Naive Bayes

The probability of an event occurring given that another event occurred

Naive Bayes gets its name because we assume that each feature, and its resulting likelihood, is independent. This "naive" assumption is frequently wrong, yet in practice does little to prevent building high quality classifiers.

The most common type of naive Bayes classifier is the Gaussian naive Bayes. In Gaussian naive Bayes, we assume that the likelihood of the feature values, x, given an observation is of class y, follows a normal distribution

Model Evaluation

K-fold Cross-validation is the best way to evaluate a model. In KFCV, we split the data into k parts called "folds." The model is then trained using k – 1 folds—combined into one training set—and then the last fold is used as a test set. We repeat this k times, each time using a different fold as the test set. The performance on the model for each of the k iterations is then averaged to produce an overall measurement.

```
# Load libraries
from sklearn import datasets
from sklearn import metrics
from sklearn.model_selection import KFold, cross_val_score
from sklearn.pipeline import make_pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler

# Load digits dataset
digits = datasets.load_digits()

# Create features matrix
features = digits.data

# Create target vector
target = digits.target

# Create standardizer
standardizer = StandardScaler()

# Create logistic regression object
logit = LogisticRegression()

# Create a pipeline that standardizes, then runs logistic regression
pipeline = make_pipeline(standardizer, logit)

# Create k-Fold cross-validation
```

```
kf = KFold(n_splits=10, shuffle=True, random_state=1)

# Conduct k-fold cross-validation
cv_results = cross_val_score(pipeline, # Pipeline
                features, # Feature matrix
                target, # Target vector
                cv=kf, # Cross-validation technique
                scoring="accuracy", # Loss function
                n_jobs=-1) # Use all CPU scores

# Calculate mean
cv_results.mean()
0.96493171942892597
```

Model Selection

GridSearchCV is a brute-force approach to model selection using cross-validation. Specifically, a user defines sets of possible values for one or multiple hyperparameters, and then GridSearchCV trains a model using every value and/or combination of values. The model with the best performance score is selected as the best model.

```
# Load libraries
import numpy as np
from sklearn import linear_model, datasets
from sklearn.model_selection import GridSearchCV

# Load data
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Create logistic regression
logistic = linear_model.LogisticRegression()

# Create range of candidate penalty hyperparameter values
penalty = ['l1', 'l2']

# Create range of candidate regularization hyperparameter values
C = np.logspace(0, 4, 10)

# Create dictionary hyperparameter candidates
hyperparameters = dict(C=C, penalty=penalty)

# Create grid search
gridsearch = GridSearchCV(logistic, hyperparameters, cv=5, verbose=0)

# Fit grid search
best_model = gridsearch.fit(features, target)
```

Unsupervised Learning

Clustering

The goal of clustering algorithms is to identify those latent groupings of observations, which if done well, allow us to predict the class of observations even without a target vector.

K-means Clustering

In k-means clustering, the algorithm attempts to group observations into k groups, with each group having roughly equal variance. The number of groups, k, is specified by the user as a hyperparameter. Specifically, in k-means:

1. **k cluster "center" points are created at random locations.**
2. **For each observation: a. The distance between each observation and the k center points is calculated. b. The observation is assigned to the cluster of the nearest center point.**
3. **The center points are moved to the means (i.e., centers) of their respective clusters.**
4. **Steps 2 and 3 are repeated until no observation changes in cluster membership.**

Hierarchical Clustering

Agglomerative clustering is a powerful, flexible hierarchical clustering algorithm. In agglomerative clustering, all observations start as their own clusters. Next, clusters meeting some criteria are merged together. This process is repeated, growing clusters until some end point is reached.

```
# Load libraries
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import AgglomerativeClustering
# Load data
iris = datasets.load_iris()
features = iris.data
# Standardize features
scaler = StandardScaler()
features_std = scaler.fit_transform(features)
# Create meanshift object
cluster = AgglomerativeClustering(n_clusters=3)
# Train model
model = cluster.fit(features_std)
```

Dimensionality Reduction

The goal of Dimensionality Reduction is to reduce the number of features with only a small loss in our data's ability to generate high-quality predictions(e.g., the number of features).

Principal Component Analysis PCA

Principal component analysis (PCA) is a popular linear dimensionality reduction technique. PCA projects observations onto the (hopefully fewer) principal components of the feature matrix that retain the most variance. PCA is an unsupervised technique, meaning that it does not use the information from the target vector and instead only considers the feature matrix.

```
# Load libraries
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn import datasets

# Load the data
digits = datasets.load_digits()

# Standardize the feature matrix
features = StandardScaler().fit_transform(digits.data)

# Create a PCA that will retain 99% of variance
pca = PCA(n_components=0.99, whiten=True)

# Conduct PCA
features_pca = pca.fit_transform(features)

# Show results
print("Original number of features:", features.shape[1])
print("Reduced number of features:", features_pca.shape[1])

Original number of features: 64
Reduced number of features: 54
```

Kernel PCA

Kernel PCA is an extension of principal component analysis that uses kernels to allow for non- linear dimensionality reduction.

```
# Load libraries
from sklearn.decomposition import PCA, KernelPCA
from sklearn.datasets import make_circles

# Create linearly inseparable data
```

```
features, _ = make_circles(n_samples=1000, random_state=1, noise=0.1, factor=0.1)

# Apply kernal PCA with radius basis function (RBF) kernel
kpca = KernelPCA(kernel="rbf", gamma=15, n_components=1)
features_kpca = kpca.fit_transform(features)

print("Original number of features:", features.shape[1])
print("Reduced number of features:", features_kpca.shape[1])

Original number of features: 2
Reduced number of features: 1
```

Xgboost

The most powerful implementation of gradient boosting in terms of model performance and speed

In Boosting, we iteratively train a series of weak models (most often a shallow decision tree, sometimes called a stump), each iteration giving higher priority to observations the previous model predicted incorrectly

Steps to Choosing the Best Algorithm for your Project

1. **Step 1:** ***Determine whether your problem is a Supervised, Unsupervised or Reinforcement Learning problem***
2. **Step 2:** ***Determine whether your problem is a Classification, Regression or Clustering problem***
3. **Step 3:** ***Determine whether your problem is a linear or non-linear problem (use GridSearch)***
4. **Step 4:** ***Try Xgboost (it conquers all)***
5. **Step 5:** ***If you have a lot of Data and Computing Power then use Deep Learning Algorithms***

Boadzie Daniel(AI ENGINEER)