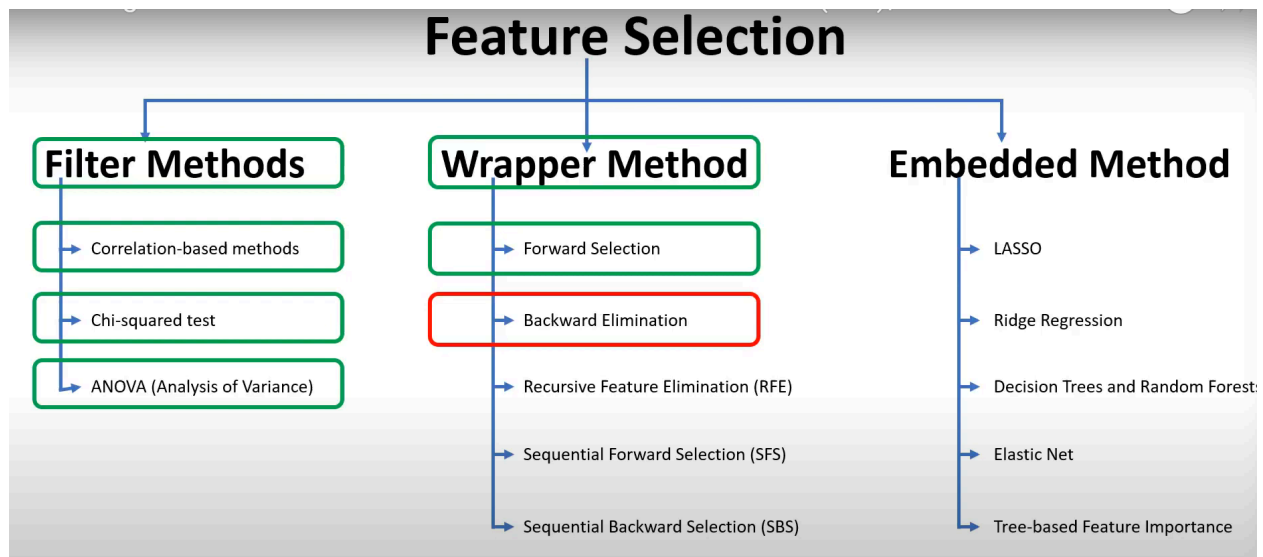# Feature Selection (Filter Methods) (VVIMP-Interview)

- **Feature selection** is the process of choosing a subset of relevant features (variables) for use in a model.

- This is important because not all features contribute to the model's performance, and irrelevant or redundant features can lead to overfitting, increased complexity, and slower computation.

- If your data has 10-15 columns, there's not much need do the feature selection.

Why Feature Selection?

- **Improve Model Performance**: Focuses the model on meaningful patterns.

- **Reduce Overfitting**: Minimizes noise from irrelevant features.

- **Speed Up Training**: Fewer features mean faster computations.

- **Enhance Interpretability**: Simplifies understanding of model decisions.

- **Curse of dimensionality:** As the dimensions increase, the distance between points also increase and therefore it is not reliable if there are too many dimensions.

Types of Feature Selection Methods

1. **Filter Methods**

2. **Wrapper Methods**

3. **Embedded Methods**

4. Hybrid

▼

a. Filter Methods

- **Approach**: Use statistical measures to score feature relevance.

- **Pros**: Fast and model-agnostic.

- **Cons**: Ignores feature interactions.

- **Techniques**:

    - **Correlation**: Pearson's, Spearman's.

    - **Variance Thresholding**: Remove low-variance features.

    - **Chi-Squared Test**: For categorical data.

    - **Mutual Information**: Measures dependency.

b. Wrapper Methods

- **Approach**: Evaluate feature subsets based on model performance.

- **Pros**: Considers feature interactions.

- **Cons**: Computationally expensive.

- **Techniques**:

    - **Forward Selection**: Add features one by one.

- **Backward Elimination**: Remove features one by one.

- **Recursive Feature Elimination (RFE)**: Iteratively removes the least important features.

c. Embedded Methods

- **Approach**: Feature selection during model training.

- **Pros**: Balances efficiency and accuracy.

- **Techniques**:

  - **Lasso (L1 Regularization)**: Shrinks less important coefficients to zero.

  - **Tree-Based Models**: Use feature importance (e.g., Random Forest, XGBoost).

1. Filter Methods

- Individually studies each feature with statistical techniques like correlation or variance and decides if to keep it or not.

- Focuses on 1 feature at a time

- **Pros**: Fast and model-agnostic.

- **Cons**: Ignores feature interactions.

Common techniques:

- **Correlation Matrix**: Select features that are **not highly correlated** with each other.

- **Chi-Square Test**: Tests the independence between features and the target variable (used for categorical data).

- **Variance Threshold**: Removes features with low variance, assuming low variance means little to no information.
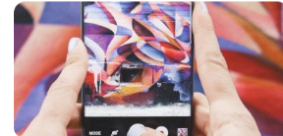
- **ANOVA**

- **Mutual Info**

Dataset Used:

- https://www.kaggle.com/datasets/uciml/human-activity-recognition-with-smartphones

Human Activity Recognition with Smartphones

Recordings of 30 study participants performing activities of daily living

Data Card   Code (426)   Discussion (14)   Suggestions (0)

Columns → 563

<u>AIM</u> : Reduce 563 → 100 without losing the results

No linear regression → Because output is classification (we'll use logistic regression)

```
df = pd.read_csv(r'Human_Activity\train.csv').drop(columns='subject')
df.head()
```

	tBodyAcc-mean()-X	tBodyAcc-mean()-Y	tBodyAcc-mean()-Z	tBodyAcc-std()-X	tBodyAcc-std()-Y	tBodyAcc-std()-Z	tBodyAcc-mad()-X	tBodyAcc-mad()-Y	tBodyAcc-mad()-Z	tBodyAcc-max()-X	...	fBodyBodyGyroJerkMag-skewness()
0	0.288585	-0.020294	-0.132905	-0.995279	-0.983111	-0.913526	-0.995112	-0.983185	-0.923527	-0.934724	...	-0.298676
1	0.278419	-0.016411	-0.123520	-0.998245	-0.975300	-0.960322	-0.998807	-0.974914	-0.957686	-0.943068	...	-0.595051
2	0.279653	-0.019467	-0.113462	-0.995380	-0.967187	-0.978944	-0.996520	-0.963668	-0.977469	-0.938692	...	-0.390748
3	0.279174	-0.026201	-0.123283	-0.996091	-0.983403	-0.990675	-0.997099	-0.982750	-0.989302	-0.938692	...	-0.117290
4	0.276629	-0.016570	-0.115362	-0.998139	-0.980817	-0.990482	-0.998321	-0.979672	-0.990441	-0.942469	...	-0.351471

5 rows × 562 columns

- We dropped 'Subject' as we're dealing with categorical variables
- All these values are **scaled (-1 to +1)**
- **Output → Activity**

```
df['Activity'].value_counts()
```

```
Activity
LAYING                  1407
STANDING                1374
SITTING                 1286
WALKING                 1226
WALKING_UPSTAIRS        1073
WALKING_DOWNSTAIRS       986
Name: count, dtype: int64
```

```
print(df.shape)

Output: (7352, 562)
```

First, apply logistic regression

```
from sklearn.preprocessing import LabelEncoder #CAT → Numeric
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# Separate features and target
X = df.drop('Activity', axis=1)
y = df['Activity']

# Encode target labels
le = LabelEncoder()
y = le.fit_transform(y)

# Split data into training and testing sets
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_stat
e=42)
```

LabelEncoder : This is used to **convert categorical labels** (like 'cat', 'dog', 'fish') into numerical values (e.g., 0, 1, 2) so that they can be used in machine learning models that require numerical inputs.

```
print(X_train.shape)
print(X_test.shape)

Output:
(5881, 561)
(1471, 561)
```

Apply Logistic Regression:

```
# Initialize and train logistic regression model
log_reg = LogisticRegression(max_iter=1000)  # Increase max_iter if it doesn't
converge
log_reg.fit(X_train, y_train)

# Make predictions on the test set
y_pred = log_reg.predict(X_test)

# Calculate and print accuracy score
accuracy = accuracy_score(y_test, y_pred)
print("Test accuracy:", accuracy)

Output:
Test accuracy: 0.9809653297076818
```

- **Accuracy score** is used here because the problem is a **classification task**, where the goal is to predict **categories** (e.g., 'yes' or 'no').

- Accuracy measures how many predictions match the true labels.

- On the other hand, **R² (R-squared)** is used for **regression tasks**, where the goal is to predict continuous values.

Delete Duplicate Columns

```python
def get_duplicate_columns(df):

    duplicate_columns = {}
    seen_columns = {}

    for column in df.columns:
        current_column = df[column]

        # Convert column data to bytes
        try:
            current_column_hash = current_column.values.tobytes()
        except AttributeError:
            current_column_hash = current_column.to_string().encode()

        if current_column_hash in seen_columns:
            if seen_columns[current_column_hash] in duplicate_columns:
                duplicate_columns[seen_columns[current_column_hash]].append(column)
            else:
                duplicate_columns[seen_columns[current_column_hash]] = [column]
        else:
            seen_columns[current_column_hash] = column

    return duplicate_columns
```

Simpler Code to delete Duplicates

```
def remove_duplicate_columns(df):
    # Transpose the DataFrame to compare columns as rows
    transposed_df = df.T

    # Find duplicate rows (which are the original columns)
    duplicate_columns = transposed_df[transposed_df.duplicated()].index.tolist
()

    # Drop duplicate columns from the original DataFrame
    df_cleaned = df.drop(columns=duplicate_columns)

    return df_cleaned
```

`.index` : In Pandas, every DataFrame or Series has an `index` property, which stores the row labels. In the case of a transposed DataFrame (`df.T`), the `index` represents the column names from the original DataFrame.

`.tolist()` : This method converts the Pandas Index object (which holds column names) into a standard Python list.

So when we call

`transposed_df[transposed_df.duplicated()].index.tolist()` , it identifies the index labels (which are the original column names) of the rows that are duplicated. These column names are returned as a list.

```
X_train=remove_duplicate_columns(X_train)
```

	tBodyAcc-mean()-X	tBodyAcc-mean()-Y	tBodyAcc-mean()-Z	tBodyAcc-std()-X	tBodyAcc-std()-Y	tBodyAcc-std()-Z	tBodyAcc-mad()-X	tBodyAcc-mad()-Y	tBodyAcc-mad()-Z
57	0.278007	-0.017803	-0.108965	-0.994425	-0.994873	-0.994886	-0.994939	-0.993994	-0.995450
4154	0.237617	-0.000782	-0.114476	-0.326331	0.069663	-0.224321	-0.343326	0.039623	-0.256327
6945	0.290924	-0.050878	-0.073518	-0.026220	-0.032163	0.393109	-0.118256	-0.030279	0.432861
527	0.275268	-0.015050	-0.114204	-0.981092	-0.901124	-0.960423	-0.984417	-0.901405	-0.965788
4196	0.278790	-0.018585	-0.106908	-0.997380	-0.983893	-0.984482	-0.997331	-0.985196	-0.983768
...	...	...	...	...	...	...	...	...	...
5191	0.278897	-0.030306	-0.096043	-0.555352	-0.104055	-0.438064	-0.572530	-0.112149	-0.429688
5226	0.289183	-0.049248	-0.125083	-0.290043	-0.212102	-0.469731	-0.307317	-0.209558	-0.528635
5390	0.293946	-0.018341	-0.119916	-0.627198	-0.216566	-0.424764	-0.648666	-0.253814	-0.417569
860	0.280475	-0.018976	-0.113756	-0.994825	-0.985314	-0.965857	-0.995170	-0.984285	-0.963293
7270	0.263582	0.006928	-0.095320	-0.368655	-0.142631	-0.151250	-0.426026	-0.130656	-0.149079

5881 rows × 540 columns

do the same for X_test

```
X_test= remove_duplicate_columns(X_test)
```

```
remove_duplicate_columns(X_test)
```
✓ 0.1s

	tBodyAcc-mean()-X	tBodyAcc-mean()-Y	tBodyAcc-mean()-Z	tBodyAcc-std()-X	tBodyAcc-std()-Y	tBodyAcc-std()-Z
4525	0.283203	-0.047024	-0.168986	0.384949	0.176898	-0.310332
1446	0.256904	-0.036623	-0.133856	0.201409	-0.154142	0.344183
5995	0.291316	-0.001065	-0.072461	-0.336609	-0.279162	-0.303323
4222	0.276116	-0.010909	-0.102886	-0.992196	-0.982169	-0.981127
6754	0.256382	0.000428	-0.113664	0.075014	0.046502	-0.369482
...	...	...	...	...	...	...
3704	0.288089	-0.024497	-0.095596	-0.947145	-0.835807	-0.895158
705	0.276959	-0.003011	-0.116333	-0.975745	-0.876568	-0.932322
1650	0.284576	-0.020942	-0.106882	-0.994198	-0.988329	-0.990618
2260	0.277980	-0.019820	-0.109165	-0.997517	-0.982207	-0.974343
5907	0.283935	-0.020993	-0.115315	-0.997183	-0.989286	-0.993920

1471 rows × 540 columns

Variance Threshold

`from sklearn.` `feature_selection` `import` `VarianceThreshold`

- Removes features with low variance, assuming low variance means little to no information.

- Here, B is **constant**

- It doesn't play any role in prediction

- We can drop such columns

- If values of 95% columns is 1 and remaining 5% is → It's **Quasi Constant column**

  - Its variance is close to 0

Steps

- *Normalize data*

1. Decide a **threshold (eg. 0.1)**

   - *0.01 to 0.1*

2. Calculate **variance** for every column

3. **Check** which columns' threshold is **less than our decided threshold**

4. Drop em👆

```
from sklearn.feature_selection import VarianceThreshold
sel = VarianceThreshold(threshold=0.05)
sel.fit(X_train)
```

Check which columns' threshold is **more**:

```
sum(sel.get_support())

Output: 349
```

- `sel.get_support()` → returns boolean values: True/False

False = columns' threshold is less than our decided one

To see the column names:

```
columns = X_train.columns[sel.get_support()]
columns
```

```
Index(['tBodyAcc-std()-X', 'tBodyAcc-std()-Y', 'tBodyAcc-std()-Z',
       'tBodyAcc-mad()-X', 'tBodyAcc-mad()-Y', 'tBodyAcc-mad()-Z',
       'tBodyAcc-max()-X', 'tBodyAcc-max()-Y', 'tBodyAcc-max()-Z',
       'tBodyAcc-min()-X',
       ...
       'fBodyBodyGyroJerkMag-meanFreq()', 'fBodyBodyGyroJerkMag-skewness()',
       'fBodyBodyGyroJerkMag-kurtosis()', 'angle(tBodyAccMean,gravity)',
       'angle(tBodyAccJerkMean),gravityMean)',
       'angle(tBodyGyroMean,gravityMean)',
       'angle(tBodyGyroJerkMean,gravityMean)', 'angle(X,gravityMean)',
       'angle(Y,gravityMean)', 'angle(Z,gravityMean)'],
      dtype='object', length=349)
```

Let's insert these columns in X_train & X_test:

```
X_train = sel.transform(X_train)
X_test = sel.transform(X_test)

X_train = pd.DataFrame(X_train, columns=columns)
X_test = pd.DataFrame(X_test, columns=columns)
```

`X_train = sel.transform(X_train)` : This is where the actual feature selection happens for your training data.

- The `sel.transform(X_train)` → It deletes all columns except the above ones

- But it returns a (**NumPy array**) containing only the *selected* features (columns) - those that had a variance greater than 0.05.

Why we used `pd.` `DataFrame` `(X_train, columns=columns)` ?

- **NumPy arrays → Pandas DataFrames**

- After `sel.transform(X_train)` and `sel.transform(X_test)`, `X_train` and `X_test` are no longer Pandas DataFrames. They become NumPy arrays.

- To convert them back into Pandas DataFrames (which are often easier to work with, especially if you want to keep column names), you use

`pd.DataFrame()` .

> 💡 **New sklearn:** `.set_output()` → **Get the output as DataFrame**

`fit` vs `transform` vs `fit_transform`

- `fit` is used on the **training data** to avoid any potential data leakage.
- `transform` is then used on both the training and test data to ensure the same features are selected in both.
- Using `fit_transform` on the test set would improperly use test data during training, which violates the principles of machine learning and can lead to biased results.

```
print(X_train.shape)
print(X_test.shape)

Output:
(5881, 349)
(1471, 349)
```

Points to Consider whiles using variance threshold:

It:

- Ignores Target Variable
- Ignores Feature Interactions
- Sensitive to Data Scaling
- Arbitrary Threshold Value

Correlation

- Pearson

Disadvantages of Correlation:

1. **Linearity Assumption**: Only measures **linear** relationships, not non-linear ones.

2. **Limited to Two Variables**: Doesn't capture **complex relationships** involving more than two variables.

3. **Threshold Determination**: Defining what's considered a **"high"** correlation is **subjective**.

4. **Sensitive to Outliers**: A few extreme values can **skew** the correlation coefficient.

```
X_train.corr()
```

|	tBodyAcc-std()-X	tBodyAcc-std()-Y	tBodyAcc-std()-Z	tBodyAcc-mad()-X	tBodyAcc-mad()-Y	tBodyAcc-mad()-Z	tBodyAcc-max()-X	tBc
tBodyAcc-std()-X	1.000000	0.927247	0.850268	0.998631	0.920936	0.845200	0.981284	0
tBodyAcc-std()-Y	0.927247	1.000000	0.895065	0.922627	0.997384	0.894128	0.917831	0
tBodyAcc-std()-Z	0.850268	0.895065	1.000000	0.842986	0.890973	0.997414	0.852711	0
tBodyAcc-mad()-X	0.998631	0.922627	0.842986	1.000000	0.916201	0.838010	0.973704	0
tBodyAcc-mad()-Y	0.920936	0.997384	0.890973	0.916201	1.000000	0.890707	0.911283	0
...	...	...	...	...	...	...	...	
BodyGyroMean,gravityMean)	0.023914	-0.002241	-0.010535	0.024098	-0.005865	-0.014838	0.029230	-0
GyroJerkMean,gravityMean)	-0.035176	-0.028881	-0.016002	-0.035629	-0.026679	-0.016949	-0.038935	-0
angle(X,gravityMean)	-0.374114	-0.383095	-0.344114	-0.370629	-0.379578	-0.346350	-0.386159	-0
angle(Y,gravityMean)	0.472605	0.524945	0.475241	0.467965	0.526803	0.476498	0.482312	0
angle(Z,gravityMean)	0.393209	0.432180	0.480824	0.389139	0.430548	0.477627	0.404088	0

Here, we can take 2 approaches:

1. Find out corr with y

- Decide a range - eg. -0.3 to +0.3

- If it's below -0.3 or above 0.3, keep the column

2. Find corr **between columns**

- Select a threshold

- If Corr is above absolute value, drop that column

- This is to check **multicollinearity**

- This is mostly used approach

What's our approach?

- Find out corr between columns

- If it's more than 0.95 → drop 1 of the columns

```
import matplotlib.pyplot as plt
plt.figure(figsize=(12,10))
sns.heatmap(X_train.corr())
```
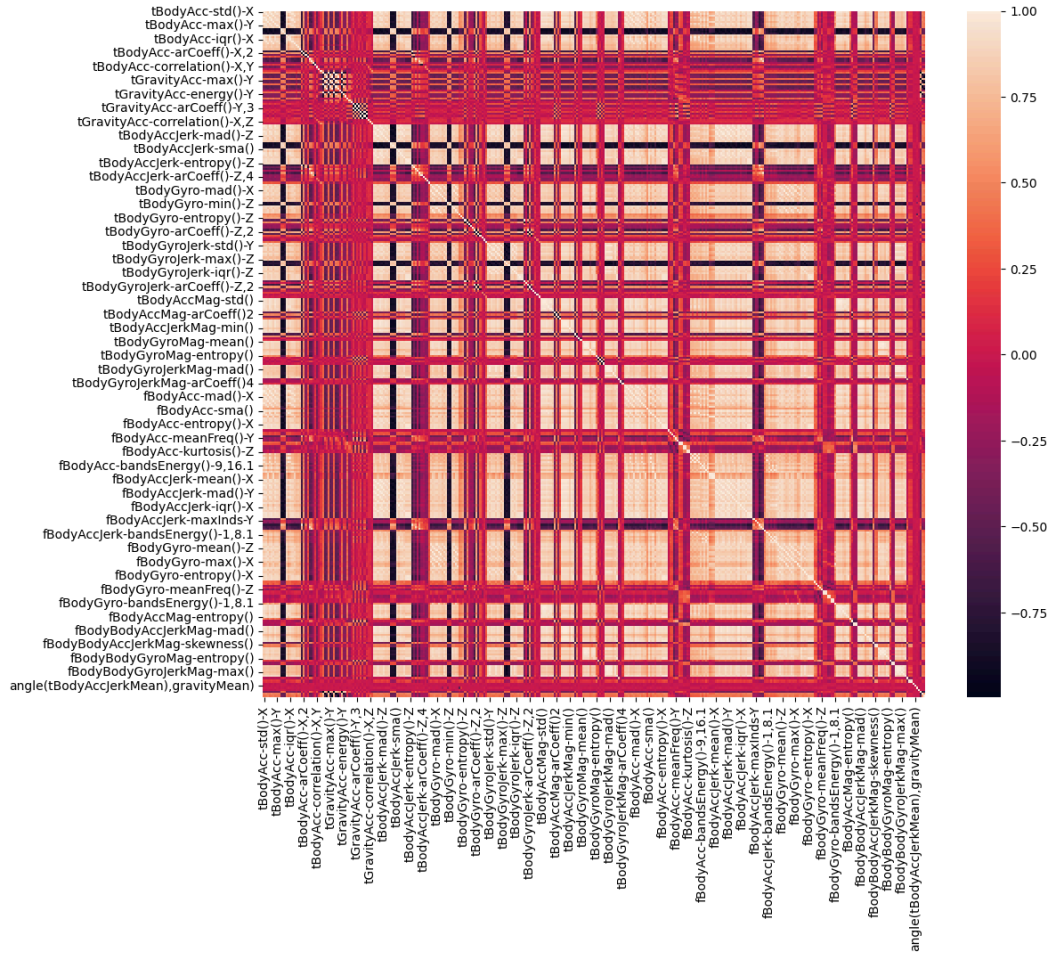
```
corr_matrix = X_train.corr()
```

```
# Get the column names of the DataFrame
columns = corr_matrix.columns

# Create an empty list to keep track of columns to drop
columns_to_drop = []

# Loop over the columns
for i in range(len(columns)):
    for j in range(i + 1, len(columns)):
        # Access the cell of the DataFrame
```

```
        if corr_matrix.loc[columns[i], columns[j]] > 0.95:
            columns_to_drop.append(columns[j])

  print(len(columns_to_drop))
```

`columns = corr_matrix.columns` : This line extracts the column names

`for i in range(len(columns)):`
`for j in range(i + 1, len(columns)):`

Starting
`j` from `i + 1` ensures we only iterate through the *upper triangle* of the correlation matrix (excluding the diagonal and the lower triangle), thus avoiding redundant comparisons and also avoiding comparing a column with itself.

Remember

```
  corr_matrix.loc[columns[0],columns[1]]
```

This will access this value:

	tBodyAcc-std()-X	tBodyAcc-std()-Y	tBodyAcc-std()-Z
tBodyAcc-std()-X	1.000000	0.927247	0.850268
tBodyAcc-std()-Y	0.927247	1.000000	0.895065
tBodyAcc-std()-Z	0.850268	0.895065	1.000000
tBodyAcc-mad()-X	0.998631	0.922627	0.842986
tBodyAcc-mad()-Y	0.920936	0.997384	0.890973

- `columns[0]` which is `'tBodyAcc-std()-X'` is used as the **row label**.
- `columns[1]` which is `'tBodyAcc-std()-Y'` is used as the **column label**.

How It Avoids Duplicate Checking:

- The key part is `for j in range(i + 1, len(columns))` .

  - When `i = 0` , `j` will start from **1**, so it compares column 0 with column 1, column 2, and so on.

  - When `i = 1` , `j` will start from **2**, so it compares column 1 with column 2, column 3, and so on.

  - When `i = 2` , `j` will start from **3**, so it compares column 2 with column 3, column 4, and so on.

> 💡 **It will go through the upper triangle.**

```
if corr_matrix.loc[columns[i], columns[j]] > 0.95:
  columns_to_drop.append(
columns[j]  )
```

- If two features have a correlation **greater than 0.95**, the second feature (`columns[j]`) is marked for removal.

```
columns_to_drop = set(columns_to_drop)
len(columns_to_drop)

Output: 197 #columns to drop
```

- We converted the list in set to avoid duplicates

Drop all these columns:

```
X_train.drop(columns = columns_to_drop, axis = 1, inplace=True)
X_test.drop(columns = columns_to_drop, axis = 1, inplace=True)
```

```
print(X_train.shape)
print(X_test.shape)

Output:
(5881, 152)
(1471, 152)
```

- Now we have 152 columns left.

ANOVA

- ANOVA is used when:

  - Input: Numeric

  - Output: Categorical (Usually 2+)

- Can also be used when output is numeric

How does ANOVA work?

- You take 1 column & study its relationship with output column

- You calculate F-stats/F-ratio

$$F = \frac{Between\ Group\ Variance}{Withing\ Group\ Variance}$$

There is a less difference between groups

ANOVA using sklearn Python:

```
from sklearn.feature_selection import f_classif
from sklearn.feature_selection import SelectKBest


sel = SelectKBest(f_classif, k=100).fit(X_train, y_train)

# display selected feature names
X_train.columns[sel.get_support()]
```

```
Index(['tBodyAcc-std()-X', 'tBodyAcc-std()-Y', 'tBodyAcc-std()-Z',
       'tBodyAcc-max()-Z', 'tBodyAcc-min()-X', 'tBodyAcc-min()-Y',
       'tBodyAcc-min()-Z', 'tBodyAcc-entropy()-X', 'tBodyAcc-entropy()-Y',
       'tBodyAcc-entropy()-Z', 'tBodyAcc-arCoeff()-X,1',
       'tBodyAcc-arCoeff()-X,2', 'tBodyAcc-arCoeff()-X,3',
       'tBodyAcc-arCoeff()-Y,1', 'tBodyAcc-arCoeff()-Z,1',
       'tBodyAcc-correlation()-X,Y', 'tBodyAcc-correlation()-Y,Z',
       'tGravityAcc-mean()-X', 'tGravityAcc-mean()-Y', 'tGravityAcc-mean()-Z
       'tGravityAcc-sma()', 'tGravityAcc-energy()-Y', 'tGravityAcc-energy()-
       'tGravityAcc-entropy()-X', 'tGravityAcc-entropy()-Y',
       'tGravityAcc-arCoeff()-Y,1', 'tGravityAcc-arCoeff()-Y,2',
       'tGravityAcc-arCoeff()-Z,1', 'tGravityAcc-arCoeff()-Z,2',
       'tGravityAcc-correlation()-Y,Z', 'tBodyAccJerk-std()-Z',
       'tBodyAccJerk-min()-X', 'tBodyAccJerk-min()-Y', 'tBodyAccJerk-min()-Z
       'tBodyAccJerk-entropy()-X', 'tBodyAccJerk-arCoeff()-X,3',
```

`f_classif` → f classification

- calculates the **ANOVA F-value** for each feature

`SelectKBest` → Select the best features (in this case, 100 based on a scoring function (here, f_classif).

- Here, **we'll calculate ANOVA for all 152 columns & select best 100**

```
columns = X_train.columns[sel.get_support()]
```


- Now, transform the data

```
X_train = sel.transform(X_train)
X_test = sel.transform(X_test)

X_train = pd.DataFrame(X_train, columns=columns)
X_test = pd.DataFrame(X_test, columns=columns)
```

```
print(X_train.shape)
print(X_test.shape)

Output:
(5881, 100)
(1471, 100)
```

Disadvantages of ANOVA

- Assumption of Normality

- Assumption of Homogeneity of Variance

- Independence of Observations

- Effect of Outliers

- Doesn't Account for Interactions

NOW, APPLY LOGISTIC REGRESSION AGAIN WITH NEW DATA

```
# Initialize and train logistic regression model
log_reg = LogisticRegression(max_iter=1000)  # Increase max_iter if it doesn't
converge
log_reg.fit(X_train, y_train)

# Make predictions on the test set
y_pred = log_reg.predict(X_test)
```

```
# Calculate and print accuracy score
accuracy = accuracy_score(y_test, y_pred)
print("Test accuracy:", accuracy)
```

Output: Test accuracy: 0.9694085656016316

Chi-square Test

Formula

The Chi-Square statistic is calculated as:

$$\chi^2 = \sum \frac{(O_{ij} - E_{ij})^2}{E_{ij}}$$

Where:

- O_{ij}: Observed frequency in cell (i, j).
- E_{ij}: Expected frequency in cell (i, j), calculated as:

$$E_{ij} = \frac{(\text{Row Total}) \times (\text{Column Total})}{\text{Grand Total}}$$

Degrees of Freedom

$$df = (r - 1) \times (c - 1)$$

Where:

- r: Number of rows.
- c: Number of columns.

Chi Square

- Used when **both input & output** columns are **categorical**
- We'll use Titanic dataset

```
titanic = pd.read_csv('titanic/train.csv')[['Pclass','Sex','SibSp','Parch','Embarke
d','Survived']]
titanic.head()
```

	Pclass	Sex	SibSp	Parch	Embarked	Survived
0	3	male	1	0	S	0
1	1	female	1	0	C	1
2	3	female	0	0	S	1
3	1	female	1	0	S	1
4	3	male	0	0	S	0

- Survived ll be output

How Chi Square works

- We'll ask question: Is there any relationship betweent the columns sex & survived?

  ○ If no → we'll drop the sex column

- You form a contingency table like: 👇

Contingency Table			
	Boy	Girl	Sum
like Snickers	43	30	73
doesn't like Snickers	8	19	27
Sum	51	49	100

- We can do the same in python with `pd.crosstab()`

```
ct = pd.crosstab(titanic['Survived'],titanic['Sex'],margins=True)
```

ct

Sex	female	male	All
Survived			
0	81	468	549
1	233	109	342
All	314	577	891

- You can this 👆 observed value

- You make one more table "**Expected**" with ideal data

Suppose you have a 2x2 contingency table with the following observed values:

	Category A	Category B	Total
X	30	10	40
Y	20	40	60
Total	50	50	100

For the cell at row **X** and column **Category A**, the expected value would be:

$$E_{X,A} = \frac{(R_X \times C_A)}{N} = \frac{(40 \times 50)}{100} = 20$$

```
ct2= pd.crosstab(titanic['Survived'],titanic['Sex'])
ct2

from scipy.stats import chi2_contingency
chi2_contingency(ct2)
```

Output:
Chi2ContingencyResult(statistic=260.71702016732104, pvalue=1.1973570627 755645e-58, dof=1, expected_freq=array([[193.47474747, 355.52525253], [120.52525253, 221.47474747]]))

- We used ct2 **without the call column** because the all column interferes with results

- Run a loop and calculate `chi2_contingency` for each column

```
score = []

for feature in titanic.columns[:-1]:

    # create contingency table
    ct = pd.crosstab(titanic['Survived'], titanic[feature])

    # chi_test
    p_value = chi2_contingency(ct)[1]
    score.append(p_value)
```

`titanic.columns[:-1]` → All columns except last one i.e. "Survived"
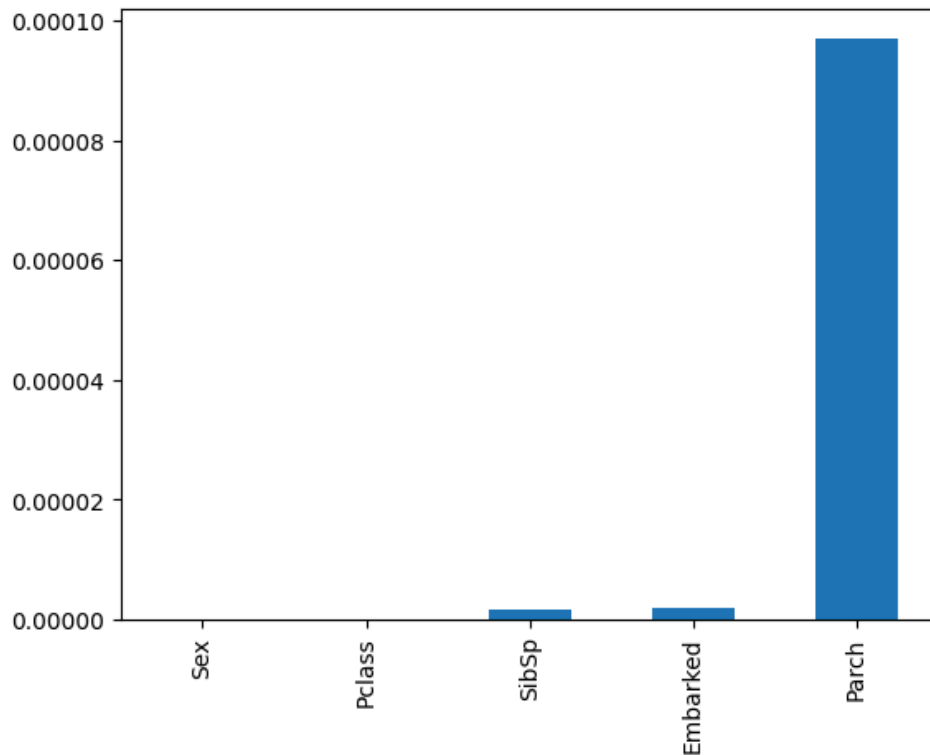
`chi2_contingency(ct)[1]` → p-value


```
score

Output:
[4.549251711298793e-23,
 1.1973570627755645e-58,
 1.55858104659021l8e-06,
 9.703526421039996e-05,
 1.769922284120912e-06]
```

- Plot these on graph

```
pd.Series(score, index=titanic.columns[:-1]).sort_values(ascending=True).plot
(kind='bar')
```


- More the p-value, less the importance of the feature.

Alternate code with `sklearn`

```
from sklearn.preprocessing import LabelEncoder
from sklearn.feature_selection import chi2
import matplotlib.pyplot as plt

# assuming titanic is your DataFrame and 'Survived' is the target column
```

```python
# Encode categorical variables
le = LabelEncoder()
titanic_encoded = titanic.apply(le.fit_transform)

X = titanic_encoded.drop('Survived', axis=1)
y = titanic_encoded['Survived']

# Calculate chi-squared stats
chi_scores = chi2(X, y)

# chi_scores[1] are the p-values of each feature.
p_values = pd.Series(chi_scores[1], index = X.columns)
p_values.sort_values(inplace = True)

# Plotting the p-values
p_values.plot.bar()

plt.title('Chi-square test - P-values')
plt.xlabel('Feature')
plt.ylabel('P-value')

plt.show()
```
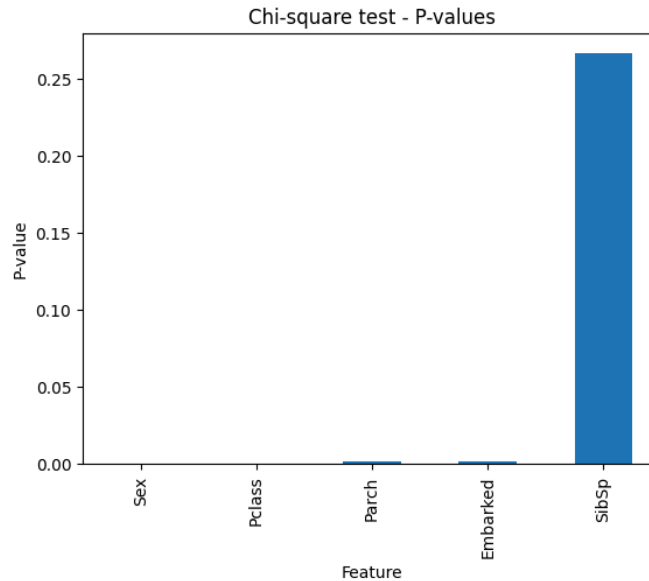
Chi-square test - P-values

```
le = LabelEncoder()
titanic_encoded = titanic.apply(le.fit_transform)
```

- In this method, you have to convert the variables into Category

- This above code works with `SelectKBest`

`.apply(...)` : This is a Pandas DataFrame method that applies a function to each column (or row, depending on the `axis` argument, which defaults to columns if not specified) of the DataFrame.

`chi2(X, y)` returns→ ***Chi2 statistics & p_values***

Advantages and Disadvantages of Filter Methods

Advantages

- Simplicity

- Speed

- Scalability

- Pre-processing Step:

- They can serve as a pre-processing step for other feature selection methods.

- For instance, you could use a filter method to remove irrelevant features before applying a more computationally expensive method, such as a wrapper method.

Disadvantages

- Lack of Feature Interaction

- Model Agnostic:

  - This means that the selected features might not necessarily contribute to the accuracy of the specific model you want to use.

  - Can work well for one model & might not give that accurate results for another model.

- Statistical Measures Limitation:

  - For example, correlation is a measure of linear relationship and might not capture non-linear relationships effectively.

  - Similarly, variance-based methods might keep features with high variance but low predictive power.

- Threshold Determination