k-Nearest Neighbors (kNN)

This Jupyter notebook summarizes the Pros and Cons of the k-Nearest Neighbors algorithm and gives two Python examples on usage for Classification and Regression.

Theory[1,2,3]

- Is a non-probabilistic, non-parametric and instance-based learning algorithm (see References:
  - **Non-parametric** means it makes no explicit assumptions about the function form of *h*, avoiding the dangers of mis-modelling the underlying distribution of the data
    - For example, suppose our data is highly non-Gaussian but the learning model was choose assumes a Gaussian form. In that case, a parametric algorithm would make extremely poor predictions.
  - **Instance-based** learning means that the algorithm does not explicitly learn a model
    - Instead, it chooses to memorize the training instances which are subsequently used as "knowledge" for the prediction phase
    - Concretely, this means that only when a query to our database is made (i.e., when we ask it to predict a label given an input), will the algorithm use the training instances to predict the result

Pros

- **simple** to understand and implement
- with **little to zero training time**
- kNN **works just as easily with multi-class data** sets whereas other algorithms are hard-coded for the binary setting
- the non-parametric nature of kNN gives it an edge in certain settings where the data may be highly unusual, thus **without prior knowledge on distribution**

Cons

- **computationally expensive** testing phase
  - we **need to store the whole data set for each decision**!
- can **suffer from skewed class distributions**
  - for example, if a certain class is very frequent in the training set, it will tend to dominate the majority voting of the new example (large number = more common)
- the accuracy can be severally **degraded with high-dimension data** because of the little difference between the nearest and farthest neighbor

- **the curse of dimensionality** refers to various phenomena that arise when analyzing and organizing data in high-dimensional spaces that do not occur in low-dimensional settings such as the three-dimensional physical space of everyday experience
- for high-dimensional data (e.g., with number of dimensions more than 10) **scaling** and **dimension reductions** (such as PCA) is usually performed prior applying kNN

References

- [1]Wikipedia kNN, Curse of dimensionality
- [2]Sklearn KNeighborsClassifier, KNeighborsRegressor
- [3]Complete Guide to K-Nearest-Neighbors

Classification

- the output is a class membership
- an object is classified by a **majority vote** of its neighbours, with the object being assigned to the class most common among its k nearest neighbours
  - if k = 1, then the object is simply assigned to the class of that nearest neighbour

Example: predict IRIS class

Set environment

```
# Scikit-learn
from sklearn import datasets
from sklearn.model_selection import train_test_split, GridSearchCV , cross_val_score
from sklearn.neighbors import KNeighborsClassifier,KNeighborsRegressor
from sklearn.dummy import DummyClassifier, DummyRegressor
from sklearn.metrics import classification_report, mean_squared_error,r2_score, mean_absc
from sklearn.preprocessing import StandardScaler, LabelEncoder, MinMaxScaler
from sklearn.decomposition import PCA
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_california_housing
from sklearn.feature_selection import SelectKBest, f_regression
from sklearn.datasets import load_wine
import seaborn as sns
from sklearn.datasets import load_diabetes


# Use vector drawing inside jupyter notebook
%config InlineBackend.figure_format = "svg"

# Set matplotlib default axis font size (inside this notebook)
plt.rcParams.update({'font.size': 8})
```

Load data

```
iris = datasets.load_iris()

df = pd.DataFrame(iris.data,columns=iris.feature_names)
df = df.assign(target=iris.target)
```

```
df.head()
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0

Next steps: Generate code with df | View recommended plots | New interactive sheet

Changed.

```
# Check for class distribution in target variable
print("Class distribution:")
print(df['target'].value_counts())
```

```
Class distribution:
target
0    50
1    50
2    50
Name: count, dtype: int64
```

Show data summary: extend the `describe` method by selected stats

- See the Jupyter notebook on **Standard Procedure** for more details

Changed.

```
# Compute selected stats
dfinfo = pd.DataFrame(df.dtypes,columns=["dtypes"])
for (m,n) in zip([df.count(),df.isna().sum()],["count","isna"]):
    dfinfo = dfinfo.merge(pd.DataFrame(m,columns=[n]),right_index=True,left_index=True,hc
```

```
# dfinfo.T.append(df.describe())
```

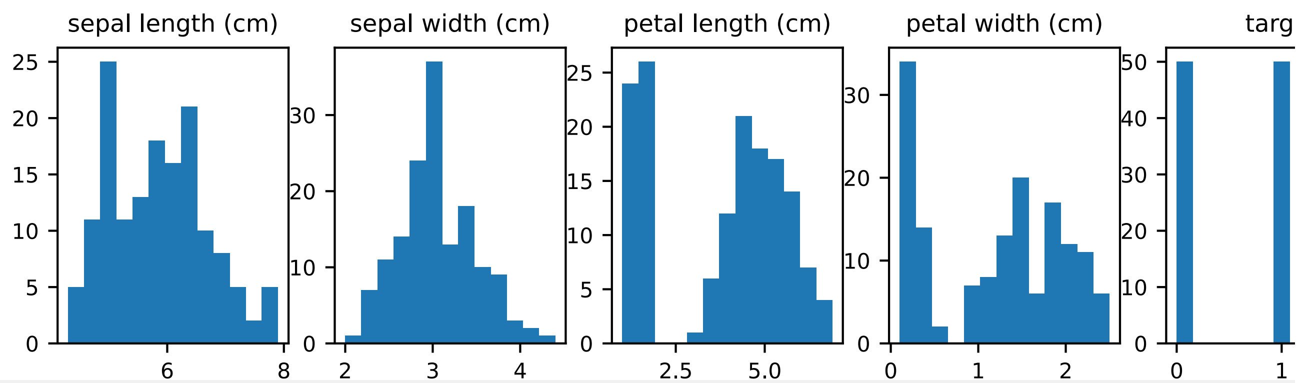
```
dfinfo = pd.concat([dfinfo.T, df.describe()])
dfinfo
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
dtypes	float64	float64	float64	float64	int64
count	150	150	150	150	150
isna	0	0	0	0	0
count	150.0	150.0	150.0	150.0	150.0
mean	5.843333	3.057333	3.758	1.199333	1.0
std	0.828066	0.435866	1.765298	0.762238	0.819232
min	4.3	2.0	1.0	0.1	0.0
25%	5.1	2.8	1.6	0.3	0.0
50%	5.8	3.0	4.35	1.3	1.0
75%	6.4	3.3	5.1	1.8	2.0
max	7.9	4.4	6.9	2.5	2.0

Next steps: Generate code with dfinfo | View recommended plots | New interactive sheet

Show histogram (distribution)

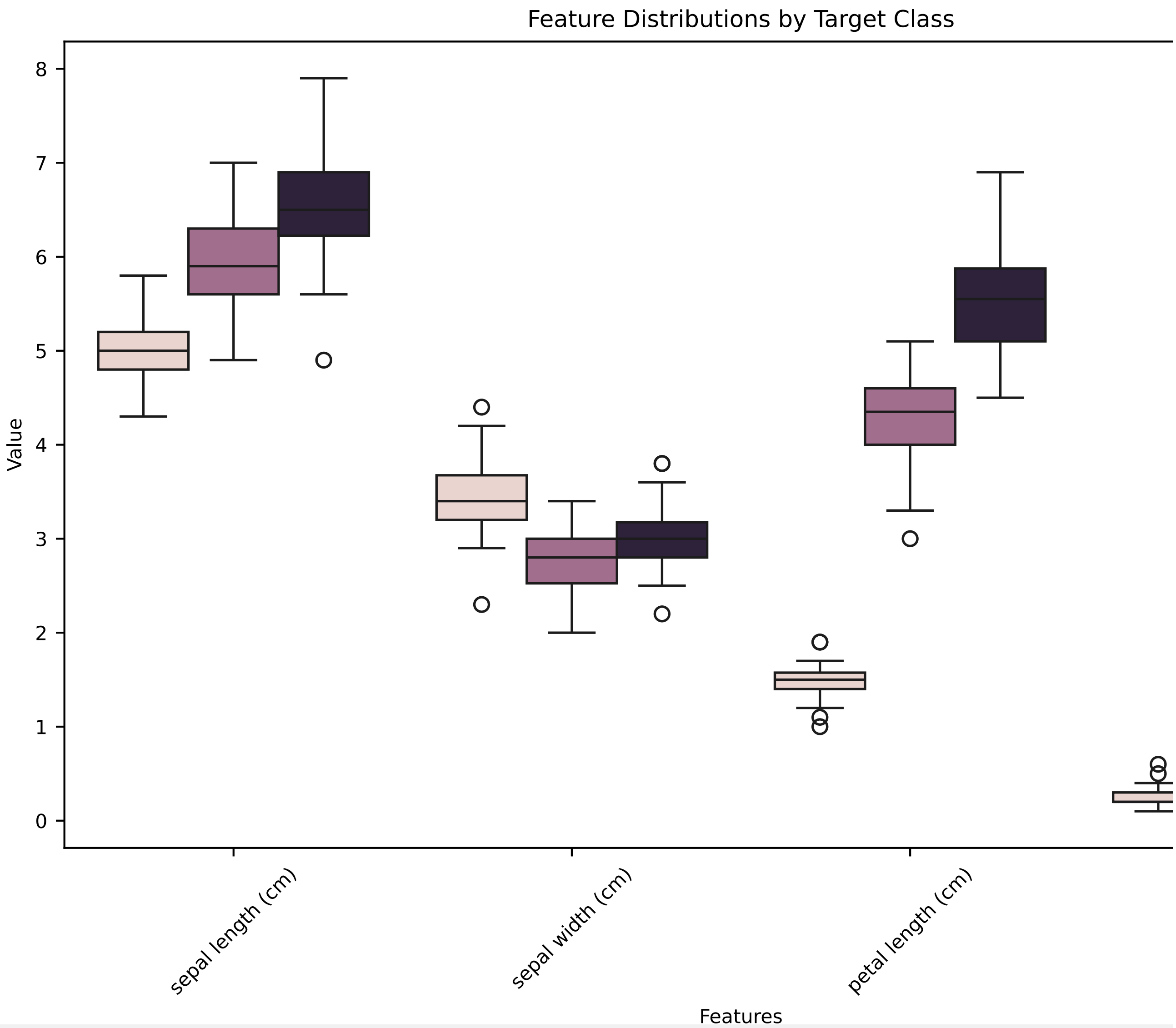
```
plt.figure(figsize=(9,2))
for (i,v) in enumerate(df.columns):
    plt.subplot(1,df.shape[1],i+1);
    plt.hist(df.iloc[:,i],bins="sqrt")
    plt.title(df.columns[i],fontsize=9);
```



Changed.

```
# Visualize feature distributions

plt.figure(figsize=(10,6))
df_melted = df.melt(id_vars="target", var_name="Features", value_name="Value")
```

```
sns.boxplot(x="Features", y="Value", hue="target", data=df_melted)
plt.xticks(rotation=45)
plt.title("Feature Distributions by Target Class")
plt.show()
```



Show correlation matrix

```
df.corr().round(2).style.background_gradient(cmap="viridis")
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
sepal length (cm)	1.000000	-0.120000	0.870000	0.820000	0.780000
sepal width (cm)	-0.120000	1.000000	-0.430000	-0.370000	-0.430000
petal length (cm)	0.870000	-0.430000	1.000000	0.960000	0.950000

Scale and try to **reduce dimensions**: what we try to do is to **always simply the model** if possible (see correlation matrix above)

- More complex model (e.g., more features, or higher *k*) will (in theory) increase the probability of higher "out of sample" error (even when "in sample" error = train set) will be smaller!
- Use either 99% threshold (own subjective) or "mle" algorithm (more objective)
- Use **linear** scaler (transformation)
- Here, the data is scaled prior train-test split.
    - In real applications, first split and scale afterwards, to simulate real-world scenario where we do not have the test set! (otherwise data snooping effect)

Changed.

Unsupported cell type. Double-click to inspect/edit the content.

```
scaler = StandardScaler()
Xo = scaler.fit_transform(df.drop(columns=["target"]))
```

```
pca = PCA(n_components=0.99)# or set n_components="mle"
X = pca.fit_transform(Xo)
print("Nr. of features after PCA = {} (input = {})".format(X.shape[1],Xo.shape[1]))
```

```
Nr. of features after PCA = 3 (input = 4)
```

Prepare for fitting

```
# encode target values (is not necessary for IRIS but still:-)
y = LabelEncoder().fit_transform(df["target"].values);

# Split 2/3 to 1/3 train to test respectively
[X_train,X_test,y_train,y_test] = train_test_split(X,y,train_size = 0.67,test_size = 0.33
```

Find optimal model

- Considering the small data set (150 samples), find "optimal" k setting it to maximum of 5
    - Optimal in terms of accuracy
    - Simple model = higher probability of lower in and out-of sample error

```
model = KNeighborsClassifier(algorithm="auto");
parameters = {"n_neighbors":[1,3,5],
              "weights":["uniform","distance"]}
model_optim = GridSearchCV(model, parameters, cv=5,scoring="accuracy");
```

```
model_optim.fit(X_train,y_train)
```

```
GridSearchCV
best_estimator_:
KNeighborsClassifier
KNeighborsClassifier
```

Show the "optimal" settings for kNN

```
model_optim.best_estimator_
```

```
KNeighborsClassifier
KNeighborsClassifier(n_neighbors=1)
```

Changed.

```
for (i,x,y) in zip(["Train","Test"],[X_train,X_test],[y_train,y_test]):
    print("Classification kNN",i," report:\n",classification_report(y,model_optim.predict
```

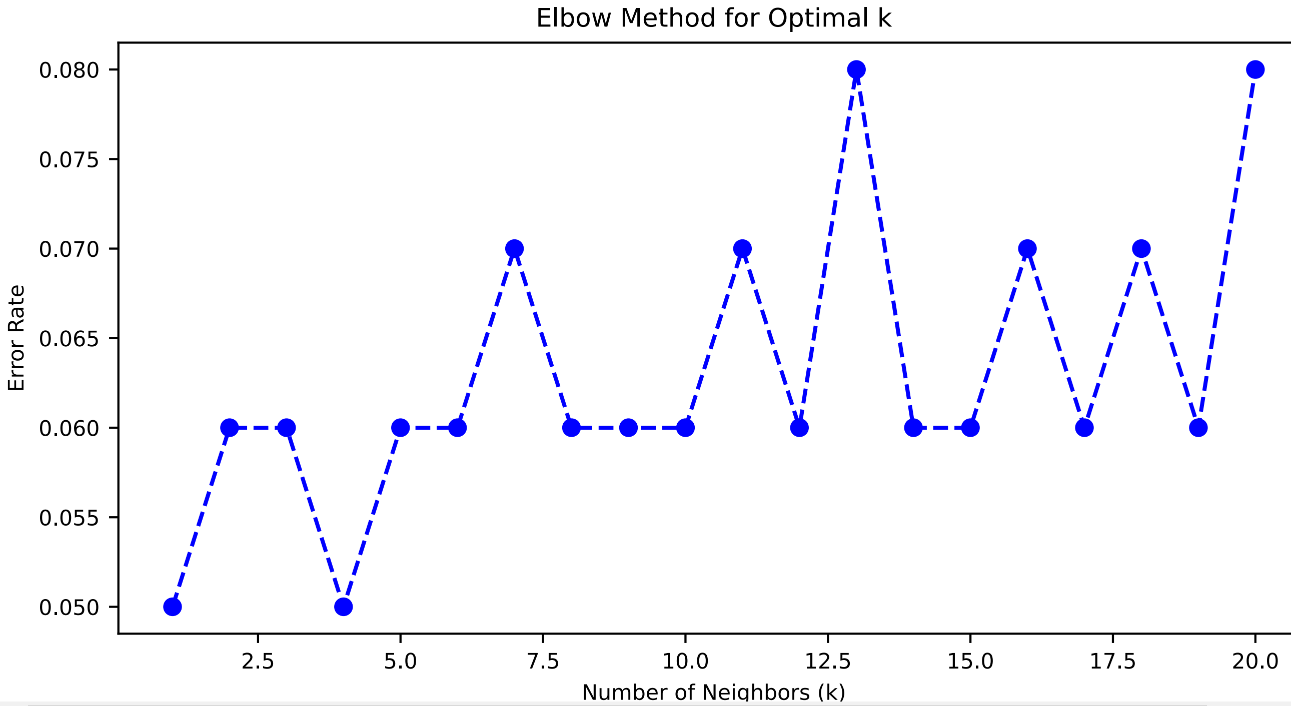
```
Classification kNN Train  report:
               precision    recall  f1-score   support

           0       1.00      1.00      1.00        34
           1       1.00      1.00      1.00        33
           2       1.00      1.00      1.00        33

    accuracy                           1.00       100
   macro avg       1.00      1.00      1.00       100
weighted avg       1.00      1.00      1.00       100

Classification kNN Test  report:
               precision    recall  f1-score   support

           0       1.00      1.00      1.00        16
           1       1.00      0.88      0.94        17
           2       0.89      1.00      0.94        17

    accuracy                           0.96        50
   macro avg       0.96      0.96      0.96        50
weighted avg       0.96      0.96      0.96        50
```

```
for i in ["most_frequent","uniform"]:
    dummy = DummyClassifier(strategy=i).fit(X_train,y_train);
    print("Classification ",i," test report:",classification_report(y_test,dummy.predict(
```

```
Classification  most_frequent  test report:              precision    recall  f1-scc
```

```
              0       0.32      1.00      0.48        16
              1       0.00      0.00      0.00        17
              2       0.00      0.00      0.00        17

       accuracy                           0.32        50
      macro avg       0.11      0.33      0.16        50
   weighted avg       0.10      0.32      0.16        50

Classification  uniform  test report:              precision    recall  f1-score   s

              0       0.47      0.50      0.48        16
              1       0.33      0.41      0.37        17
              2       0.17      0.12      0.14        17

       accuracy                           0.34        50
      macro avg       0.32      0.34      0.33        50
   weighted avg       0.32      0.34      0.33        50

/usr/local/lib/python3.11/dist-packages/sklearn/metrics/_classification.py:1565: Unde
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.11/dist-packages/sklearn/metrics/_classification.py:1565: Unde
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.11/dist-packages/sklearn/metrics/_classification.py:1565: Unde
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
```

```
# Try different values of k
k_values = range(1, 21)
error_rates = []

for k in k_values:
    knn = KNeighborsClassifier(n_neighbors=k)
    scores = cross_val_score(knn, X_train, y_train, cv=5, scoring='accuracy')
    error_rates.append(1 - scores.mean())  # Convert accuracy to error rate

# Plot the Elbow Method
plt.figure(figsize=(8,4))
plt.plot(k_values, error_rates, marker='o', linestyle='dashed', color='b')
plt.xlabel('Number of Neighbors (k)')
plt.ylabel('Error Rate')
plt.title('Elbow Method for Optimal k')
plt.show()
```




```
# Choose best k from the elbow point
optimal_k = k_values[np.argmin(error_rates)]
print(f"Optimal k found: {optimal_k}")
```

```
Optimal k found: 1
```

Show resulting accuracy

In this case, the precision (accuracy=macro avg precision) is very high. Just to show that that is not coincidence compare to "dummy" model (most frequent & uniform distribution)

Regression

- Predicts value as the **average of the values** of its k nearest neighbors

Example: Predict House price

- [Use Scikit-learn](#) [California Housing](#) data set
  - This is a large data set that allows us to use more complex model
- Nontheless, try to reduce the number of features: via visual inspection and using PCA

Load data

Changed.

Unsupported cell type. Double-click to inspect/edit the content.

```
house = datasets.fetch_california_housing()
df = pd.DataFrame(house.data,columns=house.feature_names)
df = df.assign(target=house.target)
```

```
df.head()
```

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude
0	8.3252	41.0	6.984127	1.023810	322.0	2.555556	37.88	-122.23
1	8.3014	21.0	6.238137	0.971880	2401.0	2.109842	37.86	-122.22
2	7.2574	52.0	8.288136	1.073446	496.0	2.802260	37.85	-122.24
3	5.6431	52.0	5.817352	1.073059	558.0	2.547945	37.85	-122.25
4	3.8462	52.0	6.281853	1.081081	565.0	2.181467	37.85	-122.25

Next steps: Generate code with df | View recommended plots | New interactive sheet

Inspect data: show statistics, histogram and correlation

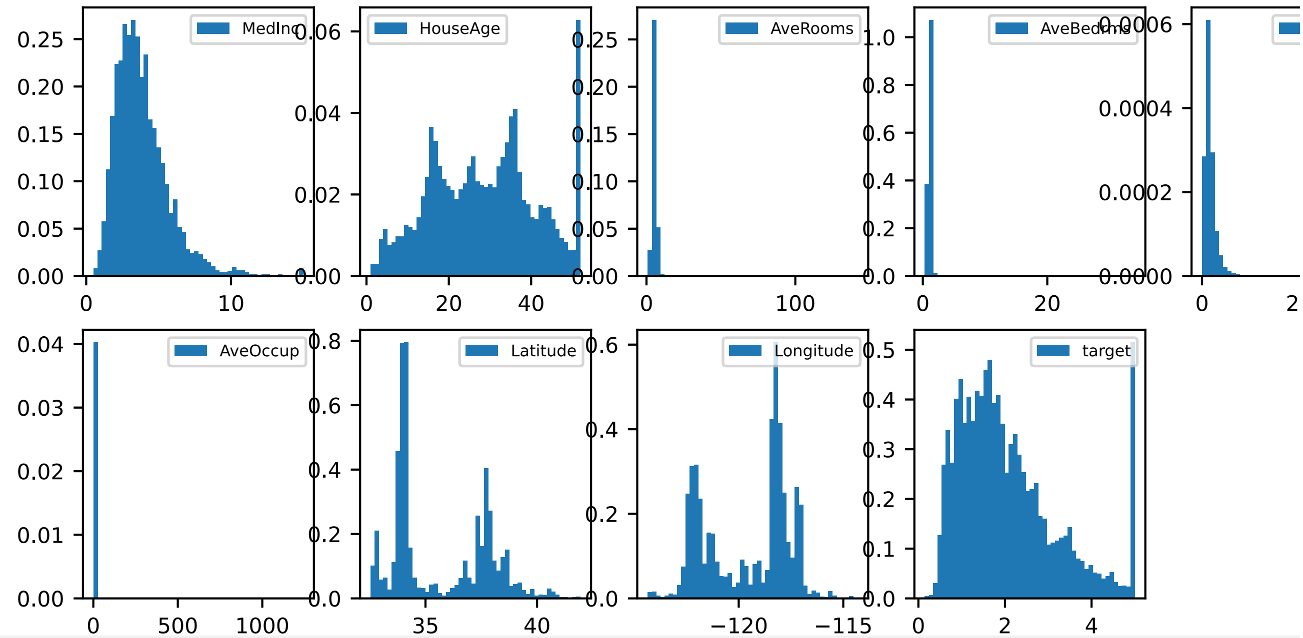
Changed.

```
# Compute selected stats
dfinfo = pd.DataFrame(df.dtypes,columns=["dtypes"])
for (m,n) in zip([df.count(),df.isna().sum()],["count","isna"]):
    dfinfo = dfinfo.merge(pd.DataFrame(m,columns=[n]),right_index=True,left_index=True,hc

#dfinfo.T.append(df.describe())
dfinfo = pd.concat([dfinfo.T, df.describe()])
dfinfo
```

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude
dtypes	float64	float64	float64	float64	float64	float64	float64
count	20640	20640	20640	20640	20640	20640	20640
isna	0	0	0	0	0	0	0
count	20640.0	20640.0	20640.0	20640.0	20640.0	20640.0	20640.0
mean	3.870671	28.639486	5.429	1.096675	1425.476744	3.070655	35.631861
std	1.899822	12.585558	2.474173	0.473911	1132.462122	10.38605	2.135952
min	0.4999	1.0	0.846154	0.333333	3.0	0.692308	32.54
25%	2.5634	18.0	4.440716	1.006079	787.0	2.429741	33.93
50%	3.5348	29.0	5.229129	1.04878	1166.0	2.818116	34.26
75%	4.74325	37.0	6.052381	1.099526	1725.0	3.282261	37.71
max	15.0001	52.0	141.909091	34.066667	35682.0	1243.333333	41.95

Next steps: Generate code with dfinfo View recommended plots New interactive sheet

```
plt.figure(figsize=(9,4))
for (i,v) in enumerate(df.columns):
    plt.subplot(2,5,i+1);
    plt.hist(df.iloc[:,i],50,density=True)
    plt.legend([df.columns[i]],fontsize=6);
```




```
df.corr().round(2).style.background_gradient(cmap="viridis")
```

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude
MedInc	1.000000	-0.120000	0.330000	-0.060000	0.000000	0.020000	-0.080000
HouseAge	-0.120000	1.000000	-0.150000	-0.080000	-0.300000	0.010000	0.010000
AveRooms	0.330000	-0.150000	1.000000	0.850000	-0.070000	-0.000000	0.110000
AveBedrms	-0.060000	-0.080000	0.850000	1.000000	-0.070000	-0.010000	0.070000
Population	0.000000	-0.300000	-0.070000	-0.070000	1.000000	0.070000	-0.110000
AveOccup	0.020000	0.010000	-0.000000	-0.010000	0.070000	1.000000	0.000000
Latitude	-0.080000	0.010000	0.110000	0.070000	-0.110000	0.000000	1.000000
Longitude	-0.020000	-0.110000	-0.030000	0.010000	0.100000	0.000000	-0.920000
target	0.690000	0.110000	0.150000	-0.050000	-0.020000	-0.020000	-0.140000

Prepare for fitting by scaling data set

- Here, the data is scaled prior train-test split.
  - In real applications, first split and scale afterwards, to simulate real-world scenario where we do not have the test set!

```
X = StandardScaler().fit_transform(df.drop("target",axis=1).values);
y = df.target.values
```

Supervised Reduction

- Considering the correlation, histogram and the summary table:
  - Remove/drop "AveOccup" (average house occupancy)

Changed.

```
#df = df.drop(["AveOccup"],axis=1)
```

```
X = df.drop(columns=["target"])
y = df["target"]

selector = SelectKBest(score_func=f_regression, k=5)  # Select top 5 features
X_selected = selector.fit_transform(X, y)
selected_features = X.columns[selector.get_support()]
print("Selected Features:", selected_features)
```

```
Selected Features: Index(['MedInc', 'HouseAge', 'AveRooms', 'AveBedrms', 'Latitude'],
```

Changed.

Unsupported cell type. Double-click to inspect/edit the content.

```
#PCA considering selected features

selected_feature_names = ['MedInc', 'HouseAge', 'AveRooms', 'AveBedrms', 'Latitude']
X_selected = df[selected_feature_names]

pca = PCA(n_components="mle")
X = pca.fit_transform(X_selected)

#Print the number of features after PCA
print(f"Number of features after PCA: {X.shape[1]} (input = {X_selected.shape[1]})")
```

```
Number of features after PCA: 4 (input = 5)
```

Fit model

```
[X_train,X_test,y_train,y_test] = train_test_split(X,y,train_size=0.67,test_size=0.33,ran
```

```
knn = KNeighborsRegressor();

parameters = {"n_neighbors":[1,3,5,7,9],"weights":["uniform","distance"]}

knn_reg = GridSearchCV(knn, parameters, cv=5, scoring="neg_mean_squared_error");
```

```
knn_reg.fit(X_train,y_train)
```

```
GridSearchCV
best_estimator_:
KNeighborsRegressor
KNeighborsRegressor
```

```
knn_reg.best_estimator_
```

```
KNeighborsRegressor
KNeighborsRegressor(n_neighbors=9)
```

Changed.

```
# from sklearn.metrics import r2_score, mean_absolute_error, mean_squared_error
```

```
rmse = np.sqrt(mean_squared_error(knn_reg.predict(X_test),y_test))
print(f"Root Mean Squared Error (RMSE): {rmse:.4f}")
print(f"R² Score: {r2_score(knn_reg.predict(X_test),y_test):.4f}")
print(f"Mean Absolute Error: {mean_absolute_error(knn_reg.predict(X_test),y_test):.4f}")
print(f"Mean Squared Error: {mean_squared_error(knn_reg.predict(X_test),y_test):.4f}")

print("Regression kNN (test) RMSE \t= {:.0f} *1000$".format(
    100*np.sqrt(mean_squared_error(knn_reg.predict(X_test),y_test))))
```

```
Root Mean Squared Error (RMSE): 0.7430
R² Score: 0.3047
Mean Absolute Error: 0.5388
Mean Squared Error: 0.5520
Regression kNN (test) RMSE      = 74 *1000$
```

Changed.

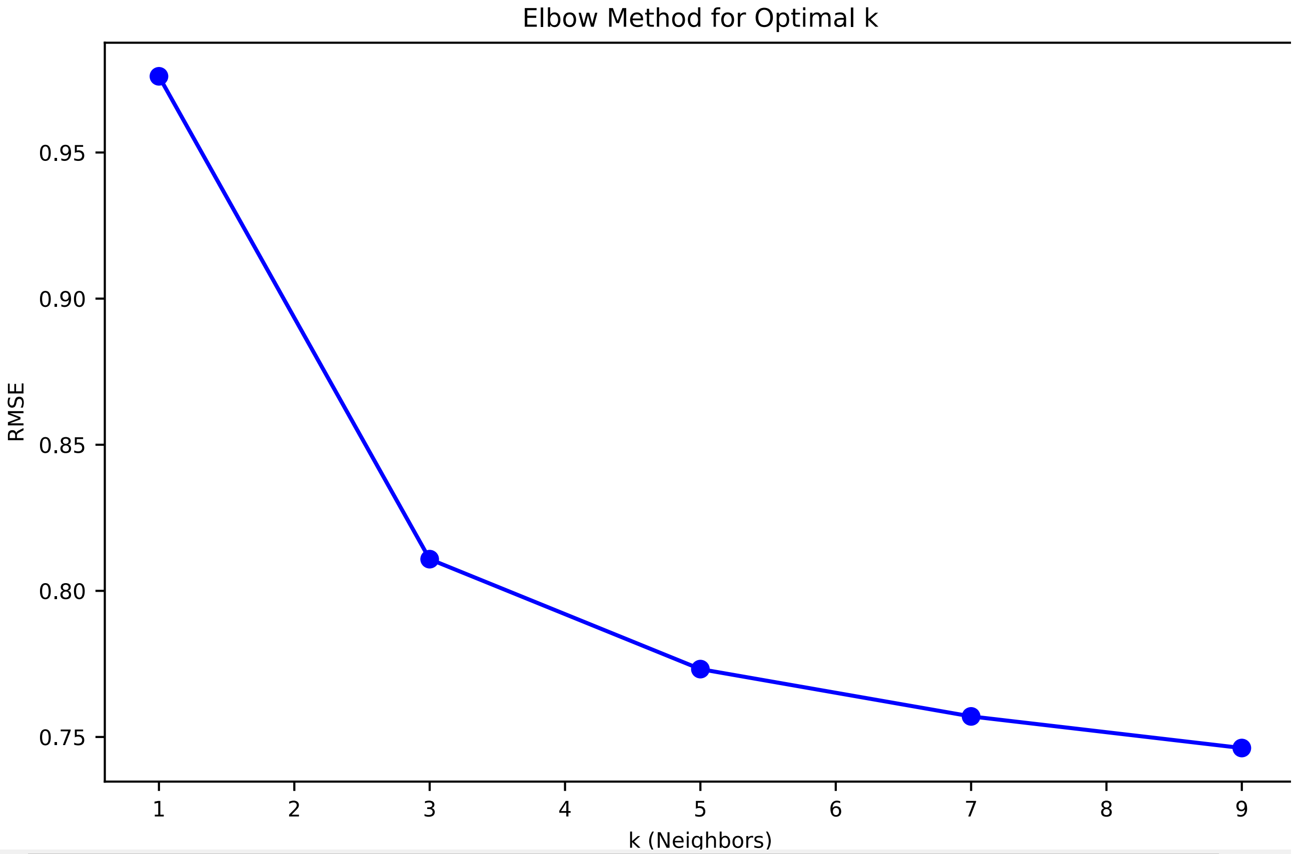
```
knn = KNeighborsRegressor()

param_grid = {
    "n_neighbors": list(range(1, 5, 1)),
    "weights": ["uniform", "distance"],
    "p": [1, 2]
}

grid_search = GridSearchCV(knn, param_grid, cv=5, scoring="r2")
grid_search.fit(X_train, y_train)

# Get best parameters
print("Best K:", grid_search.best_params_["n_neighbors"])
print("Best Weights:", grid_search.best_params_["weights"])
```

```
Best K: 4
Best Weights: uniform
```

Changed.

```
#import matplotlib.pyplot as plt
#from sklearn.neighbors import KNeighborsRegressor
#from sklearn.metrics import mean_squared_error

rmse_list = []
k_values = list(range(1, 10, 2))


for k in k_values:
    knn = KNeighborsRegressor(n_neighbors=k, weights="distance")
    knn.fit(X_train, y_train)
    y_pred = knn.predict(X_test)

    # Compute RMSE
    rmse = np.sqrt(mean_squared_error(y_test, y_pred))
    rmse_list.append(rmse)
```

```
plt.figure(figsize=(8, 5))
plt.plot(k_values, rmse_list, marker="o", linestyle="-", color="blue")
plt.xlabel("k (Neighbors)")
plt.ylabel("RMSE")
plt.title("Elbow Method for Optimal k")
plt.show()
```



Changed.

Application of kNN - Wine dataset

Importing libraries

import numpy as np import pandas as pd from sklearn.datasets import load_wine from sklearn.model_selection import train_test_split from sklearn.neighbors import KNeighborsClassifier from sklearn.metrics import accuracy_score, classification_report, confusion_matrix from sklearn.preprocessing import StandardScaler from sklearn.model_selection import GridSearchCV, cross_val_score

```
# Load the Wine dataset

data = load_wine()
df = pd.DataFrame(data.data, columns=data.feature_names)
y = data.target


# Split, train and predict k-NN model

# Splitting the dataset
X_train, X_test, y_train, y_test = train_test_split(df, y, test_size=0.2, random_state=42

# Train a basic k-NN
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train, y_train)

# Predictions
y_pred = knn.predict(X_test)


# Performance Evaluation
accuracy = accuracy_score(y_test, y_pred)
print(f"Initial Model Accuracy: {accuracy:.4f}")
print("Classification Report:\n", classification_report(y_test, y_pred))
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
```

```
Initial Model Accuracy: 0.7500
Classification Report:
               precision    recall  f1-score   support

           0       0.86      1.00      0.92        12
           1       0.75      0.64      0.69        14
           2       0.60      0.60      0.60        10

    accuracy                           0.75        36
   macro avg       0.74      0.75      0.74        36
weighted avg       0.74      0.75      0.74        36

Confusion Matrix:
 [[12  0  0]
 [ 1  9  4]
 [ 1  3  6]]
```

```
# Feature Scaling
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)


# Hyperparameter tuning using GridSearchCV
param_grid = {'n_neighbors': range(1, 31), 'weights': ['uniform', 'distance']}
grid_search = GridSearchCV(KNeighborsClassifier(), param_grid, cv=5, scoring='accuracy')
grid_search.fit(X_train, y_train)

# Best parameters
```

```
best_k = grid_search.best_params_['n_neighbors']
best_weights = grid_search.best_params_['weights']
print(f"Best k: {best_k}, Best weights: {best_weights}")
```

```
Best k: 18, Best weights: uniform
```

```
# Train the optimized model
knn_optimized = KNeighborsClassifier(n_neighbors=best_k, weights=best_weights)
knn_optimized.fit(X_train, y_train)

# Predictions with optimized model
y_pred_optimized = knn_optimized.predict(X_test)


# Performance Evaluation of optimized model
accuracy_optimized = accuracy_score(y_test, y_pred_optimized)
print(f"Optimized Model Accuracy: {accuracy_optimized:.4f}")
print("Classification Report (Optimized Model):\n", classification_report(y_test, y_pred_
print("Confusion Matrix (Optimized Model):\n", confusion_matrix(y_test, y_pred_optimized)
```

```
Optimized Model Accuracy: 0.9722
Classification Report (Optimized Model):
               precision    recall  f1-score   support

           0       0.92      1.00      0.96        12
           1       1.00      0.93      0.96        14
           2       1.00      1.00      1.00        10

    accuracy                           0.97        36
   macro avg       0.97      0.98      0.97        36
weighted avg       0.97      0.97      0.97        36

Confusion Matrix (Optimized Model):
 [[12  0  0]
 [ 1 13  0]
 [ 0  0 10]]
```

```
# Cross-validation score
cv_scores = cross_val_score(knn_optimized, X_train, y_train, cv=5)
print(f"Cross-validation mean accuracy: {cv_scores.mean():.4f}")

# Compare performance
print(f"Accuracy Improvement: {accuracy_optimized - accuracy:.4f}")
```

```
Cross-validation mean accuracy: 0.9791
Accuracy Improvement: 0.2222
```

Example - Regression KNN

Unsupported cell type. Double-click to inspect/edit the content.

```
# Load the Diabetes dataset
data = load_diabetes()
X = data.data  # Features
y = data.target  # Target variable

# Split dataset (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)


# Standardize features (Default)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)


# Default KNN Model (k=5, Euclidean)
knn_default = KNeighborsRegressor(n_neighbors=5)
knn_default.fit(X_train_scaled, y_train)
y_pred_default = knn_default.predict(X_test_scaled)


# Evaluate the model
mae = mean_absolute_error(y_test, y_pred_default)
mse = mean_squared_error(y_test, y_pred_default)
rmse = np.sqrt(mse)
r2 = r2_score(y_test, y_pred_default)

print(f"Mean Absolute Error (MAE): {mae:.4f}")
print(f"Mean Squared Error (MSE): {mse:.4f}")
print(f"Root Mean Squared Error (RMSE): {rmse:.4f}")
print(f"R² Score: {r2:.4f}")
```

```
Mean Absolute Error (MAE): 42.7775
Mean Squared Error (MSE): 3047.4499
Root Mean Squared Error (RMSE): 55.2037
R² Score: 0.4248
```

```
# KNN with Manhattan Distance
knn_manhattan = KNeighborsRegressor(n_neighbors=5, metric='manhattan')
knn_manhattan.fit(X_train_scaled, y_train)
y_pred_manhattan = knn_manhattan.predict(X_test_scaled)


mae = mean_absolute_error(y_test, y_pred_manhattan)
mse = mean_squared_error(y_test, y_pred_manhattan)
rmse = np.sqrt(mse)
r2 = r2_score(y_test, y_pred_manhattan)

print(f"Mean Absolute Error (MAE): {mae:.4f}")
print(f"Mean Squared Error (MSE): {mse:.4f}")
print(f"Root Mean Squared Error (RMSE): {rmse:.4f}")
print(f"R² Score: {r2:.4f}")
```

```
Mean Absolute Error (MAE): 43.4427
Mean Squared Error (MSE): 2919.3690
Root Mean Squared Error (RMSE): 54.0312
```

```
R² Score: 0.4490
```

```python
# MinMax Scaling
scaler_minmax = MinMaxScaler()
X_train_minmax = scaler_minmax.fit_transform(X_train)
X_test_minmax = scaler_minmax.transform(X_test)
knn_minmax = KNeighborsRegressor(n_neighbors=5)
knn_minmax.fit(X_train_minmax, y_train)
y_pred_minmax = knn_minmax.predict(X_test_minmax)


mae = mean_absolute_error(y_test, y_pred_minmax)
mse = mean_squared_error(y_test, y_pred_minmax)
rmse = np.sqrt(mse)
r2 = r2_score(y_test, y_pred_minmax)

print(f"Mean Absolute Error (MAE): {mae:.4f}")
print(f"Mean Squared Error (MSE): {mse:.4f}")
print(f"Root Mean Squared Error (RMSE): {rmse:.4f}")
print(f"R² Score: {r2:.4f}")
```

```
Mean Absolute Error (MAE): 41.6315
Mean Squared Error (MSE): 2912.1690
Root Mean Squared Error (RMSE): 53.9645
R² Score: 0.4503
```

```python
models = ["Default KNN", "Manhattan", "MinMax Scaling"]
predictions = [y_pred_default, y_pred_manhattan, y_pred_minmax]

results = []
for model, y_pred in zip(models, predictions):
```