

Below is a well-structured Python module implementing the **Simple Perceptron**, **Probabilistic Neural Network (PNN)**, and **K-Means Clustering** using only `numpy`, `scipy`, `pandas`, and `matplotlib` for educational purposes. The code includes comments to clarify each step.

```
import numpy as np

from scipy.stats import multivariate_normal

import matplotlib.pyplot as plt


class SimplePerceptron:
    """
    Simple Perceptron classifier for binary classification.
    """

    def __init__(self, input_dim, learning_rate=0.1, max_iter=1000):
        self.w = np.zeros(input_dim + 1) # +1 for bias
        self.learning_rate = learning_rate
        self.max_iter = max_iter

    def _augment(self, X):
        # Add bias term (column of ones)
        return np.hstack([X, np.ones((X.shape[0], 1))])

    def _activation(self, z):
        # Step function
        return np.where(z >= 0, 1, 0)

    def fit(self, X, y):
        """
        Train the perceptron on data X with labels y.
        """
```

```

X: np.ndarray, shape (n_samples, n_features)
y: np.ndarray, shape (n_samples,), binary labels {0,1}
"""

X_aug = self._augment(X)
for _ in range(self.max_iter):
    errors = 0
    for xi, target in zip(X_aug, y):
        z = np.dot(self.w, xi)
        y_pred = self._activation(z)
        e = target - y_pred
        if e != 0:
            self.w += self.learning_rate * e * xi
    errors += 1
    if errors == 0:
        break # Converged

def predict(self, X):
    X_aug = self._augment(X)
    z = np.dot(X_aug, self.w)
    return self._activation(z)

class ProbabilisticNeuralNetwork:
    """
    Probabilistic Neural Network (PNN) for multi-class classification.
    Uses Gaussian kernels for Parzen window density estimation.
    """

    def __init__(self, sigma=1.0):

```

```

self.sigma = sigma
self.train_X = None
self.train_y = None
self.classes_ = None

def fit(self, X, y):
    """
    Store training data.
    """
    self.train_X = X
    self.train_y = y
    self.classes_ = np.unique(y)

def _gaussian_kernel(self, x, xi):
    d = x.shape[0]
    diff = x - xi
    exponent = -np.dot(diff, diff) / (2 * self.sigma ** 2)
    coeff = 1 / ((2 * np.pi) ** (d / 2) * self.sigma ** d)
    return coeff * np.exp(exponent)

def predict(self, X):
    """
    Predict class labels for X.
    """
    y_pred = []
    for x in X:
        # Estimate class conditional densities
        densities = []
        for c in self.classes_:

```

```

X_c = self.train_X[self.train_y == c]
n_c = X_c.shape[0]
# Parzen window estimate
p_x_given_c = np.sum([self._gaussian_kernel(x, xi) for xi in X_c]) / n_c
densities.append(p_x_given_c)
densities = np.array(densities)
# Assuming uniform priors
posteriors = densities / np.sum(densities)
y_pred.append(self.classes_[np.argmax(posteriors)])
return np.array(y_pred)

class KMeansClustering:
    """
    K-Means clustering algorithm.
    """

    def __init__(self, n_clusters=3, max_iter=300, tol=1e-4, random_state=None):
        self.n_clusters = n_clusters
        self.max_iter = max_iter
        self.tol = tol
        self.random_state = random_state
        self.centroids = None
        self.labels_ = None

    def fit(self, X):
        """
        Fit the K-Means model to data X.
        """

```

```

if self.random_state is not None:
    np.random.seed(self.random_state)
    n_samples, n_features = X.shape

    # Initialize centroids randomly from data points
    indices = np.random.choice(n_samples, self.n_clusters, replace=False)
    self.centroids = X[indices]

    for iteration in range(self.max_iter):
        # Assign labels based on closest centroid
        distances = np.linalg.norm(X[:, np.newaxis] - self.centroids, axis=2) # shape
        (n_samples, n_clusters)
        labels = np.argmin(distances, axis=1)

        # Compute new centroids
        new_centroids = np.array([X[labels == i].mean(axis=0) if np.any(labels == i) else
        self.centroids[i]
        for i in range(self.n_clusters)])

        # Check for convergence
        if np.all(np.linalg.norm(new_centroids - self.centroids, axis=1) < self.tol):
            break

        self.centroids = new_centroids

    self.labels_ = labels

    def predict(self, X):
        """
        Predict the closest cluster each sample in X belongs to.

```

```

"""
distances = np.linalg.norm(X[:, np.newaxis] - self.centroids, axis=2)
return np.argmin(distances, axis=1)

def plot_clusters(self, X, title="K-Means Clustering"):
    """
    Visualize clusters in 2D (only works if X has 2 features).
    """
    if X.shape[1] != 2:
        raise ValueError("Plotting only supported for 2D data.")
    plt.figure(figsize=(8, 6))
    plt.scatter(X[:, 0], X[:, 1], c=self.labels_, cmap='viridis', marker='o', alpha=0.6)
    plt.scatter(self.centroids[:, 0], self.centroids[:, 1], c='red', marker='X', s=200,
                label='Centroids')
    plt.title(title)
    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.legend()
    plt.show()

# Example usage (uncomment to run):
if __name__ == "__main__":
    # Simple Perceptron example
    X_perc = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
    y_perc = np.array([0, 0, 0, 1]) # AND logic gate
    perc = SimplePerceptron(input_dim=2)
    perc.fit(X_perc, y_perc)
    print("Perceptron predictions:", perc.predict(X_perc))

```

```
# PNN example

from sklearn.datasets import make_classification

X_pnn, y_pnn = make_classification(
    n_samples=100,
    n_features=2,
    n_classes=2,
    n_informative=2,
    n_redundant=0,
    n_repeated=0,
    random_state=42
)

pnn = ProbabilisticNeuralNetwork(sigma=0.5)
pnn.fit(X_pnn, y_pnn)
preds_pnn = pnn.predict(X_pnn)
print("PNN accuracy:", np.mean(preds_pnn == y_pnn))


# K-Means example

X_km, _ = make_classification(
    n_samples=200,
    n_features=2,
    n_clusters_per_class=1,
    n_classes=3,
    n_informative=2,
    n_redundant=0,
    n_repeated=0,
    random_state=42
)

km = KMeansClustering(n_clusters=3, random_state=42)
km.fit(X_km)
```

```
print("K-Means cluster centers:\n", km.centroids)
km.plot_clusters(X_km)
```

Explanation:

- **SimplePerceptron:**
Implements the perceptron learning algorithm with bias augmentation, step activation, and weight updates on misclassified samples.
- **ProbabilisticNeuralNetwork:**
Implements PNN classification by storing training data and estimating class-conditional densities with Gaussian kernels. Prediction uses Bayes' rule assuming uniform class priors.
- **KMeansClustering:**
Implements Lloyd's algorithm for k-means clustering: random centroid initialization, iterative assignment, centroid update, and convergence check. Includes a 2D plotting method for visualization.