ML Models using scikit-learn

Generic scikit-learn pipeline

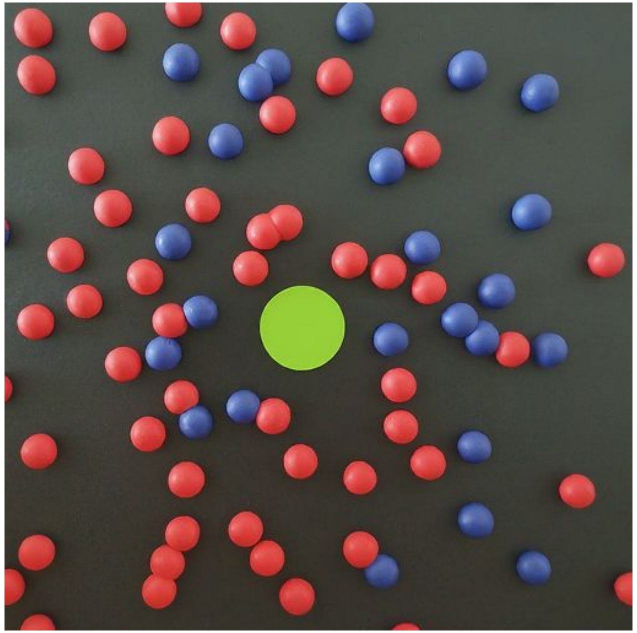
1. Apply appropriate preprocessing on the data.
2. Build train and test split from the data.
3. Import relevant model class from sklearn
4. Initialize model with appropriate (hyper)parameters — can be found in the description of the model class page in sklearn's repository.
5. Call model.fit() on train split of data
6. Get predictions on test split using model.predict()

ML models for this assignment

- K-nearest Neighbor
- Logistic Regression
- Support Vector Machine
- K-Means Clustering
- Neural Networks

K-Nearest Neighbors

K-Nearest Neighbors (KNN) in scikit-learn



What is KNN

KNN is a supervised learning algorithm used for classification and regression tasks.

It works by predicting the class or value of a new data point based on the labels or values of its k nearest neighbors in the training data.

The value of k, the number of neighbors to consider, is a crucial parameter that needs to be tuned for optimal performance.

Implementation in scikit-learn

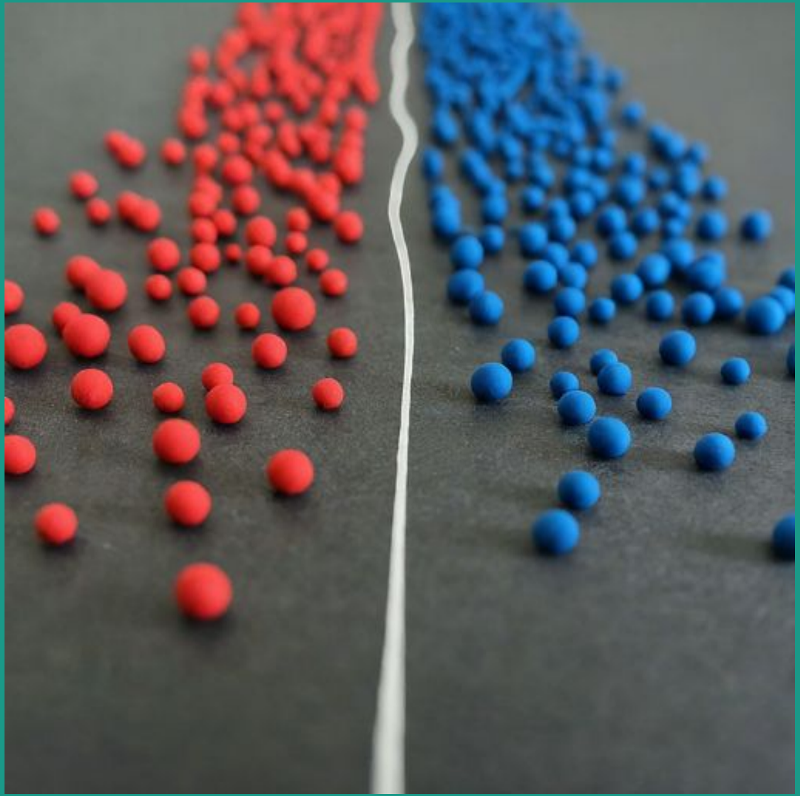
- Import the KNeighborsClassifier class from scikit-learn.
- Create an instance of the class, specifying the desired value of k and other parameters (optional).
- Fit the model to the training data using the fit method.
- Make predictions on new data using the predict method.

```
from sklearn.neighbors import
KNeighborsClassifier

# Define the model
model =
KNeighborsClassifier(n_neighbors=5
)

# Fit the model to the training
data
model.fit(X_train, y_train)

# Make predictions on new data
y_pred = model.predict(X_test)
```

Considerations for KNN

- **Choice of k:** The value of k significantly impacts performance. A high k can lead to overfitting, while a low k can lead to underfitting. Experimenting with different k values is crucial.
- **Data dimensionality:** KNN can be sensitive to the number of features in the data. Feature scaling or dimensionality reduction techniques might be needed for high-dimensional datasets.
- **Distance metric:** The choice of distance metric (e.g., Euclidean, Manhattan) can also influence the performance.

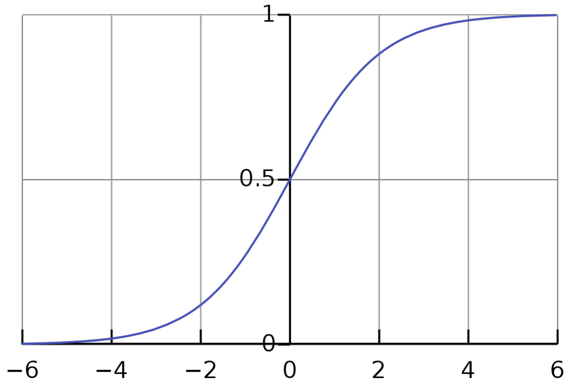
Logistic Regression



Understanding Logistic Regression

- LR is a supervised learning algorithm used for binary classification problems.
- It predicts the probability of a data point belonging to a specific class (usually labeled as 0 or 1).
- The model utilizes the sigmoid function to transform the linear combination of features into a probability between 0 and 1.

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



Implementation in scikit-learn

- Import the LogisticRegression class from scikit-learn.
- Create an instance of the class, specifying parameters like the solver and regularization method (optional).
- Fit the model to the training data using the fit method.
- Make predictions on new data using the predict method and obtain class probabilities using the predict_proba method.

```
from sklearn.linear_model import
LogisticRegression

# Define the model
model = LogisticRegression(solver='lbfgs')

# Fit the model to the training data
model.fit(X_train, y_train)

# Make predictions and obtain probabilities
y_pred = model.predict(X_test)
y_proba = model.predict_proba(X_test)
```

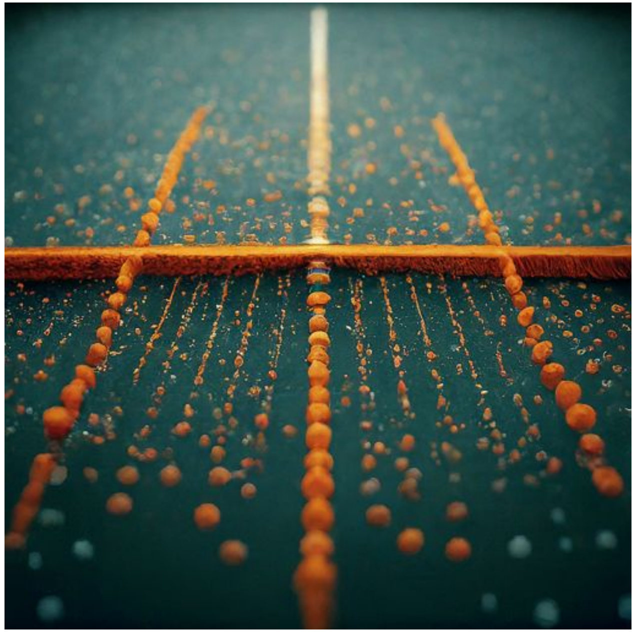
Considerations for Logistic Regression

- **Data preprocessing:** LR often performs better with scaled features. Standardization or normalization techniques might be needed.
- **Regularization:** Choosing the right regularization parameter (e.g., C) can help prevent overfitting and improve model generalization.
- **Class imbalance:** LR can be sensitive to class imbalance. Techniques like oversampling or undersampling the minority class might be necessary.

Support Vector Machines

Understanding SVMs

- SVMs are supervised learning algorithms that can be used for classification or regression.
- For classification, they aim to find a hyperplane in the feature space that maximizes the margin between the classes.
- The margin is the distance between the hyperplane and the closest data points from each class, called support vectors.



Implementation in scikit-learn

- Import the SVC class from scikit-learn for classification or SVR for regression.
- Create an instance of the class, specifying the kernel function (e.g., linear, rbf) and other parameters like regularization.
- Fit the model to the training data using the fit method.
- Make predictions on new data using the predict method.

```
from sklearn.svm import SVC

# Define the model
model = SVC(kernel='linear')

# Fit the model to the training
data
model.fit(X_train, y_train)

# Make predictions on new data
y_pred = model.predict(X_test)
```

Considerations for SVMs

- **Kernel function:** The choice of kernel function (e.g., linear, RBF) significantly impacts the model's ability to handle non-linear data. Experimenting with different kernels is often crucial.
- **Regularization:** The regularization parameter (C) controls the trade-off between maximizing the margin and avoiding overfitting. Tuning this parameter is essential.
- **Scalability:** SVMs can be computationally expensive for large datasets, especially with certain kernel functions. Consider alternatives for large-scale problems.

K-Means Clustering

Understanding K-Means Clustering

- It groups data points into <u>**k**</u> pre-defined clusters based on their similarity.
- The algorithm iteratively performs the following steps:
  - **Initializes centroids:** Randomly selects k data points as initial cluster centers (centroids).
  - **Assigns points to clusters:** Assigns each data point to the nearest centroid.
  - **Recomputes centroids:** Updates the centroids by calculating the mean of the data points assigned to each cluster.
  - **Repeats:** Repeats steps 2 and 3 until the centroids no longer change significantly (convergence).

Implementation in scikit-learn

- Import the KMeans class from scikit-learn.
- Create an instance of the class, specifying the desired number of clusters (k).
- Fit the model to the data using the fit method.
- Predict the cluster labels for new data using the predict method.

```
from sklearn.cluster import KMeans

# Define the model
model = KMeans(n_clusters=3)

# Fit the model to the data
model.fit(X)

# Predict cluster labels for new data
cluster_labels = model.predict(X_new)
```

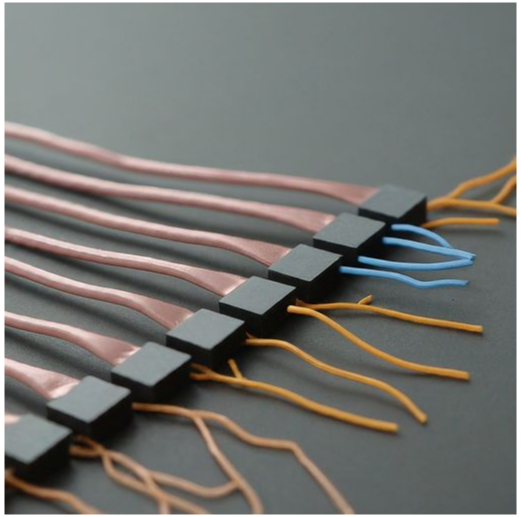
Considerations for K-Means Clustering

- **Choosing the number of clusters (k):** This is a crucial step and often requires experimentation. Different metrics like the elbow method or silhouette score can be used to evaluate different k values.
- **Data preprocessing:** K-Means is sensitive to the scale of the features. Standardization or normalization might be necessary before applying the algorithm.
- **Initialization:** The initial placement of centroids can affect the final results. Different initialization methods like k-means++ can be used to improve the stability and convergence of the algorithm.

Neural Networks

Understanding Neural Networks

- Artificial Neural Networks (ANNs) are inspired by the biological structure and function of the brain.
- They consist of interconnected nodes called neurons, arranged in layers.
- Information flows through the network from the input layer to the output layer through hidden layers.
- Each neuron applies a non-linear activation function to transform the weighted sum of its inputs.



Implementation in scikit-learn

- Scikit-learn provides the MLPClassifier and MLPRegressor classes for classification and regression tasks, respectively.
- Key parameters:
  - **hidden_layer_sizes:** A tuple specifying the number of neurons in each hidden layer.
  - **activation:** The activation function to be used in the hidden layers.
  - **solver:** The algorithm used to optimize the model's weights.
  - **learning_rate_init:** The initial learning rate used. It controls the step-size in updating the weights.

```
from sklearn.neural_network import
MLPClassifier

# Define the model
model =
MLPClassifier(hidden_layer_sizes=(1
0, 5), activation='relu',
solver='lbfgs')

# Fit the model to the training
data
model.fit(X_train, y_train)

# Predict labels for new data
y_pred = model.predict(X_test)
```

Considerations for Neural Networks

- **Architecture selection**: Choosing the optimal number of hidden layers and neurons significantly impacts performance. Experimentation and techniques like cross-validation are crucial.
- **Hyperparameter tuning**: Tuning hyperparameters like learning rate, batch size, activation function, and solver can significantly improve model performance.
- **Overfitting**: NNs are prone to overfitting, especially with large models and small datasets. Techniques like regularization and EarlyStopping can help you tackle overfitting to certain extent.

Thank You!