LogisticRegression(max\_iter=1000, solver='liblinear'

* **max\_iter=1000**:  
  Specifies the maximum number of iterations allowed for the optimization algorithm to converge. If the model struggles to converge with fewer iterations, increasing this value helps ensure convergence.
* **solver='liblinear'**:  
  This defines the optimization algorithm to be used for fitting the model. 'liblinear' is suitable for smaller datasets and uses a coordinate descent algorithm for optimization.
* **How it works**:  
  Logistic Regression is a linear model used for classification. It predicts probabilities using the **logistic function (sigmoid)**, which squashes a linear combination of input features into a value between 0 and 1. Based on a threshold (default is 0.5), it assigns a class label (e.g., 0 or 1).
* **Use case**: Best for binary classification problems and when classes are linearly separable.

RidgeClassifierCV(alphas=0.1)

* **alphas=0.1**:  
  This controls the regularization strength. A smaller alpha (e.g., 0.1) results in less regularization, while a larger alpha would apply stronger regularization. RidgeClassifierCV performs cross-validation to select the best alpha from the given values.
* **How it works**:  
  RidgeClassifier is similar to Logistic Regression but uses Ridge regularization to penalize large coefficients. It minimizes a loss function that balances fitting the data and keeping the model simple.  
  The **CV** (cross-validation) automatically selects the best regularization strength (alpha).
* **Use case**: Works well for multiclass problems and when overfitting needs to be controlled with regularization.

KNeighborsClassifier(n\_neighbors=5, metric='manhattan')

* **n\_neighbors=5**:  
  The number of nearest neighbors to consider when making a classification decision. The class of a sample is determined by a majority vote among its 5 nearest neighbors.
* **metric='manhattan'**:  
  This specifies the distance metric to compute the similarity between points. 'manhattan' refers to the Manhattan distance (or L1 distance), which sums the absolute differences between feature values.
* **How it works**:  
  KNN is a non-parametric, instance-based learning algorithm. It classifies a data point based on the **majority vote** of its **k nearest neighbors** in the feature space. The similarity (distance) is computed using metrics like Manhattan or Euclidean distance.
* **Use case**: Effective for small datasets with clearly defined clusters or when the decision boundary is non-linear. It can be slow for large datasets due to distance calculations.

DecisionTreeClassifier(max\_depth=6, min\_samples\_split=5, min\_samples\_leaf=2)

* **max\_depth=6**:  
  Sets the maximum depth of the tree. Limiting the depth helps prevent overfitting, especially when the dataset is small.
* **min\_samples\_split=5**:  
  Specifies the minimum number of samples required to split an internal node. For example, if a node has fewer than 5 samples, it will not be split further.
* **min\_samples\_leaf=2**:  
  Ensures that each leaf node has at least 2 samples. This prevents the tree from creating overly small leaf nodes, which might lead to overfitting.
* **How it works**:  
  Decision Trees split the data into subsets based on feature values to create a tree structure. At each split, it selects the feature that maximizes information gain (or minimizes impurity like Gini or Entropy). The process continues until stopping criteria like max\_depth are met.
* **Use case**: Flexible, interpretable model that works well with non-linear decision boundaries. However, prone to overfitting if not constrained.

RandomForestClassifier(max\_depth=8, min\_samples\_split=5, min\_samples\_leaf=2)

* max\_depth=8:

Limits the maximum depth of each tree in the forest to 8, reducing the risk of overfitting.

* min\_samples\_split=5:

Similar to the parameter in DecisionTreeClassifier, it controls the minimum number of samples required to split a node.

* min\_samples\_leaf=2:

Also similar to DecisionTreeClassifier, this ensures that each leaf node has at least 2 samples, making the trees less likely to overfit.

* **How it works**:  
  Random Forest builds an **ensemble of decision trees**. Each tree is trained on a random subset of data and features. The final prediction is made by **majority voting** (classification) or averaging (regression). This reduces overfitting and improves generalization.
* **Use case**: Great for tabular data with mixed feature types (numerical and categorical) and datasets prone to overfitting.

Dense

* Sure, I'd be happy to help you understand this code! This script is for building and training a neural network using Keras, a high-level neural networks API. Here's a breakdown of what each part does:
* **Importing Libraries**:
  + from keras.models import Sequential
  + from keras.layers import Dense, Dropout
* These lines import the necessary modules from Keras. Sequential is used to initialize a linear stack of layers, and Dense and Dropout are types of layers.
* **Initializing the Model**:
  + model = Sequential()
* This initializes a sequential model, which means the layers will be added one after another.
* **Adding Layers**:
  + model.add(Dense(128, activation='relu', input\_shape=(xtrain.shape[1],)))
  + model.add(Dropout(0.2))
  + model.add(Dense(64, activation='relu'))
  + model.add(Dropout(0.2))
  + model.add(Dense(len(np.unique(ytrain\_encoded)), activation='softmax'))
* Dense(128, activation='relu', input\_shape=(xtrain.shape[1],)): Adds a fully connected layer with 128 neurons and ReLU activation function. input\_shape specifies the shape of the input data.
* Dropout(0.2): Adds a dropout layer with a dropout rate of 20% to prevent overfitting.
* Dense(64, activation='relu'): Adds another fully connected layer with 64 neurons and ReLU activation.
* Dropout(0.2): Adds another dropout layer with a 20% dropout rate.
* Dense(len(np.unique(ytrain\_encoded)), activation='softmax'): Adds the output layer with a number of neurons equal to the number of unique classes in ytrain\_encoded and uses the softmax activation function for multi-class classification.
* **Compiling the Model**:
* model.compile(optimizer='adam', loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])
* This configures the model for training. It uses the Adam optimizer, sparse categorical cross-entropy loss function (suitable for multi-class classification), and tracks accuracy as a metric.
* **Training the Model**:
* model.fit(xtrain, ytrain\_encoded, epochs=100, batch\_size=32, validation\_data=(xtest, ytest\_encoded))
* This trains the model on the training data (xtrain and ytrain\_encoded) for 100 epochs with a batch size of 32. It also evaluates the model on the validation data (xtest and ytest\_encoded) after each epoch.
* **Making Predictions**:
  + y\_pred = model.predict(xtest)
  + y\_pred\_classes = np.argmax(y\_pred, axis=1)
* model.predict(xtest): Generates predictions for the test data.
* np.argmax(y\_pred, axis=1): Converts the predicted probabilities to class labels by taking the index of the highest probability.
* **Evaluating the Model**:
* print('Training Accuracy : ', accuracy\_score(ytrain\_encoded, np.argmax(model.predict(xtrain), axis=1)))
* print('Validation Accuracy : ', accuracy\_score(ytest\_encoded, y\_pred\_classes))
* accuracy\_score(ytrain\_encoded, np.argmax(model.predict(xtrain), axis=1)): Calculates and prints the training accuracy.
* accuracy\_score(ytest\_encoded, y\_pred\_classes): Calculates and prints the validation accuracy.

**ReLU (activation function)**

ReLU is an activation function used in neural networks. It is defined as: ReLU(*x*)=max(0,*x*) This means that if the input is positive, it returns the input value; if the input is negative, it returns zero. ReLU is popular because:

* **Simplicity**: It's easy to implement and computationally efficient.
* **Sparse Activation**: It activates only a portion of the neurons, making the network more efficient.
* **Mitigates Vanishing Gradient Problem**: Unlike sigmoid or tanh functions, ReLU helps in reducing the vanishing gradient problem, allowing for faster training of deep networks.

**Adam (Adaptive Moment Estimation)**

Adam is an optimization algorithm used to update network weights iteratively based on training data. It combines the advantages of two other extensions of stochastic gradient descent: AdaGrad and RMSProp. Key features of Adam include:

* **Adaptive Learning Rates**: It computes individual adaptive learning rates for different parameters.
* **Momentum**: It uses estimates of first and second moments of the gradients to adapt the learning rate.
* **Efficiency**: It's computationally efficient and requires little memory[[2]](https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/).

The update rule for Adam involves the following steps:

1. Compute the gradients of the current mini-batch.
2. Update biased first moment estimate.
3. Update biased second raw moment estimate.
4. Compute bias-corrected first and second moment estimates.
5. Update parameters using these estimates.

**Softmax**

Softmax is an activation function often used in the output layer of a neural network for multi-class classification. It converts raw scores (logits) into probabilities. The formula for softmax is:   
Et billede, der indeholder Font/skrifttype, hvid, tekst, design

Automatisk genereret beskrivelse  
where ( z\_i ) is the input to the ( i )-th neuron, and ( K ) is the number of classes. Softmax ensures that the output values are between 0 and 1 and sum to 1, making them interpretable as probabilities[[3]](https://en.wikipedia.org/wiki/Softmax_function).