Feature Selection

Fabio Tripodi
Federico Travascio
Flavia Scarfò
Giovanni Benito Tozzi

INTRODUCTION: Feature Selection and Feature Importance Algorithms

Feature selection is the process of choosing only the most useful variables in a dataset to build a model. This means eliminating variables that are:

- useless,
- redundant,
- noisy (i.e. confusing instead of helpful).

Why is this important?

- Better (more accurate) models
- Greater interpretability and understanding of data
- Savings in data collection costs
- Faster and lighter models
- Avoiding problems connected with the so-called **Curse of Dimensionality**.

The Three Main Types of Feature Selection Methods

The Three Main Types of Feature Selection Methods In the literature, feature selection methods are typically divided into three main categories:

Wrappers : In wrapper methods, the classifier is wrapped into the feature selection process itself. The model is trained and tested multiple times using different combinations of features, and the feature selection is driven directly by the model's performance.

Filters : Filter methods use statistical criteria to rank and select features, independently of any specific classifier. Examples of filter algorithms:

- Basic Filters
- Relief Algorithm
- Correlation-Based Feature Selection

Embedded Methods In embedded methods, feature selection happens as part of the model training process itself. This means that the model naturally selects the most relevant features while learning. Classic examples include:

- Decision Trees or Random Forests (which select features through splitting criteria)
- Lasso Regression (which applies L1 regularization and shrinks some coefficients to zero)

What's Next?

In the following steps, we will implement and compare**three types of filter-based feature selection methods:**. For each method, we will:

- Apply the filter to the dataset
- Select the most relevant features
- Analyze and compare the results to understand *which approach provides the most efficient feature selection.*

Dataset

In this project, we work with the `vehicle-2.csv`, which contains a collection of features extracted from the silhouettes of different vehicles observed from multiple angles. The dataset was built using four scale model vehicles (Corgi models):

- A double decker bus,
- A Chevrolet van,
- A Saab 9000 car,
- An Opel Manta 400 car.

The choice of these specific models was made with the idea that the bus, van, and at least one of the two cars would be easier to distinguish, while differentiating between the two car models might be more challenging due to their similar shapes.

1 Basic Filters

We tried also to validate and refine those rankings with a wrapper strategy using k-nearest-neighbors (k-NN), scanning through the top-k feature subsets (for k = 1, ..., p) to identify the optimal subset size via cross-validation and hold-out testing. Our goal is to discover which features carry the most information about the vehicle class via filter methods, specifically:

- the Chi-square statistic (χ^2),
- Mutual-Information (Information Gain)

We tried also to validate and refine those rankings with a *wrapper* strategy using k-nearest-neighbors (k-NN), scanning through the top-k feature subsets ($k = 1, \ldots, p$) to identify the optimal subset size via cross-validation and hold-out testing.

Data Processing, Filter Score Computation

In the first block, we import the required libraries, load the vehicle-s.csv, encode the string labels ("bus", "car", "van") as integers, and split the dataset 50/50 into training and test sets. We then impute any missing values using the feature-wise median and rescale every column into 0,1 so that both the Chi-square filter and our k-NN classifier operate correctly.

With our data cleaned and scaled, we now quantify each feature's standalone importance.

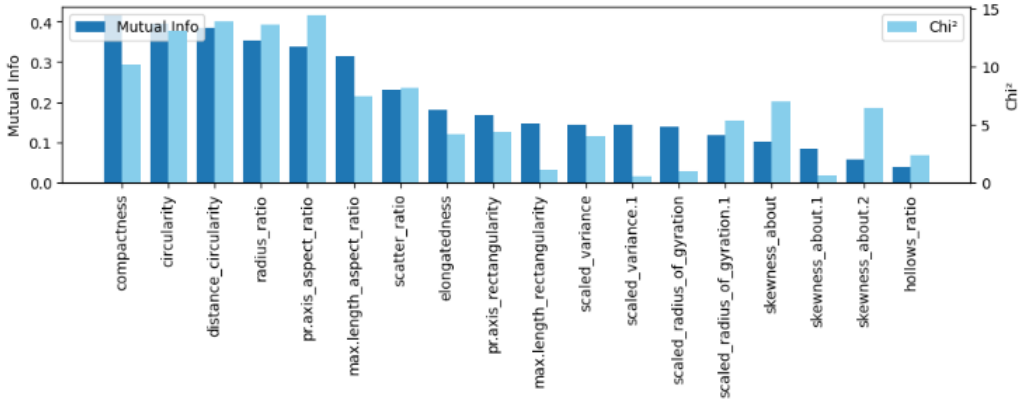
We compute two metrics on only the training set (χ^2 and Mutual Information) and then assemble them into a `DataFrame`, indexed by feature name. Finally we sort that DataFrame by the Mutual-Information column so the most informative features appear at the top.

```
# Importazione librerie
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, MinMaxScaler
from sklearn.impute import SimpleImputer
from sklearn.feature_selection import chi2, mutual_info_classif
import matplotlib.pyplot as plt

# Caricamento dataset
df = pd.read_csv('vehicle-2.csv')
print("Forma iniziale:", df.shape)
```

```
# Codifica delle etichette: 'bus', 'car', 'van'  0, 1, 2
le = LabelEncoder()
y = le.fit_transform(df.pop('class').values)

# Divisione in train/test (50% ciascuno)
X_tr_raw, X_ts_raw, y_train, y_test = train_test_split(
    df.values, y, test_size=0.5, random_state=1
)

# Imputazione valori mancanti (mediana)
imputer = SimpleImputer(strategy='median')
X_tr_imp = imputer.fit_transform(X_tr_raw)
X_ts_imp = imputer.transform(X_ts_raw)

# Scaling delle variabili in [0, 1]
scaler = MinMaxScaler()
X_train = scaler.fit_transform(X_tr_imp)
X_test  = scaler.transform(X_ts_imp)

print("Dopo preprocessing ", "X_train:", X_train.shape, "X_test:",
    X_test.shape)

# Calcolo punteggi Chi-quadro e Mutual Information
chi2_scores, _ = chi2(X_train, y_train)
i_scores = mutual_info_classif(X_train, y_train)

# Costruzione del DataFrame con i punteggi
feature_names = df.columns
df_scores = pd.DataFrame({
    'Mutual Info': i_scores,
    'Chi': chi2_scores
}, index=feature_names).sort_values('Mutual Info', ascending=False)

print("Punteggi ordinati per Mutual Info:")
print(df_scores)
```

Listing 1: Preprocessing and calculation of filter scores

	Mutual Info	Chi²
scaled_variance	0.416533	10.169726
scatter_ratio	0.396591	13.091276
scaled_variance.1	0.385587	13.876782
elongatedness	0.354504	13.642922
pr.axis_rectangularity	0.338914	14.424721
distance_circularity	0.314452	7.451420
radius_ratio	0.229298	8.121118
circularity	0.180025	4.182898
scaled_radius_of_gyration	0.168508	4.329115
max.length_rectangularity	0.144988	1.097650
compactness	0.143785	3.974582
pr.axis_aspect_ratio	0.142594	0.554717
max.length_aspect_ratio	0.139196	0.992421
scaled_radius_of_gyration.1	0.118428	5.381727
hollows_ratio	0.102694	6.938855
skewness_about.2	0.082423	0.639119
skewness_about.1	0.057660	6.450168
skewness_about	0.039478	2.343690



We can see from the *DataFrame* that the top four features **Mutual-Info** are:

- `scaled_variance`
- `scatter_ratio`
- `scaled_variance.1`
- `elongatedness`

These have mutual-info between 0.36 and 0.40, and chi-square between 13 and 14. Only three of them are at the top for both metrics: `scatter_ratio`, `scaled_variance.1` and `elongatedness`.

`scaled_variance` ranks highest by Mutual-Info but only mid-table by chi square. `pr.axis_rectangularity` ranks top by chi square but is only 5th by Mutual Info.

This tells us that the three common top features are the strongest, agreement-backed predictors. A natural cutoff appears around 6-8 features, we can safely drop the lowest-ranked 5-6 features with minimal loss in classification accuracy.

Assessing Agreement Between Filters

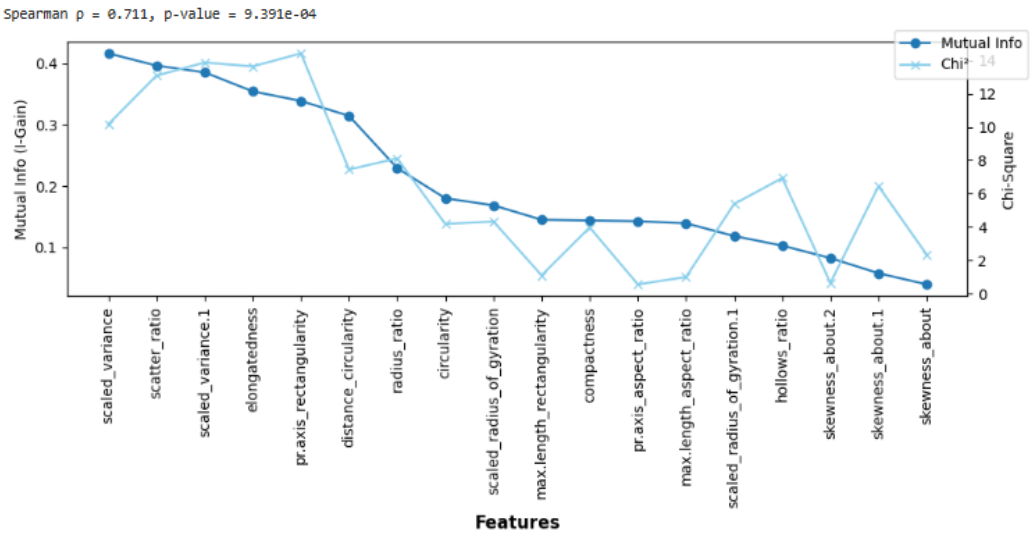
In this section, we measure and visualize how closely the two filter methods (Mutual Information and Chi-Square) agree on feature rankings.

First, we compute the **Spearman**, rank-correlation coefficient (ρ) to quantify the monotonic relationship between the two score lists. Then, we plot both score curves on a shared x-axis of features (with dual y-axes), letting us see at a glance where they coincide and where they diverge in ranking. This gives both a numeric and visual confirmation of the filters' consistency.

```
import matplotlib.pyplot as plt
from scipy.stats import spearmanr

# 1) Calcolo della correlazione di Spearman
rho, pval = spearmanr(df_scores['Mutual Info'], df_scores['Chi'])
print(f"Spearman  = {rho:.3f}, p-value = {pval:.3e}")

# 2) Impostazione grafico con due assi y
fig, ax = plt.subplots(figsize=(10,5))
ax2 = ax.twinx()  # secondo asse y per Chi

# 3) Posizioni sull'asse x
rr = range(len(feature_names))

# 4) Tracciamento linea Mutual Info
ax.plot(rr, df_scores['Mutual Info'], marker='o', label='Mutual
    Info')

# 5) Tracciamento linea Chi
ax2.plot(rr, df_scores['Chi'], color='skyblue', marker='x', label='
    Chi')

# 6) Etichette e formattazione
ax.set_xticks(rr)
ax.set_xticklabels(df_scores.index, rotation=90)
```

```
ax.set_xlabel('variabili', fontsize=12, fontweight='bold')
ax.set_ylabel('Mutual Info (Information Gain)')
ax2.set_ylabel('Chi-quadro')

# 7) Legenda combinata
lines, labels = ax.get_legend_handles_labels()
lines2, labels2 = ax2.get_legend_handles_labels()
fig.legend(lines + lines2, labels + labels2, loc='upper right')

plt.tight_layout()
plt.show()
```

Listing 2: Correlazione Spearman e grafico comparativo



We compute the Spearman rank correlation (rho = 0.711, p = 0.00094) between the Mutual-Info and chi-square feature rankings. A rho of 0.71 (with a tiny p-value) tells us the two methods mostly agree on which features are most (and least) informative, even if their absolute scales differ. The plot highlights:

- Blue circles trace Mutual-Info values per feature (left y-axis).
- Sky-blue crosses trace chi-square values (right y-axis)
- Features are listed on the x-axis in descending Mutual-Info order.

Filter-Based Subset Evaluation

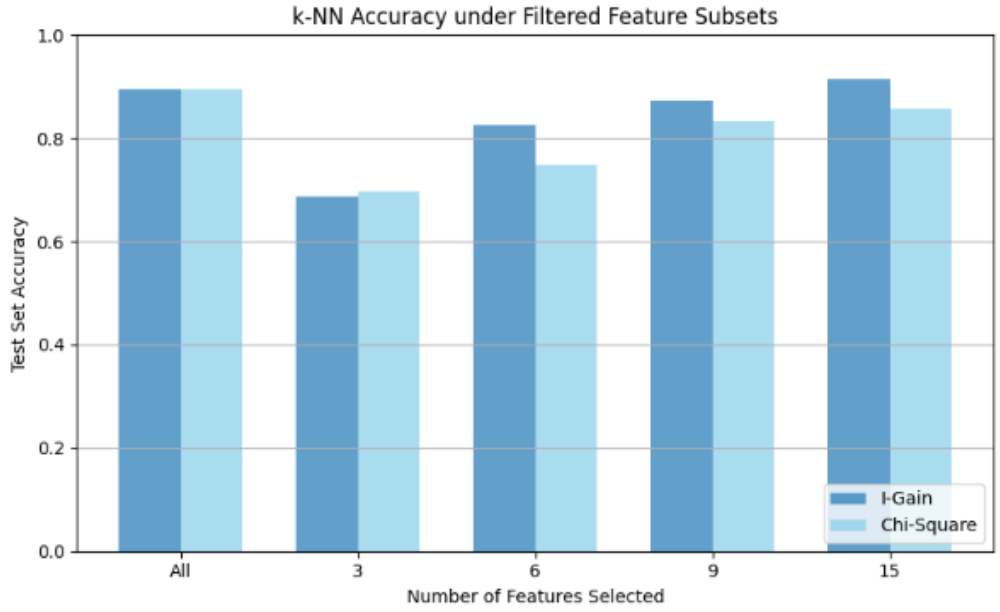
Having ranked each feature by Mutual Information and chi-square, we now test how well smaller subsets actually perform in practice. First we record a baseline k-NN accuracy using all features. Then, for each filter method, we retrain k-NN on the top-k features (for k = all, 3, 6, half, 15) and measure test-set accuracy. The resulting bar chart directly compares how each filter's chosen subsets affect k-NN performance, revealing how many features suffice to match, or even exceed, the full-feature baseline.

```
from sklearn.feature_selection import SelectKBest,
    mutual_info_classif, chi2
from sklearn.neighbors       import KNeighborsClassifier
from sklearn.metrics         import accuracy_score
import numpy as np
import matplotlib.pyplot as plt

# 1) Baseline on all features
# Train k-NN on the full feature set so we have something to
    compare against
model = KNeighborsClassifier(n_neighbors=3)
model.fit(X_train, y_train)
y_pred    = model.predict(X_test)
baseline  = accuracy_score(y_test, y_pred)
print(f"Baseline (all {X_train.shape[1]} features): {baseline:.3f}"
    )

# 2) Decide which "k" sizes to test
n_features = X_train.shape[1]

k_options = [ n_features,    # all
              3,             # top 3
              6,             # top 6
              n_features//2, # top half (~9)
              15             # top 15
            ]

filters   = [ mutual_info_classif, chi2 ] # two scoring methods
filt_scores = {}                          # to store results
# 3) Loop over each filter and each k

for filt in filters:
    accs = []  # will collect each accuracy for this filter
    for k in k_options:
```

```
        # select top-k features on the TRAINING SET
        selector = SelectKBest(filt, k=k).fit(X_train, y_train)
        Xtr_k    = selector.transform(X_train)
        Xts_k    = selector.transform(X_test)

        # re-train k-NN on the reduced training data
        model.fit(Xtr_k, y_train)
        ypred_k = model.predict(Xts_k)
        acc     = accuracy_score(y_test, ypred_k)
        accs.append(acc)


        print(f"{filt.__name__:<22} k={k:2d}   acc={acc:.3f}")

  # Save this filter's accuracies for plotting
    filt_scores[filt.__name__] = accs

# 4) Plot the results as side-by-side bars
options = ['All'] + [str(k) for k in k_options[1:]] #x-axis labels
y_pos   = np.arange(len(options))                   # positions
    0,1,2,...

# Pull out the two accuracy lists
ig = filt_scores['mutual_info_classif']
ch = filt_scores['chi2']

fig, ax = plt.subplots(figsize=(8,5))
width = 0.35

# Bars: Mutual Info shifted left, Chi2 shifted right
p1 = ax.bar(y_pos - width/2, ig, width, label='I-Gain',    alpha
    =0.7)
p2 = ax.bar(y_pos + width/2, ch, width, label='Chi-Square', alpha
    =0.7, color='skyblue')

ax.set_xticks(y_pos)
ax.set_xticklabels(options)
ax.set_ylabel('Test Set Accuracy')
ax.set_xlabel('Number of Features Selected')
ax.set_title('k-NN Accuracy under Filtered Feature Subsets')
ax.legend(loc='lower right')
ax.set_ylim(0.0, 1.0)
plt.grid(axis='y')
```

```
plt.tight_layout()
plt.show()
```

```
Baseline (all 18 features): 0.894
mutual_info_classif    k=18 →  acc=0.894
mutual_info_classif    k= 3 →  acc=0.686
mutual_info_classif    k= 6 →  acc=0.825
mutual_info_classif    k= 9 →  acc=0.872
mutual_info_classif    k=15 →  acc=0.915
chi2                   k=18 →  acc=0.894
chi2                   k= 3 →  acc=0.697
chi2                   k= 6 →  acc=0.749
chi2                   k= 9 →  acc=0.832
chi2                   k=15 →  acc=0.858
```



All 18 features: the k-NN model gets 89.4% accuracy when using every feature.

- k = 3: using only the top 3 features (by each filter) drops accuracy to 68-70% – too few features lead to underfitting.

- k = 6: jumps back up to 82.5% (I-Gain) or 74.9% (Chi-square) – six features recover most of the signal.

- k = 9: we are at87.2% (I-Gain) or 83.2% (Chi-square) – almost as good

as the baseline with half the features.

- k = 15: I-Gain tops out at 91.5%, even above the full-feature baseline. Chi-square stays at 85.8%.

This means that 6-9 features give $> 80\%$ accuracy. The I-Gain filter does a better job of choosing a larger subset, edging out the full-feature baseline; while chi-square is more conservative.

Hybrid Filter-Wrapper Selection

To pinpoint the exact number of features that maximizes our model's performance, we combine filter ranking with a wrapper evaluation. We rank features by Mutual Information, then iterate $k = 1$ through p (the total feature count), each time selecting the top-k features, running 8-fold cross-validation and hold-out testing with k-NN, and recording both scores. By plotting the filter scores alongside the CV and test accuracies, and marking the k with peak CV performance, we identify the optimal feature subset size that balances predictive power and parsimony.

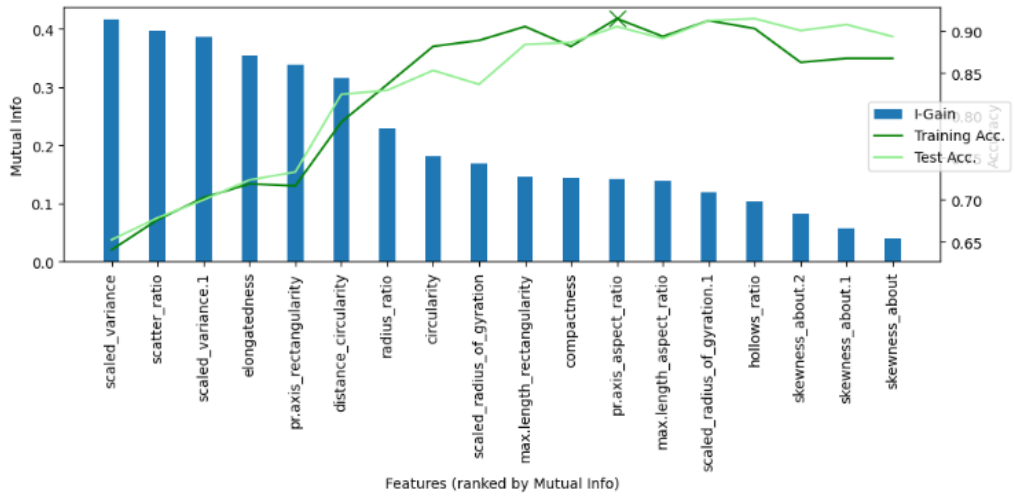
```
from sklearn.feature_selection import SelectKBest,
    mutual_info_classif
from sklearn.model_selection    import cross_val_score
from sklearn.metrics            import accuracy_score
from sklearn.neighbors          import KNeighborsClassifier
import matplotlib.pyplot         as plt
import numpy                    as np

# 1) Prepare: k-NN model & number of features p
model = KNeighborsClassifier(n_neighbors=3)
p     = X_train.shape[1]

cv_acc_scores  = [] # will hold CV accuracies for each k
tst_acc_scores = [] # will hold test accuracies for each k
best_acc = 0.0      # track best CV scores
best_k   = 0        # track k that gave best CV

# 2) For each k = 1  p
for k in range(1, p+1):
    # select top-k features by mutual_info on training set
    selector = SelectKBest(mutual_info_classif, k=k).fit(X_train,
    y_train)
```

```
    Xtr_k    = selector.transform(X_train)
    Xts_k    = selector.transform(X_test)

    # Cross-Validate on the reduced training set
    cv_scores = cross_val_score(model, Xtr_k, y_train, cv=8)
    mean_cv   = cv_scores.mean()
    cv_acc_scores.append(mean_cv)

    # Retrain on Xtr_k, test on Xts_k
    model.fit(Xtr_k, y_train)
    ypred     = model.predict(Xts_k)
    tst_acc   = accuracy_score(y_test, ypred)
    tst_acc_scores.append(tst_acc)

    # Update best-k if this CV is the highest
    if mean_cv > best_acc:
        best_acc = mean_cv
        best_k   = k

    print(f"k={k:2d}  CV acc={mean_cv:.3f}  Test acc={tst_acc:.3f}"
    )

print("\n Best CV at k =", best_k, " CV acc =", best_acc)

# 3) Attach curves back to df_scores
df_scores['Training Acc'] = cv_acc_scores
df_scores['Test Acc']     = tst_acc_scores

# 4) Plot: bars for Mutual Info, lines for the two accuracies
    curves
n = len(df_scores)
rr = np.arange(n)

fig, ax = plt.subplots(figsize=(10,5))
ax2 = ax.twinx()

# bar of filter score
ax.bar(rr, df_scores['Mutual Info'], width=0.35, label='I-Gain')

# accuracy curves
ax2.plot(rr, df_scores['Training Acc'], color='green',      label='
    Training Acc.')
ax2.plot(rr, df_scores['Test Acc'],     color='lightgreen', label='
```

```
    Test Acc.')

# mark the best CV point (at index best_k1)
ax2.plot(best_k-1, best_acc, 'gx', markersize=12)

# ticks & labels
ax.set_xticks(rr)
ax.set_xticklabels(df_scores.index, rotation=90)
ax.set_xlabel('Features (ranked by Mutual Info)')
ax.set_ylabel('Mutual Info')
ax2.set_ylabel('Accuracy')

# unified legend
lines, labs = ax.get_legend_handles_labels()
lines2, labs2 = ax2.get_legend_handles_labels()
fig.legend(lines + lines2, labs + labs2, loc='upper right',
    bbox_to_anchor=(1,0.8))

plt.tight_layout()
plt.show()
```

```
k= 1  CV acc=0.640  Test acc=0.652
k= 2  CV acc=0.676  Test acc=0.678
k= 3  CV acc=0.702  Test acc=0.700
k= 4  CV acc=0.719  Test acc=0.723
k= 5  CV acc=0.716  Test acc=0.733
k= 6  CV acc=0.792  Test acc=0.825
k= 7  CV acc=0.837  Test acc=0.830
k= 8  CV acc=0.882  Test acc=0.853
k= 9  CV acc=0.889  Test acc=0.837
k=10  CV acc=0.905  Test acc=0.884
k=11  CV acc=0.882  Test acc=0.887
k=12  CV acc=0.915  Test acc=0.905
k=13  CV acc=0.894  Test acc=0.891
k=14  CV acc=0.913  Test acc=0.913
k=15  CV acc=0.903  Test acc=0.915
k=16  CV acc=0.863  Test acc=0.901
k=17  CV acc=0.868  Test acc=0.908
k=18  CV acc=0.868  Test acc=0.894
```

```
* Best CV at k = 12 → CV acc = 0.9149129172714079
```



Hybrid Filter-Wrapper strategy results: We used Mutual-Information to rank features, then for each subset size $k = 1 \ldots 18$:

- 8-fold cross-validation on the reduced training set.
- Hold-out test on the reduced test-set.

The printed lines show, for each k:

- Small k (1–4) underfit: both CV and test are in the 0.65–0.72 range.
- Mid-range k (5–11) rapidly climb: CV reaches 0.89 at $k = 9$, test 0.84.
- Peak CV occurs at $k = 12$ with 0.915 mean CV accuracy — and the corresponding test accuracy is 0.905, nearly matching.
- Beyond $k = 12$, CV and test plateau or slightly decline, indicating the added features beyond the top-12 are mostly noise.

The chart (blue bars = Mutual-Info; green lines = CV & test accuracy) confirms that the accuracy climbs steeply as you include the top 6–12 features, it peaks around the 12th feature, then it flattens, suggesting that 12 features is the sweet-spot for k-NN on this dataset.

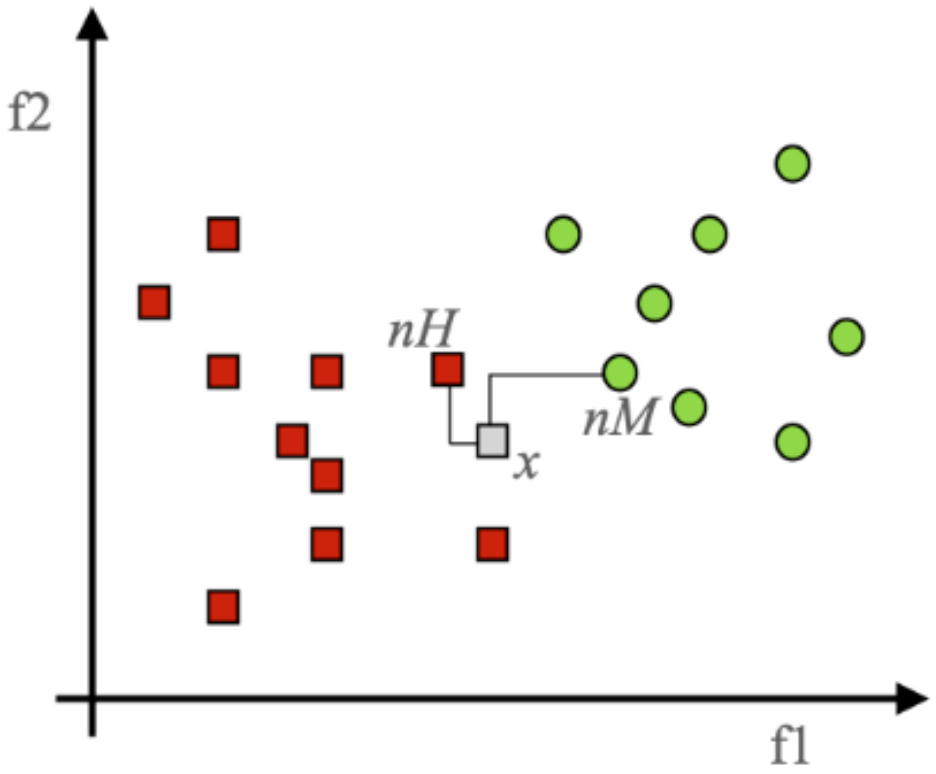
2 Relief Algorithm

The key idea behind Relief is based on the concept of "neighborhood" between data points, an idea borrowed from the k-Nearest Neighbors (k-NN) algorithm.

IMPORTANT: Relief does not use neighbors for classification, but to select the most informative features.

How it works:
For each sample in the dataset, Relief identifies two special neighbors:

- **nearHit**: the closest neighbor that belongs to the same class.
- **nearMiss**: the closest neighbor that belongs to a different class.



The main principle is:
A good feature should have similar values for samples of the same class
And different values for samples of different classes

Update Formula for Feature Weights:

$$w_f \leftarrow w_f - (x_f - nH_f)^2 + (x_f - nM_f)^2$$

Where:

- w_f is the weight of the feature f,
- x_f is the value of feature f for the current sample,
- nH_f is the value of feature f for the NearHit (nearest neighbor from the same class),
- nM_f is the value of feature f for the NearMiss (nearest neighbor from a different class).

As a result:
Relief updates the feature weights by rewarding:

- Features that have similar values for samples of the same class.
- Features where samples from different classes show clearly distinct values.

At the same time, Relief penalizes:

- Features that show clearly distinct values between samples of the same class.
- Features that show similar values between samples of different classes.

```
pip install skrebate
```

```
import pandas as pd
import numpy as np
from skrebate import ReliefF
from sklearn.feature_selection import mutual_info_classif
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
```

Objective

The main purpose of this work is to identify which features are most important for distinguishing between the vehicle types, based on their silhouette properties. To achieve this, we apply the Relief algorithm.

After selecting the most informative features with Relief, we compare the performance of a k-Nearest Neighbors (k-NN) classifier under two different conditions:

- Using all the available features from the dataset.
- Using only the subset of features selected by Relief.

Why k-NN?

The k-Nearest Neighbors (k-NN) algorithm is a simple and widely used classification method. It works by assigning a class to each sample based on the majority class among its k closest neighbors in the feature space. Because k-NN relies on distance calculations between samples, it is particularly sensitive to irrelevant or redundant features. This makes it an excellent choice for testing the impact of feature selection methods like Relief.

```
veic_data = pd.read_csv('vehicle.csv')
print(veic_data.shape)
veic_data.head()
veic_data['class'].value_counts()
```

```
(846, 19)
```

class	count
car	429
bus	218
van	199

dtype: int64

```
veic_data_2 = veic_data.dropna()
print(f"Shape del DataFrame dopo aver eliminato le righe con NaN: {
    veic_data_2.shape}")
```

Shape of DataFrame after deleting rows with NaN: (813, 19)

```
# Load the data, scale it and divide into train and test sets.
# The filters are trained using the training data and then a
    classifier is trained on the feature subset and tested on the
    test set.
y = veic_data_2.pop('class').values
X_raw = veic_data_2.values
X_tr_raw, X_ts_raw, y_train, y_test = train_test_split(X_raw, y,
                                                       random_state
    =42, test_size=0.2)
scaler = MinMaxScaler()
X_train = scaler.fit_transform(X_tr_raw)
X_test = scaler.transform(X_ts_raw)
feature_names = veic_data_2.columns
X_train.shape, X_test.shape
```

((650, 18), (163, 18))

Implementation of the Algorithm

In this step, we apply the Relief algorithm to identify the most informative features from our dataset.

The algorithm is trained only on the training set (X_{train}, y_{train}).

We configure Relief to:

- Select the top 12 most important features (`n_features_to_select=12`),
- Use 30 neighbors (`n_neighbors=30`) for each sample to evaluate the importance of each feature, considering both nearHits (same class) and nearMisses (different classes),
- Run the process in parallel on all available CPU cores (`n_jobs=-1`) to speed up computation.

Feature Selection and Transformation

After fitting the Relief algorithm on the training data, we extract the feature importance scores.

We then apply a transformation to the training set, keeping only the most relevant features selected by Relief.

Why is the transformation step important?
It will allow the classifier to be trained exclusively on the selected features (those identified as the most informative by Relief).

The same transformation will be applied to the test set, ensuring that the classifier is evaluated consistently on the same subset of features.

```
reliefFS = ReliefF(n_features_to_select=12, n_neighbors=30, n_jobs
    = -1)
reliefFS.fit(X_train, y_train)
relief_scores = reliefFS.feature_importances_
reliefFS.transform(X_train).shape
```

(650, 12)

Validating Relief Feature Importance with Information Gain and Spearman Correlation

In this section, we introduce an additional method to evaluate feature relevance: the Mutual Information (Information Gain).

The goal is to compare the feature importance scores obtained by the Relief algorithm with the scores given by Information Gain, to understand whether both methods agree on which features are the most informative for our classification task.

Why calculate Information Gain?

Information Gain:

- Does **NOT** consider how features behave between individual samples in the dataset.
- Does **NOT** consider distances or spatial relationships between data points.
- Focuses only on the direct statistical relationship between each single feature and the class label, without taking into account interactions between features or the distribution of data in the feature space.

Relief:

- Considers the relationships between samples (how close or distant the samples are to each other).

- Uses the idea of NearHit (nearest neighbor from the same class) and NearMiss (nearest neighbor from a different class).
- Takes into account how data points are distributed across the feature space, evaluating how each feature helps to keep same-class samples close and different-class samples far apart.

Why this comparison is useful

The two methods are based on very different principles:

- One focuses on statistical dependency (Information Gain),
- The other on geometric relationships between samples (Relief).

If both methods agree on which features are important, this increases our confidence that those features are truly relevant for the classification task.

```
i_scores = mutual_info_classif(X_train, y_train)
i_scores
# The i-gain scores for the features
```

array([0.20451161, 0.17144285, 0.32109623, 0.20049002, 0.06078303, 0.17183453, 0.3835734 , 0.38936793, 0.31982449, 0.13651662, 0.34875667, 0.37699539, 0.1608693 , 0.09453107, 0.04753025, 0.07029395, 0.1175032 , 0.07636723])

Contextualizing the Result (Our Case: Spearman Statistic $\approx$ 0.841)

The Spearman correlation returned a coefficient of approximately 0.841, which is high and positive.

This suggests that Relief and Information Gain strongly agree on the ranking of feature importance.

In practical terms, this means:

- The features that Relief considers important are also highlighted by Information Gain.
- The selection process is robust and reliable.
- The model built using these features is less likely to suffer from biases caused by relying on a single selection method.

```
from scipy import stats
stats.spearmanr(relief_scores, i_scores)
```

```
SignificanceResult(statistic=np.float64(0.8410732714138286), pvalue
    =np.float64(1.2271367358194706e-05))
```

```
df=pd.DataFrame({'Mutual Info.':i_scores,'ReliefF':relief_scores,'
    Feature':feature_names})
df.set_index('Feature', inplace = True)
df.sort_values('Mutual Info.', inplace = True, ascending = False)
df
```

Feature	Mutual Info.	ReliefF
elongatedness	0.389368	0.086023
scatter_ratio	0.383573	0.079957
scaled_variance.1	0.376995	0.078778
scaled_variance	0.348757	0.063969
distance_circularity	0.321096	0.064800
pr.axis_rectangularity	0.319824	0.077897
compactness	0.204512	0.041239
radius_ratio	0.200490	0.038398
max.length_aspect_ratio	0.171835	0.024709
circularity	0.171443	0.053734
scaled_radius_of_gyration	0.160869	0.036874
max.length_rectangularity	0.136517	0.056896
skewness_about.2	0.117503	0.040050
scaled_radius_of_gyration.1	0.094531	0.023226
hollows_ratio	0.076367	0.059925
skewness_about.1	0.070294	0.023950
pr.axis_aspect_ratio	0.060783	0.019478
skewness_about	0.047530	0.019520

Comparative Visualization: Information Gain vs. Relief

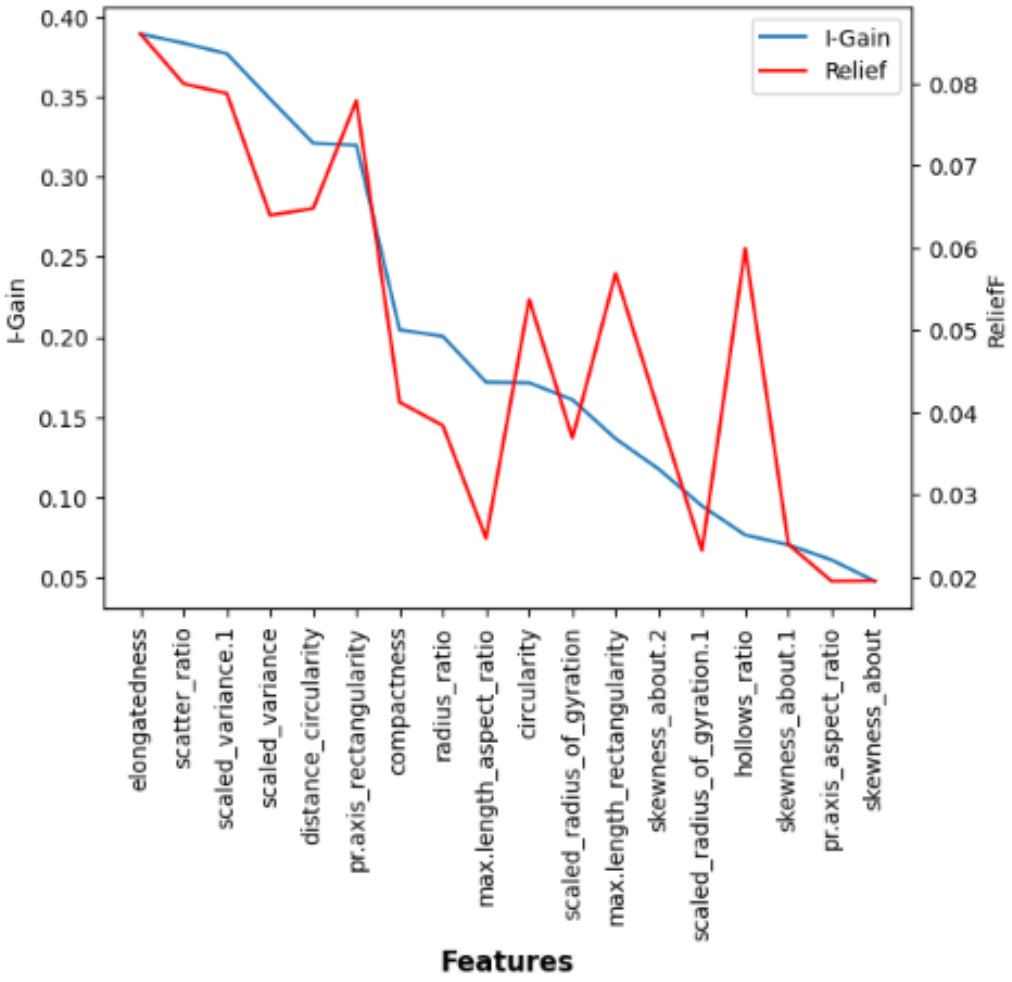
The plot below shows the comparison between the feature importance scores calculated by different methods:

- Information Gain (I-Gain)
- Relief

What we observe from the plot:

- Both methods tend to agree on the top-ranked features (on the left side of the graph), confirming that these features are consistently recognized as relevant by two different approaches.
- The Relief curve shows more variability (ups and downs) compared to Information Gain. This is expected because Relief considers local distances between data points, capturing interactions that Information Gain does not see.
- For the less informative features (right side of the graph), both methods assign lower scores, indicating general agreement on which features are less useful for the classification task.
- Some features in the middle show different scores between the two methods, highlighting how Relief and Information Gain may capture different aspects of the data.

```
fig, ax = plt.subplots()
rr = range(0, len(feature_names))
ax2 = ax.twinx()
ax.plot(df.index, df["Mutual Info."], label='I-Gain')
ax2.plot(df.index, df["ReliefF"], color='red', label='Relief')
ax.set_xticks(rr)

ax.set_xticklabels(list(df.index), rotation=90)
ax.set_xlabel('Features', fontsize=12, fontweight='bold')
ax.set_ylabel('I-Gain')
ax2.set_ylabel('ReliefF')
fig.legend(loc="upper right", bbox_to_anchor=(1,1), bbox_transform=
    ax.transAxes)
```



Evaluating the Impact of Feature Selection on Model Performance

In this section, we test whether selecting the most informative features using the Relief algorithm actually helps improve the performance of our classifier compared to using all the available features.

To do this, we implement and evaluate a k-Nearest Neighbors (k-NN) classifier under two conditions:

- **Model with all features**
  We first train a k-NN classifier using the full set of 18 features from

the dataset (without applying feature selection). This serves as our baseline performance.

- **Model with Relief-selected features**
  We then train the same k-NN classifier but only on the subset of 12 features selected as the most important by the Relief algorithm. The transformation step (`Relief.transform`) reduces the input data to these selected features, both for the training set and the test set.

Why this comparison is important

The k-NN algorithm makes predictions based on distances between samples in the feature space. Using irrelevant or redundant features may distort these distance calculations, potentially harming the model's performance. Feature selection helps to remove noise and focus on the most meaningful dimensions, which can lead to better generalization and improved accuracy.

Baseline Classifier

```
model = KNeighborsClassifier(n_neighbors=6)
model = model.fit(X_train, y_train)
y_pred = model.predict(X_test)
acc_all = accuracy_score(y_pred, y_test)
acc_all
n_features = X_train.shape[1]
n_features
```

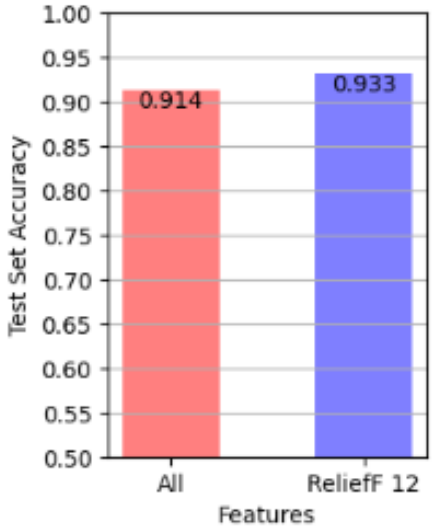
18

After feature selection

```
X_tr_relief = reliefFS.transform(X_train)
X_ts_relief = reliefFS.transform(X_test)
X_tr_relief.shape
kNN_relief = model.fit(X_tr_relief, y_train)
y_pred = kNN_relief.predict(X_ts_relief)
acc_12 = accuracy_score(y_pred, y_test)
acc_12
```

0.9325153374233128

```
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline

fig, ax = plt.subplots(figsize=(2.5,3.5))
width = 0.5
sb = 'skyblue'

options = ['All', 'ReliefF 12']
scores = [acc_all,acc_12]

y_pos = np.arange(len(options))

p1 = ax.bar(y_pos, scores, width, align='center',
            color=['red', 'blue'],alpha=0.5)

ax.set_ylim([0.5, 1])
plt.grid(axis = 'y')
plt.yticks(np.arange(0.5,1.05,0.05))
ax.text(0,acc_all, '%0.3f' % acc_all, ha='center', va = 'top')
ax.text(1,acc_12, '%0.3f' % acc_12, ha='center',va = 'top')

plt.xticks(y_pos, options)
plt.ylabel('Test Set Accuracy')
plt.xlabel('Features')
plt.show()
```



Final Results and Conclusion

The bar chart above shows the test set accuracy of the k-Nearest Neighbors (k-NN) classifier under two different conditions:

- Using all 18 available features → Accuracy: 0.914
- Using only the 12 features selected by Relief → Accuracy: 0.933

What does this result tell us?

The classifier trained on the reduced set of features selected by the Relief algorithm achieves slightly higher accuracy than the one trained on the full set of features. This suggests that removing redundant or less informative features helps the classifier generalize better to new data.

Even though the difference in accuracy is not huge, the improvement is significant because:

- It was achieved by using fewer features, which reduces computational cost.
- It may also reduce the risk of overfitting, as the classifier focuses only on the most relevant information.

3 Correlation-Based Feature Selection (CFS)

Correlation Based feature selection (CFS) is a filter strategy that relies on the principle that *"A good feature subset is one that contains features highly correlated with (predictive of) the class, yet uncorrelated with (not predictive of) each other"*.

The feature-class correlation indicates how representative of the class that feature is while the feature-feature correlation indicates any redundancies between the features.

CFS works by assigning a merit value based on feature-class and feature-feature correlations to each feature subset which becomes the measure by which subsets are evaluated. The merit score for a feature subset S is:

$$MS = \frac{k \cdot \overline{r_{cf}}}{\sqrt{k + k(k-1) \cdot \overline{r_{ff}}}}$$

Where:

- $k = |S|$ is the number of selected features,
- $\overline{r_{cf}}$ is the average correlation between the selected features and the class label,
- $\overline{r_{ff}}$ is the average correlation among the selected features.

The correlations can be measured using techniques such as symmetrical uncertainty based on information gain, feature weighting based on the Gini-index, or by using the minimum description length (MDL) principle.

```
# Import Packages
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
from sklearn.model_selection import cross_val_score
import matplotlib.pyplot as plt
from matplotlib.ticker import MaxNLocator
from CFS import cfs, merit_calculation
from CFS_ForwardSearch import CFS_FS
```

In this block, we loaded and cleaned the dataset by removing missing values. We then split the data into a training set (80%) and a test set (20%),

and applied Min-Max normalization to scale all features to the same range, ensuring comparability.

```
# Load the dataset using the correct path from the upload
df = pd.read_csv('vehicle-2.csv/vehicle-2.csv')

# Drop rows that contain missing values
df_cleaned = df.dropna()

# Copy the cleaned dataset
vehicle_data = df_cleaned.copy()

# Separate target column (class) from features
y = vehicle_data.pop('class').values
X_raw = vehicle_data.values

# Split dataset into training and testing sets (50% each)
X_tr_raw, X_ts_raw, y_train, y_test = train_test_split(X_raw, y,
                                                       random_state
    =2, test_size=0.2)

# Apply Min-Max normalization (scaling features between 0 and 1)
scaler = MinMaxScaler()
X_train = scaler.fit_transform(X_tr_raw)
X_test = scaler.transform(X_ts_raw)

# Store dataset shape information
max_length = X_train.shape[0]
feat_num = X_train.shape[1]

# Print the shape of training and test sets
X_train.shape, X_test.shape
```

((650, 18), (163, 18))

k-Nearest Neighbors Classification and Performance Evaluation

In this block, we're using one of the most intuitive yet powerful machine learning models: k-Nearest Neighbors (k-NN). The k-NN algorithm classifies a new sample based on the k closest samples from the training set. In our case, $k = 5$, so the model looks at the 5 nearest data points and assigns the

class by majority vote. Since k-NN relies on distance calculations, features with larger numeric ranges will dominate the distance metric. That's why normalizing the data beforehand (e.g., with Min-Max scaling) is crucial — it ensures all features contribute equally.

Evaluation metrics

- **Cross-validation:** Instead of a single train-test split, 8-fold cross-validation splits the training data into 8 parts. Each time, 7 folds are used for training, and 1 is used for validation. This process repeats 8 times, and the results are averaged. The cross-validation accuracy was 0.903 providing a reliable and stable estimate of the model's general performance, reducing the influence of randomness from a single data split.

- **Hold-out accuracy:** This is the percentage of correct predictions on the test set — data that the model has never seen. It indicates how well the model generalizes. The hold-out accuracy on the test set was 0.920, slightly higher than the cross-validation result. This suggests that the model generalizes very well to unseen data, and there is no clear sign of overfitting.

```
# Initialize k-Nearest Neighbors classifier with k=5
kNN = KNeighborsClassifier(n_neighbors=5)

# Train the classifier using the training data
kNN = kNN.fit(X_train, y_train)

# Predict on the test data
y_pred = kNN.predict(X_test)

# Calculate hold-out test accuracy
acc = accuracy_score(y_pred, y_test)

# Perform cross-validation on the training data (8-fold)
cv_acc = cross_val_score(kNN, X_train, y_train, cv=8)

# Print results
print("Cross-Validation on training set (all features): {0:.3f}".
    format(cv_acc.mean()))
print("Hold-Out accuracy on test set (all features): {0:.3f}".
    format(acc))
```

```
Cross-Validation on training set (all features): 0.903
Hold-Out accuracy on test set (all features): 0.920
```

Here we are transforming:

- Continuous features into discrete integers using `KBinsDiscretizer`
- String labels into integer labels using `LabelEncoder`

This is important because all information-theoretic metrics (like entropy, information gain, and symmetrical uncertainty) work with discrete variables, and internally they rely on functions (e.g. `np.bincount`) that require integer arrays.

```
from sklearn.preprocessing import KBinsDiscretizer

kbd = KBinsDiscretizer(n_bins=10, encode='ordinal', strategy='
    quantile')
X_discrete = kbd.fit_transform(X_train).astype(int)

# Discretize y if necessary
if y_train.dtype.kind == 'f':
    y_discrete = pd.qcut(y_train, q=10, labels=False, duplicates='
    drop').astype(int)
else:
    y_discrete = y_train

from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()
y_discrete = le.fit_transform(y_train)
```

Forward Search

CFS can work with any search strategy in a similar way to wrapper strategies, but rather than evaluating based on accuracy, the evaluation is based on the merit score.

When using Sequential Forward Selection (SFS), the process begins with no features selected. All single-feature combinations are evaluated according to their merit score, and the best is chosen. Then, all possible two-feature combinations that include this first selected feature are evaluated, and again the best is kept. This procedure continues iteratively, adding the next best feature at each step, until no further improvement in the merit score is observed.

Rather than focusing on maximizing classification accuracy (as a wrapper method would), this approach aims to optimize a merit score that balances two criteria:

- High correlation between the features and the target class, meaning the selected features are strongly related to what we're trying to predict.
- Low inter-feature correlation, meaning the selected features provide complementary (non-redundant) information.

In our project, the correlations between features and between features and the target variable were evaluated using Symmetrical Uncertainty (SU).

Symmetrical Uncertainty is an entropy-based metric that captures both linear and nonlinear dependencies between variables, providing a symmetric score between 0 and 1. It is derived from Information Gain (IG) and entropy calculations.

The formulas used are the following:

Information Gain

$$IG(f_1, f_2) = H(f_1) - H(f_1|f_2)$$

where:

- $H(f_1)$ is the entropy of feature f_1,
- $H(f_1|f_2)$ is the conditional entropy of f_1 given f_2.

Symmetrical Uncertainty

$$SU(f_1, f_2) = \frac{2 \times IG(f_1, f_2)}{H(f_1) + H(f_2)}$$

```
merit_score_sel, sel_comb = CFS_FS(X_discrete, y_discrete)
print("Merit Score of Selected Features: " + str(merit_score_sel.
    values[0]))
feature_names_sel = vehicle_data.columns[np.array(sel_comb)]
print("Selected features:", feature_names_sel.tolist())
```

```
Merit Score of Selected Features: [0.19341998 0.20876741 0.22288486
     0.23038749 0.23250727 0.23297074
 0.23457039 0.23482922]
Selected features: ['scatter_ratio', 'elongatedness', 'max.
    length_aspect_ratio', 'scaled_variance',
                   'distance_circularity', '
    scaled_radius_of_gyration.1',
                   'pr.axis_rectangularity', 'skewness_about.1']
```

Evaluate on Test Data

In this block, we evaluated the k-NN classifier using only the features selected through the Correlation-based Feature Selection (CFS) strategy with Forward Search. The idea is to test whether a reduced, optimized feature subset can achieve high predictive performance while reducing redundancy and dimensionality.

These results are very encouraging. Even though we're working with fewer features than the original full set, the model shows excellent generalization capacity.

```
# Apply the selected feature subset to X_train and X_test
X_train_CFS_FS = X_train[:, sel_comb]
X_test_CFS_FS = X_test[:, sel_comb]

# Train a new k-NN model using only the selected features
kNN_CFS_FS = kNN.fit(X_train_CFS_FS, y_train)

# Make predictions and evaluate performance
y_pred = kNN_CFS_FS.predict(X_test_CFS_FS)
```

```
# Compute accuracy on the test set and using cross-validation
acc_CFS_FS = accuracy_score(y_pred, y_test)
cv_acc_CFS_FS = cross_val_score(kNN_CFS_FS, X_train_CFS_FS, y_train
    , cv=8)

# Print the results
print("Cross-Validation on training set (selected features): {0:.3f
    }".format(cv_acc_CFS_FS.mean()))
print("Hold-Out accuracy on test set (selected features): {0:.3f}".
    format(acc_CFS_FS))
```

```
Cross-Validation on training set (selected features): 0.832
Hold-Out accuracy on test set (selected features): 0.896
```

Best First Search

The CFS implementation utilises a Best First Search strategy, which continues the search process until five consecutive non-improving feature subsets are found. This means that even when the merit score begins to decrease, the search still explores further combinations in the hope of finding a better global solution, rather than stopping at a local maximum.

In this block, the CFS algorithm is applied to the full discretized dataset to evaluate the relevance of all features jointly.

The subset returned includes a large number of features — nearly all from the original set. This behavior can be explained by the nature of Best First Search: since the algorithm continues exploring feature combinations even after some merit decline, it may retain additional features that contribute marginally to the overall merit score.

```
# Perform feature selection using the CFS algorithm with
    discretized data
Sel_feat = cfs(X_discrete, y_discrete)

# Remove invalid feature indices (marked as -1)
Sel_feat = Sel_feat[Sel_feat != -1]

# Display the selected feature indices
Sel_feat

# Print the names of the features selected
feature_names_sel = vehicle_data.columns[Sel_feat]
feature_names_sel
```

```
Index(['scatter_ratio', 'elongatedness', 'max.length_aspect_ratio',
       'scaled_variance', 'distance_circularity',
       'scaled_radius_of_gyration.1', 'pr.axis_rectangularity',
       'skewness_about.1', 'compactness', 'scaled_variance.1',
       'skewness_about', 'circularity', 'pr.axis_aspect_ratio', '
   radius_ratio',
       'max.length_rectangularity'],
      dtype='object')
```

Merit-based Evaluation of Feature Subsets

After obtaining the complete set of features selected by the CFS algorithm using a Best First Search strategy, we proceed with a more detailed analysis

to understand how the quality of the selected subset evolves as features are progressively added.

This process involves an incremental evaluation: we begin by taking only the top-ranked feature, then evaluate the first two, the first three, and so on. For each subset, we compute its merit score, which reflects the theoretical usefulness of the features according to the CFS criterion.

The merit score alone, however, does not reflect how well the model performs in practice. For this reason, in parallel, we also compute the cross-validation accuracy at each step.

```
merit = []
cv_acc_CFS = []

for i in range(1, len(Sel_feat) + 1):
    # Discretized data subset to calculate merit
    X_train_discrete_subset = X_discrete[:, Sel_feat[0:i]]

    # Subset of normalised continuous data per kNN
    X_train_CFS = X_train[:, Sel_feat[0:i]]

    # Calculate merit (discrete input required)
    merit.append(merit_calculation(X_train_discrete_subset,
   y_discrete))

    # Calculates cross-validation accuracy (normalised input needed
   )
    kNN_CFS = kNN.fit(X_train_CFS, y_train)
    cv_acc_CFS.append(cross_val_score(kNN_CFS, X_train_CFS, y_train
    , cv=8).mean())

# View merit scores
merit
```

```
[np.float64(0.19341998419249903),
 np.float64(0.20876741447427963),
 np.float64(0.22288485941250497),
 np.float64(0.23038749425700425),
 np.float64(0.23250727395112938),
 np.float64(0.23297074014082825),
 np.float64(0.23457039144681752),
 np.float64(0.2348292160633787),
 np.float64(0.23467935501110007),
 np.float64(0.23446515547318916),
```

```
np.float64(0.2345620791381852),
np.float64(0.23426739984638803),
np.float64(0.23395539744034613),
np.float64(0.23265085367301153),
np.float64(0.2305938803320921)]
```

Visualization of Merit Score Across Selected Features

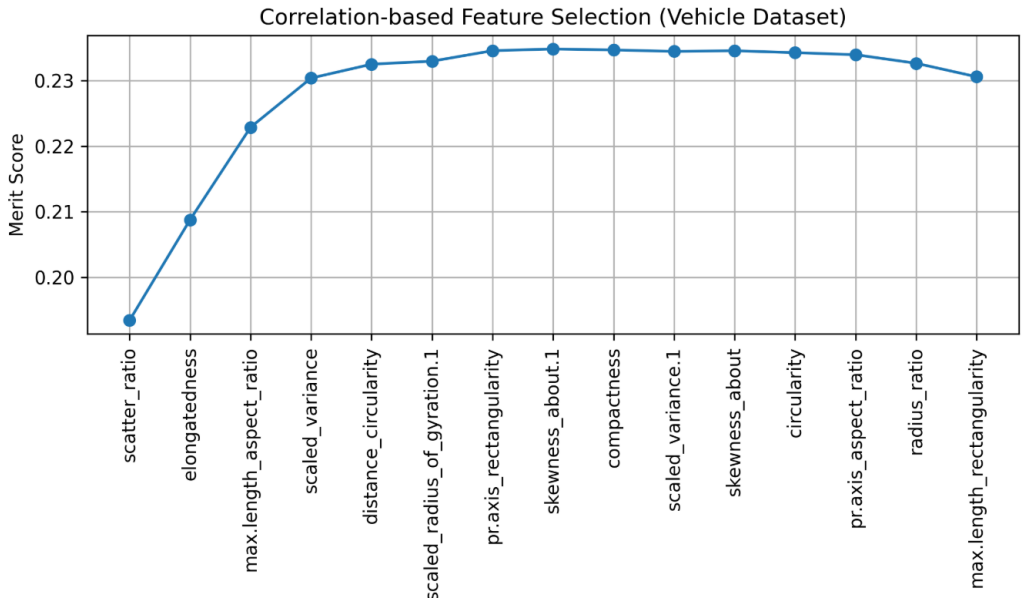
The plotted curve shows how the merit score evolves as features selected by the CFS algorithm are added one at a time, following the order chosen by the Best First Search strategy.

We observe a clear upward trend in the merit score during the initial stages, with a particularly steep rise after the first features. This indicates that the earliest selected features — such as `scatter_ratio`, `elongatedness`, and `max.length_aspect_ratio` — contribute significantly to the quality of the feature subset.

The curve then enters a plateau phase, where the merit score stabilizes around a maximum. Adding further features still maintains a high score, but the gain becomes marginal. Finally, in the last steps, a slight decline is observed, suggesting that some of the last-added features (like `radius_ratio` and `max.length_rectangularity`) might introduce some redundancy or noise, slightly decreasing the overall subset quality.

This pattern helps identify the optimal cutoff point: the moment when adding more features no longer yields meaningful improvement.

```
# Get the feature names for the selected subset
feature_names_sel = vehicle_data.columns[Sel_feat]

# Plot merit score as features are added
f1 = plt.figure(figsize=(8, 5), dpi=300)
plt.plot(feature_names_sel, merit, marker='o')
plt.title("Correlation-based Feature Selection (Vehicle Dataset)")
plt.xticks(rotation=90)
plt.xlabel("Features")
plt.ylabel("Merit Score")
plt.tight_layout()
plt.grid(True)
plt.show()
```




```
# Get the feature names for the selected subset
feature_names_sel = vehicle_data.columns[Sel_feat]

# Plot merit score as features are added
f1 = plt.figure(figsize=(8, 5), dpi=300)
plt.plot(feature_names_sel, merit, marker='o')
plt.title("Correlation-based Feature Selection (Vehicle Dataset)")
plt.xticks(rotation=90)
plt.xlabel("Features")
plt.ylabel("Merit Score")
plt.tight_layout()
plt.grid(True)
plt.show()
```

```
# Select the subset of features for training and testing sets
X_train_CFS = X_train[:, Sel_feat]
X_test_CFS = X_test[:, Sel_feat]

# Train the k-NN classifier on the selected features
kNN_CFS = kNN.fit(X_train_CFS, y_train)

# Make predictions on the test set
y_pred = kNN_CFS.predict(X_test_CFS)
```

```
# Evaluate accuracy on test and with cross-validation
acc_CFS = accuracy_score(y_pred, y_test)
cv_acc_CFS = cross_val_score(kNN_CFS, X_train_CFS, y_train, cv=8)

# Print evaluation results
print("Cross-Validation on training set (selected features): {0:.3f
    }".format(cv_acc_CFS.mean()))
print("Hold-Out accuracy on test set (selected features): {0:.3f}".
    format(acc_CFS))
```

```
Cross-Validation on training set (selected features): 0.874
Hold-Out accuracy on test set (selected features): 0.890
```

Final Comparison of Feature Selection Strategies

This chart compares the performance of three feature selection strategies applied to the vehicle dataset:

- All features
- CFS (Highest Merit) — using Best First Search
- CFS Forward Search — more restrictive, merit-driven selection

The chart highlights several key observations:

- Using all features gives the best test accuracy, but at the cost of higher dimensionality.
- CFS (Highest Merit) performs nearly as well with fewer features, striking a balance between accuracy and dimensionality.
- CFS Forward Search selects even fewer features, with a slight drop in accuracy — ideal when simplicity or model interpretability is a priority.

```
fig, ax = plt.subplots(figsize=(8, 4), dpi=300)
width = 0.2

# Label delle opzioni di feature selection
options = ['All Features', 'CFS (Highest Merit)', 'CFS Forward
    Search']

# Numero di feature per ciascun approccio
```

```
n_feat = [X_train.shape[1], X_train_CFS.shape[1], X_train_CFS_FS.
    shape[1]]

# Accuracy sui dati di test
accs = [acc, acc_CFS, acc_CFS_FS]

# Accuracy in cross-validation (training set)
xv = [cv_acc.mean(), cv_acc_CFS.mean(), cv_acc_CFS_FS.mean()]

# Posizioni sull'asse x
y_pos = np.arange(len(options))

# Barre per la cross-validation (train)
p1 = ax.bar(y_pos - width/2, xv, width, label='Train (Cross-Val)',
    color='blue', alpha=0.7)

# Barre per il test set (hold-out)
p2 = ax.bar(y_pos + width/2, accs, width, label='Test (Hold-Out)',
    color='green', alpha=0.7)

# Imposta limiti y per le accuracy
ax.set_ylim([0.7, 1])

# Secondo asse y per il numero di feature
ax2 = ax.twinx()
p3 = ax2.plot(y_pos, n_feat, color='red', label='Feature Count',
    marker='x', ms=10, linewidth=0)
ax2.set_ylim([0, max(n_feat) + 5])

# Aggiungi griglia
ax.grid(axis='y')

# Unisci legende dei due assi
h1, l1 = ax.get_legend_handles_labels()
h2, l2 = ax2.get_legend_handles_labels()
ax2.legend(h1 + h2, l1 + l2, loc='upper right')

# Mostra solo numeri interi sul secondo asse y
ax2.yaxis.set_major_locator(MaxNLocator(integer=True))

# Etichette x
plt.xticks(y_pos, options)
```

```
# Etichette assi
ax.set_ylabel('Accuracy')
ax2.set_ylabel('Feature Count')

# Titolo
plt.title("Vehicle Dataset - Feature Selection Comparison")

# Mostra il grafico
plt.show()
```