model to capture various linguistic features.
4. **Positional Encoding**:
    - Since transformers do not inherently understand the order of words (unlike Recurrent Neural Networks|RNNs), they use positional encoding to inject information about the position of each word in the sequence.
5. **Feed-Forward Neural Network**:
    - After the attention mechanism, the output is passed through a feed-forward neural network, which is applied independently to each position.
6. **Layer Normalization and Residual Connections**:
    - Transformers use layer normalization and residual connections to stabilize training and help with gradient flow, making it easier to train deep networks.
7. **Training and Applications**:
    - Transformers are trained on large corpora of text data using Unsupervised Learning|unsupervised or semi-supervised learning techniques. They are used for a variety of NLP tasks, including translation, summarization, and question answering.

Additional Concepts

- **Encoder-Decoder Structure**:
    - The encoder processes the input sequence to build a representation, while the decoder takes this representation and generates the output sequence. This setup is particularly useful for tasks like translation.
- **Parallelization**:
    - Unlike Recurrent Neural Networks (Recurrent Neural Networks), transformers do **not require sequential processing,** making them more efficient, especially when training large datasets.

Follow-up questions:

Transformers Vs Rnns

Transformer|Transformers and Recurrent Neural Networks (Recurrent Neural Networks) are both deep learning architectures used for processing sequential data, but they differ significantly in structure, operation, and performance.

While RNNs have been essential for sequence modeling, transformers have become the dominant architecture in ML due to their ability to handle large-scale data and long-range dependencies more efficiently.

RNNs still have use cases, especially for tasks where memory constraints are critical or for smaller datasets, but transformers are the go-to solution for most modern ML applications.

Summary Table:

Aspect	RNNs	Transformers
Architecture	Sequential (step-by-step)	Parallel (process all at once)
Attention	Implicit through hidden states	Explicit via self-attention
Parallelization	Not parallelizable	Fully parallelizable
Handling Long Sequences	Struggles with long dependencies	Excellent with long dependencies
Efficiency	Slower training	Faster due to parallelization
Scalability	Poor scalability to long sequences	Scalable but memory-intensive
Use Cases	Time-series, small datasets	NLP, large datasets, vision

1. Architecture

- **RNNs**:
  - RNNs process data sequentially, one time step at a time. They maintain a hidden state that is updated as the model processes each token in the sequence, making them suitable for time-dependent tasks.
  - Common variants include **LSTM (Long Short-Term Memory)** and **GRU (Gated Recurrent Units)**, which are designed to capture long-term dependencies more effectively.
- **Transformer**:
  - Transformers do not process data sequentially. Instead, they process the entire sequence in parallel, allowing them to model relationships between tokens regardless of their position. This is achieved through the **self-attention mechanism**.
  - Transformers include **positional encodings** to account for the order of tokens, since their architecture doesn't have an inherent understanding of sequence order.

2. Processing Mechanism

- **RNNs**:
  - RNNs depend on the previous hidden state to process the next token, which means they inherently process information sequentially.

- The hidden state is updated at each time step, which can lead to issues like **vanishing and exploding gradients problem**, especially in long sequences, making it difficult for RNNs to capture long-range dependencies.

- **Transformers**:
  - Transformers use Attention mechanism to allow each token to interact directly with every other token in the sequence. This allows transformers to capture long-range dependencies more effectively and efficiently.
  - The self-attention mechanism enables **parallelization** of the computation for all tokens in the sequence, which speeds up training and inference.

3. Parallelization and Efficiency

- **RNNs**:
  - Since RNNs must process sequences one step at a time, they cannot be easily parallelized. This makes them less efficient, especially for long sequences.
  - RNNs are also slower to train because of this sequential dependency.
- **Transformers**:
  - Transformers can process entire sequences at once, making it easier to parallelize computation, especially on GPUs. This leads to much faster training times compared to RNNs.
  - This parallelization is a major reason transformers have become the preferred model in large-scale tasks.

4. Handling Long-Term Dependencies

- **RNNs**:
  - RNNs often struggle with capturing long-term dependencies because the information must be passed through multiple time steps, which can lead to forgetting or corruption of information over long sequences.
  - LSTMs and GRUs were developed to mitigate this problem, but they are still not as effective as transformers for capturing long-range relationships.
- **Transformers**:
  - The self-attention mechanism in transformers allows the model to directly connect tokens from distant parts of the sequence. This makes transformers much better at modeling long-range dependencies.
  - Transformers can also model relationships across sequences regardless of their length, leading to better performance on tasks requiring a global understanding of the data.

5. Memory and Scalability

- **RNNs**:
  - RNNs are relatively more memory-efficient for shorter sequences but become inefficient for longer ones due to their sequential nature and the need to store hidden states at each time step.
  - They also scale poorly to long sequences or large datasets because of the need to compute one step at a time.
- **Transformers**:
  - Transformers, while faster, require more memory due to the computation of attention matrices, which scale quadratically with the sequence length. This can be a bottleneck for very long sequences or resource-constrained environments.
  - New transformer variants (e.g., **Longformer** or **Reformer**) have been introduced to improve memory efficiency for longer sequences.

6. Use Cases

- **RNNs**:
  - Traditionally used for time-series data, **speech recognition**, and **sequence generation** tasks.
  - They are useful when the order of data is crucial and when handling smaller datasets or shorter sequences.

7. Performance

- **RNNs**:
  - While effective in small-scale, low-latency tasks, RNNs often perform worse than transformers on complex tasks that involve large-scale data or long-range dependencies.
- **Transformers**:
  - Transformers have significantly outperformed RNNs in most tasks requiring sequential data processing, particularly in NLP. Pre-trained models like **BERT**, **GPT**, and **T5** are based on the transformer architecture and have set state-of-the-art results in many benchmarks.

Transformer-based models like BERT and GPT outperform traditional RNNs in NLP tasks for several key reasons:

1. **Parallelization**: Unlike Recurrent Neural Networks|RNNs, which process sequences sequentially (one time step at a time), transformers can process entire sequences in parallel. This significantly speeds up training and allows for more efficient use of computational resources.
2. **Self-Attention Mechanism**: Transformers utilize a self-attention mechanism that enables them to weigh the importance of different words in a sentence relative to each other. This allows the model to capture long-range dependencies and relationships between words more effectively than RNNs, which often struggle with long-term dependencies due to their sequential nature.
3. **Handling Long Sequences**: RNNs, especially vanilla ones, can suffer from issues like vanishing and exploding gradients, making it difficult to learn from long sequences. Transformers, on the other hand, can directly connect tokens from distant parts of the sequence, making them much better at modeling long-range dependencies.
4. **Multi-Head Attention**: Transformers employ multi-head attention, which allows the model to focus on different parts of the input sequence simultaneously. Each attention head can learn different aspects of the relationships between words, enhancing the model's ability to understand context and meaning.
5. **Positional Encoding**: Since transformers do not inherently understand the order of words, they use positional encoding to inject information about the position of each word in the sequence. This allows them to maintain the sequential nature of language while still benefiting from parallel processing.
6. **Scalability and Performance**: Transformers have shown to be more scalable and perform better on large datasets, which is crucial for many NLP tasks. Pre-trained models like BERT and GPT have set state-of-the-art results in various benchmarks due to their architecture and training methodologies.

The combination of parallel processing, self-attention mechanisms, and the ability to handle long-range dependencies makes transformer-based models like BERT and GPT significantly more effective than traditional RNNs in NLP tasks.

For further reading, you can refer to the note on Transformers vs RNNs for a detailed comparison of their architectures and performance.

Sources:

- Transformer
- Transformers vs RNNs
- BERT
- LSTM
- Attention mechanism
- Mathematical Reasoning in Transformers
- Recurrent Neural Networks
- Transfer Learning
- Multi-head attention
- BERT Pretraining of Deep Bidirectional Transformers for Language Understanding
- LLM
- Evaluating Language Models
- Bert Pretraining
- Reasoning tokens
- NLP
- Hugging Face
- Questions
- Neural network
- Boosting
- Named Entity Recognition
- Deep Learning
- Language Model Output Optimisation
- Small Language Models

Unsupervised Learning

Unsupervised learning is a type of machine learning where the algorithm is trained on data without explicit labels or predefined outputs.

Unsupervised learning involves discovering hidden patterns in data without predefined labels. It is valuable for exploratory data analysis, Clustering, and Isolated Forest.

The goal is to find hidden patterns, relationships, or structures in the data. Unlike supervised learning, which uses labeled input-output pairs, unsupervised learning relies solely on input data, allowing the algorithm to uncover insights independently.

Key Concepts

1. No Labeled Data: There is no ground truth or correct output associated with the input data.
2. Data Patterns: The algorithm identifies inherent structures, clusters, or associations within the dataset.
3. Objective: The primary objective is to explore the data and organize it to reveal underlying patterns.

Common Types of Unsupervised Learning

Clustering

Description: The algorithm groups similar data points together based on their features.

Example: Customer segmentation in marketing, where a clustering algorithm divides customers into groups based on purchasing behavior, demographics, or browsing history.

Popular Algorithms:

- K-means: Divides the data into (k) clusters, where each data point belongs to the nearest cluster.
- Hierarchical Clustering
- DBScan
- Support Vector Machines
- K-nearest neighbours

Dimensionality Reduction

Description: Reduces the number of input variables (features) while preserving as much information as possible. This is helpful for high-dimensional data, where visualization and analysis become challenging.

Popular Algorithms:

- Principal Component Analysis

Isolated Forest

Description: Identifies standardised/Outliers or unusual data points that don't conform to the expected pattern in the dataset.

Example: Detecting fraudulent credit card transactions by identifying transactions that deviate significantly from typical spending patterns.

Mechanism: Works by randomly partitioning the data and identifying standardised/Outliers|anomalies as points that can be isolated quickly.

Use Of Rnns In Energy Sector

For energy data problems, many **interpretable machine learning algorithms** can be applied in place of or alongside RNNs. These models offer transparency, making it easier to understand the relationships between features and predictions, which is critical in areas like energy management, where interpretability can be as important as accuracy.

For each of the energy data questions that RNNs might solve, **interpretable alternatives** Machine Learning Algorithms: such as **linear regression**, **decision trees**, **random forests**, and **ARIMA** models can be employed. These models provide **transparency** by revealing which features (e.g., weather, demand) influence predictions the most, making them suitable for stakeholders who need clear explanations of the decisions made by the model.

Demand forecasting

- **Algorithms**:
    - **Linear Regression**: Can model simple linear relationships between energy consumption and time (e.g., daily/seasonal trends).
    - **Decision Trees**: Provides clear if-then rules for predicting future energy usage based on historical consumption, time of day, and other factors.
    - **Random Forests**: An ensemble of decision trees that provides better accuracy than individual trees while still being interpretable using feature importance.

    - **Gradient Boosting (GBM)**: Can be used with feature importance or SHapley Additive exPlanations|SHAP values to understand which factors (e.g., time, weather) drive energy demand.
- **Why**: These models allow for clear interpretation of how factors like temperature, time of day, and previous energy use contribute to predictions.

2. Renewable Energy Generation Prediction

- **Algorithms**:
    - **Linear Regression**: For simple relationships, like the effect of sunlight hours or wind speed on energy generation.
    - **Support Vector Machines (SVM)**: Can create interpretable linear boundaries when predicting renewable energy outputs, with clear separation of factors (e.g., wind speed thresholds).
    - **Random Forests**: Offers feature importance metrics that explain which weather factors are most important for predicting energy generation.
    - **GBM**: Using SHapley Additive exPlanations|SHAP values or feature importance to interpret the impact of weather variables on the energy output.
- **Why**: These algorithms can provide insights into the key weather conditions driving renewable energy generation and give transparent predictions for decision-making.

3. Energy Price Forecasting

- **Algorithms**:
    - **ARIMA (AutoRegressive Integrated Moving Average)**: A traditional time series forecasting method that models linear relationships in energy prices over time.
    - **Linear Regression**: Can model the impact of factors like demand, supply, and historical prices in an interpretable way.
    - **Decision Trees**: Easy to interpret and can show thresholds where prices change based on inputs like demand or fuel costs.
    - **XGBoost**: Provides interpretability through SHAP values or feature importance, explaining which market factors (e.g., demand, fuel prices) drive price changes.
- **Why**: These algorithms offer interpretable insights into what drives price fluctuations, making them useful for energy market analysis and trading.

4. Anomaly Detection in Energy Consumption

- **Algorithms**:
    - **Isolation Forests**: Specifically designed for anomaly detection and provides interpretable results by isolating outliers.
    - **k-Nearest Neighbors (k-NN)**: Can flag anomalies by comparing new consumption data to known normal consumption patterns, with simple explanations of "closeness" to typical patterns.
    - **Logistic Regression**: Can be used to classify energy consumption data into "normal" and "anomalous" categories based on clear feature contributions.
    - **One-Class SVM**: A linear model that can classify whether energy usage deviates from typical patterns.
- **Why**: These interpretable algorithms can identify unusual patterns in energy data, providing clear reasons (e.g., thresholds exceeded) for flagging certain periods as anomalous.

5. Load Balancing and Optimization

- **Algorithms**:

  - **Linear Programming (Optimization)**: Provides interpretable rules for how energy should be distributed across the grid to minimize costs and prevent overloads.
  - **Decision Trees**: Can clearly show the impact of different factors (e.g., region, time of day) on grid load, and thresholds for balancing loads.
  - **Rule-Based Systems**: Set explicit rules for load balancing based on historical data and real-time demand, offering full transparency.
- **Why**: These interpretable models can assist grid operators in understanding which regions or time periods contribute most to load imbalances and suggest corrective actions.

6. Customer Energy Usage Profiling

- **Algorithms**:
  - **k-Means Clustering**: Can group customers into distinct profiles based on energy usage patterns, with each cluster representing a clear profile (e.g., high-energy consumers, off-peak users).
  - **Decision Trees**: Can predict customer profiles based on historical usage data and explain which features (e.g., time of usage, appliance usage) define each profile.
  - **Logistic Regression**: Can be used to classify customers into different segments based on usage characteristics, providing clear coefficient-based interpretations.
- **Why**: These models provide transparency into what factors drive a customer's energy usage profile, which is essential for creating personalized recommendations.

7. Demand Response Optimization

- **Algorithms**:
  - **Linear Programming (Optimization)**: Provides interpretable solutions for when and where to implement demand response programs to minimize peak energy use.
  - **Decision Trees**: Can clearly define rules for when demand response should be triggered based on time of day, weather, and current load.
  - **k-Nearest Neighbors (k-NN)**: Can identify similar past scenarios where demand response was implemented successfully and explain why the current situation matches.
- **Why**: These methods give clear, interpretable guidelines for when and how to reduce energy demand during peak times, based on past patterns.

8. Fault Detection in Power Systems

- **Algorithms**:
  - **Decision Trees**: Can explain why certain operational conditions (e.g., voltage drops, temperature increases) are likely to lead to faults, with clear rules and thresholds.
  - **Random Forests**: Provides feature importance scores that highlight which factors (e.g., temperature, load) are most indicative of impending faults.
  - **Logistic Regression**: Offers simple, interpretable probabilities for whether a fault will occur, based on key factors like current and voltage.
- **Why**: Fault detection requires clear, interpretable models that help engineers understand the most important factors leading to equipment failures.

9. Energy Usage Forecasting for Smart Buildings

- **Algorithms**:

- **Multiple Linear Regression**: Can model the relationship between building factors (e.g., temperature, occupancy) and energy usage, offering clear coefficients.
- **Decision Trees**: Provides an interpretable way to understand which building features (e.g., time of day, external temperature) influence energy consumption the most.
- **k-Means Clustering**: Can group similar time periods or usage patterns to explain different operational modes of the building.

- **Why**: These algorithms provide interpretable insights into how building features and external factors impact energy consumption, allowing for more efficient energy management.

10. Time Series Forecasting for Energy Production in Microgrids

- **Algorithms**:
  - **ARIMA**: Traditional interpretable time series model that predicts future production based on past production data.
  - **Linear Regression**: Can predict energy production based on simple factors like weather data, fuel availability, and historical output.
  - **Decision Trees**: Helps identify which weather or resource factors are most critical for predicting energy production at a given time.
- **Why**: Time series models like ARIMA are highly interpretable and useful for understanding how different factors contribute to energy production in microgrids.

11. Battery Storage Optimization

- **Algorithms**:
  - **Linear Programming (Optimization)**: Provides a clear, interpretable approach to optimizing charge/discharge schedules based on forecasted energy generation and consumption.
  - **Decision Trees**: Can explain when and why batteries should be charged or discharged based on energy production, consumption, and cost factors.
  - **Rule-Based Systems**: Establish clear rules for battery storage optimization, offering fully interpretable decision-making processes.
- **Why**: Optimizing battery storage requires clear, rule-based or linear models to understand how different variables (e.g., energy prices, consumption) impact storage decisions.

Wcss And Elbow Method

USE: WCSS (within-cluster sum of squares)

WCSS is a measure developed within the ANOVA framework. It gives a very good idea about the different distance between different clusters and within clusters, thus providing us a rule for deciding the appropriate number of clusters.

The plot will resemble an "elbow," and the goal is to find the point where the decrease in WCSS slows down, forming an elbow-like shape.

Elbow numbers are the point where the rate of decrease in WCSS starts to flatten out

The rationale behind the elbow method is that

Rationale: as you increase the number of clusters (K), the WCSS will generally decrease because each cluster becomes smaller. However, there is a point where the addition of more clusters provides diminishing returns in terms of reducing WCSS. The elbow point represents a good balance between capturing the variance in the data

and avoiding excessive fragmentation.

Code

```
# Use WCSS and elbow method
# number of clusters
wcss=[]
start=2
end=10
# Create all possible cluster solutions with a loop
for i in range(start,end):
    # Cluster solution with i clusters
    kmeans = KMeans(i)
    # Fit the data
    kmeans.fit(df_scaled)
    # Find WCSS for the current iteration
    wcss_iter = kmeans.inertia_
    # Append the value to the WCSS list
    wcss.append(wcss_iter)



# Create a variable containing the numbers from 1 to 6, so we can use it as X axis of the future plot

number_clusters = range(start,end)

# Plot the number of clusters vs WCSS

plt.plot(number_clusters,wcss)

# Name your graph

plt.title('The Elbow Method')
# Name the x-axis
plt.xlabel('Number of clusters')
# Name the y-axis
plt.ylabel('Within-cluster Sum of Squares')

# Identify the elbow numbers (there may be more than one thats best)
elbow_nums=[4,5,6,7,8]
```

plotting

```
# function to give scatter for each elbow number


def scatter_elbow(X, elbow_num, var1, var2):
    """
    Apply clustering with elbow method and plot a scatter plot with cluster information.

    Parameters:
    - X: DataFrame, input data for clustering
    - elbow_num: int, number of clusters determined by elbow method
    - var1, var2: str, names of the variables for the scatter plot

    Returns:
    None (plots the scatter plot)
    """
    # Apply [clustering](#clustering) with elbow number
    kmeans = KMeans(elbow_num)
    kmeans.fit(X)

    # Add cluster information
    identified_clusters = kmeans.fit_predict(X)
    X['Cluster'] = identified_clusters

    # Plot
    plt.scatter(X[var1], X[var2], c=X['Cluster'], cmap='rainbow')
    plt.xlabel(var1)
    plt.ylabel(var2)
    plt.title(f"{elbow_num}-Clustering for {var1}-{var2}")
    plt.show()
# Example usage:
# scatter_elbow(data, elbow_num, 'var1', 'var2')
for elbow_num in elbow_nums:
    scatter_elbow(df, elbow_num, var1, var2)
```

Xgboost

XGBoost (eXtreme Gradient Boosting) is a highly efficient and flexible implementation of Gradient Boosting that is widely used for its accuracy and performance in machine learning tasks.

How does XGBoost work

It works by building an Model Ensemble - ensemble of decision trees, where each tree is trained to correct the errors made by the previous ones. Here's a breakdown of how XGBoost works:

Key Concepts

1. Gradient Boosting Framework:
    - XGBoost is based on the gradient boosting framework, which builds models sequentially. Each new model aims to reduce the errors (residuals) of the combined ensemble of previous models.

2. Decision Trees:
    - XGBoost typically uses decision trees as the base learners. These trees are added one at a time, and existing trees in the model are not changed.
3. Objective Function:
    - The objective function in XGBoost consists of two parts: the loss function and a regularization term.
    - Loss function: Measures how well the model fits the training data. For regression, this might be mean squared error; for classification, it could be logistic loss.
    - Regularisation: Helps prevent overfitting by penalizing complex models. XGBoost supports both L1 (Lasso) and L2 (Ridge) regularization.
4. Additive Training:
    - XGBoost adds trees to the model sequentially. Each tree is trained to minimize the loss function, taking into account the errors made by the previous trees.
5. Gradient Descent
    - The model uses gradient descent to minimize the loss function. It calculates the gradient of the loss function with respect to the model's predictions and uses this information to update the model.
6. learning rate (η):
    - A parameter that scales the contribution of each tree. A smaller learning rate requires more trees but can lead to better performance.
7. Tree Pruning:
    - XGBoost uses a technique called "max depth" to control the complexity of the trees. It also employs a "max delta step" to ensure that the updates are not too aggressive.
8. Handling Missing Data
    - XGBoost can handle missing data internally by learning the best direction to take when a value is missing.
9. Parallel and Distributed Computing:
    - XGBoost is designed to be highly efficient and can leverage parallel and distributed computing to speed up training.

Key Features:

- Tree Splitting: Builds Decision Tree in a level-wise manner, leading to balanced trees and efficient computation.
- Parameters: Key parameters include `eta` (learning rate) and `max_depth` (maximum depth of a tree), which control the model's complexity and learning process.

Workflow

1. Initialization:
    - Start with an initial prediction, often the mean of the target values for regression or a uniform probability for classification.
2. Iterative Training:
    - For each iteration, compute the gradient of the loss function with respect to the current predictions.
    - Fit a new decision tree to the negative gradient (residuals).
    - Update the model by adding the new tree, scaled by the learning rate.
3. Model Output:
    - The final model is a weighted sum of all the trees, where each tree contributes to the final prediction.

Advantages:

- Accuracy: Known for its high accuracy and robustness across various machine learning tasks.
- Regularisation: Supports L1 (Lasso) and L2 (Ridge) regularization to prevent overfitting.
- Flexibility: Offers a wide range of hyperparameters for fine-tuning models.

Use Cases:

- Structured Data: Particularly effective for structured data and tabular datasets.
- Interpretability: Suitable when model interpretability is important.
- Hyperparameter Tuning: Ideal for scenarios where extensive hyperparameter tuning is feasible.

Implementing XGBoost in Python

Step 2: Import Necessary Libraries

```
import xgboost as xgb
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
```

Step 3: Prepare Your Data

Split your dataset into training and testing sets:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Step 4: Convert Data to DMatrix

Convert the data into DMatrix, the optimized data structure used by XGBoost:

```
dtrain = xgb.DMatrix(X_train, label=y_train)
dtest = xgb.DMatrix(X_test, label=y_test)
```

Step 5: Set Parameters

Define the parameters for the XGBoost model:

```
params = {
    'max_depth': 6,
    'eta': 0.1,
    'objective': 'binary:logistic',  # Use 'reg:squarederror' for regression tasks
    'eval_metric': 'logloss'
}
```

Step 6: Train the Model

Train the XGBoost model using the training data:

```
num_rounds = 100
bst = xgb.train(params, dtrain, num_rounds)
```

Step 7: Make Predictions and Evaluate

Make predictions on the test set and evaluate the model's performance:

```
y_pred = bst.predict(dtest)
y_pred_binary = [1 if y > 0.5 else 0 for y in y_pred]
accuracy = accuracy_score(y_test, y_pred_binary)
print(f"Accuracy: {accuracy:.2f}")
```

Notes

Set up an example of XGBoost. Plot the paramater space slices "Min_Samples_split", "Max_Depth" vs accuracy.

```
xgb_model = XGBClassifier(n_estimators = 500, learning_rate = 0.1,verbosity = 1, random_state = RANDOM_STATE)
xgb_model.fit(X_train_fit,y_train_fit, eval_set = [(X_train_eval,y_train_eval)], early_stopping_rounds = 10)
xgb_model.best_itersation
```