> I have a strong foundational understanding and practical experience with several core machine learning techniques, including K-Nearest Neighbors (KNN), Naive Bayes, Logistic Regression, Neural Networks, Linear Regression, and K-Means Clustering. I've implemented these using libraries like scikit-learn and TensorFlow, and have applied them to various classification and clustering tasks. Currently, I am focusing on building a deeper understanding of Support Vector Machines (SVM) and Principal Component Analysis (PCA) — exploring how SVMs separate data using optimal hyperplanes and how PCA reduces dimensionality by capturing maximum variance in fewer components.

## K-Nearest Neighbors (KNN)

**Definition**: K-Nearest Neighbors is a non-parametric, instance-based learning algorithm that classifies a data point based on the majority class of its k nearest neighbors in the feature space, using a distance metric.

**How it works**:

- During training, the algorithm simply stores all training examples
- For prediction, it calculates the distance between the query instance and all training examples
- It identifies the K nearest neighbors based on distance metric (commonly Euclidean)
- The majority class among these K neighbors becomes the prediction
- For regression tasks, it returns the average of the K neighbors' values

**Key Parameters**:

- `n_neighbors`: Number of neighbors to consider (higher values reduce noise sensitivity but may blur decision boundaries)
- `weights`: Uniform (all neighbors weighted equally) or distance-based (closer neighbors have more influence)
- `metric`: Distance measure (Euclidean, Manhattan, Minkowski, etc.)
- `algorithm`: Method used to compute nearest neighbors ('auto', 'ball_tree', 'kd_tree', 'brute')
- `leaf_size`: Affects speed of tree-based algorithms

**Implementation**:

```python
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

# Create the model
knn = KNeighborsClassifier(
    n_neighbors=5,
    weights='uniform',
    algorithm='auto',
    metric='minkowski',
    p=2  # p=2 for Euclidean distance
)

# Train the model
knn.fit(X_train, y_train)

# Make predictions
```

```
y_pred = knn.predict(X_test)

# Evaluate
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.4f}")
```

## Naive Bayes

**Definition**: Naive Bayes is a probabilistic classifier based on Bayes' theorem with an assumption of conditional independence between features given the class label.

**How it works**:

- Applies Bayes' theorem: P(y|X) = P(X|y) * P(y) / P(X)
- "Naive" because it assumes all features are conditionally independent given the class
- Calculates the probability of each class for a given input
- Selects the class with the highest probability as the prediction
- Different variants handle different types of features (Gaussian for continuous, Multinomial for counts, Bernoulli for binary)

**Key Parameters**:

- `var_smoothing`: Portion of the largest variance of all features that is added to variances for calculation stability (Gaussian NB)
- `alpha`: Additive (Laplace/Lidstone) smoothing parameter (Multinomial NB)
- `fit_prior`: Whether to learn class prior probabilities or use a uniform prior
- `class_prior`: Prior probabilities of the classes

**Implementation**:

```
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import classification_report

# Create model for continuous features
nb = GaussianNB(var_smoothing=1e-9)

# Train the model
nb.fit(X_train, y_train)

# Make predictions
y_pred = nb.predict(X_test)

# Evaluate
print(classification_report(y_test, y_pred))

# For text classification (discrete features):
from sklearn.naive_bayes import MultinomialNB

# Create model for text/count data
```

```
mnb = MultinomialNB(alpha=1.0)
mnb.fit(X_train_text, y_train)
```

## Logistic Regression

**Definition**: Logistic Regression is a statistical model that uses a logistic function to model a binary dependent variable, estimating the probability that an instance belongs to a particular category.

**How it works**:

- Computes a weighted sum of input features: $z = w_0 + w_1x_1 + w_2x_2 + ... + w_nx_n$
- Transforms this sum using the sigmoid function: σ(z) = 1/(1 + e^(-z))
- Output is interpreted as probability of belonging to positive class
- Uses threshold (usually 0.5) to convert probability to class label
- Parameters are typically learned using maximum likelihood estimation

**Key Parameters**:

- `C`: Inverse of regularization strength (smaller values = stronger regularization)
- `penalty`: Type of regularization ('l1', 'l2', 'elasticnet', 'none')
- `solver`: Algorithm for optimization ('lbfgs', 'liblinear', 'newton-cg', 'sag', 'saga')
- `max_iter`: Maximum number of iterations for solver
- `multi_class`: Strategy for multi-class classification ('ovr' - one-vs-rest, 'multinomial')
- `class_weight`: Weights associated with classes to handle imbalanced datasets

**Implementation**:

```python
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_auc_score

# Create model with L2 regularization
log_reg = LogisticRegression(
    C=1.0,
    penalty='l2',
    solver='lbfgs',
    max_iter=1000,
    class_weight='balanced'
)

# Train model
log_reg.fit(X_train, y_train)

# Predict probabilities
y_prob = log_reg.predict_proba(X_test)[:, 1]
y_pred = log_reg.predict(X_test)

# Evaluate
roc_auc = roc_auc_score(y_test, y_prob)
print(f"ROC AUC: {roc_auc:.4f}")
```

```
# Get coefficients
coefficients = pd.DataFrame({
    'Feature': X_train.columns,
    'Coefficient': log_reg.coef_[0]
}).sort_values('Coefficient', ascending=False)
```

---

## Support Vector Machines (SVM)

**Definition**:
Support Vector Machines (SVM) are powerful supervised learning algorithms used for both classification and regression tasks. They aim to find the **optimal hyperplane** that best separates data points of different classes in a high-dimensional space, with the **maximum margin** between the classes.

**How It Works (In Depth):**

- The core idea is to **maximize the margin** between two classes — the distance between the hyperplane and the nearest points (called **support vectors**). These support vectors are critical to defining the decision boundary.
- SVM can handle **linearly separable** and **non-linearly separable** data. For non-linear cases, it applies the **kernel trick** to map data into higher-dimensional spaces where separation is possible.
- Common **kernel functions** include:
  - `linear`: for linearly separable data
  - `poly`: polynomial kernel, useful when decision boundaries are curved
  - `rbf`: radial basis function, the most commonly used for non-linear data
  - `sigmoid`: behaves like a neural network's activation
- **Soft Margin SVM** allows some misclassifications to improve generalization on unseen data. This is controlled using the `C` parameter.
- The optimization behind SVM is formulated as a **convex quadratic programming** problem, ensuring a global minimum.
- It also supports **probability estimates** (though computationally more expensive), and handles **imbalanced datasets** by adjusting class weights.

**Key Hyperparameters in scikit-learn's `SVC`:**

| Parameter | Description |
|---|---|
| `C` | Regularization strength. Smaller values allow wider margin (more tolerance). |
| `kernel` | Kernel type to be used. Default is `'rbf'`. |
| `gamma` | Defines influence of a single training example. Lower = far, higher = close. |
| `degree` | Degree for polynomial kernel (used if `kernel='poly'`). |
| `class_weight` | Adjusts weights for imbalanced datasets. `'balanced'` auto-adjusts. |
| `probability` | If `True`, enables probability estimates via Platt scaling. |

## Practical Implementation using scikit-learn:

```python
from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

# Pipeline: Scaling is important for SVM performance
svm_pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('svm', SVC(
        C=1.0,
        kernel='rbf',
        gamma='scale',
        probability=True,
        class_weight='balanced'
    ))
])

# Train the model
svm_pipeline.fit(X_train, y_train)

# Predict
y_pred = svm_pipeline.predict(X_test)
```

### Hyperparameter Tuning with Grid Search:

```python
from sklearn.model_selection import GridSearchCV

param_grid = {
    'svm__C': [0.1, 1, 10, 100],
    'svm__gamma': ['scale', 'auto', 0.1, 0.01],
    'svm__kernel': ['rbf', 'poly']
}

grid = GridSearchCV(svm_pipeline, param_grid, cv=5)
grid.fit(X_train, y_train)

print("Best parameters:", grid.best_params_)
```

## Neural Networks

**Definition**: Neural Networks are computational models inspired by the human brain's structure, consisting of interconnected nodes (neurons) organized in layers that transform input data into desired outputs through a process of learning from examples.

**How it works**:

- Organizes artificial neurons in layers: input, hidden, and output
- Each connection between neurons has an associated weight
- Each neuron computes weighted sum of inputs, applies activation function

- Forward propagation passes inputs through the network to generate predictions
- Backpropagation calculates gradients of the error with respect to weights
- Optimization algorithms (like gradient descent) adjust weights to minimize error
- Deep networks have multiple hidden layers to learn hierarchical representations

**Key Parameters**:

- `hidden_layer_sizes`: Number and size of hidden layers (tuple)
- `activation`: Activation function ('relu', 'tanh', 'logistic', 'identity')
- `solver`: Weight optimization algorithm ('adam', 'sgd', 'lbfgs')
- `alpha`: L2 regularization parameter
- `learning_rate`: Learning rate schedule ('constant', 'invscaling', 'adaptive')
- `max_iter`: Maximum number of iterations
- `batch_size`: Size of minibatches for stochastic optimizers
- `early_stopping`: Whether to use early stopping to terminate training when validation score stops improving

**Implementation**:

```python
from sklearn.neural_network import MLPClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

# Create pipeline with scaling
nn_pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('mlp', MLPClassifier(
        hidden_layer_sizes=(100, 50),
        activation='relu',
        solver='adam',
        alpha=0.0001,
        batch_size='auto',
        learning_rate='adaptive',
        max_iter=1000,
        early_stopping=True,
        validation_fraction=0.1,
        random_state=42
    ))
])

# Train model
nn_pipeline.fit(X_train, y_train)

# Make predictions
y_pred = nn_pipeline.predict(X_test)

# Learning curves
plt.figure(figsize=(10, 6))
plt.plot(nn_pipeline.named_steps['mlp'].loss_curve_)
plt.title('Learning Curve')
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.grid(True)
```

## Linear Regression

**Definition**: Linear Regression is a linear approach to modeling the relationship between a dependent variable and one or more independent variables, finding the linear function that minimizes the sum of squared differences between observed and predicted values.

**How it works**:

- Models relationship as $y = \beta_0 + \beta_1x_1 + \beta_2x_2 + \ldots + \beta_nx_n + \varepsilon$
- Finds coefficients (β) that minimize the sum of squared errors
- Can be solved analytically using Ordinary Least Squares (OLS)
- Can incorporate regularization to prevent overfitting (Ridge, Lasso)
- Neural network equivalent is a single neuron with no activation function

**Key Parameters**:

- `fit_intercept`: Whether to calculate the intercept ($\beta_0$)
- `normalize`: Whether to normalize features (deprecated in newer versions)
- `copy_X`: Whether to copy X
- `n_jobs`: Number of parallel jobs for computation
- For regularized variants:
  - `alpha`: Regularization strength
  - `l1_ratio`: Mix ratio for ElasticNet (0 = Ridge, 1 = Lasso)

**Implementation**:

```
from sklearn.linear_model import LinearRegression, Ridge, Lasso
from sklearn.metrics import mean_squared_error, r2_score
import numpy as np

# Create and train standard linear regression model
lin_reg = LinearRegression(fit_intercept=True)
lin_reg.fit(X_train, y_train)

# Make predictions
y_pred = lin_reg.predict(X_test)

# Evaluate
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
r2 = r2_score(y_test, y_pred)

print(f"RMSE: {rmse:.4f}")
print(f"R²: {r2:.4f}")

# Feature importance
coefficients = pd.DataFrame({
    'Feature': X_train.columns,
    'Coefficient': lin_reg.coef_
}).sort_values('Coefficient', ascending=False)

# With regularization (Ridge)
ridge = Ridge(alpha=1.0)
ridge.fit(X_train, y_train)
```

## Unsupervised Learning

### K-Means Clustering

**Definition**: K-Means is an unsupervised learning algorithm that partitions n observations into k clusters, where each observation belongs to the cluster with the nearest mean, minimizing within-cluster variances.

**How it works**:

- Randomly initializes k cluster centroids
- Iteratively performs two steps until convergence:
    1. Assignment: Assign each data point to the nearest centroid
    2. Update: Recalculate centroids as the mean of all points in the cluster
- Aims to minimize the sum of squared distances between points and their assigned centroids
- Final result depends on initial centroid positions (may find local optimum)
- Uses distance metrics (usually Euclidean) to determine similarity

**Key Parameters**:

- `n_clusters`: Number of clusters (k)
- `init`: Method for initialization ('k-means++', 'random', or an array)
- `n_init`: Number of times algorithm runs with different centroid seeds
- `max_iter`: Maximum number of iterations per run
- `tol`: Tolerance for declaring convergence
- `algorithm`: Implementation algorithm ('auto', 'full', 'elkan')

**Implementation**:

```
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import silhouette_score
import matplotlib.pyplot as plt

# Scale data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Create and fit model
kmeans = KMeans(
    n_clusters=3,
    init='k-means++',
    n_init=10,
```

```python
    max_iter=300,
    random_state=42
)
clusters = kmeans.fit_predict(X_scaled)

# Evaluate using silhouette score
silhouette_avg = silhouette_score(X_scaled, clusters)
print(f"Silhouette Score: {silhouette_avg:.4f}")

# Finding optimal k using elbow method
inertia = []
range_k = range(1, 11)
for k in range_k:
    kmeans = KMeans(n_clusters=k, random_state=42)
    kmeans.fit(X_scaled)
    inertia.append(kmeans.inertia_)

plt.figure(figsize=(10, 6))
plt.plot(range_k, inertia, marker='o')
plt.xlabel('Number of clusters')
plt.ylabel('Inertia')
plt.title('Elbow Method')
plt.grid(True)
```

## Principal Component Analysis (PCA)

#### Definition

Principal Component Analysis (PCA) is an unsupervised learning technique used for **dimensionality reduction**. It transforms the original features into a new set of uncorrelated variables called **principal components**, ordered by the amount of variance they capture from the original data. The main goal is to simplify the dataset, reduce redundancy, and retain the most important information.

#### How it Works

1. **Standardization**:
   PCA starts by **standardizing** the dataset — each feature is centered (mean = 0) and scaled (unit variance). This is crucial because PCA is affected by the scale of the variables.

2. **Covariance Matrix Calculation**:
   It then calculates the **covariance matrix** of the data, which measures how much the variables vary together.

3. **Eigendecomposition**:
   The covariance matrix is decomposed into **eigenvectors** and **eigenvalues**:

   - **Eigenvectors** → directions (axes) of the new feature space (principal components)
   - **Eigenvalues** → magnitude of variance carried by each principal component

4. **Component Selection**:
   The eigenvectors are sorted based on their corresponding eigenvalues (variance), and the top `k` components are selected that together retain a desired proportion of the total variance (e.g., 95%).

5. **Projection**:
   The data is projected onto the selected principal components, effectively reducing the number of dimensions while preserving the structure that contributes most to its variance.

**Key Parameters in `sklearn.decomposition.PCA`**

- `n_components`:
  Specifies the number of principal components to keep.
  Can be:
  - An integer (e.g., `5`)
  - A float (e.g., `0.95` to keep 95% of the variance)
  - `'mle'` to use Minka's MLE for dimensionality estimation

- `svd_solver`:
  Determines the algorithm used for Singular Value Decomposition (SVD):
  `'auto'`, `'full'`, `'arpack'`, `'randomized'`

- `whiten`:
  If `True`, it scales the principal components to have unit variance (removes correlations between them)

- `random_state`:
  Seed for reproducibility when using randomized solver

- `tol`:
  Tolerance for small singular values (used when filtering components)

---

**Implementation (with scikit-learn)**

```python
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
import seaborn as sns

# Step 1: Standardize the data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Step 2: Fit PCA model (retain 95% of variance)
pca = PCA(n_components=0.95)
X_pca = pca.fit_transform(X_scaled)

# Step 3: Analyze explained variance
explained_variance = pca.explained_variance_ratio_
cumulative_variance = explained_variance.cumsum()
```

```
print(f"Number of components selected: {pca.n_components_}")
print(f"Total explained variance: {sum(explained_variance):.4f}")

# Step 4: Plot individual and cumulative explained variance
plt.figure(figsize=(10, 6))
plt.bar(range(1, len(explained_variance) + 1), explained_variance, alpha=0.6,
label='Individual')
plt.plot(range(1, len(cumulative_variance) + 1), cumulative_variance, marker='o',
color='red', label='Cumulative')
plt.xlabel('Principal Components')
plt.ylabel('Explained Variance Ratio')
plt.title('Explained Variance by PCA Components')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

# Step 5: Visualize data on first two principal components
if X_pca.shape[1] >= 2:
    plt.figure(figsize=(10, 8))
    sns.scatterplot(x=X_pca[:, 0], y=X_pca[:, 1], hue=y if 'y' in locals() else
None)
    plt.xlabel('First Principal Component')
    plt.ylabel('Second Principal Component')
    plt.title('PCA: First Two Principal Components')
    plt.grid(True)
    plt.tight_layout()
    plt.show()
```